Vivado Design Suite User Guide

Designing IP Subsystems Using IP Integrator

UG994 (v2016.3) October 5, 2016

XILINX
ALL PROGRAMMABLE™

Revision History

The following table shows the revision history for this document.

Date	Version	Revision
10/05/2016	2016.3	Documented change of the default to OOC per IP, with the IP Cache enabled, in Generating Output Products in Chapter 4. Discussed IP Cache settings in Out-of-Context per IP in Chapter 4. Added limitations related to Adding Read-Only Block Designs in Chapter 4. Updated Adding ELF and Associating it With an Embedded Processor in Chapter 4. Added System ILA debug core to Chapter 6, Debugging IP Integrator Designs. The ILA debug core is legacy that should not be used for debugging new block designs. Added discussion of Reset signal polarity, X_INTERFACE_PARAMETER and X_INTERFACE_IGNORE attributes in Inferring Control Signals in a RTL Module in Chapter 12. Add **Automotive Applications Disclaimer** to Please Read: Important Legal Notices in Please Read: Important Legal Notices in Appendix A. Minor editorial changes throughout.
06/08/2016	2016.2	Minor editorial update.
04/06/2016	2016.1	Documented Module Reference flow in Chapter 12, Referencing RTL Modules with a brief mention of it in Adding RTL Modules to the Block Design in Chapter 2. Added references to utility IP datasheets under Glue Logic IP in IP Integrator in Chapter 2. Added details on improvements to Hierarchical IP in IP Integrator in Chapter 2. Added a note about Support for Address Width 64-bits and Greater in Chapter 3. Added details on Design Runs specified for Out-of-Context synthesis under Generate Output Products Dialog Box in Chapter 4.

Table of Contents

Appendix A: Additional Resources and Legal Notices

XILINX
ALL PROGRAMMABLE™

Chapter 1

Getting Started with Vivado IP Integrator

Introduction

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 feature 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 will be 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 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, runtime DRCs to ensure, for example, that connections between the IP in an IP Integrator design are compatible and that the IP themselves are properly configured.

Creating a Block Design

Introduction

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

Creating a Project

While entire designs can be created using IP Integrator, the typical design will consist of HDL, IP, and IP Integrator block designs. This section is an introduction to creating a new IP Integrator-based design.

As shown in the figure below, you start by clicking on **Create New Project** in the Vivado® IDE graphical user interface (GUI) to create a new project. The Vivado Design Suite supports many different types of design projects. Refer to this link in the *Vivado Design Suite User Guide: System-Level Design Entry* (UG895) [Ref 3] for more information.

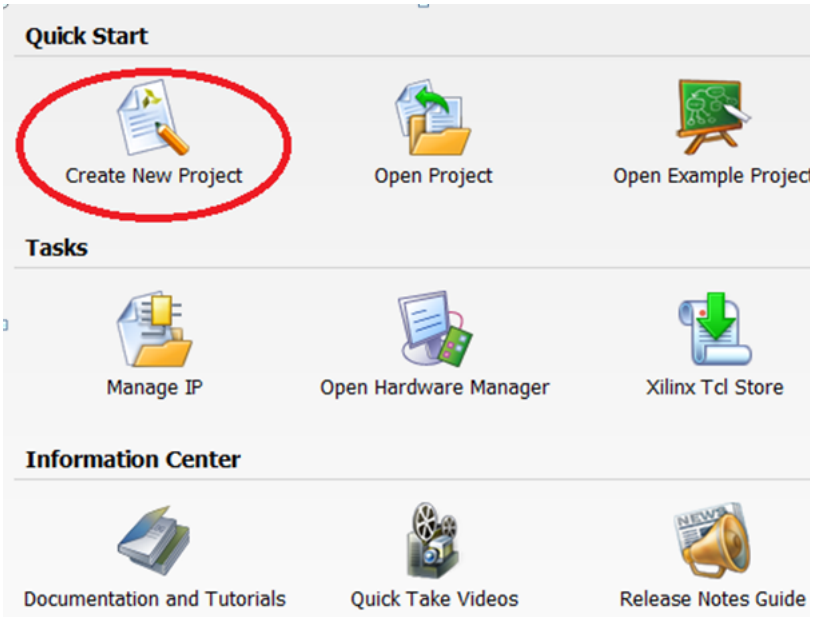


Figure 2-1: **Create New Project**

To add or create a block design in a project, you must create an RTL project, or open an Example Project as shown in Figure 2-2. You can add VHDL or Verilog design files, IP from the Vivado IP catalog, and other types of design source files to the project using the New Project wizard.

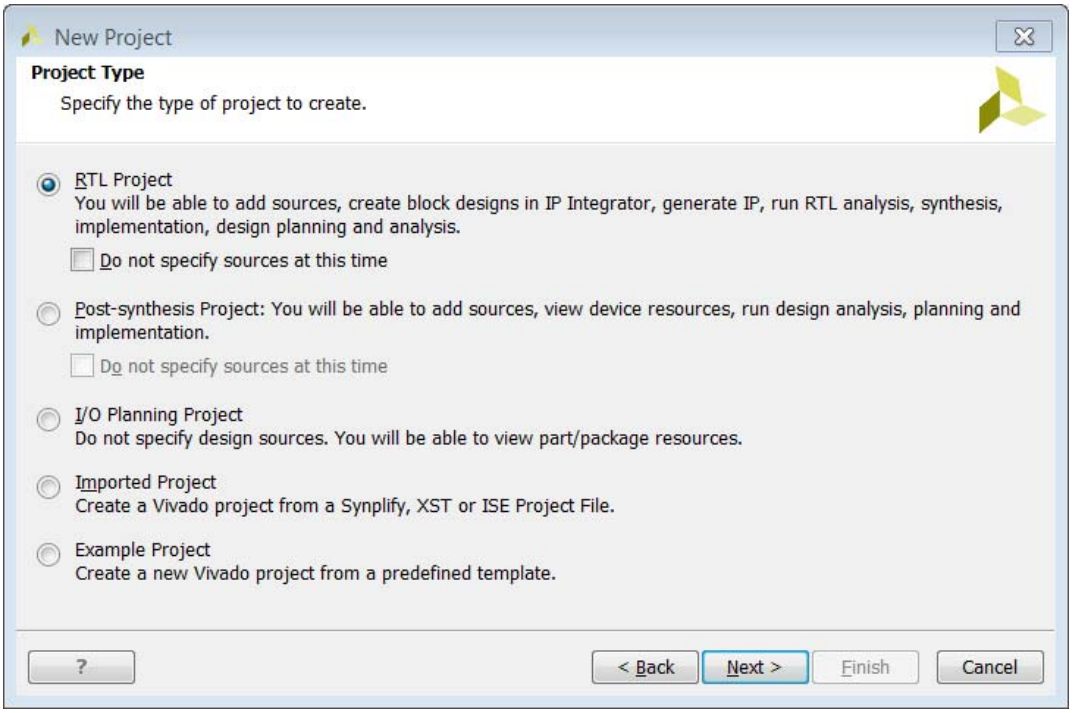


Figure 2-2: **New Project Wizard**

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 Figure 2-3, page 9. For more information, see Chapter 10, 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.*

The Tcl equivalent commands for creating a project are:

```
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. The < > symbols should not be included in the command string.

See the *Vivado Design Suite Tcl Command Reference Guide* (UG835) [Ref 1] for information on specific Tcl commands.

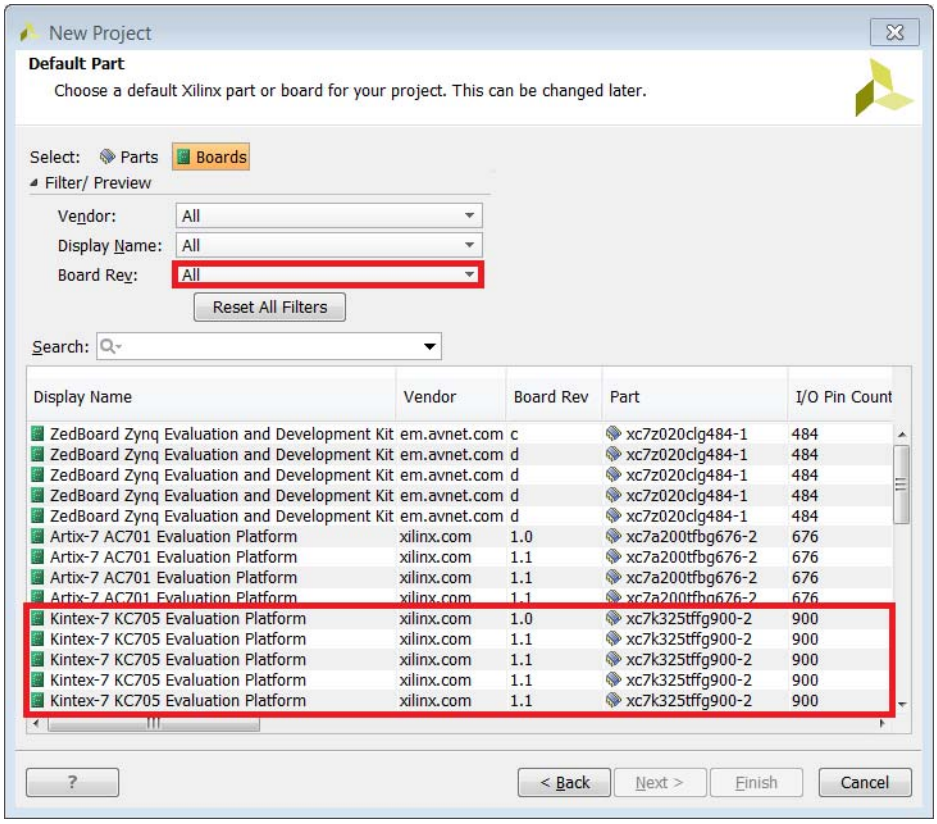


Figure 2-3: **New Project Wizard: Default Part Page**

Creating a Block Design

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

You create a new block design in the Flow Navigator by clicking the **Create Block Design** under the IP Integrator heading, as shown in the following figure.

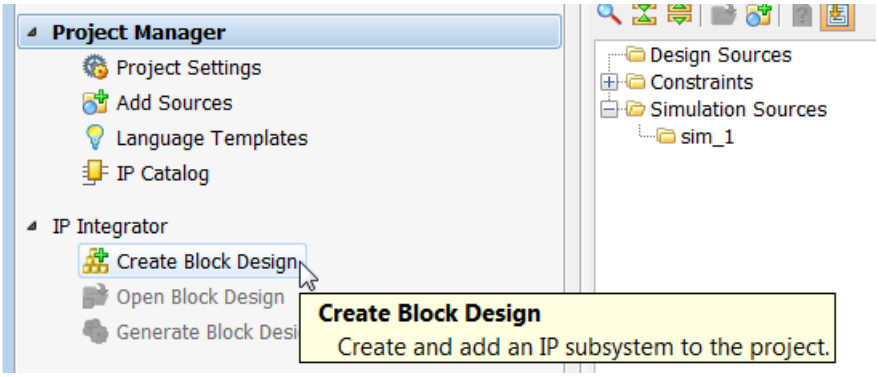


Figure 2-4: **Create Block Design**

1. Select **Flow Navigator >IP Integrator > Create Block Design**.
2. In the Create Block Design dialog box specify the **Design name**, **Directory**, and **Source Set** for the design, as shown below:

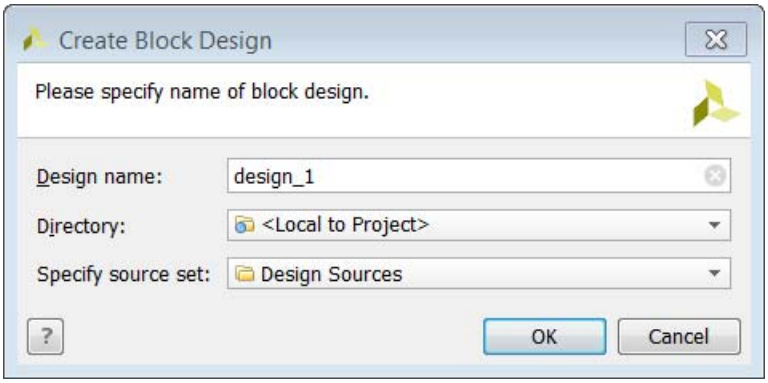


Figure 2-5: **Create Block Design Dialog Box**

The default value for the Directory field is `<Local to Project>`. You can override this default value by clicking the **Directory** field and selecting **Choose Location**.

3. Click **OK**.

The equivalent Tcl command to create a block design is:

```
create_bd_design <your_design_name>
```

IMPORTANT: *Block design names should be limited to 25 character or less to avoid any problems with the path length limitation of the Windows OS. When the specified name exceeds 25 characters the following dialog box will be displayed.*

Figure 2-6: **Dialog box warning of long block design names**

The **Create Block Design** will create an empty block design on disk, that is not automatically removed if the block design is closed without saving. The empty block design should be manually deleted from the Sources window of the Vivado IDE, with the **Remove File from Project** command, or with the following Tcl commands:

```
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>
```

Designing with IP Integrator

Once the block design is created, 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 even move the design canvas to a separate monitor by clicking on 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 the layers, click the top-left icon in the Diagram window, as shown by the red circle in the following figure. You can select the Attributes, Nets and Interface connections that you want to view or hide by checking or un-checking the boxes against these.

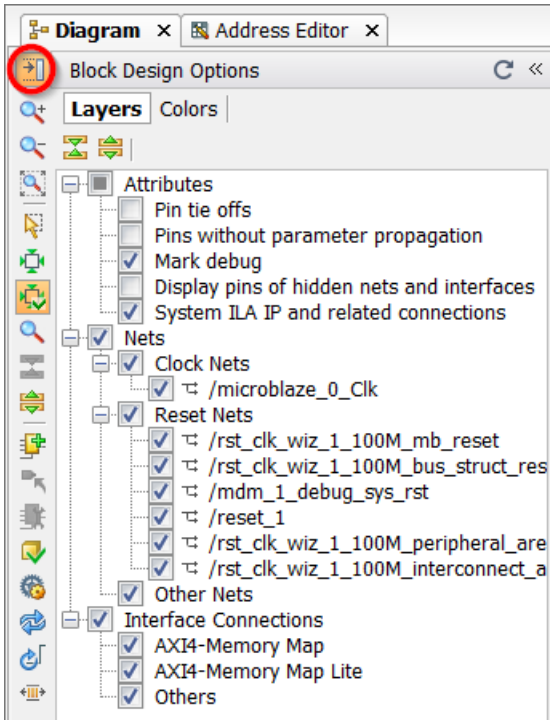


Figure 2-7: **Viewing/Hiding Information on the IP Integrator Canvas**

Attributes

Several attributes of the block design can be displayed or hidden 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 e.g. '0' or '1' can be displayed by checking the **Pin tie offs** option.

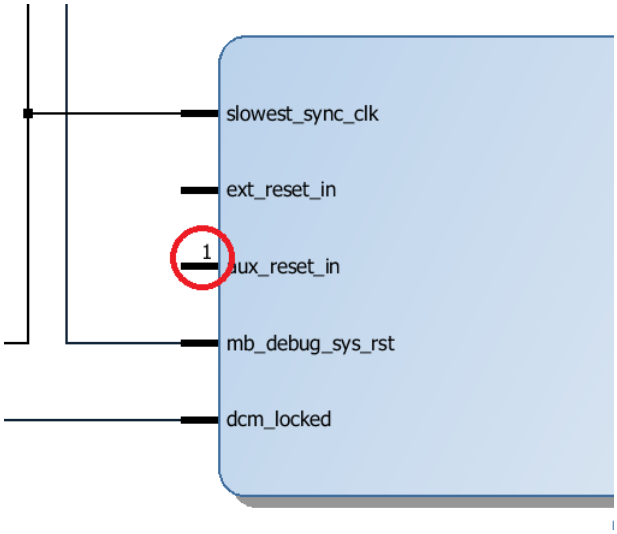


Figure 2-8: **Viewing/Hiding Pin tie-offs on the pins of IP Symbols**

- **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.

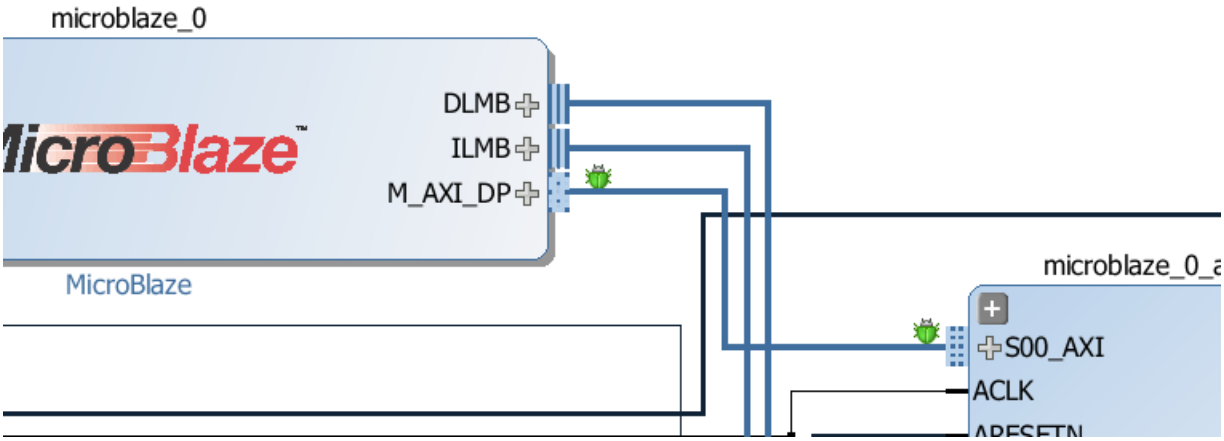


Figure 2-9: **Viewing/Hiding nets marked for debug**

- **Display pins of hidden nets and interfaces**: This option 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 may be hidden.
- **System ILA IP and related connections**: This option hides or shows 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 block design, this may unnecessarily clutter the block design 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 block design 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.

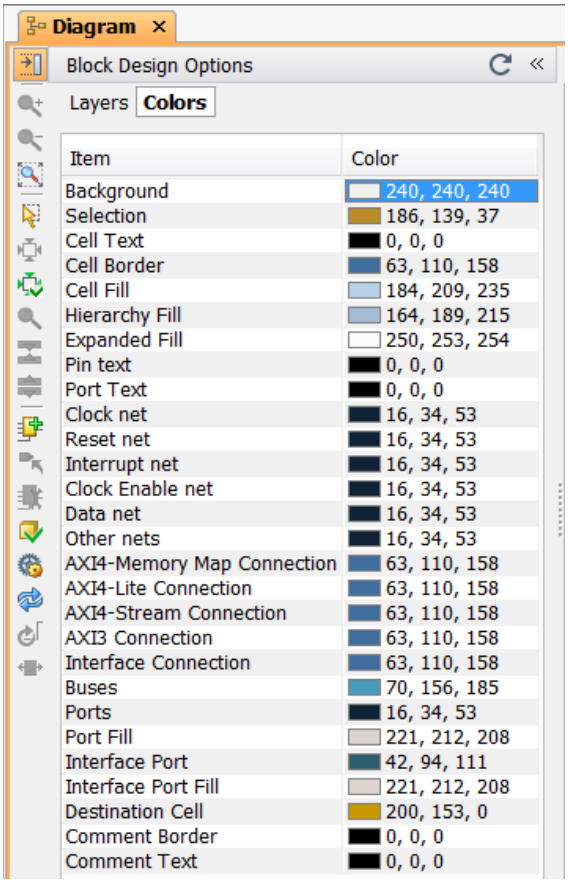


Figure 2-10: **Changing the IP Integrator Background Color**

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 **Block Design Options**, either click the close button in the upper-right corner, or click the **Block Design Options** button again.

Using Mouse Strokes and the Toolbar Buttons

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

The toolbar buttons on the left side of the design canvas allow the following commands to be invoked:

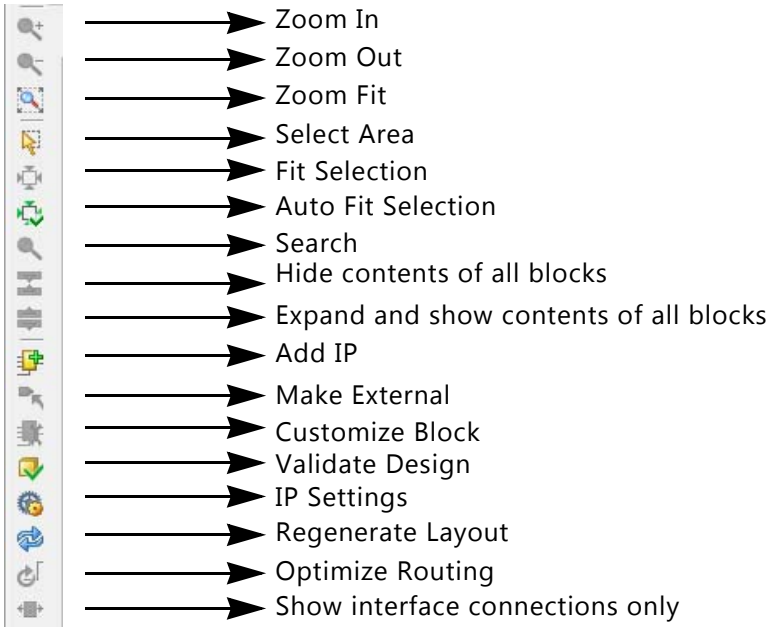


Figure 2-11: **IP Integrator Action Buttons**

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 search-able IP Catalog opens.

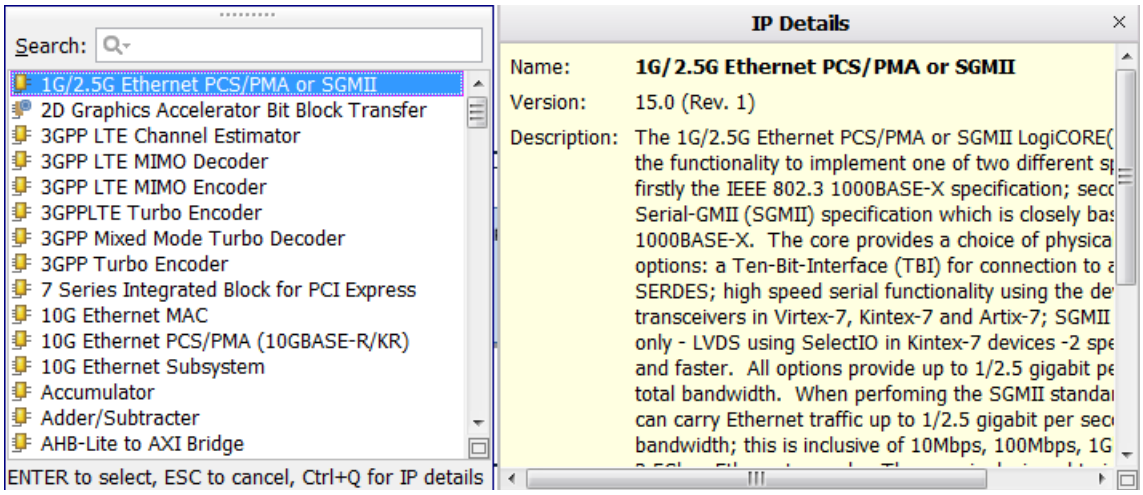


Figure 2-12: **IP Catalog**

TIP: *To enable the IP Details window of the IP Catalog, type Ctrl-Q in the catalog window.*

2. By typing in the first few letters of the IP name in the Search filter at the top of the catalog, only IP modules matching the search string are displayed.
3. To add a single IP, you can either click on the IP name and press the **Enter** key on your keyboard, or double click on the IP name.
4. To add multiple IP to the Block Design, you can highlight the additional desired IP (**Ctrl+Click**) and press the **Enter** key.

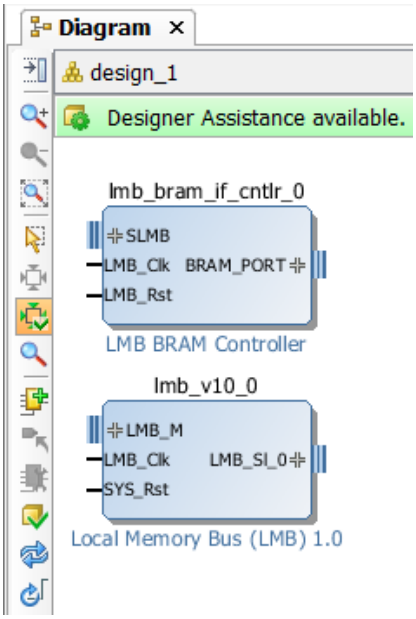


Figure 2-13: **Adding Multiple IP at the Same Time**

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

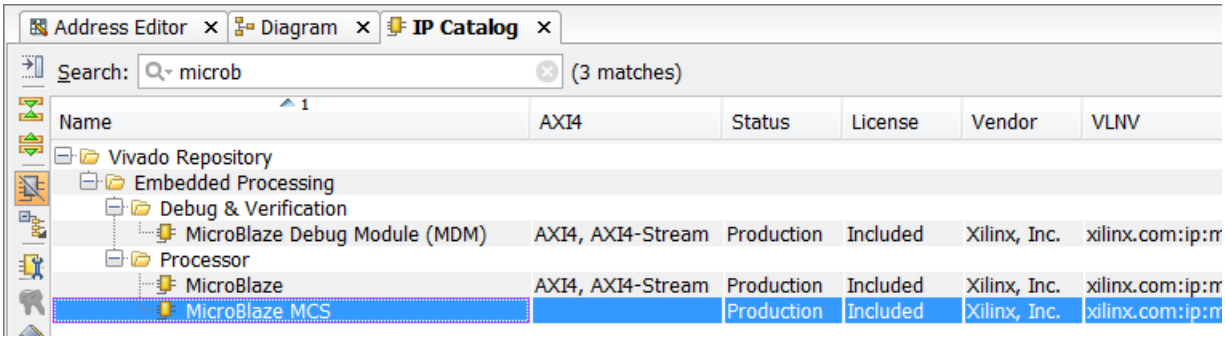


Figure 2-14: **Adding IP from the IP Catalog**

Double-click on a listed IP to add it to the open block design.

You can also float the IP Catalog by clicking on the Float icon at the upper right corner of the catalog window. Then drag and drop the IP of your choice from the IP Catalog in the block design canvas.

TIP: *Different fields associated with an IP such as Name, Version, Status, License, Vendor VLNV etc. 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 block design 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 a Verilog or VHDL source file directly into your block design. 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 block design 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 block design.

The selected module is added to the block design and you can make connections to it just as you would with any other IP in the design. The added RTL module is displayed in the block design with special markings that identify it as an RTL referenced module, as shown in Figure 2-15 on the next page. Refer to Chapter 12, Referencing RTL Modules for more information on this feature.

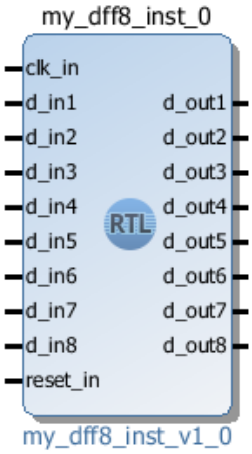


Figure 2-15: **Modules Referenced from an RTL Source file**

Hierarchical IP in IP Integrator

Some IP in the IP Catalog are hierarchical, and offer a child block design inside the top-level block design to display the logical configuration of the IP. These hierarchical IP let you see the contents of the block, but do not let you directly edit the hierarchy. Changes to the child block design 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 are Hierarchical IP in the Vivado IP Catalog. You would instantiate these IP just as any other IP by searching and selecting the IP.

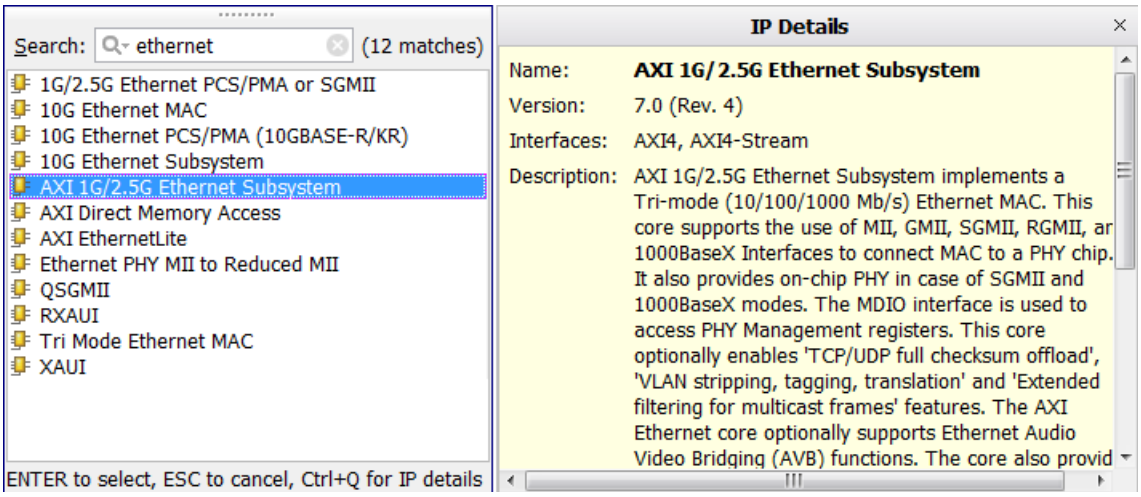


Figure 2-16: **Adding Hierarchical IP to the Block Design**

When the IP has been instantiated into a block design, double-click the IP to open the Re-customize IP dialog box where you can configure the IP parameters, as shown in Figure 2-17, page 18.

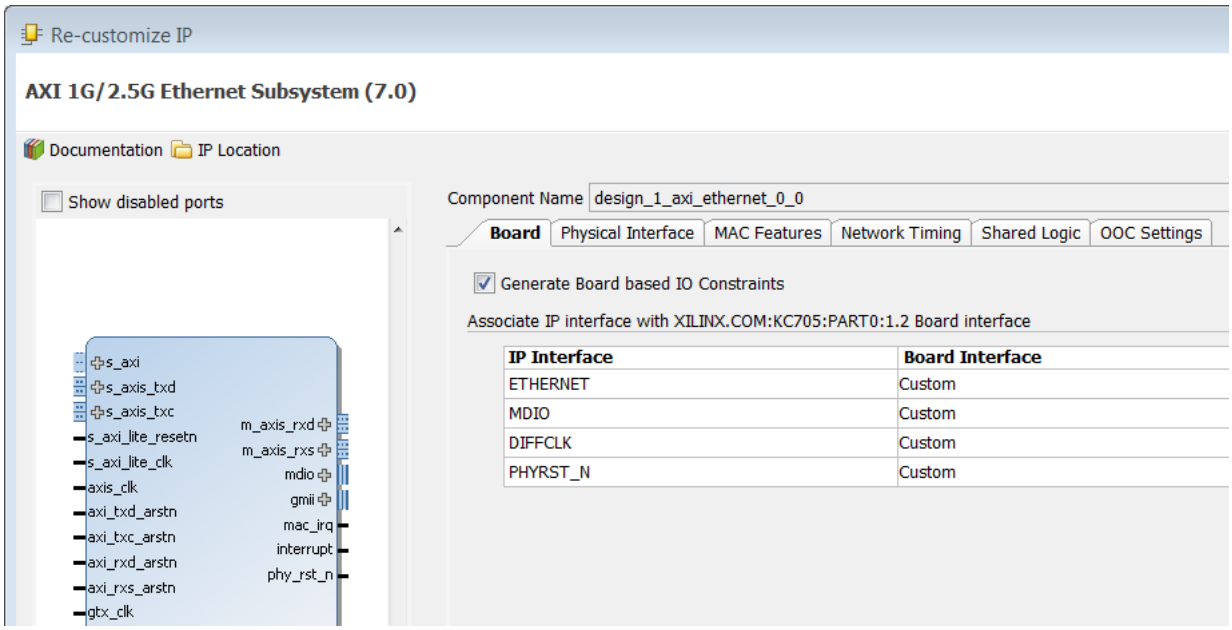


Figure 2-17: **Setting Hierarchical IP Parameters**

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

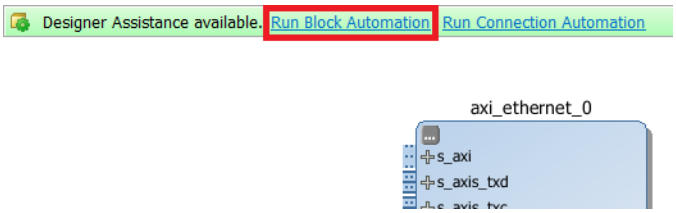


Figure 2-18: **Running Block Automation for Hierarchical IP**

Using the **Run Block Automation** dialog box you can select various parameters of the IP subsystem to be created. This puts together an IP subsystem for the mode selected.

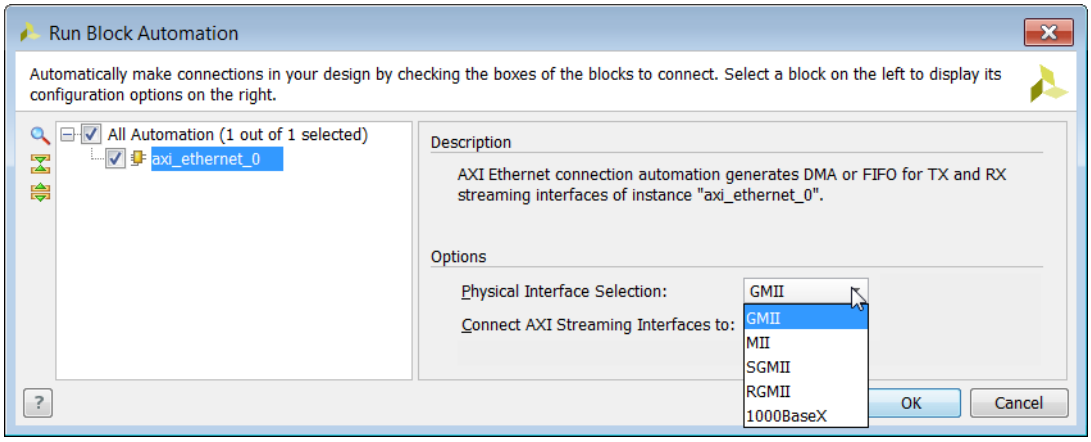


Figure 2-19: **Run Block Automation Dialog Box**

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

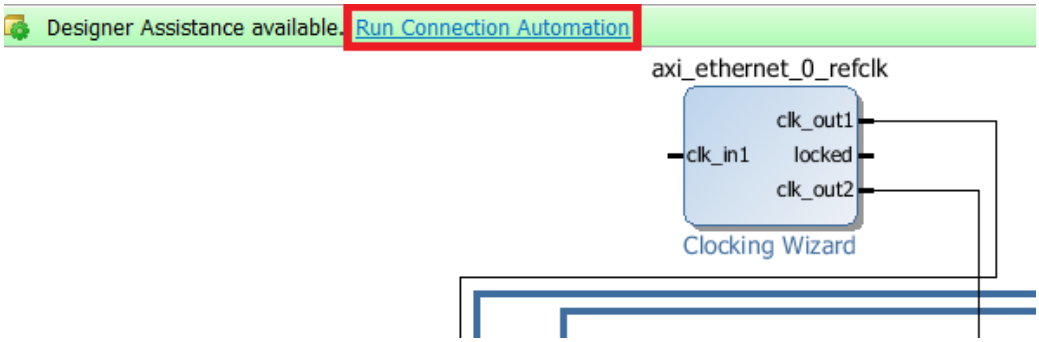


Figure 2-20: **Running Connection Automation for Hierarchical IP**

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

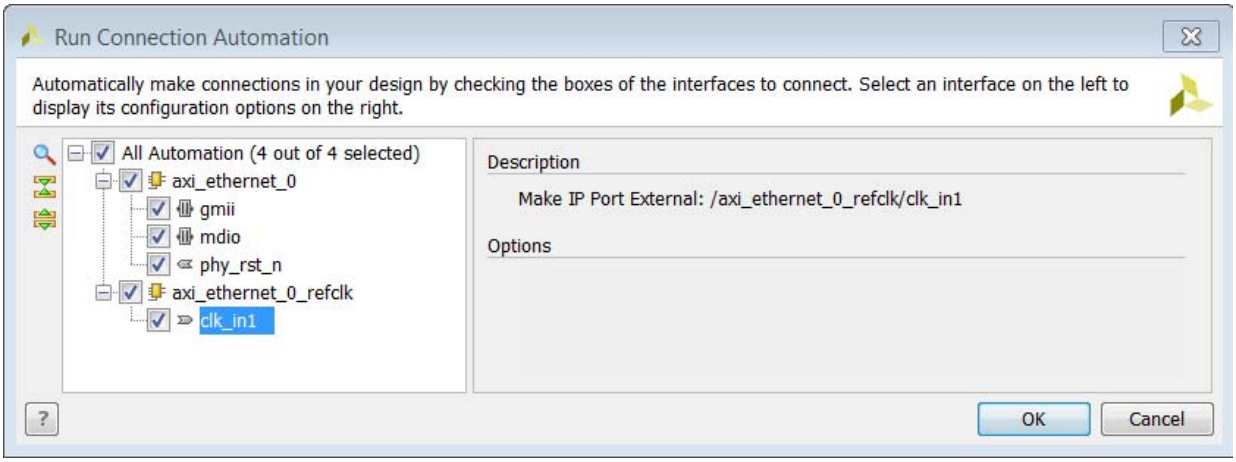


Figure 2-21: **Run Connection Automation Dialog Box**

The complete hierarchical IP subsystem should look as follows.

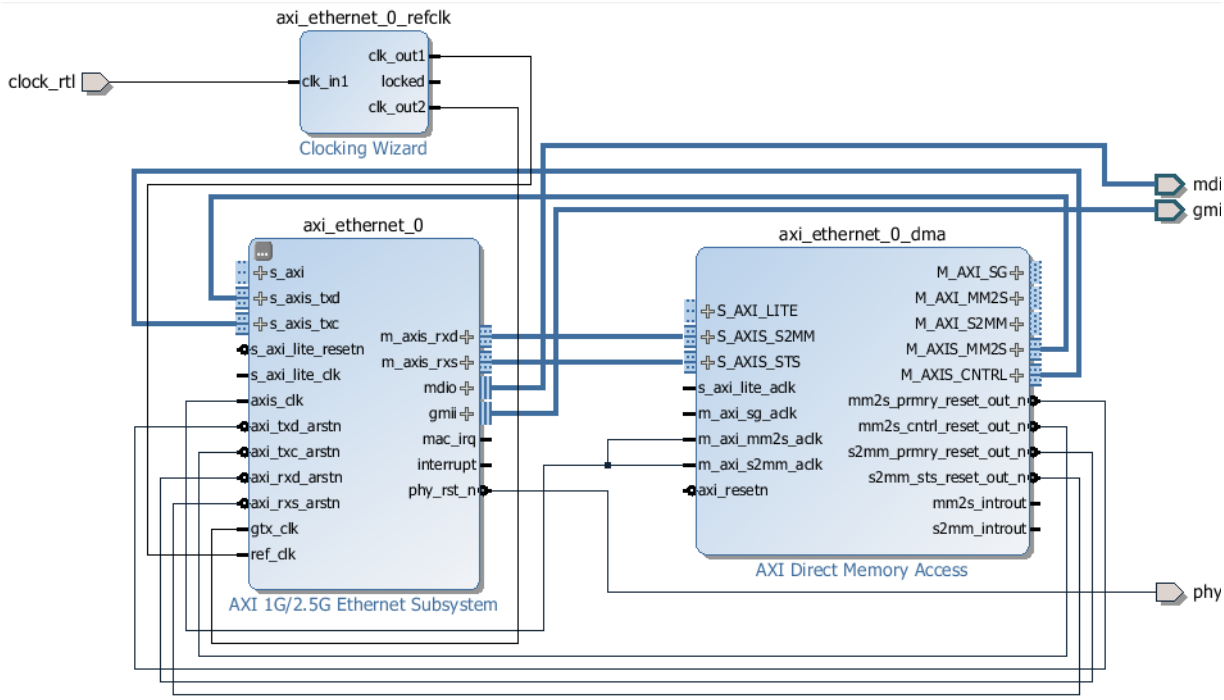


Figure 2-22: **Hierarchical IP Subsystem**

You can see the child block design inside the AXI Ethernet subsystem IP by right-clicking and selecting **View Block Design** command, as shown in Figure 2-23, page 20. You can also view the block design by clicking on the **View Block Design** icon at the top left corner of the IP symbol. This opens a block design window showing the child-level block design, as shown in Figure 2-24, page 21.

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

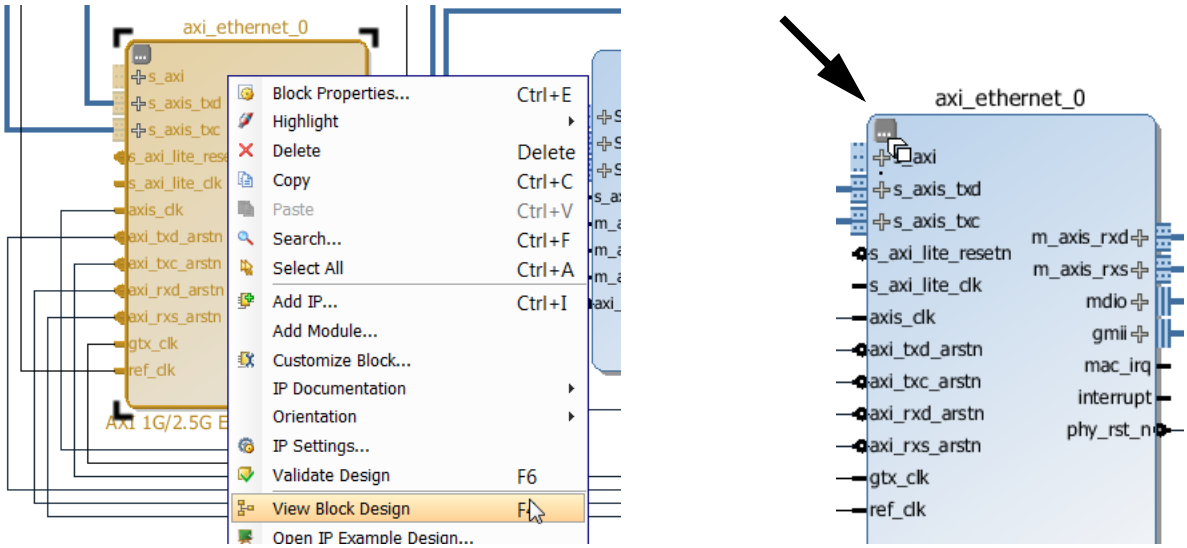


Figure 2-23: **View Block Design**

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

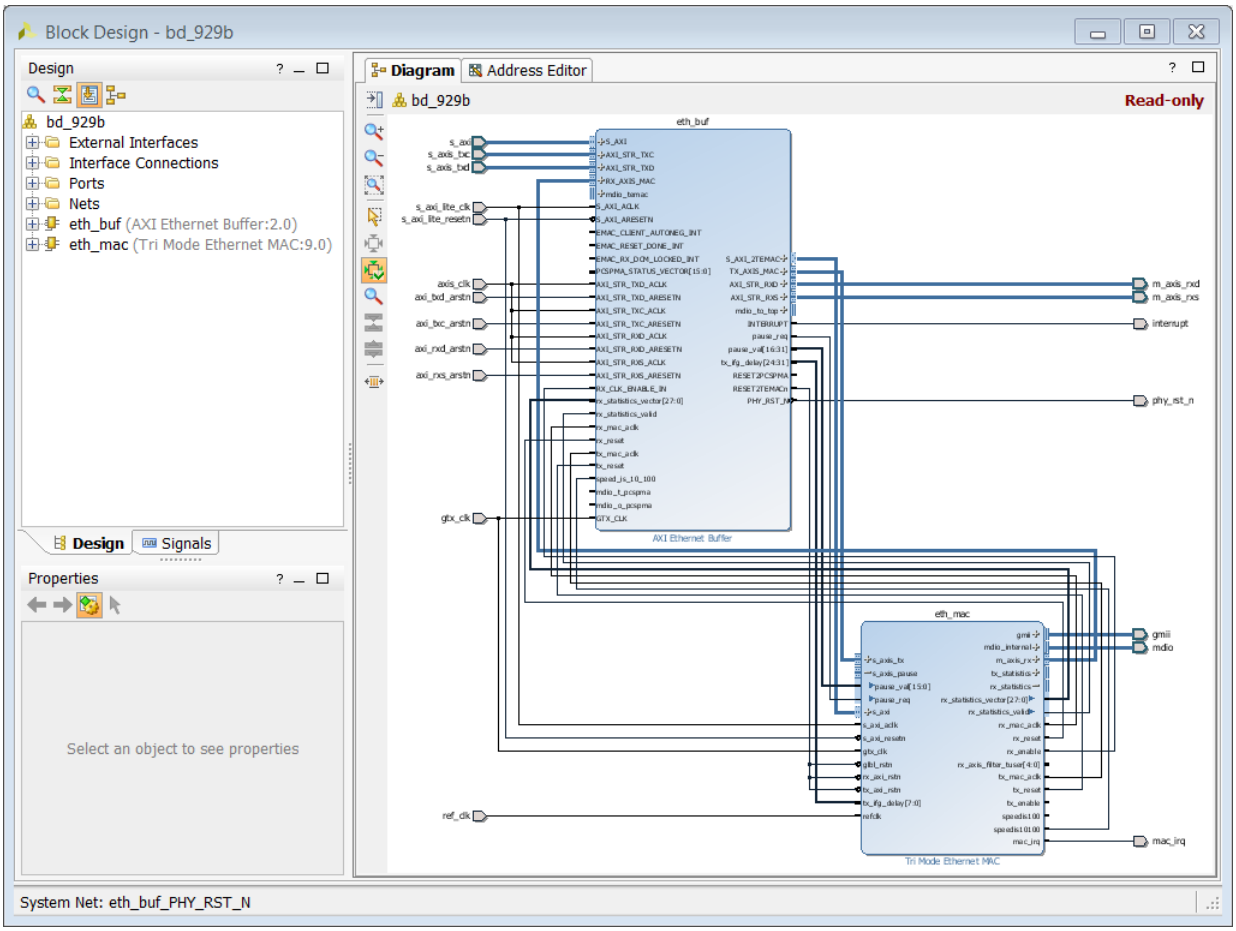


Figure 2-24: **Child Block Design in Hierarchical IP**

Glue Logic IP in IP Integrator

There are several IP available in the IP Catalog for use in Vivado IP Integrator designs as glue logic. These IP are briefly described below, 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 logic operations supported 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. Refer to the *LogiCORE IP Utility Vector Logic* (PB046) [Ref 18] for more information.

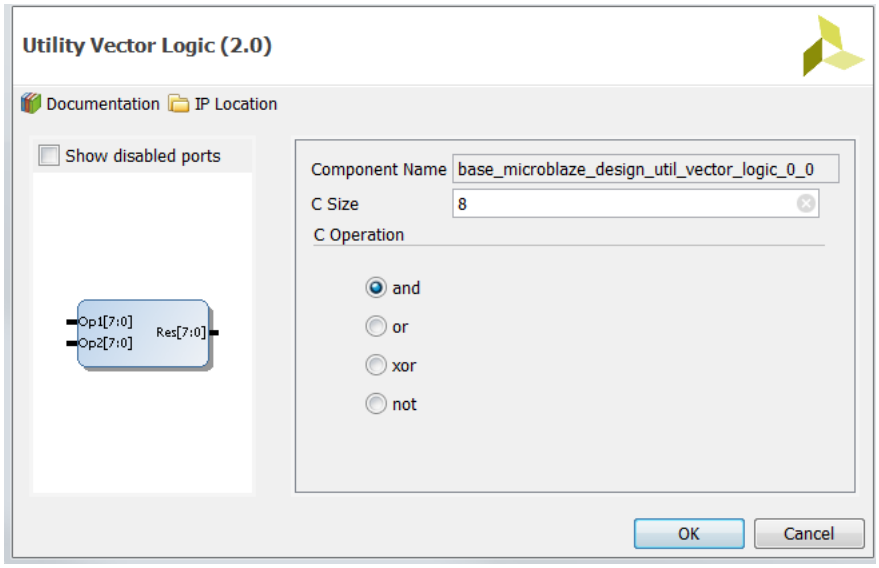


Figure 2-25: **Utility Vector Logic IP dialog box**

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* (PB045) [Ref 19] for more information.

For example, setting the C Size to 8 as an "AND" function will create one 8 input AND gate, with a single output.

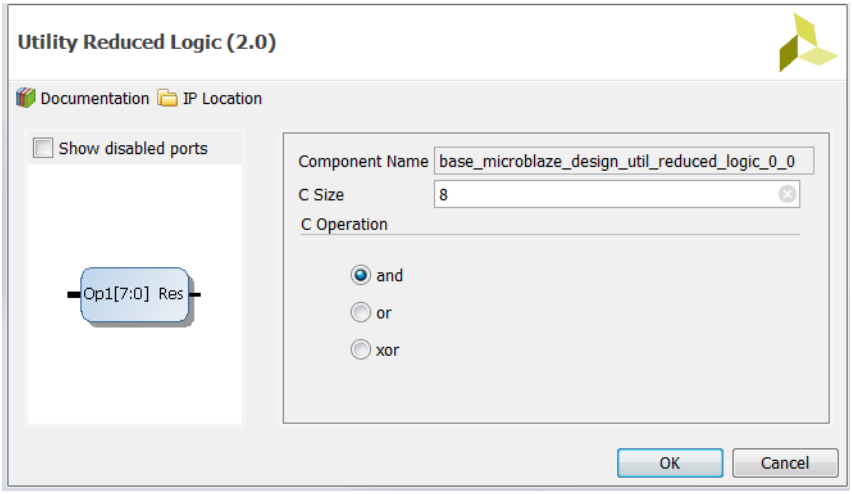


Figure 2-26: **Utility Reduced Logic IP dialog box**

Constant

Use the Constant IP to tie signals up or down, and specify a constant value. Refer to the *LogiCORE IP Constant* (PB040) [Ref 20] for more information.

Utility Buffer

There are occasions when you need to manually insert a clock or signal buffer into a block design. You can use the Utility Buffer IP in these situations to configure and instantiate one of several different buffer types into the design. Refer to the *LogiCORE IP Utility Buffer* (PB043) [Ref 23] 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 the sources can be of different width, as automatically determined by IP Integrator or specified by you. The resulting output is a bus that combines the source signals together. Refer to the *LogiCORE IP Concat* (PB041) [Ref 21] for more information.

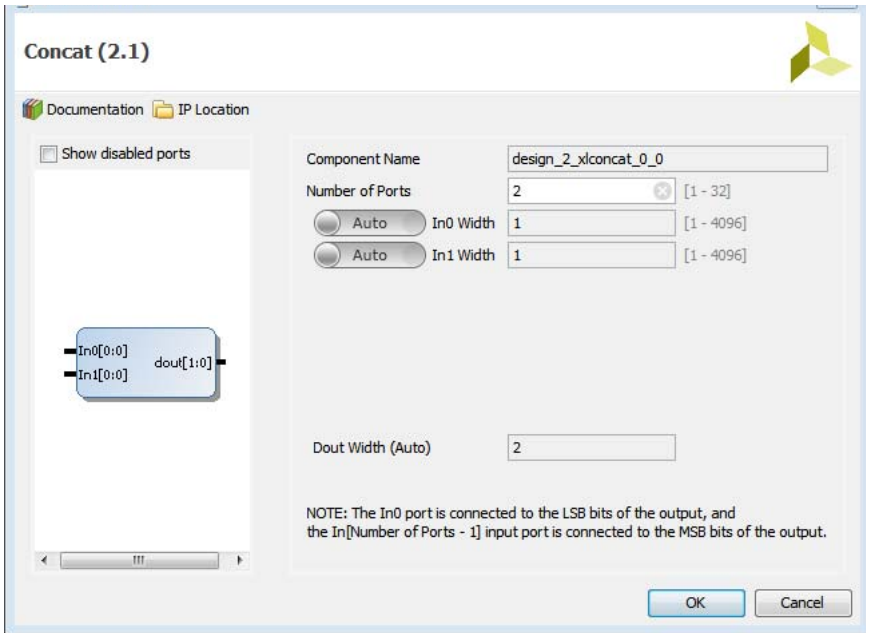


Figure 2-27: **Concat IP dialog box**

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. Refer to the *LogiCORE IP Slice* (PB042) [Ref 22] for more information.

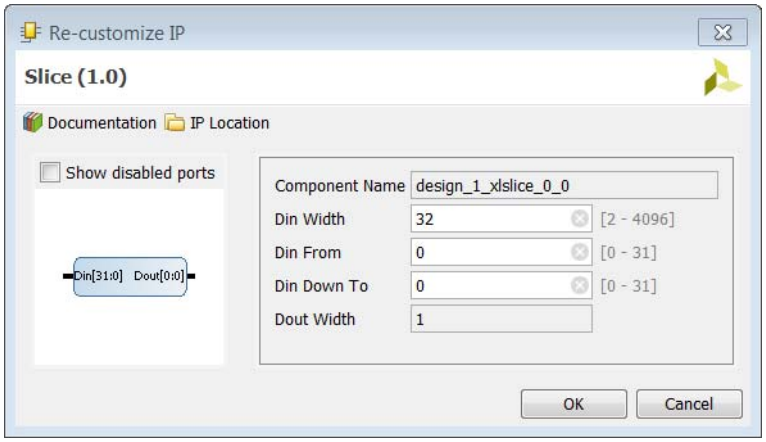


Figure 2-28: **Slice IP dialog box**

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

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 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 on an interface or pin connector on 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.

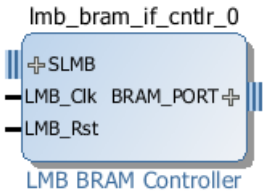


Figure 2-29: **Connection Box on a Symbol**

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.

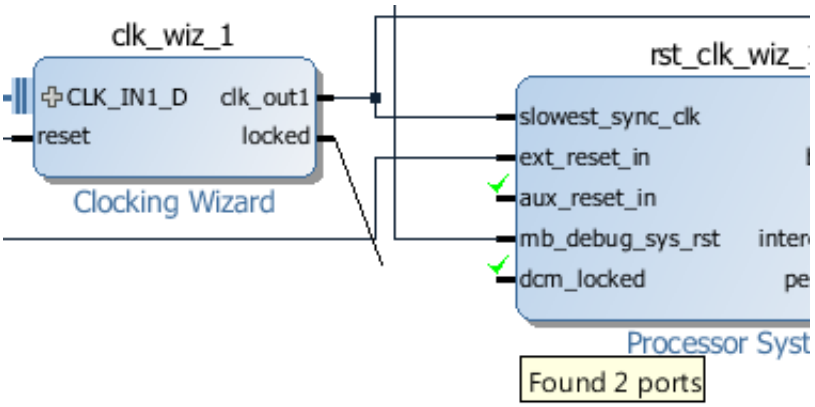


Figure 2-30: **Signal or Bus Connection on a Symbol**

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 Figure 2-31, 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 Figure 2-31, 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 block design. The individual signal is essentially removed from the interface. The entire signal must be manually connected.*

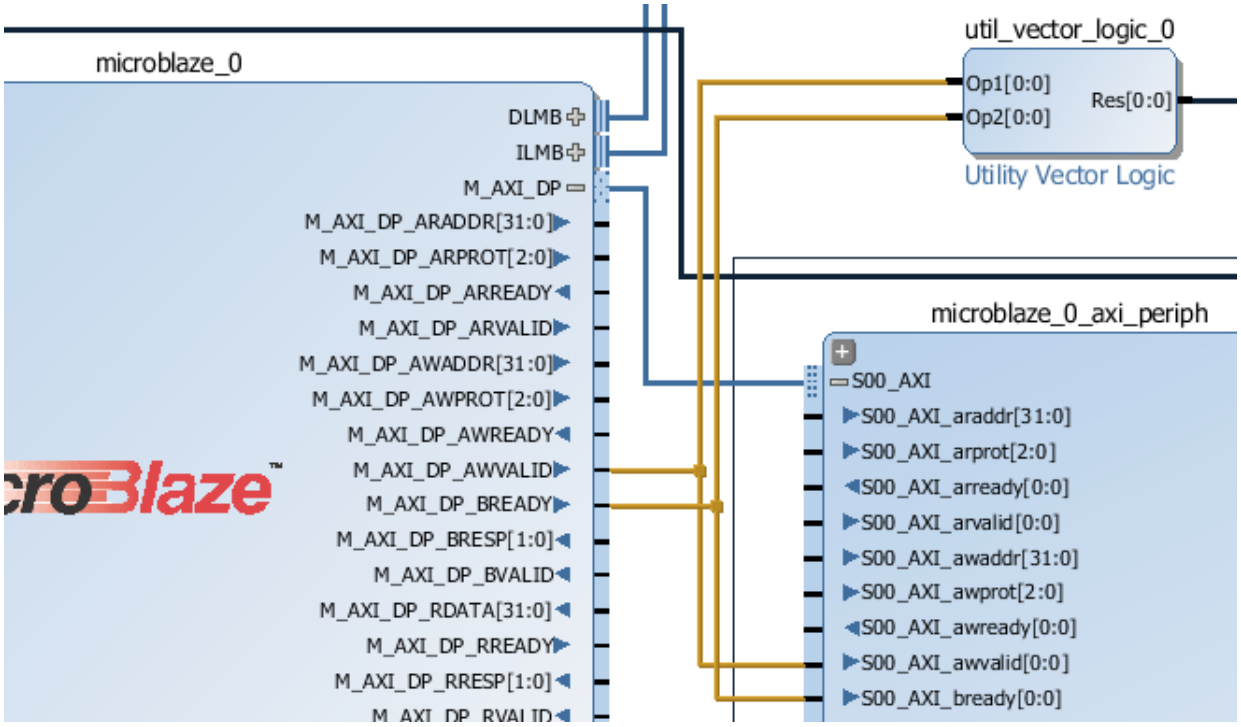


Figure 2-31: **Expanding the Interface to Make a Connection**

When connections to an interface pin are overridden by connection to individual signals or bus pins of the interface, a warning is issued 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 since the connection will 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. However, as seen in Figure 2-32, the separately connected signals or buses of the interface will continue to be shown as needed to properly display the connections of the block design.

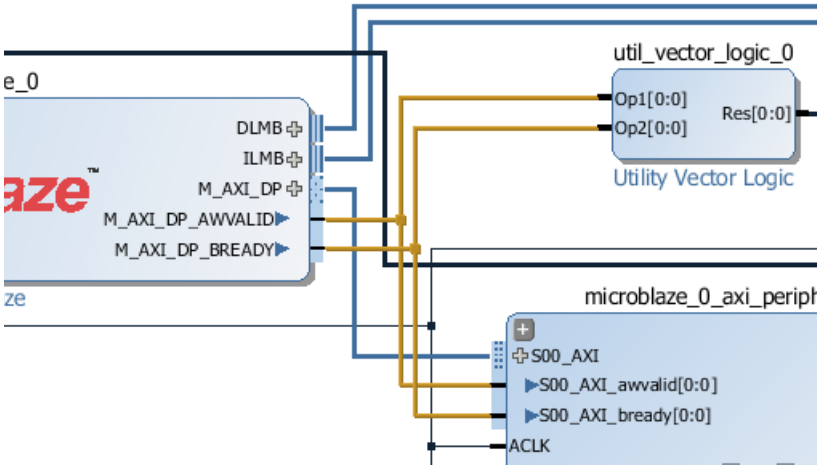


Figure 2-32: **Collapsing the Interface**

External Connections

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

- Make External
- Create Port
- Create Interface Port

These options are described in the following sections.

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

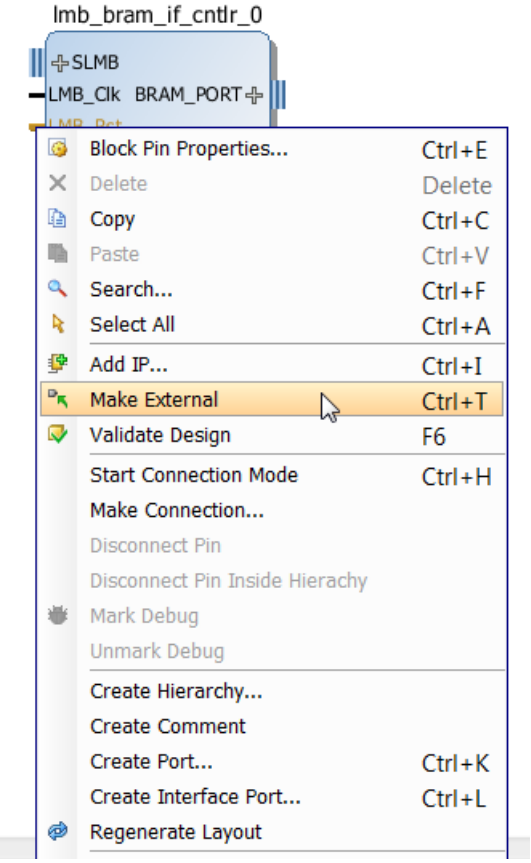


Figure 2-33: **Making External Connections**

This command ties a pin on an IP to an I/O port on the block design. 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`.

   Create Port gives you more control in specifying the input and output, the bit-width and the type (such as `clk`, `reset`, and `data`).

   When you specify a clock, you can also specify the input frequency.

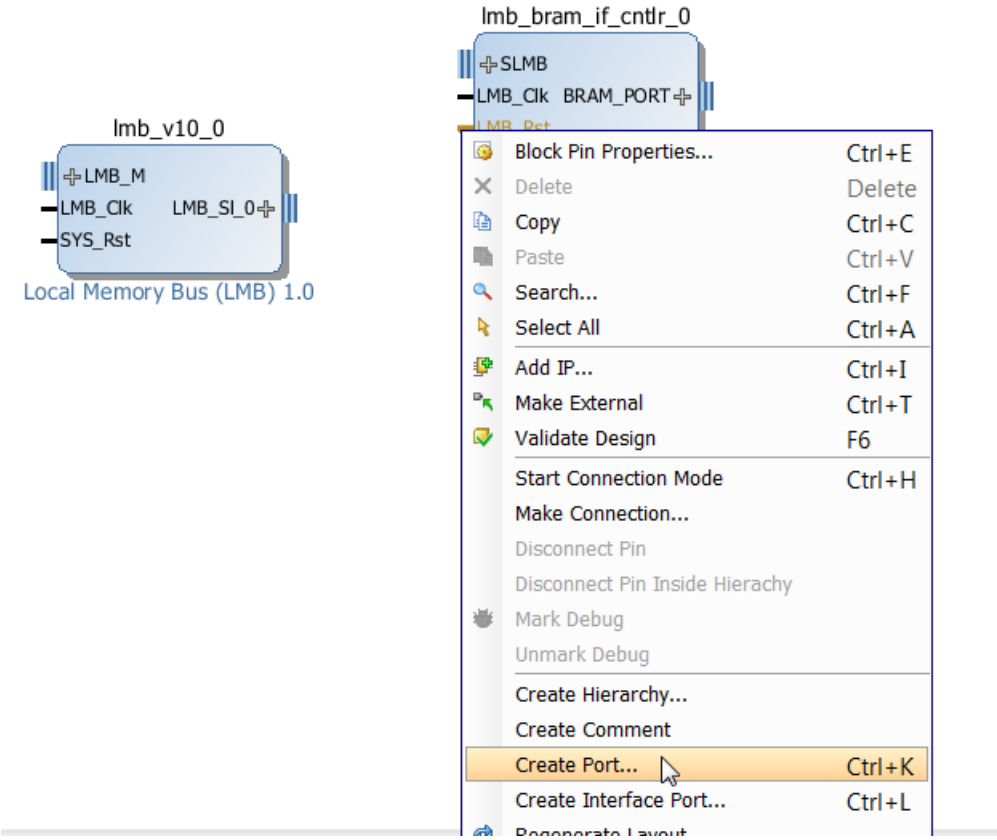


Figure 2-34: **Create Port Command**

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

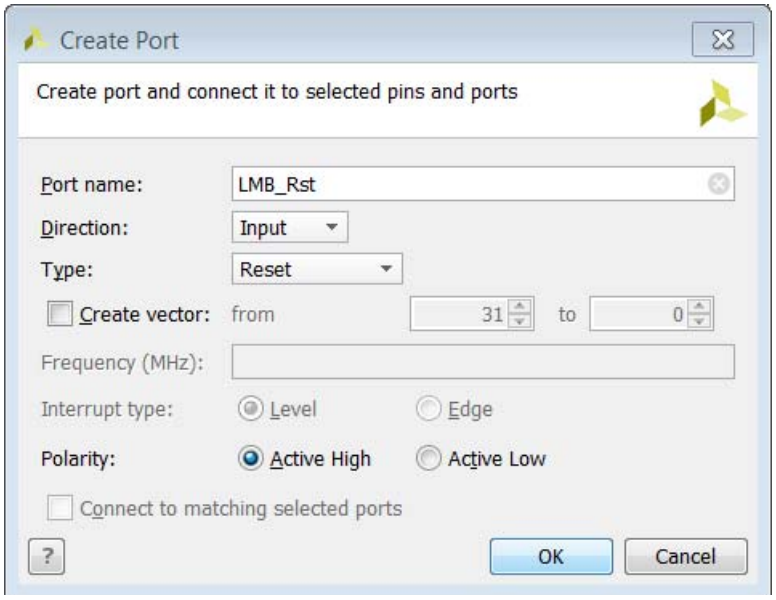


Figure 2-35: **Create Port Dialog Box**

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.

Creating Interface Ports

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

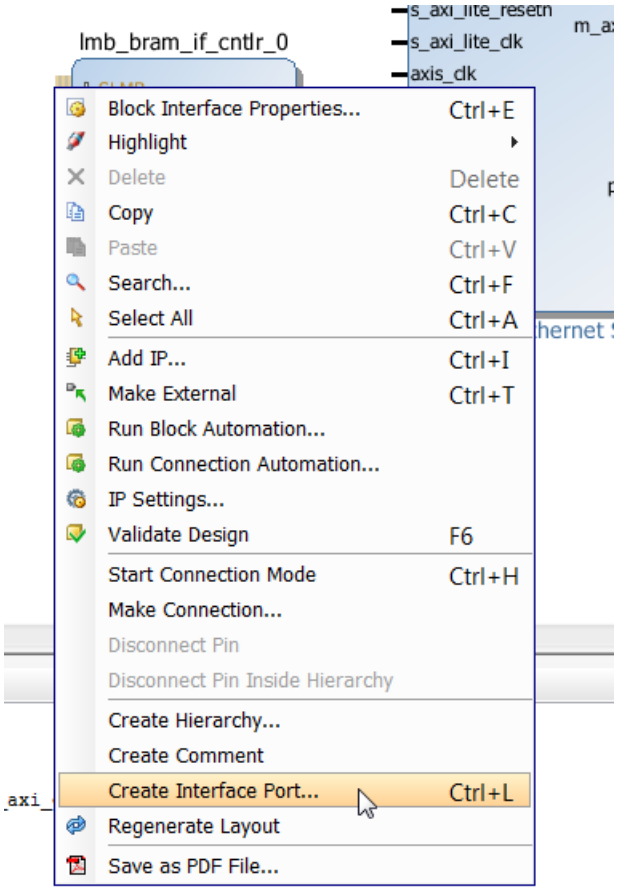


Figure 2-36: **Create Interface Port Command**

This command creates ports on the interface pins which are groupings of signals that share a common function. For example, the `LMB_M` and `LMB_SI_0` are interface pins in the figure above. 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**.

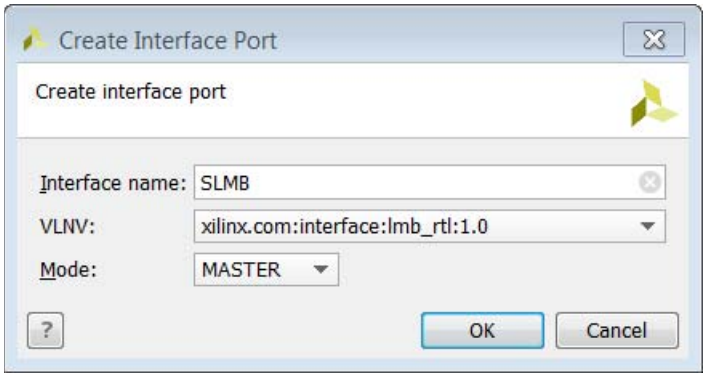


Figure 2-37: **Create Interface Port Dialog Box**

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

   In the figure below, 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).

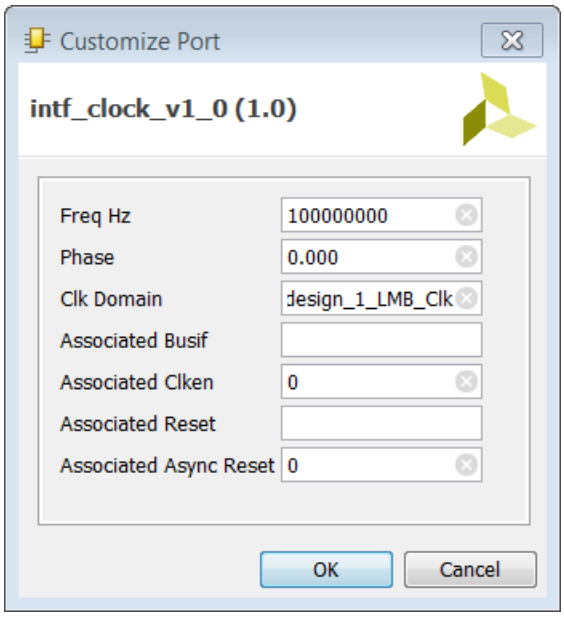


Figure 2-38: **Customize Port Dialog Box**

On an AXI interface, double-clicking the port shows the following configuration dialog box.

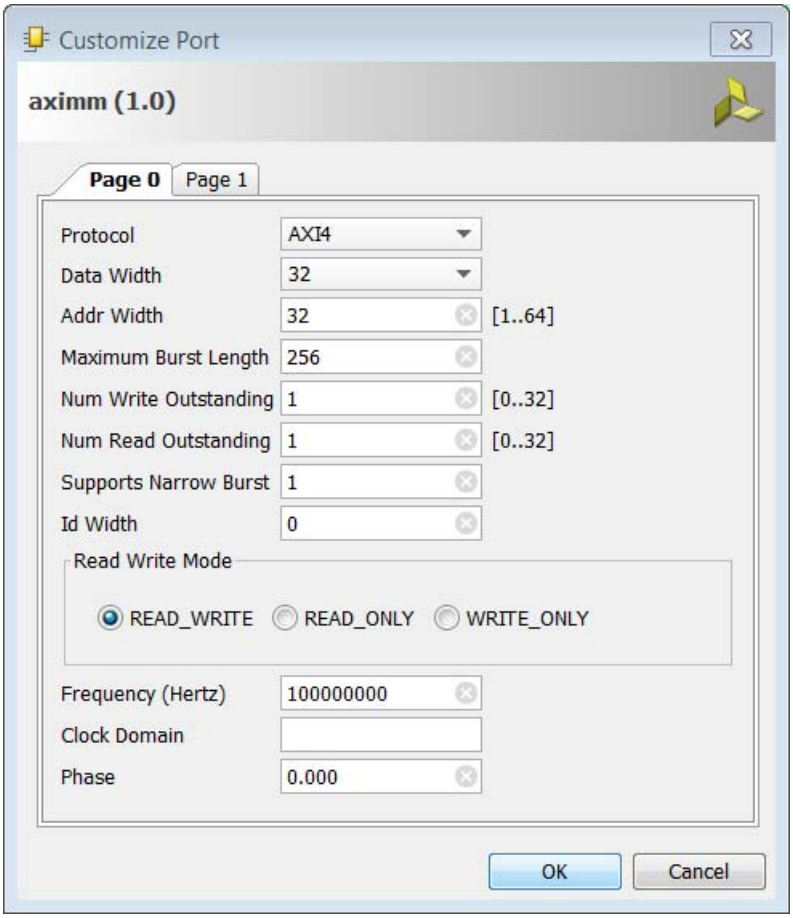


Figure 2-39: **Customizing Port Properties of aximm**

Handling Interrupts

Interrupt handling in the Vivado Design Suite IP Integrator tool depends on the processor being used. For a Zynq®-7000 processor, the Generic Interrupt Controller block within the Zynq-7000 processor handles the interrupt.

For a MicroBlaze™processor, the AXI Interrupt Controller IP must be used to manage interrupt. Regardless of the processor used in the design, a Concat IP consolidates and drives the interrupt pins.

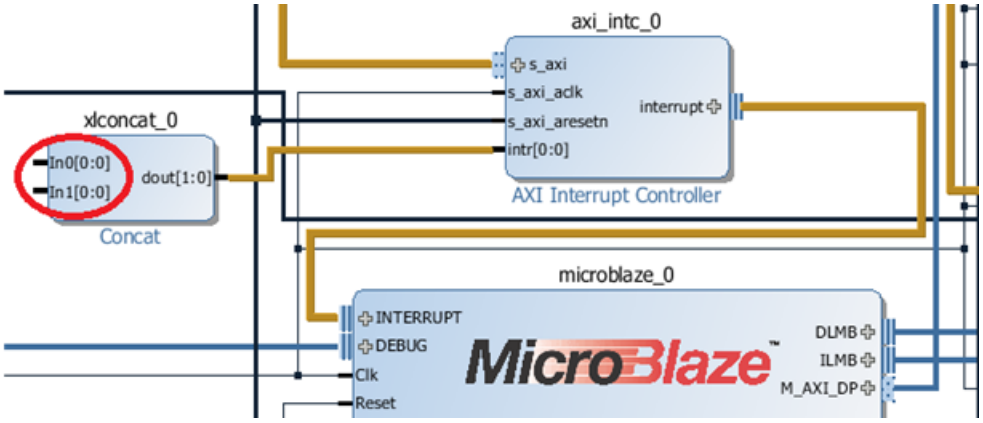


Figure 2-40: **Concat IP Driving Interrupt Input to AXI Interrupt Controller**

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

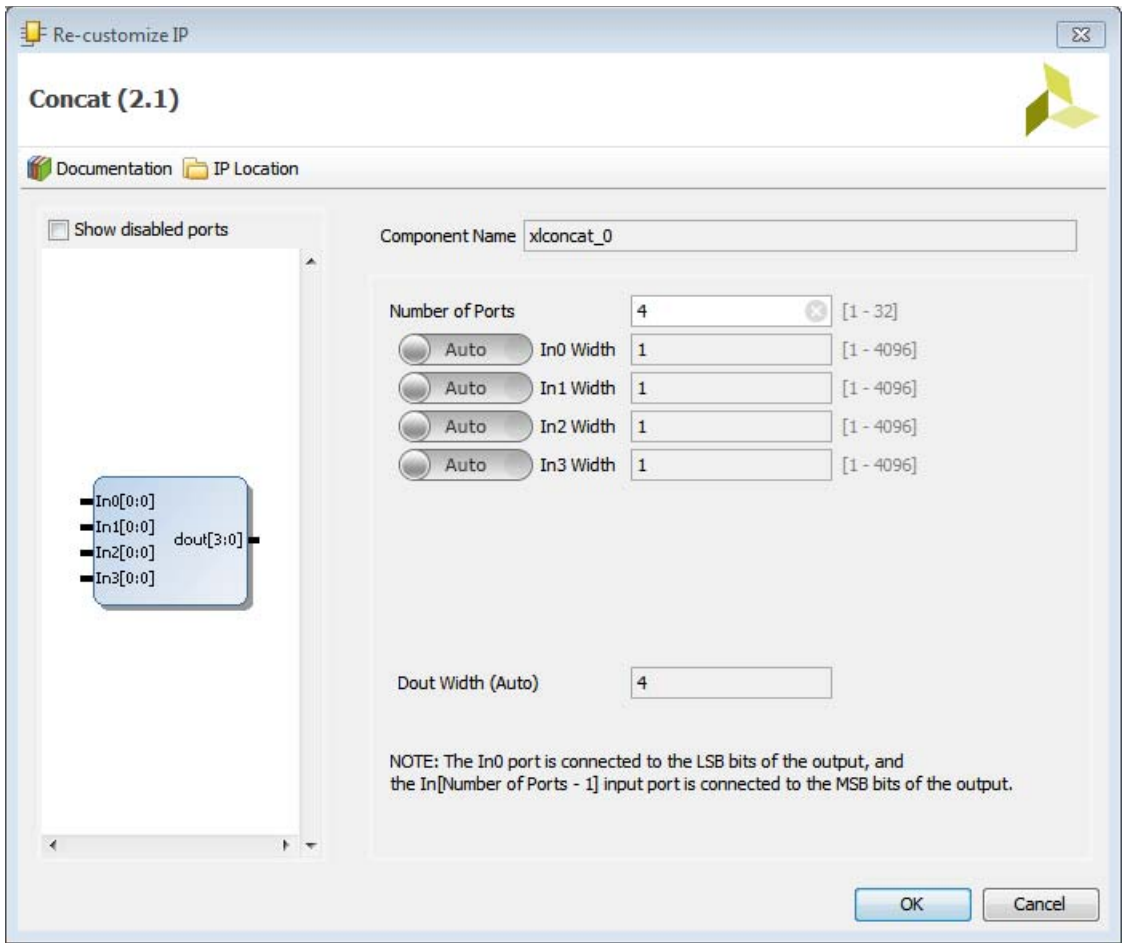


Figure 2-41: **Concat Re-customize IP Dialog Box**

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:

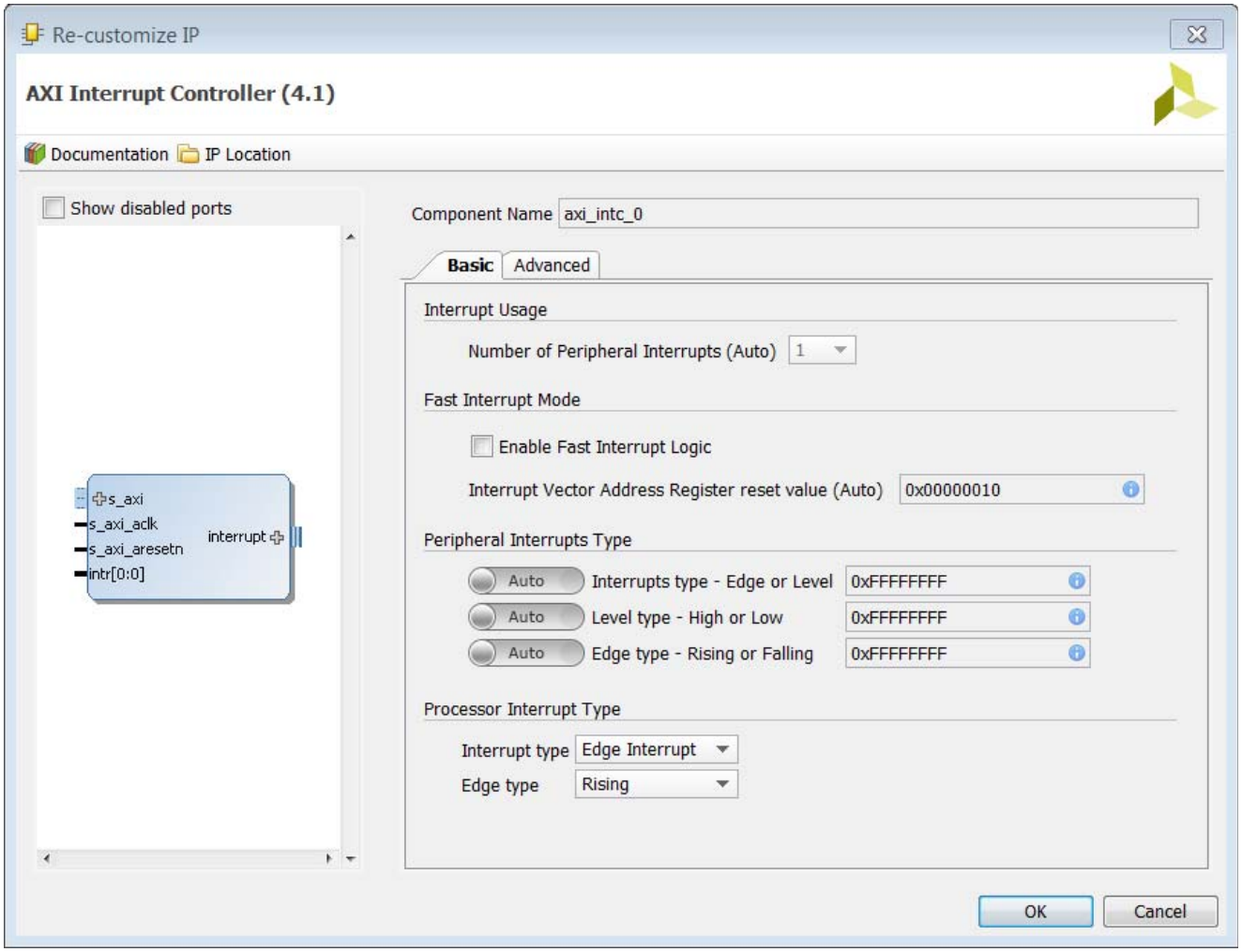


Figure 2-42: **AXI Interrupt Controller Basic Tab Parameters**

- The **Number of Peripheral Interrupts** cannot be set by the user. This is automatically set during parameter propagation. This 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, users 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**.

    Users can select if the interrupt source is either **Edge-triggered** or **Level-triggered**. Accordingly, then can also select whether the interrupt is rising or falling edge and in case of 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 *LogiCORE IP AXI Interrupt Controller* (PG099) [Ref 14] for details of these parameters.

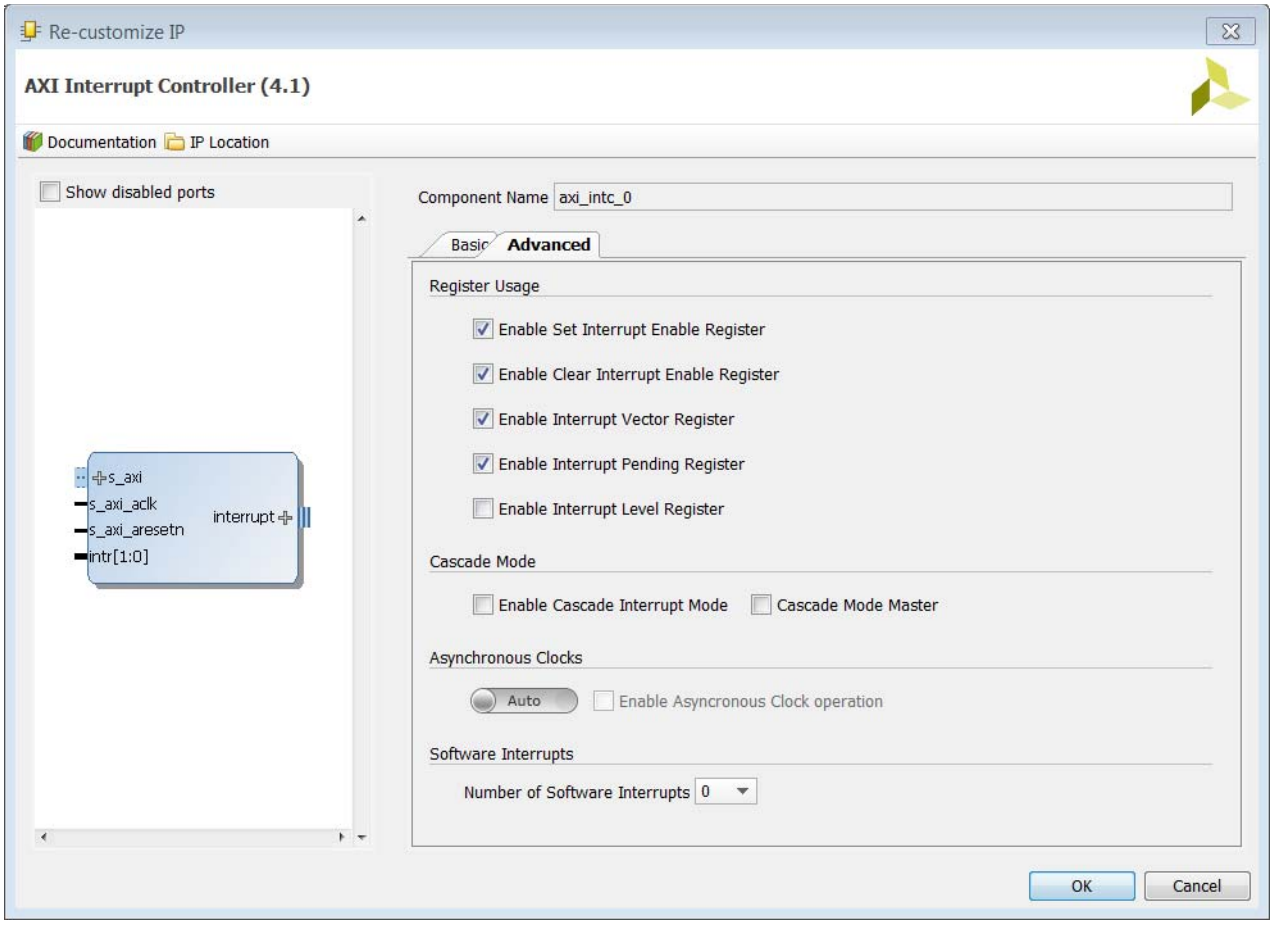


Figure 2-43: **Interrupt Controller Advanced Tab**

One option to notice 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 **Enable Asynchronous Clock operation** is enabled 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.*

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 microprocessor 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® Processing System 7 or MicroBlaze™ processor, or some other hierarchical IP such as an Ethernet is instantiated in the IP Integrator block design.

Click the **Run Block Automation** link in the banner of the design canvas, as shown in the following figure, for assistance in putting together a simple MicroBlaze system.

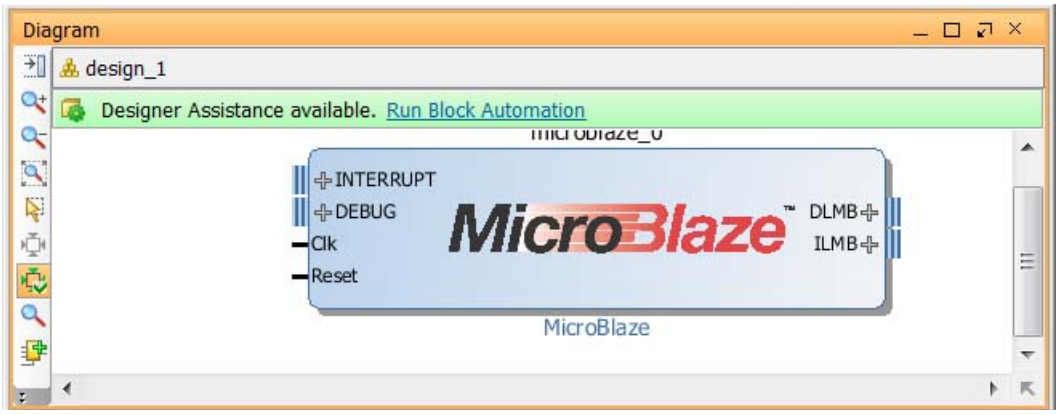


Figure 2-44: **Run Block Automation Feature**

The **Run Block Automation** dialog box opens, as shown in Figure 2-45, page 37, and lets you provide input about basic features that the microprocessor system needs.

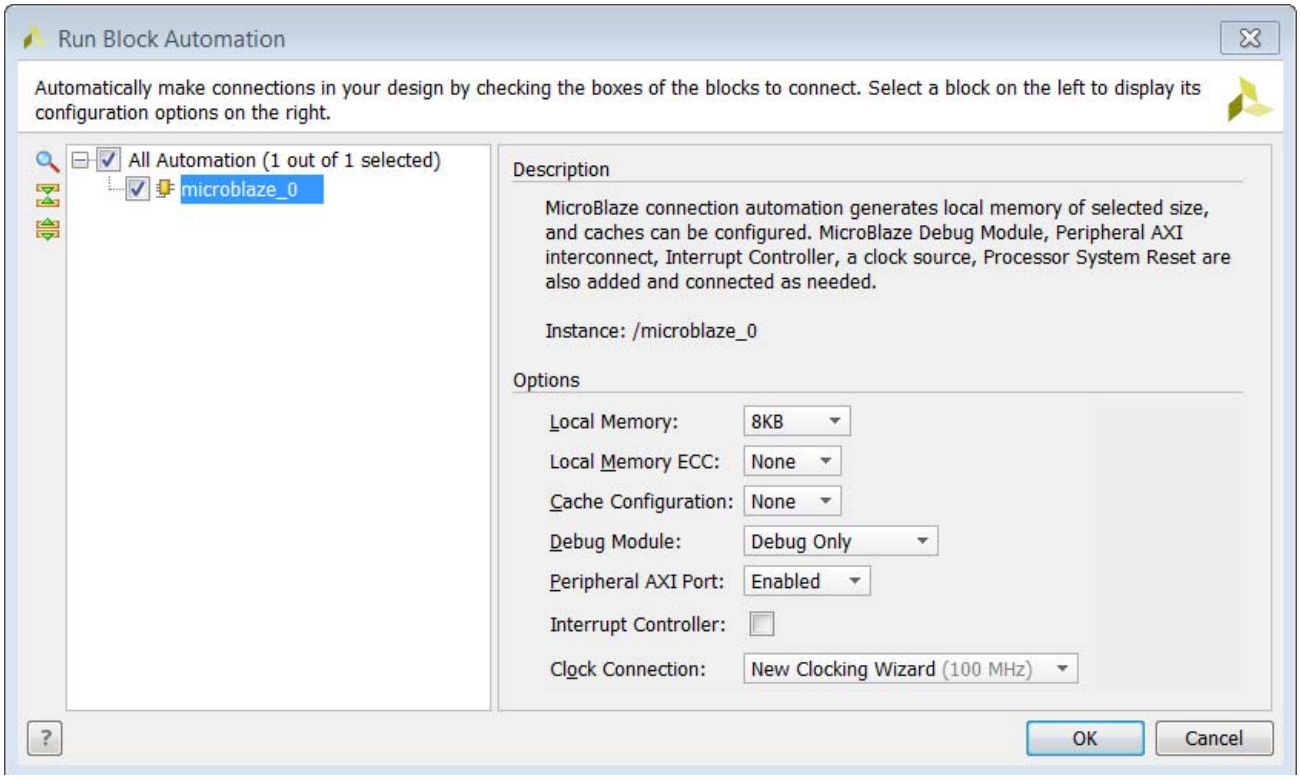


Figure 2-45: **Run Block Automation Dialog Box**

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

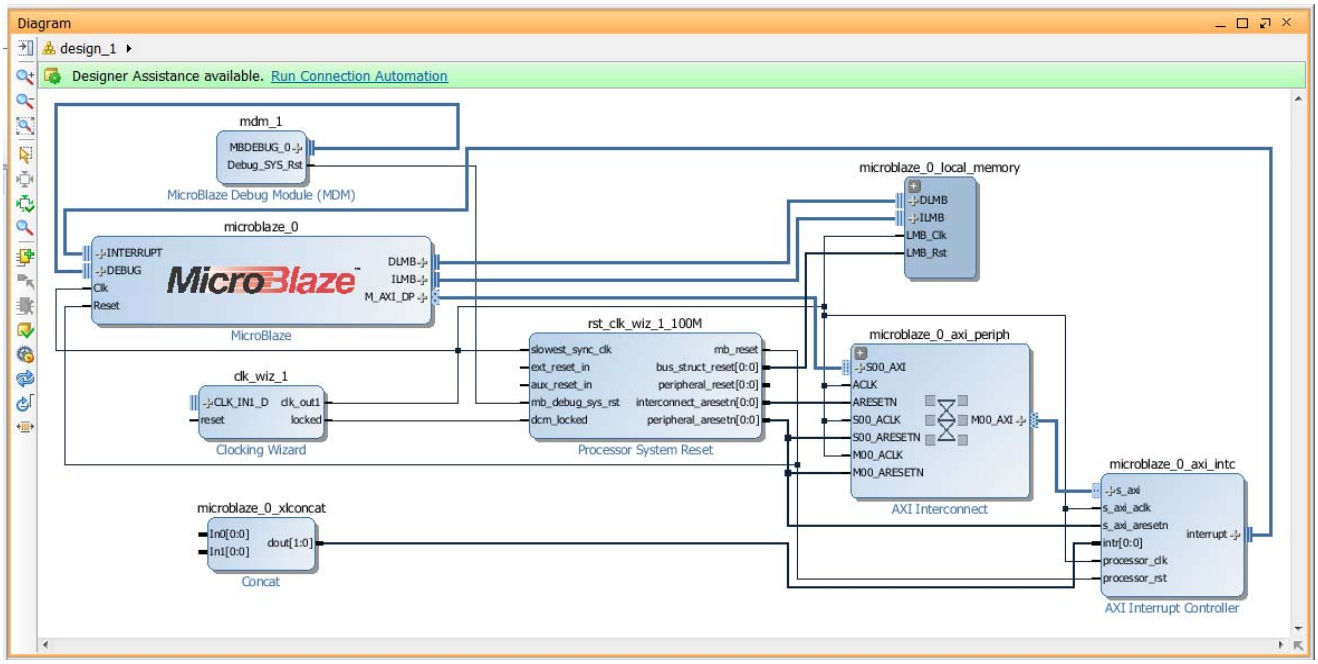


Figure 2-46: **MicroBlaze System Created by Block Automation**

The MicroBlaze System shown in Figure 2-46, page 37 consists of a MicroBlaze Debug Module, which is 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 and an AXI Interrupt Controller.

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

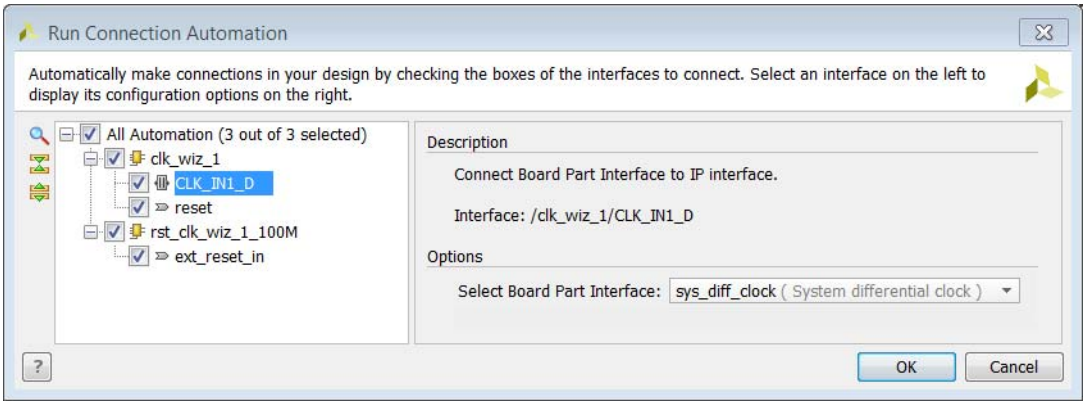


Figure 2-47: **Ports and Interfaces that can use Connection Automation**

For Xilinx Target Reference Platforms or evaluation boards, IP Integrator has knowledge of the FPGA pins that are used on the target boards. 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 in Figure 2-46, page 37, 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.

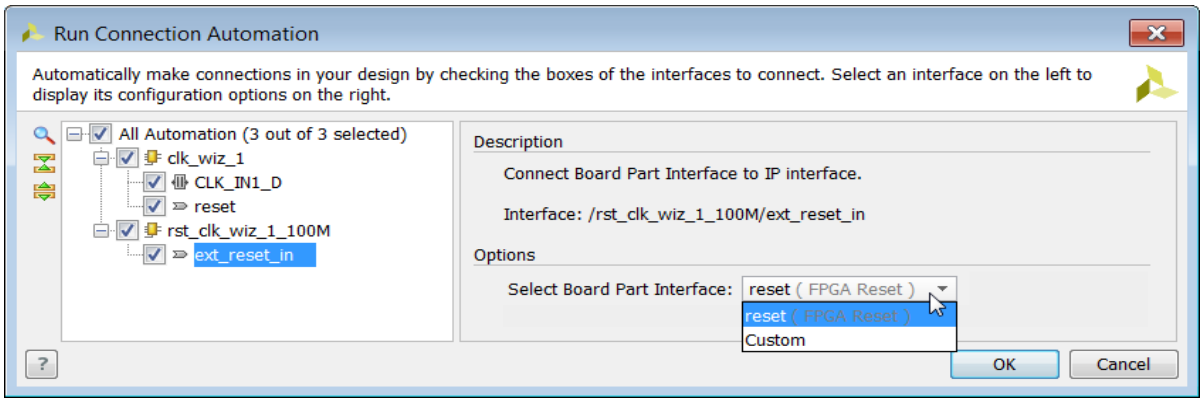


Figure 2-48: **Run Connection Automation**

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.

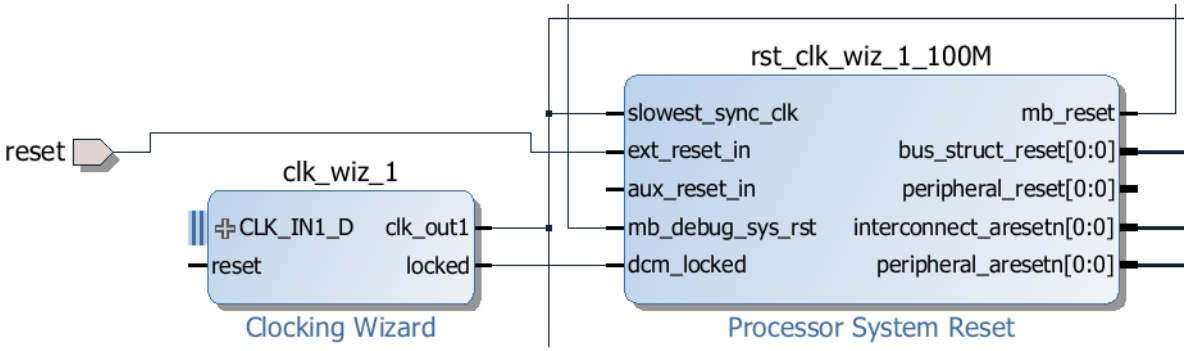


Figure 2-49: **Connecting the Reset Pin to the Board Reset Pin**

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.

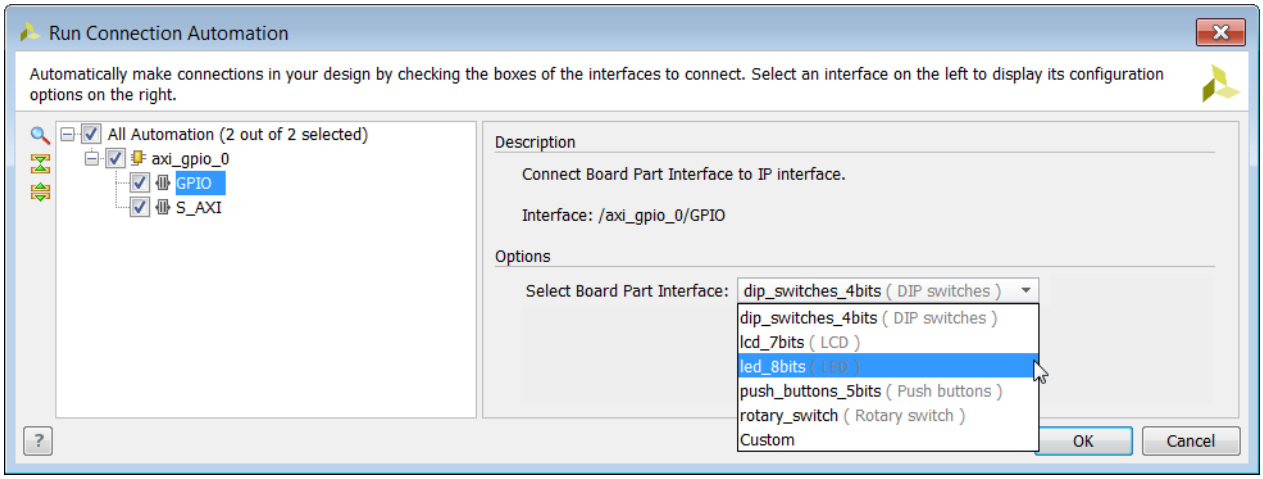


Figure 2-50: **Using Connection Automation to Show Potential Connections**

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

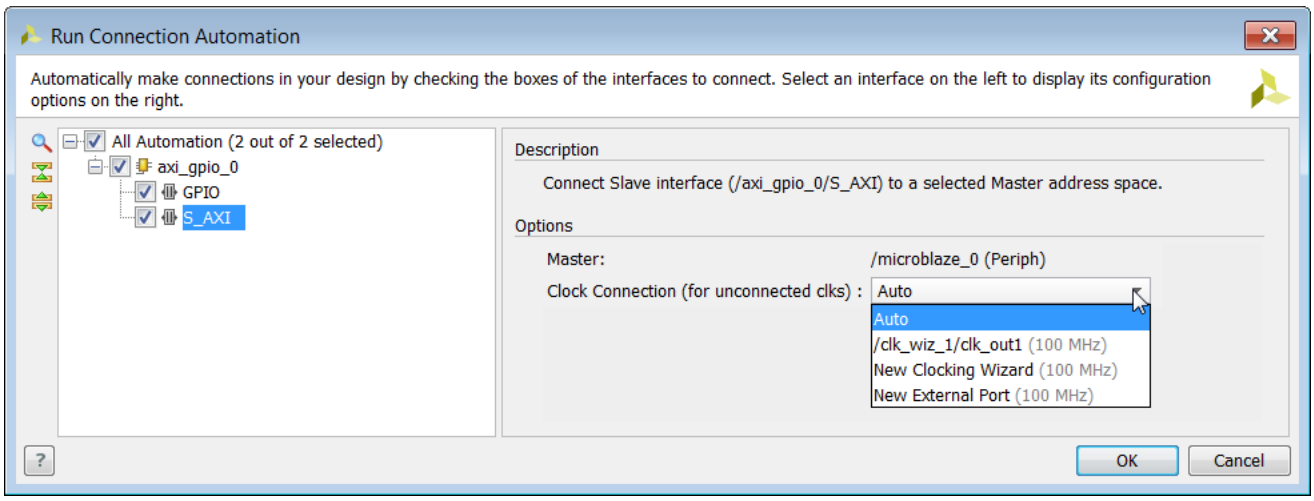


Figure 2-51: **Connecting the Slave Interface S_AXI to the MicroBlaze Master**

When you click the **OK** in the Run Connection Automation dialog box, the connections are made and highlighted as shown in Figure 2-52, page 41.

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 Hardware Design* (UG898) [Ref 5] for more information on this feature.

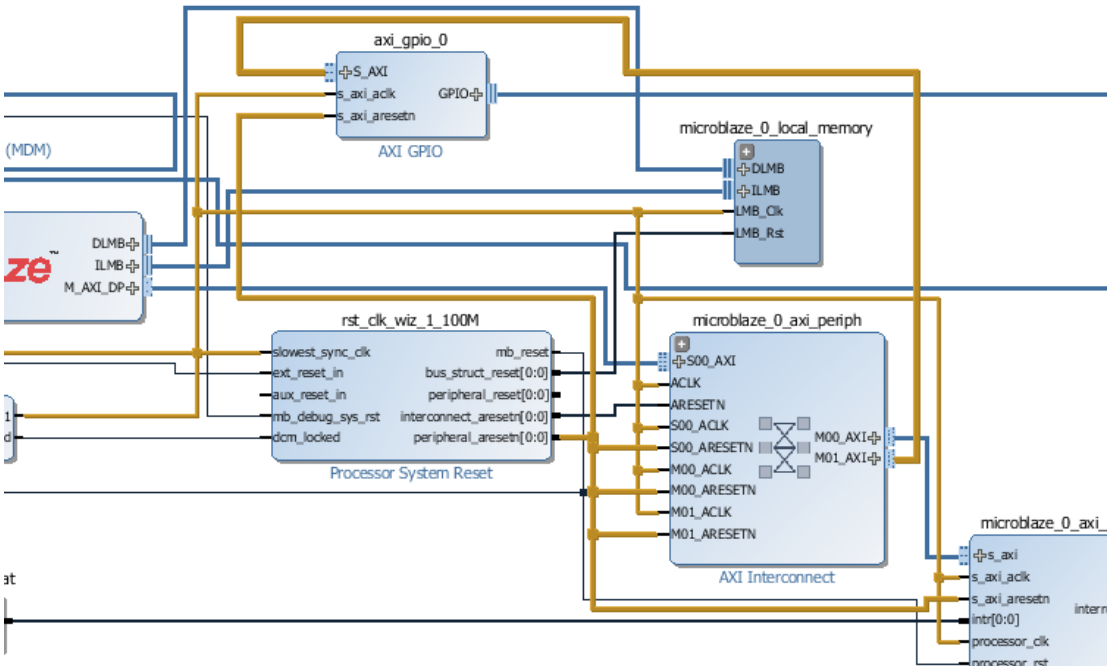


Figure 2-52: **Master/Slave Connections**

Using the Signals View to Make Connections

After a block design is open, the Signals window displays, as shown in Figure 2-53, page 42, 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 Figure 2-53, page 42, 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 block design. Right-clicking the selected clock presents you with several options.

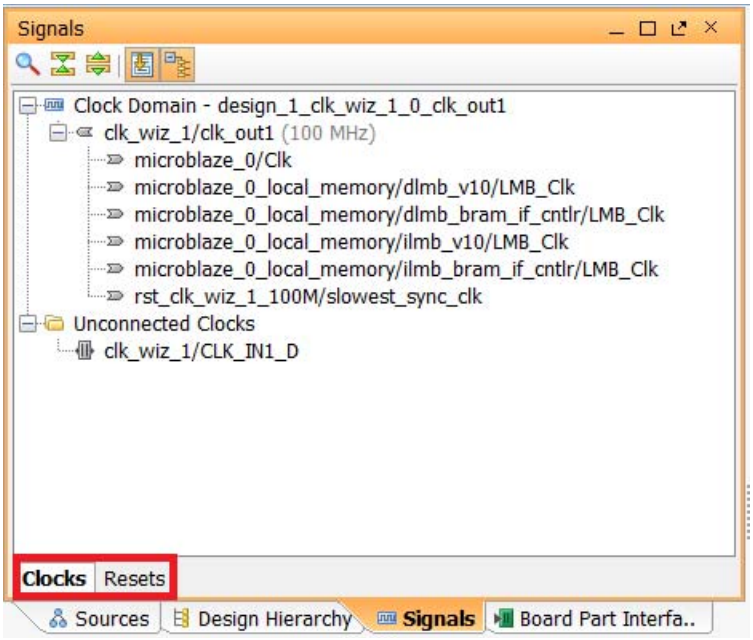


Figure 2-53: **Signals Window**

In the case shown in Figure 2-47, page 38, the Designer Assistance is in the form of the **Run Connection Automation** command which can be used 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.

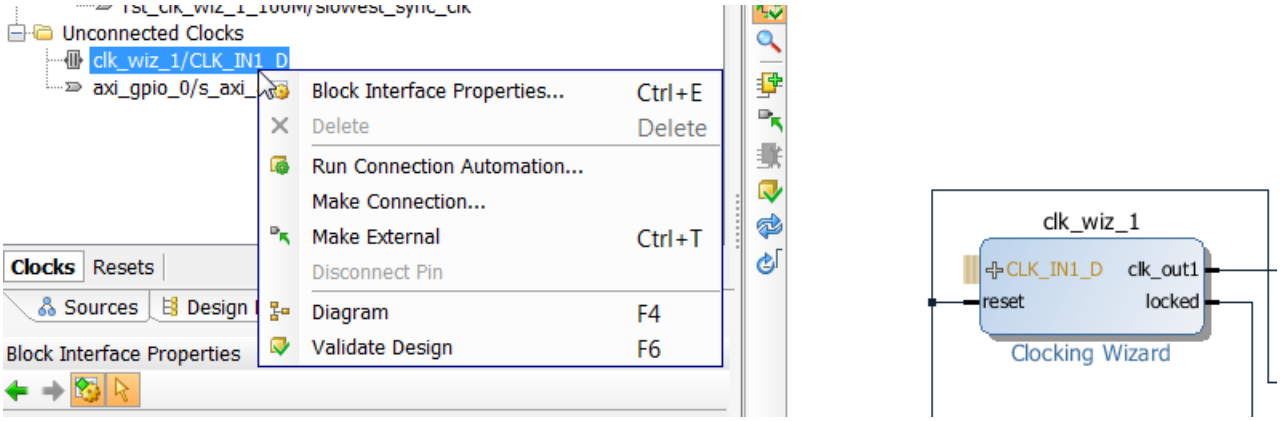


Figure 2-54: **Making Connection using the Signals Window**

When you select **Make Connection**, a dialog box opens, as shown in the following figure, if a valid connection can be made.

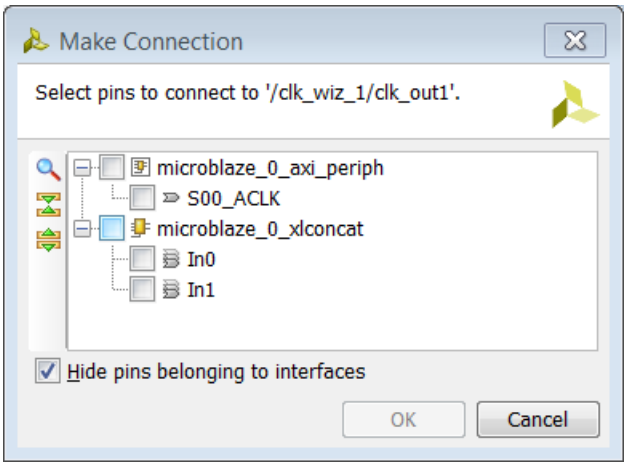


Figure 2-55: **Make Connection Dialog Box**

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 block design, they will be listed 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.

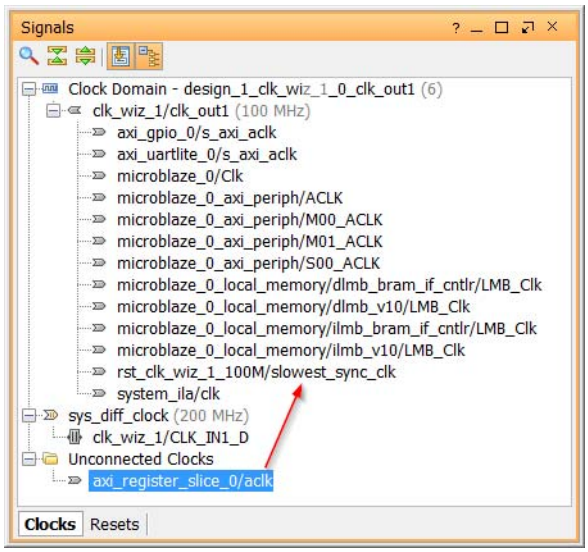


Figure 2-56: **Make Connection Dialog Box**

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.

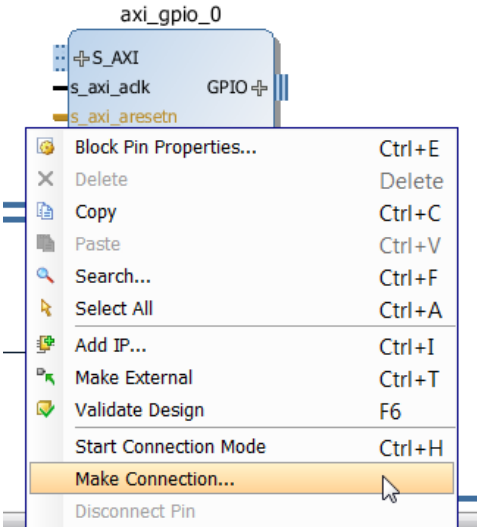


Figure 2-57: **Make Connection Command**

If a valid connection to the selected pin exists, the **Make Connection** dialog box opens that shows 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/pin can be selected.

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.

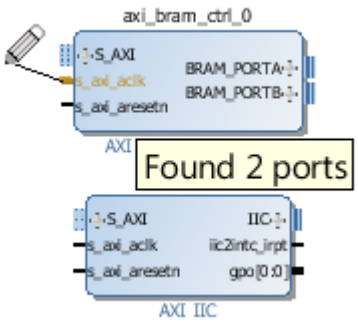


Figure 2-58: **Start Connection Mode**

After the connection is made to the `s_axi_aclk` pin of the AXI BRAM Controller in Figure 2-58, page 44, 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 block design, connecting to AXI slaves inside the design. These external masters are typically connected to the block design using an AXI Interconnect. Once 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 block design shown in the following figure.

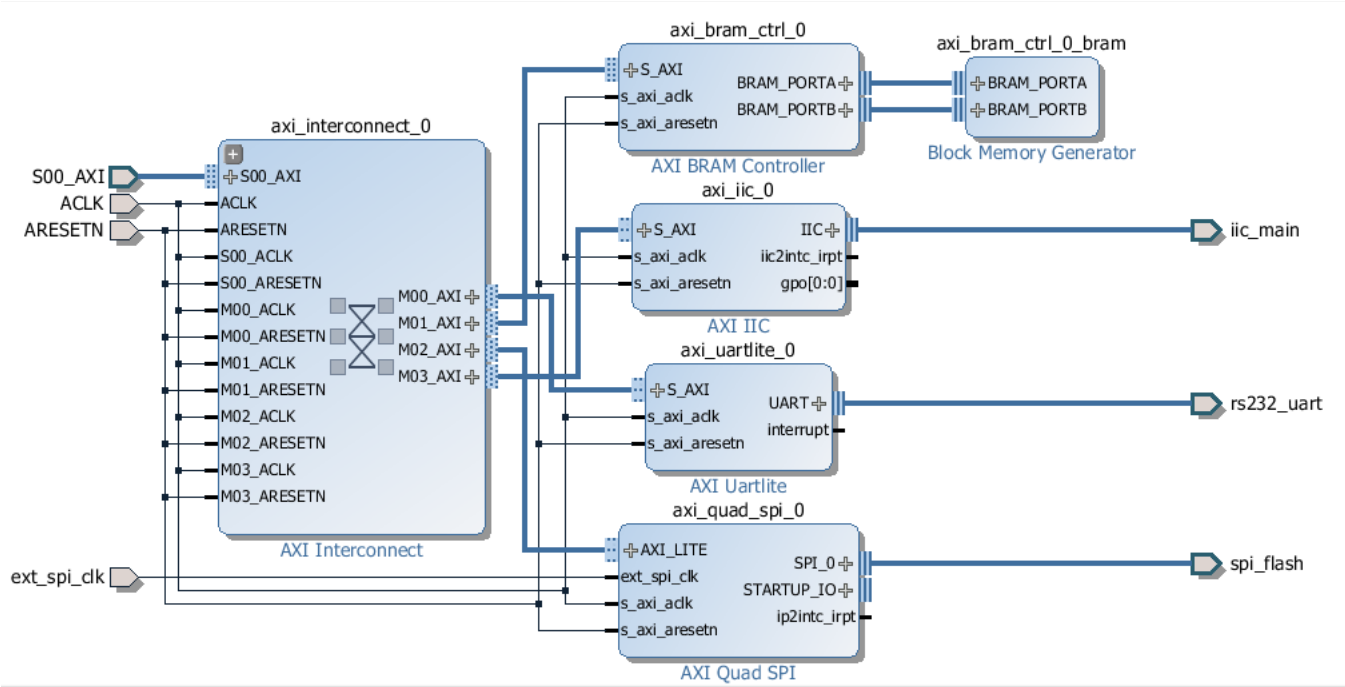


Figure 2-59: **Example Design with External AXI Master Interfacing with Block Design**

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 block design can be done in the normal manner.

Re-arranging the Design Canvas

IP blocks placed on the canvas can be re-arranged to get a better layout of the block design, 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 on 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 block design select the **Show interface connections only** button from the block design toolbar. This shows only the interface level connections, and hides all the other connections on the block design.

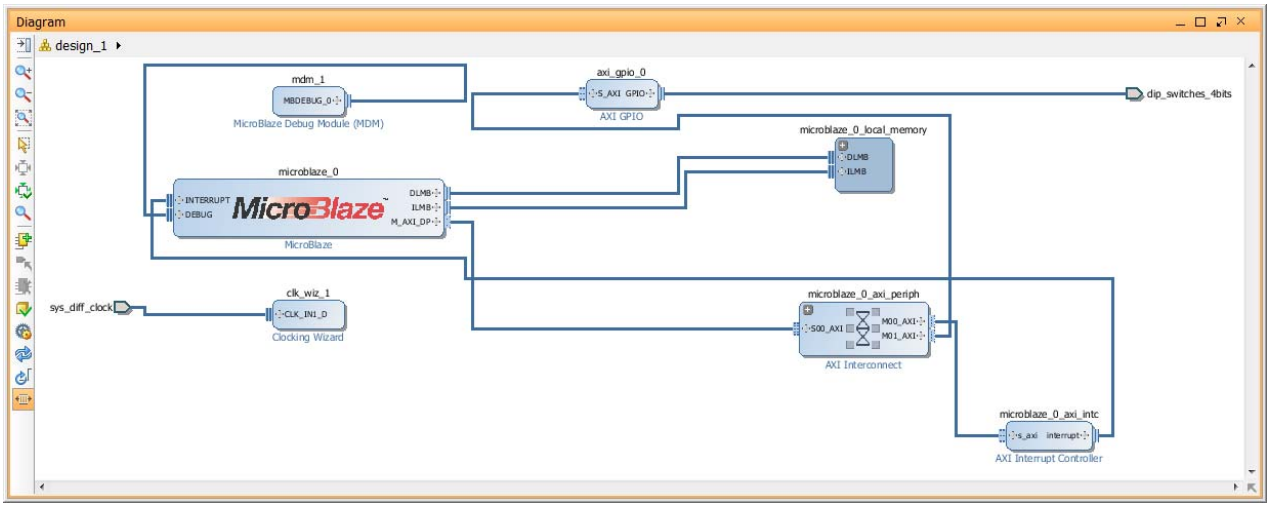


Figure 2-60: **Interface Connections Only**

Clicking the **Show interface connections only** button again restores all the connections in the block design.

Creating Hierarchies

As shown in the following figure, 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**. IP Integrator creates a new level of hierarchy containing the selected blocks.

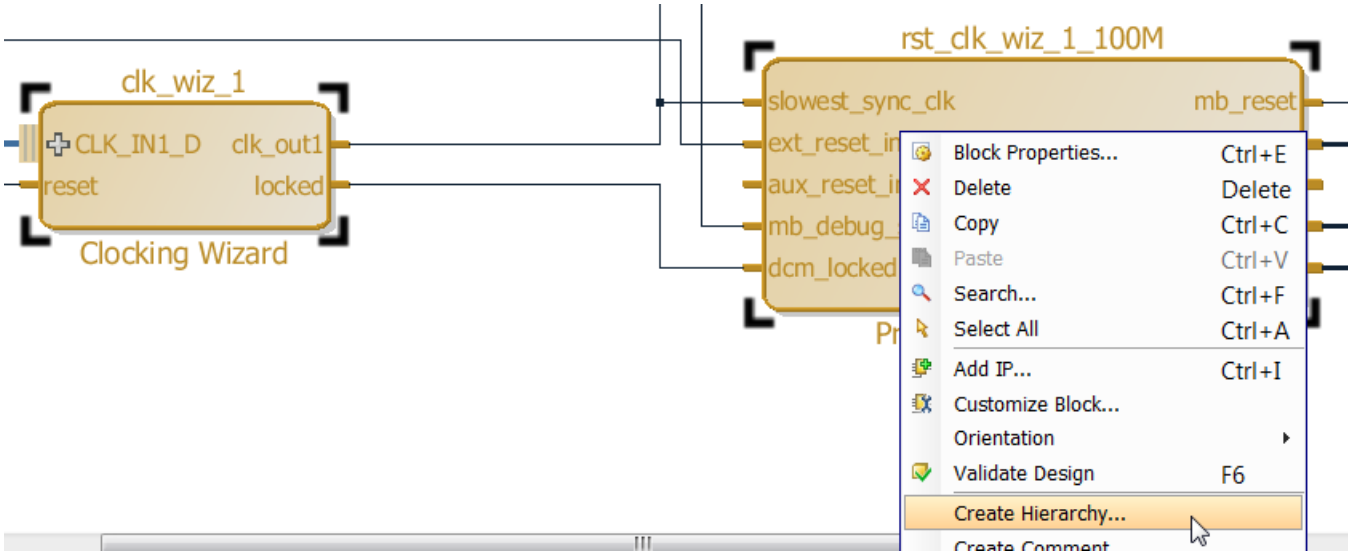


Figure 2-61: **Create Hierarchical Block Design**

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.

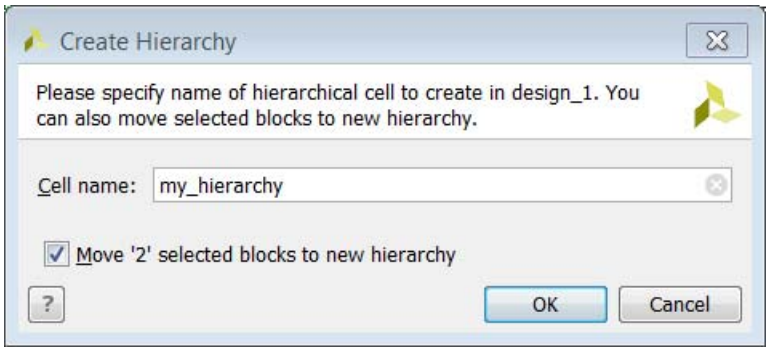


Figure 2-62: **Create Hierarchy Dialog Box**

This action groups the selected IP blocks under one block, as shown below.

- 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.

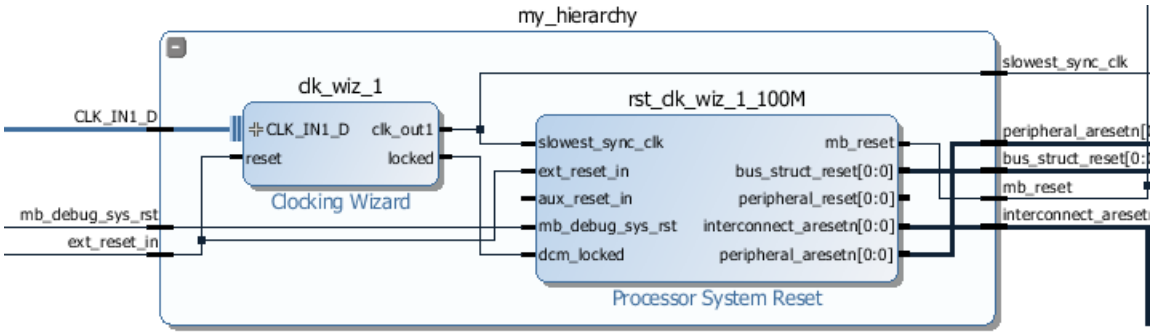


Figure 2-63: **Grouping Blocks Together**

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 on 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. Once the empty hierarchy has been created, the block design should look like the following figure.

Figure 2-64: **Empty Hierarchy**

You can add pins to this hierarchy by typing the following command on 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 block design instance to the appropriate hierarchy where the interface is to be added, using the following command:

```
current_bd_instance /hier_0
```

Next, create the interface using command as specified below:

```
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.

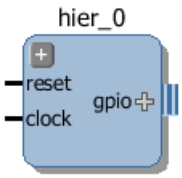


Figure 2-65: **Create Pins**

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.

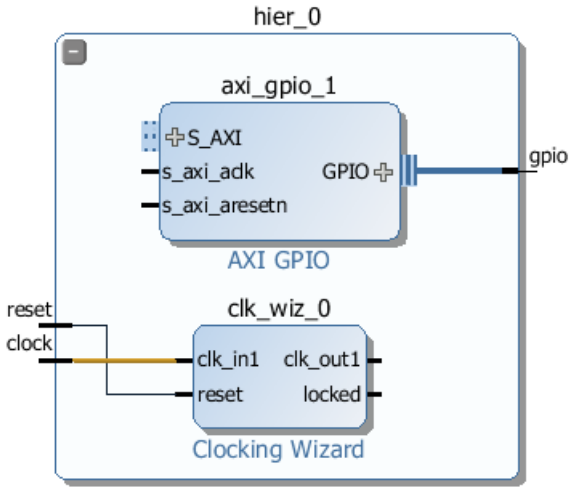


Figure 2-66: **Connected IP to Hierarchical Pin Interface**

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 **Validate Design**.

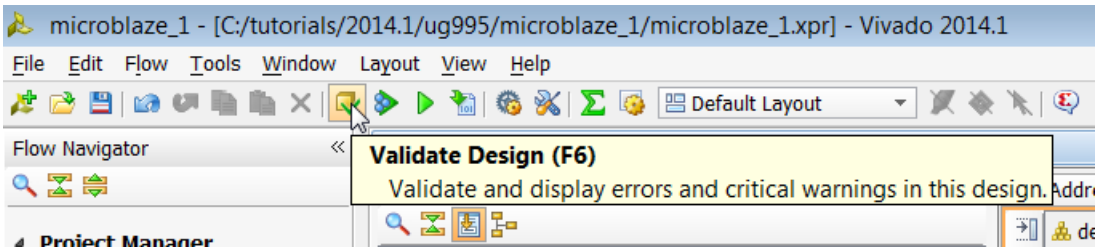


Figure 2-67: **Validate Design**

You can also click 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.

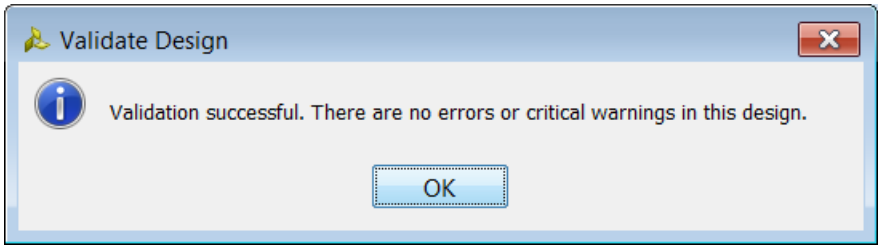


Figure 2-68: **Validation Successful Message**

Connecting Ports with Different Widths

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

As can be seen in Figure 2-69, page 51, the `bram_addr_a` pin of the AXI BRAM controller which is 14-bits wide is connected to the `addra` pin of the Block Memory Generator which 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:

```
[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.
```

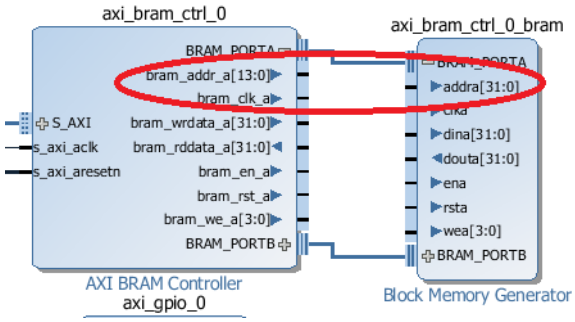


Figure 2-69: **Connecting pins of differing widths**

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.

Packaging a Block Design

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

For more information on packaging a block design for use in the Vivado IP Catalog, see this link in the *Vivado Design Suite User Guide: Creating and Packaging Custom IP* (UG1118) [Ref 11].

Creating a Memory-Map

Introduction

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, a master bd_address_seg object is created, 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 only appears 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 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 a master's overall memory space. 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 in the Vivado IP Integrator tool is used to 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.

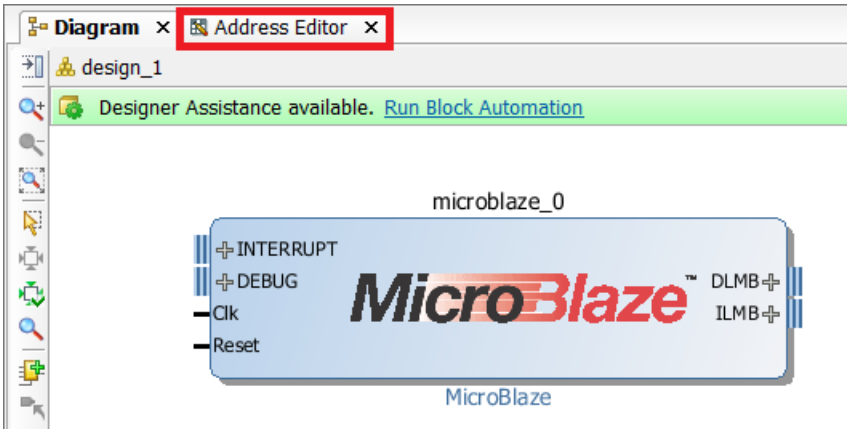


Figure 3-1: **Address Editor window**

As the peripherals are instantiated and connected to the processor in the block design canvas using connection automation, the IP Integrator automatically enters a corresponding memory assignment that peripheral in the Address Editor, as shown in Figure 3-2, page 54.

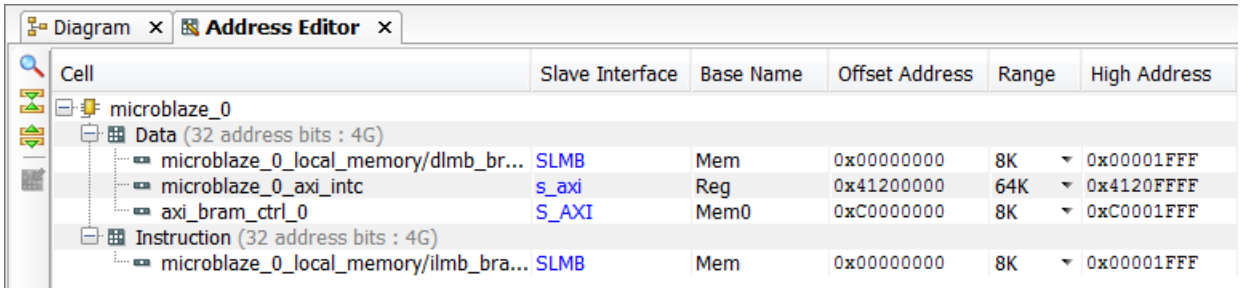


Figure 3-2: **Peripheral Memory-Map in Address Editor**

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, or by clicking the **+** sign to expand the selection.

  As shown in Figure 3-2, 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 `microblaze_0_local_memory/dlmb_bram_if_cntlr` and `microblaze_0_local_memory/ilmb_bram_if_cntlr` each have an interface called **SLMB** as shown in Figure 3-3, page 55.

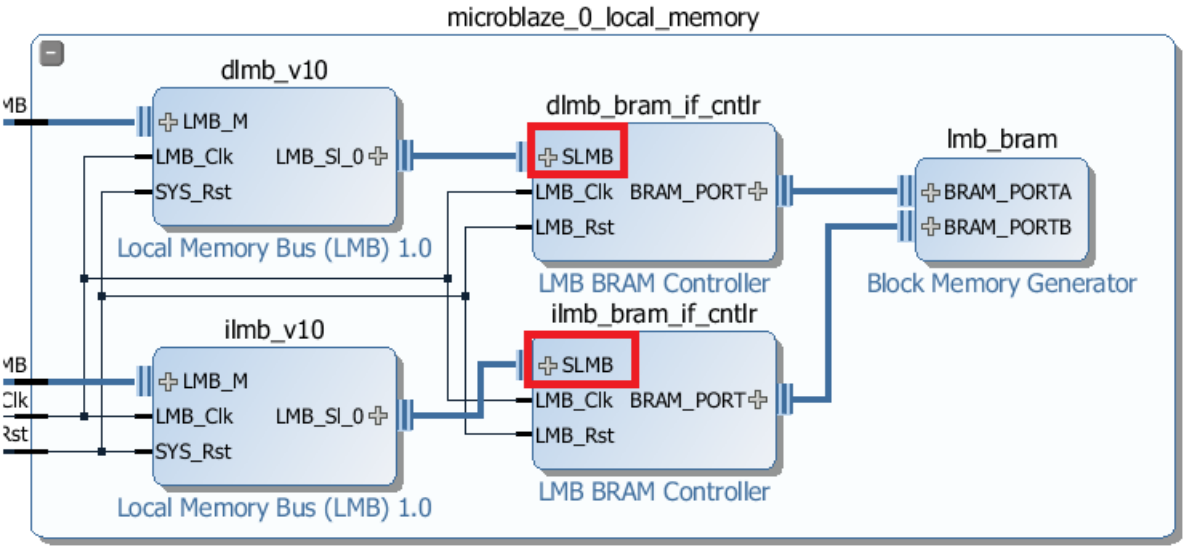


Figure 3-3: **Interface Names listed in the Slave Interface column**

- **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 Figure 3-5, which shows a design with multiple memory instantiations.

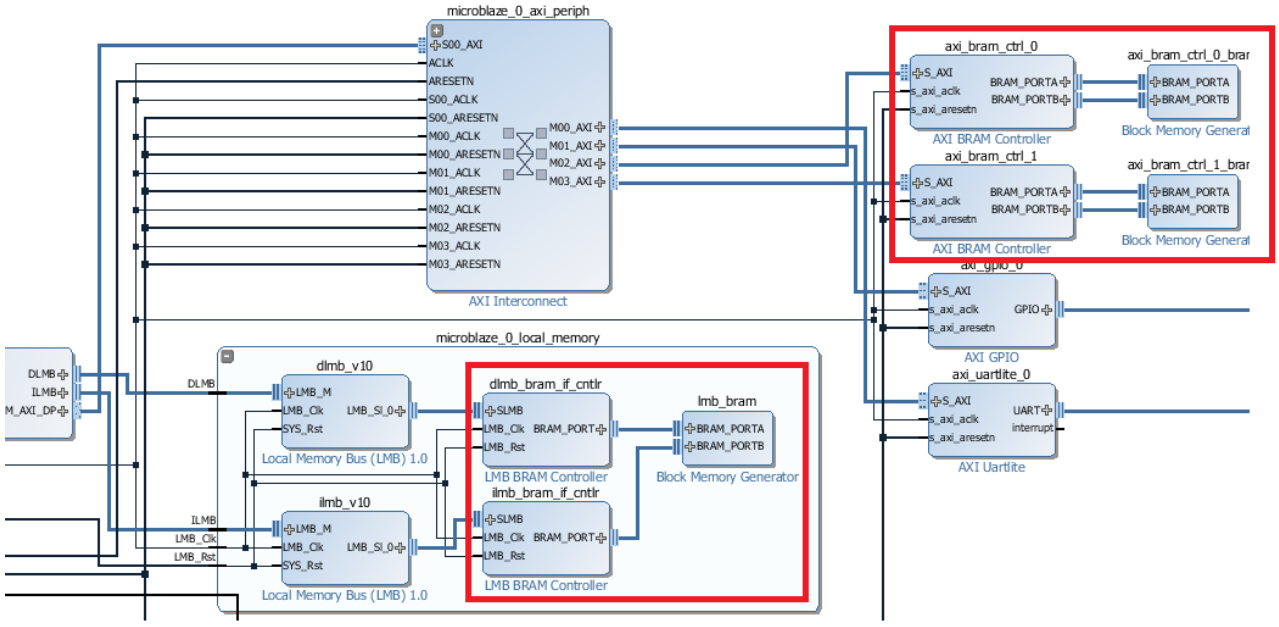


Figure 3-4: **Multiple Memory Instantiations in a block design**

These are given the base names in the address editor as shown in Figure 3-5, page 56.

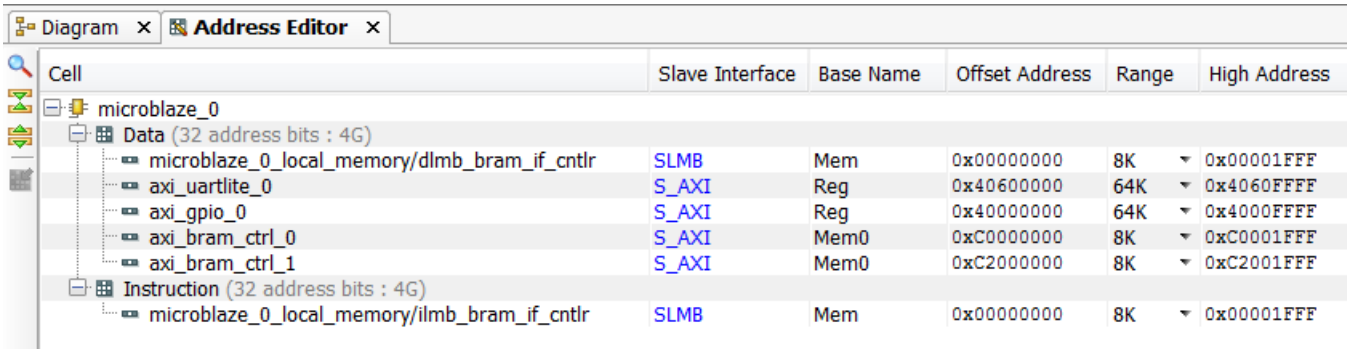


Figure 3-5: **Base Names given to multiple memory instantiations**

- **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 Figure 3-5. 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 also be changed by the user.

  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 Figure 3-6 displays.

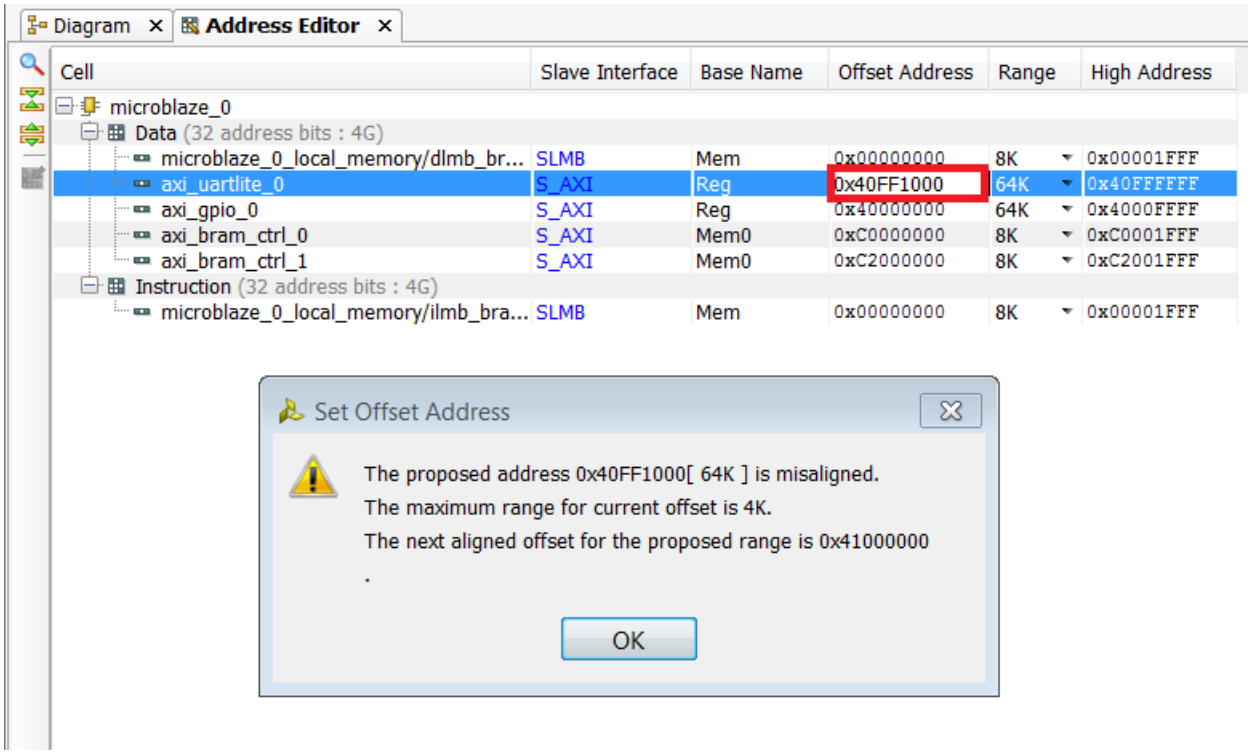


Figure 3-6: **Example of Misaligned Offset Address**

In Figure 3-6, page 57, the user set an offset address 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, the message pops up informing the user that the address is misaligned. The message also tells the user 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. Setting the Range field such that it exceeds this number can cause the message, shown in the following figure, to display.

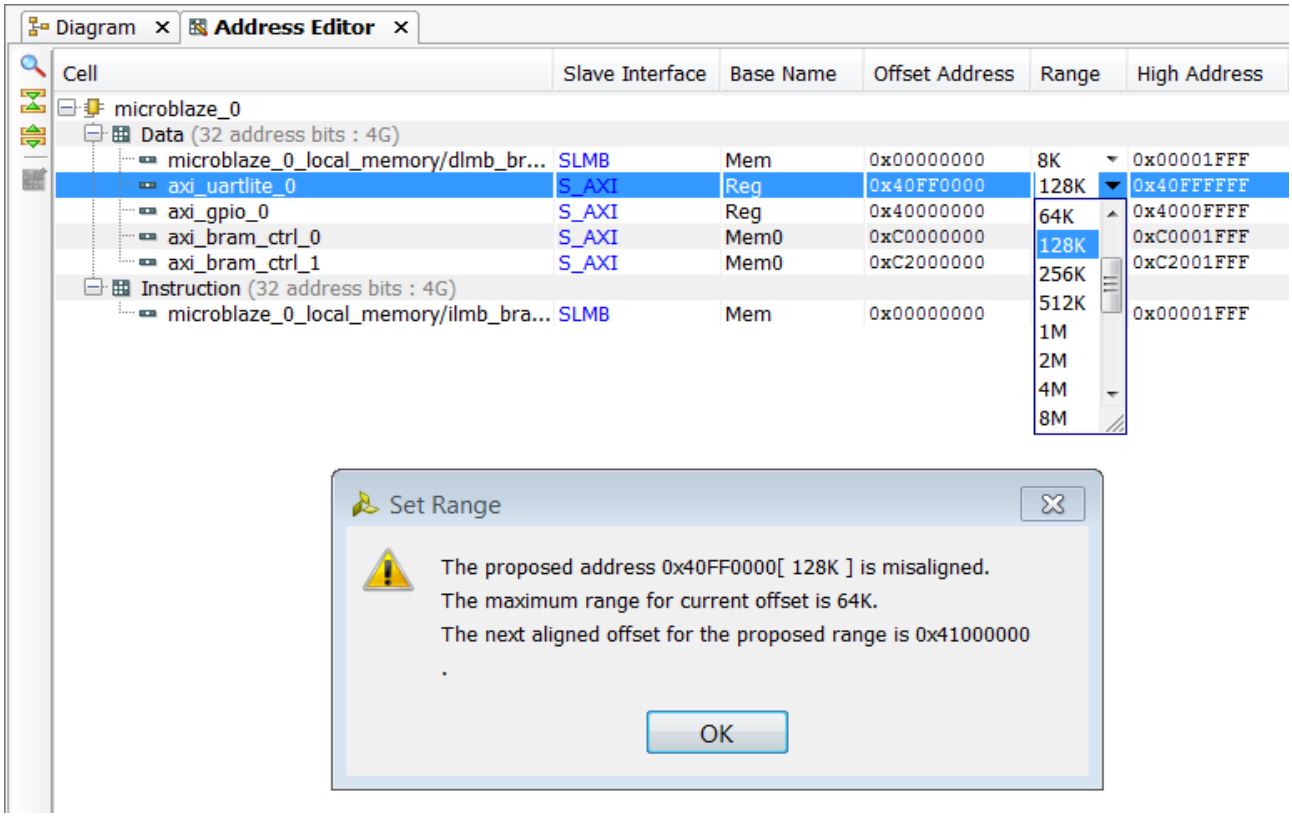


Figure 3-7: **Example of not setting the range field properly**

In Figure 3-7, the user tried to set the range to 128K or 2^{17}, for an offset in the form `0xXXXX0000`, which is an offset with only 16 least significant bits (LSBs). To accommodate a range of 128K, the form of the address must be at least `0xXXX20000`, that is with at least 17-bits in the LSB of the starting offset.

- **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 block design, 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 block design tool bar as shown in the following figure.

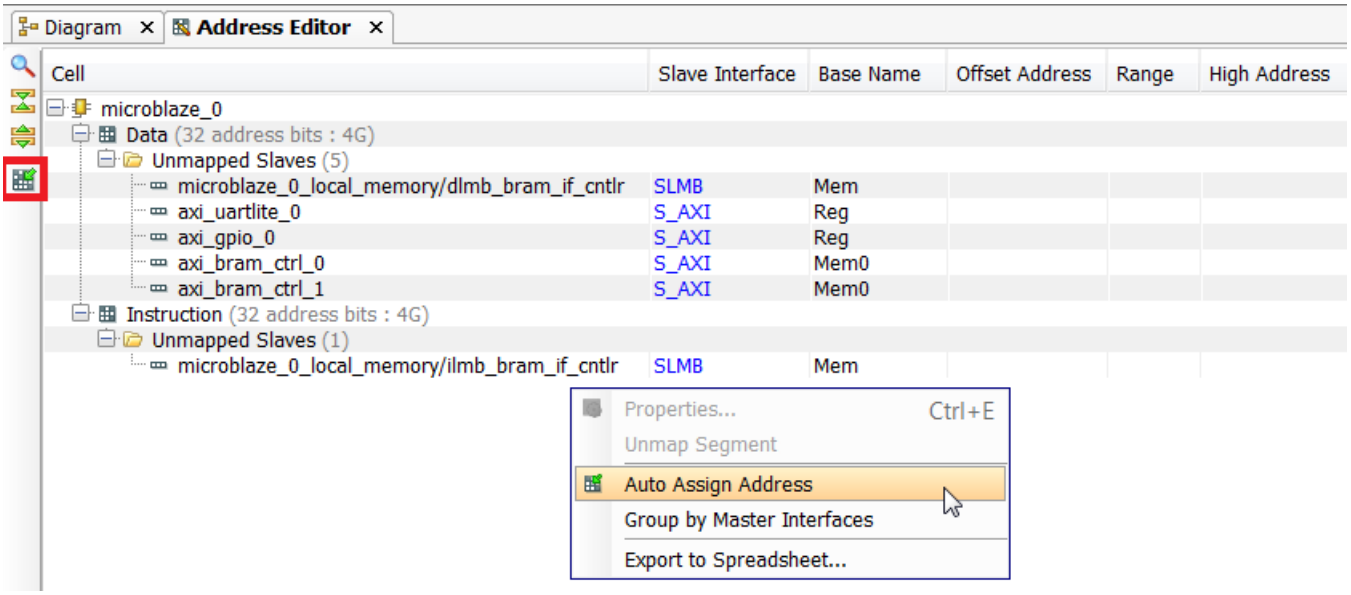


Figure 3-8: **Auto Assign Address Command**

This maps the slave segments as shown in Figure 3-9, page 59.

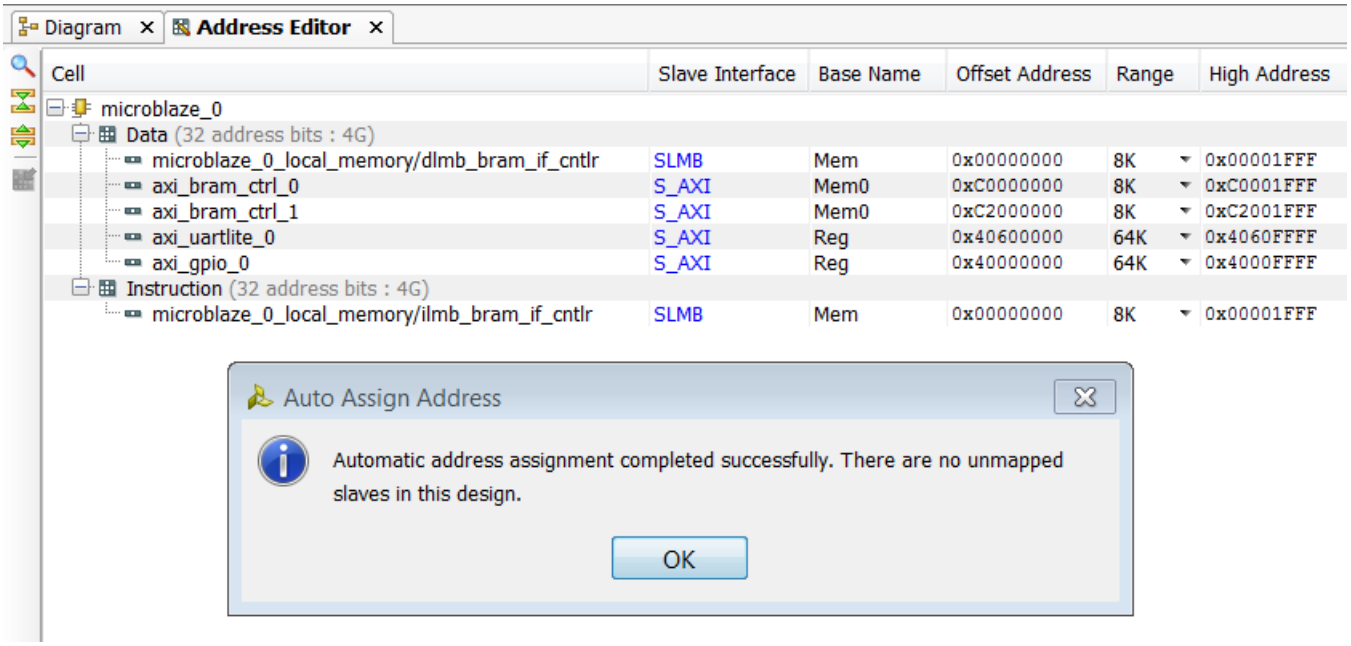


Figure 3-9: **Memory map after auto assignment of address blocks**

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

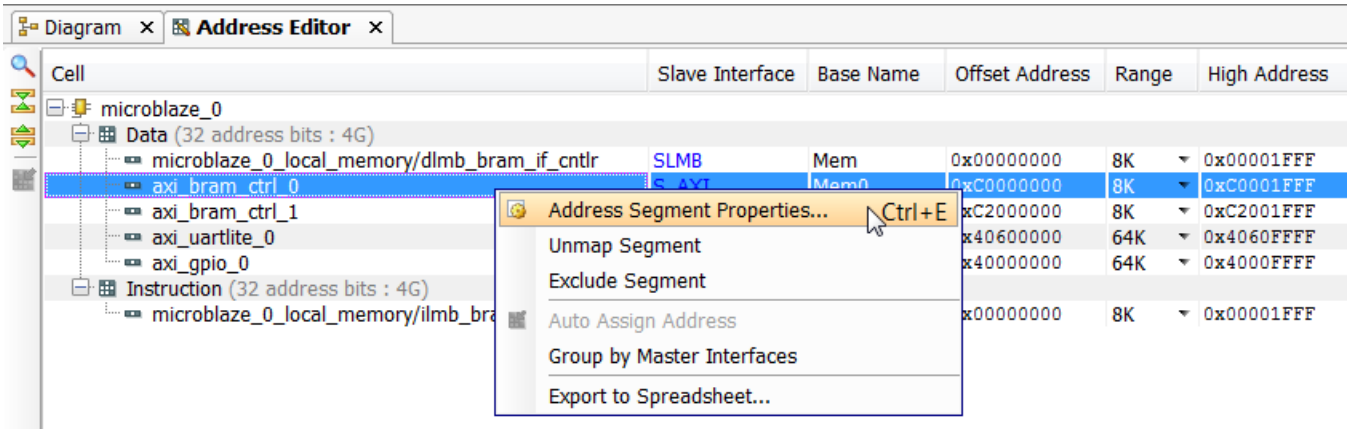


Figure 3-10: **Address Editor Options**

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.

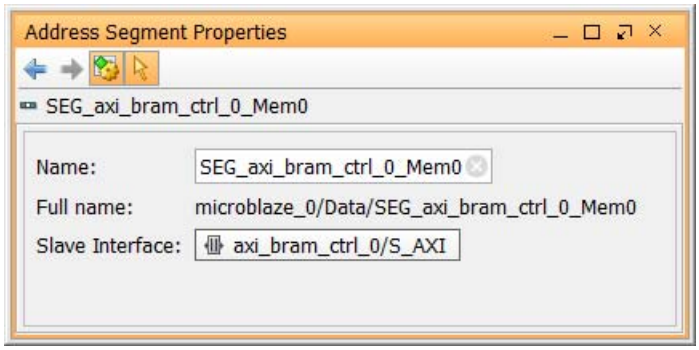


Figure 3-11: **Address Segment Properties Window**

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 changed by the user if desired.
- **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 Figure 3-12, page 61. The user 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).

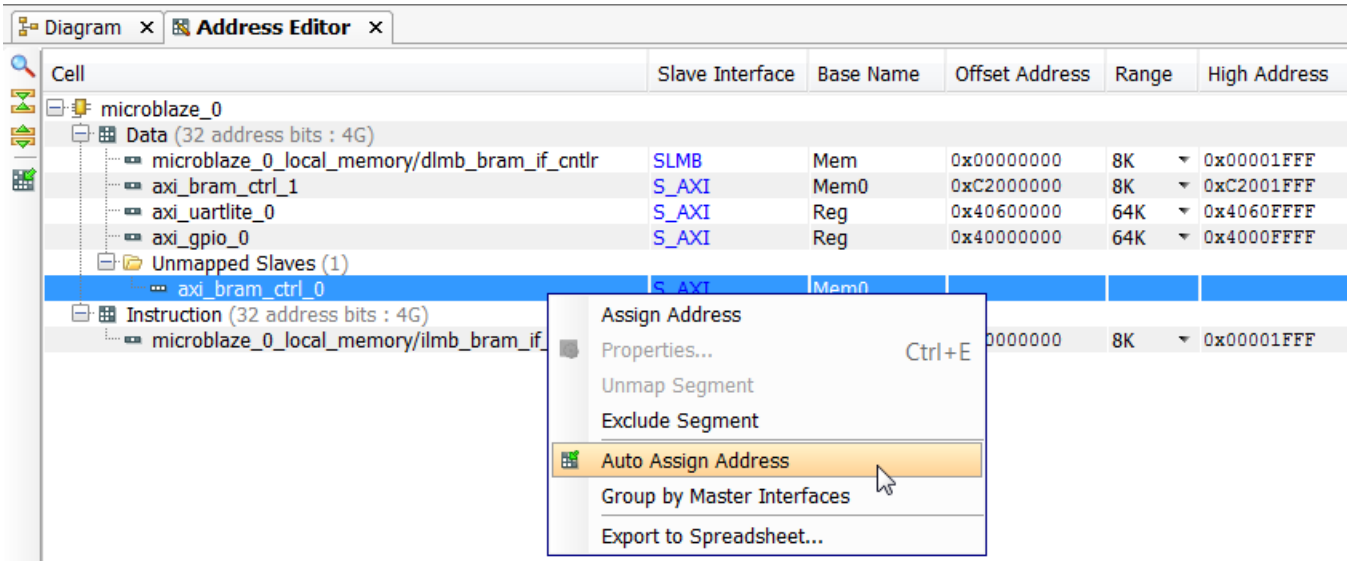


Figure 3-12: **Auto Assign Address Command**

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, page 62 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 in the following block design has three different master interfaces accessing various address segments: `DLMB`, `ILMB`, and `M_AXI_DP` within the Data Address Space.

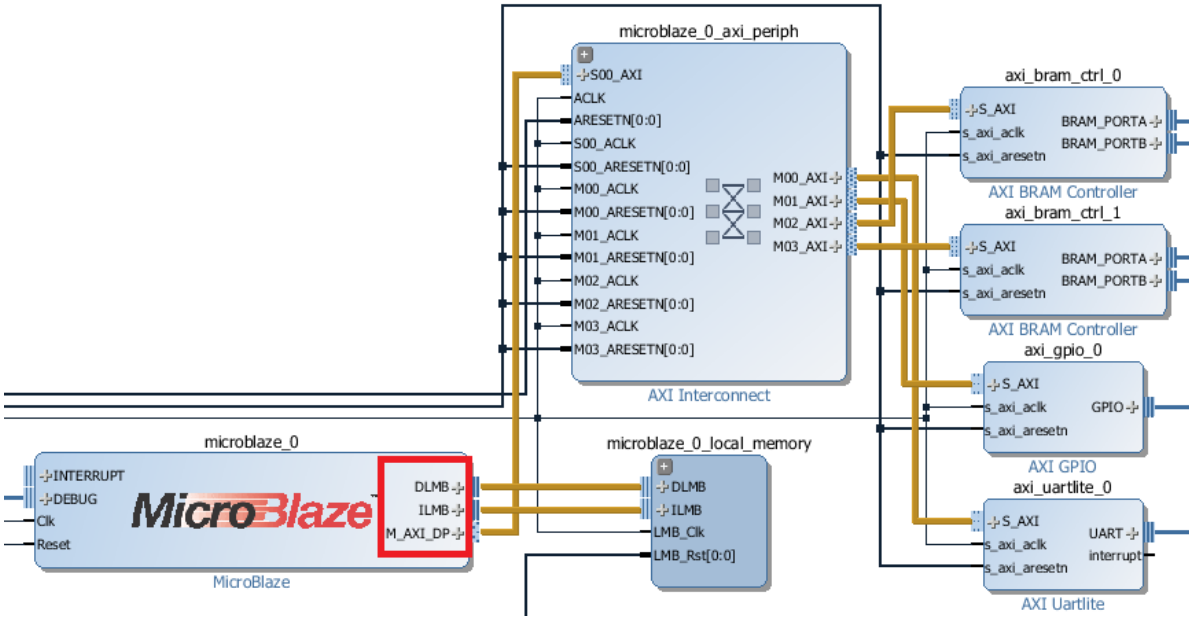


Figure 3-13: **MicroBlaze Block Grouped by Master Interfaces**

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

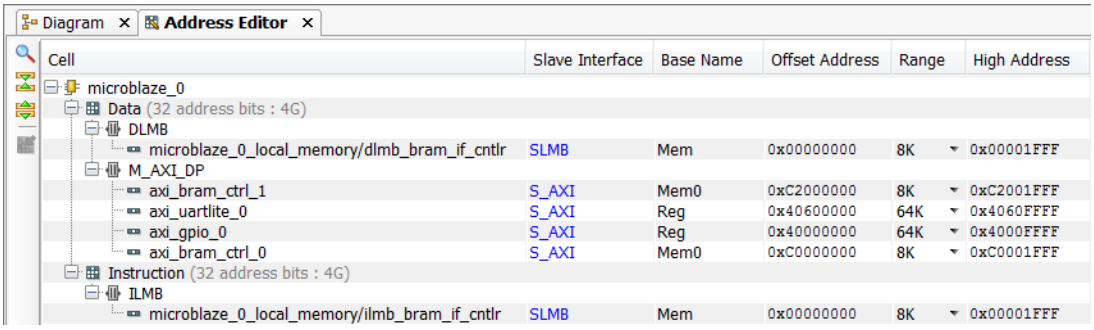


Figure 3-14: **Address Editor with Grouped Address Segments under Master Interfaces**

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 block design with two masters.

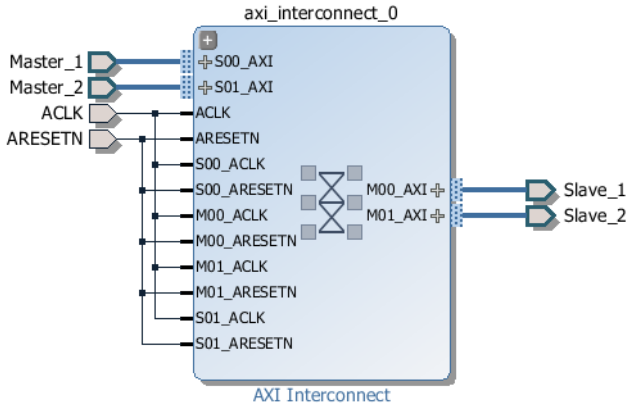


Figure 3-15: **Multiple Master and Slave Example**

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.

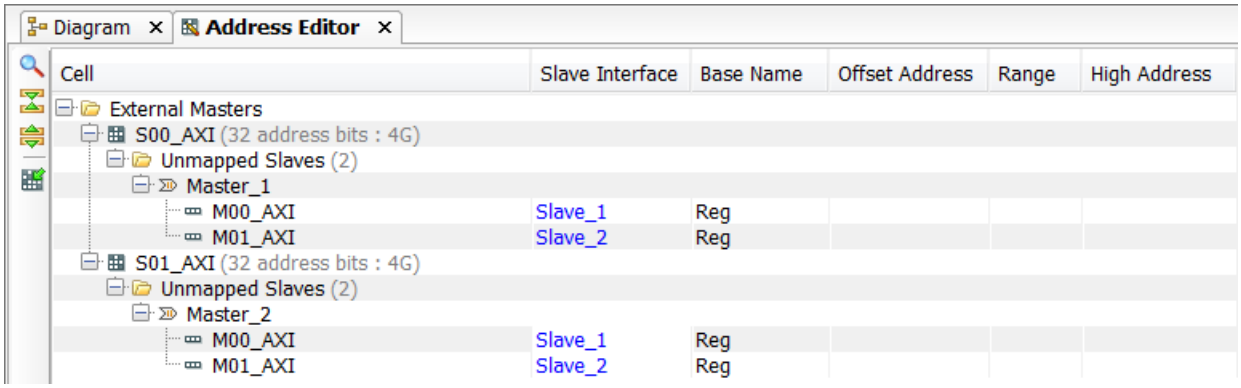


Figure 3-16: **Address Editor with Multiple Master Memory-Map**

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 Figure 3-17, page 64.

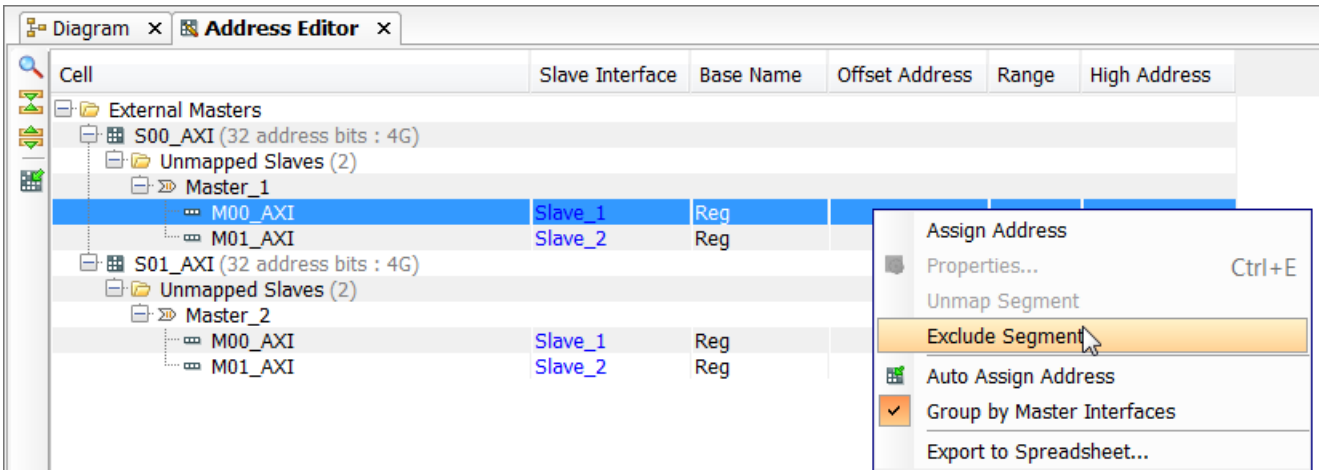


Figure 3-17: **Exclude Segment Command**

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

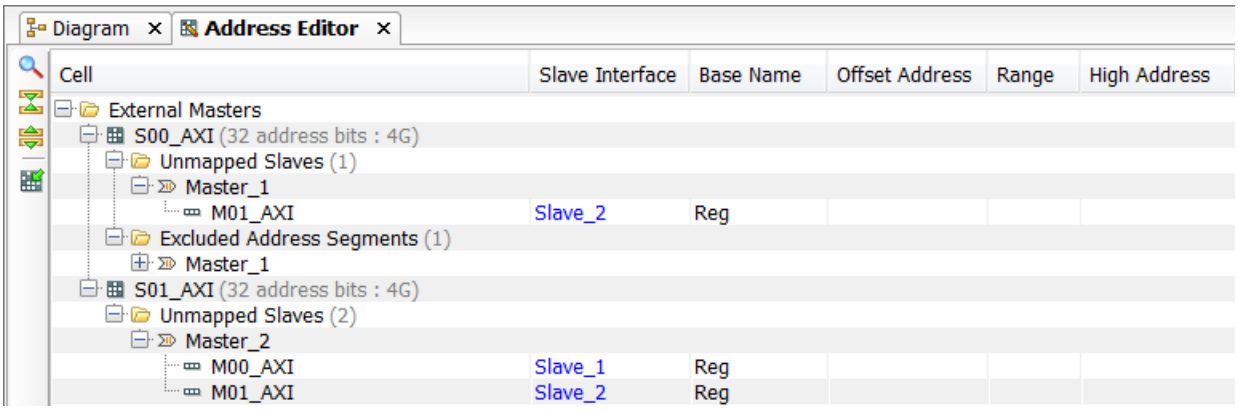


Figure 3-18: **Excluded Address Segment in Address Editor**

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.

Including an Address Segment

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

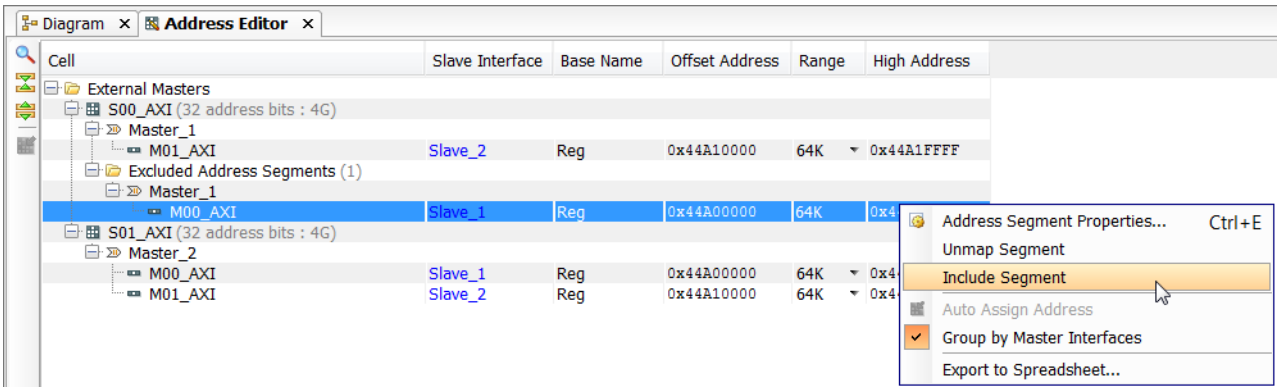


Figure 3-19: **Including an Excluded Segment into Master Memory-Map**

Common Addressing-Related Critical Warnings and Errors

```
[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 master's address space or excluded from it.

```
[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.

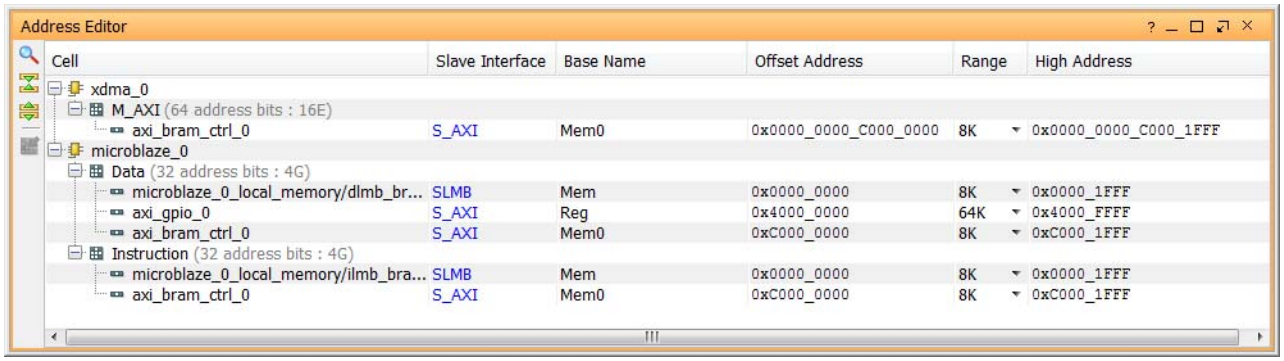


Figure 3-20: **Support for 64-bit Address Width**

Working with Block Designs

Overview

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

Generating Output Products

After the block design is complete and the design is validated, you must generate output products for synthesis and simulation, in order to integrate the block design 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 block design based upon the Target Language that you specified during project creation, or in the Project 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.

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

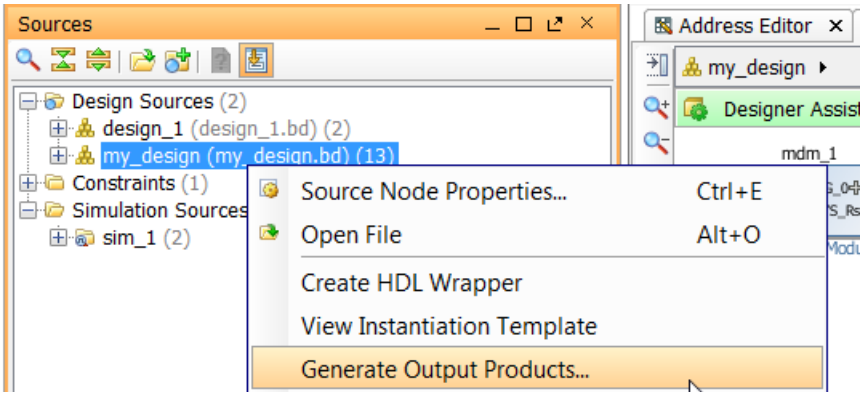


Figure 4-1: **Generate Output Products Command**

Alternatively, from the Flow Navigator, click **IP Integrator > Generate Block Design**, as shown in Figure 4-2.

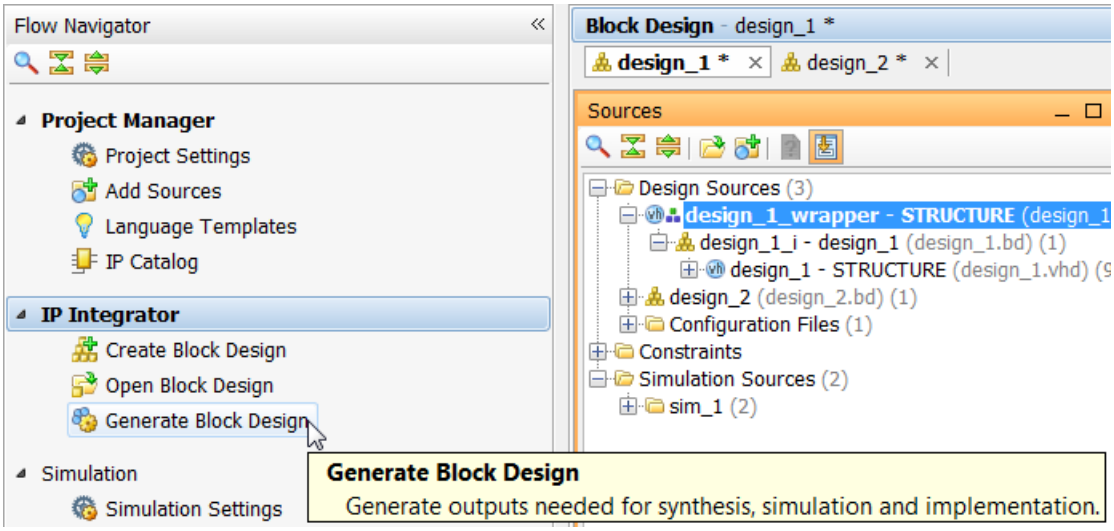


Figure 4-2: **Generate Block Design Command**

Output product can also be generated from the IP Sources tab in the source window by selecting and right-clicking on the block design and selecting Generate Output Products from the context menu as shown in Figure 4-3.

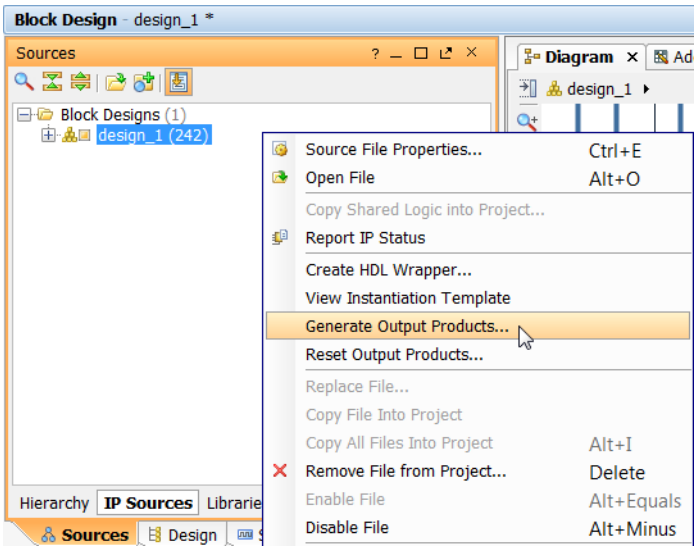


Figure 4-3: **Generate Output Products from the IP Sources tab in the Sources window**

Generating the output products generates the top-level netlist of the block design. The netlist is generated in either VHDL or Verilog based upon the Target Language settings in Project Settings.

Generate Output Products Dialog Box

Output products can be generated 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 block design, and simply synthesizes it with the whole design.
- **Out of context per IP**: Generates the output product for each individual IP used in the block design. A DCP is created for every IP used in the block design. 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 have not been changed. For more information on using the IP Cache, refer to this link in the *Vivado Design Suite User Guide: Designing with IP* (UG896) [Ref 4].
- **Out of context per Block Design**: This lets you synthesize the complete block design separately from, or out of the context of the top-level design. This option is generally selected when third party synthesis is used. A design checkpoint (DCP) is generated for the block design when this option is selected.

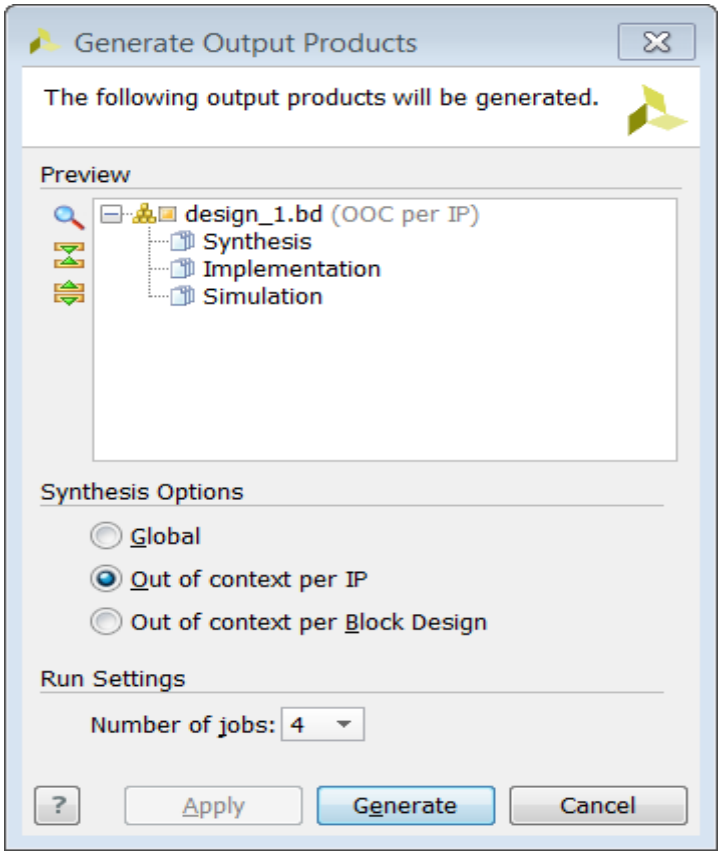


Figure 4-4: **Generate Output Products Dialog Box**

TIP: *The default mode of Synthesis is Out of context per IP, and IP caching is also enabled by default. This combination provides the greatest benefits for reducing synthesis demands.*

Global Synthesis

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

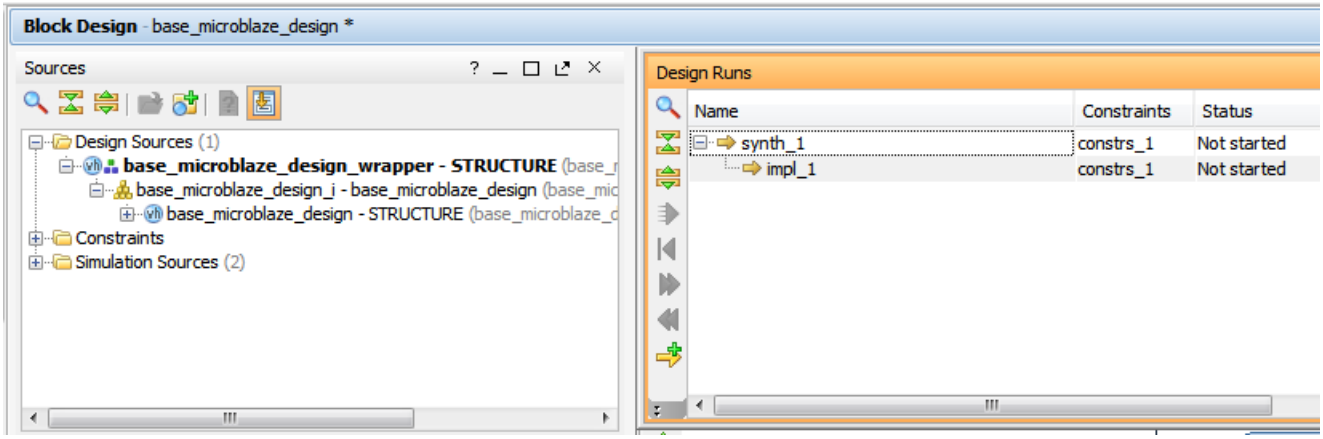


Figure 4-5: **Design Runs window for Global Synthesis**

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 block design 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 block design, as shown in Figure 4-6.

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, page 17.*

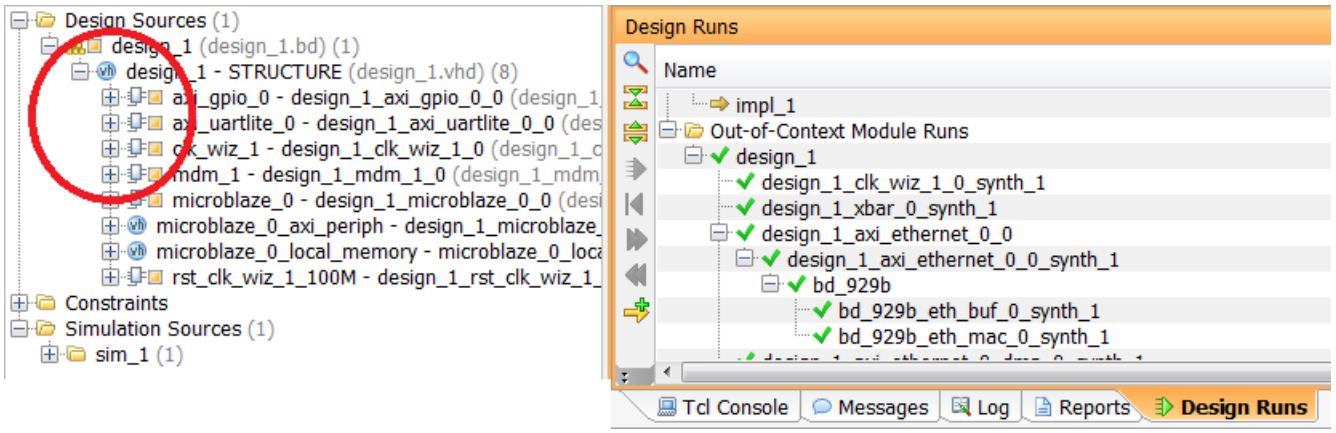


Figure 4-6: **Design Runs window for Out-of-Context per IP Synthesis**

Generation of the individual output products in OOC per IP mode takes a little longer than a single global synthesis run. However, runtime improvements are realized in subsequent runs as 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 runtime improvements as only IP that have been modified, either because of re-customization or because of an impact from parameter propagation, are re-synthesized.

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

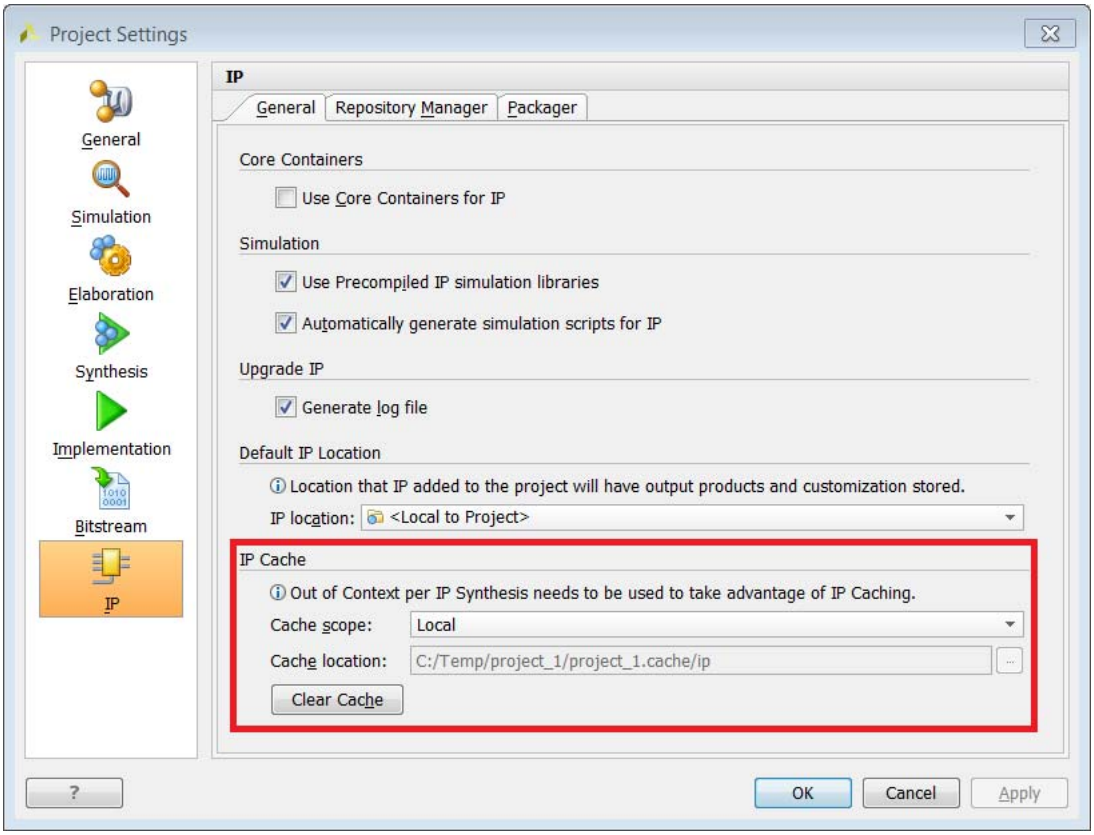


Figure 4-7: **Setting IP Cache in the Project Setting Dialog Box**

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**, a ***<project_name>*.cache** folder is created in the project folder, that holds the configuration data and synthesis results for all the IP in the block design. 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 block design as an OOC module, and creates a design checkpoint for the entire block design. As can be seen from the figure below, the Sources window shows that a Design Checkpoint file (DCP) was created for the block design. Notice that the block design 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 block design.

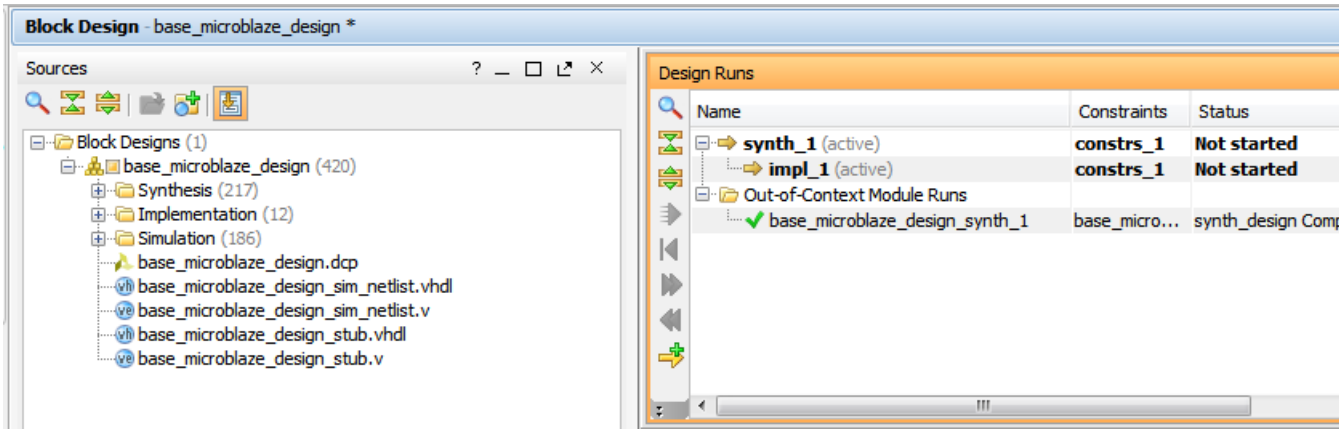


Figure 4-8: **Design Runs window for Out-of-Contextper Block Design Synthesis**

If the block design is added as a synthesized netlist to other projects through the Add Sources wizard, the DCP file is added to the project. Refer to this link in the *Vivado Design Suite User Guide: System-Level Design Entry* (UG895) [Ref 3] for more information on adding block designs as design sources.

Examining Generated Output Products

The generated output products for a block design can be found in the `<project_name>/<project_name>.srcs/sources_1/bd` folder. Inside the folder is a separate directory for each block design. In Figure 4-9, `config_mb_design` is the only block design.

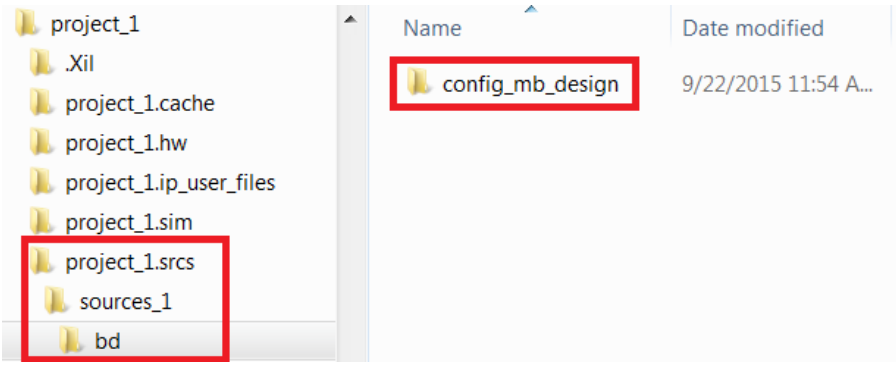


Figure 4-9: **Locating Output Products for Block Designs**

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

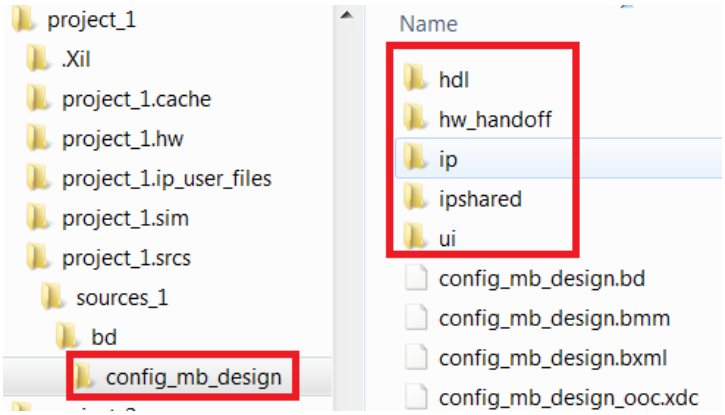


Figure 4-10: **Sub folders under the block design**

- `hdl`: This folder contains the top level netlist of the block design as well as the Vivado managed wrapper file for the block design.
- `hw_handoff`: This folder contains intermediate files needed for hardware handoff to SDK.
- `ip`: The `ip` folder contains several sub-folders, one per IP inside the block design. 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`: This folder 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 output products for the block design are generated, a folder called <*project_name*>/<*project_name*>`.ip_user_files` is created as shown below. Inside of the <*project_name*>`.ip_user_files` folder there are a number of folders depending on the contents of your project (IP, Block Designs, etc.).

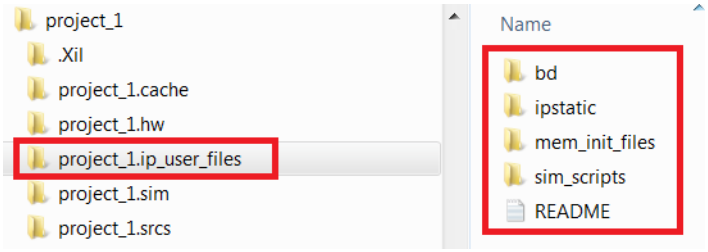


Figure 4-11: **Sub-folders Under the ip_user_files Folder**

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

- `bd`: Contains a sub-folder for each IP Integrator Block Design (BD) in the project. These sub-folders will have support files for the various IP used in the block designs.
- `ipstatic`: Contains common IP static files from all IP/BDs in the project.
- `mem_init_files`: This directory will be 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 Block Design present.

To manually export IP or block design 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 block design, this command will be automatically run. For more information refer to this link in the *Vivado Design Suite User Guide: Designing with IP* (UG896) [Ref 4].

When the Output Products for a block design 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 block design. The individual cache-ID folders can be found in the IP Cache location.

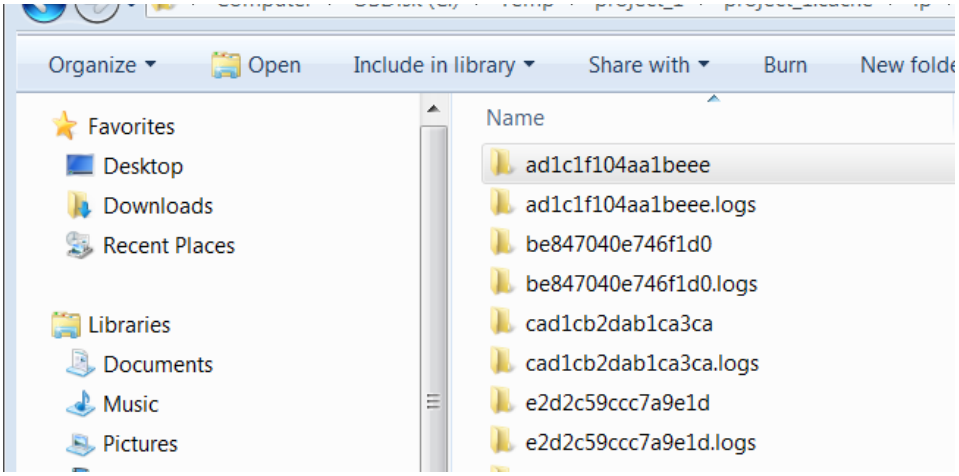


Figure 4-12: **Cache-ID Directories**

Integrating the Block Design into a Top-Level Design

An IP Integrator block design 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 block design. Right-click the block design in the Vivado IDE Sources window and select **Create HDL Wrapper**.

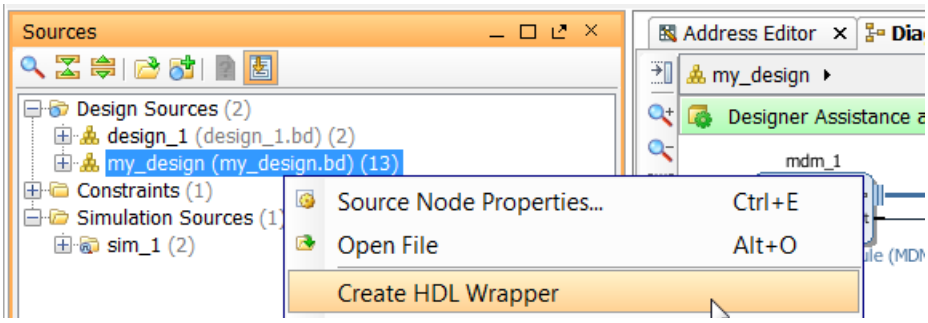


Figure 4-13: **Create HDL Wrapper Command**

This command generates a top-level HDL file with an instantiation template for the IP Integrator block design. The Create HDL Wrapper dialog box opens, as shown below.

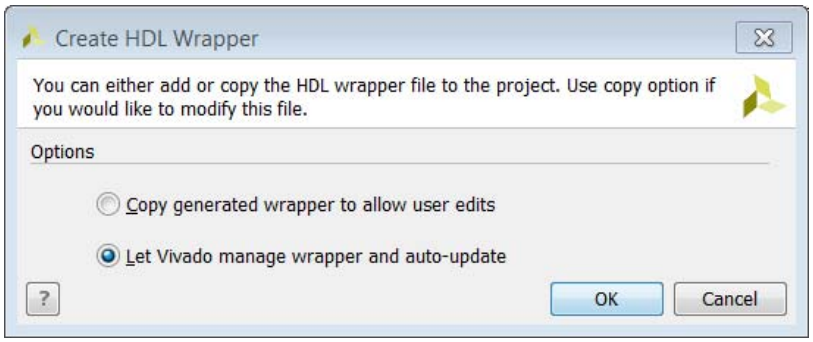


Figure 4-14: **Create HDL Wrapper Dialog Box**

The Create HDL Wrapper options are as follows:

- **Copy generated wrapper to allow user edits**. When a block design is a subset of an overall design hierarchy, you need to 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 block design 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 block design. 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 block design, 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. Refer to the *LogiCORE IP Utility Buffer* (PB043) [Ref 23] for more information.

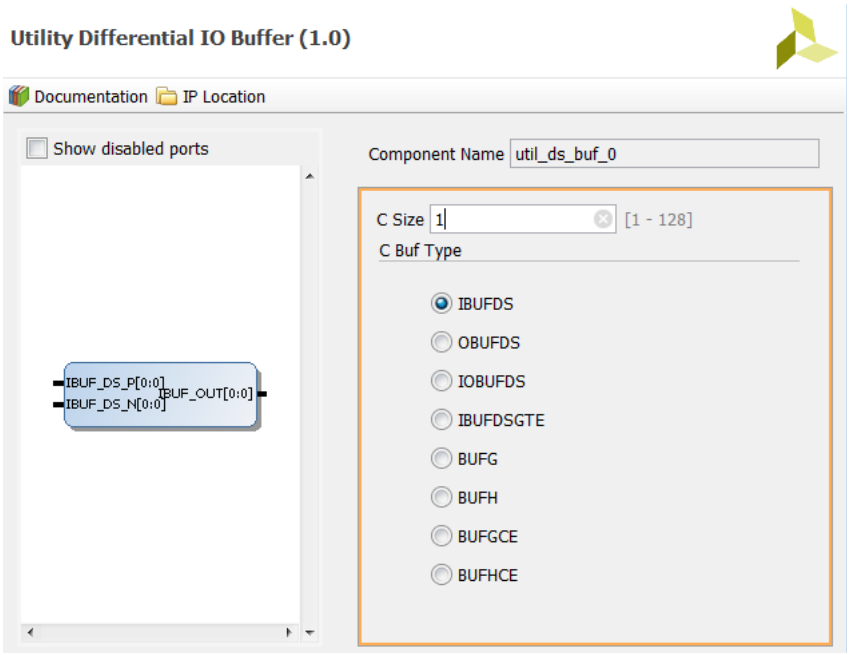


Figure 4-15: **Utility Buffer IP**

Adding Existing Block-Designs

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

Assuming that a block design was created using a project-based flow, and all the directory structure including and within the block design folder is available, the block design 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 block design 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.*

To add a remote block design:

1. Select **Flow Navigator > 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**.

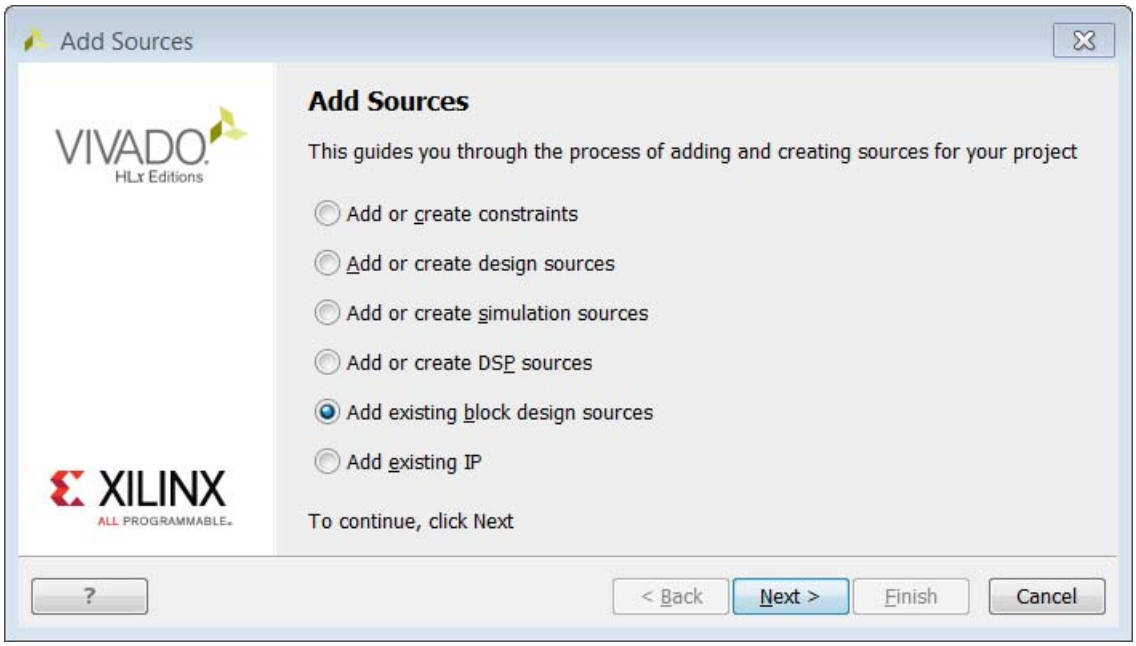


Figure 4-16: **Add Sources Wizard**

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

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

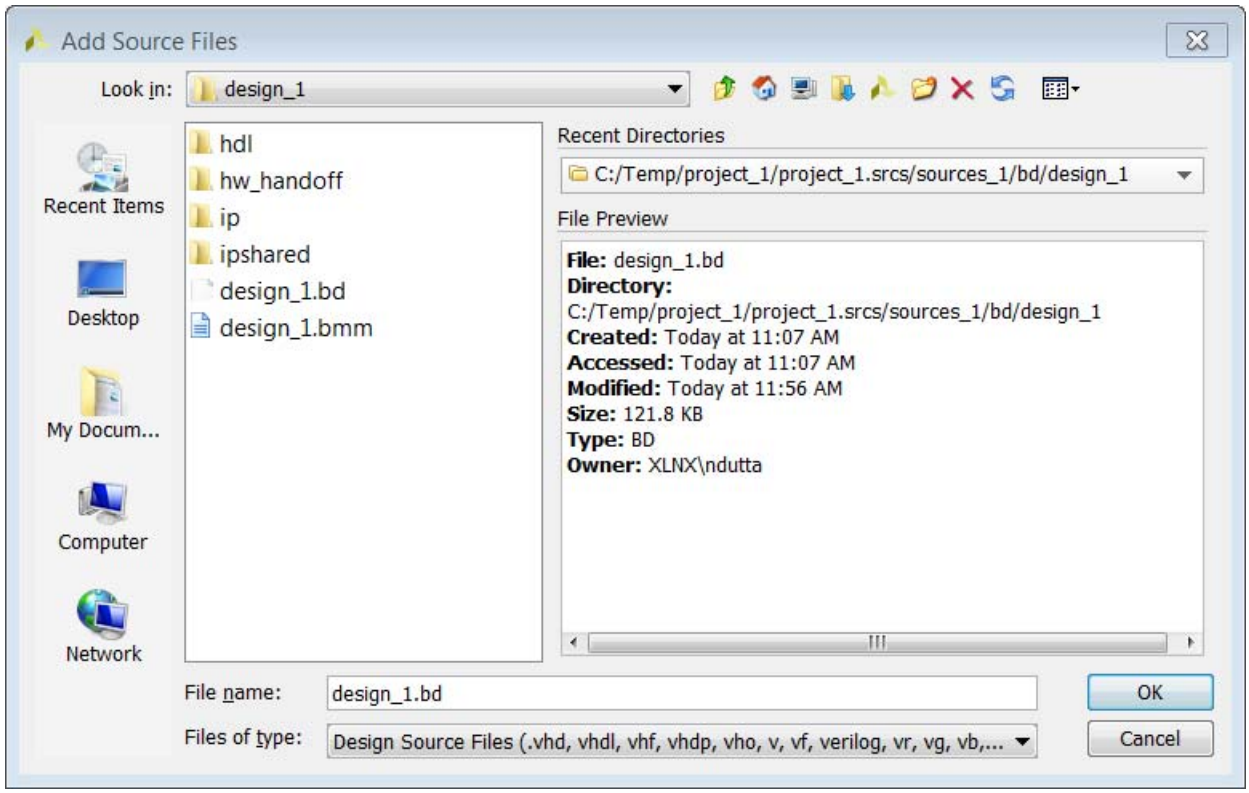


Figure 4-17: **Add block Design Source**

5. In the Add Existing Block Design Sources page you can enable or disable the **Copy sources into project** check box as needed for your current project.

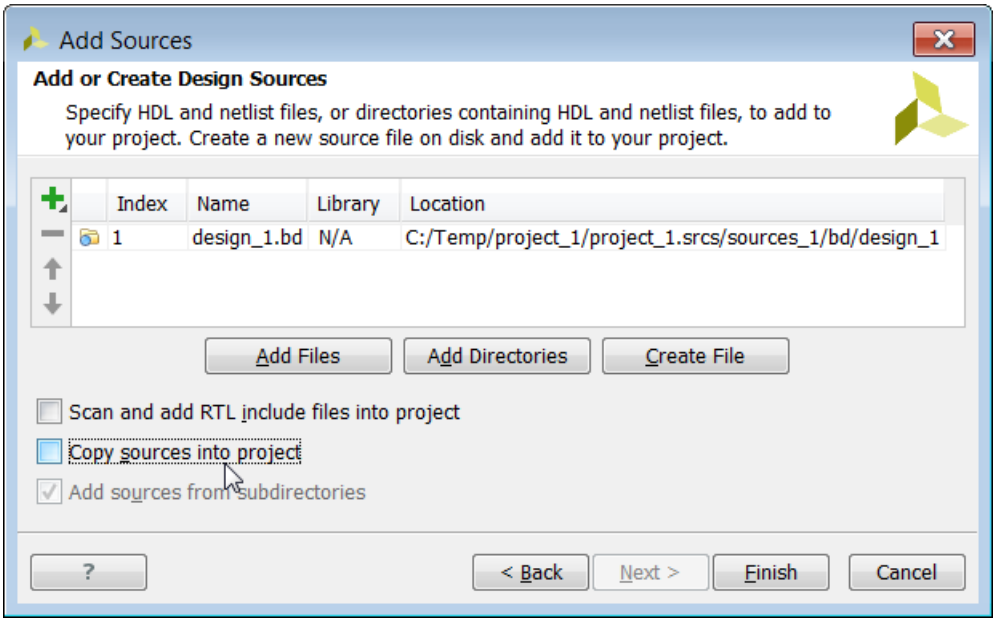


Figure 4-18: **Add Existing Block Design Sources**

You can reference the block design from its original location, or copy it into the local project directory. Managing the block design remotely is the recommended practice when working with revision control systems. See Revision Control for Block Designs, page 80. However, if someone edits the remote block design, they may inadvertently change your referenced copy. To avoid that, you can enable the **Copy sources into project** check box, as seen in Figure 4-18, page 79, so that you can change the block design when needed, but remote users won't be able to affect your design.

You can also set the block design as read-only to prevent modification. See Adding Read-Only Block Designs, page 80 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 block design to your project.

   In the Sources window, you can see the block design added under Design Sources.

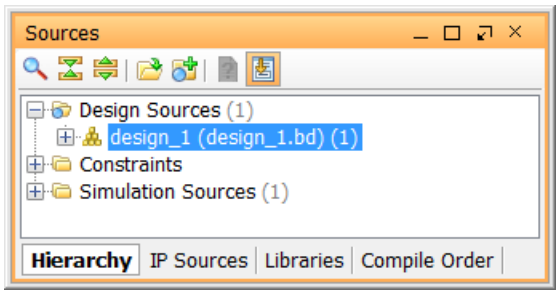


Figure 4-19: **Imported Block Design in the Sources Window**

7. Double-click the block design 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 block designs as read-only for use in other projects. This will prevent the block designs from being inadvertently modified.

If you have generated output products for the block design, you can change the file permissions on all files (i.e. `chmod 555 bd -R`). The block design, 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 (i.e. `chmod 555 bd/design_1/design_1.bd`). From this read-only block design you can still generate the output products needed for the design, but the block design itself cannot be changed. You can generate the output products for read-only block designs, if they have not been previously generated, provided the block design has been validated and saved.

Typically, for read-only block designs, 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 block design.

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

Revision Control for Block Designs

Revision control systems can be used to manage the various source files associated with Vivado IP Integrator block designs, in both Project and Non-Project Mode. As block designs are developed, and get 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.

Refer to this link in the *Vivado Design Suite User Guide: Design Flows Overview* (UG892) [Ref 2] 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 toolkit (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.

To export the hardware prior to synthesis, follow the steps below:

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

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

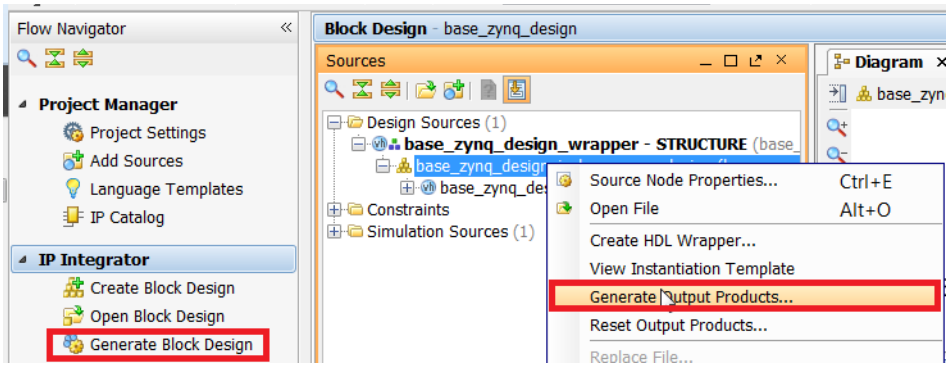


Figure 4-20: **Generating Block Design**

2. In the Generate Output Products dialog box, select the appropriate option and click **Generate**.
3. Export the hardware by selecting **File > Export > Export Hardware**.
4. In the Export Hardware dialog box, shown in Figure 4-21, page 82, disable the **Include bitstream** option, as there is no bitstream at this time.
5. Leave the **Export to:** field to its default value of **Local to Project**.
6. 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
```

For exporting the hardware after bitstream generation, follow the steps outlined below.

1. Select **File > Export > Export Hardware** from the menu.
2. In the Export Hardware dialog box, select **Include Bitstream**.
3. Leave the **Export to:** field to its default value of **Local to Project**.
4. Click **OK**.

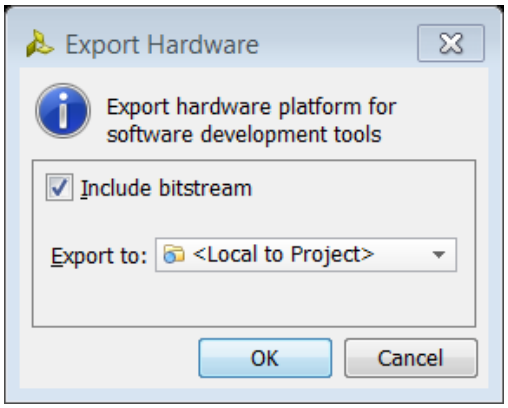


Figure 4-21: **Export Hardware - Include Bitstream**

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, refer to *Generating Basic Software Platforms Reference Guide* (UG1138) [Ref 12].

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 the Flow Navigator, 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 Figure 4-22.

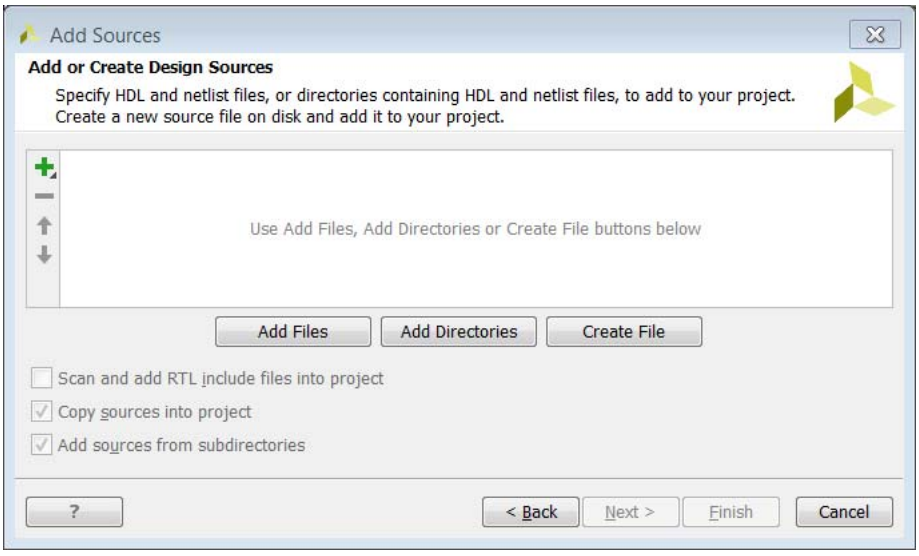


Figure 4-22: **Add Sources: Add or Create Design Sources Page**

3. Click **Add Files**.

The Add Source Files dialog box opens, as shown in Figure 4-23.

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

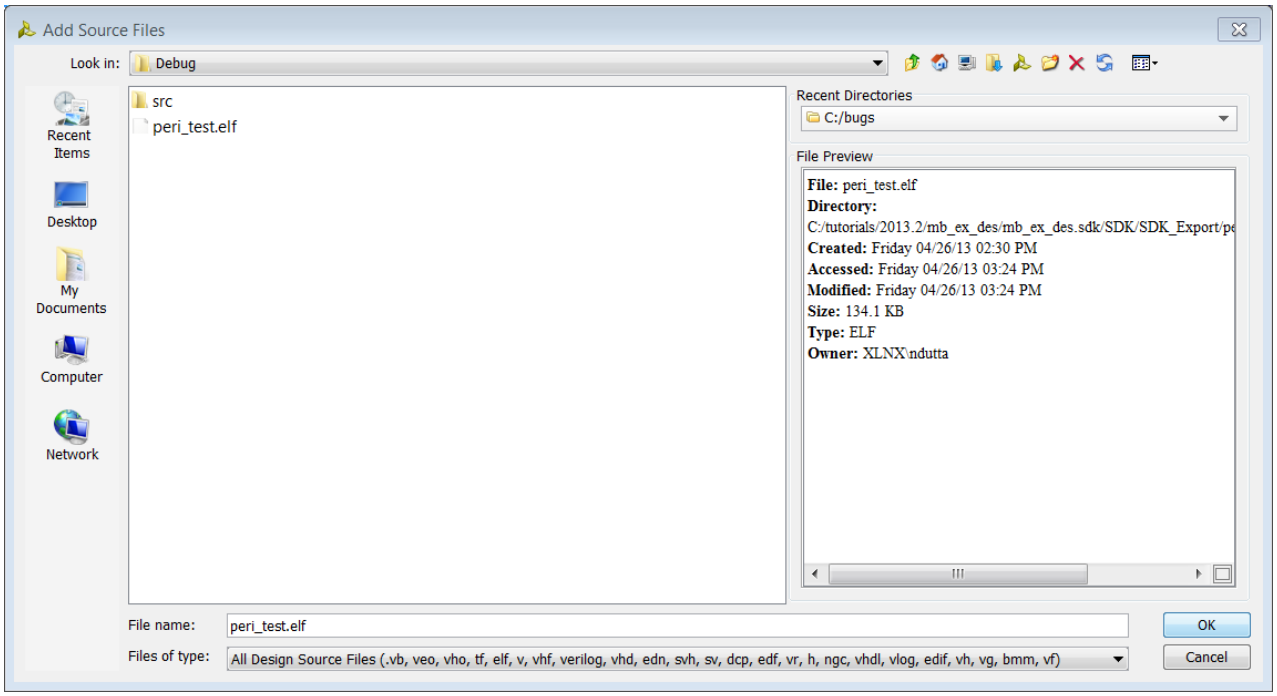


Figure 4-23: **Add Source Files Dialog Box**

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

5. Copy the ELF file into the local project by checking **Copy sources into 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.

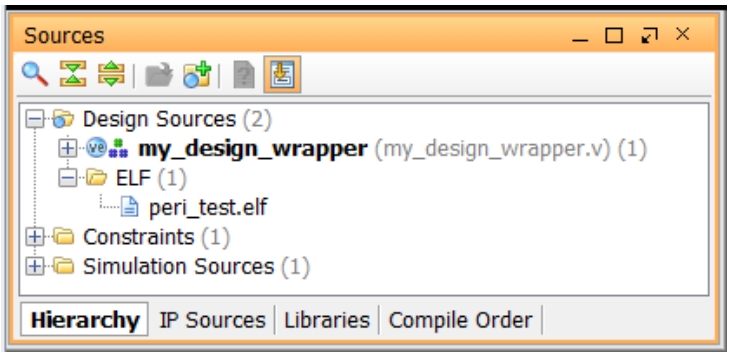


Figure 4-24: **Sources Window with ELF File Displayed**

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 below.

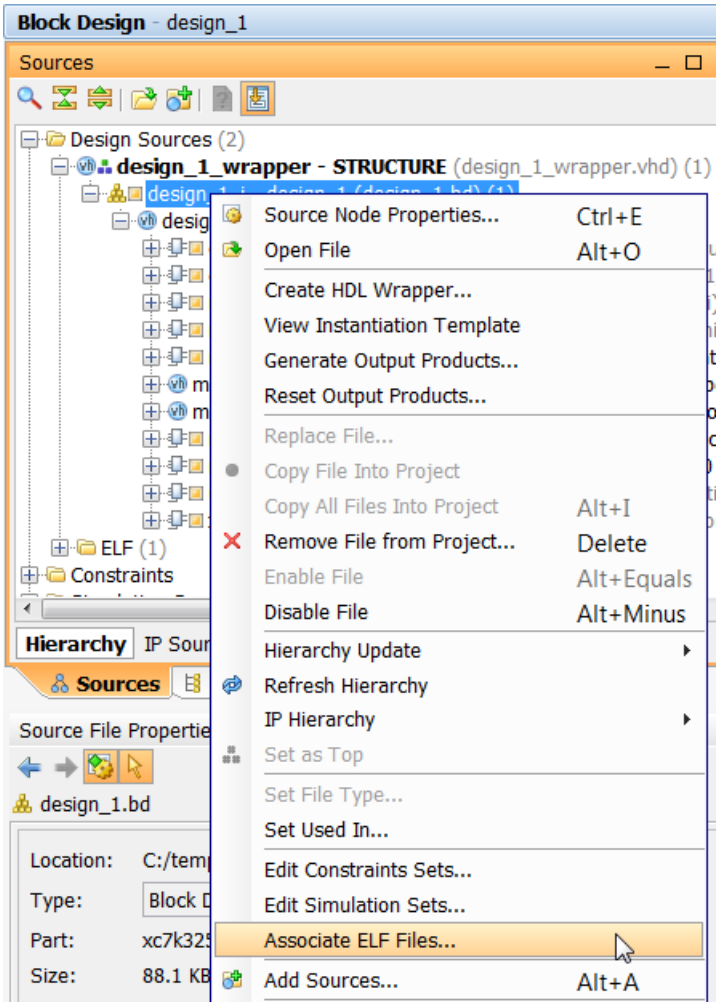


Figure 4-25: **Associate ELF Files Command**

The Associate ELF Files dialog box opens as shown in Figure 4-26, page 86.

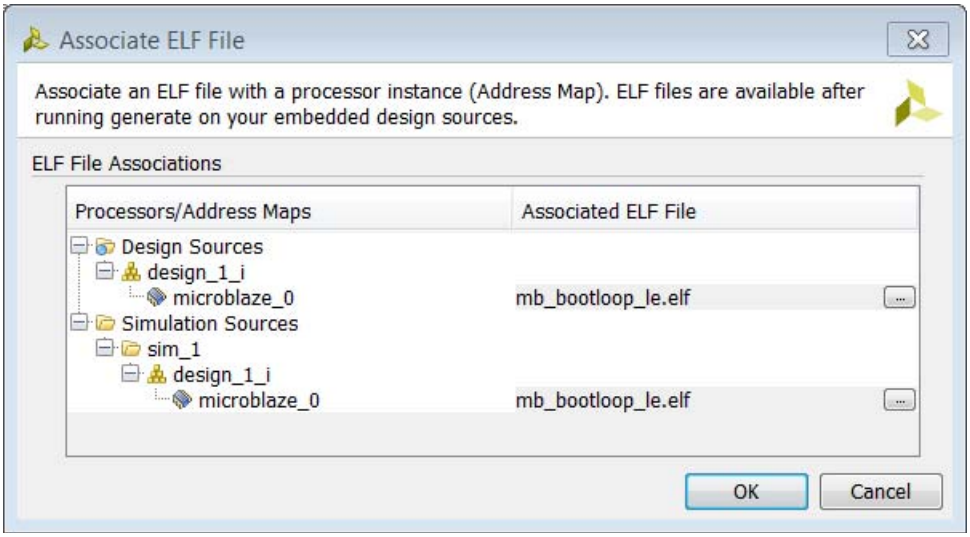


Figure 4-26: **Associating ELF Files with a Microprocessor**

8. Associate an ELF as a design source for including in the bitstream, or as a source for use during simulation, by clicking the appropriate **Browse** command. 

   The Select ELF Files dialog box opens.

9. Highlight the ELF file that you added to the project earlier, as shown below.

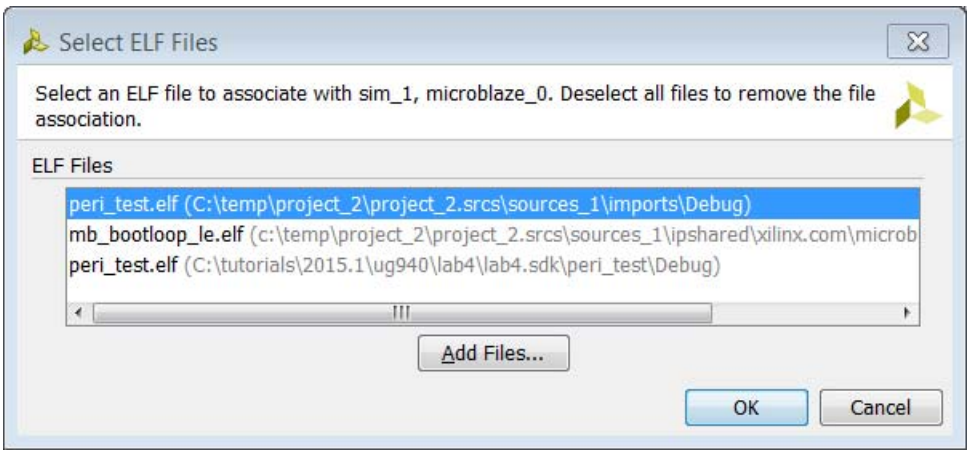


Figure 4-27: **Highlight the ELF File to Associate**

TIP: *You can also use the **Add Files** button 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. Make sure that the ELF file is displayed in the **Associated ELF File** column, as shown in Figure 4-28, page 87, 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.

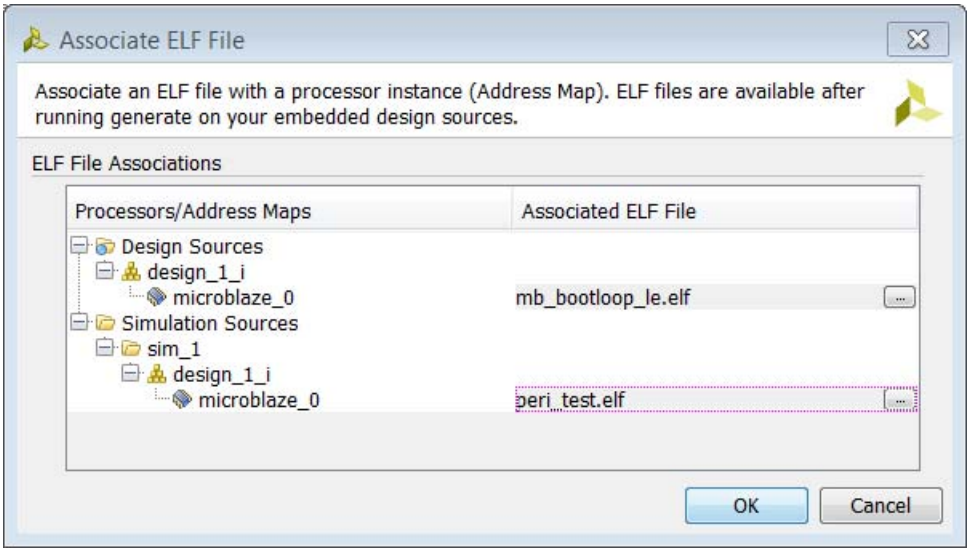


Figure 4-28: **Ensuring the ELF File is Populated**

TIP: *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 Hardware Design (UG898) [Ref 5] for more information.*

Propagating Parameters in IP Integrator

Introduction

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.

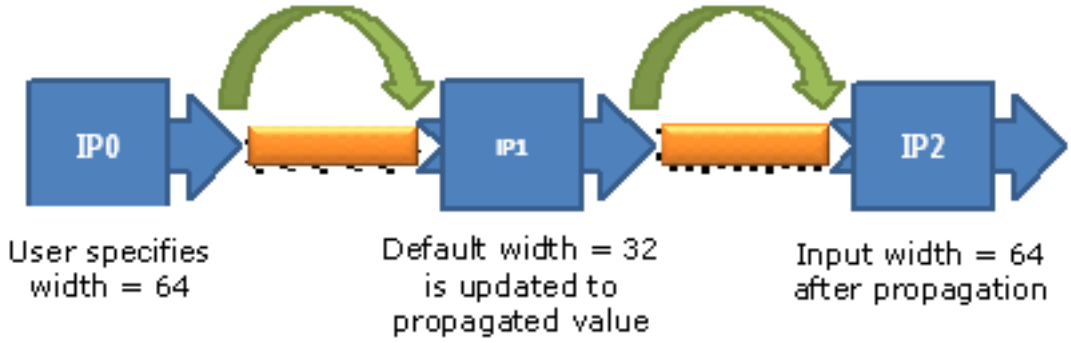


Figure 5-1: **Parameter Propagation Concept**

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, then 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 IP's symbol will show an extremely long and unusable list of low-level ports, which will be 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 IP's RTL or netlist) 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 AXI-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 and subtractors, 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.

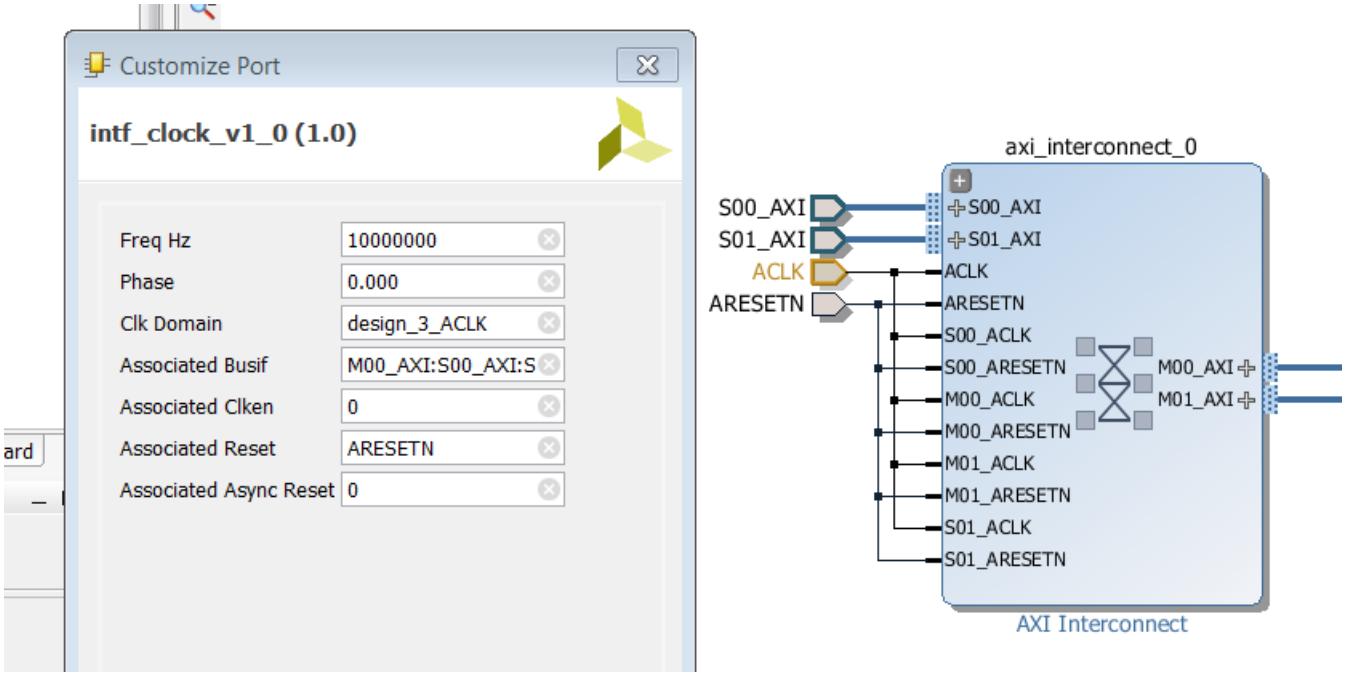


Figure 5-2: **ASSOCIATED_BUSIF**

In the figure above, 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. However, if one of the interfaces, such as M00_AXI, doesn't run at this clock frequency you will 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:

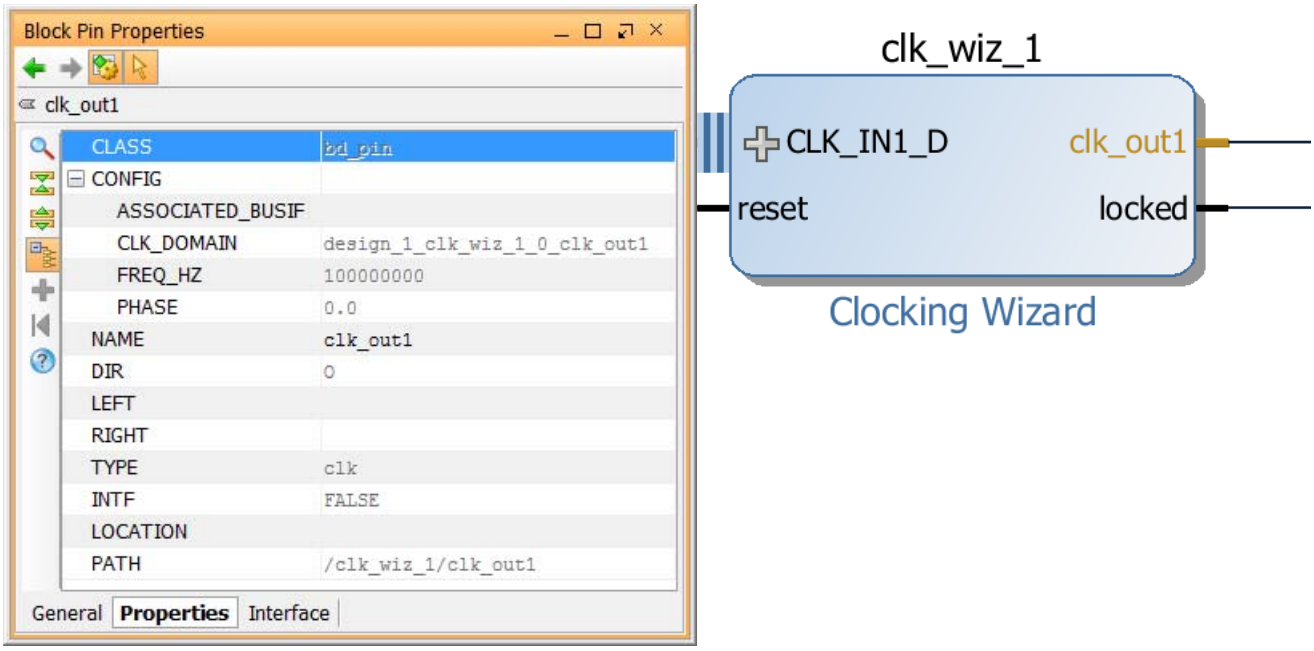


Figure 5-3: **Clock Properties**

These properties can also be reported by the following Tcl command:

```
report_property [get_bd_pins clk_wiz_1/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.

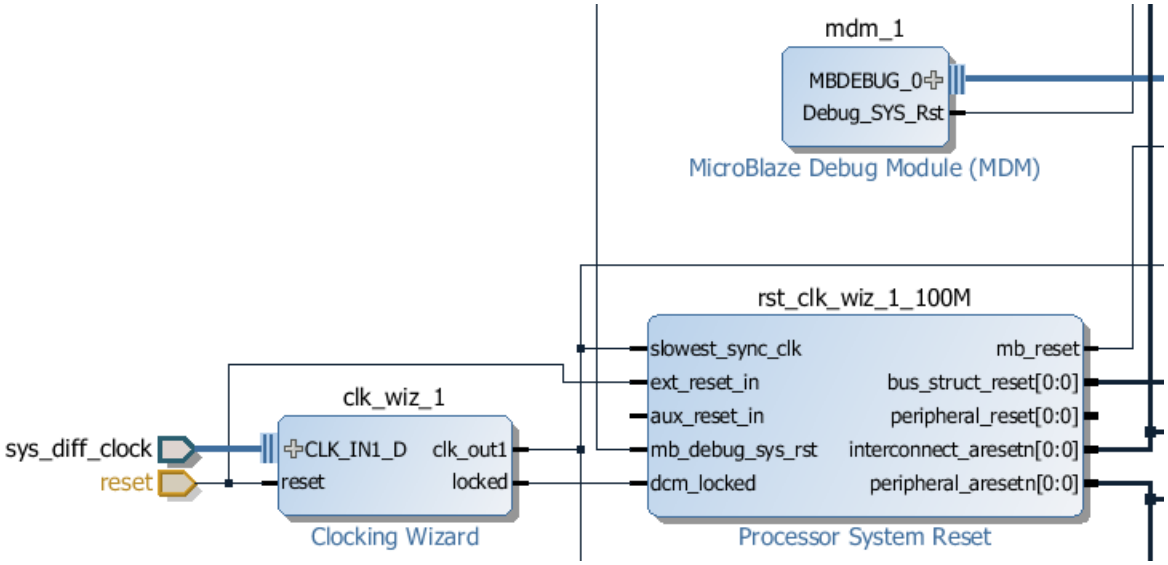


Figure 5-4: **Reset Signal**

The following figure shows the Properties window.

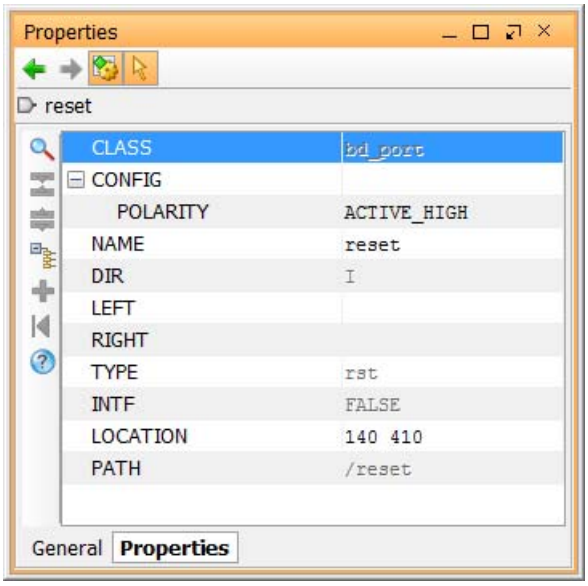


Figure 5-5: **Properties Window**

These properties can also be reported by the following Tcl command:

```
report_property [get_bd_ports reset]
```

This command writes the following output to the Tcl Console.

```
report_property [get_bd_ports /reset]
Property          Type    Read-only  Visible  Value
CLASS             string  true       true     bd_port
CONFIG.POLARITY   string  false      true     ACTIVE_HIGH
DIR               string  true       true     I
INTF              string  true       true     FALSE
LEFT              string  false      true
LOCATION          string  false      true     140 410
NAME              string  false      true     reset
PATH              string  true       true     /reset
RIGHT             string  false      true
TYPE              string  true       true     rst
```

Figure 5-6: **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 as shown in the following figure, and look at the properties window.

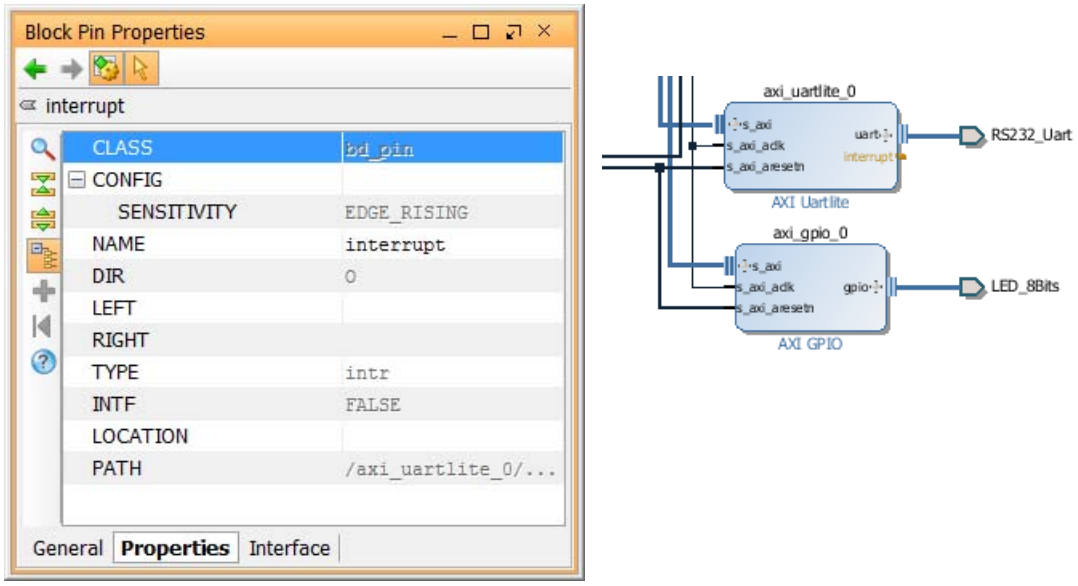


Figure 5-7: **Interrupt Properties: Block Diagram and Properties Window**

These properties can also be reported by using the following Tcl command:

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

The above command returns the following information.

```
report_property [get_bd_pins /axi_uartlite_0/interrupt]
Property            Type    Read-only  Visible  Value
CLASS               string  true       true     bd_pin
CONFIG.SENSITIVITY  string  true       true     EDGE_RISING
DIR                 string  true       true     O
INTF                string  true       true     FALSE
LEFT                string  true       true
LOCATION            string  false      true
NAME                string  false      true     interrupt
PATH                string  true       true     /axi_uartlite_0/interrupt
RIGHT               string  true       true
TYPE                string  true       true     intr
```

Figure 5-8: **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** in the Design Canvas toolbar.
- Select **Tools > Validate Design** from the Vivado Menu.
- Use the Tcl command: `validate_bd_design`

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.

IP Integrator's parameter propagation 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. However, 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 greyed-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 grey, and **Auto** is placed next to the parameter to indicate you cannot change this parameter.

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

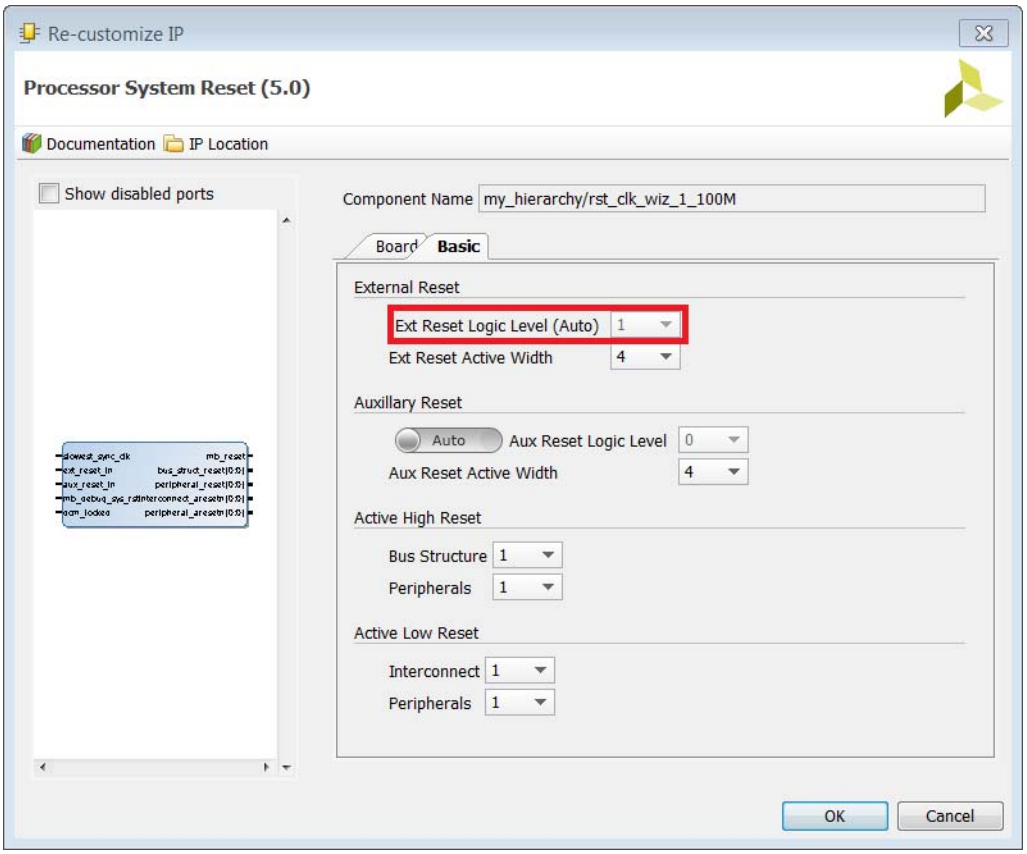


Figure 5-9: **Auto-Computed Parameter**

- **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 will tell 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.

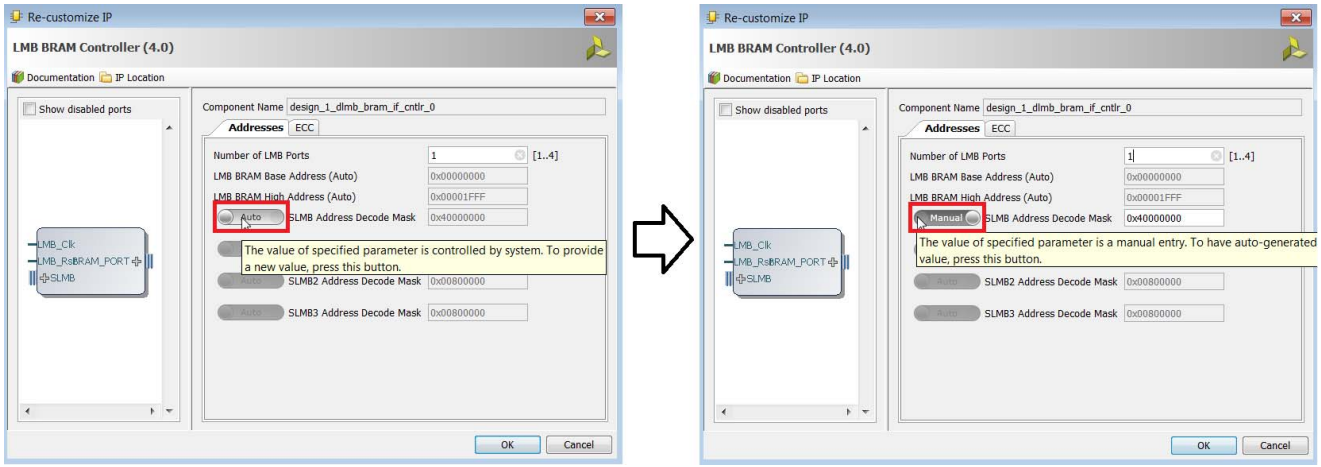


Figure 5-10: **Parameter to Override**

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

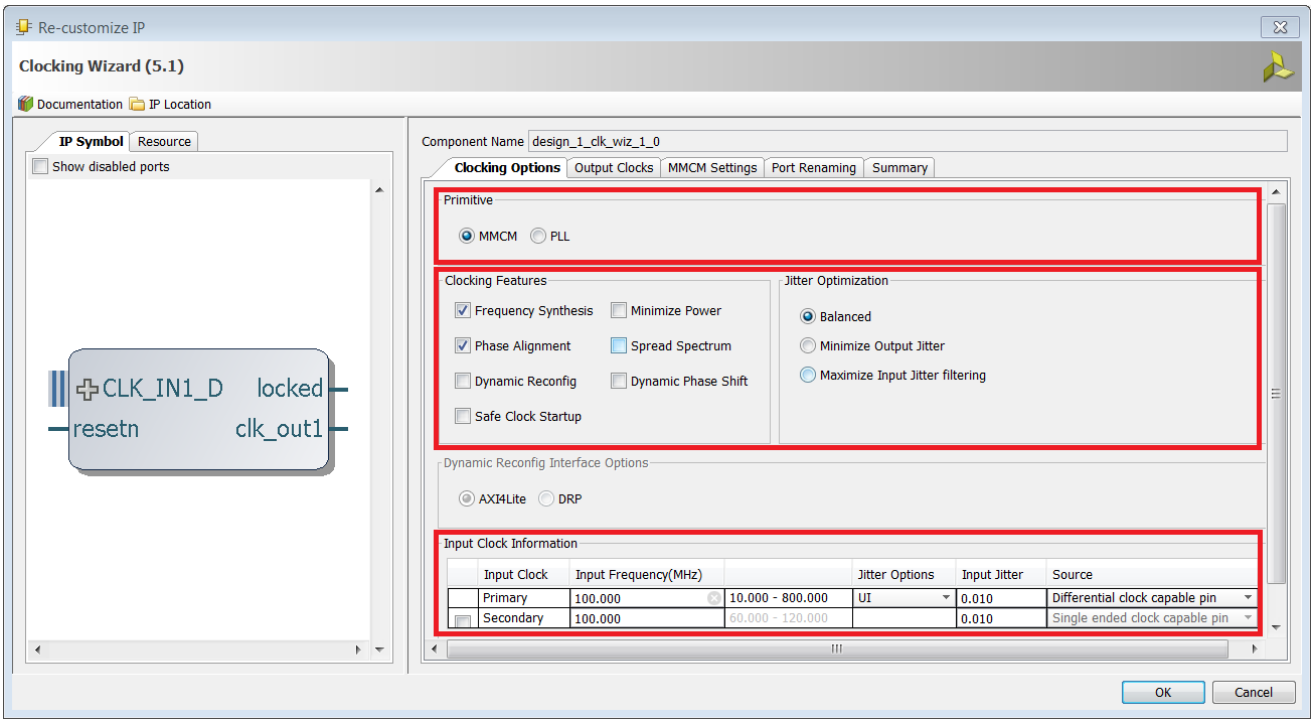


Figure 5-11: **User-Configurable Parameter**

- **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.

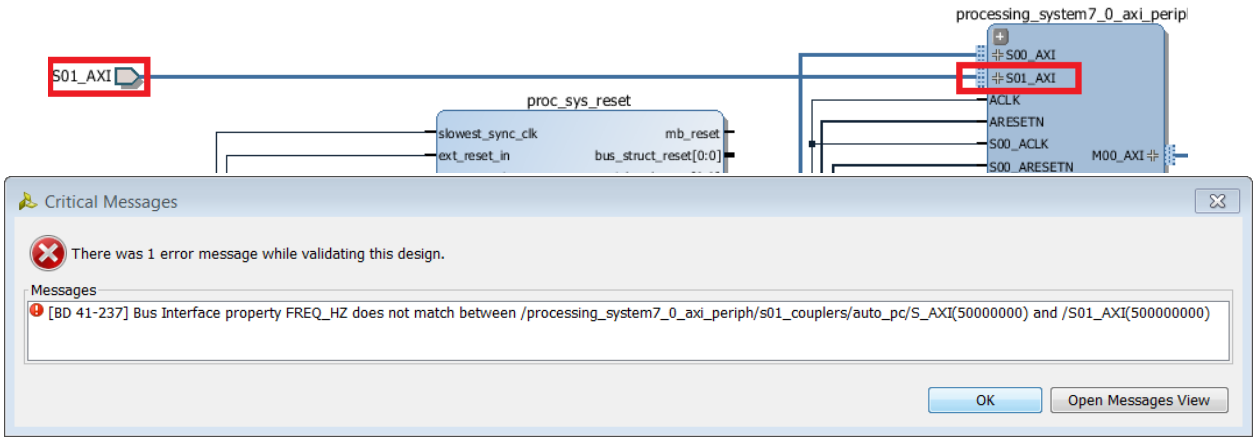


Figure 5-12: **FREQ_HZ property mismatch between a port and an interface pin**

- The S01_AXI port has a frequency of 500 MHz as can be seen in the properties window.
- The S01_AXI interface of the AXI Interconnect is set to a frequency of 50 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.

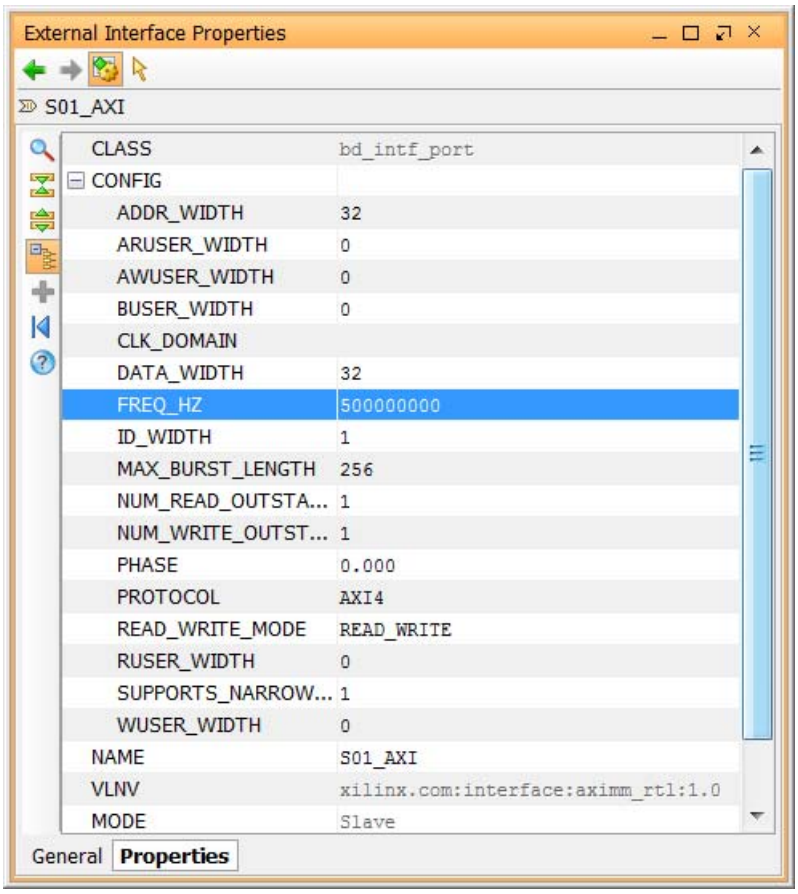


Figure 5-13: **Change Frequency Port in Properties Window**

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

Chapter 6

Debugging IP Integrator Designs

Overview

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.

Using the HDL Instantiation Flow in IP Integrator

For debugging the elements of a block design using the Vivado Hardware Manager, the IP Integrator feature 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 *LogiCORE IP Integrated Logic Analyzer Product Guide* (PG172) [Ref 16].
- **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 *LogiCORE IP System Integrated Logic Analyzer Product Guide* (PG261) [Ref 17].

IMPORTANT: *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 4 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 Figure 6-1.

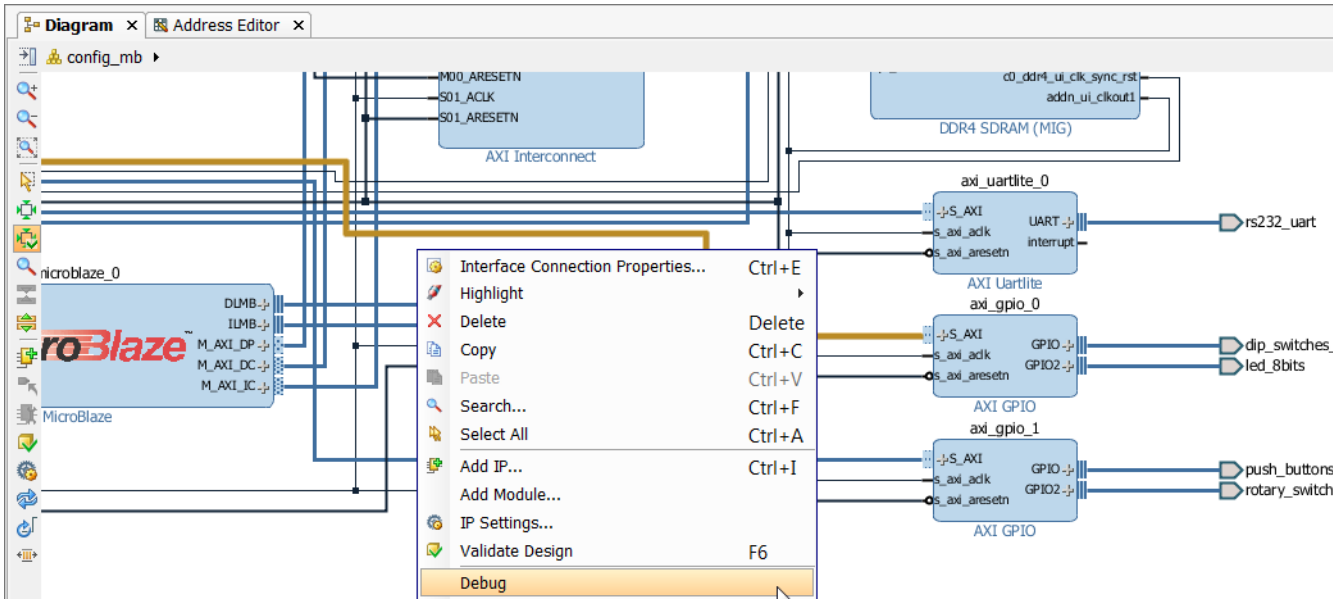


Figure 6-1: **Mark Nets to Debug from Context Menu**

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

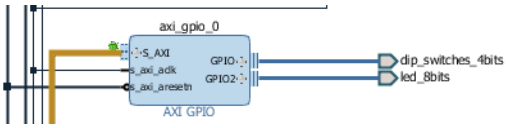


Figure 6-2: **Mark Nets to Debug from Context Menu**

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

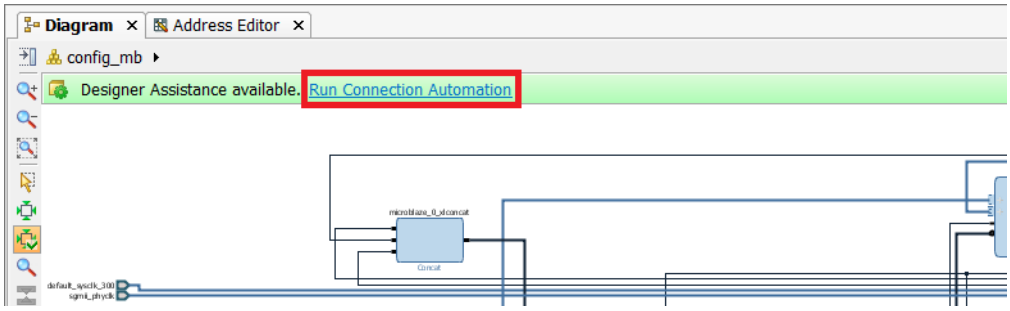


Figure 6-3: **Mark Nets to Debug from Context Menu**

Clicking on **Run Connection Automation** link displays the Run Connection Automation dialog box.

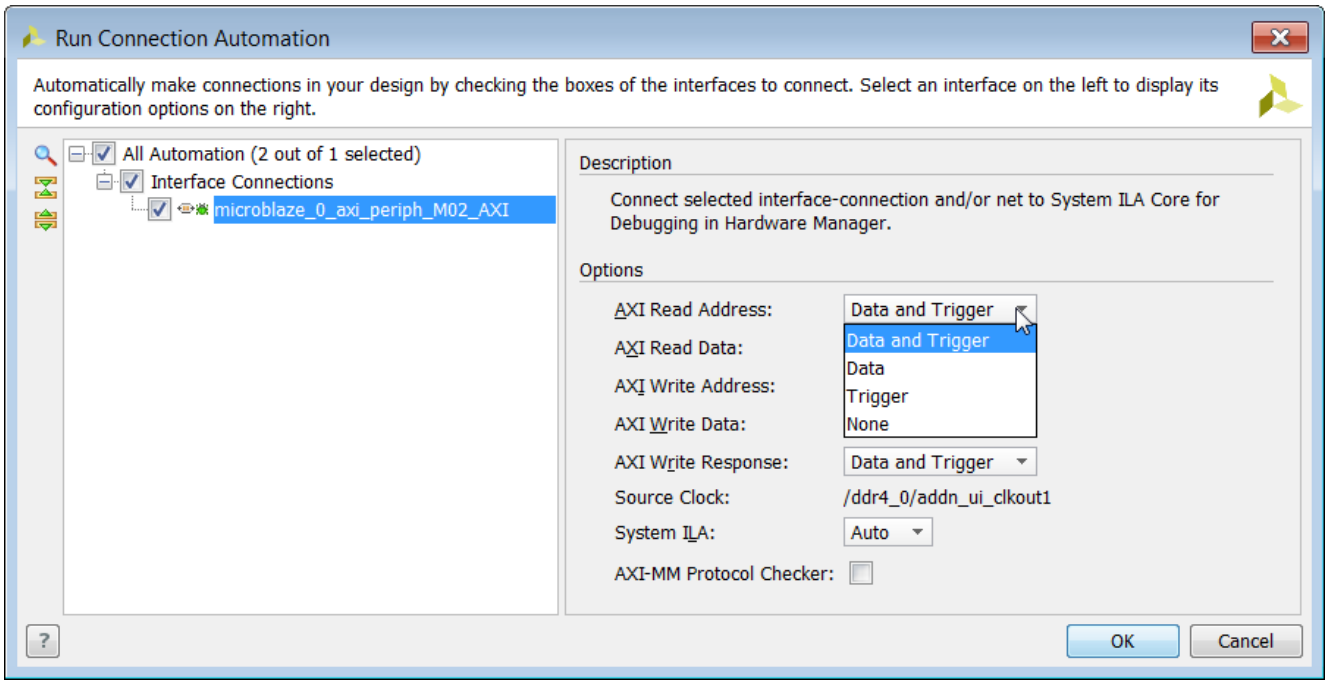


Figure 6-4: **Selecting Data and/or Trigger option for interface signals**

Since 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 a non-interface net is marked for debug and **Run Connection Automation** link is clicked.

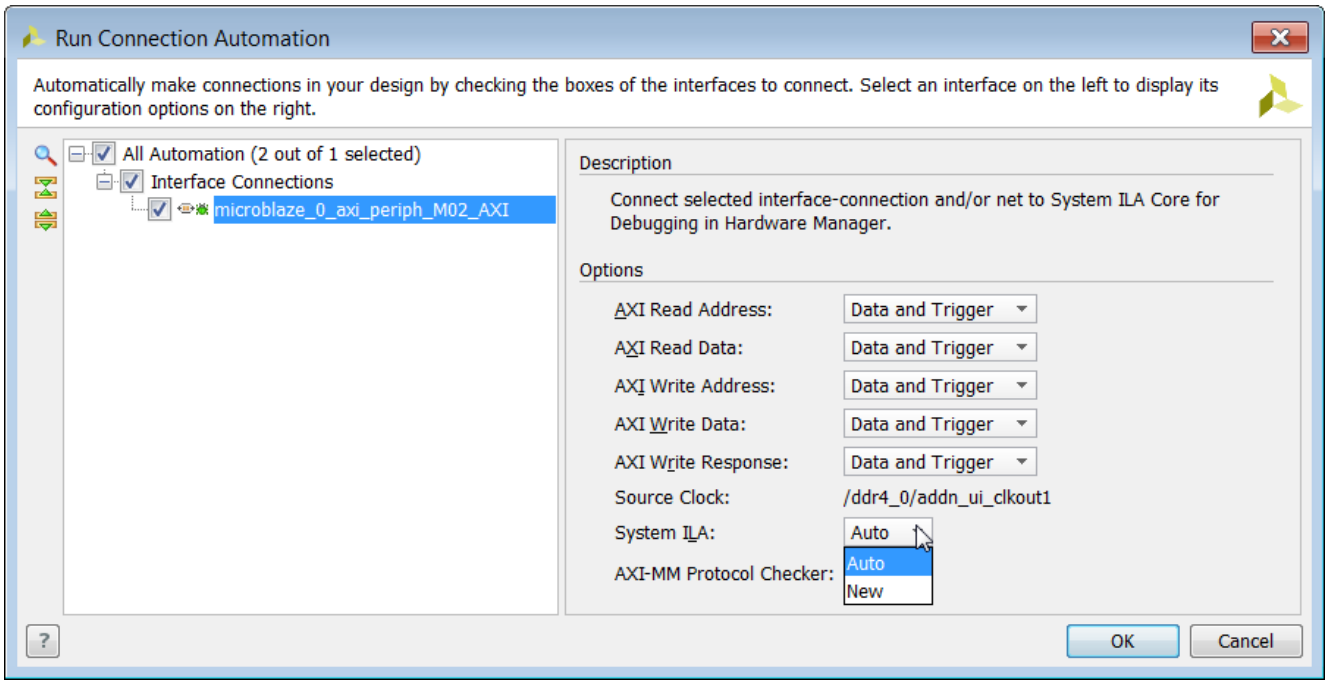


Figure 6-5: **Setting the System ILA options**

As seen in Figure 6-5, page 102, 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 Debug (UG908) [Ref 7].*

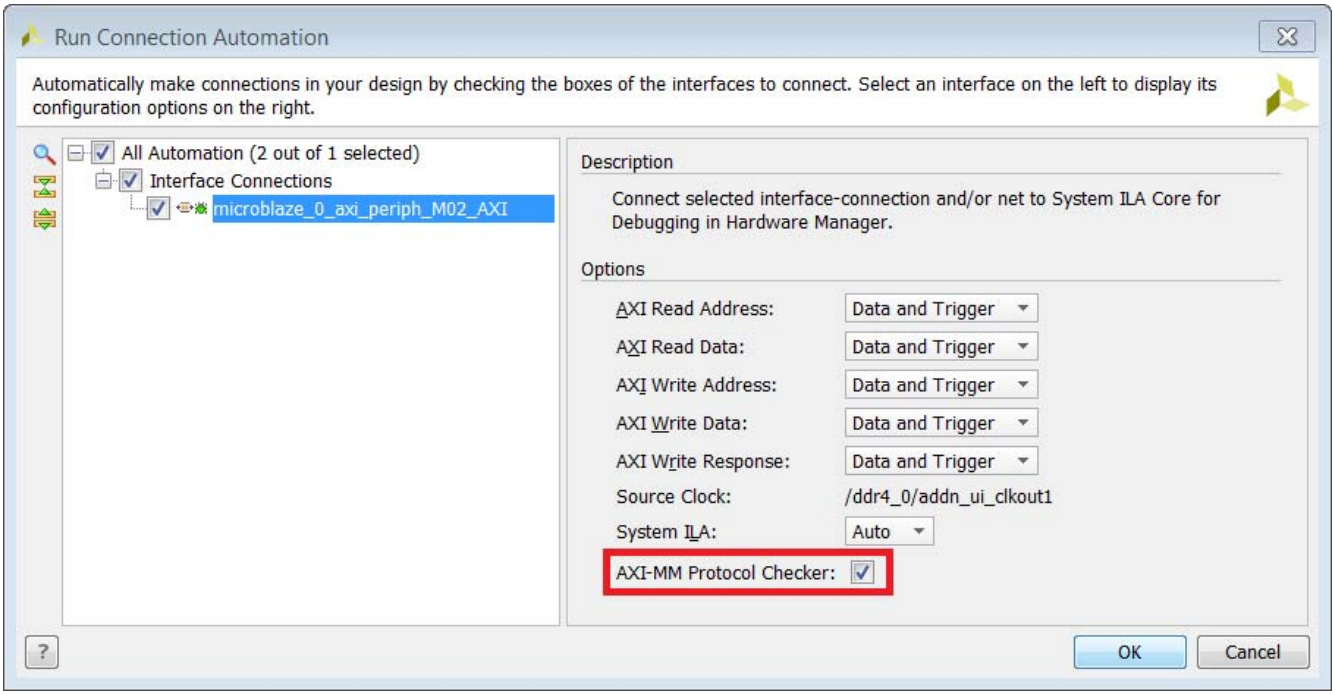


Figure 6-6: **Setting the System ILA options**

When you click **OK** on the Run Connection Automation dialog box you will see messages such as the one shown below, indicating what action was taken by the tool.

```
Debug Automation : Instantiating new System ILA block '/system_ila' 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
```

After a net has been marked for debug, you can remove the DEBUG attribute by right-clicking on the net and selecting **Clear Debug** from the context menu. 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.

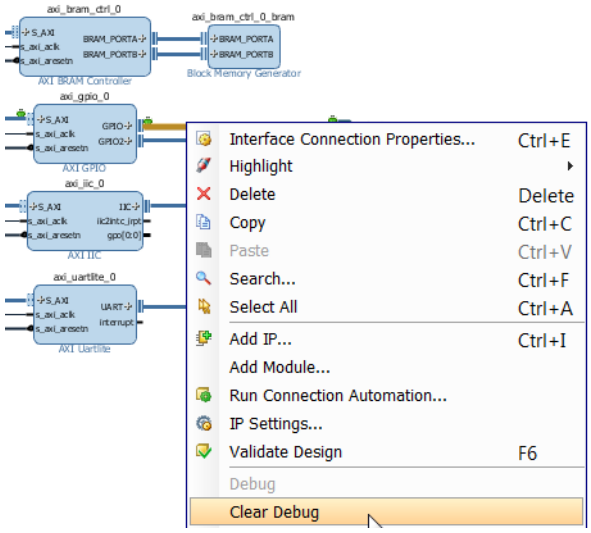


Figure 6-7: **Removing Debug Cores from the block design**

Manually Configuring the System ILA

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 on the IP in the block design, or right-click on 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 Figure 6-8, page 105. The **IP Symbol** and **Resources** tab of the System ILA dialog box shows the pins present on the System ILA IP, and the BRAM resources that are consumed by the System ILA debug core.

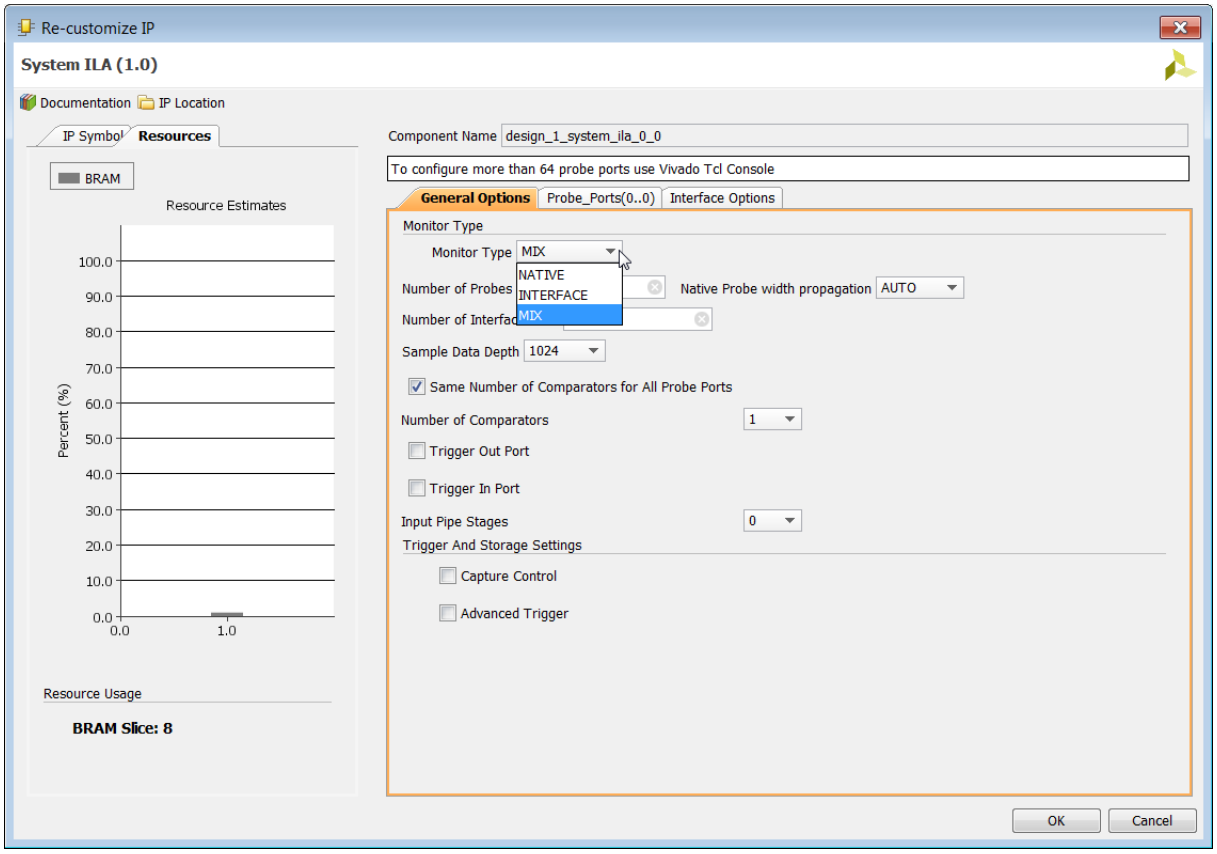


Figure 6-8: **The System ILA Configuration Wizard**

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.

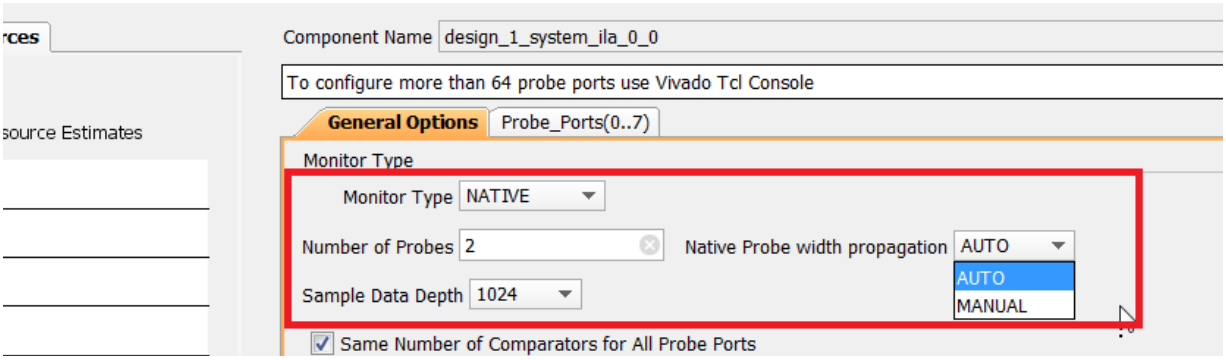


Figure 6-9: **The System ILA Configuration Wizard**

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 in Figure 6-9. 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.

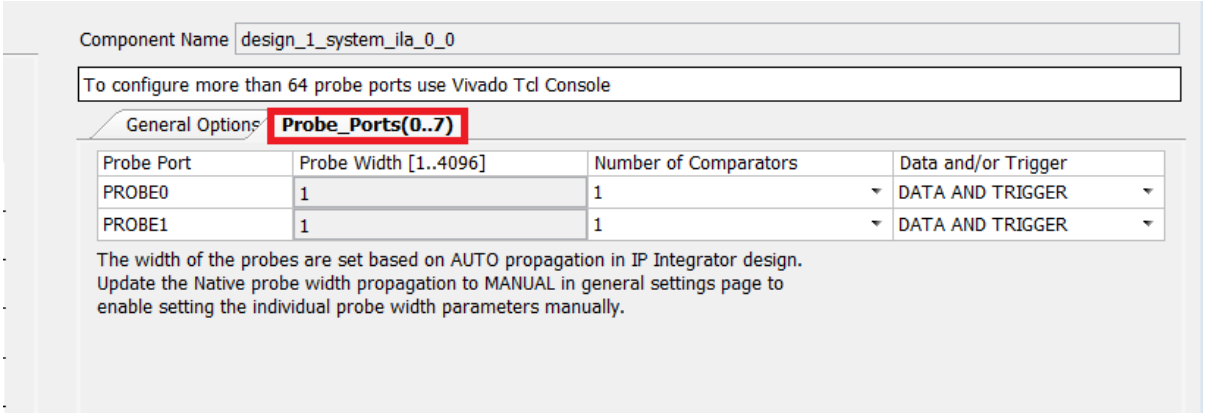


Figure 6-10: **Setting the System ILA options in MANUAL mode propagation**

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 is displayed, 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.*

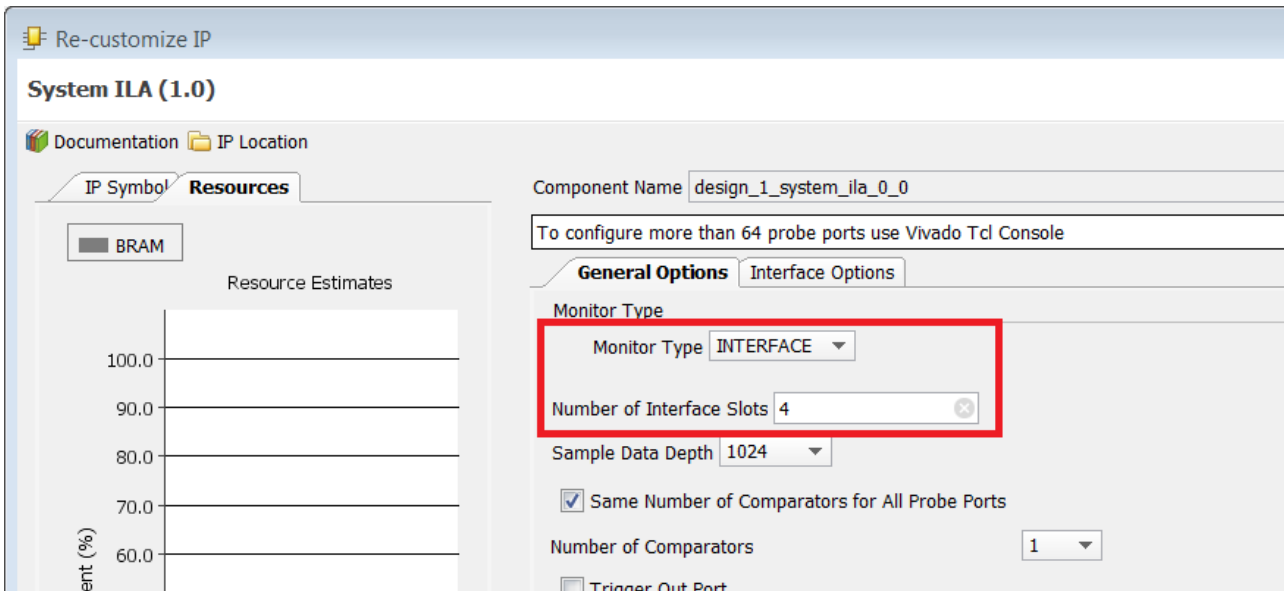


Figure 6-11: **Setting the System ILA options when Monitor Type is set to INTERFACE**

Additionally, the **Interface Options** tab is added to the Re-customize IP dialog box to let you configure the interface slots as shown in Figure 6-12. 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.

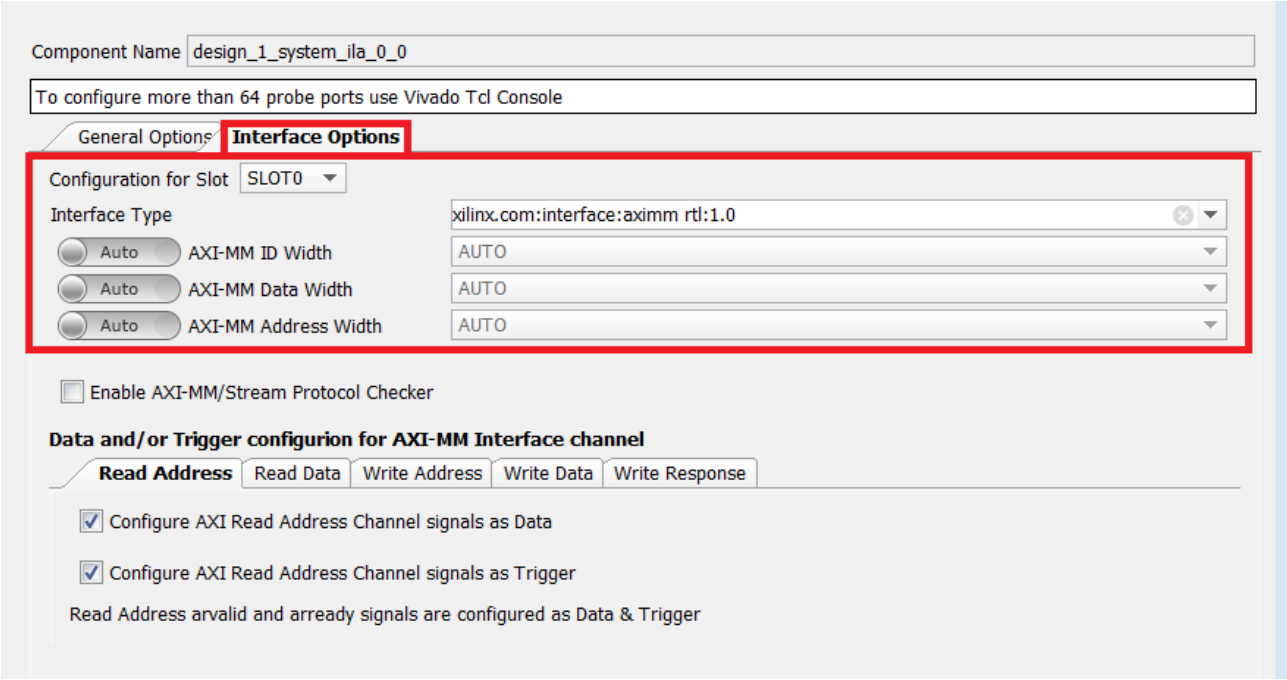


Figure 6-12: **Setting the System ILA options using the Interface Options tab**

When the Monitor Type field is set to MIX, both the Probe Options and the Interface Options tabs are displayed, as shown in Figure 6-8, page 105.

Validating the System ILA

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, you will need to use the Netlist Insertion flow to connect those signals to an ILA debug core in the top-level design. Refer to Using the Netlist Insertion Flow, page 115 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 below. See Displaying Layers in the Block Design, page 11 for more information.

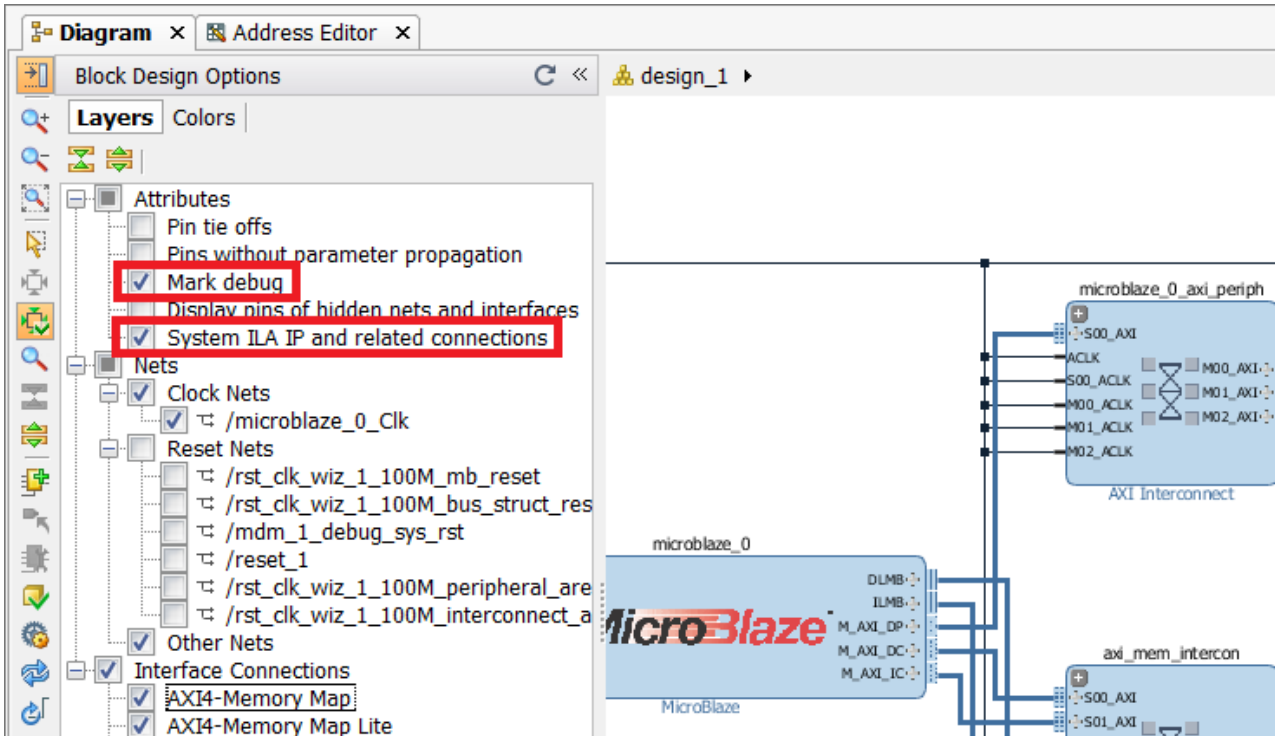


Figure 6-13: **Viewing nets marked for debug and System ILA connectivity using Layers view**

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, page 75.

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 Debug (UG908) [Ref 7].*

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, page 100*.

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. You can instantiate an ILA by following the steps described below.

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

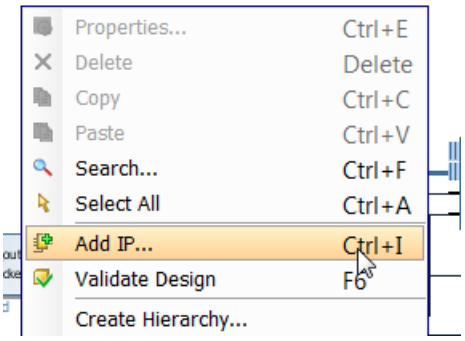


Figure 6-14: **Add IP from Context Menu**

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.

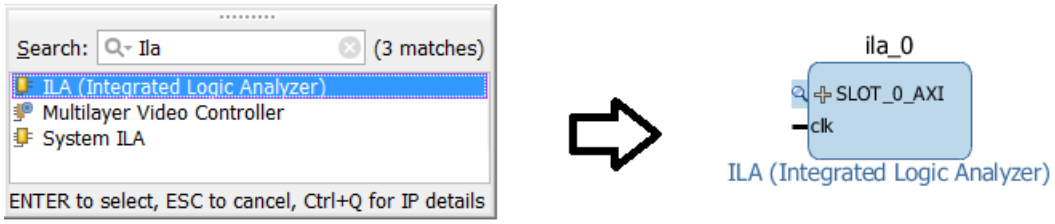


Figure 6-15: **Instantiated ILA Core**

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

   The Re-Customize IP dialog box opens, as shown in Figure 6-16, page 110.

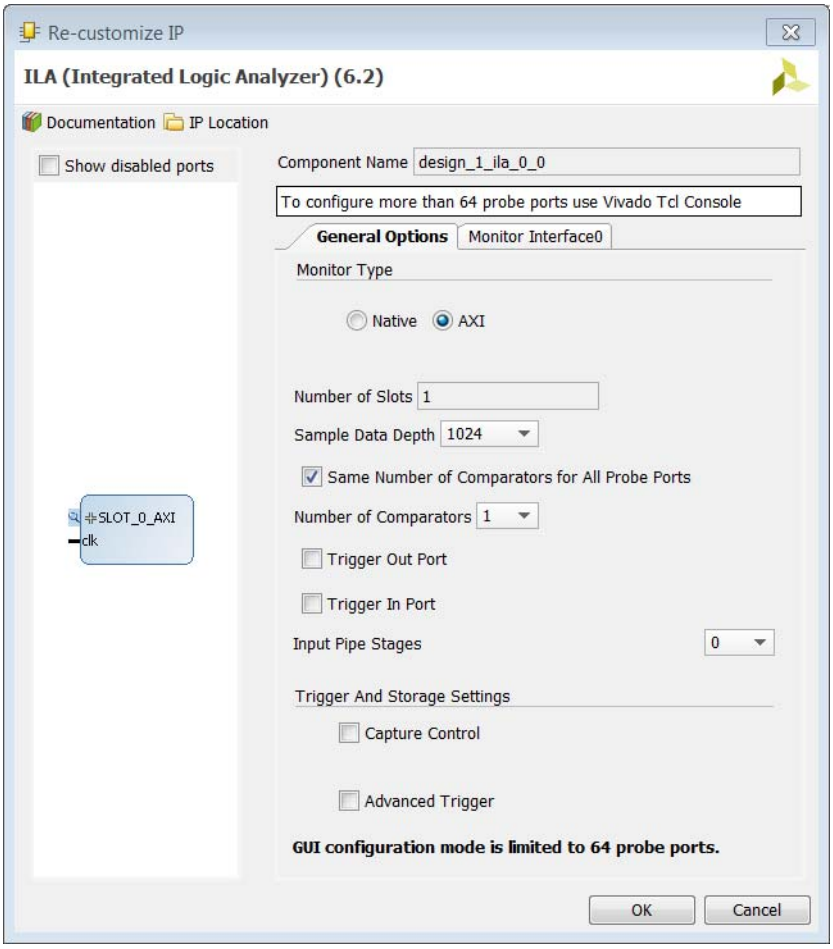


Figure 6-16: **Re-customized IP Dialog Box for the ILA Core**

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) [Ref 7].

CAUTION! *You can only monitor one AXI interface using an ILA. Do not change the value of the Number of Slots. If more than one AXI interface is desired to be debugged, 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 Figure 6-17, page 111. Set this value to the number of signals you want to be monitored.

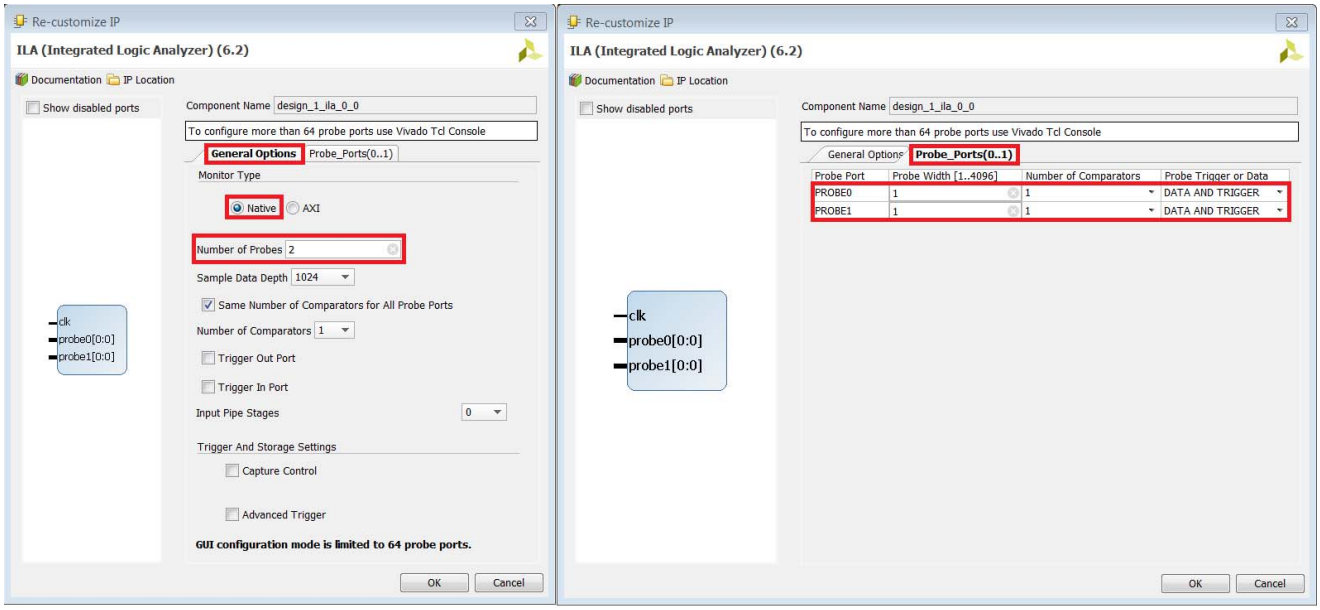


Figure 6-17: **Native Monitor Type Option**

In Figure 6-17, the **Number of Probes** is set to 2 in the General Options tab. You can see under the Probe_Ports tab that two ports are displayed. 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.

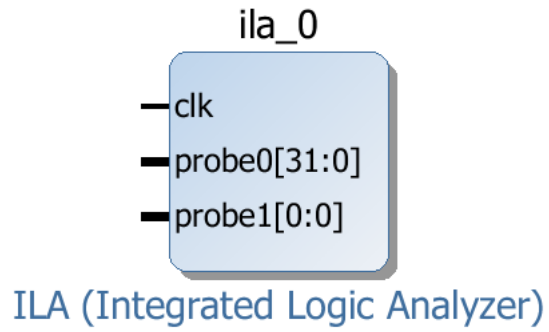


Figure 6-18: **ILA Core after Changes in the Re-customize IP Dialog Box**

5. After configuring the ILA, make the required connections to the pins of the ILA on the IP Integrator canvas, as shown in the following figure.

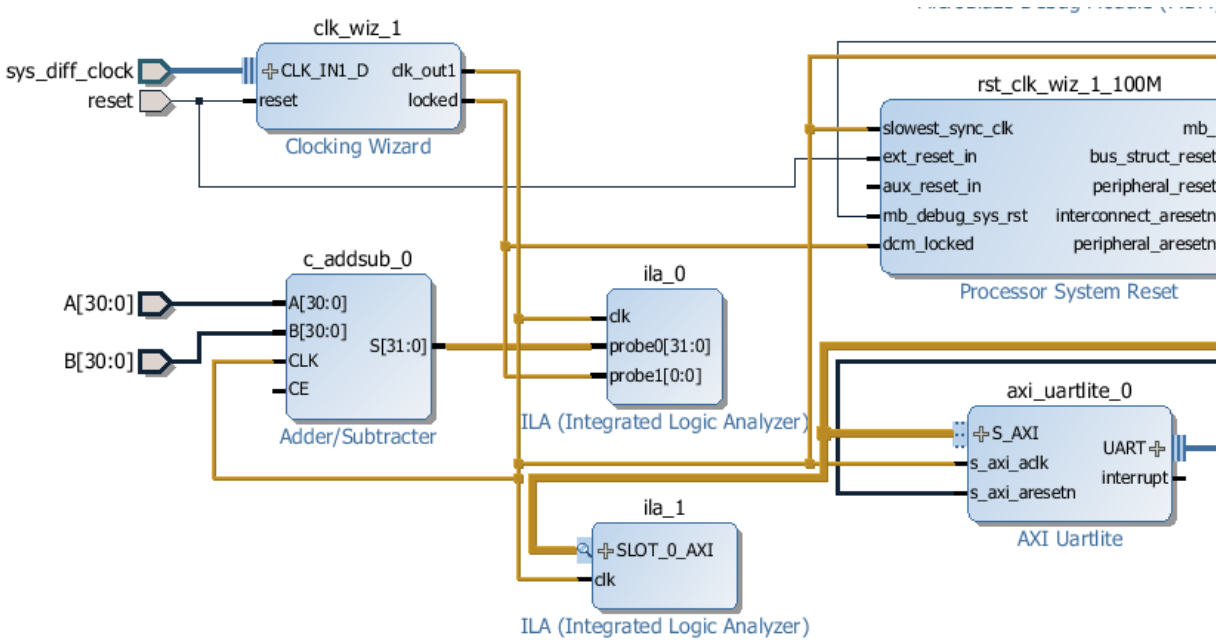


Figure 6-19: **Instantiated ILAs to Monitor AXI and Non-AXI Signals**

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.

Connecting Interface Ports to an ILA or VIO Debug Core

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, page 25.

As an example, consider the MicroBlaze processor design for the KC705 board, shown in Figure 6-20, page 113. This design has a GPIO which is configured to use both the 8-bit LED interface and the 4-bit dip switches on the KC705 board.

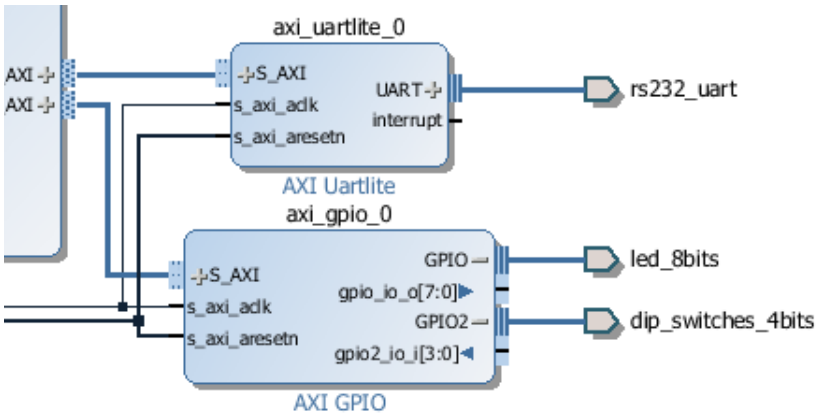


Figure 6-20: **Monitoring Interface Signals in a Block Design**

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 Figure 6-21, 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**.

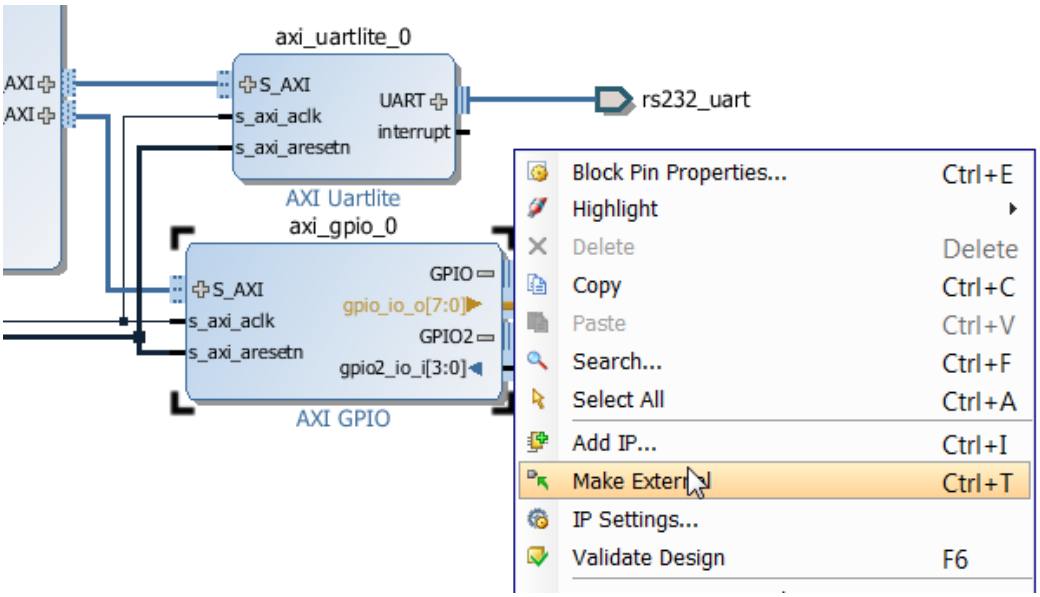


Figure 6-21: **Using Make External Command to Connect I/O Pin to an I/O Port**

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.

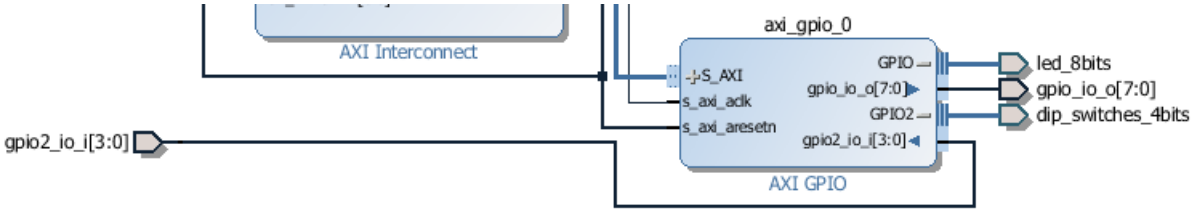


Figure 6-22: **External Ports Connected to Pins**

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 Figure 6-23, page 115, 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.

   You also need to 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

4. 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.

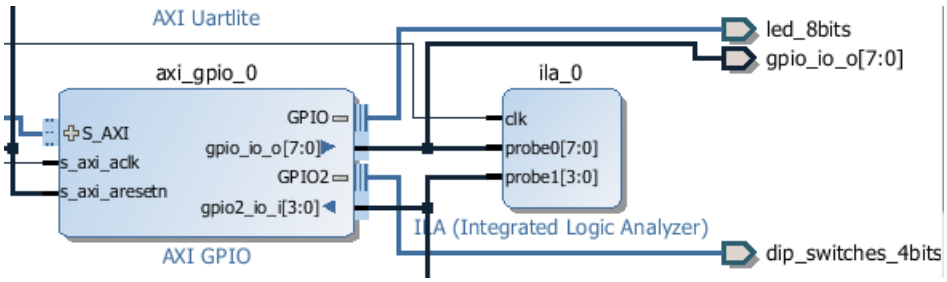


Figure 6-23: **ILA Probes Connected to the Input/Output Pins for Monitoring**

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) [Ref 7].

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

To mark nets for debug, in the block design, highlight the net, right-click and select **Mark Debug**.

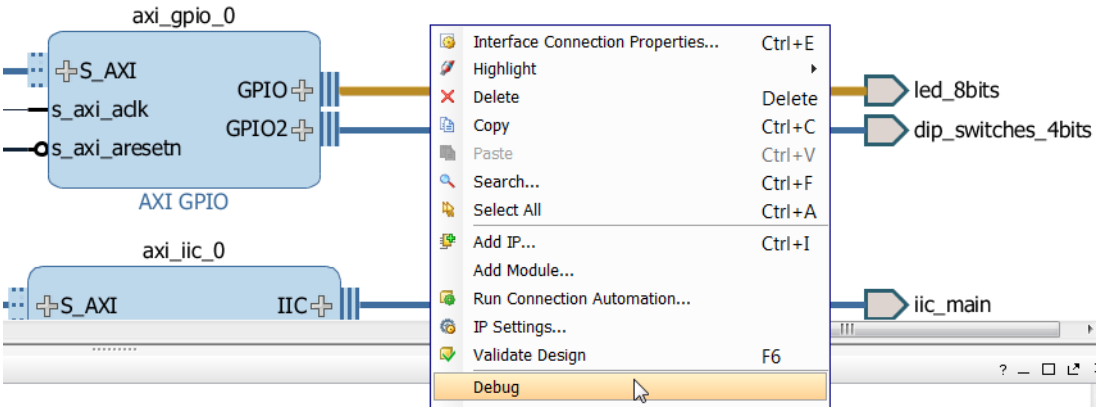


Figure 6-24: **Mark Debug Command**

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.

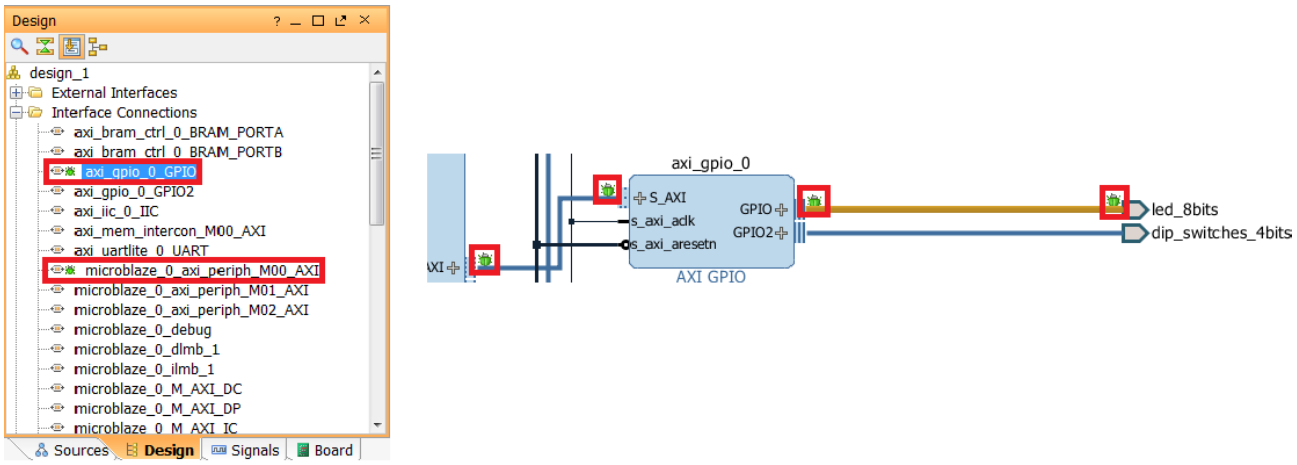


Figure 6-25: **Identify Nets Marked for Debug**

TIP: *You can mark multiple nets for debug at the same time by highlighting them together, right-clicking and selecting* ***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.

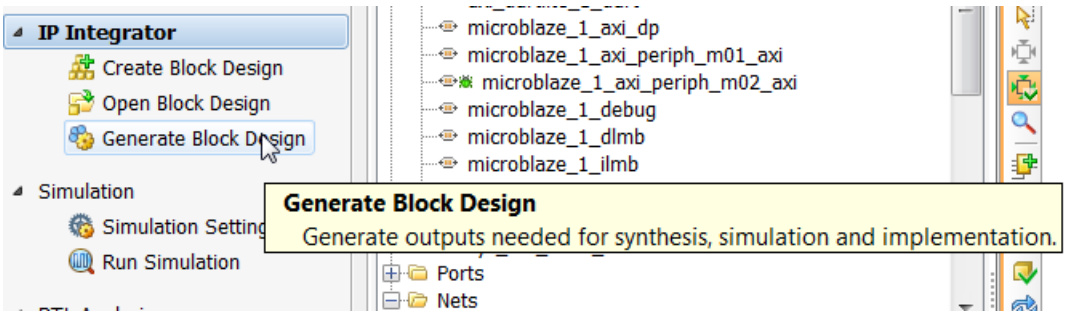


Figure 6-26: **Generate Block Design Command**

2. In the Generate Output Products dialog box, click **Generate**.

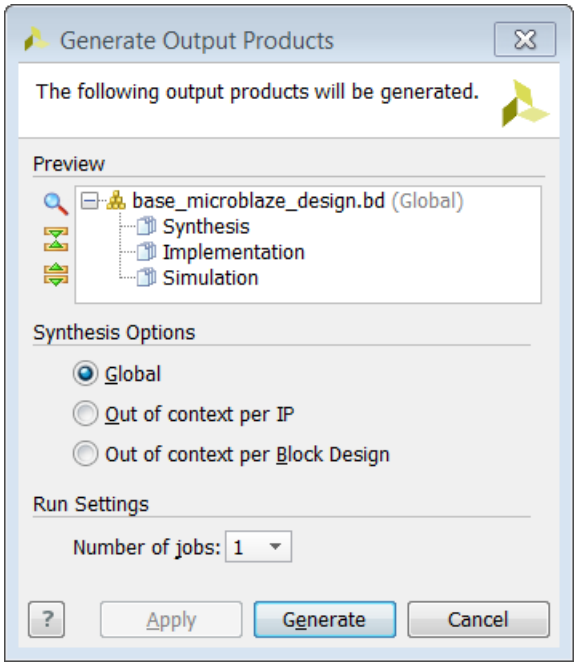


Figure 6-27: **Generate Output Products Dialog Box**

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.

This prevents the Vivado tools from optimizing and renaming the nets.

```
3462  attribute DEBUG : string;
3463  attribute DEBUG of axi_gpio_0_GPIO_TRI_O : signal is "true";
3464  attribute MARK_DEBUG : boolean;
3465  attribute MARK_DEBUG of axi_gpio_0_GPIO_TRI_O : signal is std.standard.true;
```

Figure 6-28: **MARK_DEBUG Attributes in the Generated HDL File**

Synthesize the Design and Insert the ILA Core

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

1. From the **Flow Navigator > Synthesis >** 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.

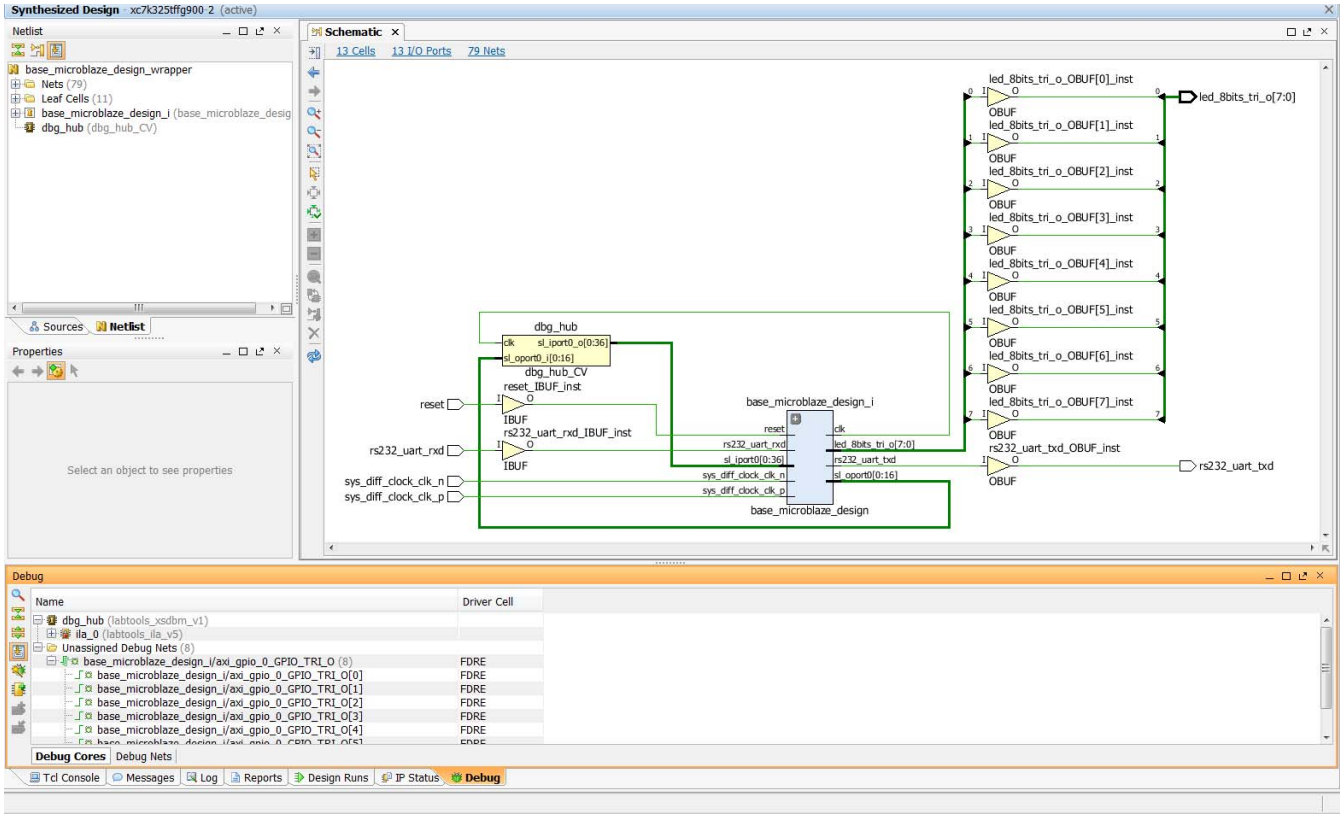


Figure 6-29: **Schematic and Debug Window in the Vivado IDE**

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 tool bar.

   Alternatively, you can also select **Tools > Set Up Debug**.

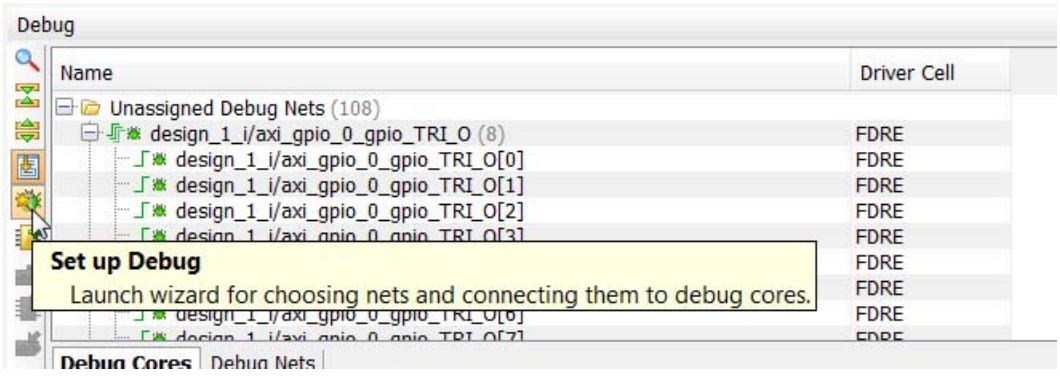


Figure 6-30: **Set Up Debug Button**

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

5. Click **Next**.

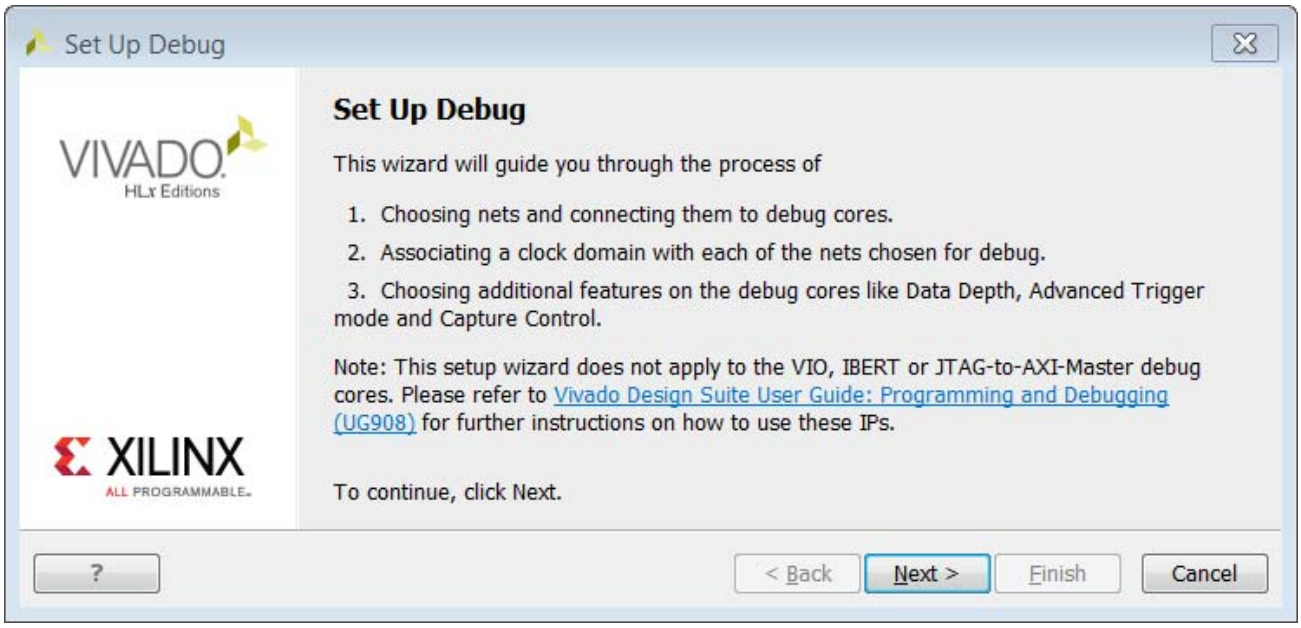


Figure 6-31: **Set Up Debug Wizard**

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

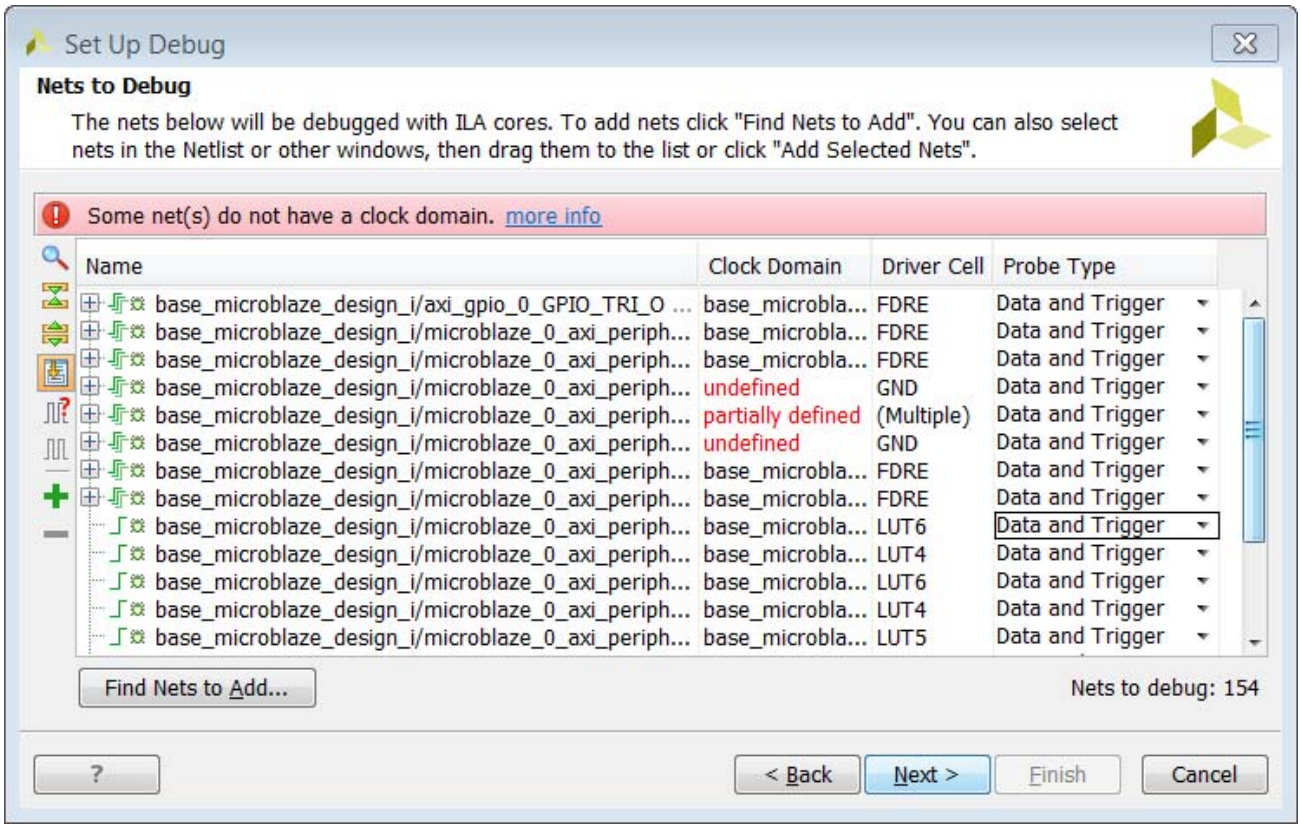


Figure 6-32: **Select Nets Marked for Debug**

6. 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.*

7. To associate a clock domain to the signals that have an undefined **Clock Domain**, select the nets, right-click, and choose **Select Clock Domain** as shown in Figure 6-33, page 120.

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

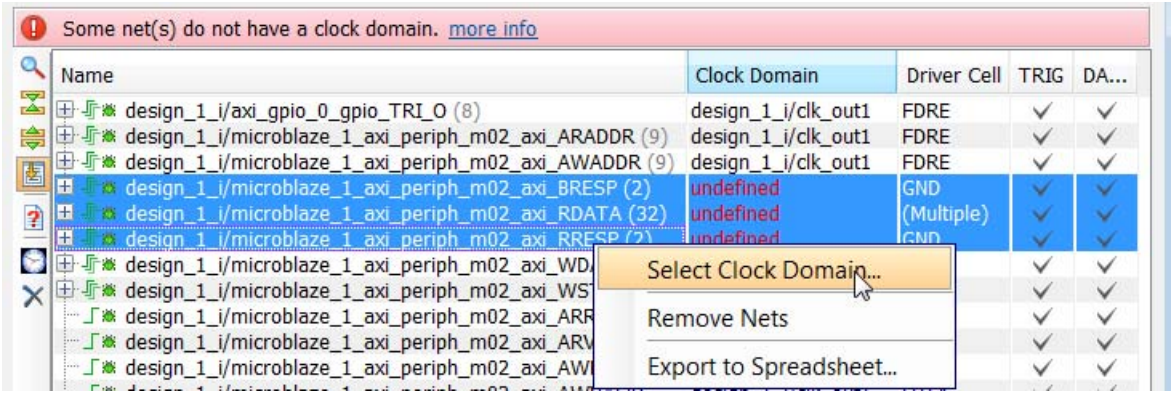


Figure 6-33: **Select Clock Domain Command**

8. In the Select Clock Domain dialog box, select the clock and click **OK**.

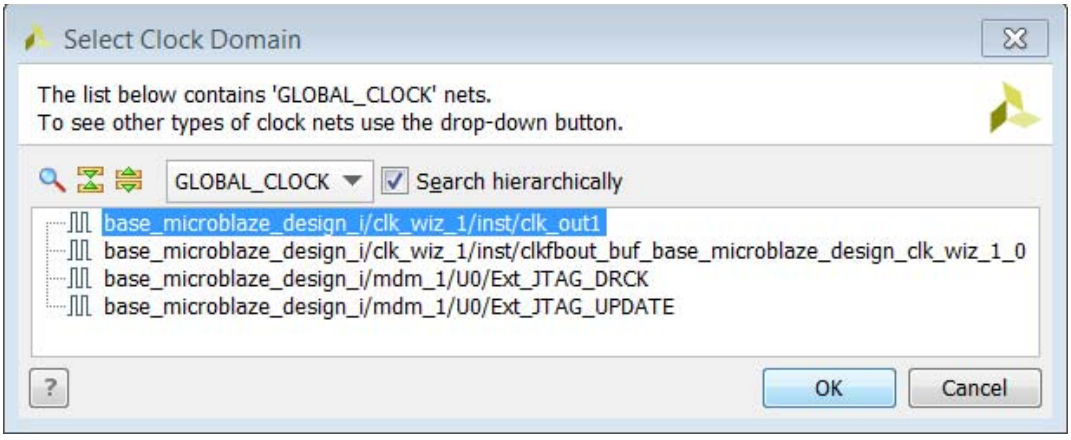


Figure 6-34: **Select Clock Domain Dialog Box**

9. In the Specify Nets to Debug dialog box, click **Next**.
10. In the ILA Core Options page, shown in Figure 6-35, page 121, select the appropriate options for triggering and capturing data, and click **Next**.

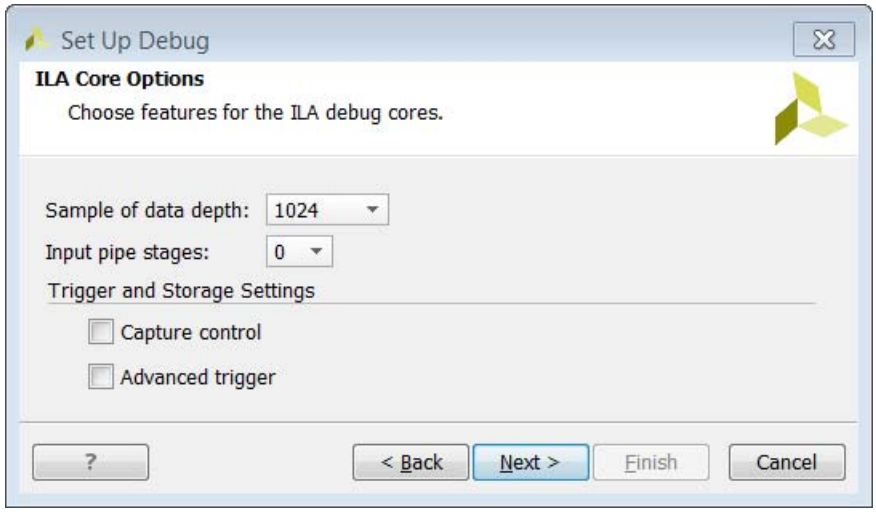


Figure 6-35: **Setup the Trigger and Capture Modes in the ILA**

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 runtime.

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.

11. In the Summary page, verify that all the information looks correct, and click **Finish**.

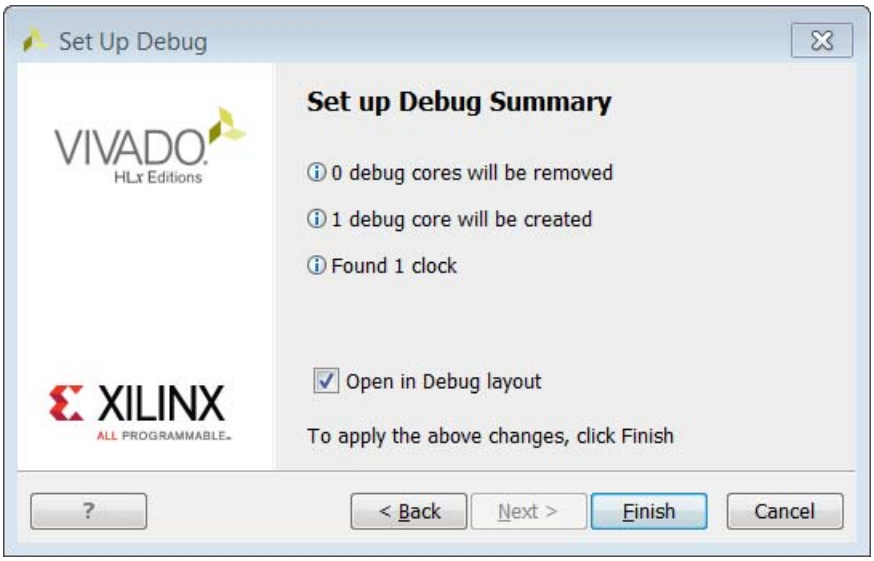


Figure 6-36: **Setup Debug Summary**

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.

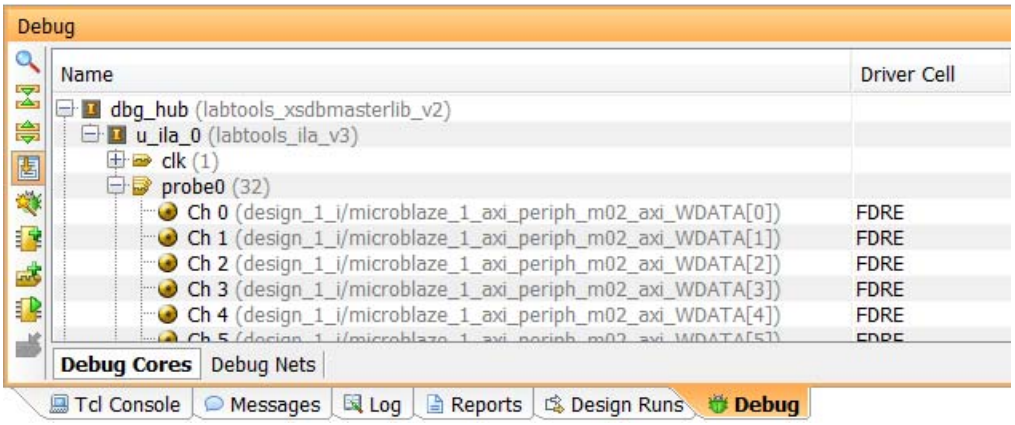


Figure 6-37: **Debug Window after ILA Insertion**

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

12. In the **Flow Navigator > Program and Debug**, 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.

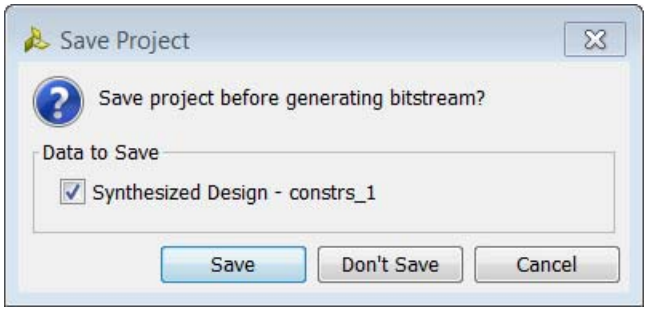


Figure 6-38: **Save Modified Constraints after ILA Insertion**

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 Figure 6-39, page 123.

C:/temp/my_migration/my_migration.srcs/constrs_1/new/design_1_wrapper.xdc

```
1 create_debug_core u_ila_0 labtools_ila_v3
2 set_property ALL_PROBE_SAME_MU true [get_debug_cores u_ila_0]
3 set_property ALL_PROBE_SAME_MU_CNT 4 [get_debug_cores u_ila_0]
4 set_property C_ADV_TRIGGER true [get_debug_cores u_ila_0]
5 set_property C_DATA_DEPTH 1024 [get_debug_cores u_ila_0]
6 set_property C_EN_STRG_QUAL true [get_debug_cores u_ila_0]
7 set_property C_INPUT_PIPE_STAGES 0 [get_debug_cores u_ila_0]
8 set_property C_TRIGIN_EN false [get_debug_cores u_ila_0]
9 set_property C_TRIGOUT_EN false [get_debug_cores u_ila_0]
10 set_property port_width 1 [get_debug_ports u_ila_0/clk]
11 connect_debug_port u_ila_0/clk [get_nets [list design_1_i/clk_out1]]
12 set_property port_width 32 [get_debug_ports u_ila_0/probe0]
```

Figure 6-39: **XDC Constraints for ILA Core Insertion**

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) [Ref 7].

Using Tcl Scripts to Create Block Designs within Projects

Overview

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.

Creating a Design in the Vivado IDE

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.

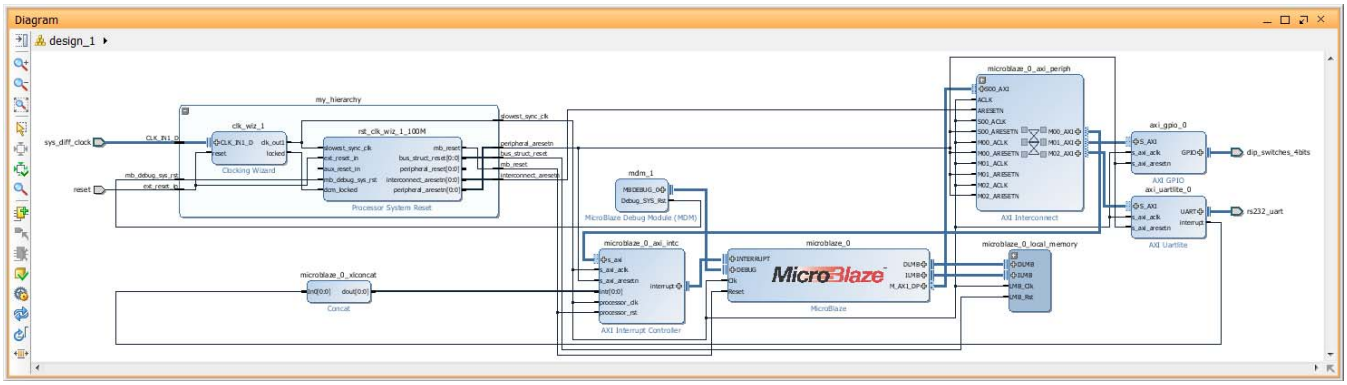


Figure 7-1: **Block Design Canvas**

With the block design open, from the menu select **File > Export > Export Block Design**.

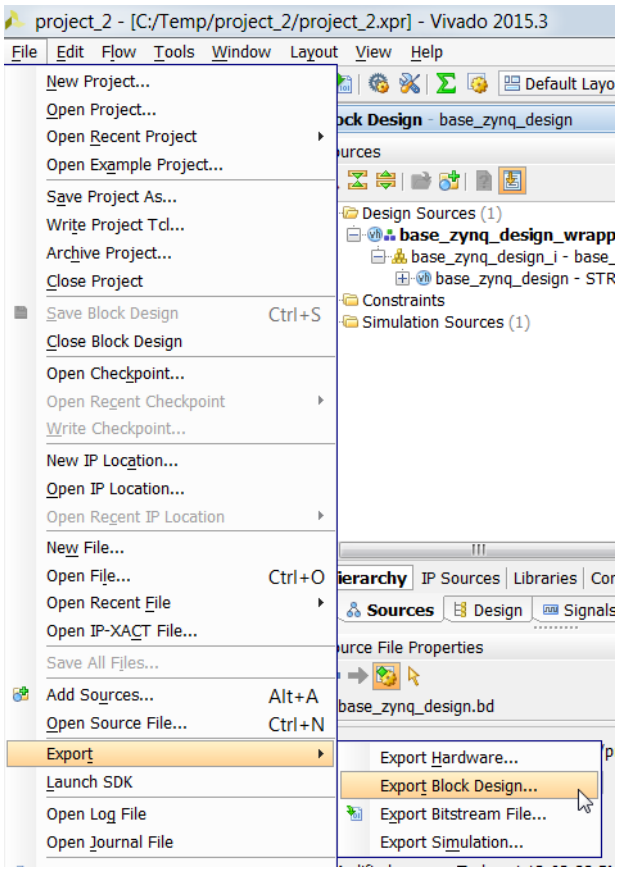


Figure 7-2: **Exporting a Block Design**

Specify the name and location of the Tcl file in the Export Block Design dialog box.

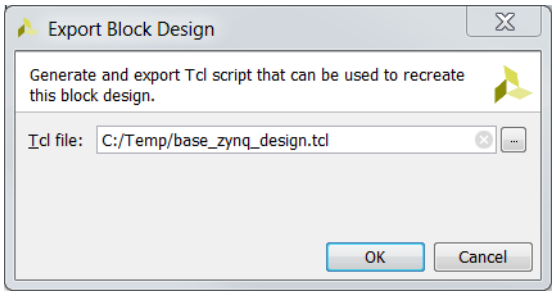


Figure 7-3: **Export Block Design dialog box**

Alternatively you can also type the following 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.

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.*

Saving the Vivado Project Information in a Tcl File

The overall project settings can be saved by selecting **File > Write Project Tcl**.

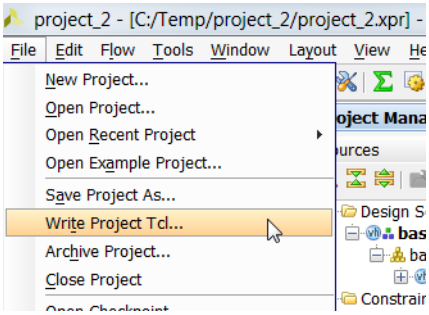


Figure 7-4: **Writing a Tcl file for the Project**

In the Write Project to Tcl dialog box specify the name and location of the Tcl file and select any other options desired.

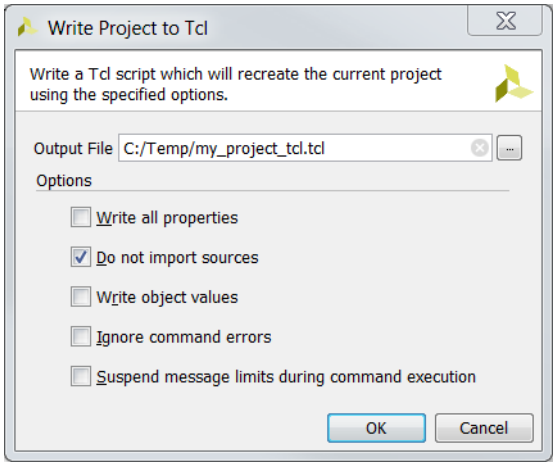


Figure 7-5: **Write Project to Tcl dialog box**

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>
```

For a Vivado project that consists of a block diagram, the Tcl file generated from `write_project_tcl` command could look like the following:

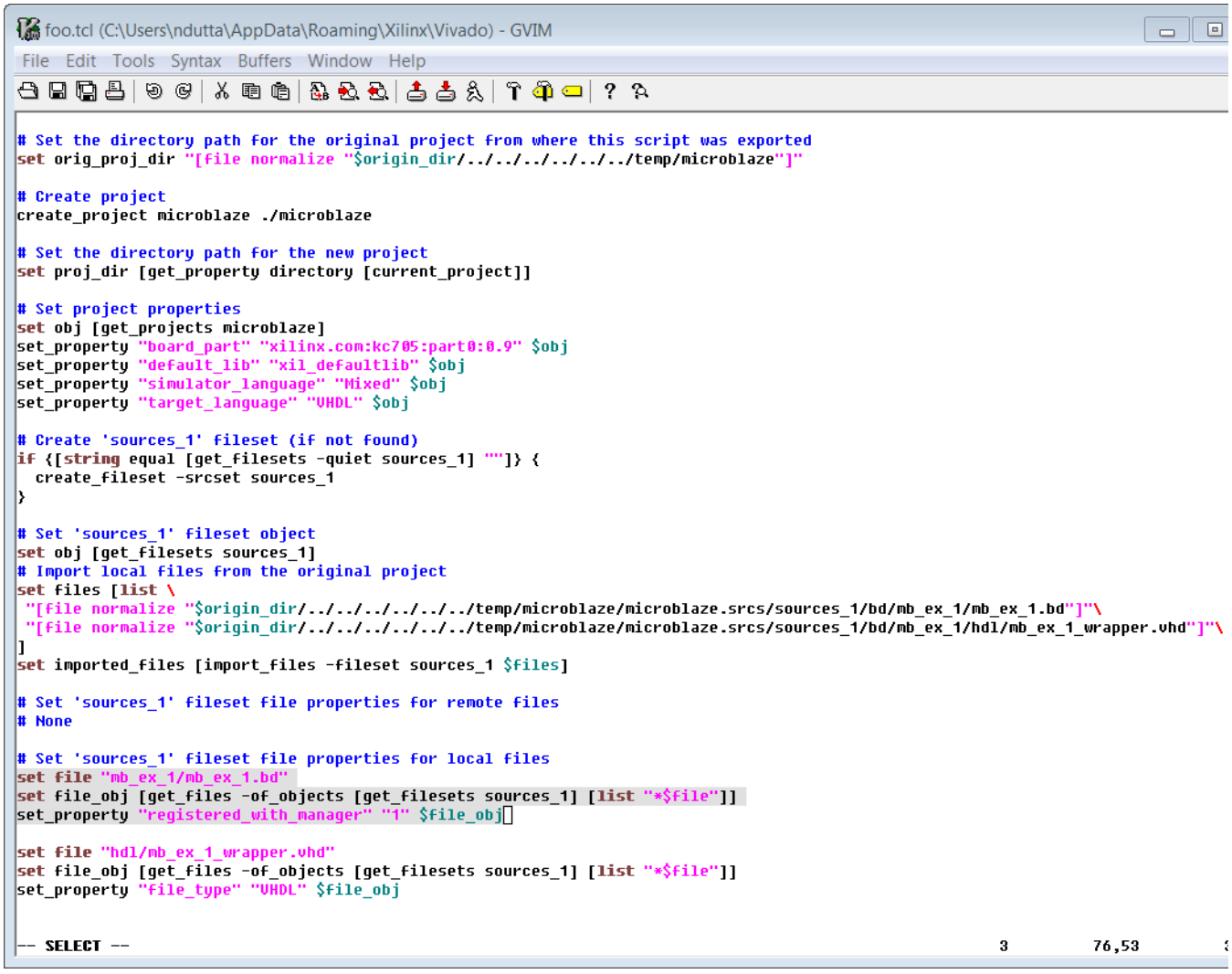


Figure 7-6: **Code Snippet from the Tcl File Generated by using the write_project_tcl Command**

In the preceding Tcl file, the block design file (`.bd`) is read explicitly as shown by the highlighted code.

There are cases in which you may want to re-create the block design, rather than simply read the block design file. In this case, you will need to modify the Tcl file generated by the `write_project_tcl` command as shown in Figure 7-7, page 128:

TIP: *If you choose to simply read the existing block design file, and not re-create the block design, then the Tcl file does not need to be modified.*

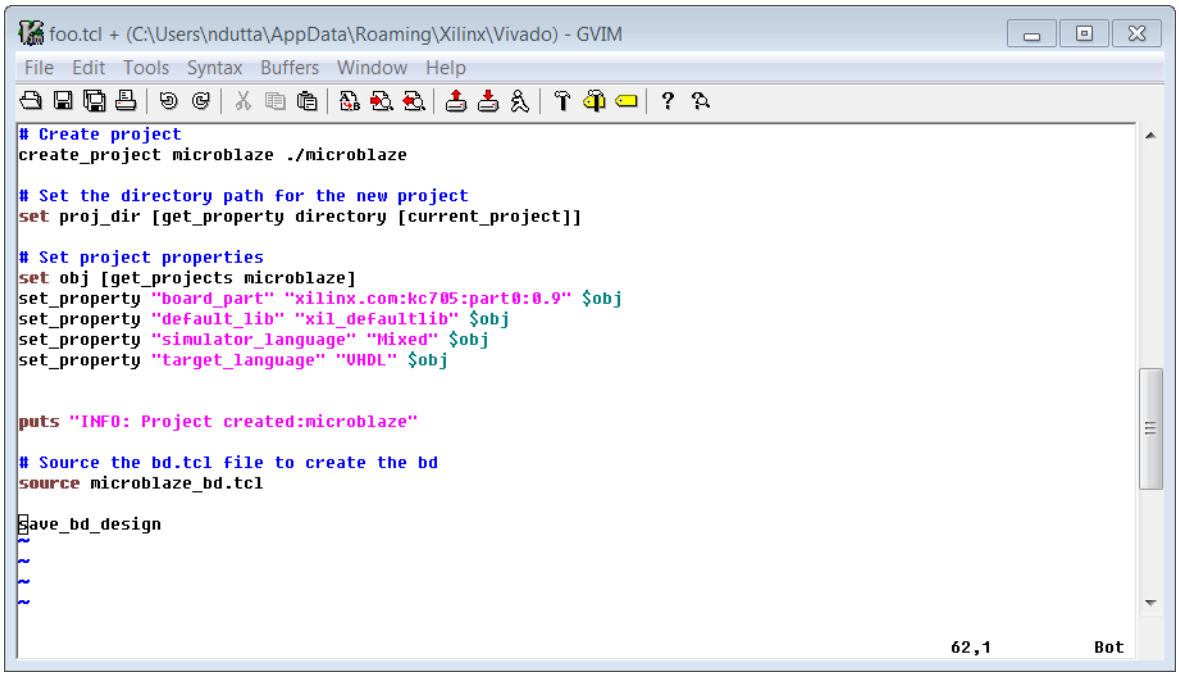


Figure 7-7: **Code Snippet from the Tcl File to Recreate the Block Design using the Output File**

As can be seen in the figure above, the Tcl file from the `write_project_tcl` file has been modified to source another Tcl script that was created using the `write_bd_tcl` command. The sourced block design Tcl script re-creates the block design every time the project Tcl script is run, rather than reading a block design file.

Using IP Integrator in Non-Project Mode

Overview

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 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 your script 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) [Ref 2].

IMPORTANT: *Because the Non-Project Mode does not support out-of-context synthesis, a block design with Synthesis Mode set to Out-of-context per IP or Out-of-context per BD cannot be added to a non-project flow. Only block designs with Synthesis Mode set to Global are supported by the non-project flow.*

Creating a Flow in Non-Project Mode

The recommended approach for running Non-Project Mode mode is to launch the Vivado Design Suite in Tcl mode, or to create a Tcl script and run the tool in batch mode:

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

In non-project mode, set your project options as shown below:

```
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 project has been created the source file for the block design can be added to the project.

This can be done in two different ways. First, assuming that there is an existing block design with the entire directory structure of the block design intact, you can add the block design using the `read_bd` tcl command as follows:

```
read_bd <path to the bd file>
```

CAUTION! *The project settings (board, part, and user repository) of the new design must match the project 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.

```
read_verilog <top-level>.v
read_xdc <top-level>.xdc
```

You can also create top-level HDL wrapper file using the command below because a BD source cannot be synthesized directly.

```
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.

For a MicroBlaze™-based processor design you need to add and associate an ELF with the MicroBlaze instance in the block design. This will populate the BRAM 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]
```

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.

TIP: *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 Hardware Design (UG898) [Ref 5] 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
```

See the *Vivado Design Suite Tcl Command Reference Guide* (UG835) [Ref 1] for more on the write_sysdef or write_hwdef commands.

Non-Project Script

The following is a sample script for creating a block design in Non-Project Mode.

IMPORTANT: *Out-of-Context synthesis as described in Generating Output Products in Chapter 4 is not supported 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
update_compile_order -fileset sources_1
update_compile_order -fileset sim_1
synth_design -top mb_ex_1_wrapper
opt_design
place_design
route_design
write_bitstream top
write_mem_info ./top.mmi
file mkdir c:/temp/export_hw_np_mode/sdk
write_sysdef -hwdef mb_ex_1_wrapper.hwdef -bitfile top.bit -file mb_ex_1_wrapper.hdf
```

Updating Designs for a New Release

Overview

As you upgrade your Vivado® Design Suite to the latest release, you must upgrade the block designs created in the Vivado IP Integrator as well.

- The IP version numbers 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.

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 > Open Project** 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**.

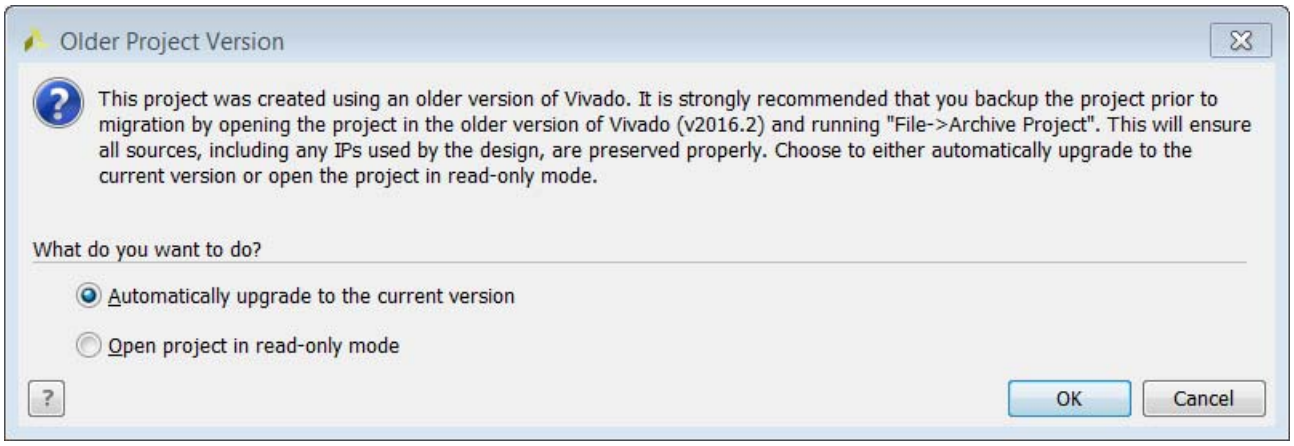


Figure 9-1: **Open Project in Read-Only Mode**

The Project is Read-Only dialog box opens.

4. Select **Save Project As** as shown in the following figure.

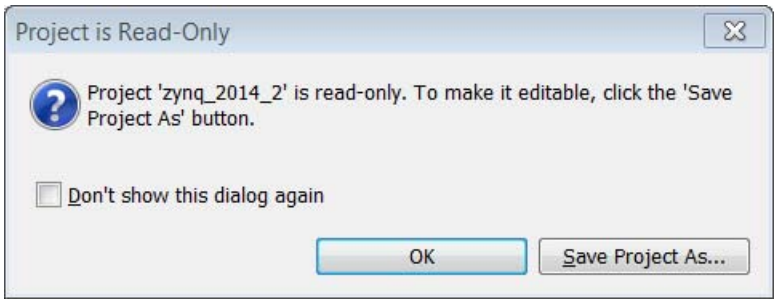


Figure 9-2: **Save Project As**

5. When the Save Project As dialog box opens, as shown in the following figure, type the name of the project, and click **OK**.

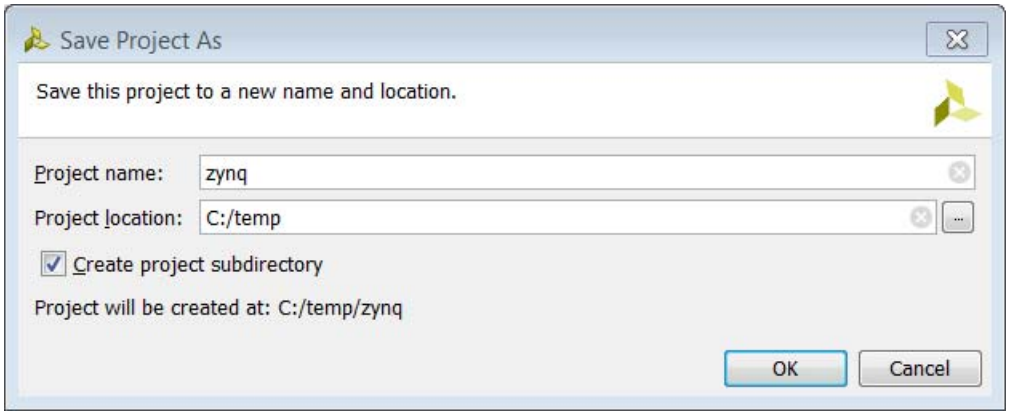


Figure 9-3: **Specify Project Name**

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.

Figure 9-4: **Project Upgraded Dialog Box**

6. Click **Report IP Status**.

   Alternatively, from the menu select **Tools > Report > Report IP Status**.

7. If any of the IP in the design has gone through a major version change, then the following message opens. Click **OK**.

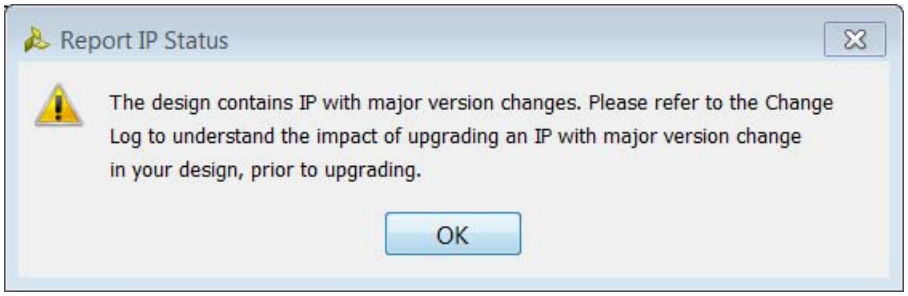


Figure 9-5: **Report IP Status Results**

8. In the IP Status window, look at the different columns and familiarize yourself with the IP Status report. Expand the block design by clicking on the **+** sign and look at the changes that the IP cores in the block design.

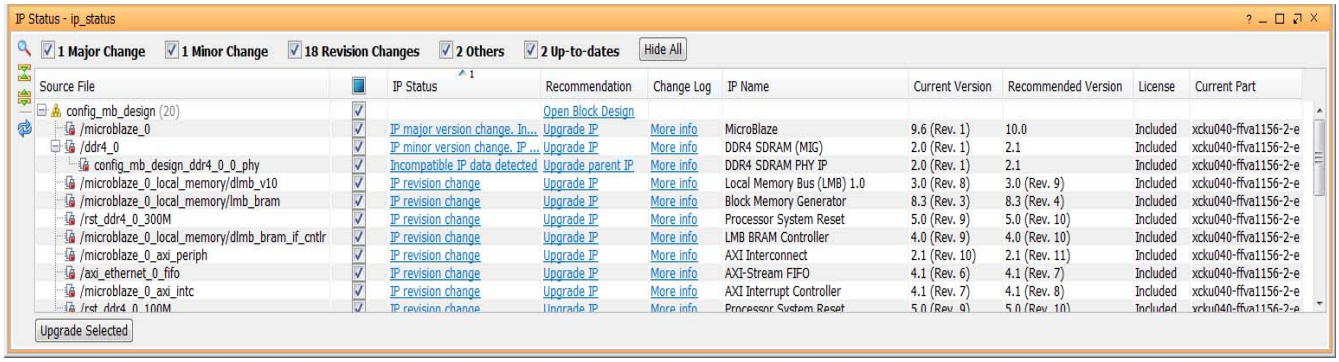


Figure 9-6: **IP Status Report**

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)

9. Click the **More info** link in the Change Log column to see a description of the change.

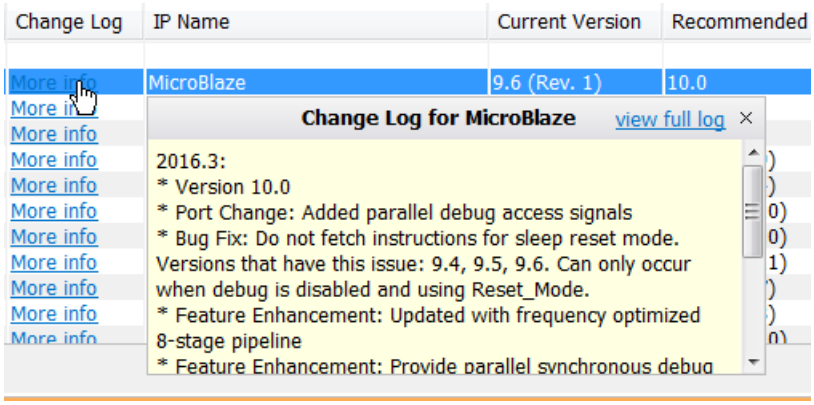


Figure 9-7: **Inspect the Change Log by in More Info link**

The Recommendation column also suggests that you need to understand what the changes are before selecting the IP for upgrade.

10. When you understand the changes and the potential impact on your design, you can click **Upgrade Selected**.

    The Upgrade IP dialog box opens to confirm that you want to proceed with upgrade.

TIP: *You cannot select individual IP of a block design to upgrade, and let the others remain in their current versions. You must upgrade all IP in the block design at the same time.*

11. Click **OK**.

    When the upgrade process is complete, a Critical Messages dialog box opens, such as the one in the following figure, informing you about any critical issues to which you need to pay attention.

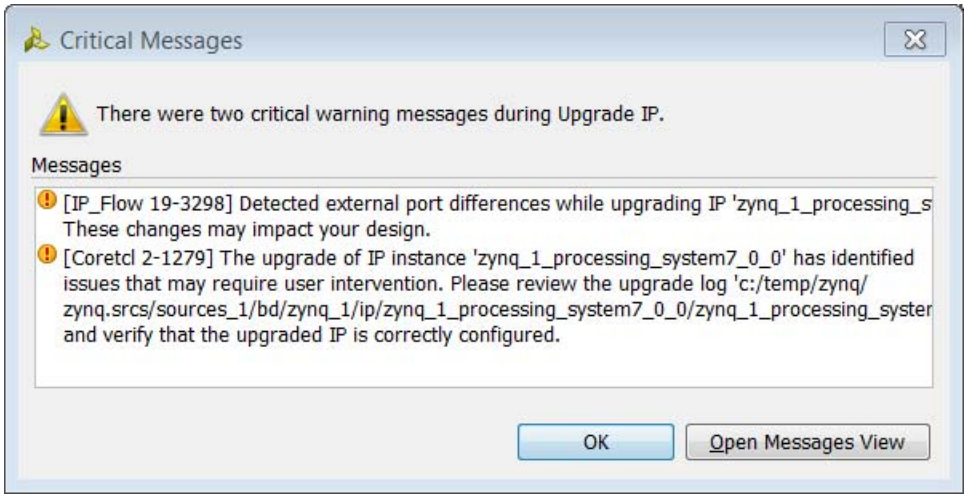


Figure 9-8: **Critical Messages Dialog Box**

12. Review any critical warnings and other messages that may be flagged as a part of the upgrade. Click **OK**.

    If there are multiple diagrams in the design, the IP Status window shows the status of IP in all the diagrams as shown in Figure 9-9, page 137.

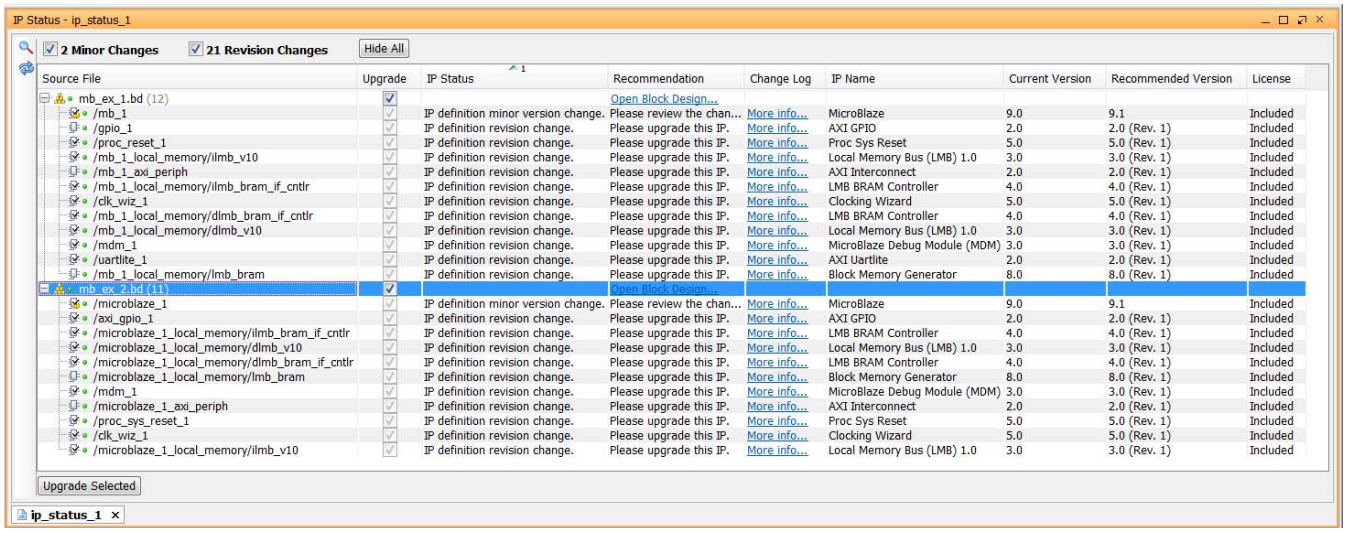


Figure 9-9: **IP Status Window for Multiple Diagrams**

Running Design Rule Checks

From the toolbar, click **Validate Design**, as shown in the following figure.

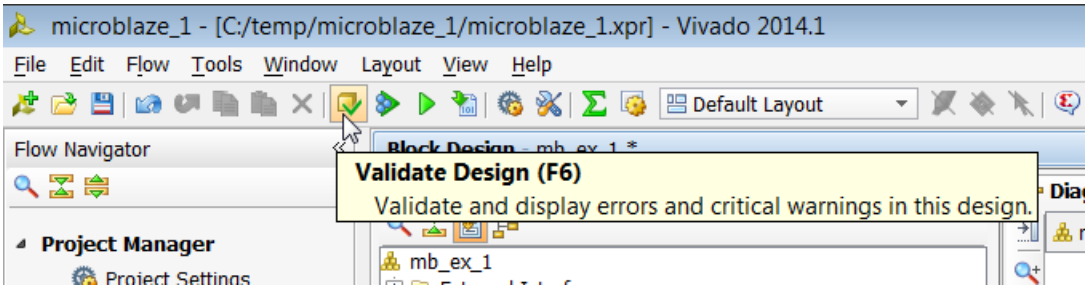


Figure 9-10: **Validate Design**

You can also do this by clicking the **Validate Design** button in the block design toolbar.

Regenerating Output Products

1. In the Sources window, right-click the block diagram, and select **Generate Output Products**.

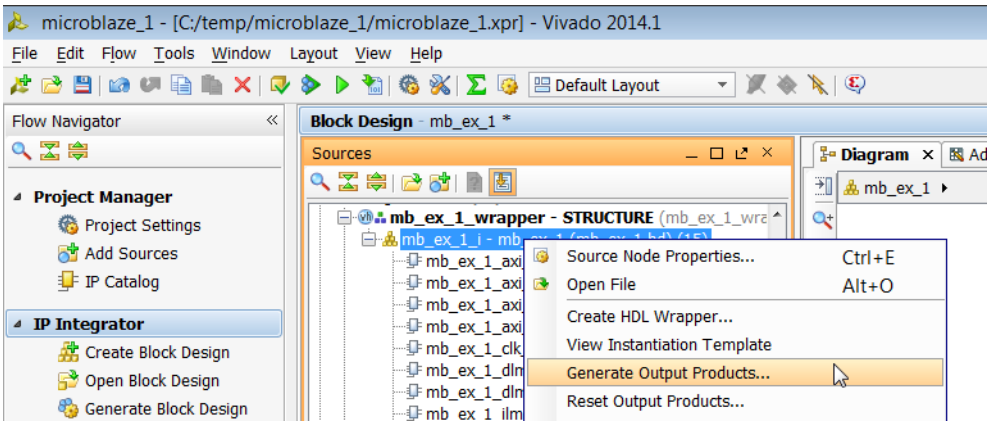


Figure 9-11: **Generate Output Products Command**

Alternately, in the **Flow Navigator > IP Integrator**, click the **Generate Block Design**.

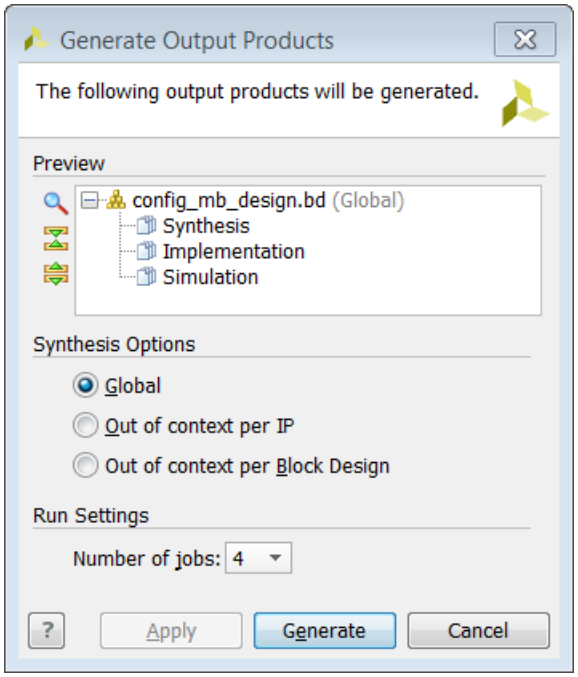


Figure 9-12: **Generate Output Products Dialog Box**

2. In the Generate Output Products dialog box, click **Generate**.

Creating or Changing the HDL Wrapper

If you previously created an HDL wrapper in the previous version of the design, you may want to re-create it to reconcile any design changes. If you had chosen the option to let the Vivado tools create and manage the top-level wrapper for you, then the wrapper file will be updated as a part of generating the block design or generating output products as defined in the previous section. If you modified the HDL wrapper manually, then you will need to manually make any updates that may be necessary in the HDL wrapper.

1. In the Sources window, shown in the following figure, right-click the block diagram, and select **Create HDL Wrapper**.

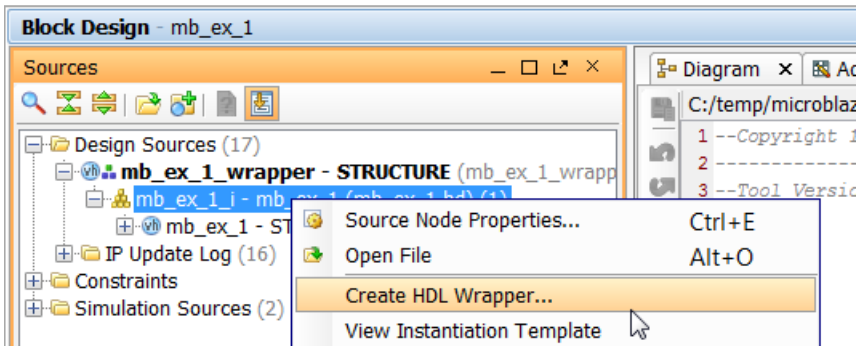


Figure 9-13: **Create HDL Wrapper Command**

The Create HDL Wrapper dialog box opens. You have two choices to make at this point. You can create a wrapper file that can be edited, or you can let the Vivado tools manage the wrapper file for you.

2. Make your selection from the dialog box shown in the following figure, and click **OK**.

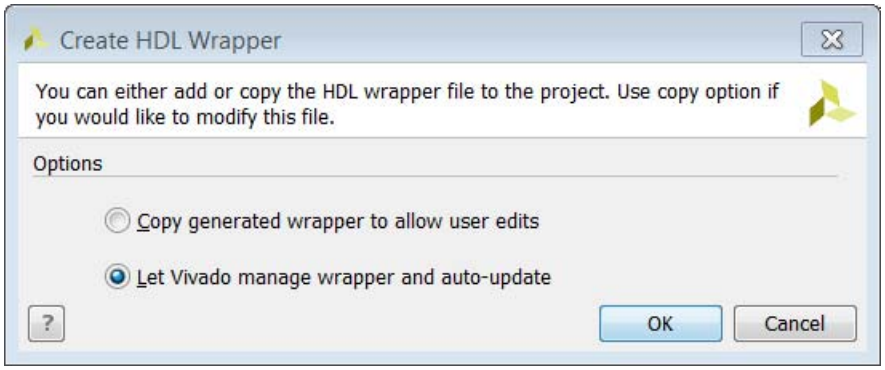


Figure 9-14: **Create HDL Wrapper Dialog Box**

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. You can use the following script as a guideline to upgrade the IP in the block diagram.

```
# 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
```

XILINX
ALL PROGRAMMABLE™

Chapter 10

Using the Platform Board Flow in IP Integrator

Overview

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 board. However, you also have the ability to define custom board files to add to the tool. Refer to 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 window called the Board window. When you use this window to select the desired components and the Designer Assistance offered by IP Integrator, you can easily connect your block design to the board components of your choosing. All the I/O constraints are automatically generated as a part of using this flow.

User-defined or third-party Board Interface files, and associated files, can 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"]
```

Select 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.

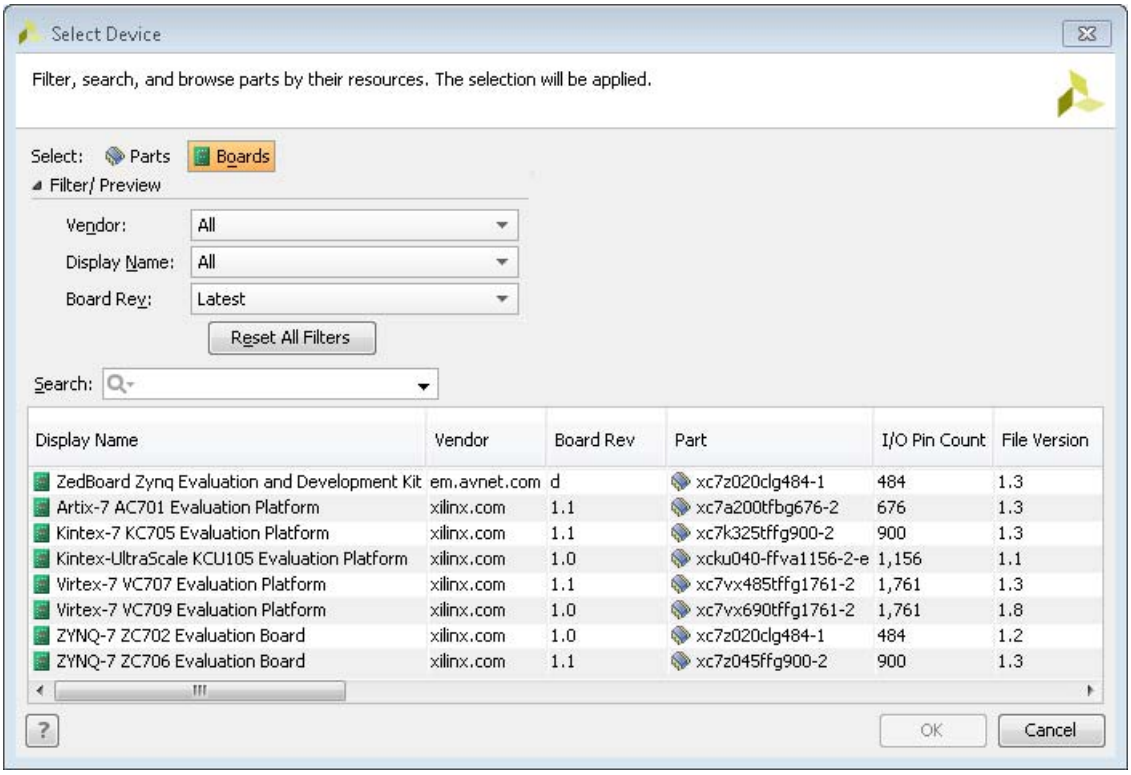


Figure 10-1: **Select a Target Board**

You can filter the list of available boards based on **Vendor**, **Display Name**, and **Board Revision**.

- **Vendor**: Specifies the board manufacturer.
- **Display 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.

Create a Block Design to use the Board Flow

The real power of the board flow can be seen in the IP Integrator tool.

From **Flow Navigator > IP Integrator**, start a new block design by clicking **Create Block Design**.

As the design canvas opens, you see a **Board** window, as shown in the following figure.

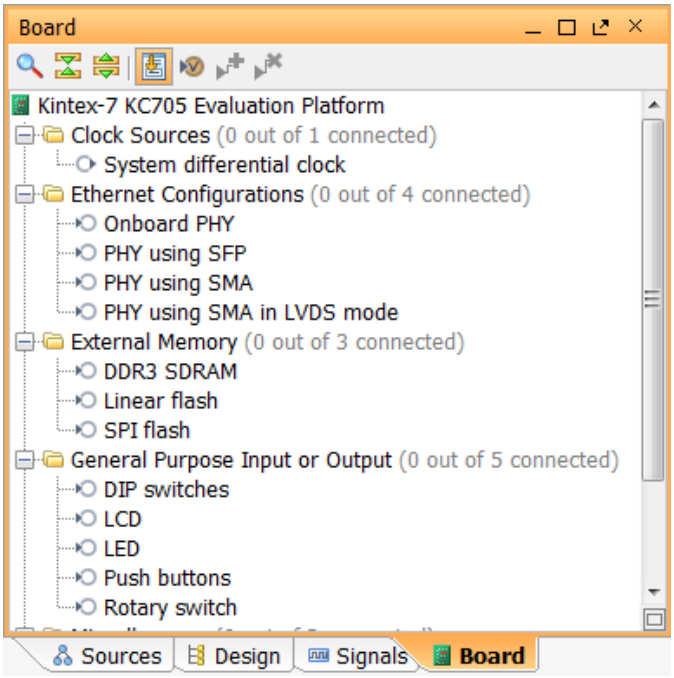


Figure 10-2: **Board Window**

This Board window lists all the possible components for an evaluation board (in the preceding figure the KC705 board is displayed). 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 on 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 an drop the Linear Flash component under the External Memory folder, on the IPI canvas, the AXI EMC IP is instantiated and the interface called EMC_INTF is connected. See Figure 10-3, page 144.

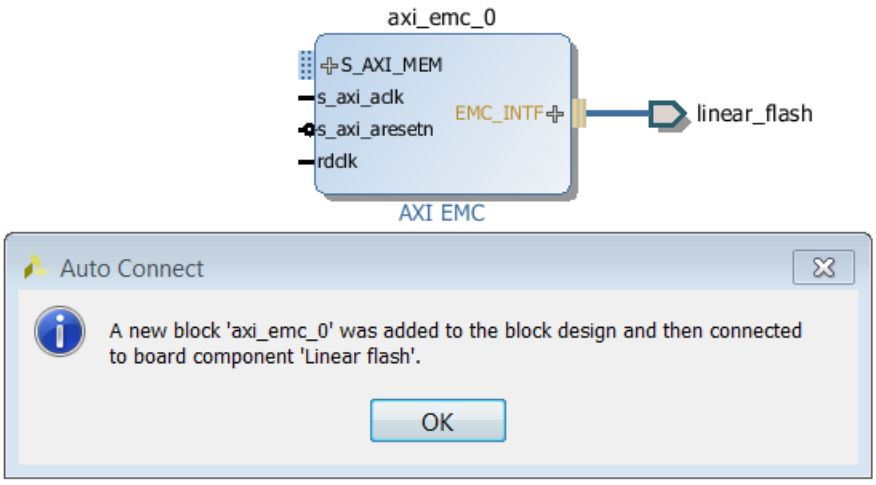


Figure 10-3: **Dragging and Dropping an Interface on the Block Design Canvas**

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.

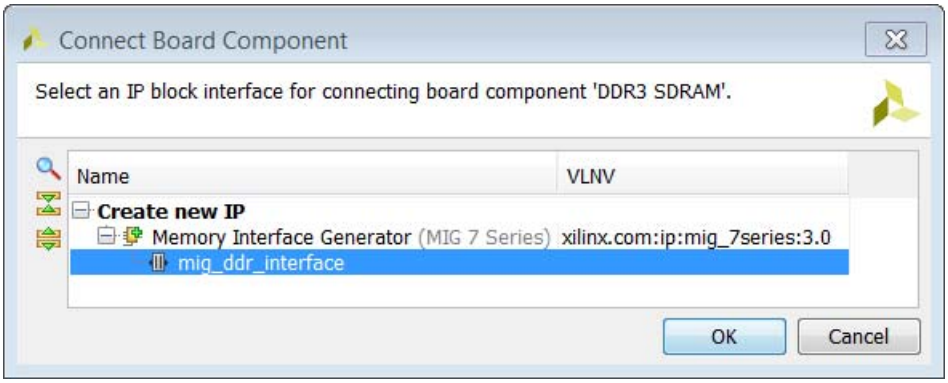


Figure 10-4: **Connect Board Component Dialog Box**

mig_ddr_interface is selected by default. If there are multiple interfaces listed under the IP then 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.

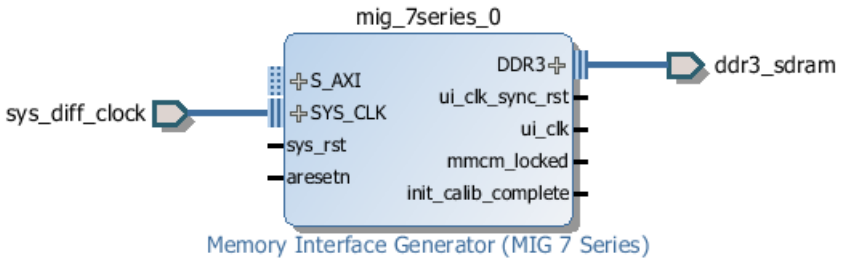


Figure 10-5: **IP instantiated, Configured, and Connected to Interfaces on the Diagram Canvas**

As an interface is connected, that particular interface now shows up as a shaded circle in the Board window.

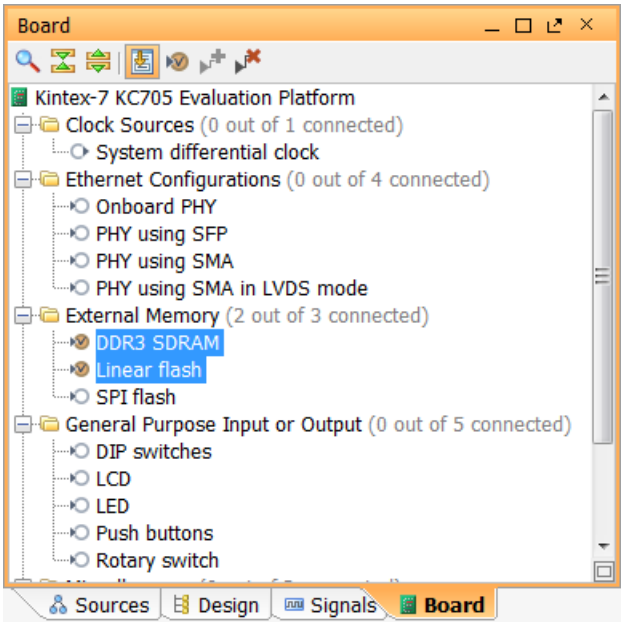


Figure 10-6: **Board Window after Connecting to an Interface**

A component can also be connected using the **Auto Connect** command.

To do this, select and right-click the component and from the menu, select **Auto Connect**.

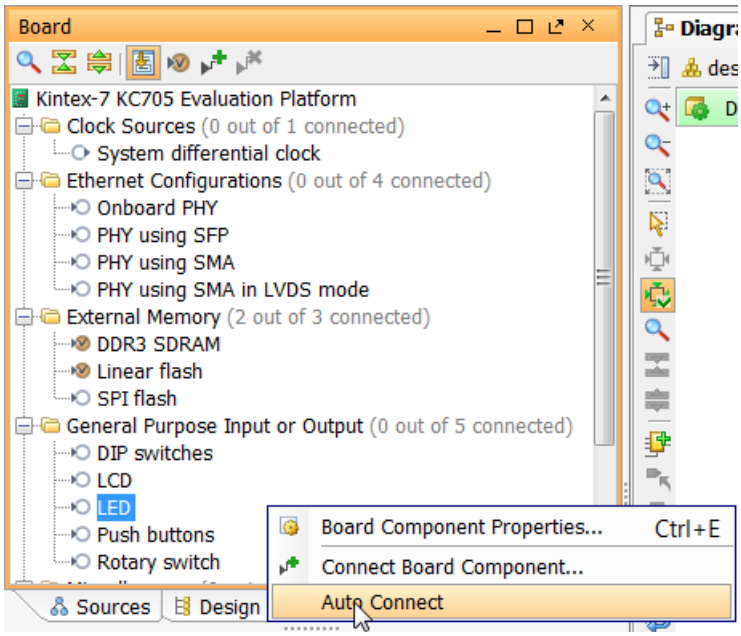


Figure 10-7: **Auto Connect Command**

You will 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 Figure 10-8, page 146.

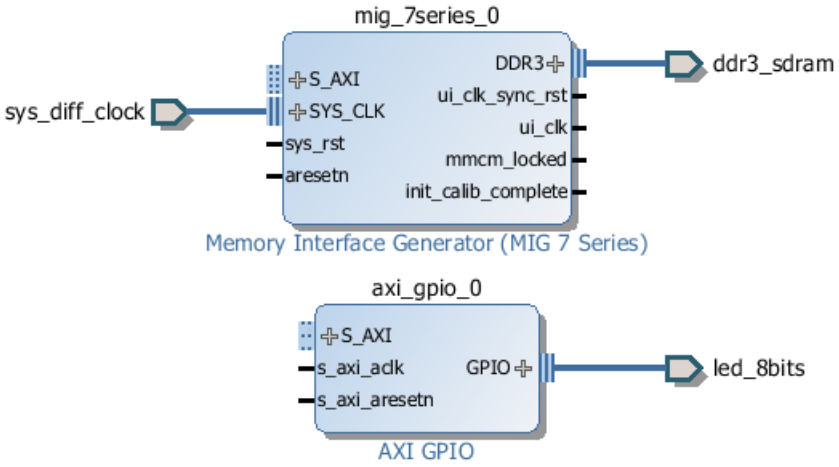


Figure 10-8: **Instantiating an IP using Auto Connect**

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.

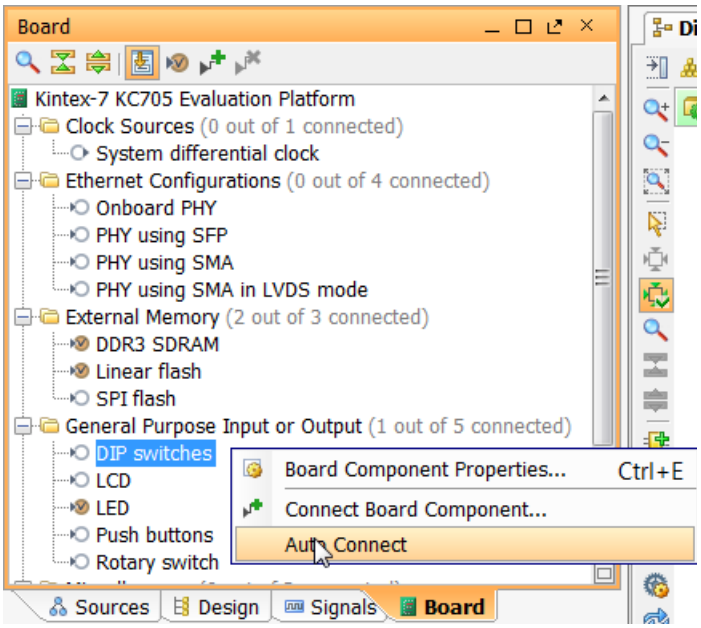


Figure 10-9: **GPIO Auto Connection**

The already instantiated GPIO is re-configured to use the second channel of the GPIO as shown in the following figure.

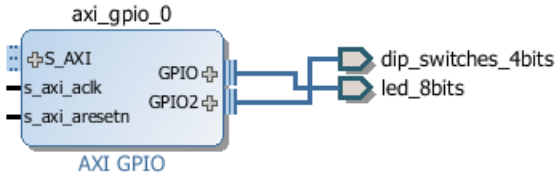


Figure 10-10: **GPIO IP Configured to Use the Second Channel**

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 as only one of these interfaces can be used on the target board. In this case, the following message will pop-up when the user tries to drag the other interface such as the SPI Flash on the block design canvas.

Figure 10-11: **Auto Connect Warning**

Complete Connections in the Block Design

After the desired interfaces are used 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.

To do this, right-click on the canvas and select add IP. From the IP catalog choose the processor, such as MicroBlaze™ processor, as shown in the following example.

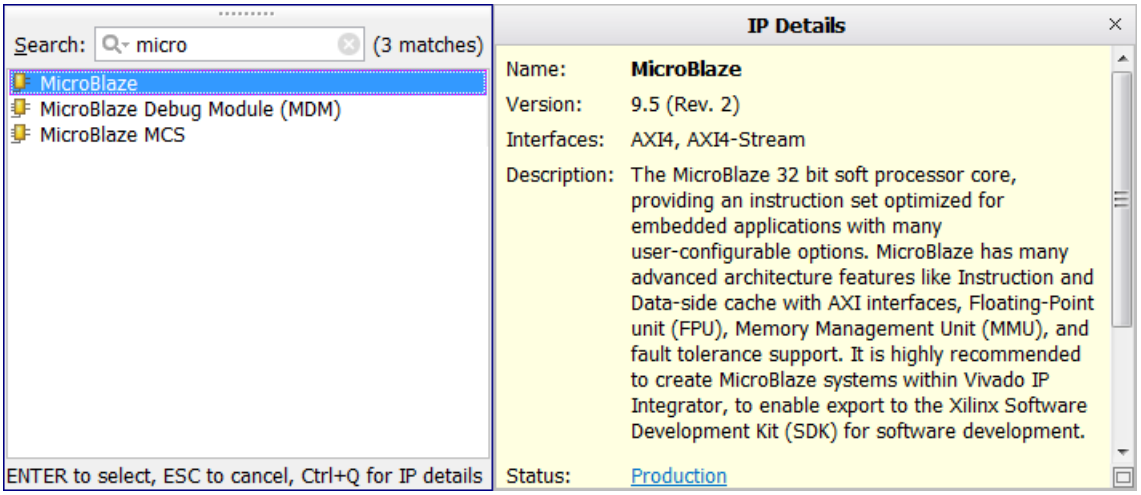


Figure 10-12: **Instantiate a Processor to Complete the Design**

As the processor is instantiated, Designer Assistance becomes available, as shown in the following figure.

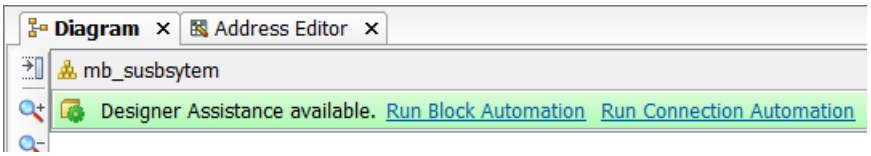


Figure 10-13: **Use Designer Assistance to Complete Connection**

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**.

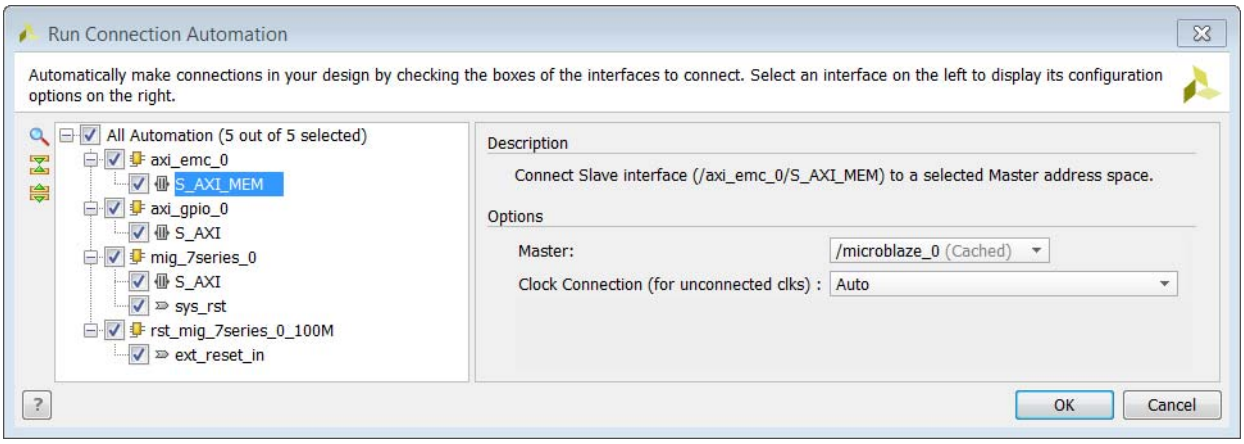


Figure 10-14: **Run Connection Automation to Complete Connections**

The rest of the process is the same as needed for designing in IP Integrator as described in of this document.

Using Third-Party Synthesis Tools in IP Integrator

Overview

Sometimes it is necessary to use a third-party synthesis tool as a part of the design flow. In this case, you will 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, and select the **Generate Output Products** command.

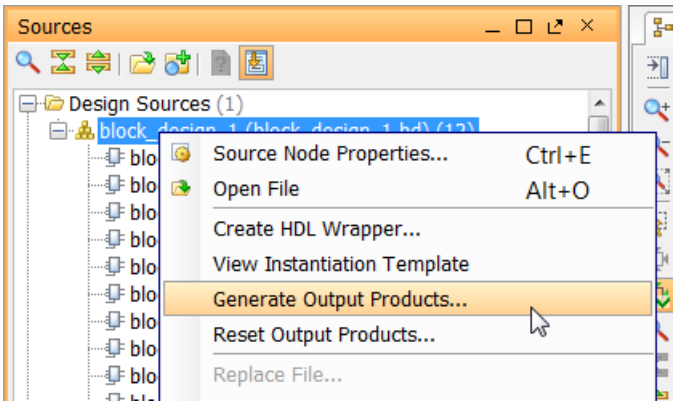


Figure 11-1: **Generate Output Products**

2. In the Generate Output Products dialog box, enable the **Out-of-Context per Block Design** option, as shown below. See Generating Output Products, page 67 for more information.

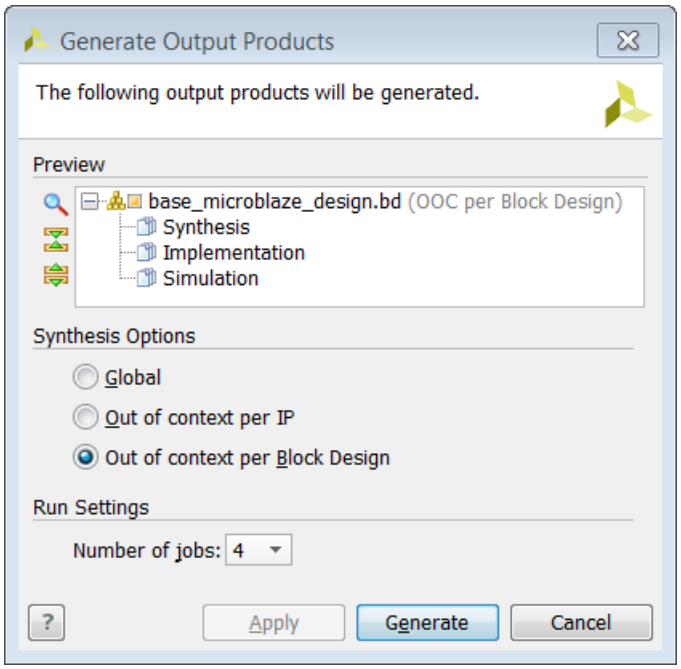


Figure 11-2: **Generate Output Products Dialog Box**

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 module. The Design Runs window also shows an Out-of-Context Module Run for the block design.

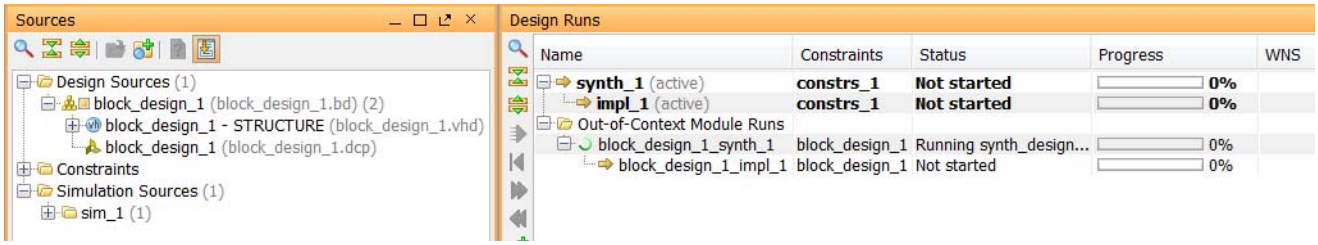


Figure 11-3: **Out-of-Context Module Runs**

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 Figure 11-4. These files are written to the following directory:

```
<project_name>/<project_name>.srcs/<sources_1>/<bd>/<block_design_name>
```

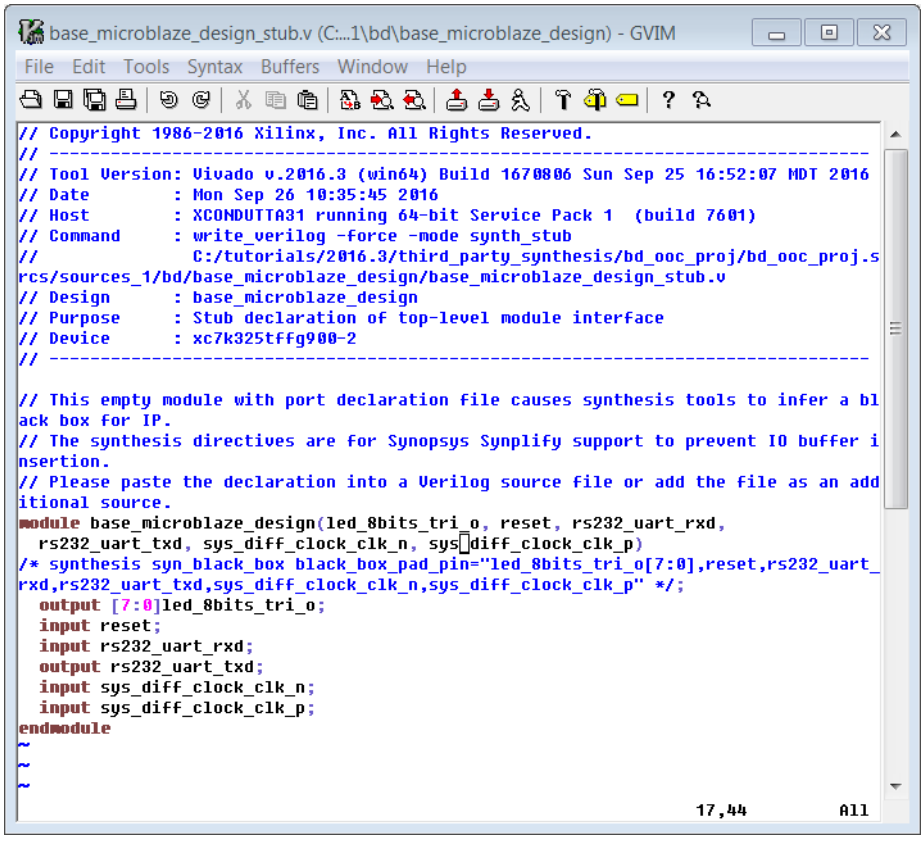


Figure 11-4: **Example Stub File**

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. Refer to this link in *Vivado Design Suite User Guide: System-Level Design Entry* (UG895) [Ref 3] for more information.

1. Create a new Vivado project, and select the **Post-synthesis Project** option in the New Project wizard.

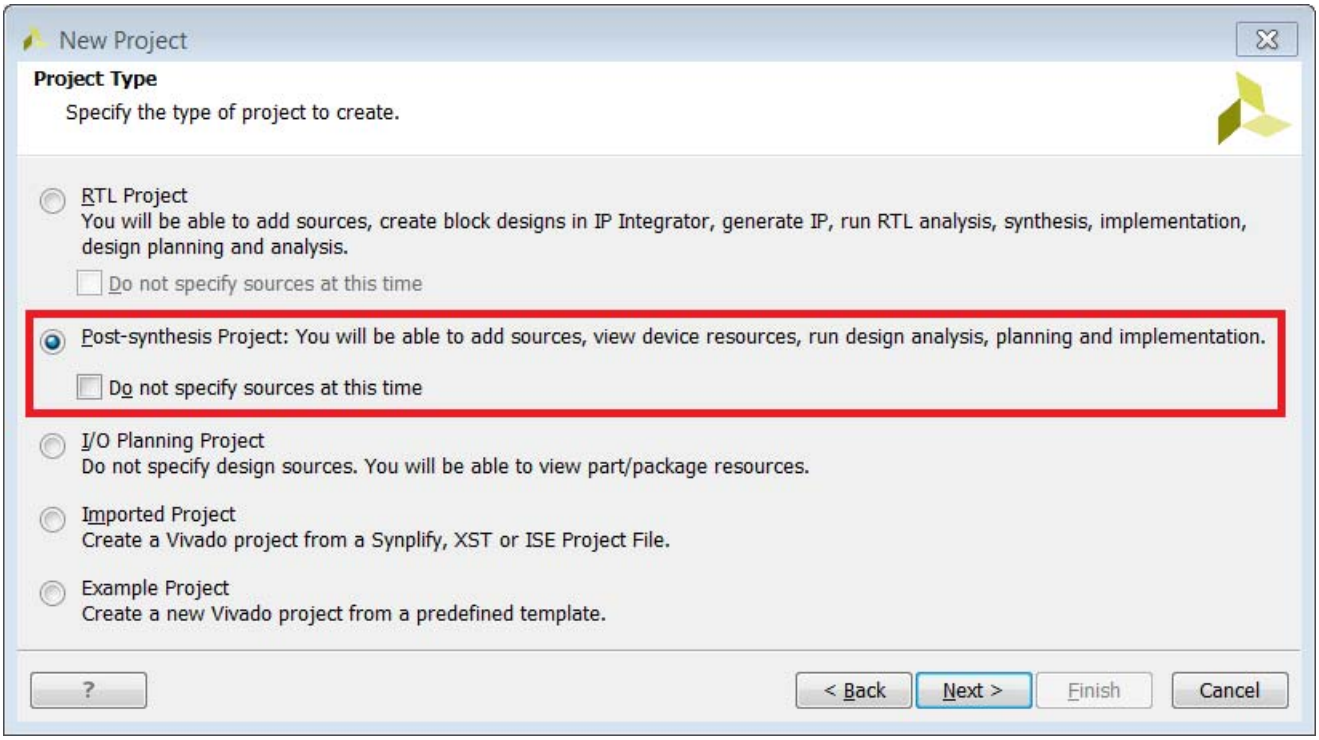


Figure 11-5: **Creating a Post-Synthesis Project**

Note: If the **Do not specify sources at this time** option is enabled, you can add design sources after project creation.

2. Click **Next**.

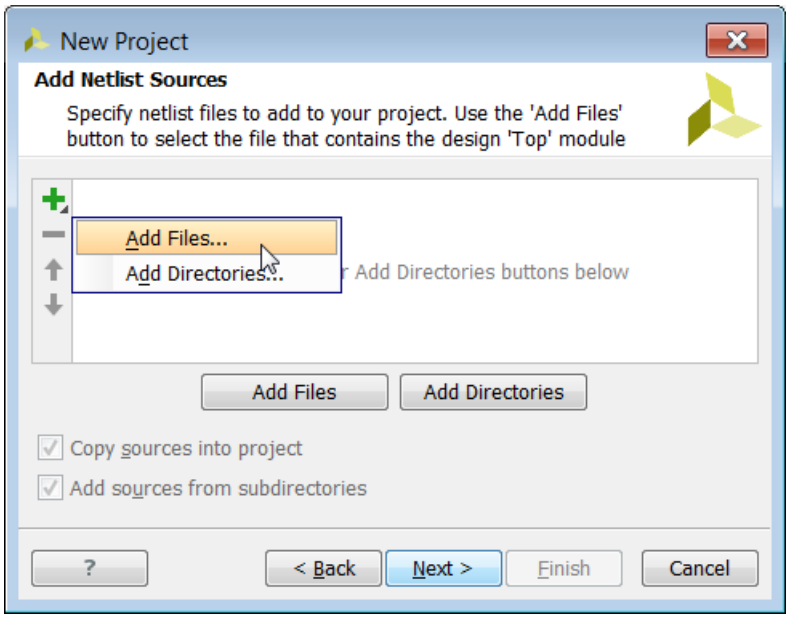


Figure 11-6: **Adding Files to the Post-Synthesis Project**

3. In the Add Netlist Sources Page click on the '**+**' sign to **Add Files**, as seen in Figure 11-6, page 154.

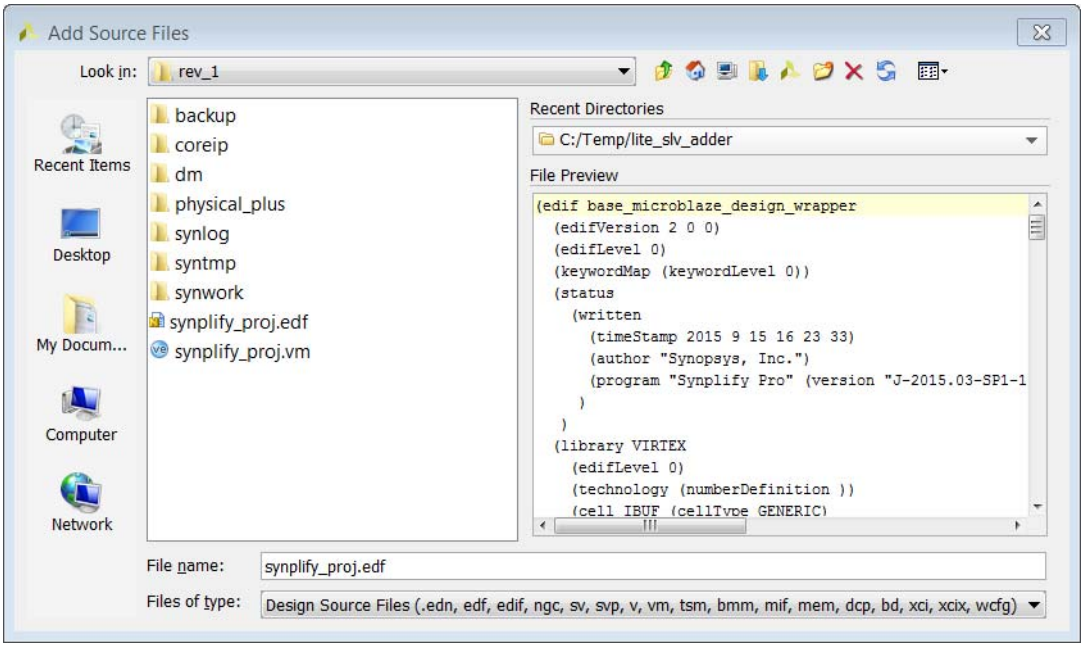


Figure 11-7: **Changing File Type in Add Sources File dialog box**

4. Select the EDIF netlist for the top-level design and click **OK**.
5. Select the block design (.bd file) and add that file to the project as well.

As the block design is added all the relevant constraints and the DCP file for the block design will be picked up by Vivado. The block design will not be re-synthesized. The constraints, however, will be reprocessed.

6. Click **Next**.

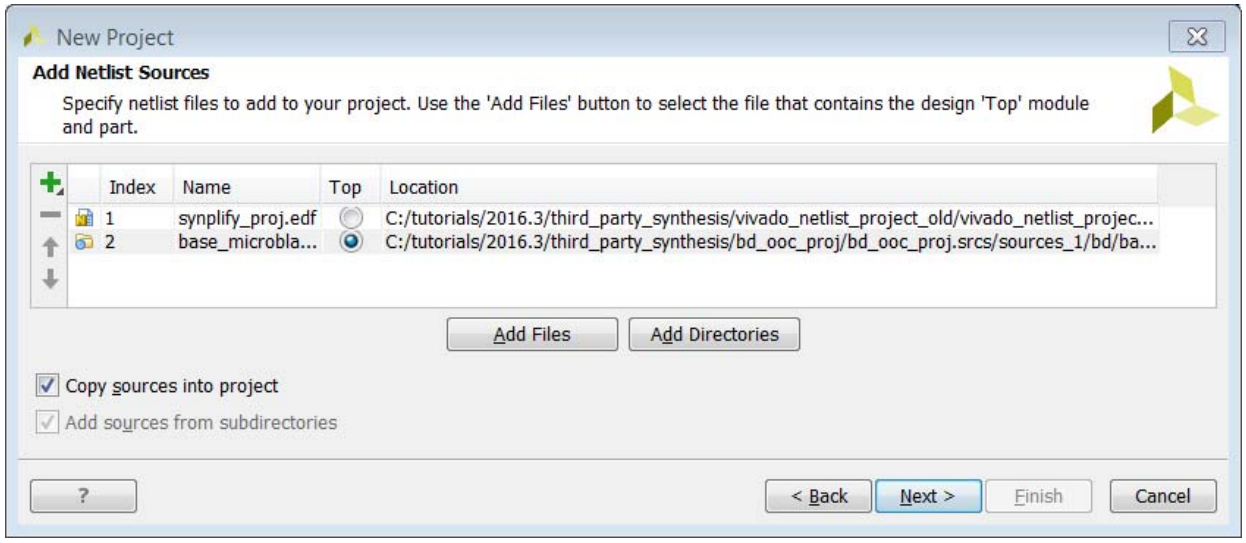


Figure 11-8: **Add the EDIF and BD files to the Project**

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 that 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 in order 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, refer to this link in the *Vivado Design Suite User Guide: Using Constraints* (UG903) [Ref 6].

A constraints file can be added to the project at the time it is created, as discussed previously, or by right-clicking in the Sources window and choosing **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, page 83 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.*

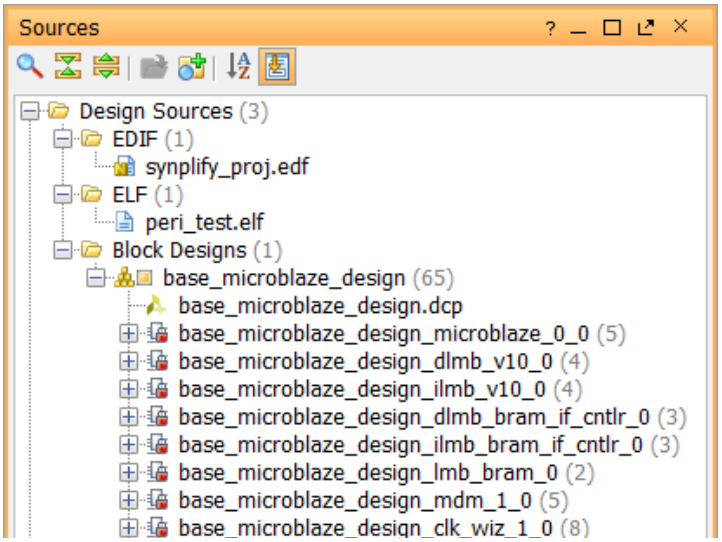


Figure 11-9: **ELF File in Project**

The added ELF file can be seen in the Sources window, as shown in Figure 11-9, page 156. After the ELF file has been 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.

   In the Source File Properties window click in the text field of the SCOPED_TO_CELLS and SCOPED_TO_REF properties to edit them.

2. Set the **SCOPED_TO_REF** property to the name of the block design.
3. Set the **SCOPED_TO_CELLS** property to the instance name of the embedded processor cell in the block design,

   In Figure 11-10 for example, SCOPED_TO_REF is `base_microblaze_design`, and SCOPED_TO_CELLS is `microblaze_0`.

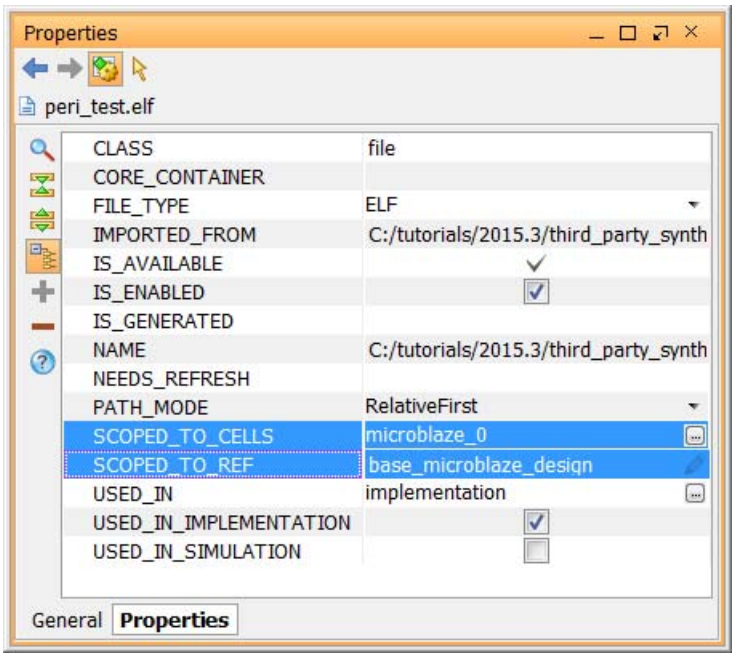


Figure 11-10: **Setting SCOPE Properties of ELF File**

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 on **Run Implementation** or **Generate Bitstream**.

   You will be prompted as needed by the Vivado tool to save constraints, and launch implementation.

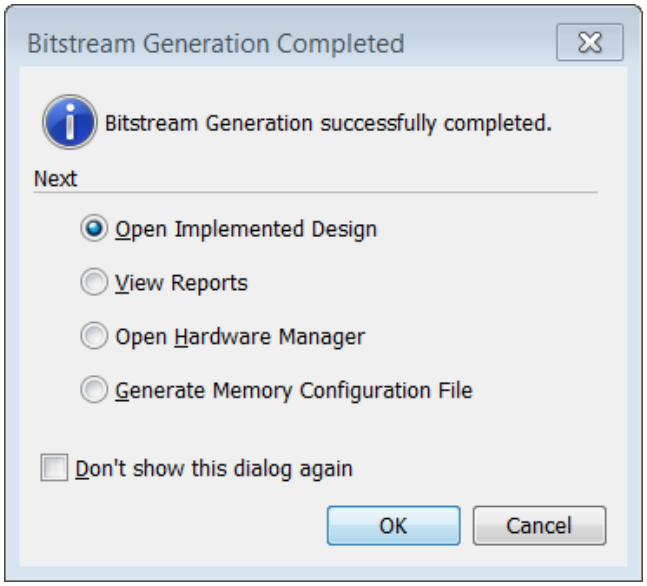


Figure 11-11: **Open Implemented Design**

2. In the Bitstream Generation Completed dialog box, click on **Open Implemented Design**.

   Verify timing by looking at the Timing Summary report, and ensure that BRAM INIT strings are populated with the ELF data.

3. From the main menu, select **Edit > Find**.

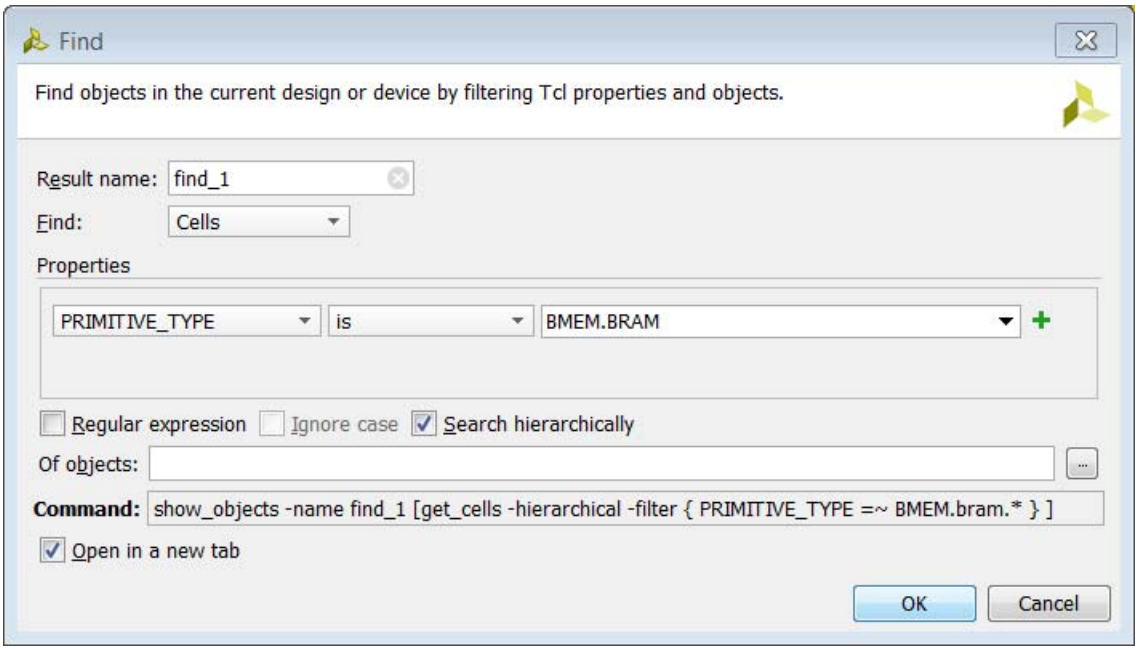


Figure 11-12: **Find BRAM**

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 **BRAM** and verify that the **INIT** properties have been populated in the Cell Properties window.

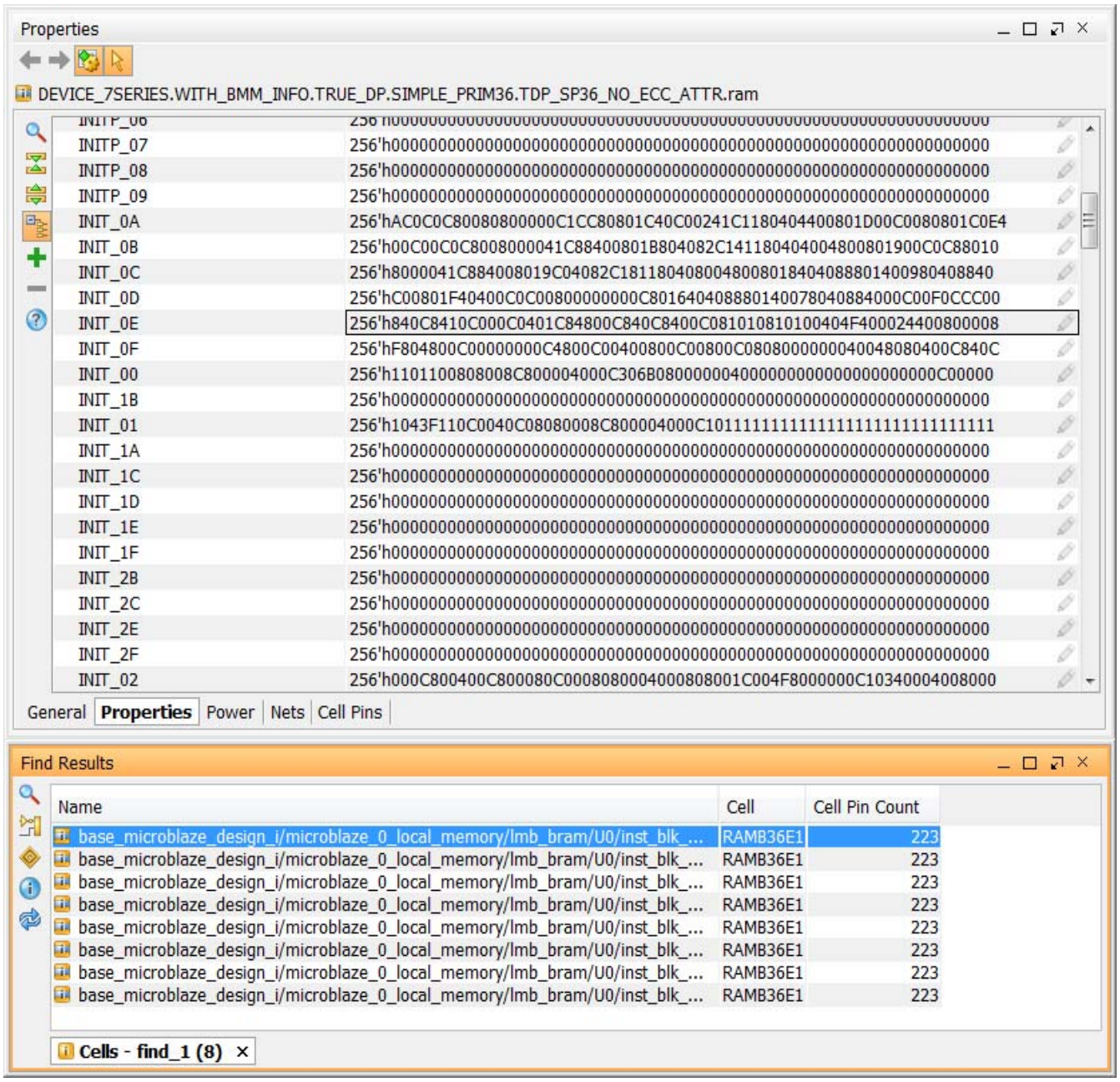


Figure 11-13: **Verify BRAM INIT Properties**

Referencing RTL Modules

Overview

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 the 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

In order 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) [Ref 3] 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 feature can access any of the modules defined within an added source file.

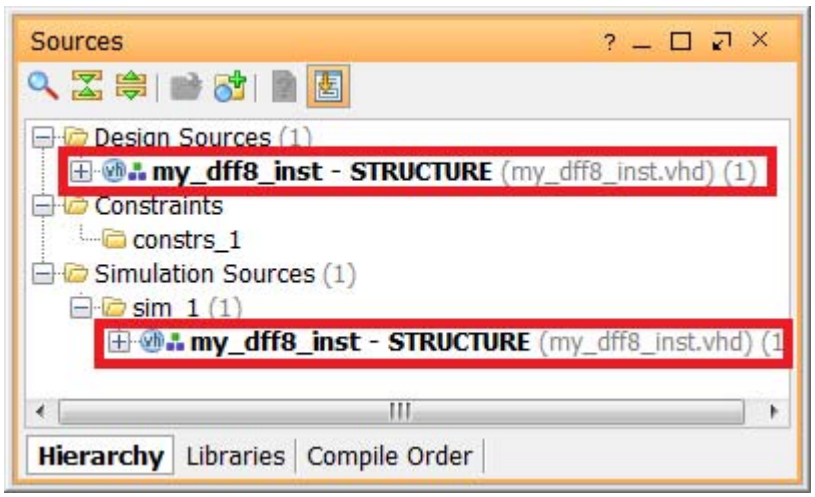


Figure 12-1: **RTL Sources in the Sources window**

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.

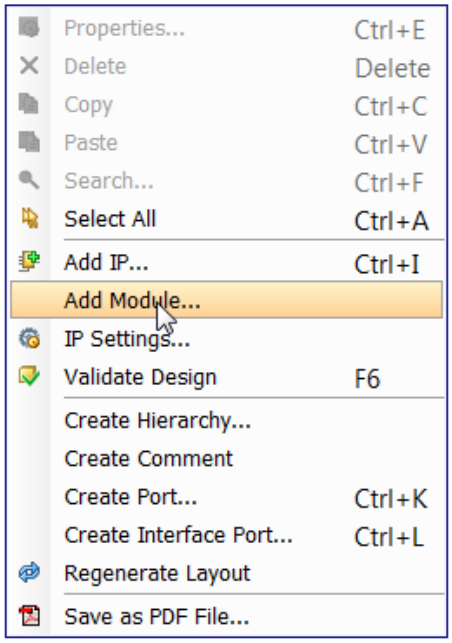


Figure 12-2: **Add Module command**

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.

TIP: *You can only select one module from the list.*

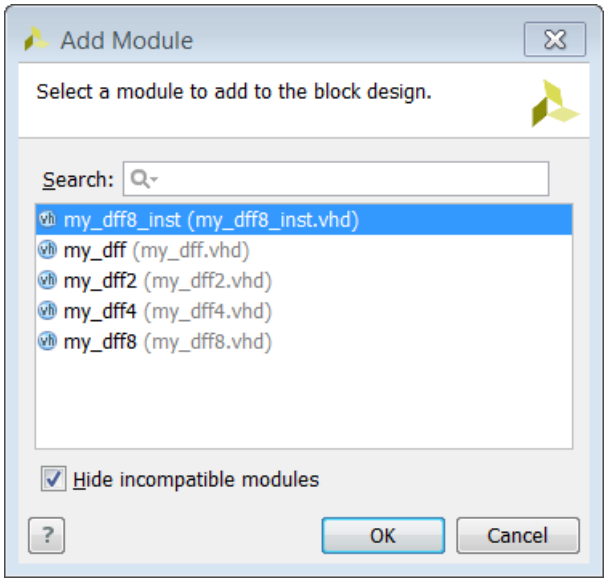


Figure 12-3: **The Add Module dialog box**

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 so cannot be added to the block design.

You can unselect this checkbox 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 may see when unselecting this option includes 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 an IP definition (XCI).

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 Figure 12-3, `my_dff8_inst` is the top-level entity as shown in the following code sample.

```
--------------------------------
library IEEE;
use IEEE.STD_LOGIC_1164.ALL;
library UNISIM;
use UNISIM.VCOMPONENTS.ALL;
entity my_dff8_inst is

  port (
    clk_in : in STD_LOGIC;
    d_in1 : in STD_LOGIC;
```

Figure 12-4: **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.

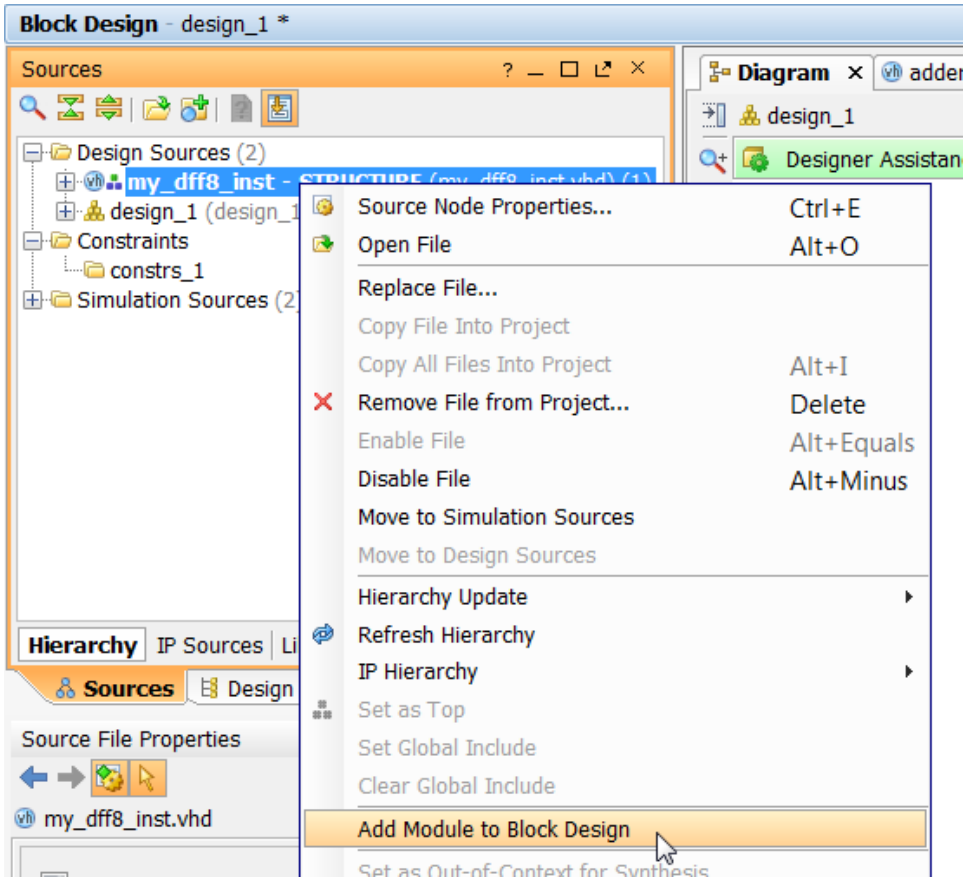


Figure 12-5: **Alternate way of adding a module from the Sources window**

The selected module is added to the block design and you can make connections to it just as you would with any other IP in the design. The IP is displayed in the block design with special markings that identify it as an RTL referenced module, as shown in Figure 12-6 on the next page.

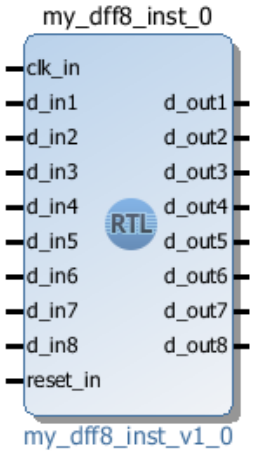


Figure 12-6: **Modules Referenced from an RTL Source File**

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 on the block design in the Sources window and use **Create HDL Wrapper** from the context menu. Refer to Integrating the Block Design into a Top-Level Design, page 75 for more information.

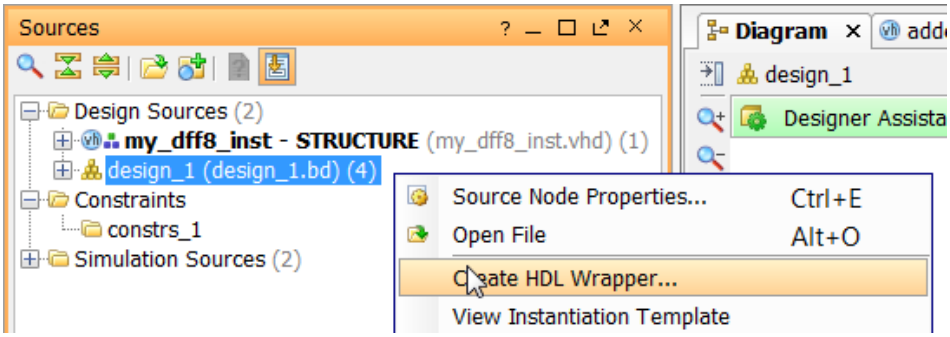


Figure 12-7: **Generate HDL Wrapper**

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 Figure 12-8, page 166.

If you delete a referenced module from the block design, then the module is moved outside the block design hierarchy in the Sources window.

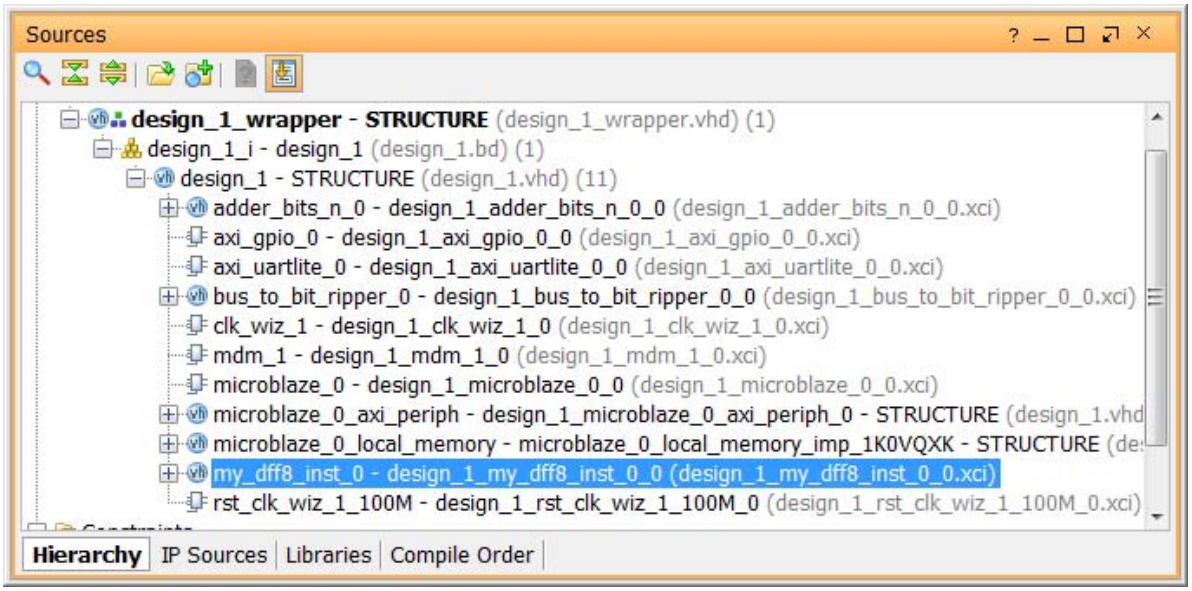


Figure 12-8: **Referenced RTL Module under the Block Design tree**

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 below.

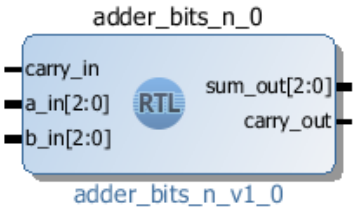


Figure 12-9: **RTL Logo on a the RTL Module symbol**

You can also see some differences between packaged IP and referenced modules when viewing the source files in the Sources window. As an example when you reset the output products of a block design, all the source file, constraint files and other meta data associated with IP blocks are deleted. However, a module reference block just contains the source HDL, so there is nothing to delete. Also, note that a module reference block shows up as "**Module Reference Wrapper**" and not as an XCI file.

After resetting the output products of a block design, the Sources window will appear as shown in the following figure:

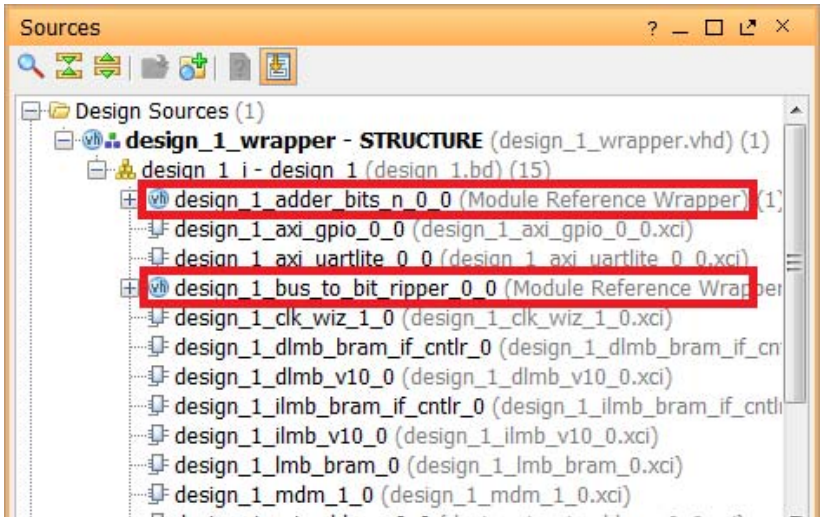


Figure 12-10: **IP and Module Ref Both Out-of-date**

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 Figure 12-25, page 177. IP can be upgraded by clicking on the **Upgrade Selected** button in the IP Status window.

Out-of-date reference modules are also reported by a link in the design canvas window, as shown in Figure 12-23, page 177. 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 Editing the RTL Module after Instantiation, page 176 for more information.

Since 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 IP* (UG1118) [Ref 11].

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, page 170 in order 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 will be 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 *n* is the generic that controls the width of the adder.

```
entity adder_bits_n is
    Generic (n : integer := 2);
    Port ( carry_in : in STD_LOGIC;
           a_in : in STD_LOGIC_VECTOR(n - 1 downto 0);
           b_in : in STD_LOGIC_VECTOR(n -1 downto 0);
           sum_out : out STD_LOGIC_VECTOR(n -1 downto 0);
           carry_out : out STD_LOGIC);
end adder_bits_n;
```

Figure 12-11: **Code snippet for an n-bit full adder**

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 *n*. In this case the port width would be 2-bits.

You can double click on the module to open the Re-customize Module Reference dialog box. You can also right-click on 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.

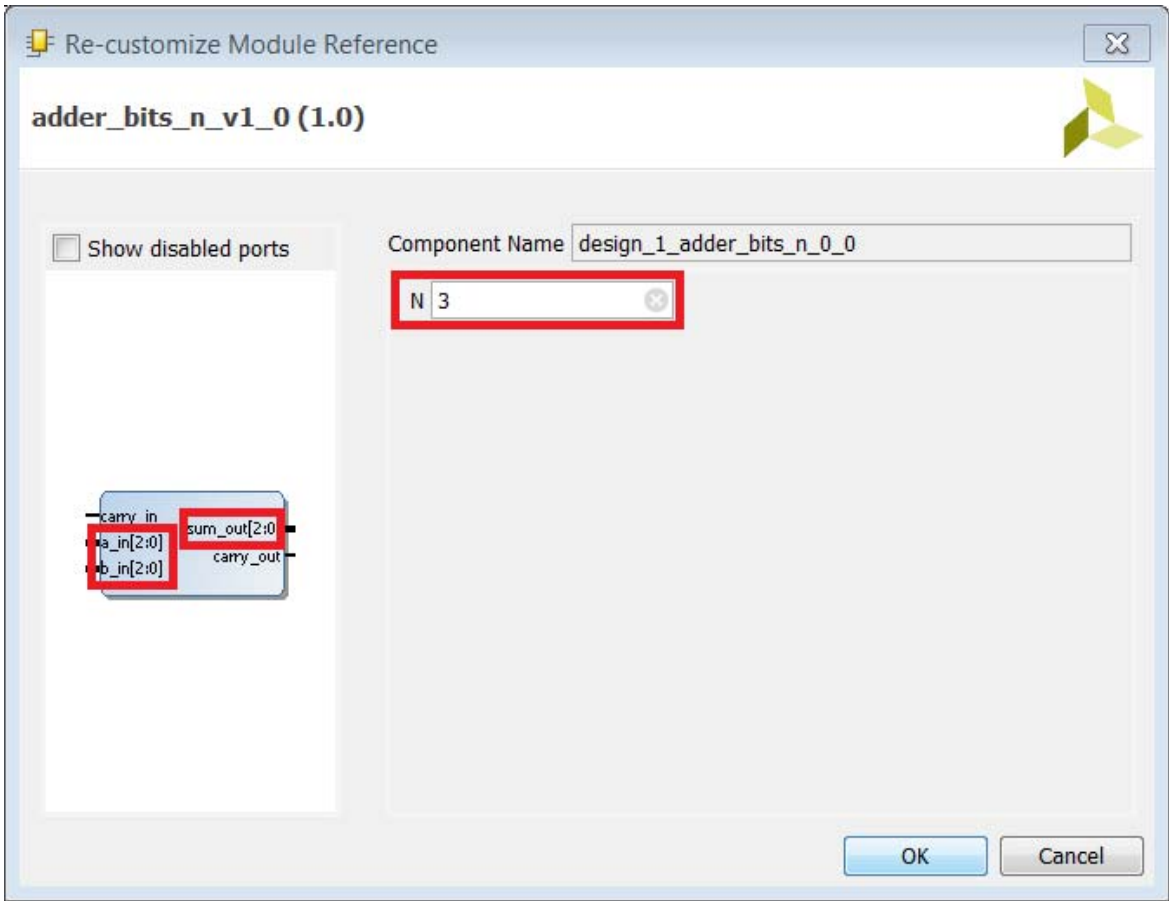


Figure 12-12: **Re-customize Module Reference dialog box**

The symbol in the block design is changed accordingly.

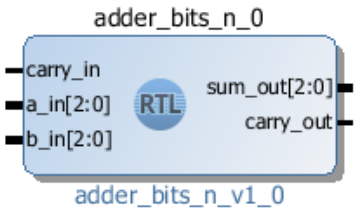


Figure 12-13: **RTL Module post customization**

Inferring Control Signals in a RTL Module

You must also 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. You can access these templates by clicking on **Language Templates** under Project Manager in the Flow Navigator.

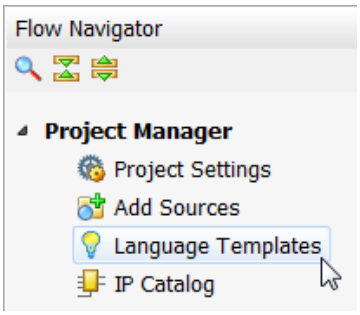


Figure 12-14: **Select Language Templates**

This opens up the Language Templates dialog box as shown below.

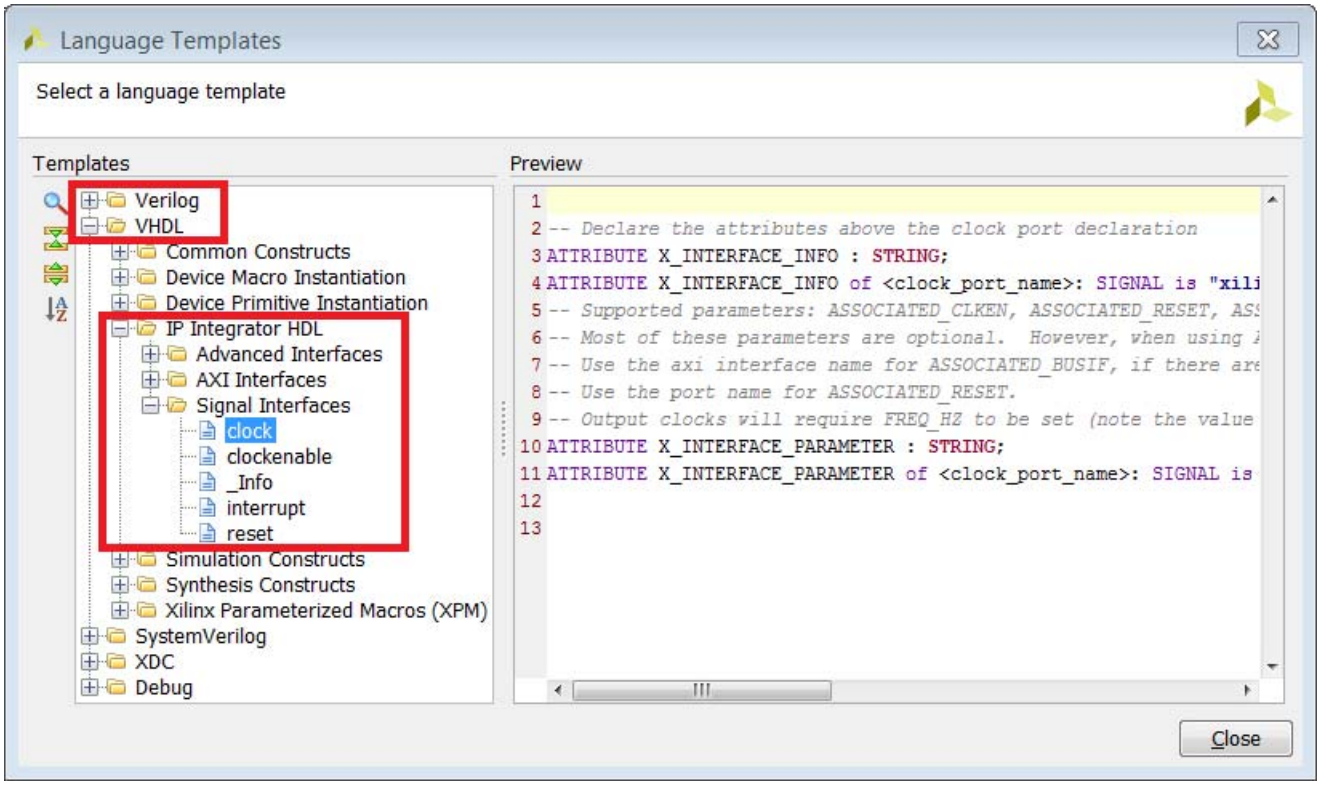


Figure 12-15: **Language Templates dialog box**

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 Figure 12-16, which shows the declaration of the attributes and the definition of attribute values for both the clock and reset signals.

Figure 12-16: **Using Language Template**

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 properties associated with it.

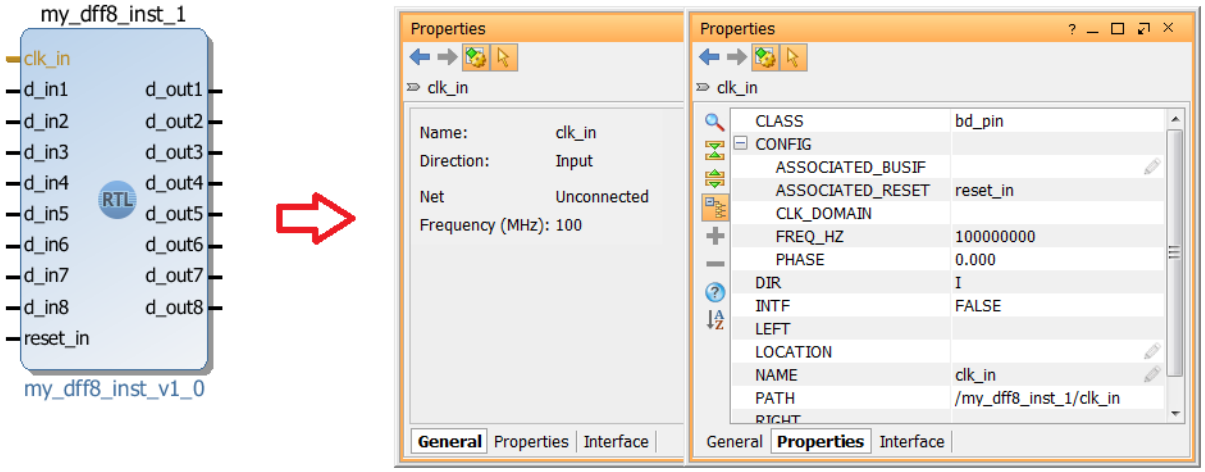


Figure 12-17: **Inspect inferred properties of a clock pin**

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. Refer to Chapter 5, 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 on an instance, and selecting Refresh Module from the context menu, or the following 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 19-2228] Inferred bus interface 'clk_in' of definition
'xilinx.com:signal:clock:1.0'.
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 19-2228] Inferred bus interface 'reset_in' of definition
'xilinx.com:signal:reset:1.0'.
INFO: [IP_Flow 19-4728] Bus Interface 'reset_in': Added interface parameter
'POLARITY' with value 'ACTIVE_HIGH'.
```

This command can also be used to 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 will see what is being inferred on the Tcl console.

Sometimes you may 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";
```

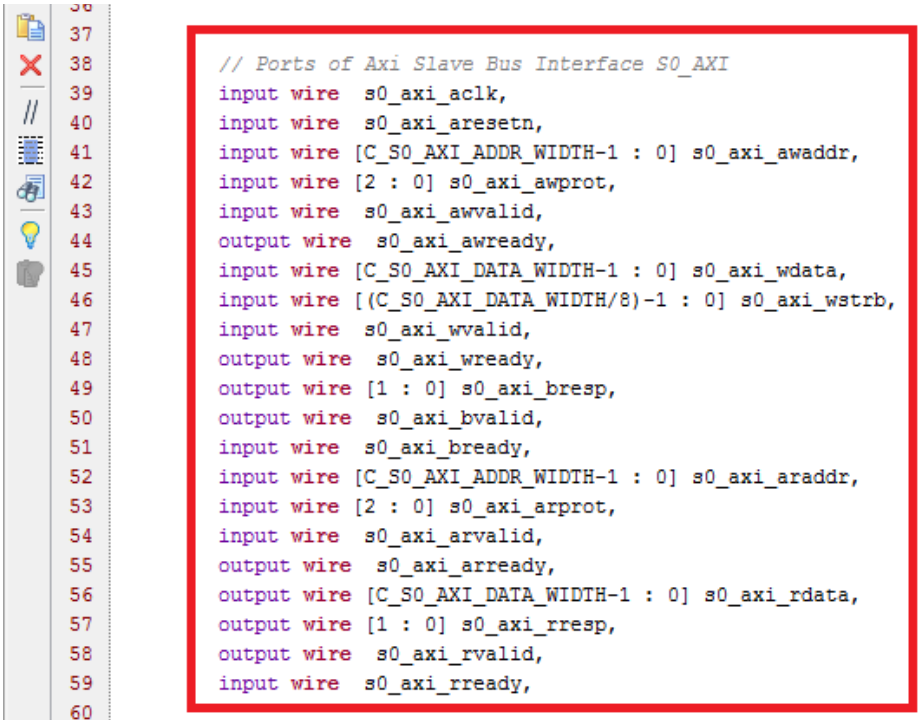
The syntax for Verilog is as follows:

```
(* X_INTERFACE_IGNORE = "true" *)

input <port_name>,
```

Inferring AXI Interface

If the standard naming convention for an AXI interface is used, the interface is automatically inferred by the Vivado IP Integrator. As an example the following code sample shows standard AXI names being used:



```
// Ports of Axi Slave Bus Interface S0_AXI
input wire  s0_axi_aclk,
input wire  s0_axi_aresetn,
input wire [C_S0_AXI_ADDR_WIDTH-1 : 0] s0_axi_awaddr,
input wire [2 : 0] s0_axi_awprot,
input wire  s0_axi_awvalid,
output wire  s0_axi_awready,
input wire [C_S0_AXI_DATA_WIDTH-1 : 0] s0_axi_wdata,
input wire [(C_S0_AXI_DATA_WIDTH/8)-1 : 0] s0_axi_wstrb,
input wire  s0_axi_wvalid,
output wire  s0_axi_wready,
output wire [1 : 0] s0_axi_bresp,
output wire  s0_axi_bvalid,
input wire  s0_axi_bready,
input wire [C_S0_AXI_ADDR_WIDTH-1 : 0] s0_axi_araddr,
input wire [2 : 0] s0_axi_arprot,
input wire  s0_axi_arvalid,
output wire  s0_axi_arready,
output wire [C_S0_AXI_DATA_WIDTH-1 : 0] s0_axi_rdata,
output wire [1 : 0] s0_axi_rresp,
output wire  s0_axi_rvalid,
input wire  s0_axi_rready,
```

Figure 12-18: **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.

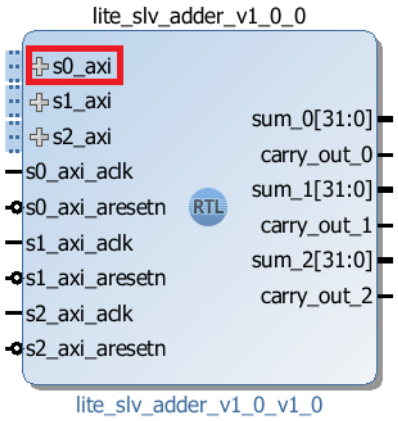


Figure 12-19: **AXI Interface on a Module Reference**

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 AXI Memory Mapped interface listing the attributes that need to be inserted into the module definition.

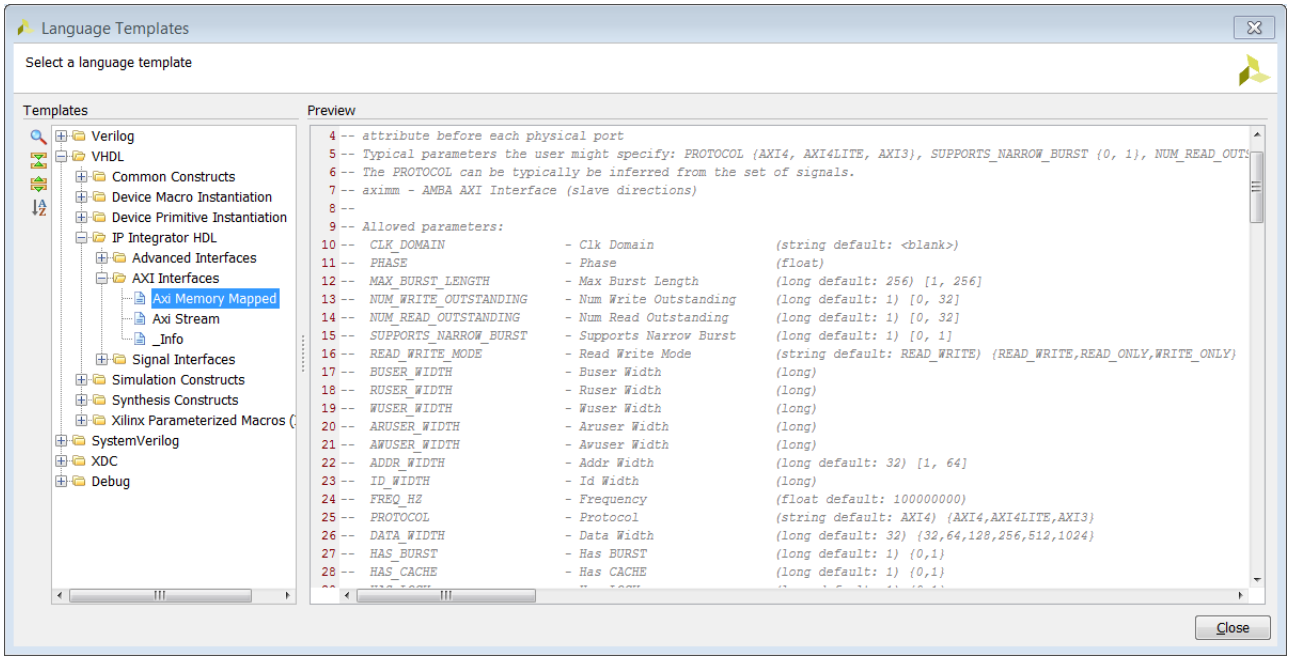


Figure 12-20: **Language Templates for Non-Standard AXI Names**

Editing the RTL Module after Instantiation

You can edit the source code of a module by right-clicking on it and selecting Go To Source from the context menu.

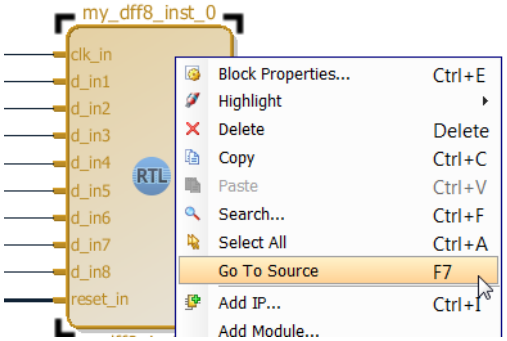


Figure 12-21: **Go to Source of an RTL module**

This opens the module source file for editing.

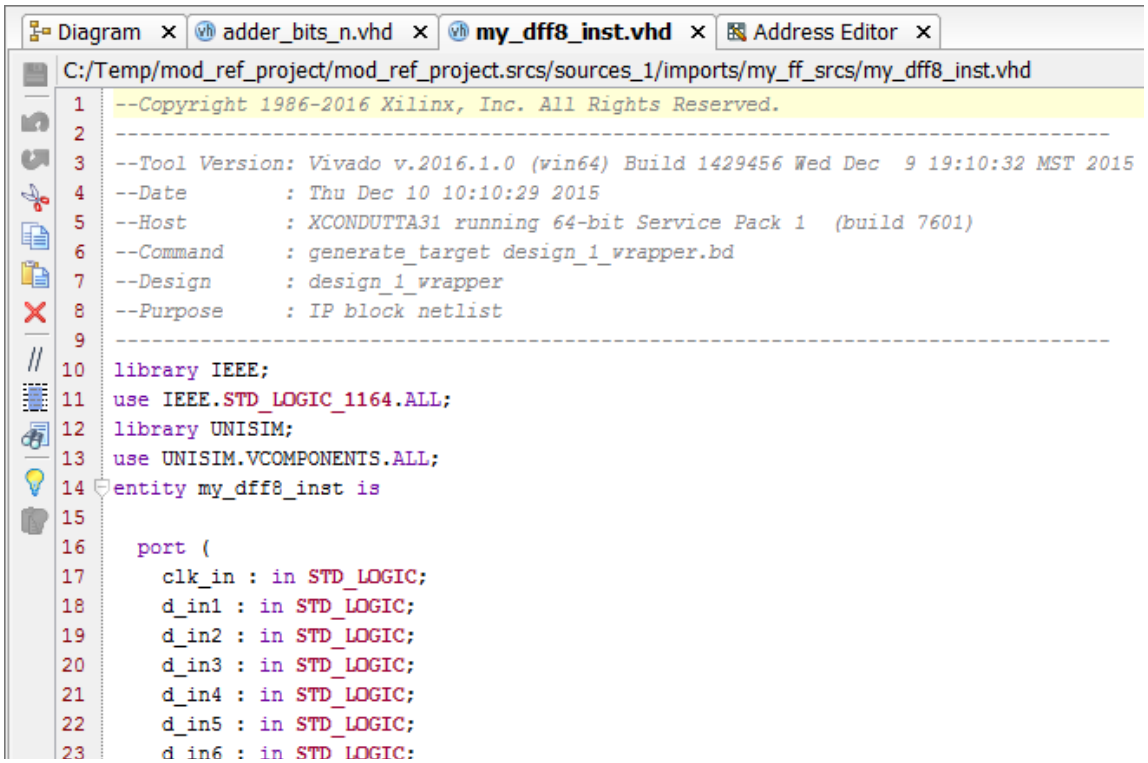


Figure 12-22: **Editing Top Level Source File in the editor**

If you modify the source and save it, you will notice that the **Refresh Changed Modules** link becomes active in the banner of the block design canvas.

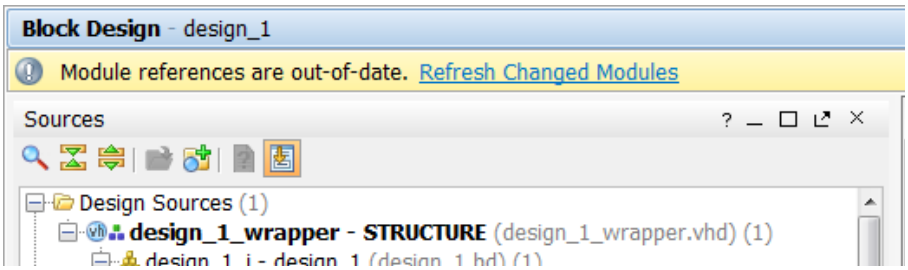


Figure 12-23: **Updating a RTL module**

Click on the **Refresh Changed Modules** link 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 may see a message as follows.

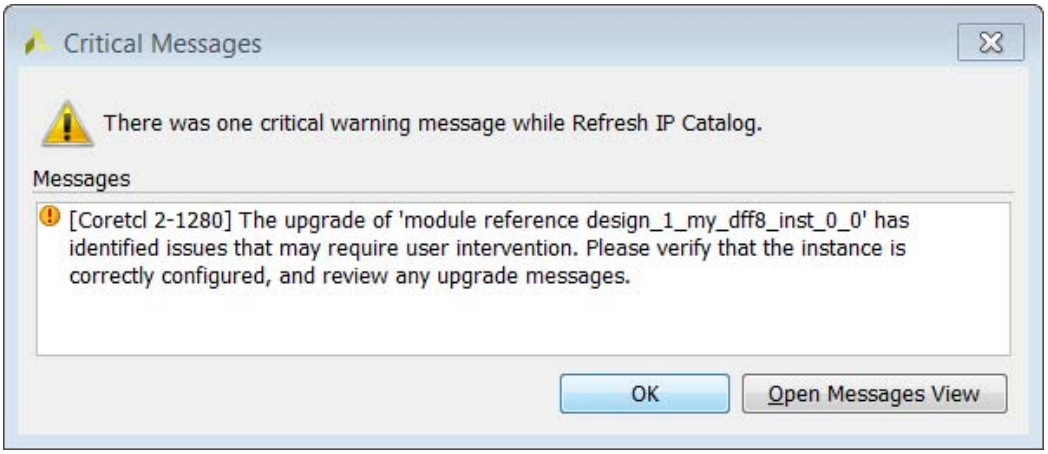


Figure 12-24: **Critical Warning dialog box after updating a RTL module**

On the Tcl console you will see the changes that were made to the module.

```
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.
```

When some of the IP in the block design and the referenced modules are out-of-date, you will see the following in the banner of the block design canvas:

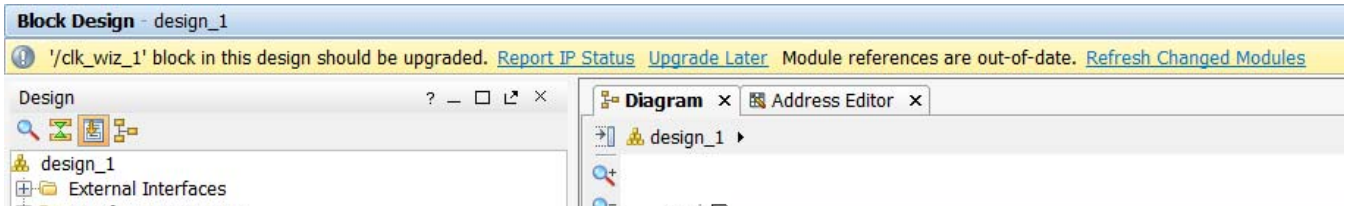


Figure 12-25: **Both IP and Module References Out-of-date**

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 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.

Limitations of the Module Reference Feature

- Since a module reference is not an IP, you cannot specify the Vendor, Library, Name and Version.
- The RTL module definition cannot include other IP definitions (XCI), 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.

TIP: *SystemVerilog and VHDL 2008 are not supported for the module or entity definition at the top-level of the RTL module.*

- You cannot associate an XDC file with a module reference as you can with a packaged IP. You can synthesize the module as an OOC module, however, you cannot associate XDC files with the module.
- A block design containing a module reference cannot be packaged as an IP. Instead, you could separately package the module as an IP, and then package the BD including that IP.

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.

References

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 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 All Programmable SoC and *7 Series Devices Memory Interface Solutions User Guide* (UG586)
14. *AXI Interrupt Controller LogiCORE IP Product Guide* (PG099)
15. *UltraScale Architecture-Based FPGAs Memory IP Product Guide* (PG150)
16. *LogiCORE IP Integrated Logic Analyzer Product Guide* (PG172)
17. *LogiCORE IP System Integrated Logic Analyzer Product Guide* (PG261)
18. *LogiCORE IP Utility Vector Logic* (PB046)
19. *LogiCORE IP Utility Reduced Logic* (PB045)
20. *LogiCORE IP Constant* (PB040)
21. *LogiCORE IP Concat* (PB041)
22. *LogiCORE IP Slice* (PB042)
23. *LogiCORE IP Utility Buffer* (PB043)
24. 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. Essentials of FPGA Design Training Course
4. Vivado Design Suite Embedded Systems Design Training Course
5. Vivado Design Suite Advanced Embedded Systems Design Training Course

Please Read: Important Legal Notices

The information disclosed to you hereunder (the "Materials") is provided solely for the selection and use of Xilinx products. To the maximum extent permitted by applicable law: (1) Materials are made available "AS IS" and with all faults, Xilinx hereby DISCLAIMS ALL WARRANTIES AND CONDITIONS, EXPRESS, IMPLIED, OR STATUTORY, INCLUDING BUT NOT LIMITED TO WARRANTIES OF MERCHANTABILITY, NON-INFRINGEMENT, OR FITNESS FOR ANY PARTICULAR PURPOSE; and (2) Xilinx shall not be liable (whether in contract or tort, including negligence, or under any other theory of liability) for any loss or damage of any kind or nature related to, arising under, or in connection with, the Materials (including your use of the Materials), including for any direct, indirect, special, incidental, or consequential loss or damage (including loss of data, profits, goodwill, or any type of loss or damage suffered as a result of any action brought by a third party) even if such damage or loss was reasonably foreseeable or Xilinx had been advised of the possibility of the same. Xilinx assumes no obligation to correct any errors contained in the Materials or to notify you of updates to the Materials or to product specifications. You may not reproduce, modify, distribute, or publicly display the Materials without prior written consent. Certain products are subject to the terms and conditions of Xilinx's limited warranty, please refer to Xilinx's Terms of Sale which can be viewed at http://www.xilinx.com/legal.htm#tos; IP cores may be subject to warranty and support terms contained in a license issued to you by Xilinx. Xilinx products are not designed or intended to be fail-safe or for use in any application requiring fail-safe performance; you assume sole risk and liability for use of Xilinx products in such critical applications, please refer to Xilinx's Terms of Sale which can be viewed at http://www.xilinx.com/legal.htm#tos.

Automotive Applications Disclaimer

XILINX PRODUCTS ARE NOT DESIGNED OR INTENDED TO BE FAIL-SAFE, OR FOR USE IN ANY APPLICATION REQUIRING FAIL-SAFE PERFORMANCE, SUCH AS APPLICATIONS RELATED TO: (I) THE DEPLOYMENT OF AIRBAGS, (II) CONTROL OF A VEHICLE, UNLESS THERE IS A FAIL-SAFE OR REDUNDANCY FEATURE (WHICH DOES NOT INCLUDE USE OF SOFTWARE IN THE XILINX DEVICE TO IMPLEMENT THE REDUNDANCY) AND A WARNING SIGNAL UPON FAILURE TO THE OPERATOR, OR (III) USES THAT COULD LEAD TO DEATH OR PERSONAL INJURY. CUSTOMER ASSUMES THE SOLE RISK AND LIABILITY OF ANY USE OF XILINX PRODUCTS IN SUCH APPLICATIONS.