

# Vivado Design Suite User Guide

## *Designing IP Subsystems Using IP Integrator*

UG994 (v2014.4) November 19, 2014



## Revision History

The following table shows the revision history for this document.

| Date       | Version | Changes   |
|------------|---------|---|
| 11/19/2014 | 2014.4  | Added <a href="#">Interrupt handling in IP Integrator</a> in <a href="#">Chapter 2</a> .<br>Added <a href="#">Connecting I/O Ports to an ILA or VIO Debug Core</a> in <a href="#">Chapter 6</a> .<br>Replace <code>export_hardware</code> command with <code>write_sysdef</code> command in <a href="#">Chapter 8</a> . |
| 10/16/2014 | 2014.3  | Republish to fix missing links.   |
| 10/14/2014 | 2014.3  | Minor editorial update.   |
| 10/01/2014 | 2014.3  | Added <a href="#">Creating a Memory Map</a> , reorganized content, and made minor editorial changes.  |
| 06/04/2014 | 2014.2  | Information updated on the topic <b>Exporting the Hardware Definition to SDK</b> (starting on page 42).   |
| 05/07/2014 | 2014.1  | Added two new chapters, Using Tcl Scripts to Create IP Integrator Designs within Projects and Using the Board Flow in IP Integrator. Moved the Legal Notices to Appendix B.   |

# Table of Contents

|   |    |
|---|----|
| Revision History .....  | 2  |
| Chapter 1 Designing IP Subsystems.....                          | 6  |
| Introduction.....   | 6  |
| Chapter 2 Creating a Block Design.....                          | 7  |
| Overview .....  | 7  |
| Creating a Project .....  | 7  |
| Designing with IP Integrator .....                              | 9  |
| Making Connections .....  | 17 |
| Re-arranging the Design Canvas.....                             | 37 |
| Running Design Rule Checks .....                                | 41 |
| Chapter 3 Creating a Memory Map.....                            | 42 |
| Overview .....  | 42 |
| The Address Editor in IP Integrator .....                       | 43 |
| Chapter 4 Working with Block Designs.....                       | 55 |
| Overview .....  | 55 |
| Generating Output Products.....                                 | 55 |
| Integrating the Block Design into a Top-Level Design.....       | 56 |
| Creating a Block-Design Outside of the Project.....             | 58 |
| Packaging a Block Design.....                                   | 62 |
| Exporting the Hardware Definition to SDK .....                  | 62 |
| Adding and Associating an ELF File to an Embedded Design.....   | 65 |
| Setting the Block Design as an Out-of-Context (OOC) Module..... | 71 |
| Chapter 5 Parameter Propagation in IP Integrator .....          | 75 |
| Overview .....  | 75 |
| Bus Interfaces .....  | 76 |
| How Parameter Propagation Works .....                           | 81 |
| Parameters in the Customization GUI.....                        | 82 |
| Example of a Parameter Mismatch.....                            | 84 |

|  |     |
|--|-----|
| Chapter 6 Using the ILA to Debug IP Integrator Designs.....                  | 86  |
| Overview .....   | 86  |
| Using the HDL Instantiation Flow in IP Integrator.....                       | 86  |
| Using the Netlist Insertion Flow in IP Integrator .....                      | 92  |
| Connecting to the Target Hardware .....                                      | 103 |
| Chapter 7 Using Tcl Scripts to Create Block Designs within Projects .....    | 109 |
| Overview .....   | 109 |
| Create a Design in the Vivado IDE GUI.....                                   | 109 |
| Save the Vivado Project Information in a Tcl File .....                      | 109 |
| Chapter 8 Using Non-Project Mode in IP Integrator .....                      | 112 |
| Overview .....   | 112 |
| Creating a Flow in Non-Project Mode .....                                    | 112 |
| Chapter 9 Updating Designs for a New Release.....                            | 114 |
| Overview .....   | 114 |
| Upgrading a Block Design in Project Mode.....                                | 114 |
| Upgrading a Block Design in Non-Project Mode.....                            | 121 |
| Chapter 10 Revision Control for IP Integrator Designs.....                   | 122 |
| Overview .....   | 122 |
| Design Files Needed to be Checked In for Revision Control.....               | 123 |
| Creating a Block Design for Use in a Different Project.....                  | 124 |
| Importing an Existing Block Design into a Different Vivado IDE Project ..... | 124 |
| Chapter 11 Using Third-Party Synthesis Tools.....                            | 128 |
| Overview .....   | 128 |
| Creating a Design Check Point (DCP) File for a Block Design .....            | 128 |
| Create a Verilog or VHDL Stub File for the Block Design.....                 | 129 |
| Create a HDL or EDIF Netlist in the Synplify Project.....                    | 130 |
| Create a Post-Synthesis Project in Vivado and Implement.....                 | 131 |
| Add Top-Level Constraints .....  | 133 |
| Add ELF File (if present) .....  | 133 |
| Implement the Design.....  | 134 |
| Chapter 12 Using the Board Flow in IP Integrator.....                        | 135 |
| Overview .....   | 135 |

|   |     |
|---|-----|
| Select a Target Board .....                       | 135 |
| Create a Block Design to use the Board Flow ..... | 136 |
| Complete Connections in the Block Design.....     | 141 |
| <br>  |     |
| Appendix A Additional Resources.....              | 143 |
| Xilinx Resources .....                            | 143 |
| Solution Centers.....                             | 143 |
| References .....                                  | 143 |
| <br>  |     |
| Appendix B Legal Notices .....                    | 145 |
| Please Read: Important Legal Notices .....        | 145 |

---

## 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<sup>®</sup> recognizes the challenges designers face, and to aid designers with design and reuse issues, has created a powerful feature within the Vivado<sup>®</sup> 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. First, a single connection in IP integrator (or Tcl command) will make a master to slave connection. Next, the graphical representation of this connection will be simple – a single connection. Finally, 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.

---

### Overview

This chapter describes the basic operations and functionality of 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. You can add VHDL or Verilog design files, any custom IP, and other types of design source files to the project using this wizard.

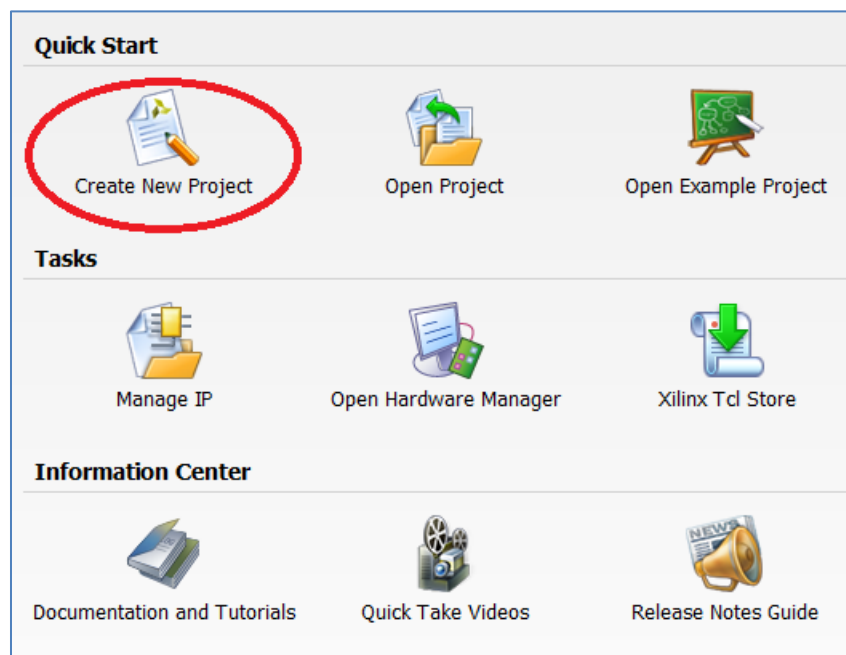


Figure 1: Creating a New Project

As shown in the figure below, you can also select the target device or a Xilinx target board. Vivado tools also support multiple versions of Xilinx target boards, so carefully select your target hardware.

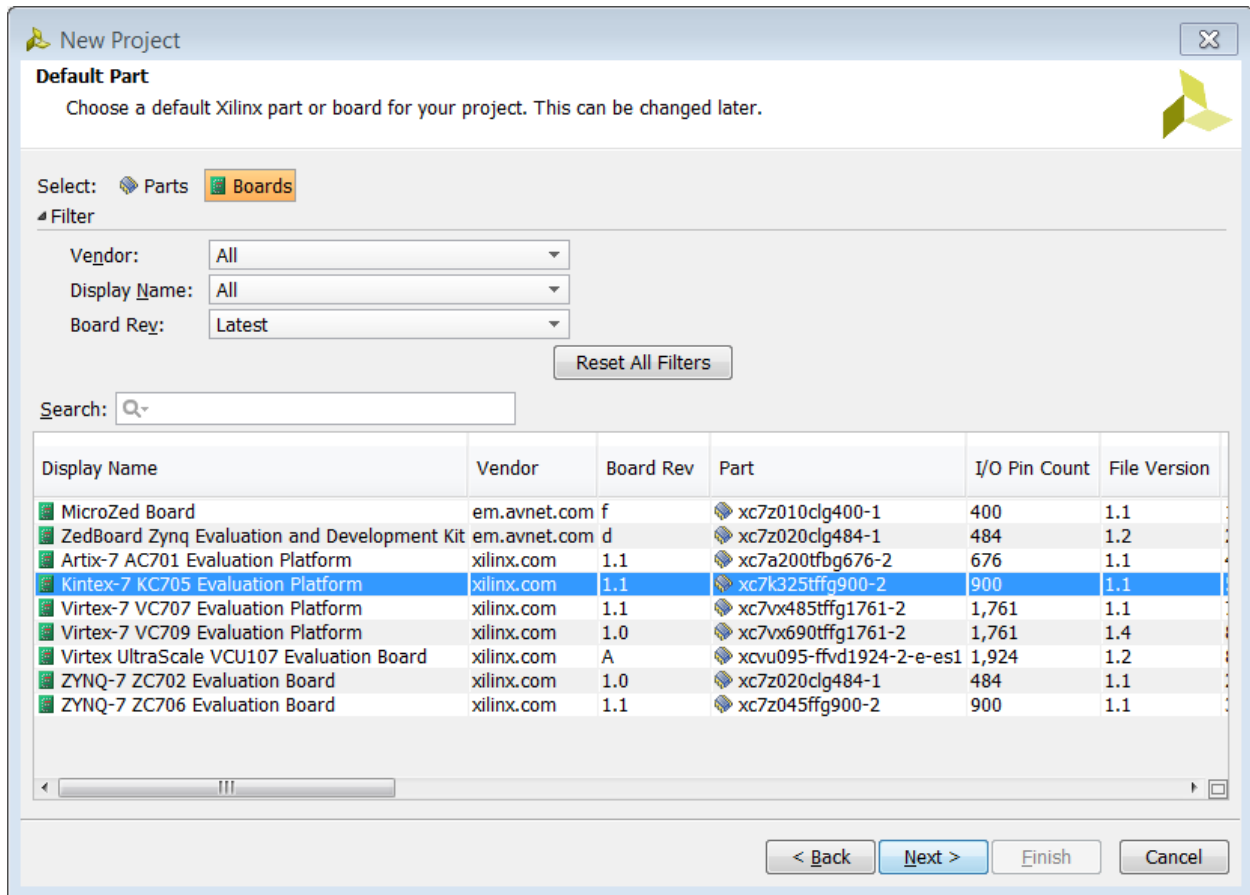


Figure 2: Choosing a New Project Target Device

**Note:** You can perform the same actions by using the following Tcl commands. When displaying the Tcl commands in this document, the symbols < > are used to surround parameters that are specific to your design. The < > symbols should not be included in the command string.

The Tcl equivalent commands are:

```
create_project xx <your_directory>/xx -part xc7k325tffg900-2
set_property board_part Xilinx.com:kc705:part0:1.0 [current_project]
set_property target_language VHDL [current_project]
```



## Designing with IP Integrator

You create a new block design in the Flow Navigator by clicking on **Create Block Design** under the IP Integrator heading.

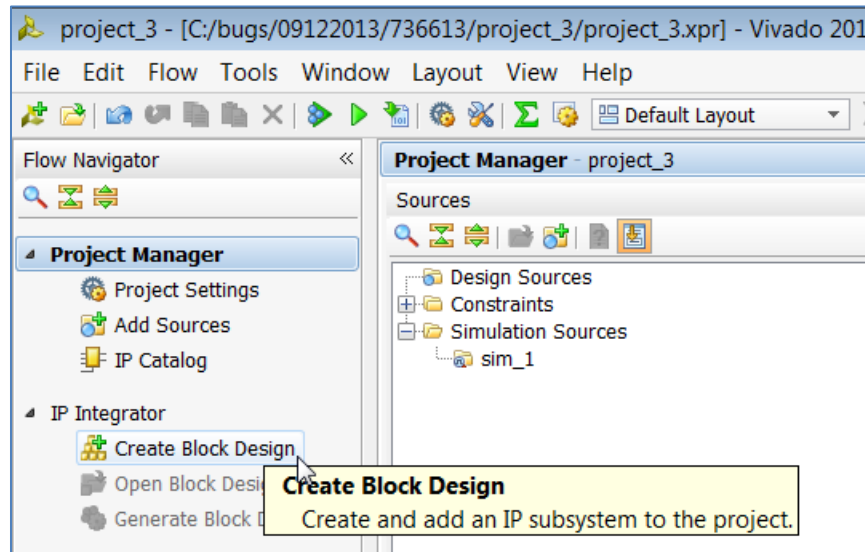


Figure 3: Creating a Block Design

The Tcl equivalent command is:

```
create_bd_design "<your_design_name>"
```

## Re-sizing the IP Integrator Diagram

Once the design is created, a canvas is presented that you can use to construct your design. This canvas can be re-sized as much as possible by re-sizing panes in the Vivado IDE GUI. It is possible to move the diagram to a separate monitor by clicking on the **Float Window** button in the upper-right corner of the diagram. You can also double click on the **Diagram** 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.

## Changing Layers

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.

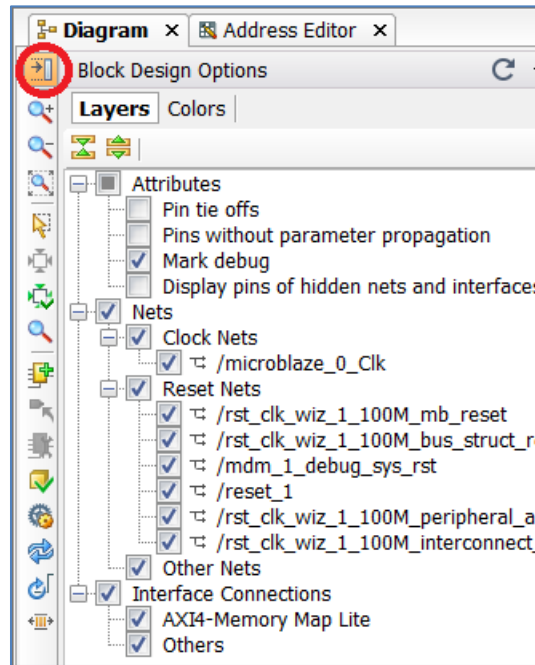


Figure 4: Viewing/Hiding Information on the IP Integrator Canvas

## Changing the Window Background Color

You can change the background color of the diagram canvas from the default white 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.

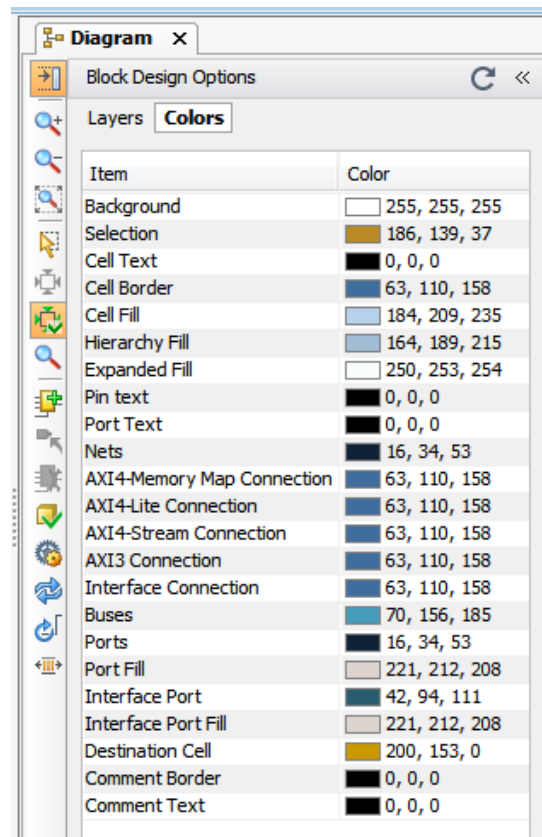


Figure 5: Changing the IP Integrator Background Color

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 Left-Button Panel

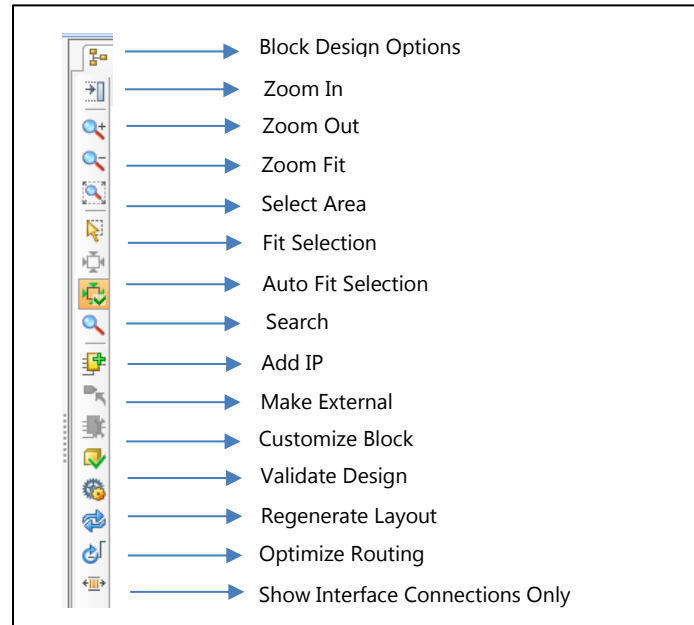
A northwest stroke (lower-right to upper-left) is **Zoom Fit**

A southwest stroke (upper-right to lower-left) is **Zoom In**

A northeast stroke (lower-left to upper-right) is **Zoom Out**

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

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

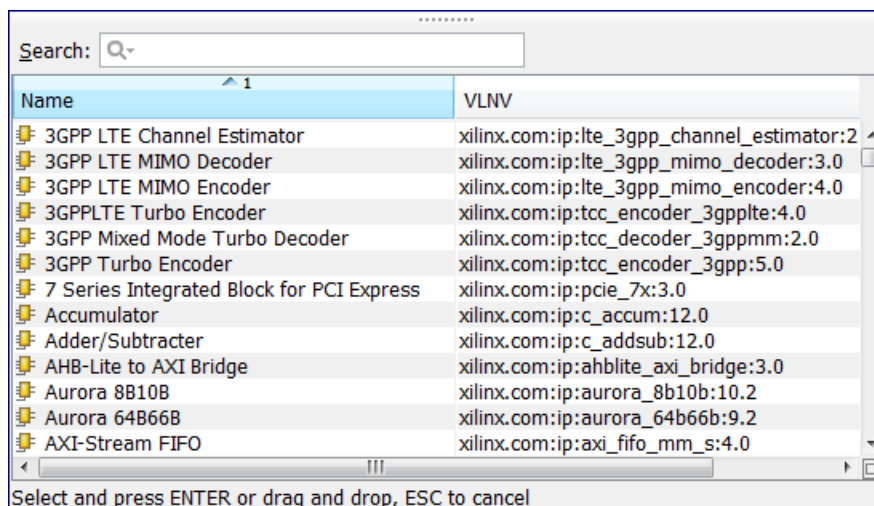


**Figure 6: IP Integrator Action Buttons**

## Adding IP Modules to the Design Canvas

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

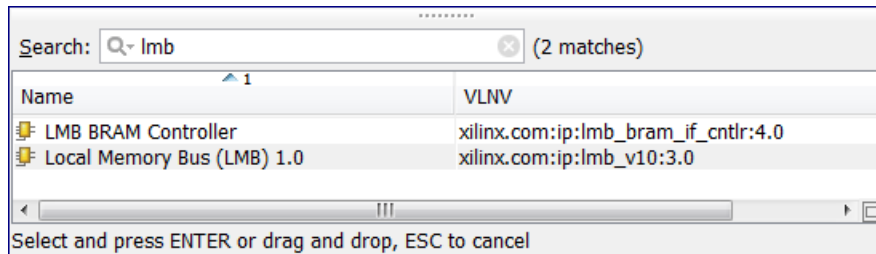
1. Right click in the diagram and select **Add IP**. A searchable IP Catalog opens.



**Figure 7: Opening the Vivado IP Catalog**

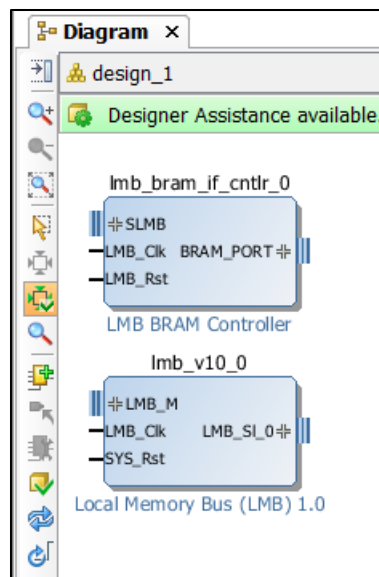
By typing in the first few letters of the IP name in the Search filter at the top of the catalog, only IP modules matching the search string are displayed.

**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.



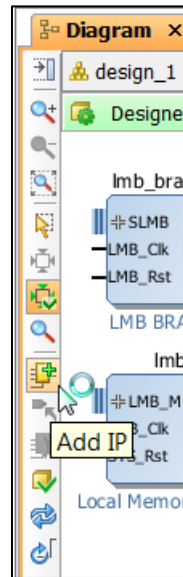
**Figure 8: Using the Search Filter in the IP Catalog**

- 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.
- To add multiple IP to the Block Design, you can highlight the additional desired IP (Ctrl-click) and press the **Enter** key.



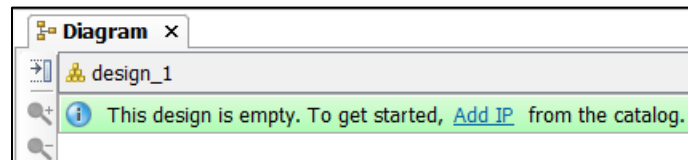
**Figure 9: Adding Multiple IP at the Same Time**

- You can also add IP by clicking on the Add IP button on the left side of the canvas.



**Figure 10: Adding IP by Clicking the Add IP Icon**

Alternatively, IP can also be added by clicking on the link Add IP, in the green banner towards the top of the IP integrator canvas.



**Figure 11: Adding IP by Clicking Add IP Link**

You can also add an IP by selecting the IP in the IP Catalog and dragging and dropping it in the canvas.

For the actions described above, the IP is placed near the cursor location when the **Add IP** command is invoked.

The Vivado IP catalog entry in the Flow Navigator can also be displayed and used. If you are using dual monitors, you can open the IP catalog in its own monitor. If you are using a single monitor, you can float the IP catalog to move it away from the diagram. To add IP from the main IP catalog, you can drag and drop a selected piece of IP from the IP catalog onto the diagram.

---

 **CAUTION!** If you double click on an IP in the IP Catalog, that IP will be added to the Vivado project, but not to the block design.

---

## 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 parent. These hierarchical blocks let you see the contents of the block, but do not let you edit the hierarchy. Changes cannot be made to the child block design. Changes can only be made in the Re-customize IP dialog box of the IP.

As an example, the AXI Ethernet Subsystem is a Hierarchical IP. You would instantiate this IP just as any other IP by searching and selecting the IP from the IP catalog.

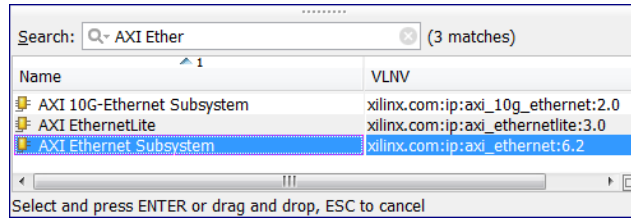


Figure 12: Adding a hierarchical IP to the block design

When the IP has been instantiated in the block design, double click on it to re-customize it. This opens the Re-customize IP dialog box where parameters of the IP can be configured.

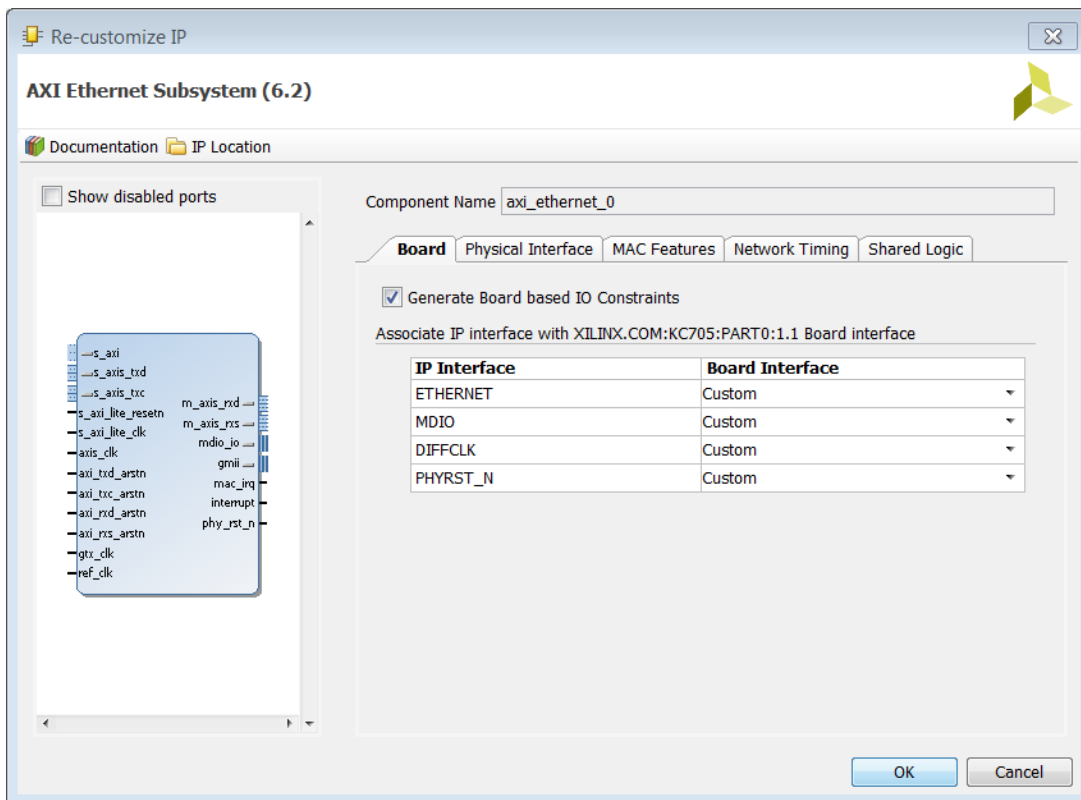
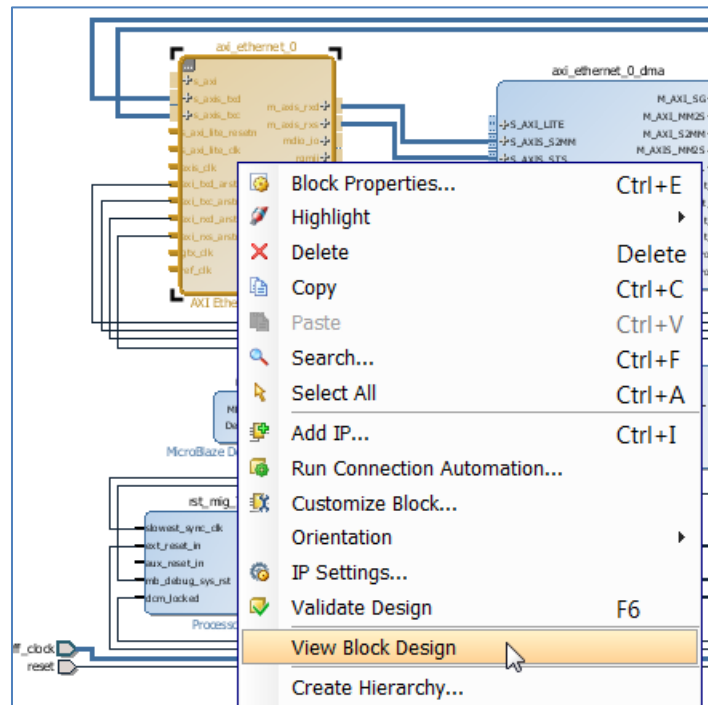


Figure 13: Setting parameters of the hierarchical IP

The child block design inside the AXI Ethernet Subsystem IP can be seen by right-clicking and selecting the **View Block Design** option from the menu.



**Figure 14: Viewing the child block design under the hierarchical IP**

This opens a block design window showing the child level block design. Again, the block design cannot be directly edited.



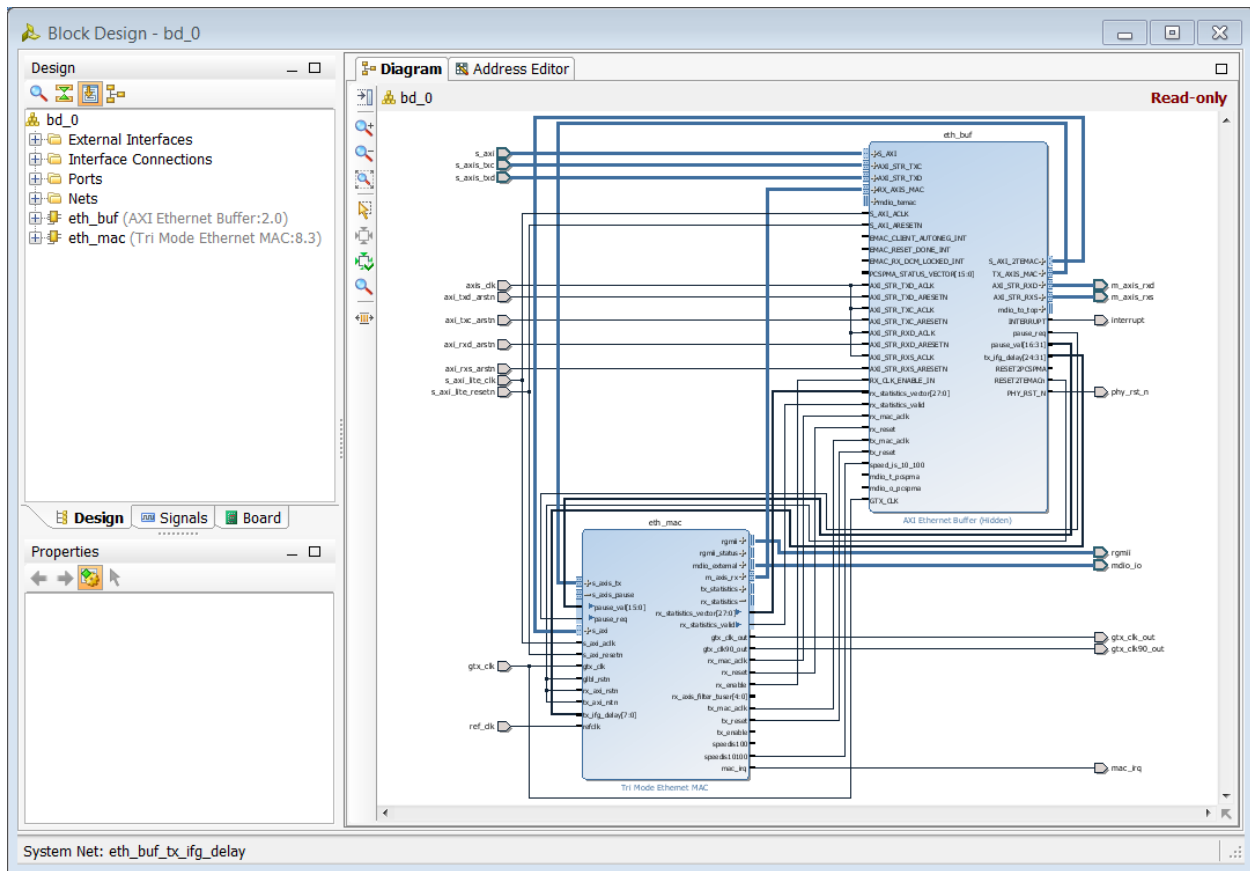


Figure 15: Child block design of the Hierarchical IP



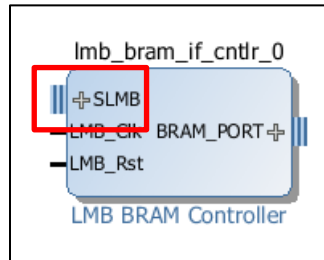
**TIP:** If you re-customize the IP while the child-level block design is open, it will be closed.

## 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.

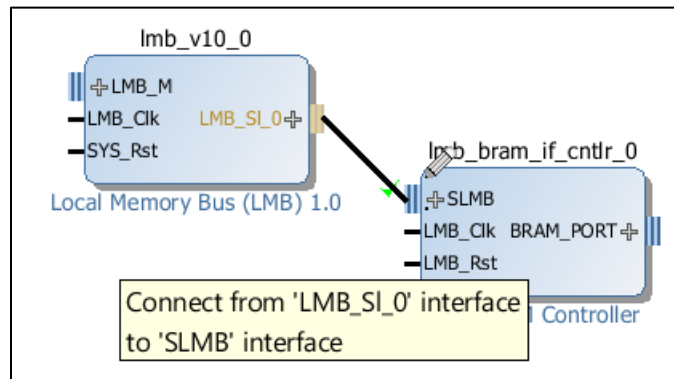
As shown in the figure below, an interface-level connection is indicated by the more prominent connection box on a symbol.

Clicking on the + symbol on a block expands the interface and displays the associated signals and buses.



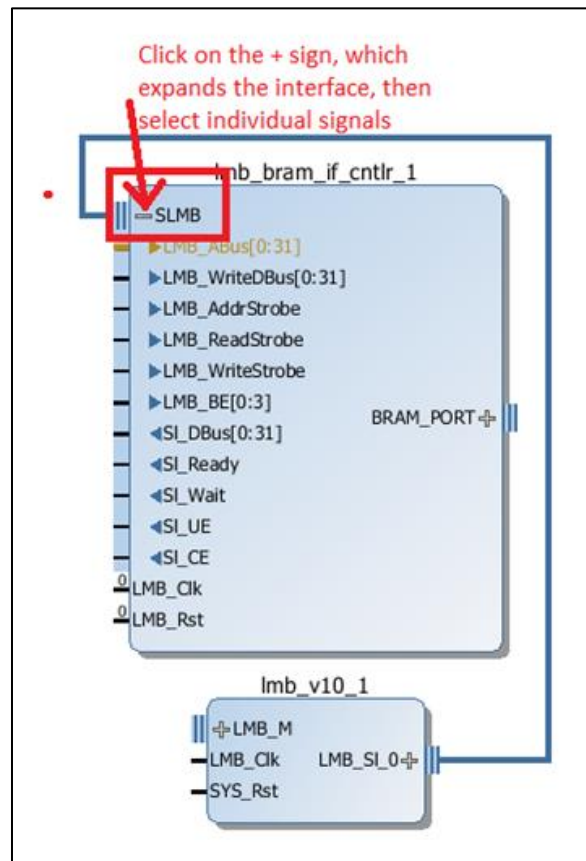
**Figure 16: Connection Box on a Symbol**

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. As shown in the figure below, when you are making a connection, a green check box appears near any possible destination connections, highlighting the potential connections.



**Figure 17: Signal or Bus Connection on a Symbol**

As shown in the following figure, when signals are grouped as an interface, you can expand the interface in order to make connections to individual signal or bus pins.

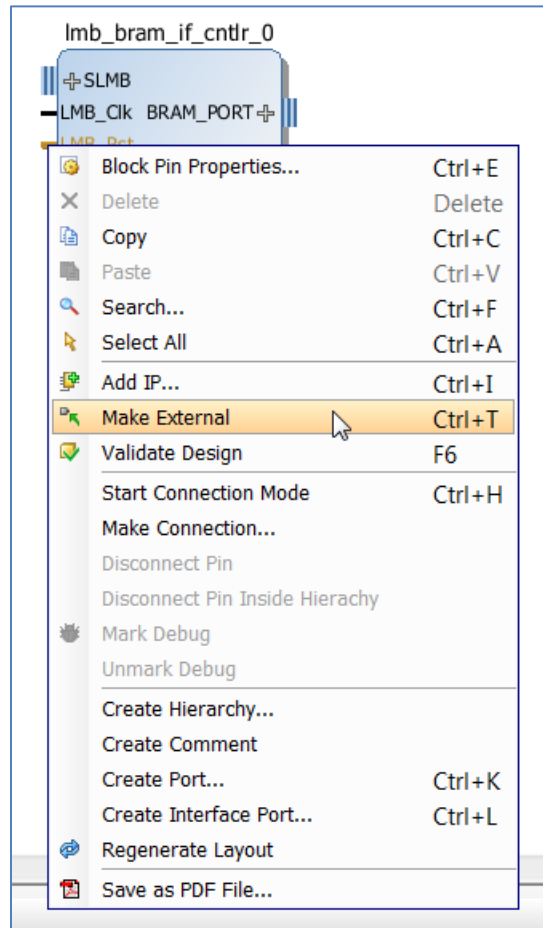


**Figure 18: Expanding the Interface before Making a Connection**

There are three ways to connect signals and interfaces to external I/O ports:

- Make External
- Create Port
- Create Interface Port

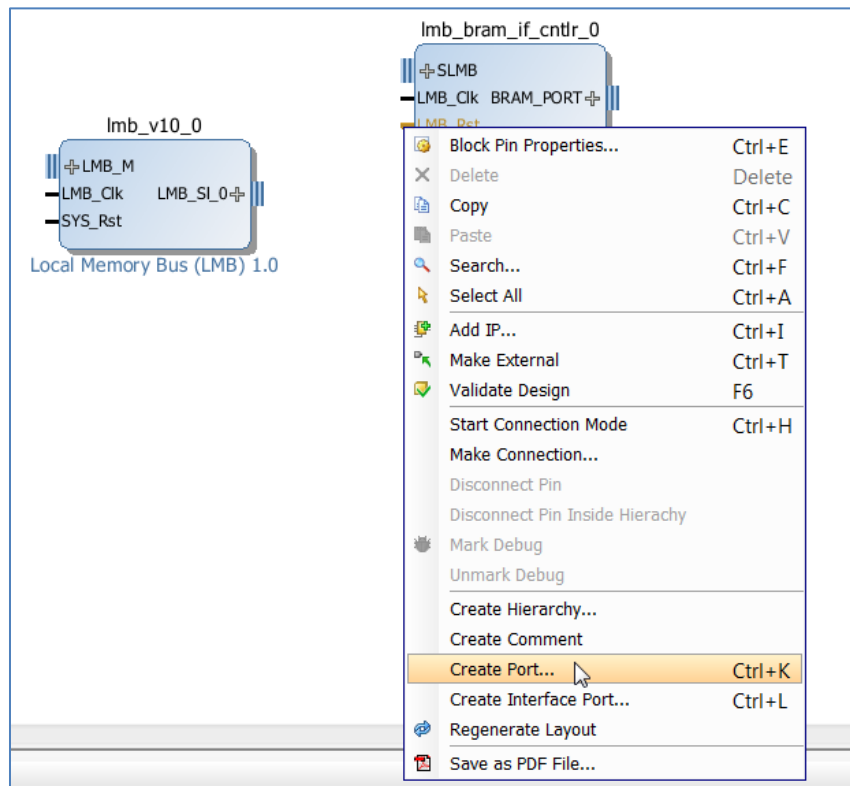
1. As shown in the following figure, to connect signals or interfaces to external ports on a diagram, you first select a pin, bus, or interface connection. You can then right-click and select **Make External**. You can also use Ctrl-click to select multiple pins and invoke the **Make External** command for all pins at one time.



**Figure 19: Making External Connections**

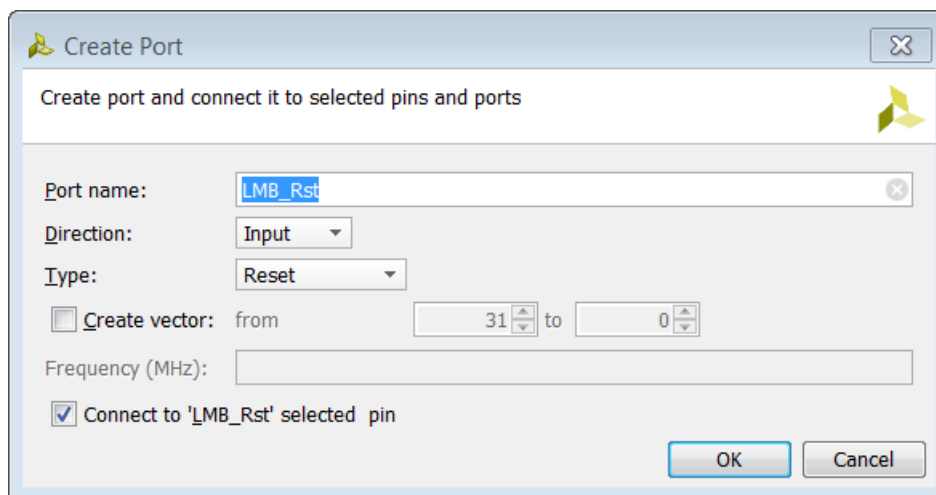
This command is used to tie a pin on an IP to an I/O port on the block design. IP integrator simply connects the port on the IP to an external I/O.

- For the second method of making a connection, you can 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 terms of specifying the input/output, the bit-width and the type (clk, reset, data) etc. In case of a clock, you can also specify the input frequency.



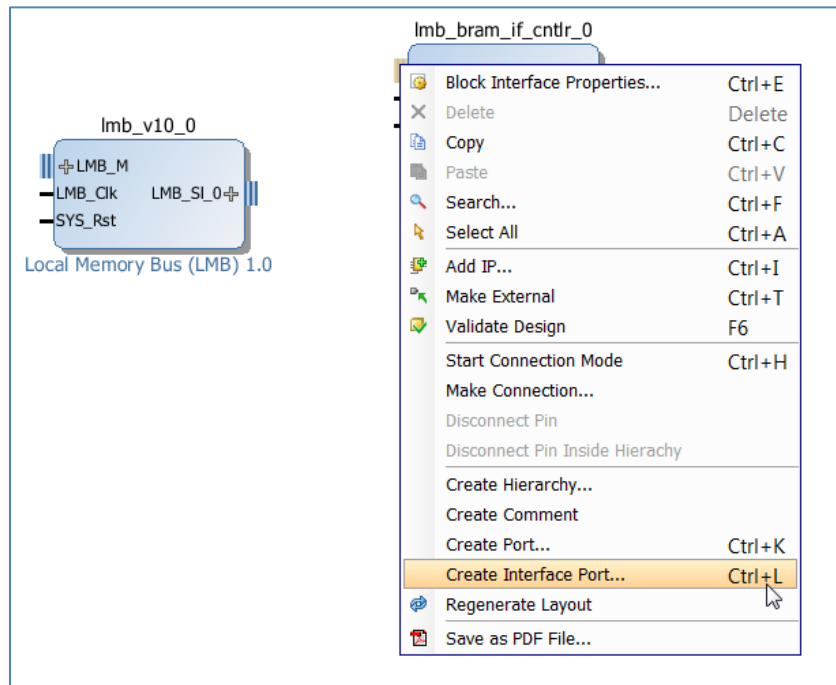
**Figure 20: Creating a Port**

In the Create Port dialog box, you specify the port name, the direction such as input, output or bi-directional, and 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. When the type is selected as clock, you can also specify the frequency of the clock.



**Figure 21: Create Port dialog box**

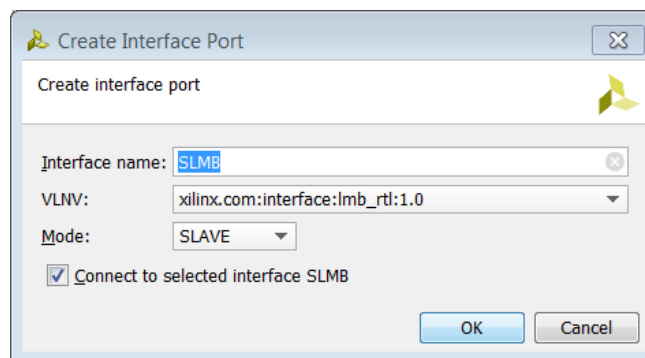
- For the third connection method, you can right-click and select **Create Interface Port**, as shown in the following figure.



**Figure 22: Creating an Interface Port**

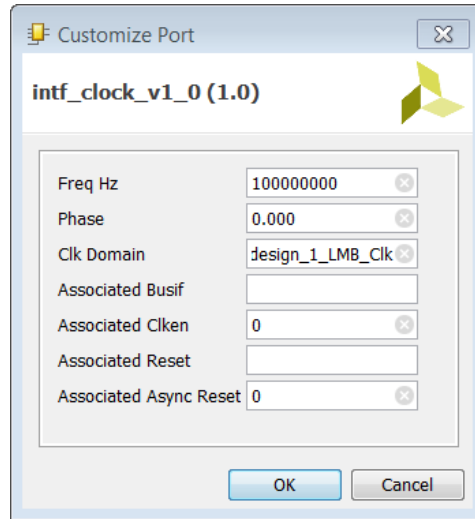
This command is used to create ports on the interface pins which are groupings of signals that share a common function. For example, the LMB\_M and LMB\_SI\_0 are interface pins in the figure above. The Create Interface Port command gives more control in terms of specifying the interface type and the mode (master/slave).

In the Create Interface Port dialog box, you specify the interface name, the vendor, library, name and version (VLNV) field, and the mode field such as MASTER or SLAVE.



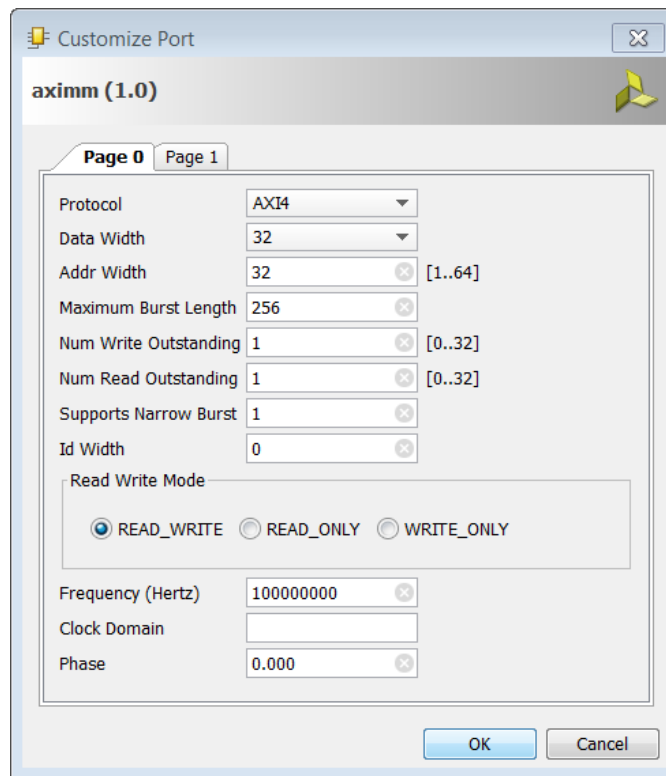
**Figure 23: Create Interface Port dialog box**

You can double-click on external ports to see their properties and modify them. In this case, the port 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) associated with it.



**Figure 24: Customizing clock Port Properties**

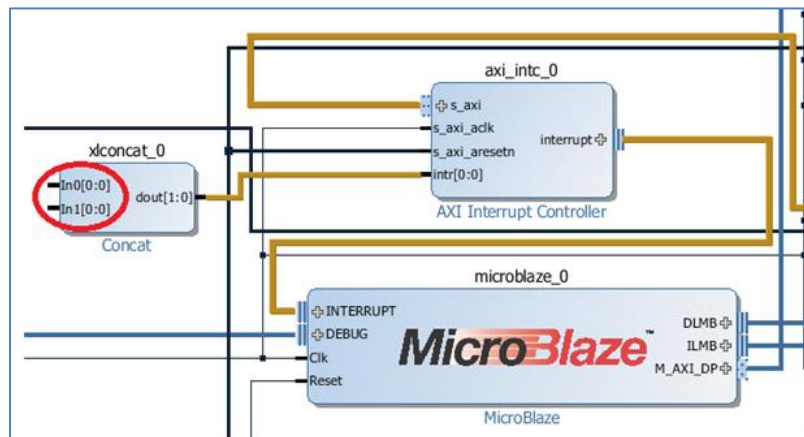
On an AXI interface, double clicking on the port shows the following configuration dialog box.



**Figure 25: Customizing Port Properties of aximm**

## Interrupt handling in IP Integrator

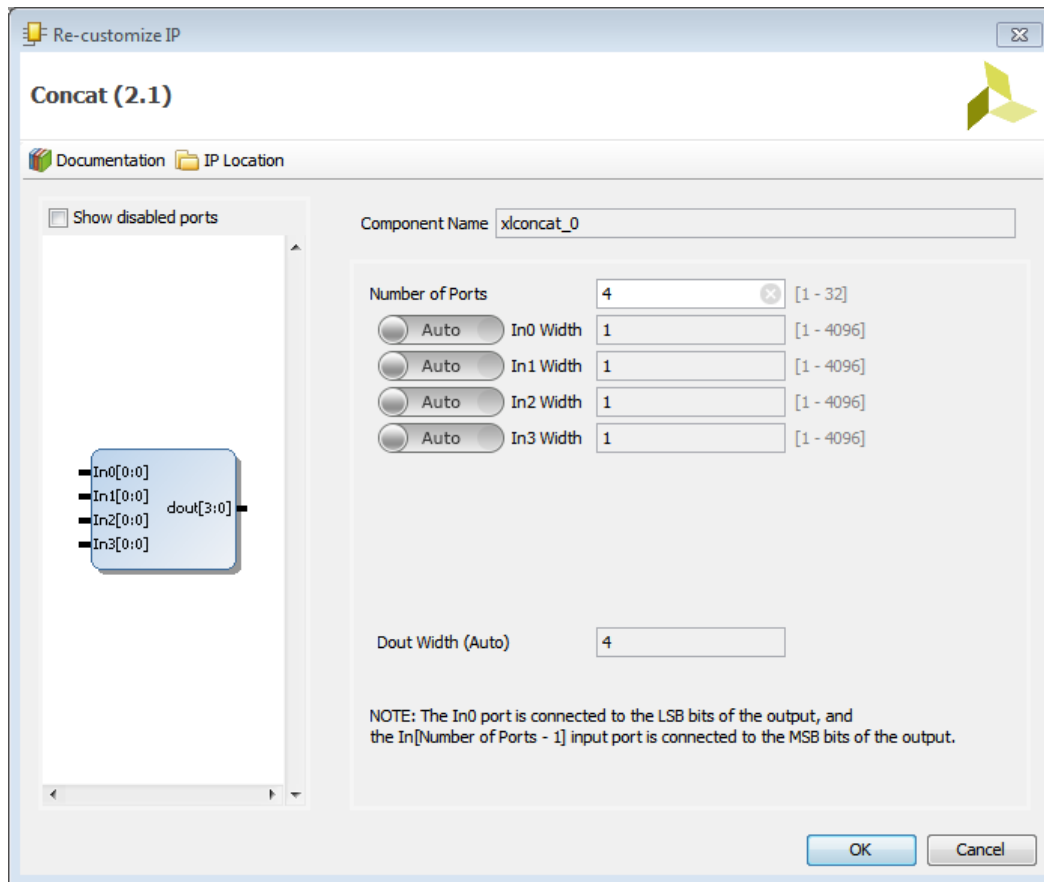
Interrupt handling in the Vivado Design Suite IP Integrator tool depends on the processor being used. For a Zynq® processor the Generic Interrupt Controller block within the Zynq processor handles the interrupt. For a MicroBlaze™ processor the AXI Interrupt Controller IP must be used to manage interrupt. Regardless of the processor used in the design, a Concat IP is used to consolidate and drive the interrupt pins.



**Figure 26: Using a Concat IP to drive interrupt input of the AXI Interrupt Controller**

The inputs of the Concat IP are driven by different interrupt sources. Accordingly, the Concat IP needs to 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 [Figure 27](#). Notice that the width of the output ( $D_{out}$ ) is usually set automatically during parameter propagation.



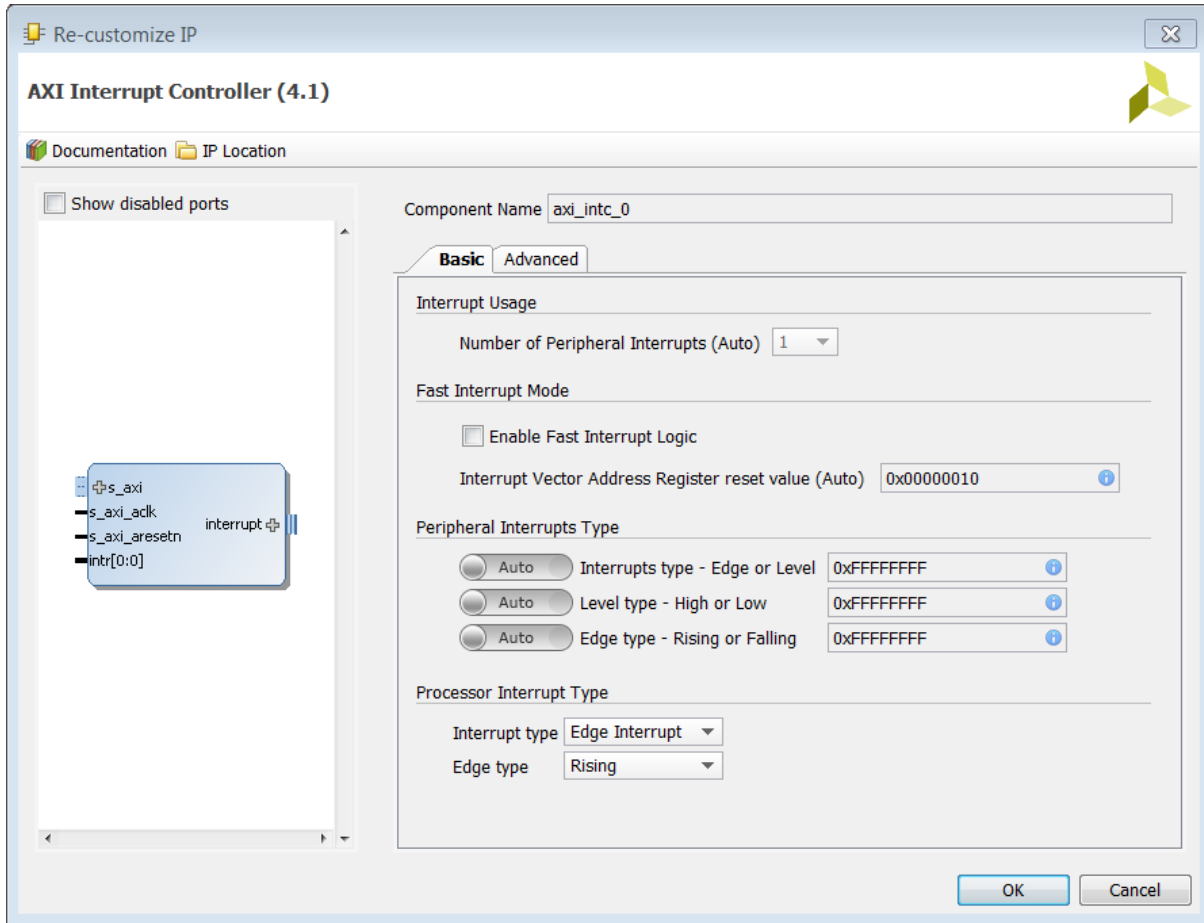


**Figure 27: The Re-customize IP dialog box of Concat**

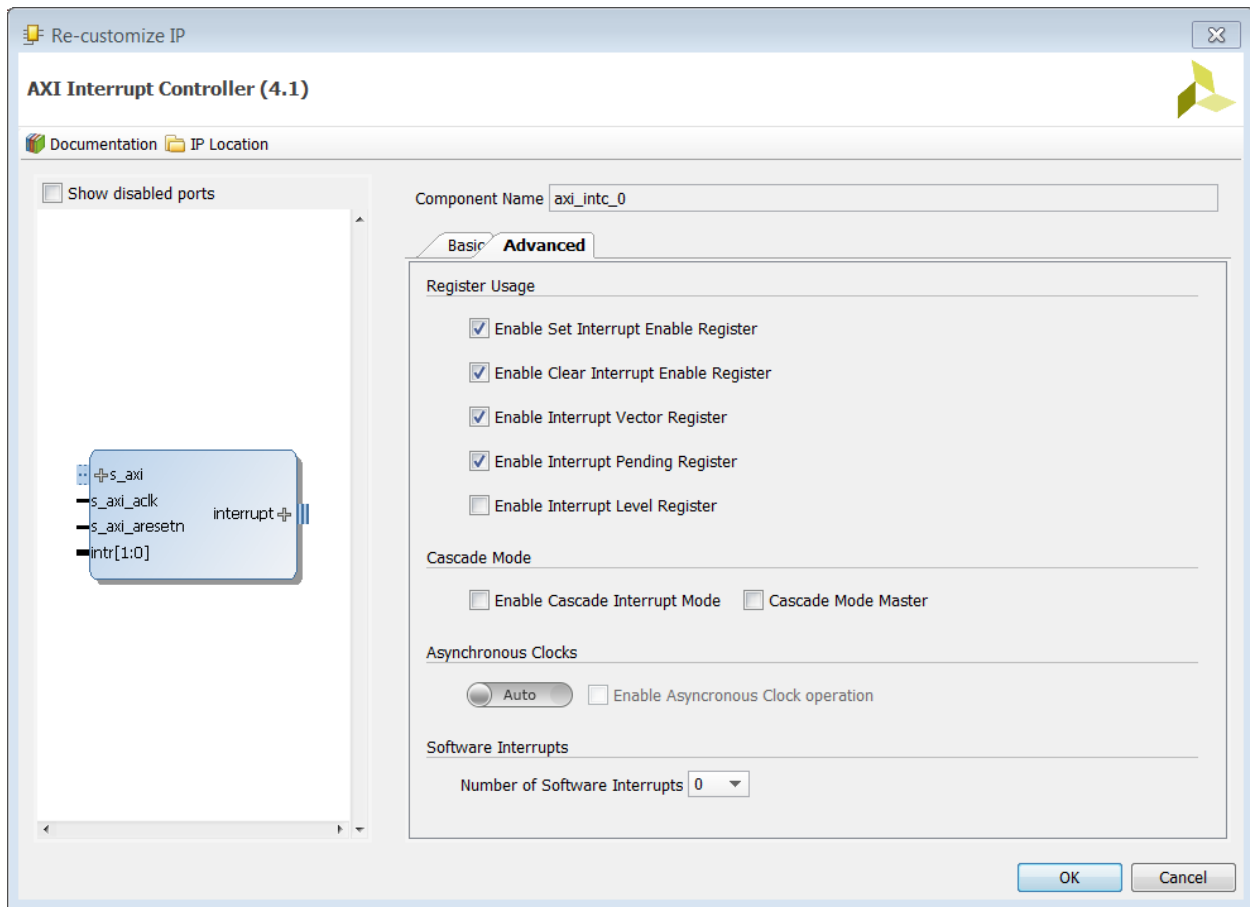
You can configure several of the parameters for the AXI Interrupt Controller. [Figure 28](#) shows the parameters available from the Basic tab of the AXI Interrupt Controller.

- 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 to Manual. In manual mode, users can specify the custom values in these fields.
- The **Processor Interrupt Type** field offers two choices: the **Interrupt type** and depending on the **Interrupt type** the other choice is **Level type** of **Edge Type**. As an example if the **Interrupt type** is **Edge Interrupt** then the other choice is **Edge Type**. If the **Interrupt type** is **Level Interrupt** then 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 if necessary.

**CAUTION!** You must ensure that the interrupt inputs of the Concat IP are all of the same type (e.g. Edge/Level triggered and Rising/Falling or Active High/Active Low respectively) when consolidating into the Interrupt Controller.



**Figure 28: Parameters on the Basic tab of the AXI Interrupt Controller**



**Figure 29: The Advanced tab options of the Interrupt Controller**

**Figure 29** shows parameters on the Advanced tab of the AXI Interrupt Controller. Refer to *LogiCORE IP AXI Interrupt Controller (PG099)* for details of these parameters.

One option worth mentioning here, 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 automatically enabled. In this case, cascading synchronizing registers are added to the interrupt sources.

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

## The Designer Assistance Feature of IP Integrator

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. You click on 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.

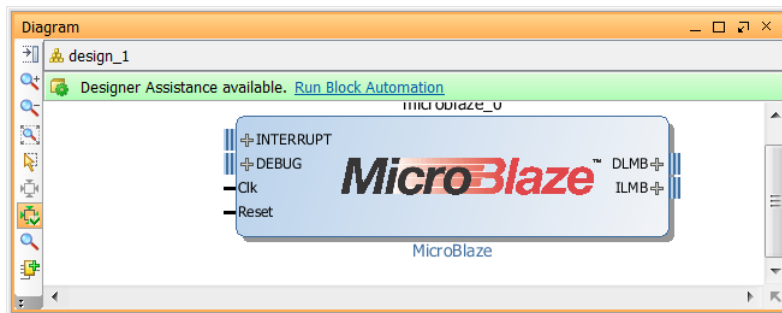


Figure 30: Run Block Automation

The **Run Block Automation** dialog box lets you provide input about basic features that the microprocessor system needs.

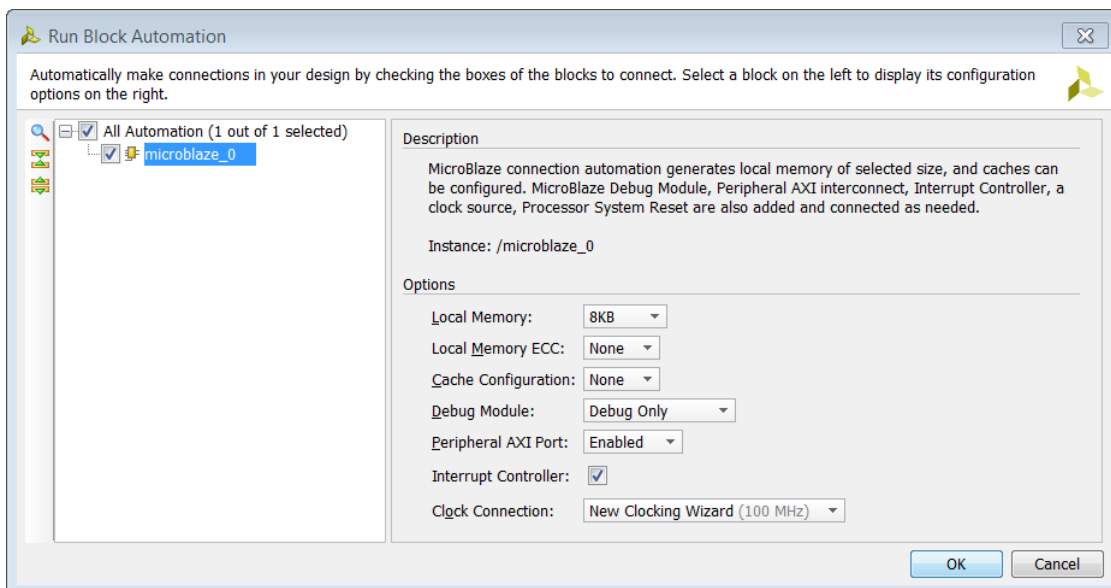
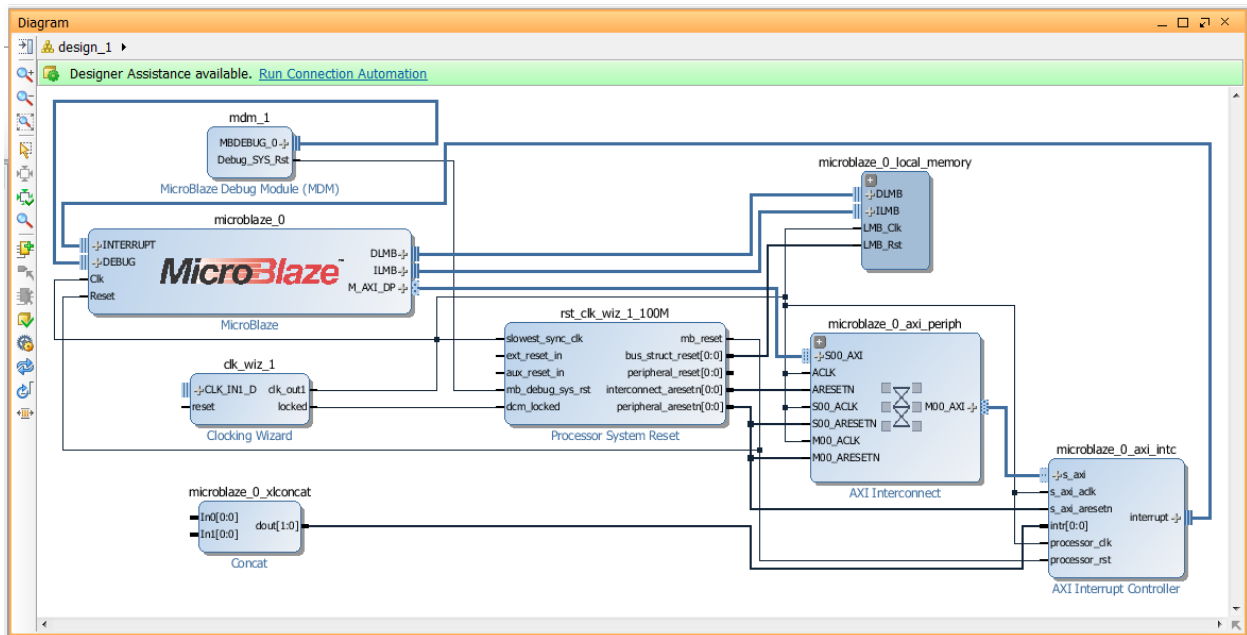


Figure 31: The Run Block Automation Dialog Box

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

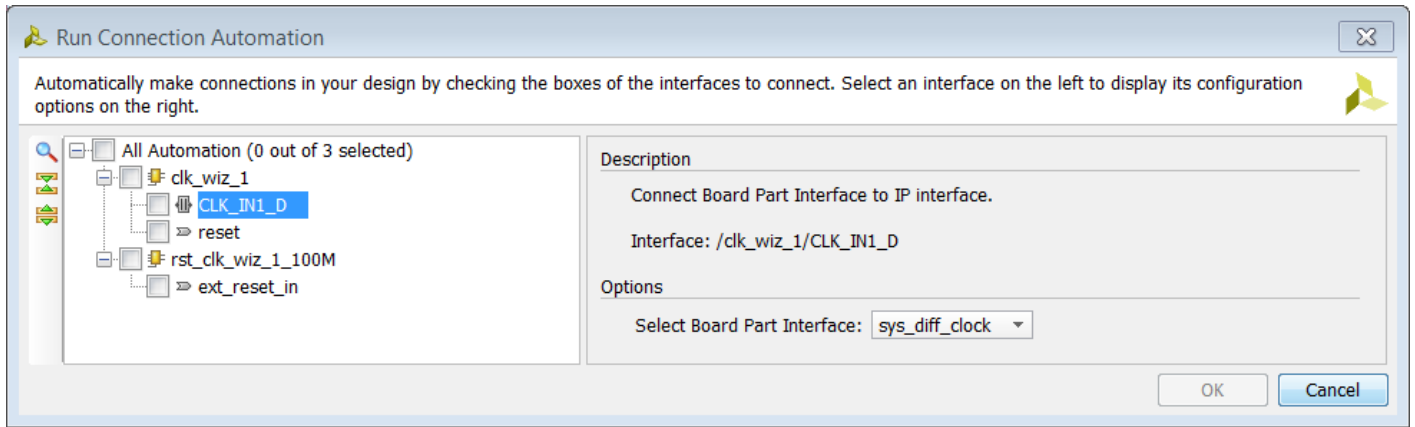


**Figure 32: A MicroBlaze System Created by Block Automation**

In this case, the MicroBlaze System that is created consists of a MicroBlaze Debug Module, a hierarchical block called the `microblaze_1_local_memory` that has the Local Memory Bus, the Local Memory Bus Controller and the Block Memory Generator, a Clocking Wizard, an AXI Interconnect and an AXI Interrupt Controller.

Since 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 on **Run Connection Automation**, IP integrator provides assistance in hooking interfaces and/or ports to external I/O ports.

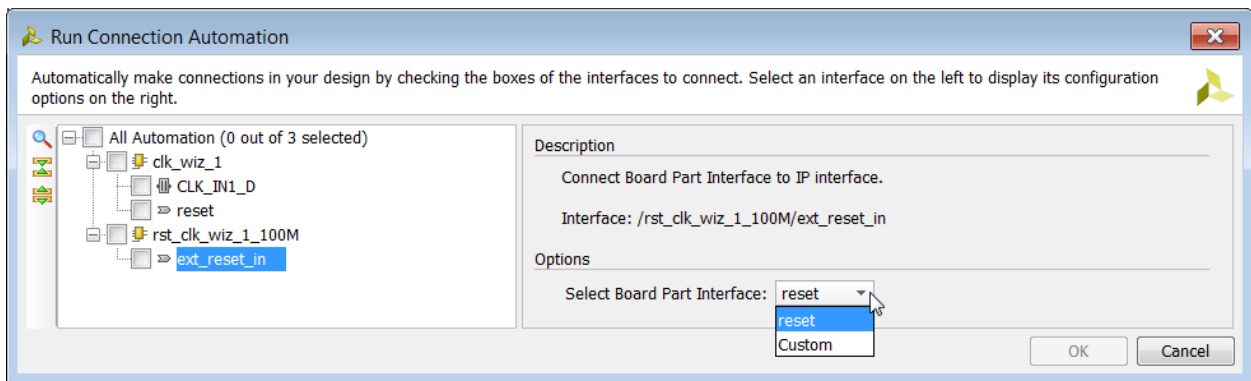
The Run Connection Automation dialog box, as shown in [Figure 33](#), lists ports and interfaces that the Connection Automation feature supports, along with a brief description of the available automation, and available options for each automation.



**Figure 33: Listing the Ports and Interfaces the can use Connection Automation**

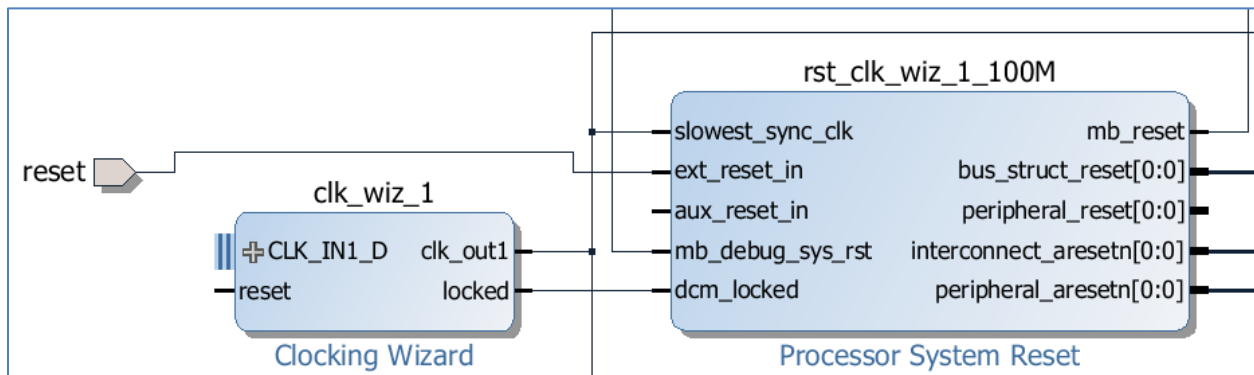
For Xilinx’s Target Reference Platforms or evaluation boards, IP integrator knows the FPGA pins that are used on the target boards. Based on that information, the IP integrator connection automation feature can assist you in tying the ports in the design to external ports on the board. IP integrator 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 32](#), the Processor System Reset IP needs to be connected to an external reset port, and the Clocking Wizard needs to be connected to an external clock source as well as an external reset. By selecting the appropriate options you can tie the clock and the reset ports to the appropriate sources on the target board.



**Figure 34: Run Connection Automation**

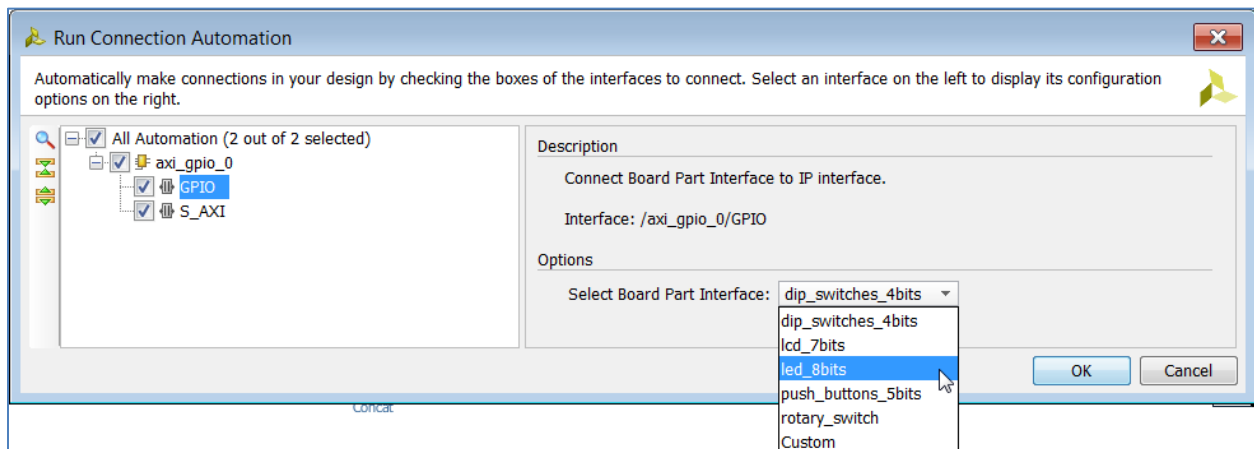
You can select the reset pin that already exists on the KC705 target board in this case, or you can specify a custom reset pin for your design. Once specified, the reset pin is tied to the `ext_reset_in` pin of the Proc Sys Rst IP.



**Figure 35: Connecting the reset Pin to the board Reset Pin**

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

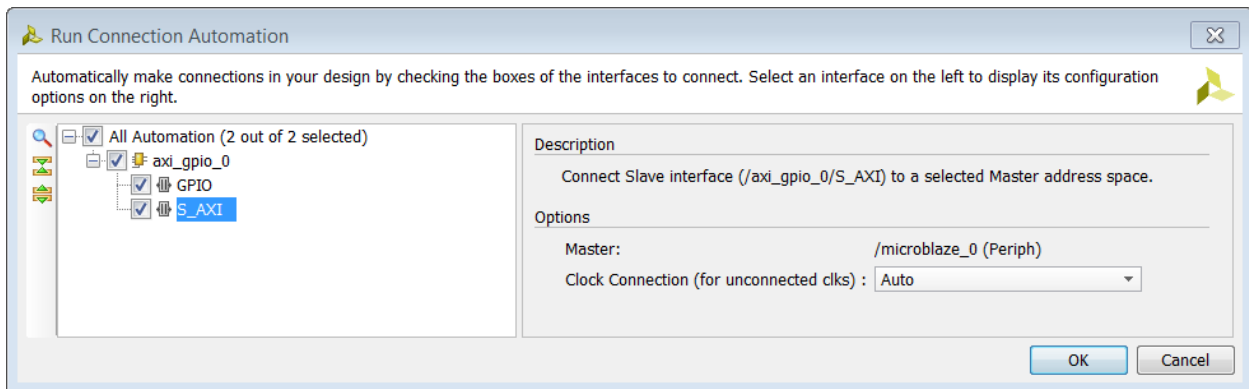
For example, assume that you instantiate the AXI GPIO IP into the design. The Run Connection Automation link reappears in the banner on top of the design canvas. You can then click on **Run Connection Automation** and the S\_AXI port of the newly added AXI GPIO can be connected to the MicroBlaze processor via the AXI Interconnect. Likewise the GPIO interface can be tied to one of the several interfaces present on the target board.



**Figure 36: Using Connection Automation to Show Potential Connections**

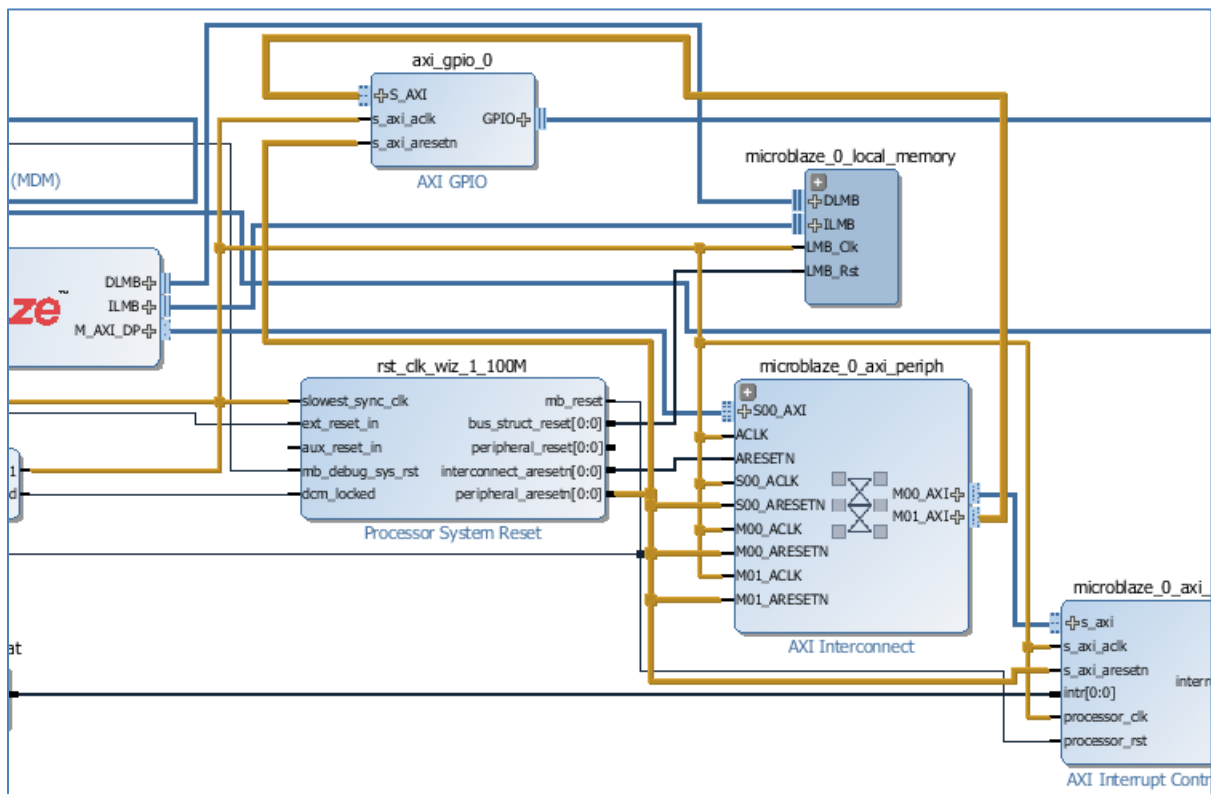
In this case, six different choices are presented. The **GPIO** interface port can be connected to either the **Dip Switches** that are 4-bits, or to the **LCD** that are 7-bits, **LEDs** that are 8-bits, 5-bits of **Push Buttons**, the **Rotary Switch** on the board, or can be connected to a custom interface. Selecting any one of the choices above will connect the **GPIO** port to the existing connections on the board.

Selecting the S\_AXI Interface for automation 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 will 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 37: Connecting the Slave Interface S\_AXI to the MicroBlaze Master**

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



**Figure 38: Master/Slave Connections**

Enhanced Designer Assistance is available for advanced users who want to connect a Streaming 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. Please refer to *Vivado Design Suite User Guide: Embedded Hardware Design (UG998)* for more information on this feature.



## Using the Signals Tab to Make Connections

IP integrator also provides for an easy way to make connections to clocks and resets via the Signals window. Once the block design is opened, the Signals window is displayed, as shown below, with two tabs listing the Clocks and Reset signals present in the design.

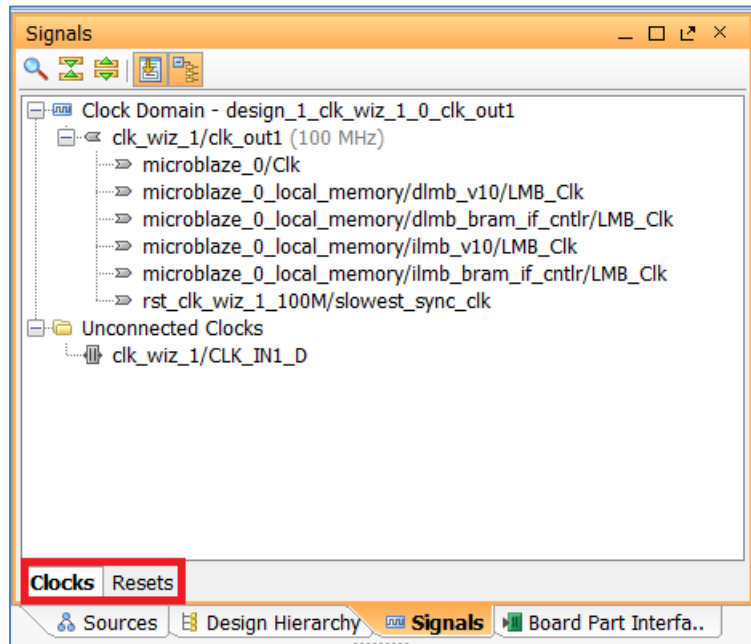


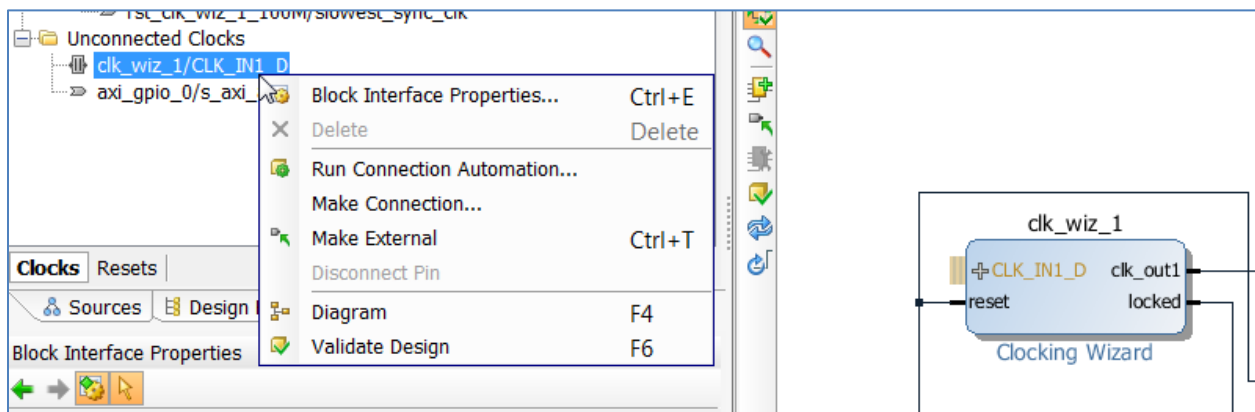
Figure 39: Signals window in IP Integrator

Selecting the appropriate tab shows all the clocks or resets in the design.

In the Clocks tab Clocks are listed based on the clock domain name. In the figure above, 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.

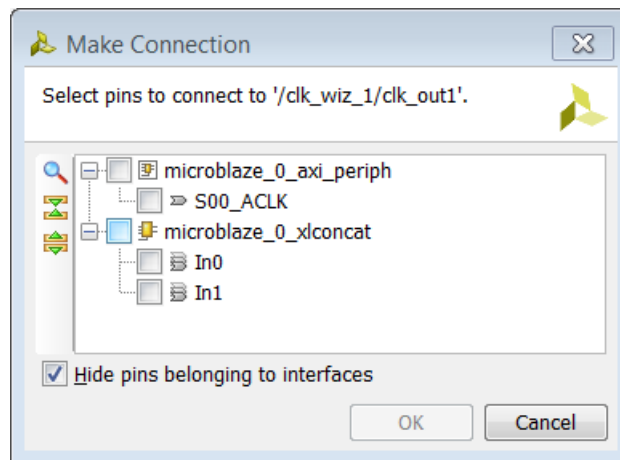
When you select a clock from the **Unconnected Clocks** folder, the respective clock port in the block design is highlighted. Right-clicking on the selected clock presents you with several options as shown in [Figure 40](#). In this case, Designer Assistance is available in the form of the **Run Connection 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** option, 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** option.

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



**Figure 40: Making Connection using the Signals tab**

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



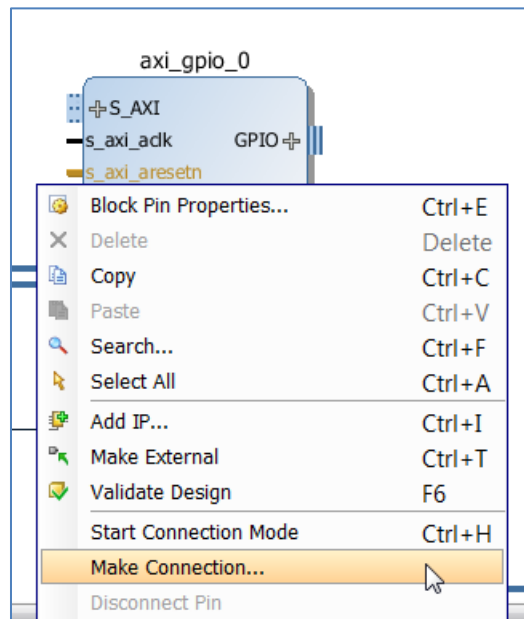
**Figure 41: Make Connection dialog box**

Selecting the appropriate clock source will make the connection between the clock source and the port/pin in question.

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

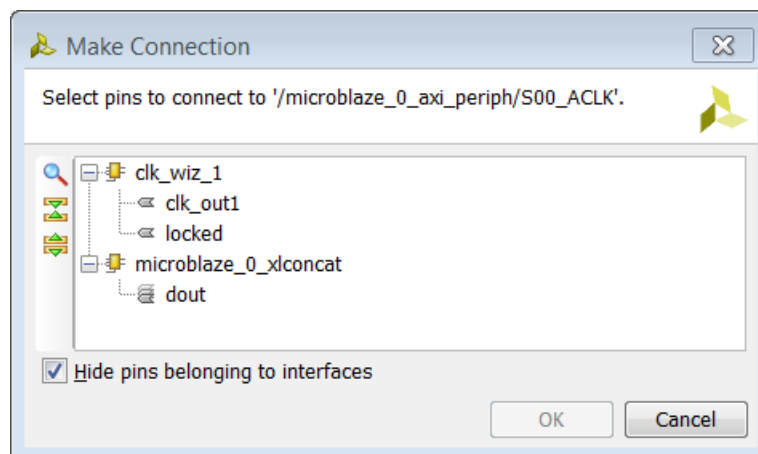
## Using Make Connections to Connect Ports and Pins

Connections to unconnected ports or pins can be made by selecting the port or pin in question and then selecting Make Connection from the right-click menu.



**Figure 42: Making Connections**

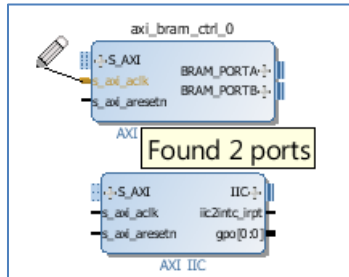
If a valid connection to the pin in question exists, then the **Make Connection** dialog box showing all the possible sources that the net can be connected to pops-up. From this dialog box you can select the appropriate source to drive the port/pin in question can be selected.



**Figure 43: Making Connections using the Make Connection option**

## Making Connections with Start Connection Mode

You can also make multiple connections at once by clicking on a pin and when the pencil shows up dragging and releasing the mouse.



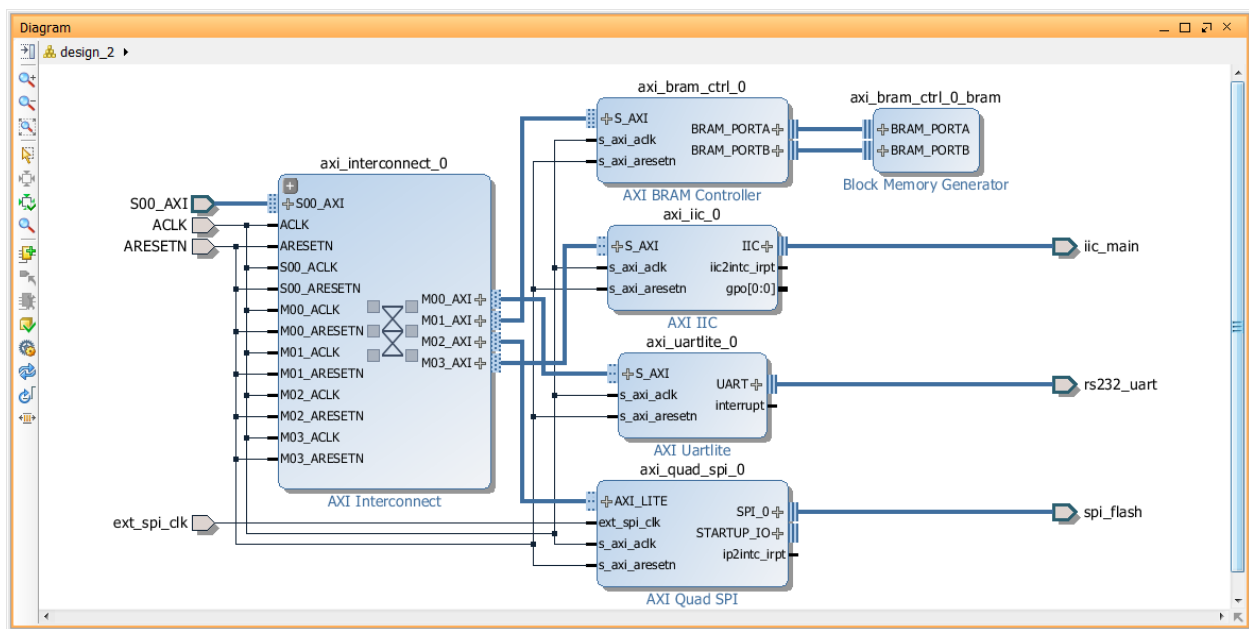
**Figure 44: Using Start Connection Mode to make connections**

After the connection is made to the `s_axi_aclk` pin of the AXI BRAM Controller in the figure above, the Start Connection Mode will offer to connect the pin to the `s_axi_aclk` pin of AXI IIC. In this way connections from the same source pin can be made to multiple different destinations at once.

## Interfacing with AXI IP Outside of the Block Design

There are situations when an AXI master is outside of the block design. These external masters are typically connected to the block design using an AXI Interconnect. Once the ports on the AXI interconnect are made "external", the address editor is available and memory mapping can be done in these cases.


As an example, consider the block design shown below.




**Figure 45: Example of a Design Consisting of an External AXI Master Interfacing the Block Design**

When the AXI interface of the Interconnect is made external, the Address Editor tab 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 on **Optimize Routing** . This preserves the placement of the blocks (unlike the Regenerate Layout function) and only modifies the routing to the various blocks. In other words, the optimize routing function keeps the location of different blocks intact and only modifies the nets connecting different blocks.

## Showing Interface Level Connectivity Only

To see only the connectivity between interfaces present on the block design select the **Show interface connections only** icon  from the block design toolbar. This shows only the interface level connections, and hides all the other connections on the block design.

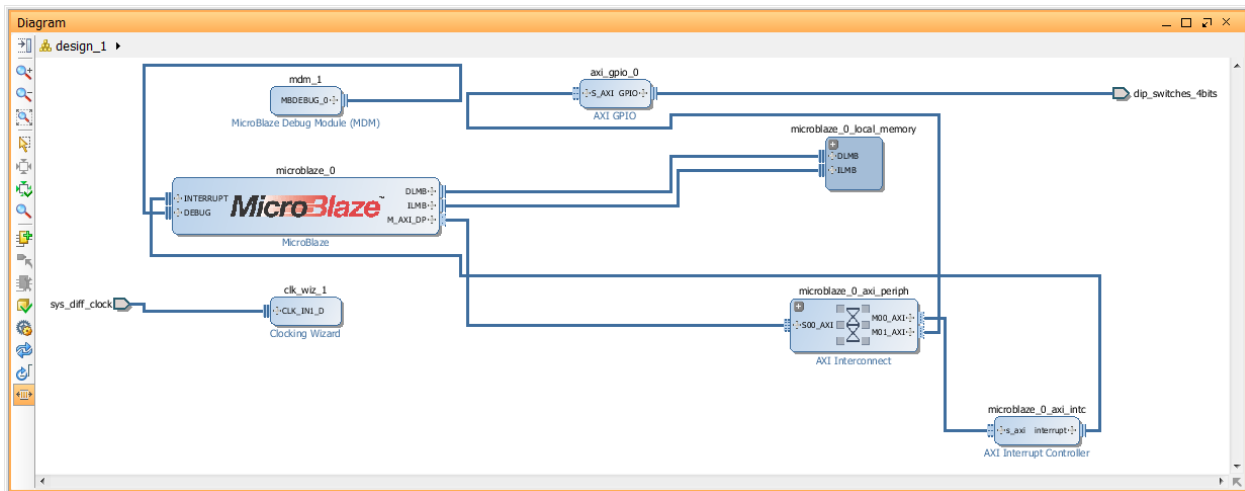
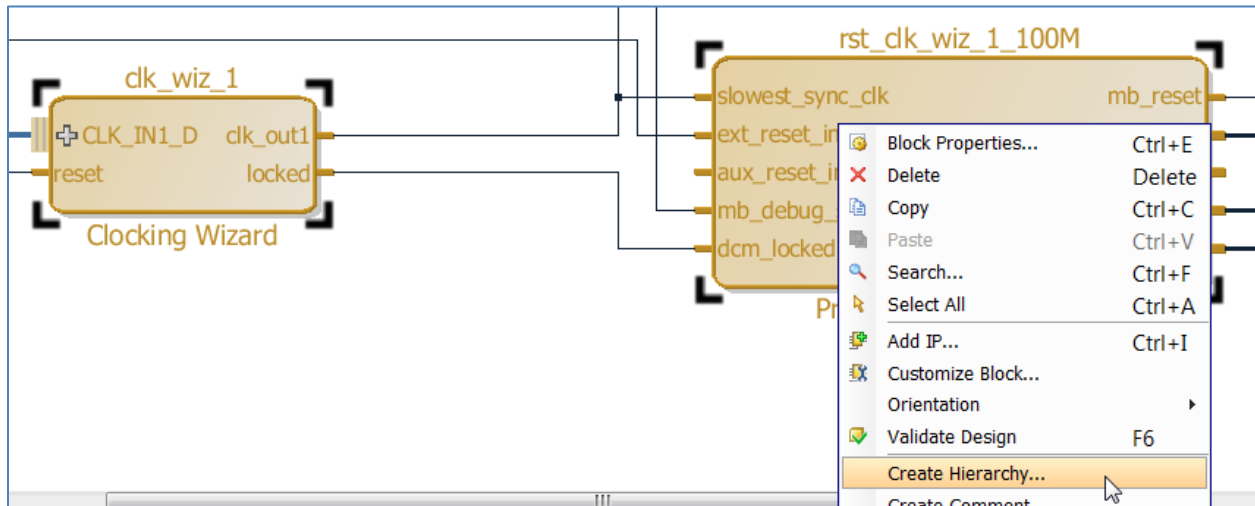


Figure 46: Showing interface connections only

Clicking on the **Show interface connections only** icon 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.

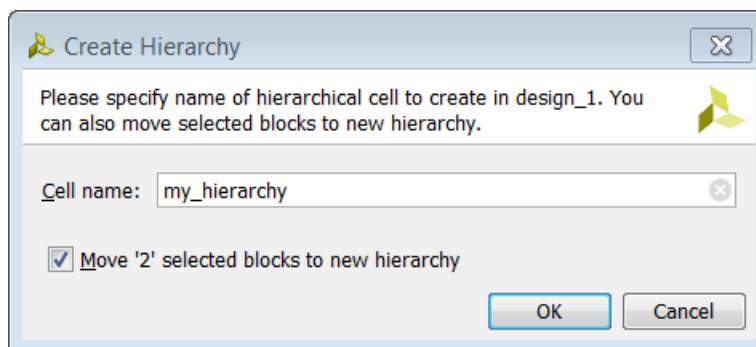


**Figure 47: Creating a Hierarchical Block Design**

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.

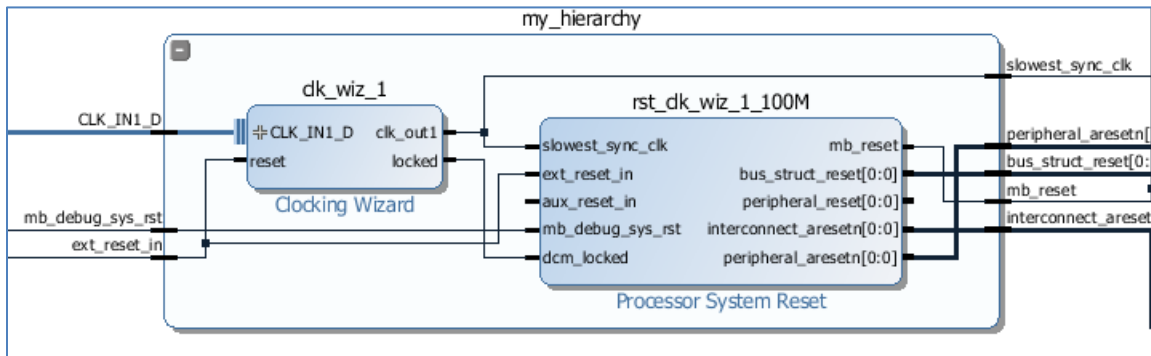
Hierarchies can be expanded when you click on the + sign in the upper-left corner of an expandable block. You can traverse levels of hierarchy in a diagram using the Explorer type path information displayed in the upper-left corner of the IP integrator diagram.

Clicking on **Create Hierarchy** pops-up the Create Hierarchy dialog box, as shown below, where you can specify the name of the new hierarchy.



**Figure 48: The Create Hierarchy Dialog Box**

This action groups the selected IP blocks under one block, as shown below. You can click on the + sign of the hierarchy to view the components underneath. Click on the – sign on the expanded hierarchy to collapse it back to the grouped form.

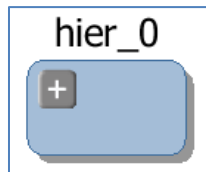


**Figure 49: Grouping Two Blocks into One Block**

## Adding Pins and Interfaces to Hierarchies

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

Right-click 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 50: Creating an 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 reset of rst type 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
```

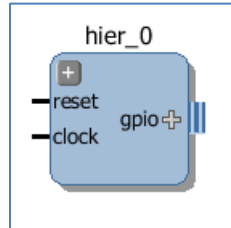
You can also add interfaces to a hierarchy by using the following Tcl commands. First set the block design instance to the appropriate hierarchy where the interface is to be added, using the following command:

```
current_bd_instance /hier_0
```

Next, create the interface using command as specified below:

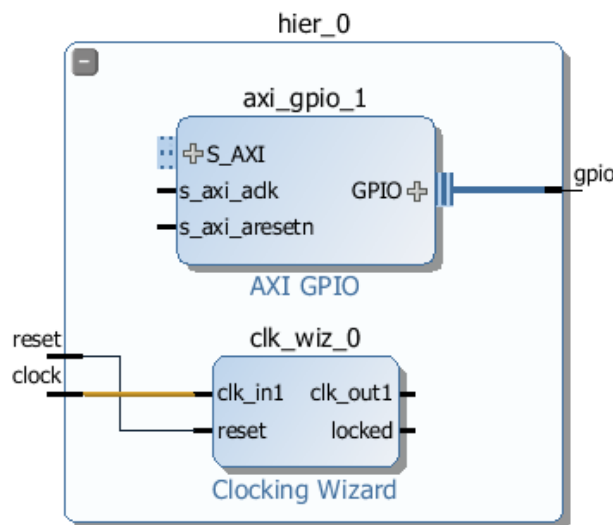
```
create_bd_intf_pin -mode Master -vlnv xilinx.com:interface:gpio_rtl:1.0 gpio
```

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 51: Create Pins in a Hierarchy**

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



**Figure 52: Making Connections of IP to the 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 on **Validate Design**.

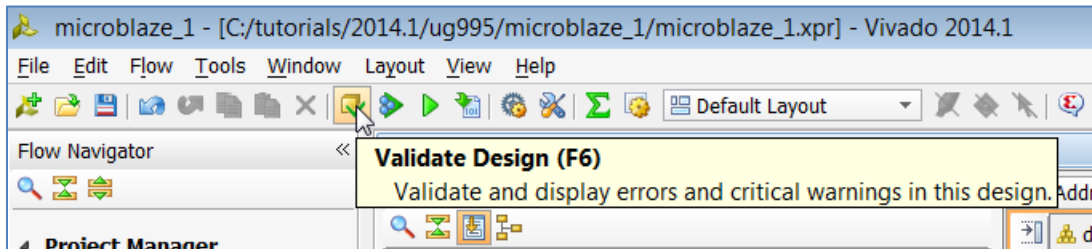



Figure 53: Validating the Design

Design validation can also be run by clicking on the Validate Design icon  in the toolbar on the IP integrator canvas. If the design is free of Warnings and/or Errors, a pop-dialog box such as that shown in the figure below is displayed after running **Validate Design**.

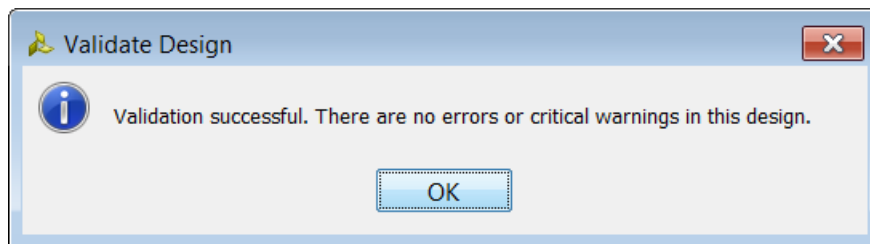


Figure 54: Successful Validation Message

---


### Overview

Master interfaces reference an assigned memory range container called an address spaces, or address\_space objects. Slave interfaces reference a requested memory range container, called a memory map. By convention memory maps are usually 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. Also, by convention, Address Space names often are related to its usage, for example, the MicroBlaze has a **Data** address space and an **Instruction** address space.

The memory map for each slave interface pin contains address segments, or slave\_segment objects. These address segments correspond to the address decode window for that slave. A typical AXI4-Lite slave will have only one address segment, representing a range of memory. However, some slaves, like a bridge, will 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 address\_segment 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 tab only appears if the diagram contains an IP block that functions as a bus master (such as the MicroBlaze processor in the following diagram) or an external bus master (outside of IP integrator) is present.*

---

You can click on the **Address Editor** tab 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, a prompt will let you automatically assign addresses at that point.

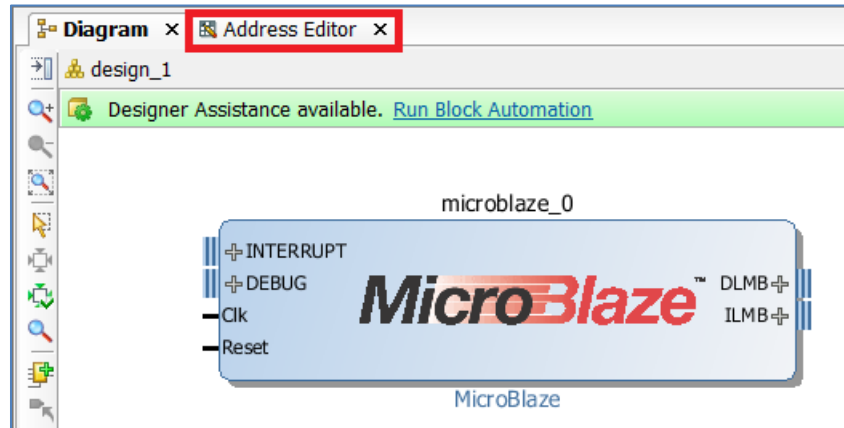
You can also manually set addresses by entering values in the **Offset Address** and **Range** columns.

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.

The IP Integrator tool 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.

## The Address Editor in IP Integrator

The Address in the IP Integrator tool is used to allocate memory ranges to peripherals from a master's perspective. The Address Editor tab becomes available when a master with an address space, such as the MicroBlaze or the Zynq processor is instantiated in the Diagram canvas.




**Figure 55: The Address Editor Tab**

As the peripherals are instantiated and connected to the processor in the block design canvas using connection automation a corresponding memory assignment is automatically entered for that peripheral in the Address Editor.

| Cell                                  | Slave Interface | Base Name | Offset Address | Range | High Address |
|---------------------------------------|-----------------|-----------|----------------|-------|--------------|
| microblaze_0                          |                 |           |                |       |              |
| Data (32 address bits : 4G)           |                 |           |                |       |              |
| microblaze_0_local_memory/dlmb_br...  | SLMB            | Mem       | 0x00000000     | 8K    | 0x00001FFF   |
| microblaze_0_axi_intc                 | s_axi           | Reg       | 0x41200000     | 64K   | 0x4120FFFF   |
| axi_bram_ctrl_0                       | S_AXI           | Mem0      | 0xC0000000     | 8K    | 0xC0001FFF   |
| Instruction (32 address bits : 4G)    |                 |           |                |       |              |
| microblaze_0_local_memory/ilmb_bra... | SLMB            | Mem       | 0x00000000     | 8K    | 0x00001FFF   |

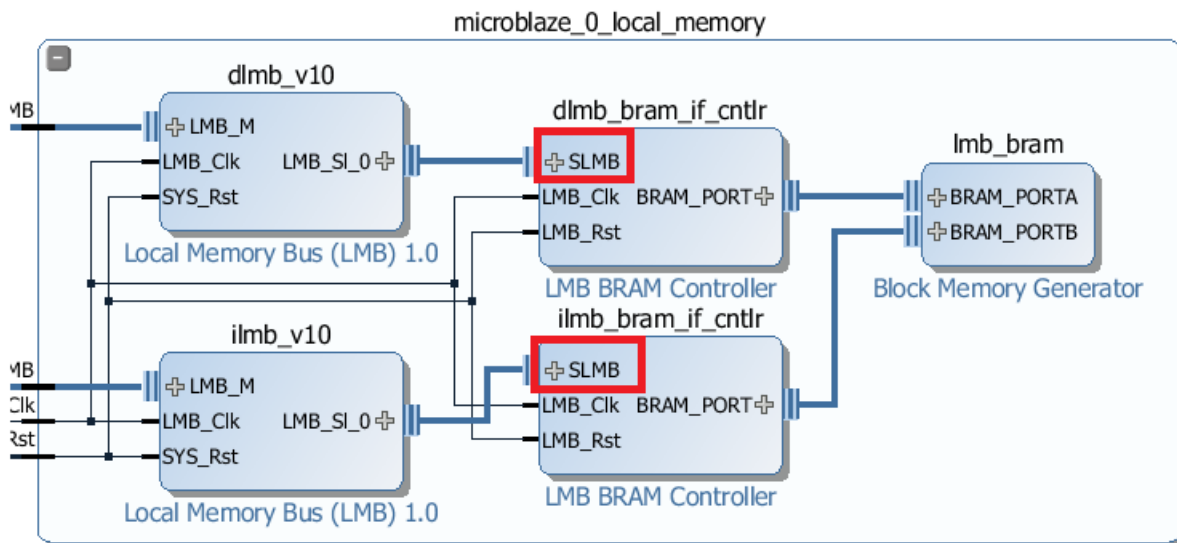
**Figure 56: Memory map of peripherals in Address Editor**

Following are the descriptions of various columns of the Address Editor.

**Cell** – This column describes the master and all the peripherals that are connected to it and can thus be addressed by it. The tree can be expanded by clicking on the Expand All icon  or by clicking on the + sign to expand the selection. As an example in [Figure 56](#) the instance name of the “master” is microblaze\_0 which addresses the Data and Instruction address spaces.

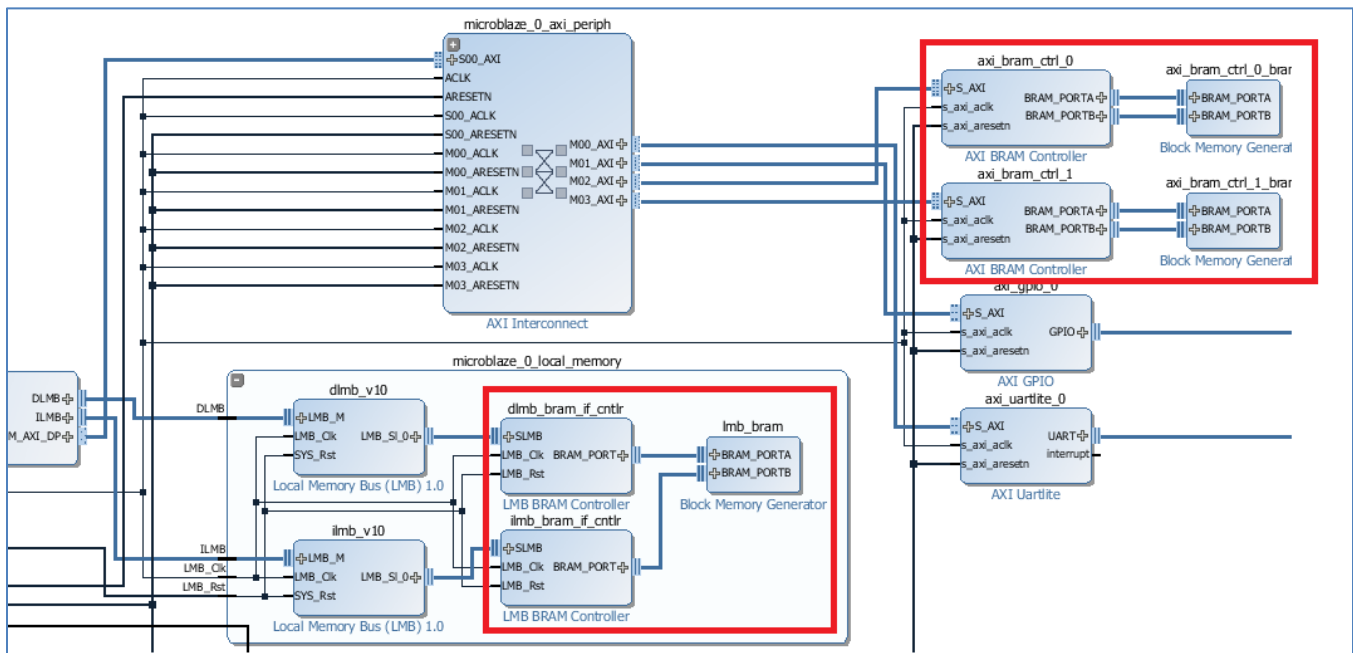
The peripherals `microblaze_0_local_memory/dlmb_bram_if_cntlr` and `microblaze_0_local_memory/ilmb_bram_if_cntlr` have been mapped into the Data and Instruction address spaces respectively, whereas, the rest of the peripheral are only accessible by the Data address space.

**Slave Interface** – This column 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 57](#).



**Figure 57: Interface Names listed in the Slave Interface column**

**Base Name** – This column specifies the name of the slave segment. By convention, there are two types of names created on the fly. These are Mem (memory) and Reg (register). This can be seen in [Figure 58](#), which shows a design with multiple memory instantiations.



**Figure 58: Multiple Memory Instantiations in a block design**

These are given the base names in the address editor as shown in [Figure 59](#).

| Cell   | Slave Interface | Base Name | Offset Address | Range | High Address |
|--|-----------------|-----------|----------------|-------|--------------|
| microblaze_0                                 |                 |           |                |       |              |
| Data (32 address bits : 4G)                  |                 |           |                |       |              |
| microblaze_0_local_memory/dlmb_bram_if_cntrl | SLMB            | Mem       | 0x00000000     | 8K    | 0x00001FFF   |
| axi_uartlite_0                               | S_AXI           | Reg       | 0x40600000     | 64K   | 0x4060FFFF   |
| axi_gpio_0                                   | S_AXI           | Reg       | 0x40000000     | 64K   | 0x4000FFFF   |
| axi_bram_ctrl_0                              | S_AXI           | Mem0      | 0xC0000000     | 8K    | 0xC0001FFF   |
| axi_bram_ctrl_1                              | S_AXI           | Mem0      | 0xC2000000     | 8K    | 0xC2001FFF   |
| Instruction (32 address bits : 4G)           |                 |           |                |       |              |
| microblaze_0_local_memory/ilmb_bram_if_cntrl | SLMB            | Mem       | 0x00000000     | 8K    | 0x00001FFF   |

**Figure 59: Base Names given to multiple memory instantiations**

**Offset Address** – This field 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 59](#). 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 on each other. The Offset 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 following type of error message will be seen.

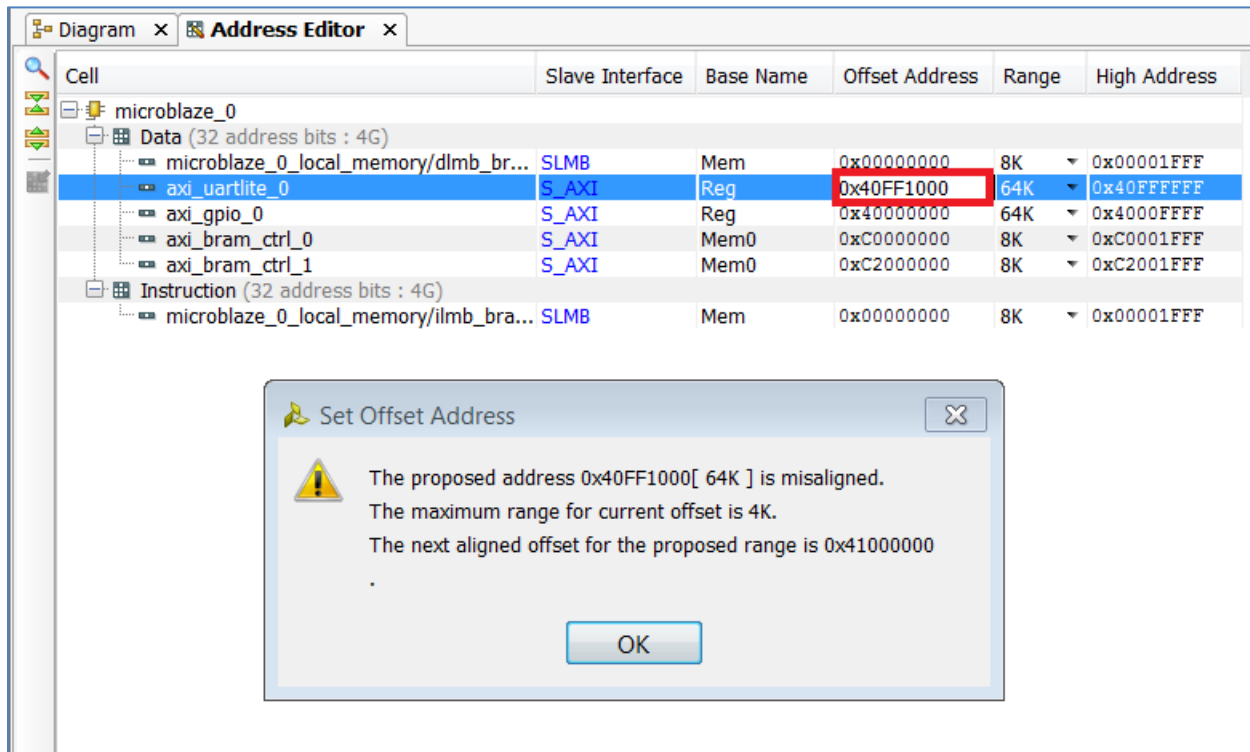
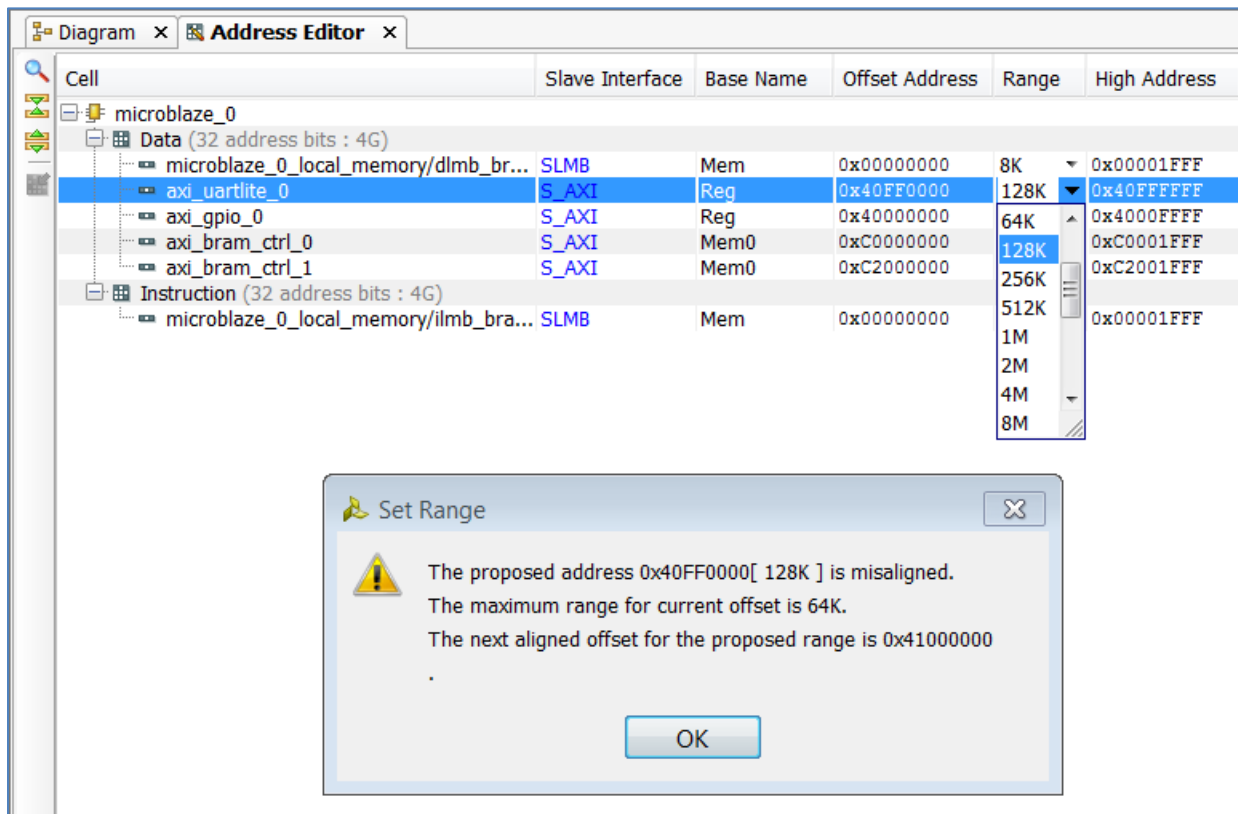


Figure 60: Example of misaligned Offset Address

In [Figure 60](#), the user set an offset 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** – This field 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 on the drop-down menu and selecting the appropriate value for this field. The Range and the Offset Address fields are interdependent on each other and as described in the Offset field, the  $2^N$  Range field must be aligned with the N number of least significant bits of the Offset field. Setting the Range field such that it exceeds this number can cause the following type of message to show up.



**Figure 61: Example of not setting the range field properly**

In [Figure 61](#), the user tried to set the range to 128K or  $2^{17}$ , for an offset in the form  $0xXXXX0000$  i.e. an offset with only 16 least significant bits. In order to accommodate a range of 128K, the form of the address must be at least  $0xXXXX20000$ , i.e. with at least 17-bits in the least significant bits of the starting offset.

**High Address** – This field 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, they 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 on the Auto Assign icon on the block design tool bar as shown in [Figure 62](#).

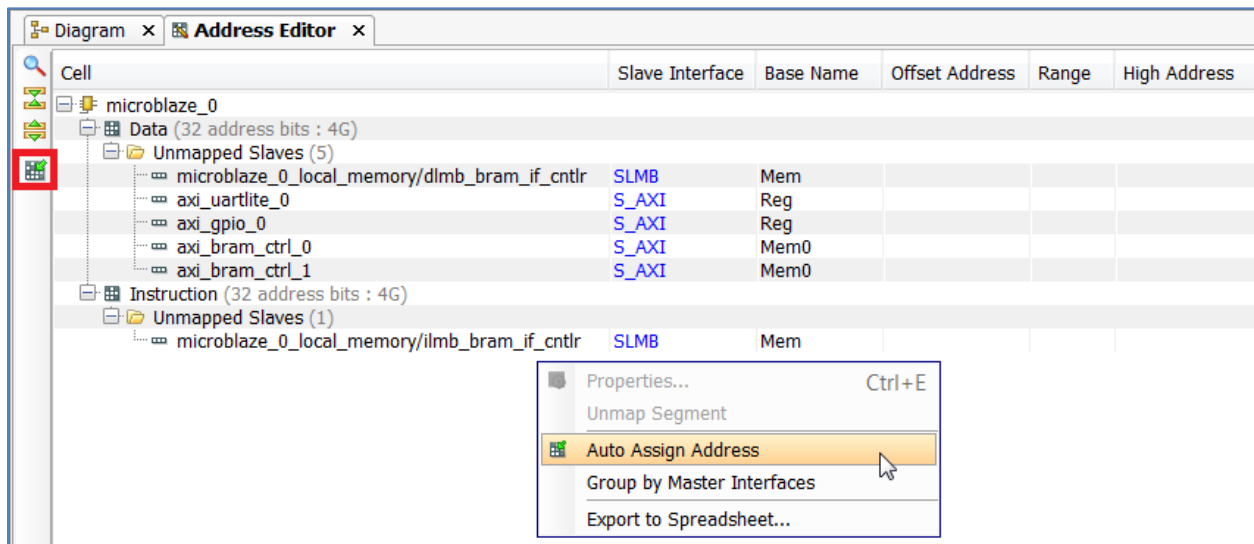


Figure 62: Auto Assign Address

This will map all the slave segments as shown in [Figure 63](#).

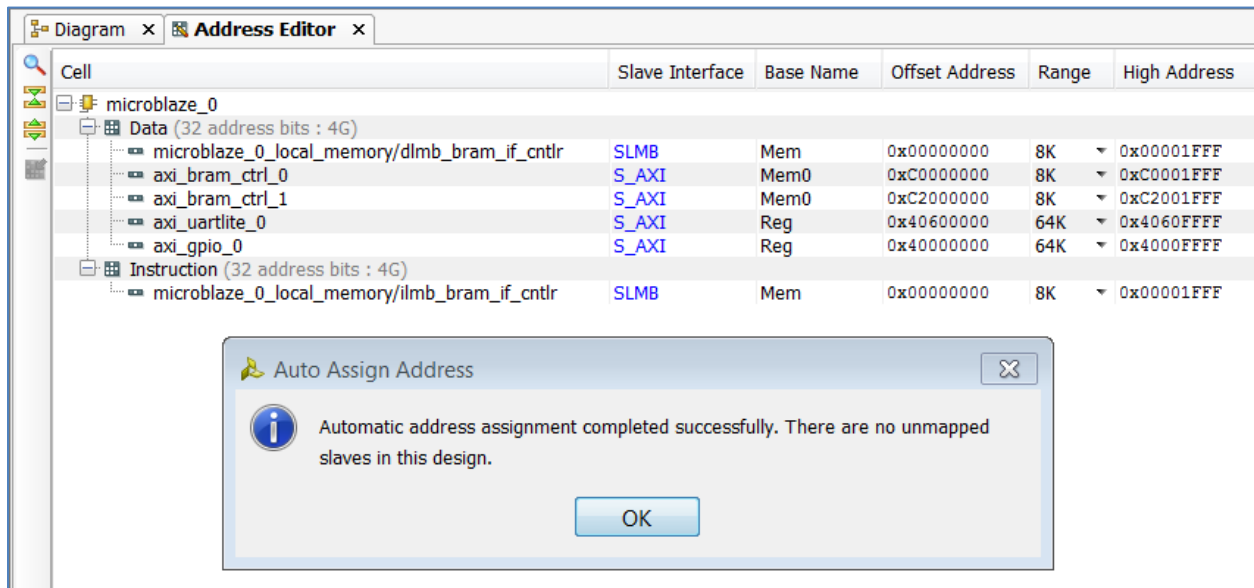


Figure 63: Memory map after auto assignment of address blocks

Once the slave segments have been mapped several options are presented to the user for other actions.



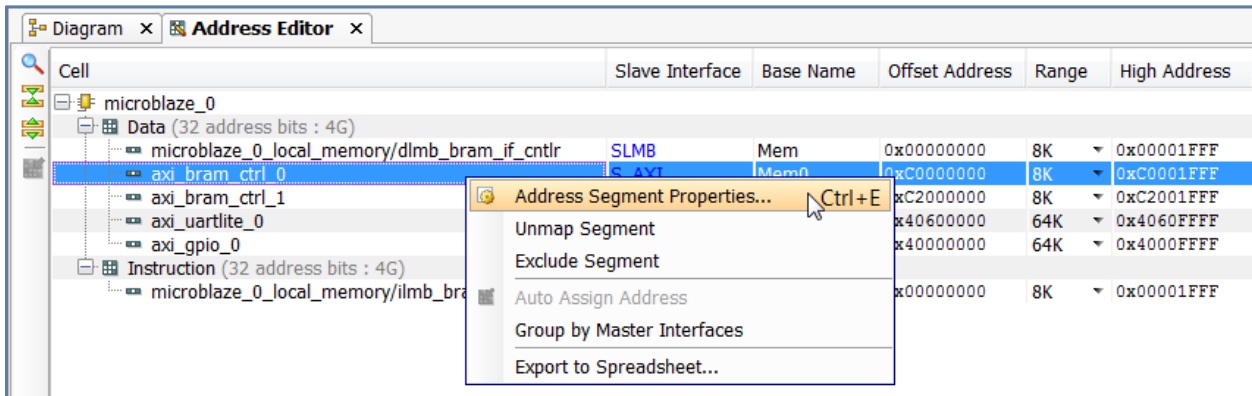


Figure 64: Address Editor Options

Right-clicking on a mapped address segment shows the various options available in the context menu as shown in [Figure 64](#).

### Address Segment Properties

The Address Segment Properties shows the details of the address segment in the Address Segment Properties window.

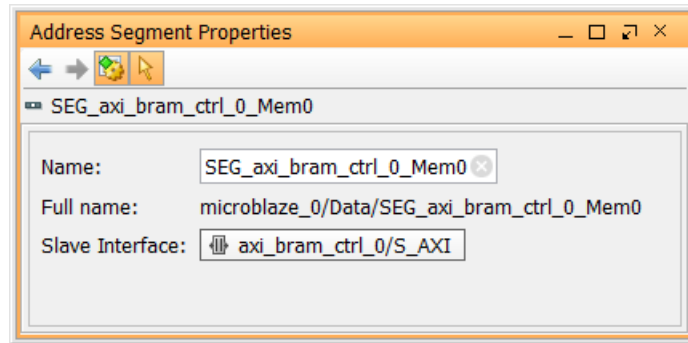


Figure 65: Address Segment Properties window

The Name field shows the name of the master segment that was automatically assigned. This name can be changed by the user if desired. The Full Name field is not editable and shows the full name of the mapped slave segment. The Slave Interface Field 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 66](#). The user can also right-click and select **Assign Address** (which will map only the selected address) or **Auto Assign Address** (which will assign all unmapped address segments in the design).

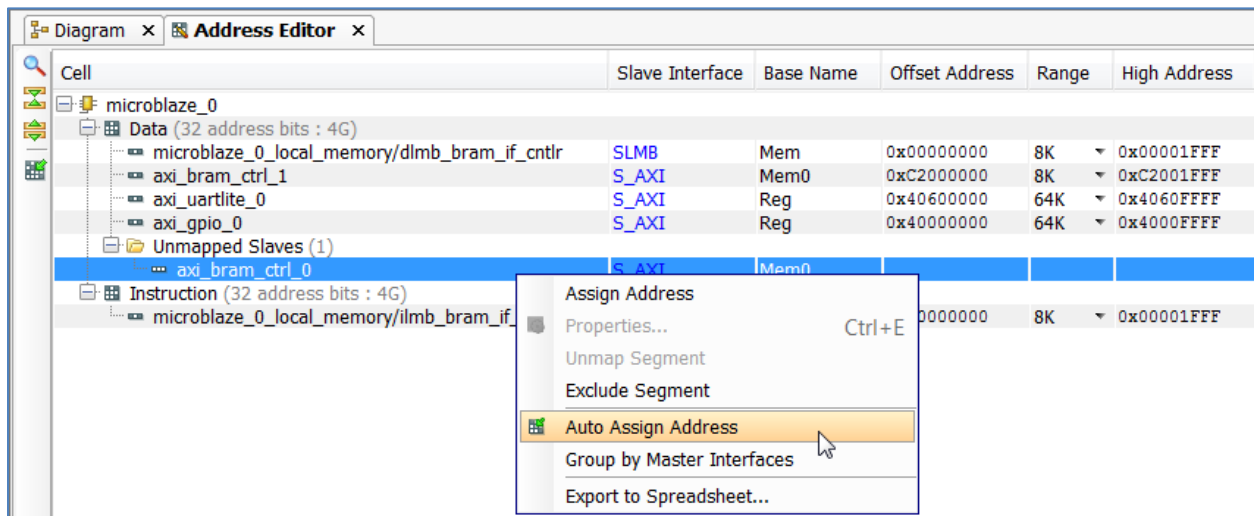


Figure 66: Unmapping and Mapping an address segment

### Exclude Segment

Excluding a segment makes a mapped segment un-addressable 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. Please see Sparse Connectivity for more information.

### Group by Master Interfaces

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

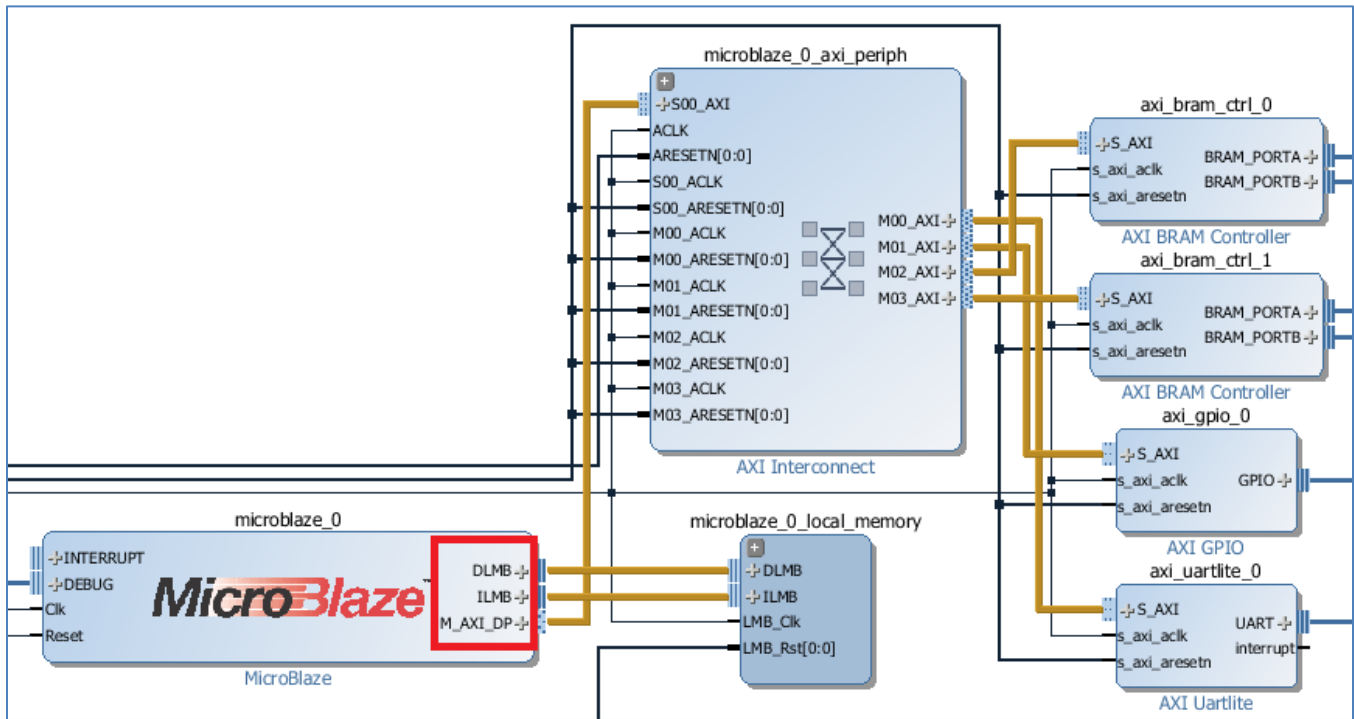


Figure 67: Grouping by master interfaces

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

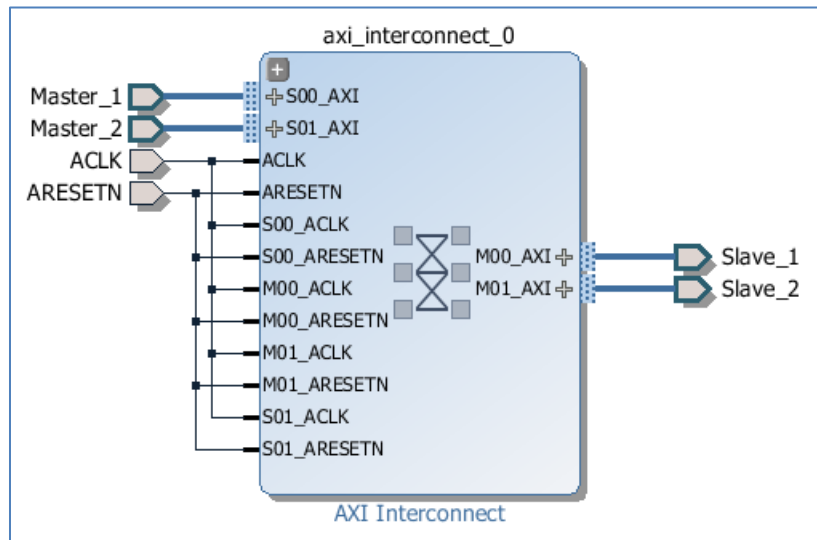
| Cell   | Slave Interface | Base Name | Offset Address | Range | High Address |
|--|-----------------|-----------|----------------|-------|--------------|
| microblaze_0                                 |                 |           |                |       |              |
| Data (32 address bits : 4G)                  |                 |           |                |       |              |
| DLMB   |                 |           |                |       |              |
| microblaze_0_local_memory/dlmb_bram_if_cntlr | SLMB            | Mem       | 0x00000000     | 8K    | 0x00001FFF   |
| M_AXI_DP                                     |                 |           |                |       |              |
| axi_bram_ctrl_1                              | S_AXI           | Mem0      | 0xC2000000     | 8K    | 0xC2001FFF   |
| axi_uartlite_0                               | S_AXI           | Reg       | 0x40600000     | 64K   | 0x4060FFFF   |
| axi_gpio_0                                   | S_AXI           | Reg       | 0x40000000     | 64K   | 0x4000FFFF   |
| axi_bram_ctrl_0                              | S_AXI           | Mem0      | 0xC0000000     | 8K    | 0xC0001FFF   |
| Instruction (32 address bits : 4G)           |                 |           |                |       |              |
| ILMB   |                 |           |                |       |              |
| microblaze_0_local_memory/ilmb_bram_if_cntlr | SLMB            | Mem       | 0x00000000     | 8K    | 0x00001FFF   |

Figure 68: Address Editor showing groups of address segments under respective master interfaces

## Sparse Connectivity

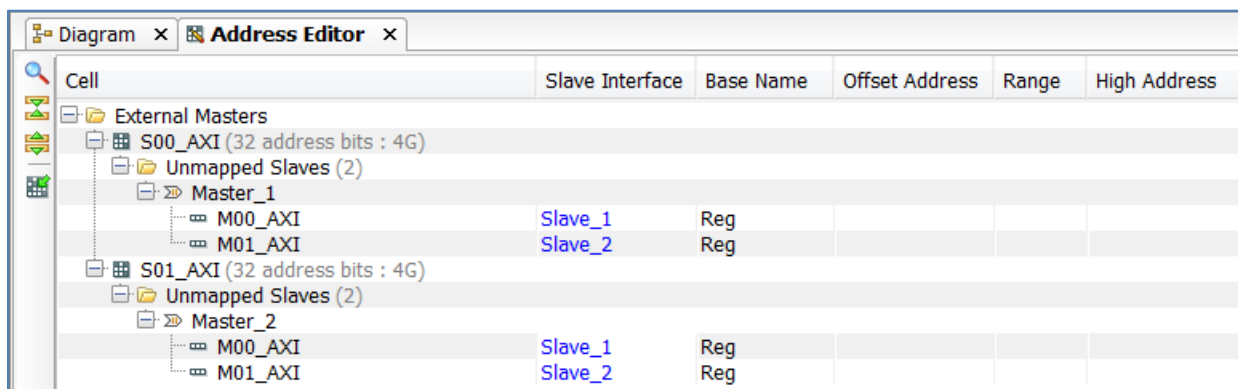
In a multiple master design users may want to specify slaves that could potentially be accessed by all masters or by certain masters only. This feature of memory mapping in IP Integrator is called sparse connectivity.

### Excluding an address segment from a master's memory map



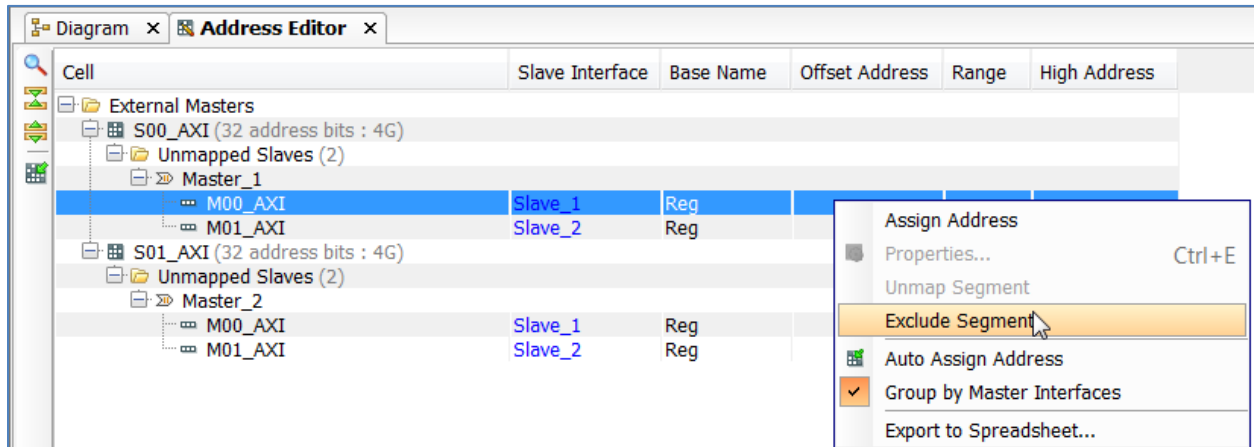
**Figure 69: Multiple Master and Slave Example**

In [Figure 69](#), there are two masters, Master\_1 and Master\_2 accessing two slaves Slave\_1 and Slave\_2 via the same interconnect. In this case, the address editor will look as follows.



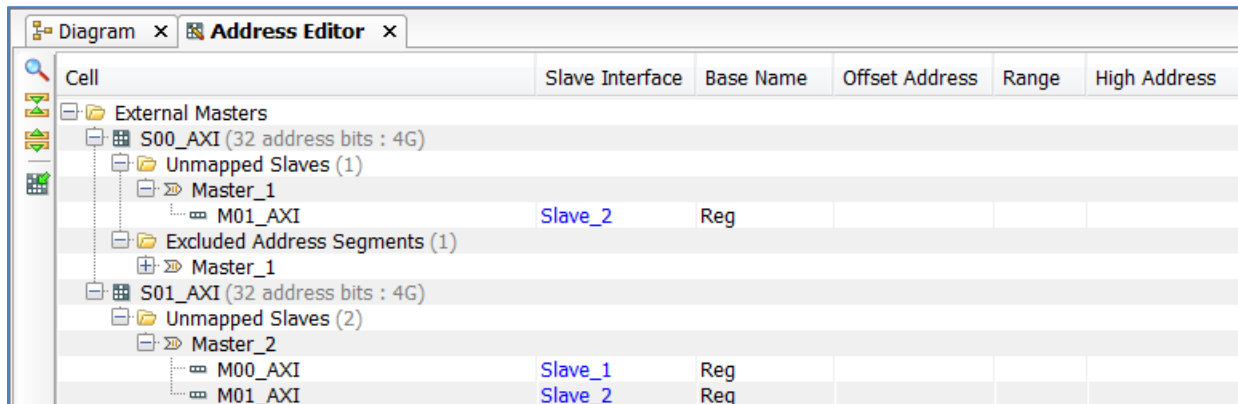
**Figure 70: Address Editor with multiple masters' memory map**

In this case, let us say that Master\_1 needs to access Slave\_2 only and Master\_2 needs to access both Slave\_1 and Slave\_2. In order to exclude Slave\_1 from Master\_1's memory map, right-click on M00\_AXI and select Exclude Segment from the context menu.



**Figure 71: Excluding address segment from the memory map of a master**

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



**Figure 72: Excluded Address Segment shown in Address Editor**

Both mapped and unmapped slaves can be excluded. It is important to note that an excluded master segment still occupies a range within the address space despite the fact that it is inaccessible by the master. If, after excluding a slave within a master address space, the user attempts to manually place another slave to address that overlaps with the excluded slave, an error will then be thrown 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 [Figure 73](#).

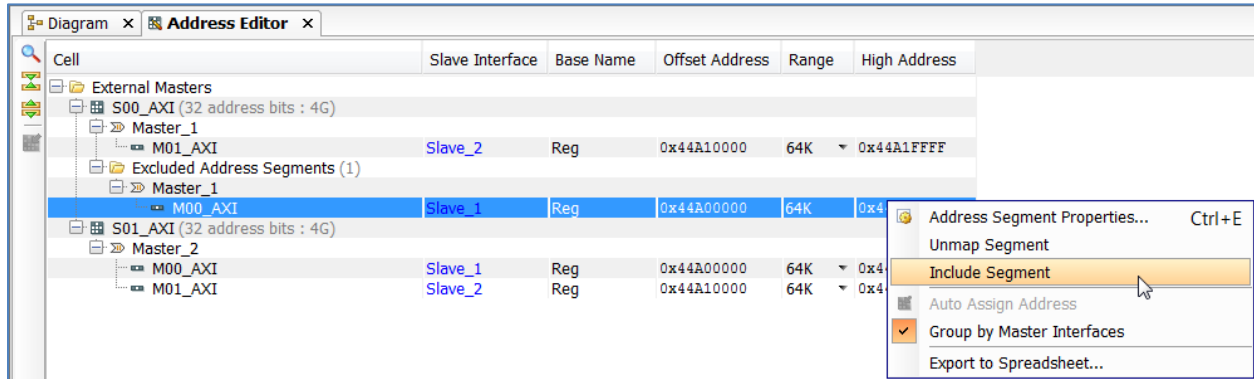


Figure 73: Including an excluded segment back into the master’s memory map

### Common Addressing related Critical Warnings and Errors

1. [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.

2. [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.

3. [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.

### Overview

With the block design populated with IP, connected, with connections external to the design, and validated. You can work with the block design to generate the output products needed to use the block design in your top-level design (or as your top-level design) for simulation, synthesis, and implementation.

### Generating Output Products

Once the block design is complete and the design is validated, output products must be generated. This is when the source files and the appropriate constraints for all the IP will be generated and made available in the Vivado Sources pane. Depending upon the target language selected during project creation, appropriate files will be generated. If the source files for a particular IP cannot be generated in the target language specified, then a message stating so will be displayed in the Tcl Console. To generate output products, in the Vivado sources pane, you right click on the block design, as shown in the following figure, and select **Generate Output Products**.

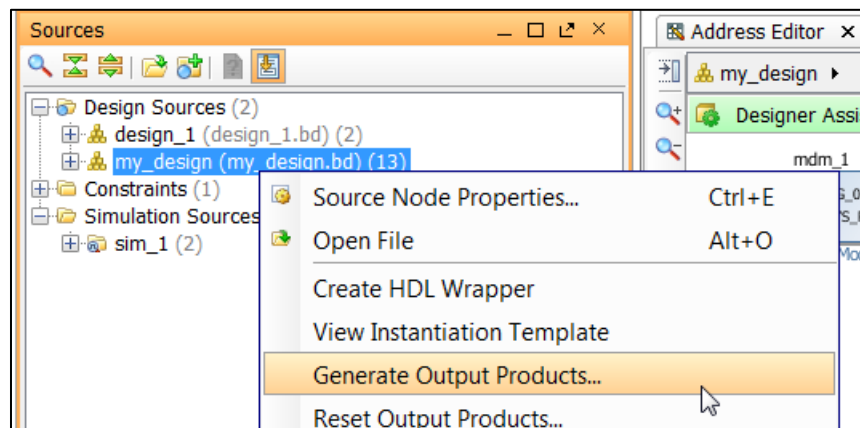


Figure 74: Generating Output Products

Alternatively, you can also click on **Generate Block Design** in the Flow Navigator under IP Integrator drop-down list.

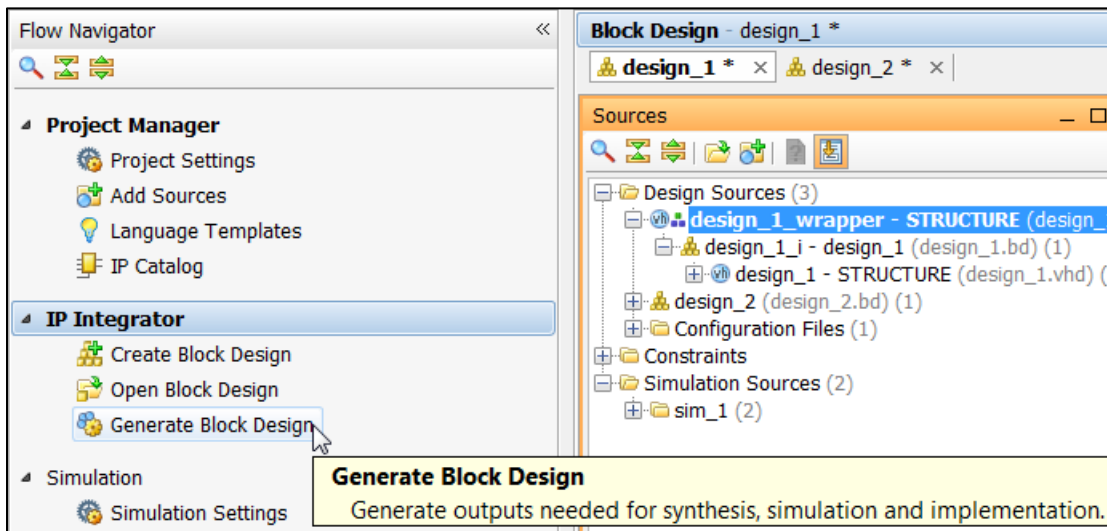


Figure 75: Generate Block Design

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

## 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 the highest level in the design hierarchy. To integrate the IP integrator design into a higher-level design, simply instantiate the design in the top-level HDL file.

A higher-level instantiation of the block design can also be done by selecting the block design in the Vivado IDE **Sources** pane and selecting **Create HDL Wrapper** (shown below). This will generate a top-level HDL file for the IP integrator sub-system.

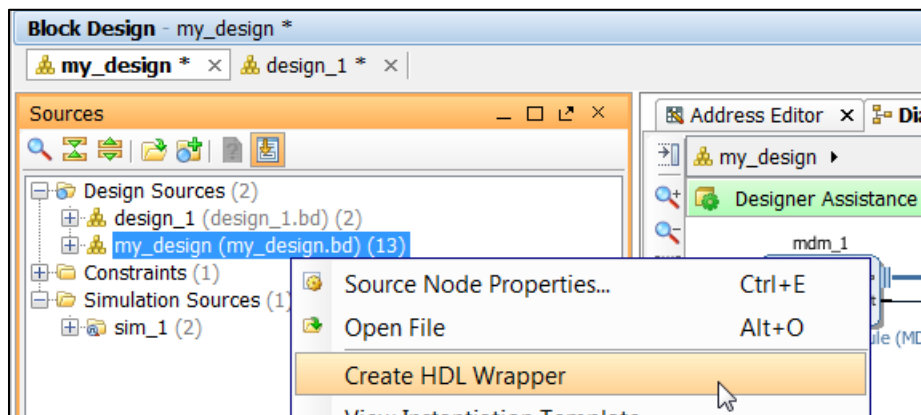
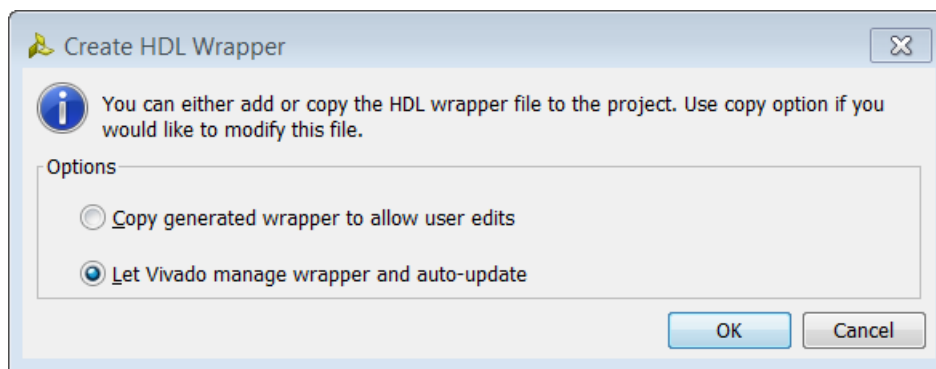


Figure 76: Creating an HDL Wrapper



Selecting Create HDL Wrapper offers two choices. The first choice is to make the wrapper file user editable. You may want to choose this option if you would like to manually edit the wrapper file. Often times a block design is a subset of an overall design hierarchy. This option can be used in this case. You can then instantiate other design components in the wrapper file. You need to make sure that this file is updated any time an I/O interface of the block design changes. The wrapper file created using this method is placed in the `<project_name>.srcs/sources_1/imports/hdl` directory.

The second choice offered is to allow the Vivado tools to create and manage the top-level wrapper file. If the block design is the only design component in the project or if edits to the wrapper file are not desired, then this option should be chosen. In this case the wrapper file is updated every time output products are generated. The wrapper file created using this method is located in the directory `<project_name>.srcs/sources_1/bd/<bd_name>/hdl`.



**Figure 77: Create HDL Wrapper Dialog Box**

At this point, you are ready to take the design through elaboration, synthesis, and implementation.

## I/O Buffer instantiation in IP integrator

When generating the wrapper, IP integrator looks for I/O Interfaces that are made external in the design. If the tool finds IO Interfaces that are made external, it then looks at the port maps of that interface. If three ports which match the pattern "`<name>_I`", "`<name>_O`", and "`<name>_T`" are found, then the tool instantiates an I/O Buffer and connects the signals appropriately. If any of the three ports are not found, then I/O Buffer is not inserted. Other conditions in which I/O Buffers are not inserted include the following:

1. If any of the `_I`, `_O` and `_T` ports are manually connected by the user, either by making them external or by connecting it to another IP in the design.
2. If the interface has a parameter called "BUFFER\_TYPE", and it is set to "NONE".

In some cases you may want to hand instantiate I/O buffers in the block design. In such a case, you need to use the Utility Differential I/O Buffer IP which is available in the IP catalog. This IP can be configured as different kinds of I/O buffers as shown below.

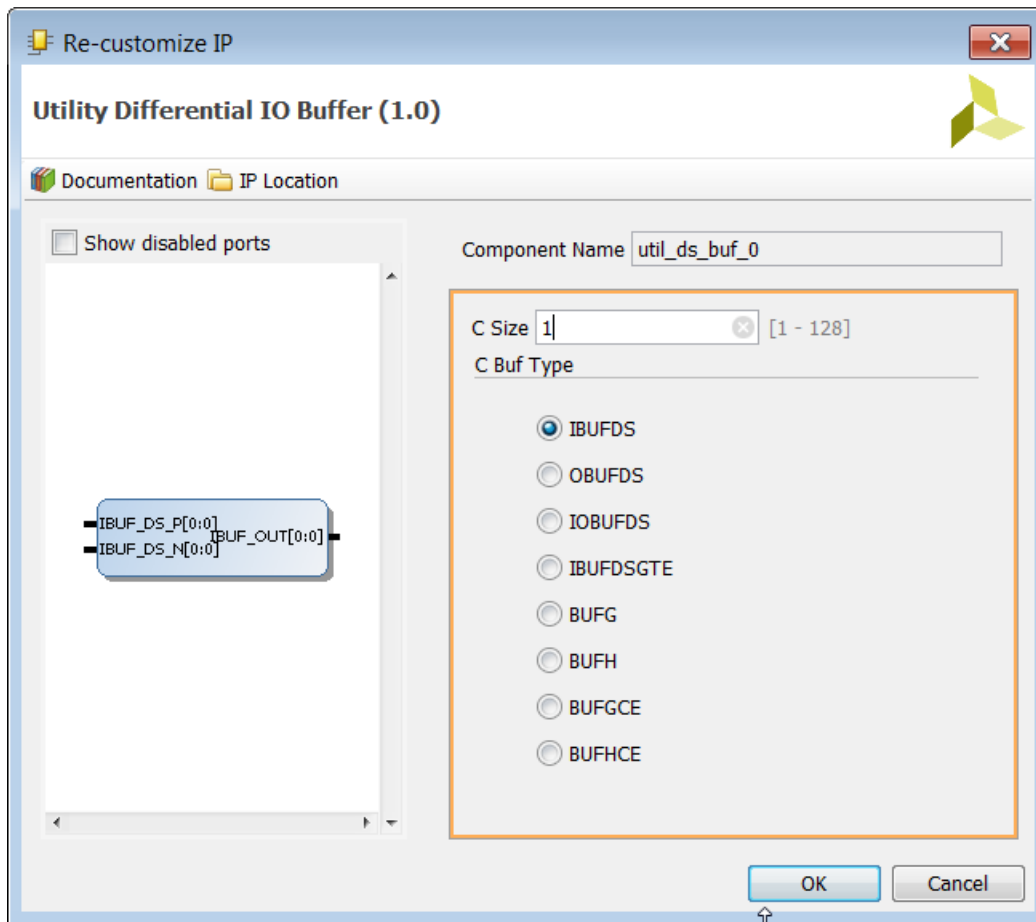
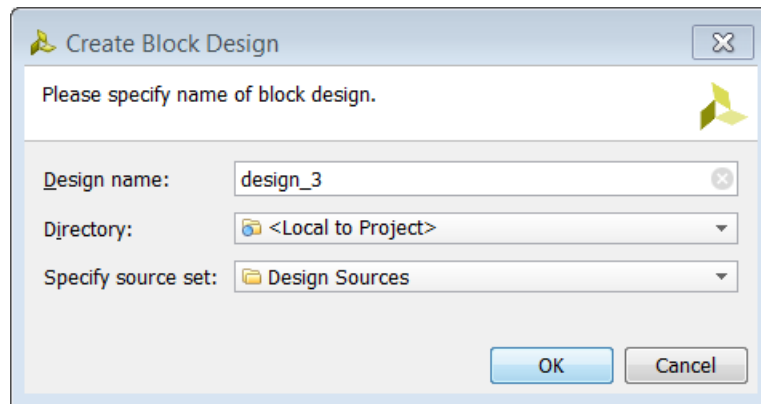


Figure 78: The Utility Differential I/O Buffer

## Creating a Block-Design Outside of the Project

A block design can be created within a project 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 a non-project mode or to use it in a team-based environment.

To create a block design outside the project, click on **Create Block Design** in the IP integrator drop-down list in the Flow Navigator. The Create Block Design dialog box opens. In the Create Block Design dialog box you can specify the name of the block design and also the Directory location. The default value for the Directory field is <Local to Project>. However, you can override this default value by clicking on Directory field and selecting **Choose Location**.



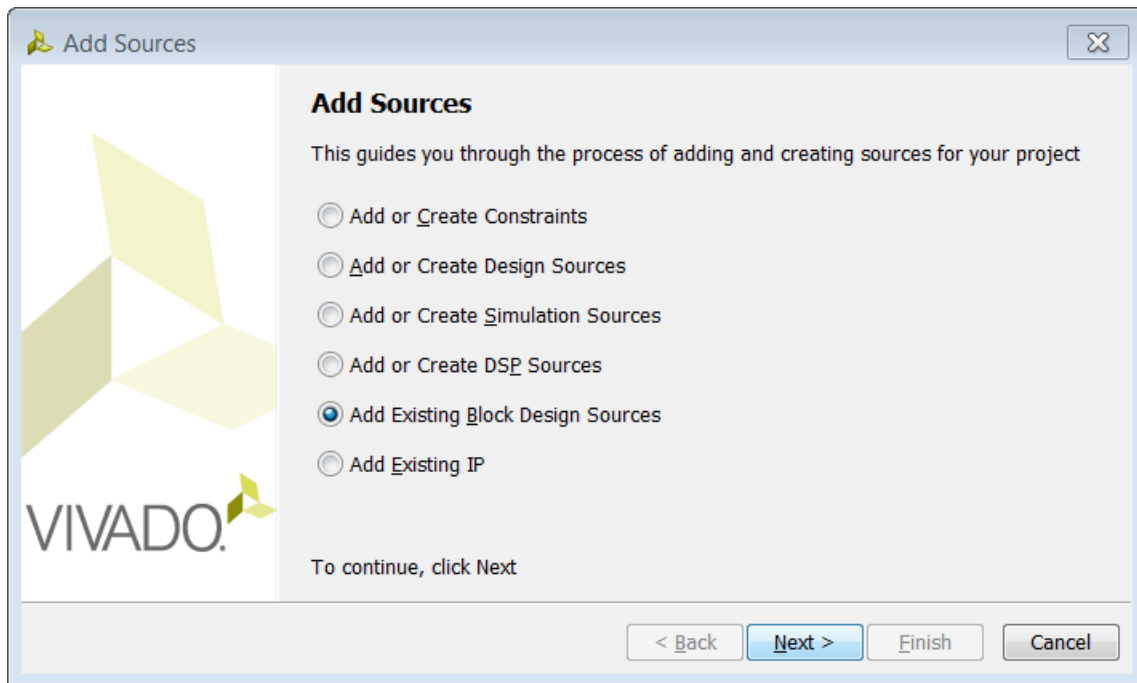
**Figure 79: Create Block Design Dialog Box**

Click **OK** after choosing the desired location to create the block design. You can continue to work on the block design just as you would in a normal project-based flow. The entire block design directory is created at the chosen location with its own sub-directory structure. You need to keep the entire directory structure of the bd in order to be able to re-open this block design from a different project or by a different user.

You can also create the block design as Design Sources or Simulation Sources by selecting the drop down menu from the **Specify source set** field.

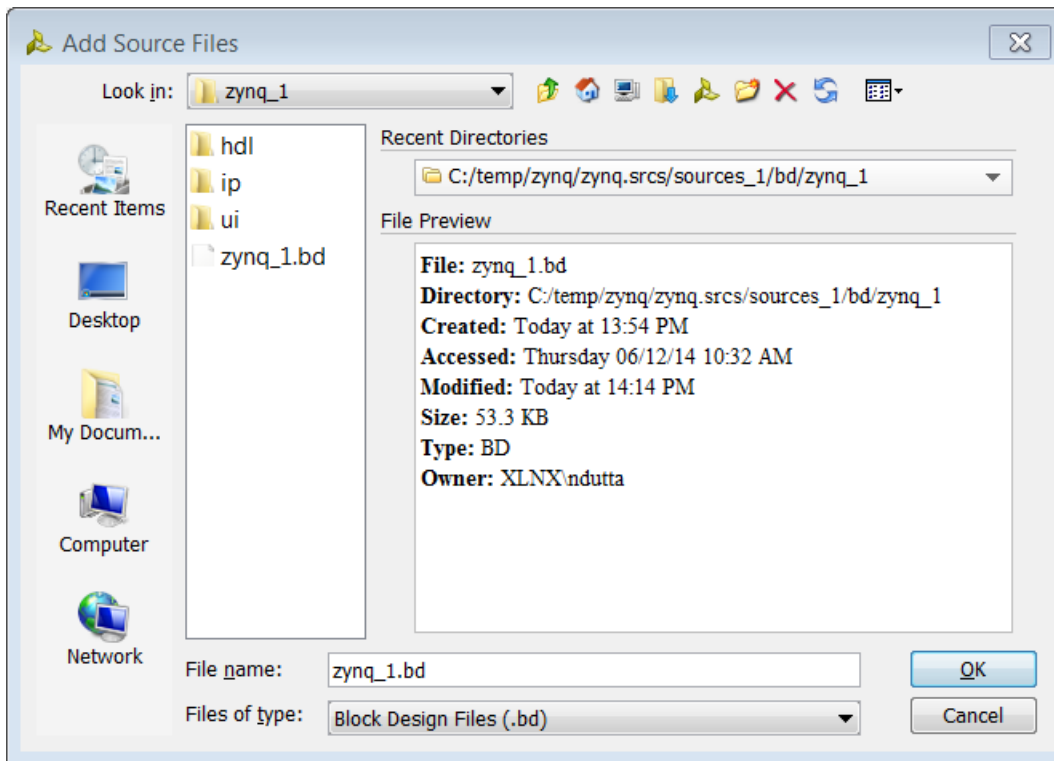
## Adding a Block-Design outside of a Project

A block design that has been created outside of a project or a remote location can be added to an existing Vivado project. To add a remote block design, you click on **Add Sources** under Project Manager in the Flow Navigator. Alternatively, you can also right click in the Sources window and select Add Sources. In the Add Sources dialog box select **Add Existing Block Design Sources** and click **Next**.



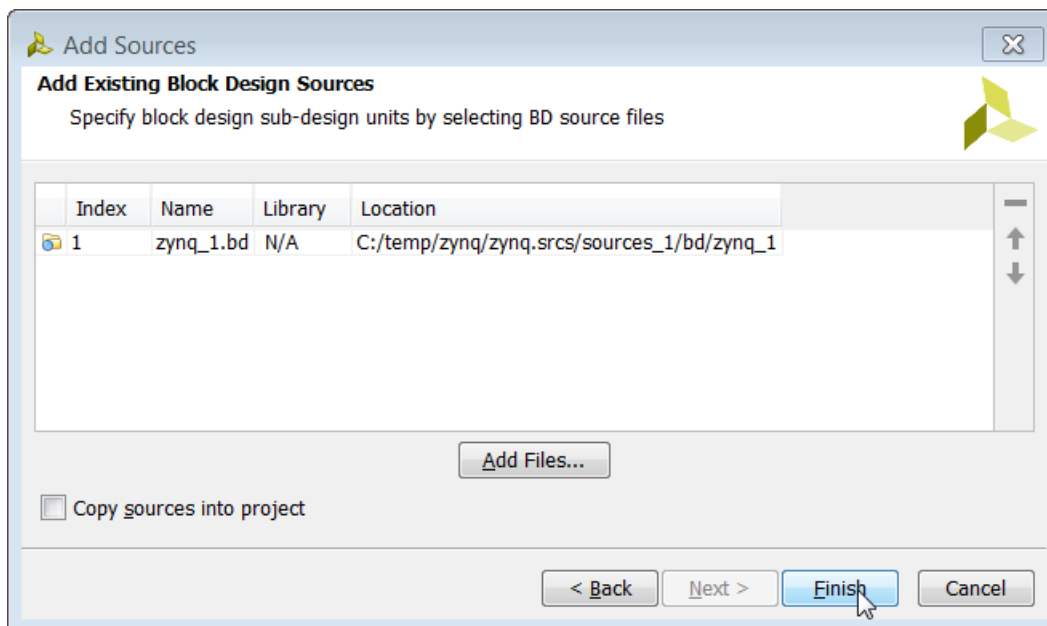
**Figure 80: Add Sources dialog box**

In the Add Existing Block Design Sources page, click on **Add Files**. In the Add Sources File window, navigate to the block design to be added.



**Figure 81: Add block design source**

Back in the Add Existing Block Design Sources page click on **Finish**.



**Figure 82: Finish adding block design sources**

Note that if the remote block design has been set as an out-of-context module, the design-check-point (DCP) file for this block design will also be added to the project. If you want to reference the DCP file for a block design, then you must add the block design as shown above.



**CAUTION!** While adding a block design from a remote location, ensure that no-one else is editing the block design at the same time. One way to overcome this issue would be to copy the remote block design locally into the project.

---

## Packaging a Block Design

Often times it happens that you create an IP integrator design, implement it, and test it on target hardware. If you are satisfied with the functionality, you may want to “package” it and convert it into an IP that can be reused in another design. When you package a design, it gets converted into an IP and is available for you in the IP catalog. You can instantiate that IP as part of a different design.

Vivado also provides you with the capability to create a custom interface that can be created and re-used as an interface definition on a custom IP. For more information on packaging a block design, refer to *Vivado Design Suite User Guide: Creating and Packaging Custom IP* ([UG1118](#)).

---

## Exporting the Hardware Definition to SDK

If you want to start software development before a bitstream is created, you can export the hardware definition to SDK after generating the design. This action exports the necessary XML files needed for SDK to understand the IP used in the design and also the memory mapping from the processor’s 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.

When the output products for the block design are generated, an archive that has all the pertinent information for exporting the hardware to SDK is created. This archive is called `<top_level_design_name>.hwdef` and can be found in the synthesis directory such as `<project_name>.runs/synth_1`.

This archive contains the following files for a Zynq processor-based design.

- ps7\_init.c
- ps7\_init.h
- ps\_init.html
- ps7\_init.tcl
- hwdef.xml
- <bd\_name>.hwh

For a MicroBlaze-based design this archive contains the following files:

- hwdef.xml

- `<bd_name>.hwh`

Once the design is implemented and the bitstream is generated, a new archive is created called `<top_level_design_name>.sysdef`.

For a Zynq processor-based design, the sysdef archive contains the following files:

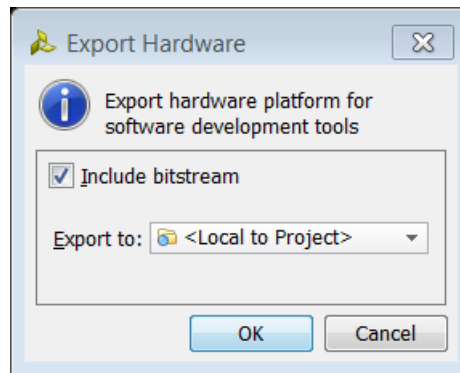
- `ps7_init.c`
- `ps7_init.h`
- `ps_init.html`
- `ps7_init.tcl`
- `sysdef.xml`
- `<bd_name>.hwh`
- `<top_level_design_name>.bit`
- `<top_level_design_name>.bmm`

For a MicroBlaze-based design, this archive contains the following files:

- <bd\_name>.hwh
- <top\_level\_design\_name>.bit
- <top\_level\_design\_name>.bmm
- sysdef.xml

Exporting the hardware simply copies the sysdef file into the location specified by the user.

1. Select **File > Export > Export Hardware** from the menu to open the Export Hardware dialog box.



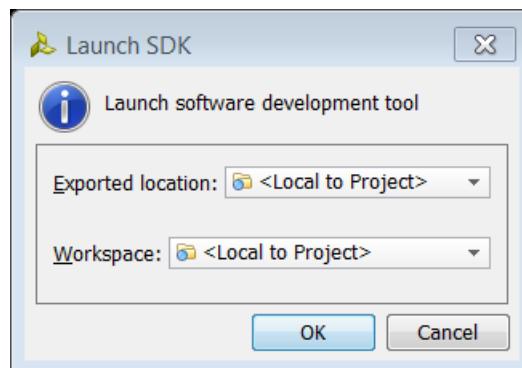
**Figure 83: Exporting the Hardware Definition for a Project**

There are a couple of options that are presented as can be seen from the figure above. Checking the Include bitstream checkbox will include the bitstream as a part of the exported data to SDK. The Export to field can be set as deemed appropriate by the user. In a typical project based flow, this should be left to its default value of **<Local to Project>**.

In a project-based flow, the hardware is exported at the following location:

`project_name/project_name.sdk`

2. To launch SDK after the hardware has been exported, click **File > Launch SDK**. The Launch SDK dialog box opens.



**Figure 84: Launch SDK Dialog Box**



In a typical project based flow, the default options should be left as defined **<Local to Project>**, for both the Exported location and the Workspace fields. If you choose a different location while exporting the hardware, then the Exported location field should be set to that particular location. The Workspace field can be set as needed as well.

Once SDK launches, a custom application project can be created in it using the hardware definitions exported. SDK creates the necessary drivers and board support package for the target hardware.

## Adding and Associating an ELF File to an Embedded Design

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

### Add an ELF file and Associate it With an Embedded Processor

1. The first way to accomplish this task is by selecting and right-clicking **Design Sources** in the Sources window and then selecting **Add Sources** as shown below. Design sources can also be added by clicking on **Add Sources** in the Flow Navigator under the Project Manager drop-down list.
2. The Add Sources dialog box opens. **Add or Create Design Sources** is selected by default. Selecting this option will add the ELF file as a design as well as a simulation source. If you are adding an ELF file for simulation purposes only, select **Add or Create Simulation Sources**. Click **Next**.
3. In the Add or Create Design Sources dialog box, click on **Add Files**.

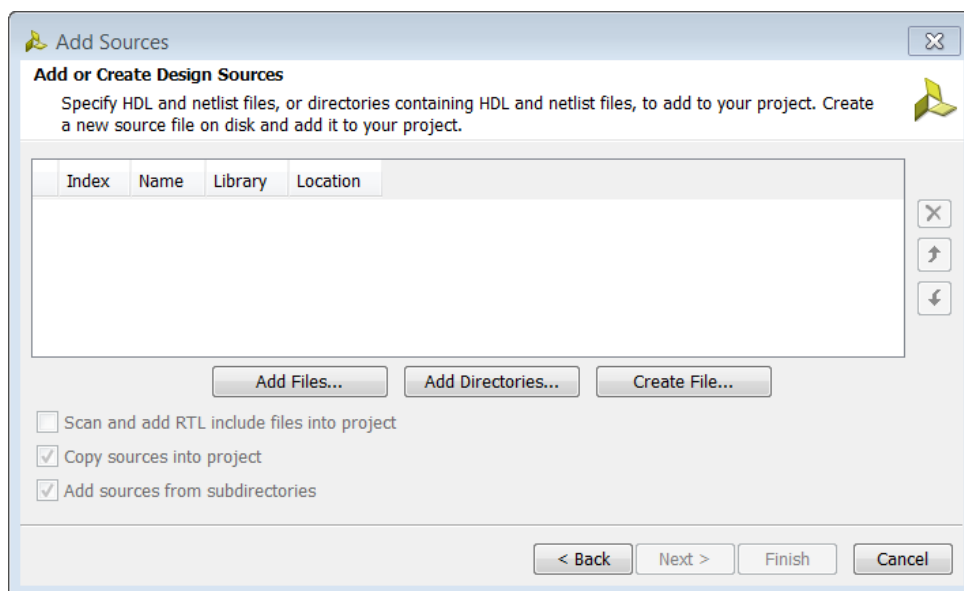
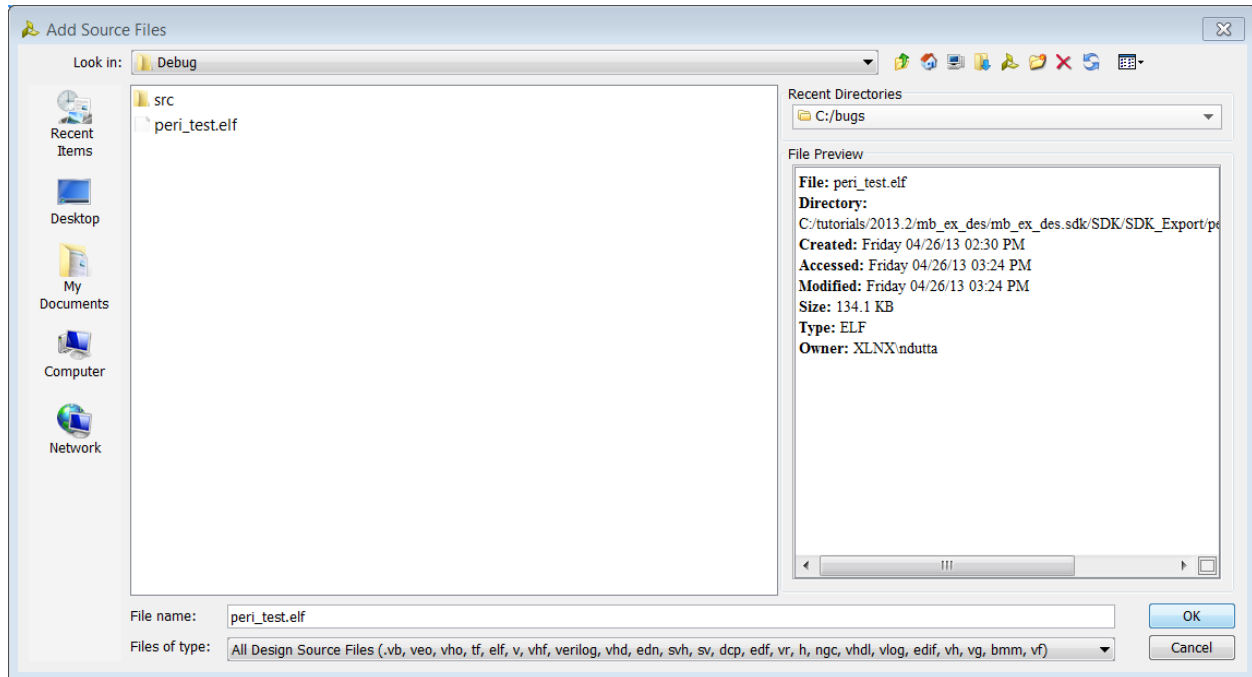


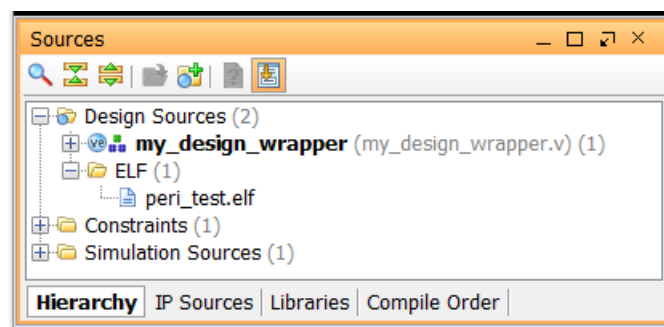
Figure 85: Add Sources Dialog Box

- The Add Source Files dialog box opens. You then navigate to the ELF file, select it and click **OK**.



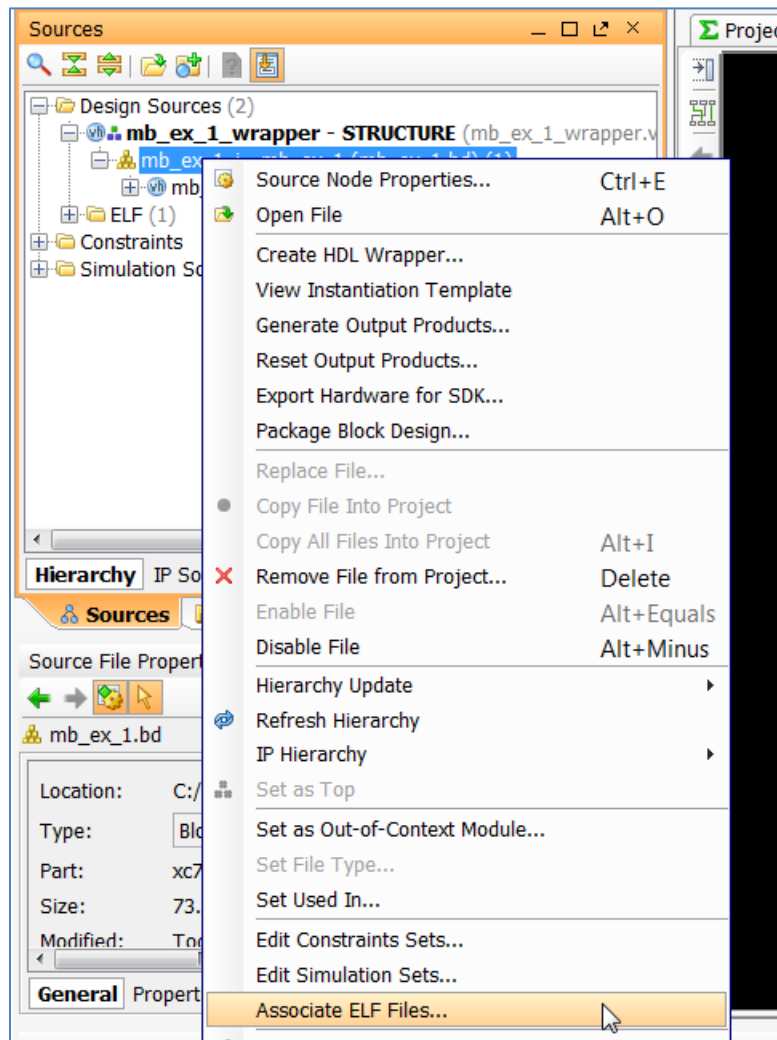
**Figure 86: Add Sources Files Dialog Box**

- In the Add or Create Sources dialog box, you can see the ELF file added to the project. Depending on your preference, you can either copy the ELF file into the project by checking **Copy sources into project** or leave that option unchecked if you want to work with the original ELF file. You then click **Finish**.
- Under the Sources window, you will see the ELF file added under the ELF folder.



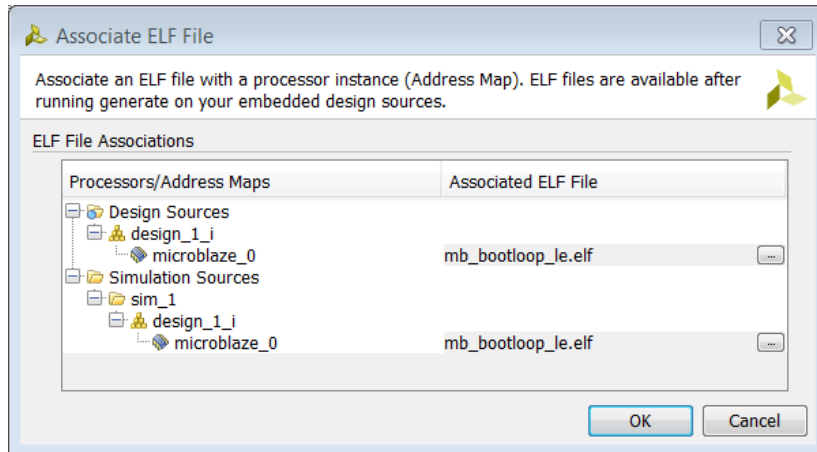
**Figure 87: Sources Window with ELF File Displayed**

- Next, you associate that ELF file with the microprocessor design in question. To do this, select and right-click on the block design in the Sources window and select **Associate ELF Files**.



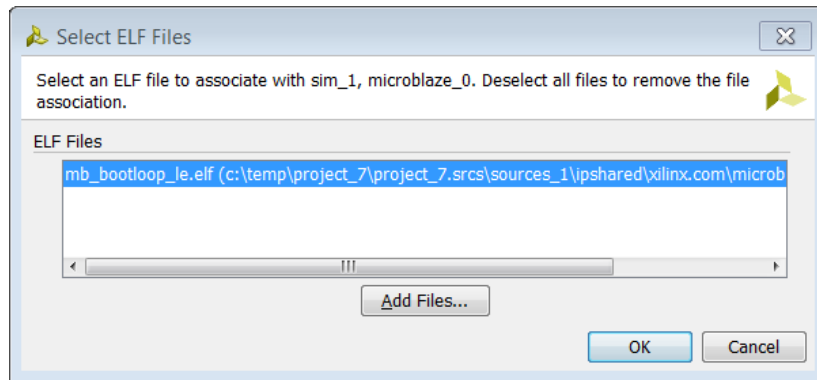
**Figure 88: Selecting Associate ELF Files**

8. You can associate an ELF for synthesis as well as simulation. Click on the appropriate browse icon (Under Design Sources or Simulation Sources) and browse to the newly added ELF file to the design.



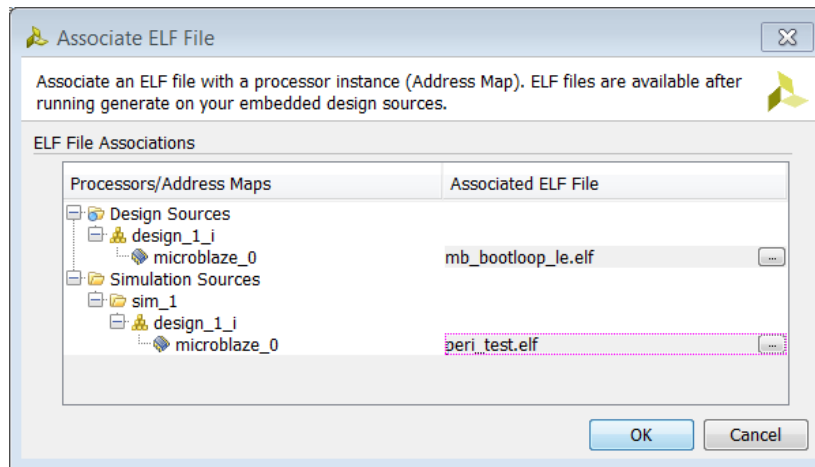
**Figure 89: Associating ELF Files with a Microprocessor**

The Select ELF file dialog box opens. Highlight the file, as shown below, and click **OK**.



**Figure 90: Highlight the ELF File to Associate**

Make sure that the ELF file is populated in the Associated ELF File Column and click **OK**.

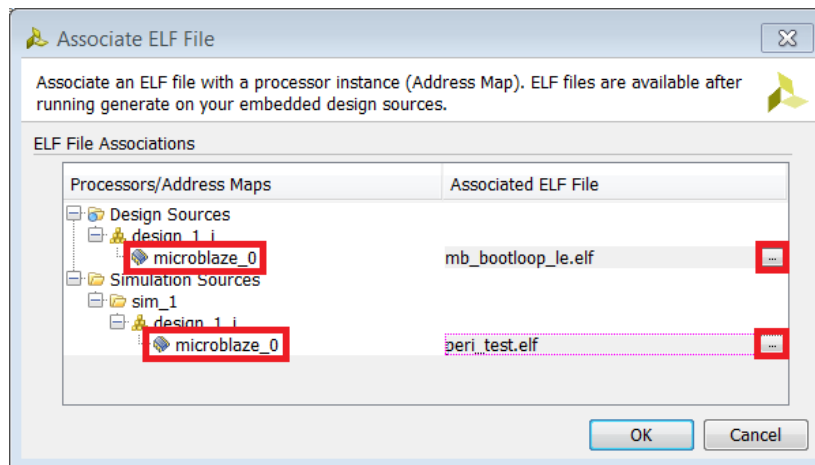


**Figure 91: Making Sure the ELF File is Populated**

## Adding and Associating an ELF File in a Single Step

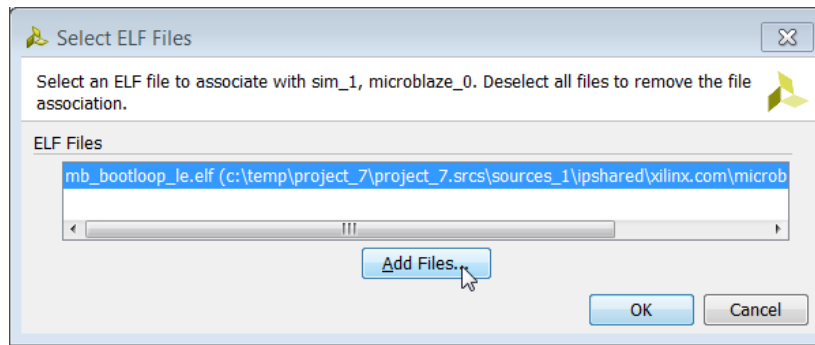
You can accomplish adding and associating an ELF file at once by following the following steps.

1. Right-click on the block design in Sources window and select Associate ELF files.
2. In the Associate ELF Files dialog box, click on the browse button on either the processor instance under Design Sources or Simulation Sources.



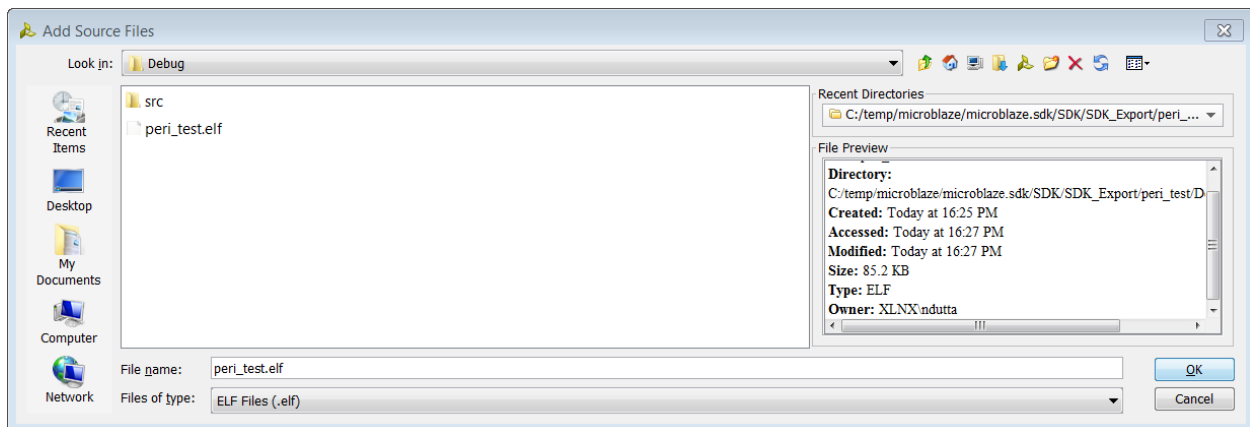
**Figure 92: Adding and Associating an ELF file using the Associate ELF Files Dialog Box**

3. When the Associate ELF File dialog box pops up, click on **Add Files**.



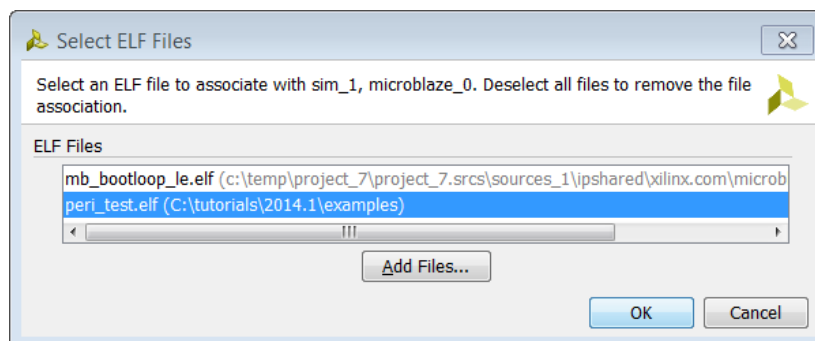
**Figure 93: Click on Add Files Button to Add an ELF File**

4. In the Add Source Files dialog box, navigate to the folder where the ELF file is located. Select the file and click **OK**.



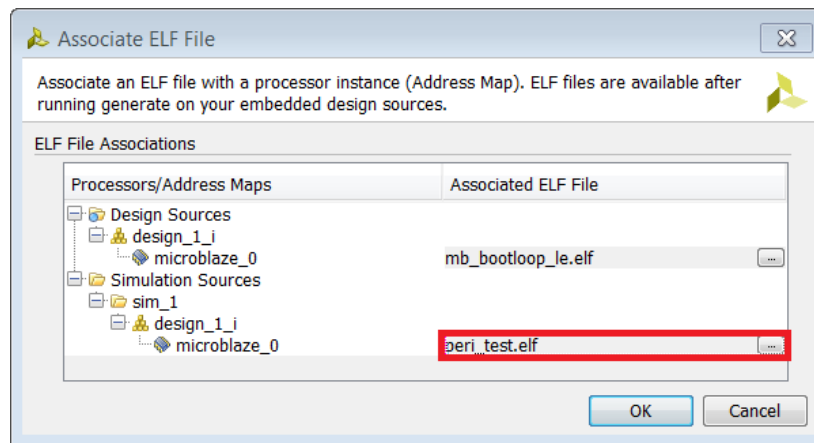
**Figure 94: Navigate to the Directory where the ELF File is Located**

5. Back in the Associate ELF file dialog box, select the newly added ELF file and click OK.



**Figure 95: Associate the Newly Added ELF File**

- In the Associate ELF Files dialog box, verify that the new ELF file is shown in the processor instance field and click **OK**.



**Figure 96: Verify that the Newly Added File is Associated to the Processor Instance**

With the ELF file added to the project, the Vivado Design Suite will 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.

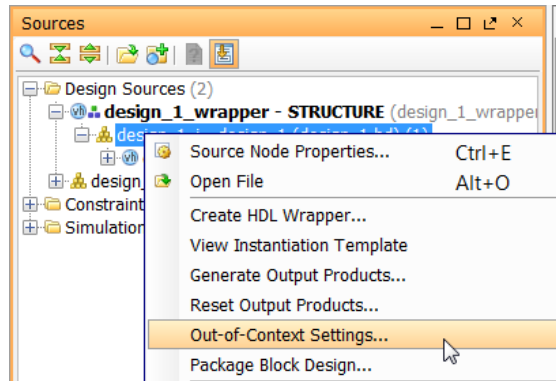
You can also do this manually using the UpdateMEM utility. Refer to this [link](#) in the *Vivado Design Suite User Guide: Embedded Processor Hardware Design (UG898)* for more information on UpdateMEM.

## Setting the Block Design as an Out-of-Context (OOC) Module

Hierarchical Design flows enable you to partition the design into smaller, more manageable modules to be processed independently. In the Vivado Design Suite, these flows are based on the ability to implement a partitioned module out-of-context (OOC) from the rest of the design. The most common use situation in the context of IP integrator is that you can set the block design as an out-of-context module which can be synthesized and a design checkpoint (DCP) file created. This block design, if used as a part of the larger Vivado design, does not have to be re-synthesized every time other parts of the design (outside of IP integrator) are modified. This provides in considerable run-time improvements.

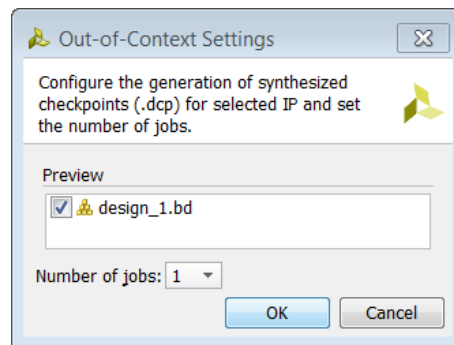
**CAUTION!** The Out-of-Context Mode can only be enabled for the entire block design. You cannot select individual IP in a block design and set them as Out-of-Context modules.

To set a block design as an out-of-context module, you highlight the block design, right-click and select **Out-of-Context Settings**.



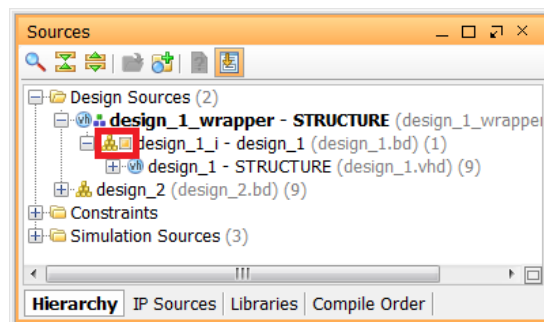
**Figure 97: Setting a Block Design as an Out-of-Context Module**

The **Out-of-Context Settings** dialog box opens informing you that a design checkpoint (.dcp) file will be created for the block design. In the Preview pane check the checkbox against the block design.



**Figure 98: Set as Out-of-Context Dialog Box**

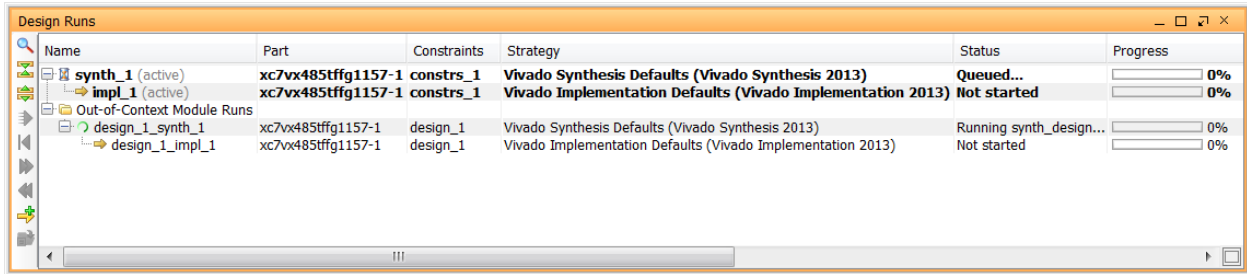
You can see if the OOC feature has been enabled on the block design by verifying the square box against the block design in the Sources window as shown by the highlighted box in the following figure.



**Figure 99: Verify that the Block Design has been Set as an OOC Module**



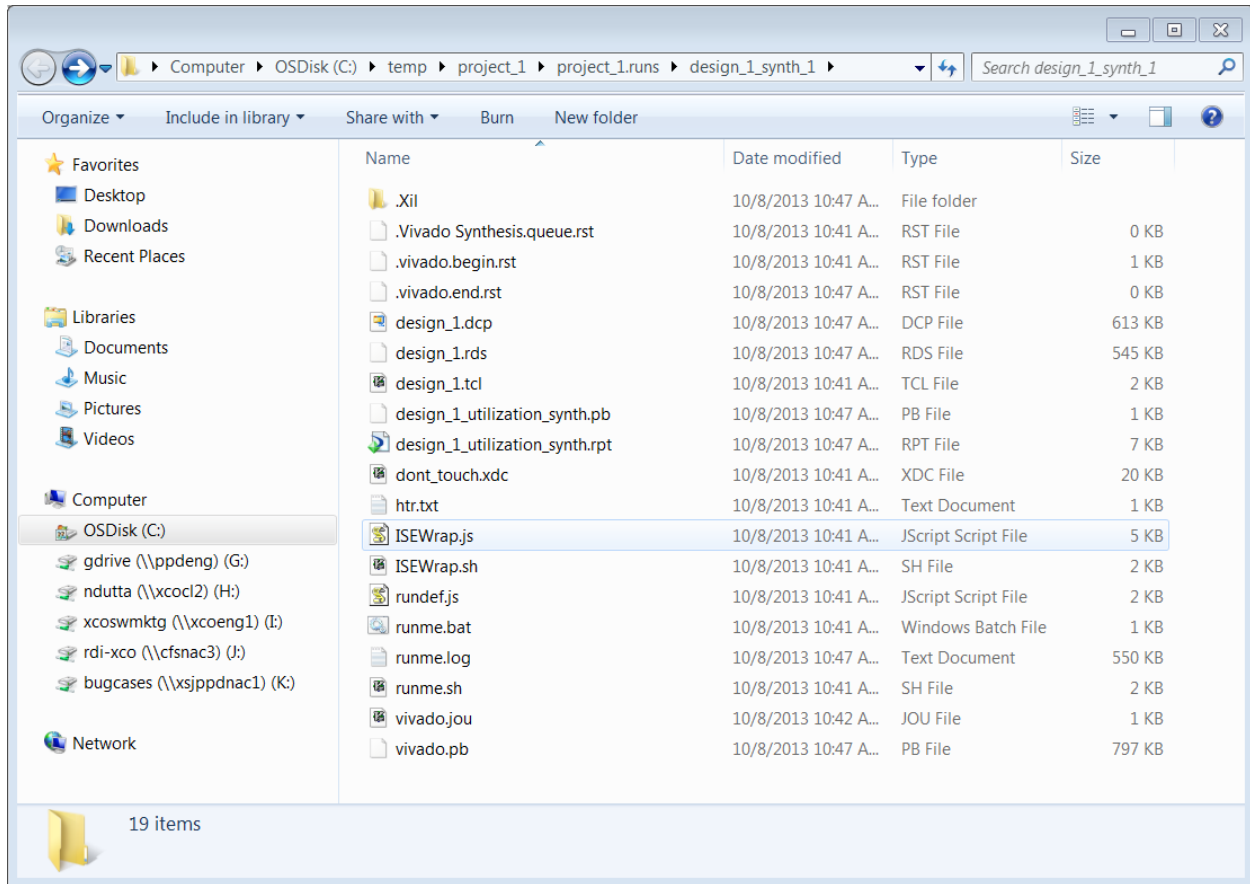
You can then synthesize the block design by selecting **Run Synthesis** from the Flow Navigator. As synthesis launches, you can see the Out-of-Context Module runs in the Design Runs window.



**Figure 100: Design Runs Window View for Out-of-Context Flow**

When the synthesis run completes, a design checkpoint file (DCP) is created which is an archive consisting of the synthesized netlist and all the constraints necessary for the block design. This DCP file can be found in the synthesis run directory which can be opened by selecting the `design_1_synth_1` run, right-clicking and selecting **Open Run Directory**.

The run directory opens in Windows Explorer.

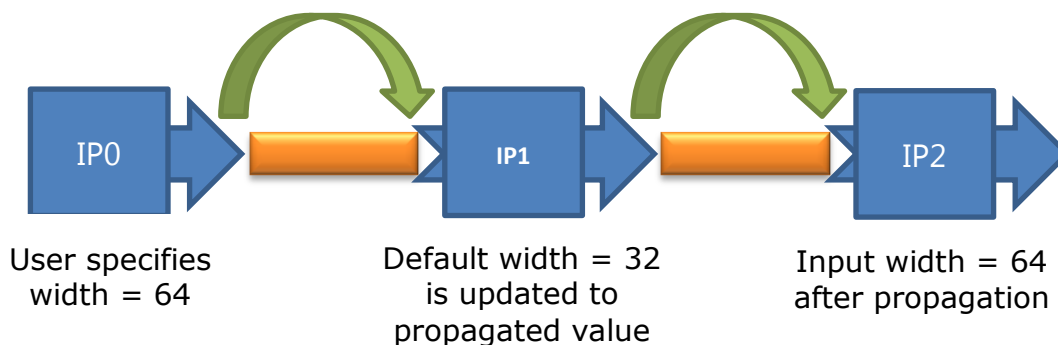


**Figure 101: The Run Directory for the OOC Module Containing the DCP File**

If the block design is added as a synthesized netlist in other designs, this DCP file can be added to the project where the block design is instantiated. The recommended way to do this is to add the block design to the project. Adding the block design brings in the DCP file as well.

### Overview

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. In this case, IP2 has a default data bus width of 32 bits. When the parameter propagation rules are run, the user is 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 IP cannot be updated to match properties based on its connection, then an error will be shown, alerting users of potential issues in the design. This is a simple example, but 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.



**Figure 102: Parameter Propagation Concept**

---

## 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 IO 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, AXI4Lite and AXI-Stream. The AXIMM interface includes all three subsets (AXI4, AXI3, and AXI4Lite). Other interfaces include BRAM.

### IO Bus Interfaces

Some Bus Interfaces that group a set of signals going to IO ports are called I/O interfaces. Examples include UART, I2C, SPI, Ethernet, PCIe, DDR etc.

### Special Signals

There are five standard signals identified that are used across a wide variety of IP. These interfaces are:

- Clock
- Reset
- Interrupt
- Clock Enable
- Data (for traditional arithmetic IP which do not have any AXI interface e.g. adder/subtractor, multiplier)

These special signals are described in the following sections:

## Clock

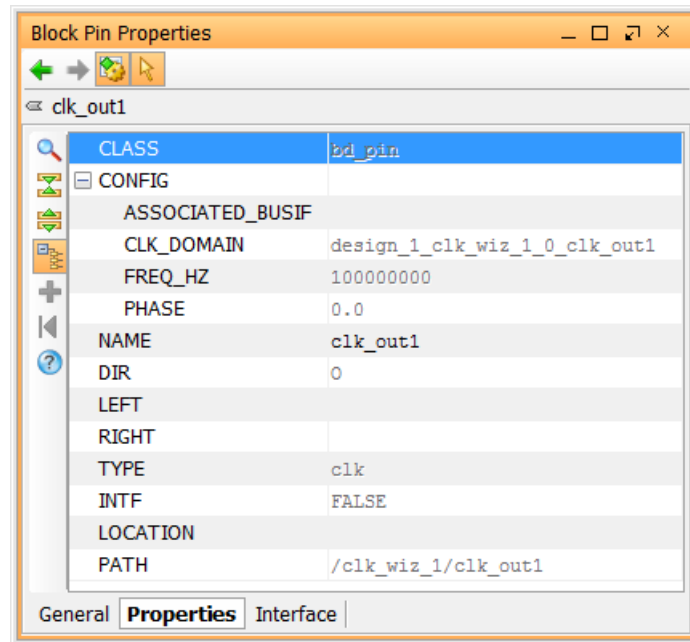
The clock interface can have the following parameters associated with them. These parameters are used in the design generation process and are necessary when the IP is used with other IP in the design.

- **ASSOCIATED\_BUSIF:** The list contains names of bus interfaces, which run at this clock frequency. This parameter takes a colon ":" separated list of strings as its value. If all interface signals at the boundary do not run at this clock rate, then this field is left blank.
- **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, then 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, then 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 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.



Figure 103: Analyzing the Clock Properties in IP Integrator



**Figure 104: Clock Properties**

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

```
report_property [get_bd_intf_ports sys_diff_clock]
```

```
report_property [get_bd_intf_ports /sys_diff_clock]
Property      Type    Read-only  Visible  Value
CLASS         string true      true     bd_intf_port
CONFIG.FREQ_HZ string false true     200000000
LOCATION       string false true     140 390
MODE         string true   true     Slave
NAME         string false true     sys_diff_clock
PATH         string true   true     /sys_diff_clock
VLNV        string true   true     xilinx.com:interface:diff_clock_rtl:1.0
```

**Figure 105: Reporting Clock Properties using Tcl Command**

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

## Reset

This container bus interface should also include the following parameters with it:

- POLARITY: Valid values for this parameter are ACTIVE\_HIGH or ACTIVE\_LOW. The default is ACTIVE\_LOW.

To see the properties on the clock net, select the reset port or pin and analyze the properties on the object.

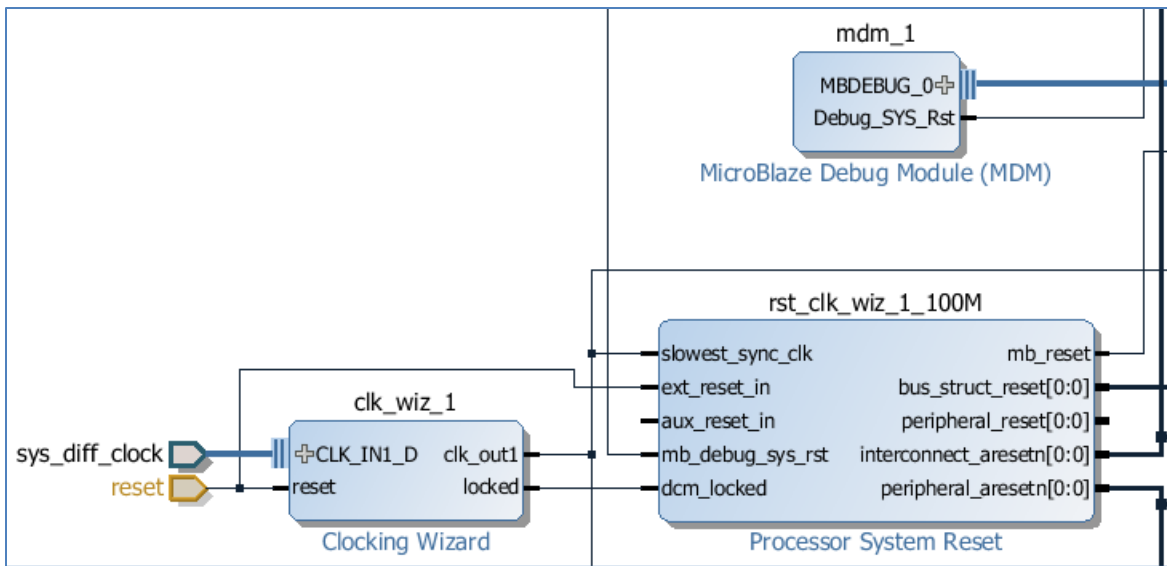


Figure 106: Reset Signal

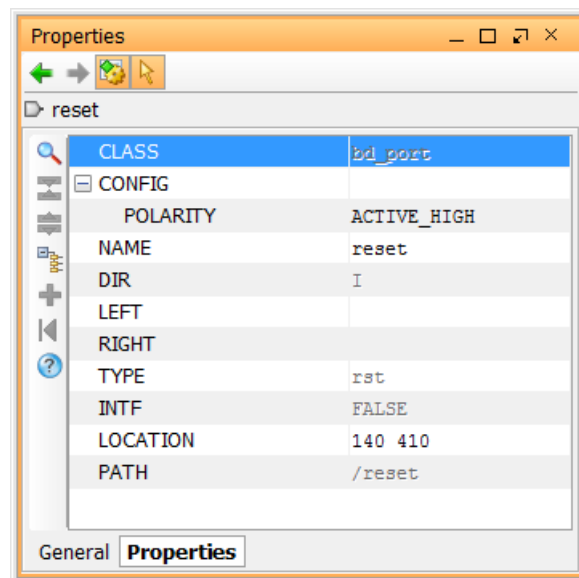


Figure 107: Reset Properties

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

```
report_property [get_bd_ports reset]
```

```
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     I
LOCATION        string   false      true     140 410
NAME          string   false      true     reset
PATH          string   true       true     /reset
RIGHT         string   false      true     I
TYPE          string   true       true     rst
```

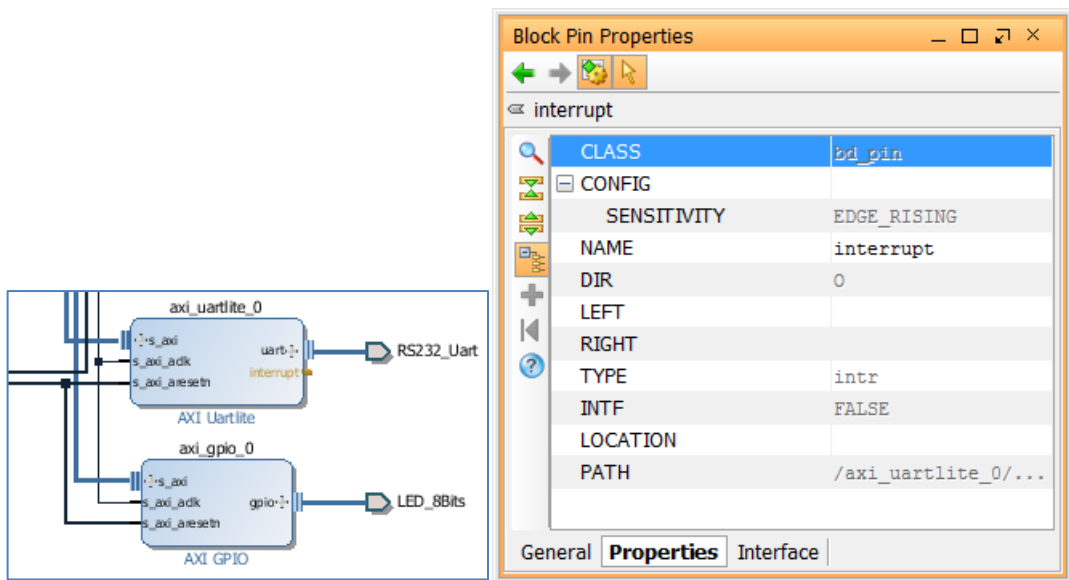
**Figure 108: Reporting reset Properties using Tcl Command**

### Interrupt

This bus interface includes the following parameters:

- 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 below and look at the properties window.



**Figure 109: Interrupt Properties**



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

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

```
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     0
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 110: 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 on **Validate Design** in the Vivado® IDE toolbar.
- Click on **Validate Design** in the Design Canvas toolbar.
- Click on **Tools > Validate Design** from the Vivado Menu.
- Use the Tcl command: `validate_bd_design`

The propagation Tcl provides a mechanism to synchronize an IP instance's configuration with that of other instances connected to it. 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.

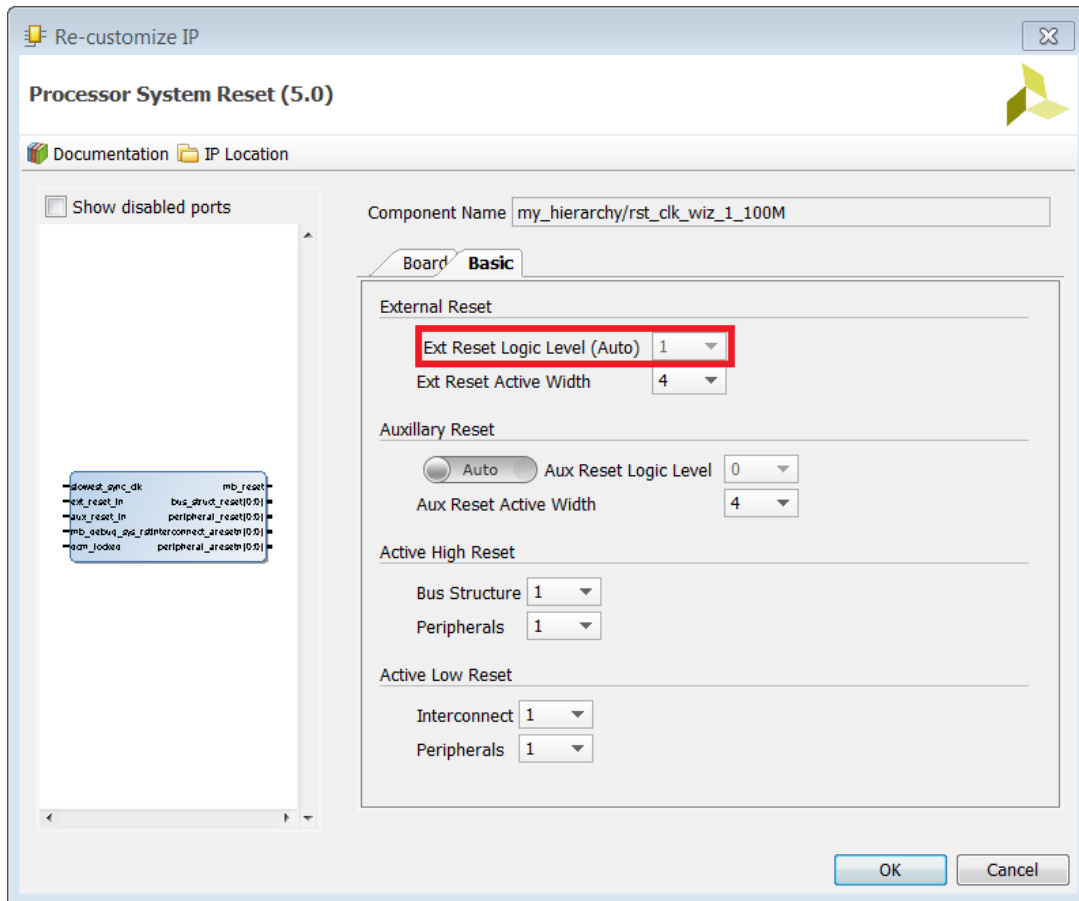
## Parameters in the Customization GUI

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 grayed-out parameter is an indication that you should not 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 may not be optimal. In those circumstances, you may override these propagated values.

There are four different cases that you will encounter related to parameter propagation.

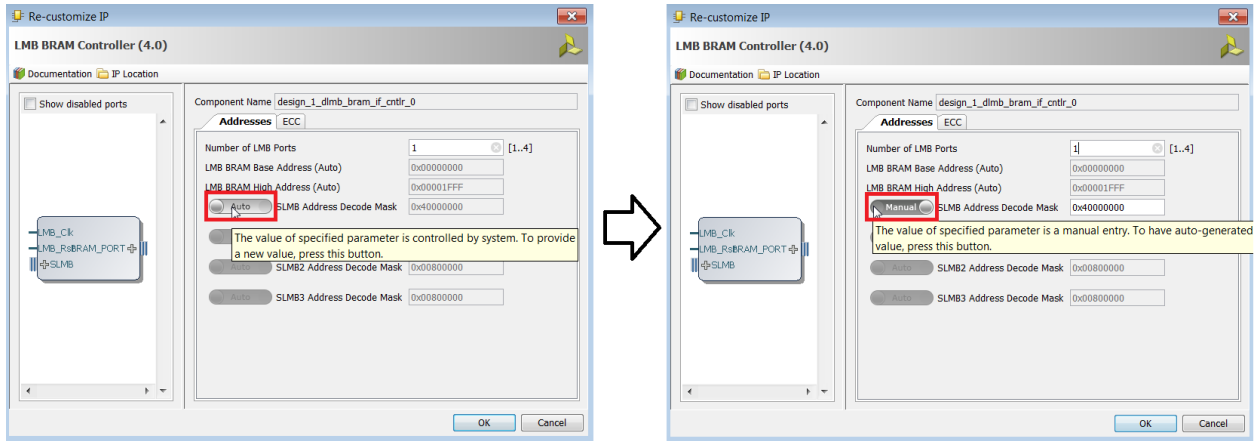
- Auto-computed parameters – these parameters are auto-computed by IP integrator and you cannot override them. For example, the **Ext Reset Logic Level** parameter in the following figure is greyed out and **Auto** is placed next to the parameter denoting that you cannot change it.



**Figure 111: Auto-Computed Parameter**

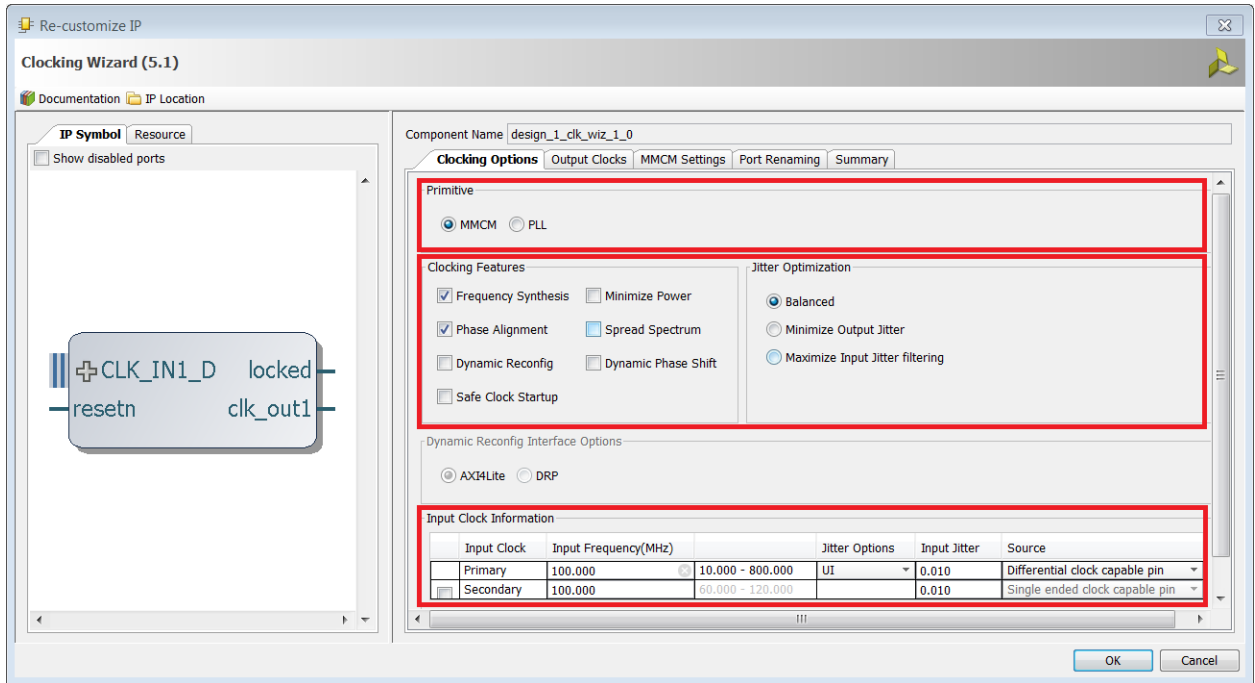
- Override-able parameters –Auto-computed parameters that you can override. For example, you can change the SLMB Address Decode Mask for the LMB BRAM Controller. When you hover the mouse on

top of the slider button, it will tell you that the parameter is controlled by the system; however, you can change it by toggling the button to User from Auto.



**Figure 112: Parameter to Override**

- User configurable parameters – These parameters are user configurable only and are to be set by the user.

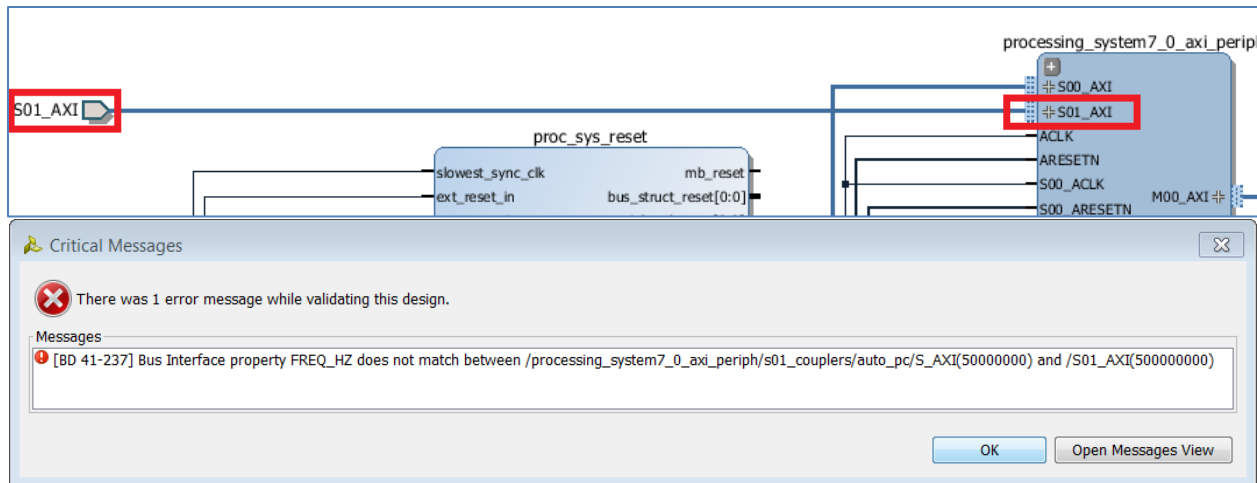


**Figure 113: User-Configurable Parameter**

- Constants – These are parameters that cannot be set by anyone.

## Example of a Parameter Mismatch

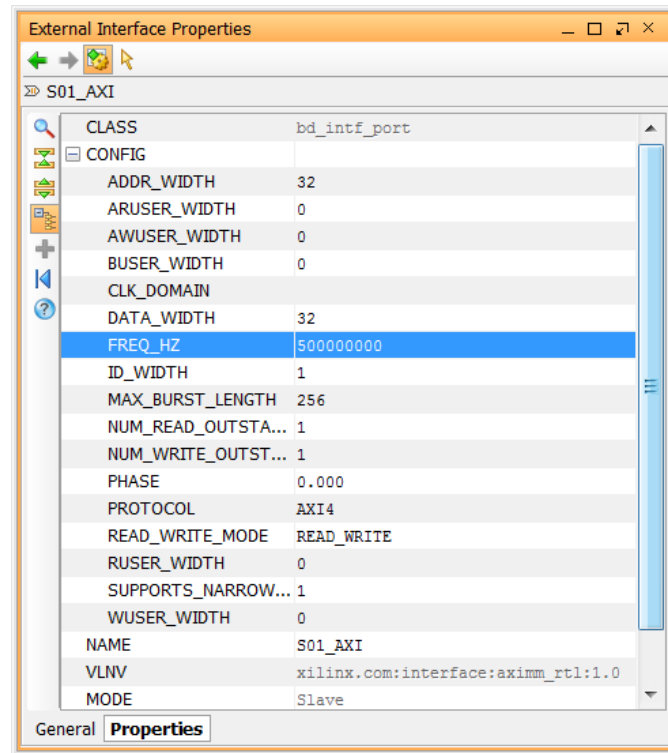
The following is an example of a parameter mismatch on the `FREQ_HZ` property of a clock pin. In this example, the frequency does not match between the `S01_AXI` port and the `S_AXI` interface of the AXI Interconnect. This error is revealed when the design is validated.



**Figure 114: `FREQ_HZ` property mismatch between a port and an interface pin**

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

This type of error can be easily fixed by changing the frequency in the property, or by double-clicking on the `S01_AXI` port and correcting the frequency in the Frequency field of the customization dialog box.



**Figure 115: Change the Frequency of the Port in the Properties Window**

Once the frequency has been changed, you can validate the design again to make sure that there are no errors.

### Overview

In-system debugging allows you to debug your design in real-time on your target hardware. This is an essential step in design completion. Invariably, you will come across a situation which is extremely hard to replicate in a simulator. Therefore, there is a need to debug the problem right there 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.

IP integrator provides ways to instrument your design for debugging which is explained in the following sections. There are two flows explained: the HDL instantiation flow and the netlist insertion flow. Choosing the flow depends on your preference and types of nets/signals that you are interested in debugging. As an example, if you are interested in performing hardware-software co-verification using the cross-trigger feature of MicroBlaze or Zynq processor, then you can use the HDL instantiation flow. If you are interested in analyzing I/O ports and internal nets, then netlist insertion will be the way to go. In most of the cases, you will be using a combination of both flows to debug your design.

### Using the HDL Instantiation Flow in IP Integrator

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

1. Right-click on the block design canvas and select Add IP.

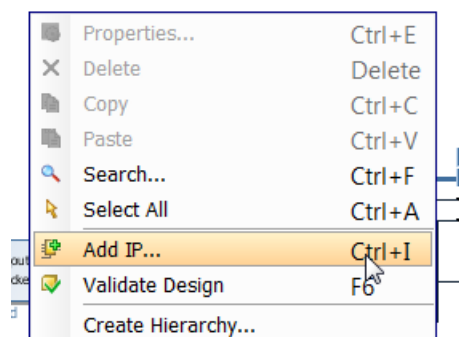
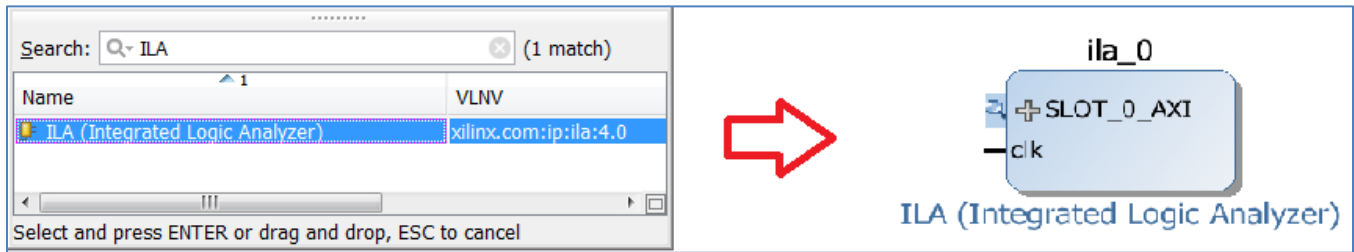


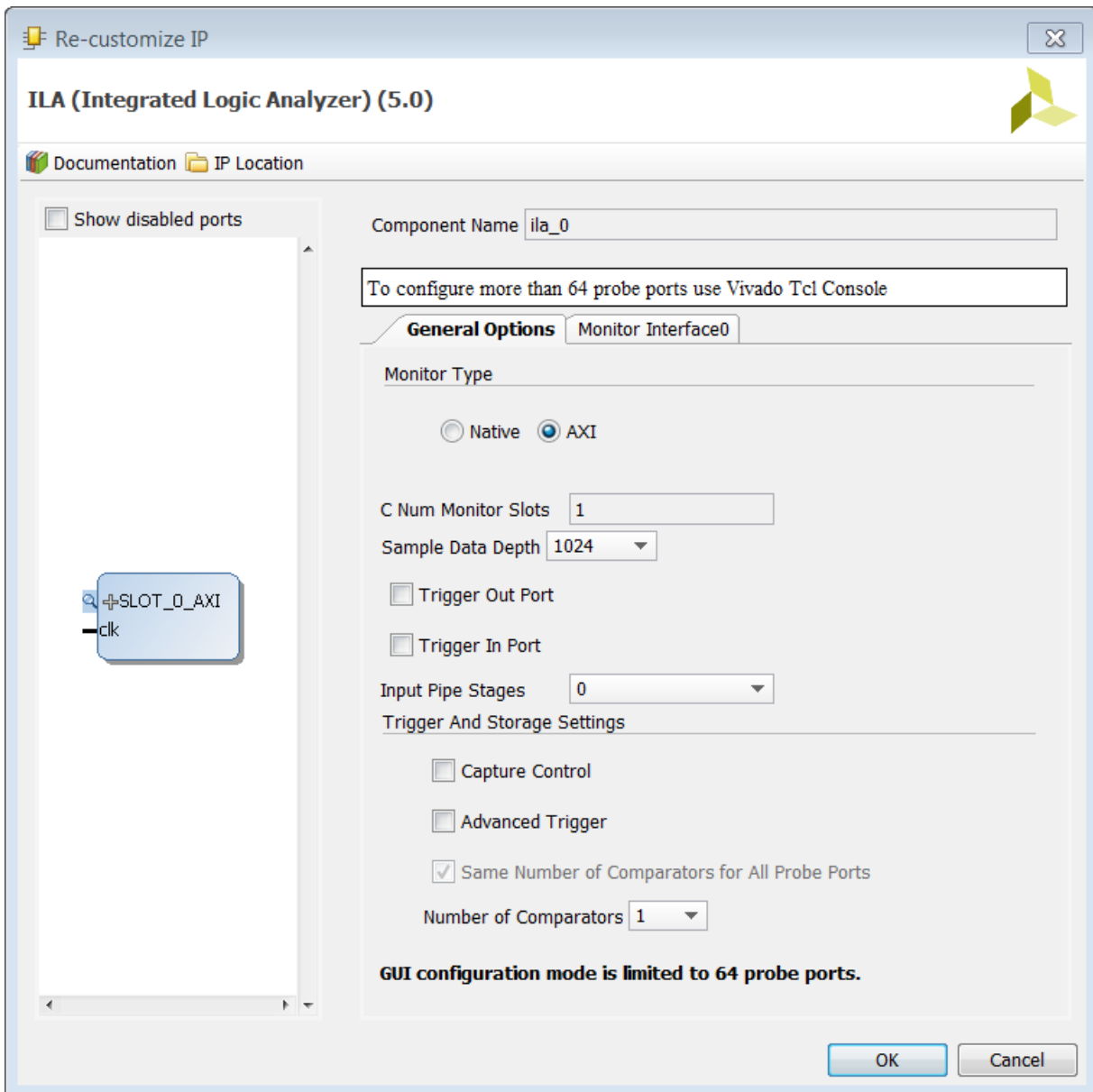
Figure 116: Add IP from the context menu

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



**Figure 117: Instantiate ILA Core**

3. Double click on the ILA core to reconfigure it. The Re-Customize IP dialog box opens.

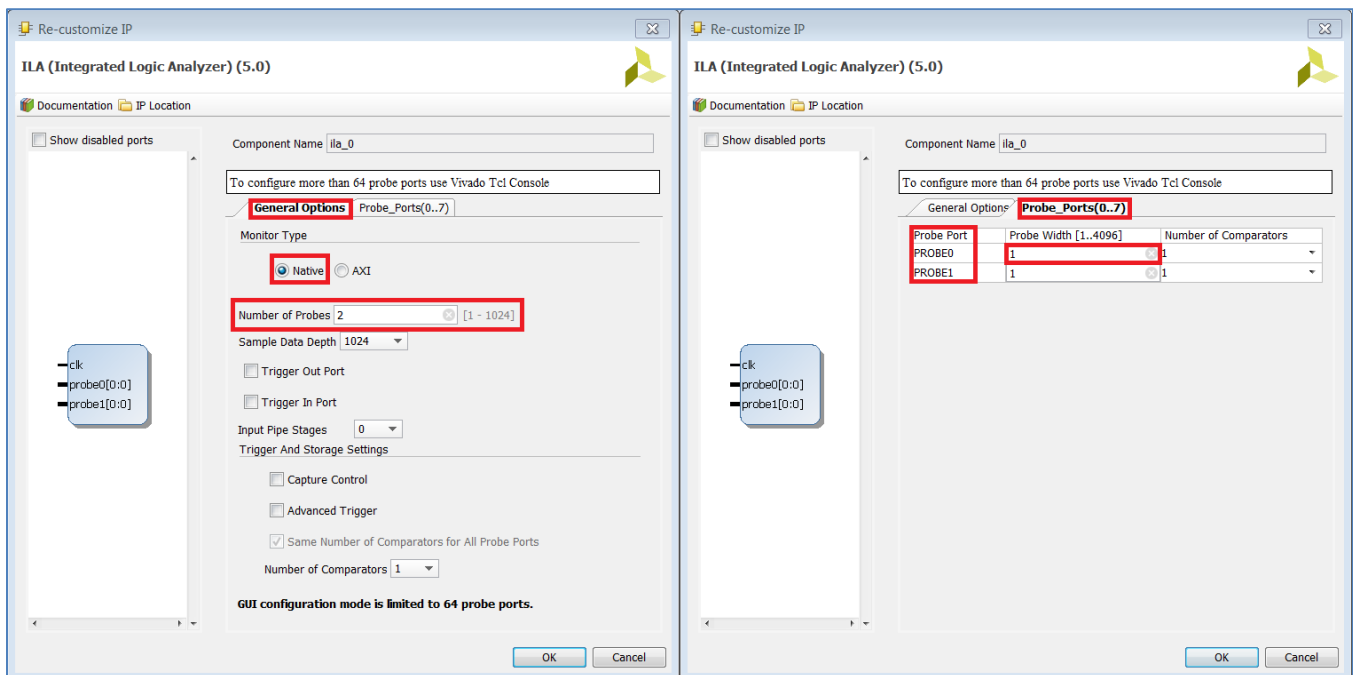


**Figure 118: Re-customize IP Dialog Box for the ILA Core**

The default option under the General Options tab shows **AXI** as the Monitor Type. Keep this selection in case you are monitoring an entire AXI interface. Change the Monitor Type to **Native** if you are monitoring non-AXI interface signals. You can change the Sample Data Depth and other fields as desired. For more information, please refer to the following document: *Vivado Design Suite User Guide: Programming and Debugging (UG908)*

**CAUTION!** You can only monitor one AXI interface using an ILA. Do not change the value of the C Num Monitor Slots. If more than one AXI interface is desired to be debugged, then instantiate more ILA cores as needed.

In case the Monitor Type is set to Native, you can set the Number of Probes to the desired value. This value should be set to the number of signals that need to be monitored.



**Figure 119: Choosing the Native Monitor Type**

In the figure above, 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. So assuming that you want to monitor a 32-bit bus, set the Probe Width for Probe 0 to 32. Once the ILA is configured, the changes are reflected on the IP integrator canvas.



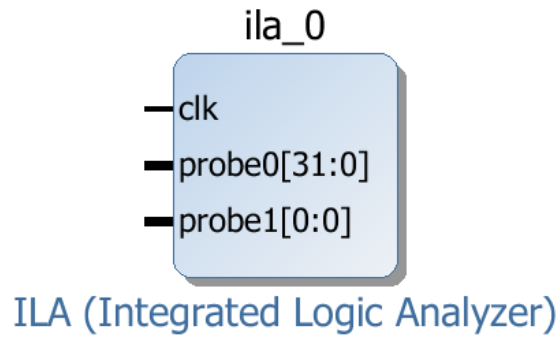


Figure 120: ILA Core after Making Changes in the Re-customize IP Dialog Box

- Once the ILA is configured as desired, connection can be made to the pins of the ILA on the IP integrator canvas.

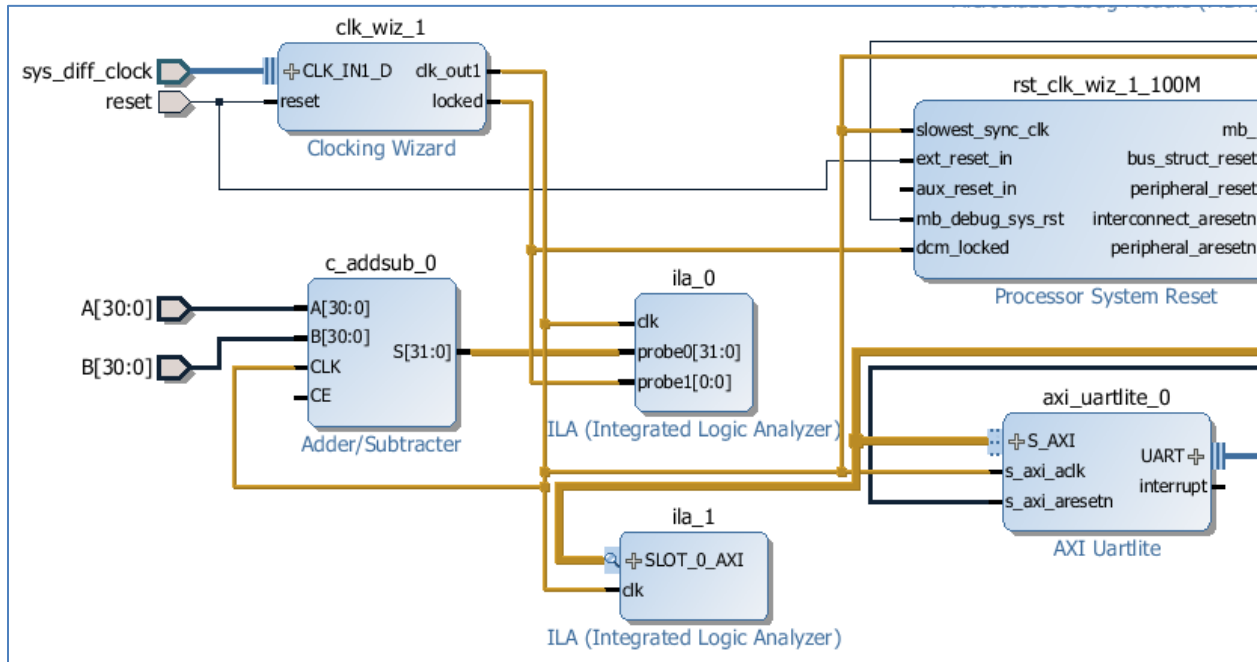


Figure 121: Instantiating ILAs to Monitor AXI and Non-AXI signals.

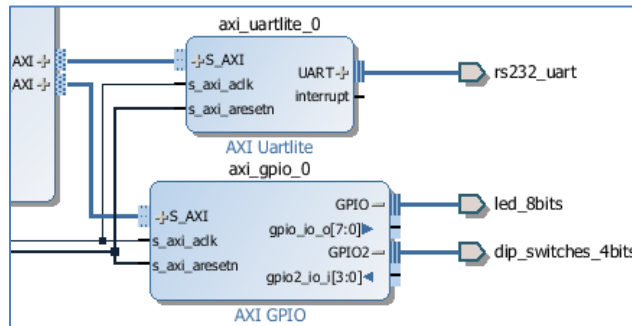
**CAUTION!** If a pin connected to an I/O port needs to be debugged, then Mark Debug should be used to mark the nets for debug. This is explained in the following section

- Follow on to synthesize, implement and generate bitstream.

## Connecting I/O Ports to an ILA or VIO Debug Core

Often times the I/O ports of a block design need to be probed for debugging. If the I/O ports are a part of the interface ports then you must take care when connecting these interface port or pins to the ILA or VIO core.

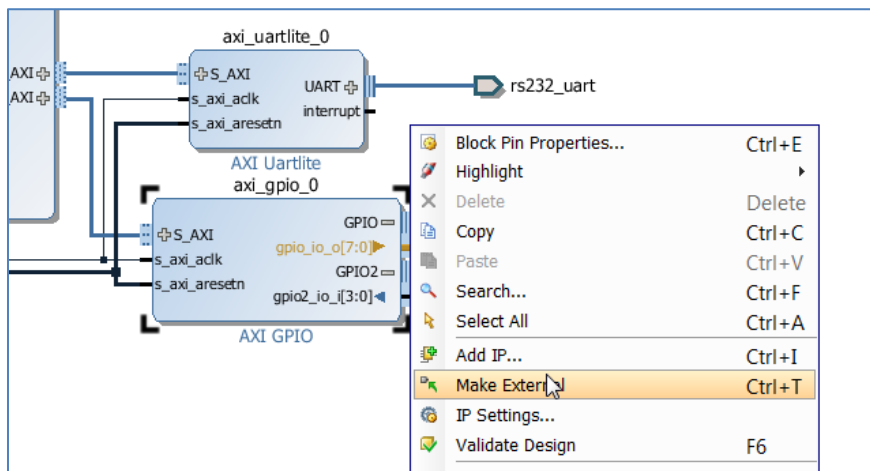
As an example, we will take the following MicroBlaze design for the KC705 board. This design has a GPIO which is configured for using both the 8-bit LED interface and the 4-bit dip switches on the KC705 board.



**Figure 122: Monitoring the I/O interfaces of a block design**

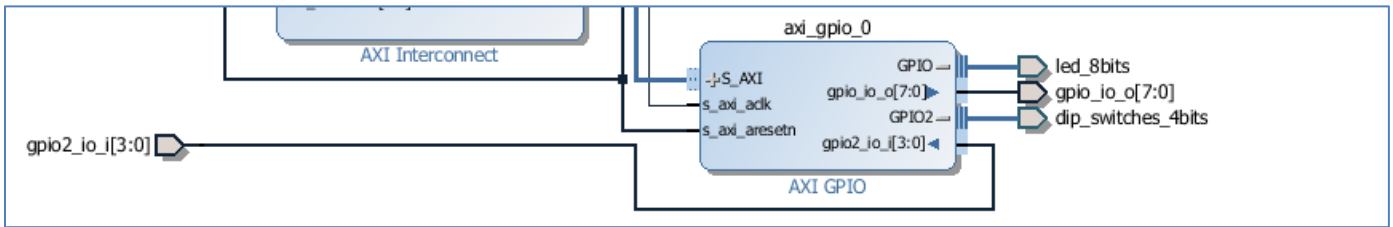
To monitor these I/O interfaces, you need to expand the GPIO interface pins so that you can see the pins that make up the interface pin. As you can see in the figure above, the GPIO interface consists of an 8-bit output pin called `gpio_io_o[7:0]` and the GPIO2 interface consists of a 4-bit input pin called `gpio2_io_i[3:0]`.

To monitor these pins 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. This can be done by right-clicking on the pin, and selecting **Make External** from the menu.



**Figure 123: Connect the I/O pin to an I/O port in the design**

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.



**Figure 124: External ports connected to pins within an interface**

**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 125**, leave the existing top-level interface pin connected externally to the appropriate interface. You will see a warning as shown below:

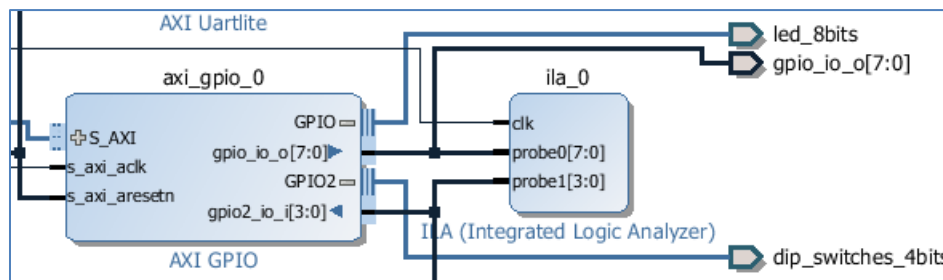


WARNING: [BD 41-1306] The connection to interface pin /axi\_gpio\_0/gpio2\_io\_i is being overridden by the user. This pin will not be connected as a part of interface connection GPIO2

Next you will connect the interface pins to an ILA debug core.

Use the **Add IP** command to instantiate an ILA core into the design, and configure it to support either Native or AXI mode. In this case you must configure the ILA to support Native mode, since you are not monitoring an AXI interface. You also need to configure two probes on the ILA core: one that is 8-bits wide, and one that is 4-bits wide.

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.



**Figure 125: Connect the ILA probes to the input/output pins that need to be monitored**

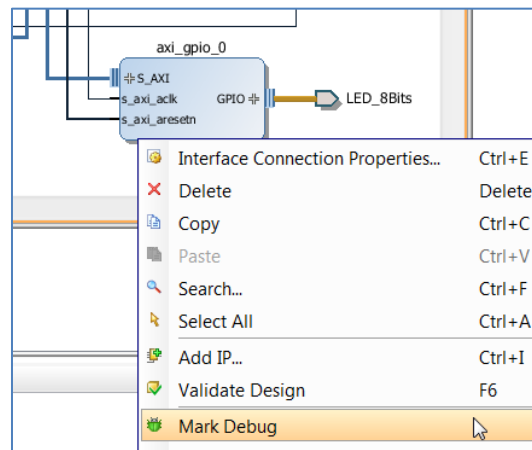
**Figure 125** shows the ILA probe pins connected to the internal pins of the GPIO interface pins.

## Using the Netlist Insertion Flow in IP Integrator

In this flow, you mark nets that you are interested in analyzing in the block design. Marking nets for debug in the block design offers more control in terms of identifying debug signals during coding and enabling/disabling debugging later in the flow.

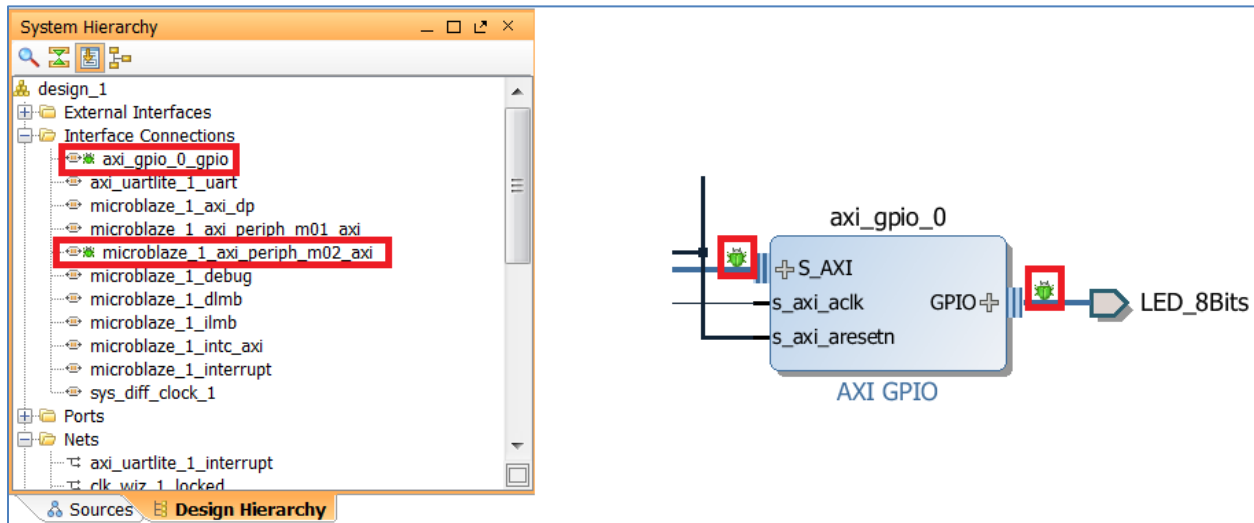
### Marking Nets for Debug in the Block Design

1. Nets can be marked for debug in the block design by highlighting them, right-clicking and selecting **Mark Debug**.




**Figure 126: Mark Nets for Debug**

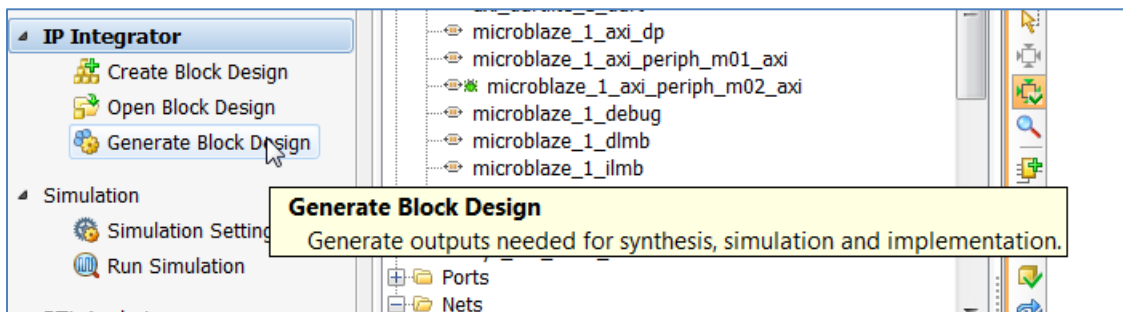
The nets that have been marked for debug will show a small bug icon placed on top of the net in the block design. Likewise, a bug icon can be seen placed on the nets to be debugged in the Design Hierarchy window as well, as shown in the following figure.



**Figure 127: Identify Nets Marked for Debug**

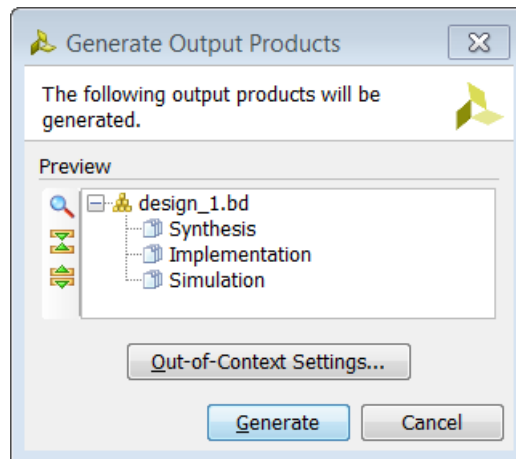
 TIP: Multiple nets can be marked for debug at the same time by highlighting them together, right-clicking and selecting **Mark Debug**.

2. You can Generate Output Products by either clicking on **Generate Block Design** in the Flow Navigator or by highlighting the block design in the sources window, right-clicking and selecting **Generate Output Products**.



**Figure 128: Generate Output Products**

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



**Figure 129: Generate Output Products Dialog Box**

4. Marking the nets for debug places the MARK\_DEBUG attribute 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.

```

1799 signal VCC_1 : STD_LOGIC;
1800 signal axi_gpio_0_gpio_TRI_0 : STD_LOGIC_VECTOR ( 7 downto 0 );
1801 attribute MARK_DEBUG : boolean;
1802 attribute MARK_DEBUG of axi_gpio_0_gpio_TRI_0 : signal is true;
1803 signal axi_uartlite_1_interrupt : STD_LOGIC;
1804 signal axi_uartlite_1_uart_RxD : STD_LOGIC;
1805 signal axi_uartlite_1_uart_TxD : STD_LOGIC;
1806 signal clk_wiz_1_locked : STD_LOGIC;
1807 signal mdm_1_debug_sys_rst : STD_LOGIC;

```

**Figure 130: MARK\_DEBUG Attributes in the Generated HDL File**

## Synthesize the Design and Insert the ILA Core

1. The next step is to synthesize the design by clicking on **Run Synthesis** from the Flow Navigator under the Synthesis drop-down list.
2. After synthesis finishes, the Synthesis Completed pop-up dialog box appears. You select **Open Synthesized Design** to open the netlist, and then click **OK**.
3. 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.

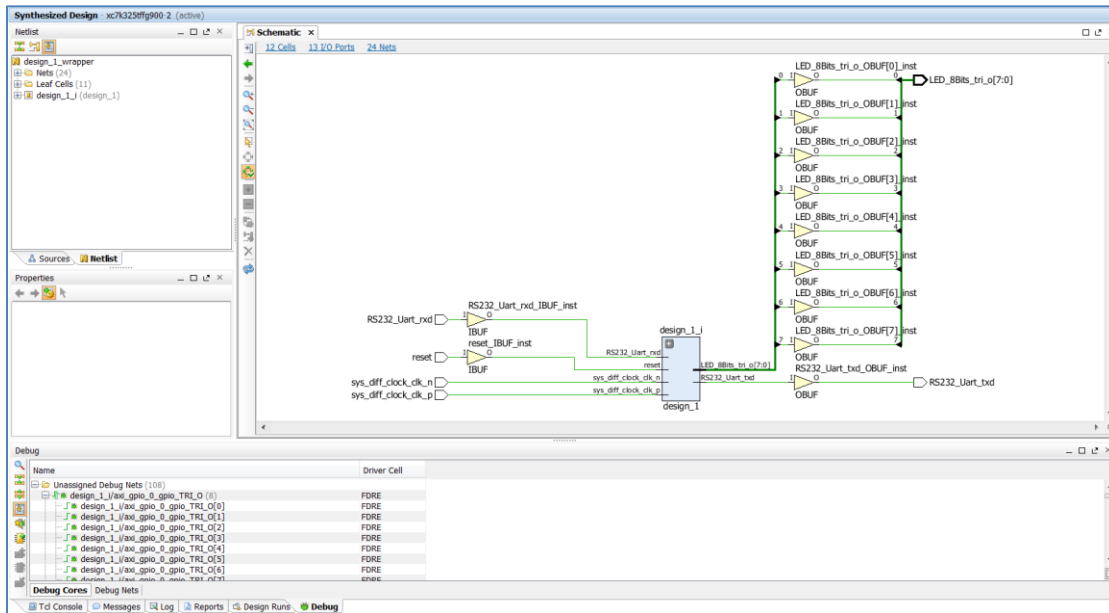


Figure 131: Schematic and Debug Window View in the Vivado IDE

- 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. This is the step where you insert an ILA core and connect these unassigned nets to the probes of the ILA. You click on the **Setup Debug** icon in the Debug window toolbar. Alternatively, you can also select **Tools > Setup Debug** from the menu.

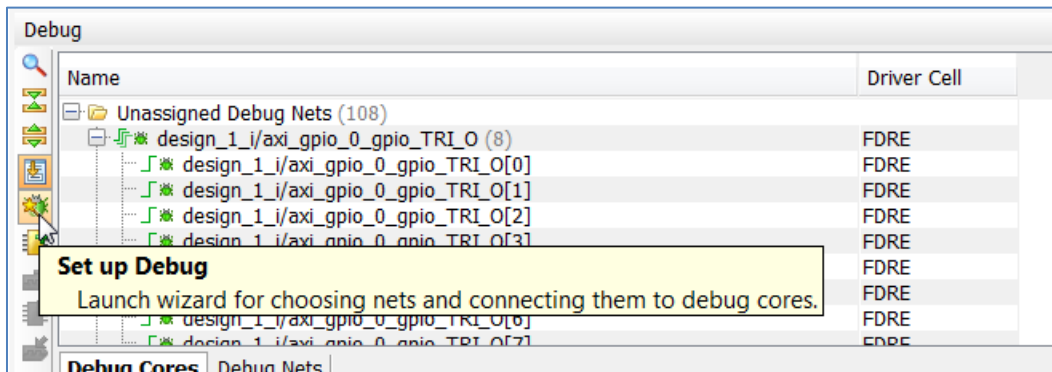
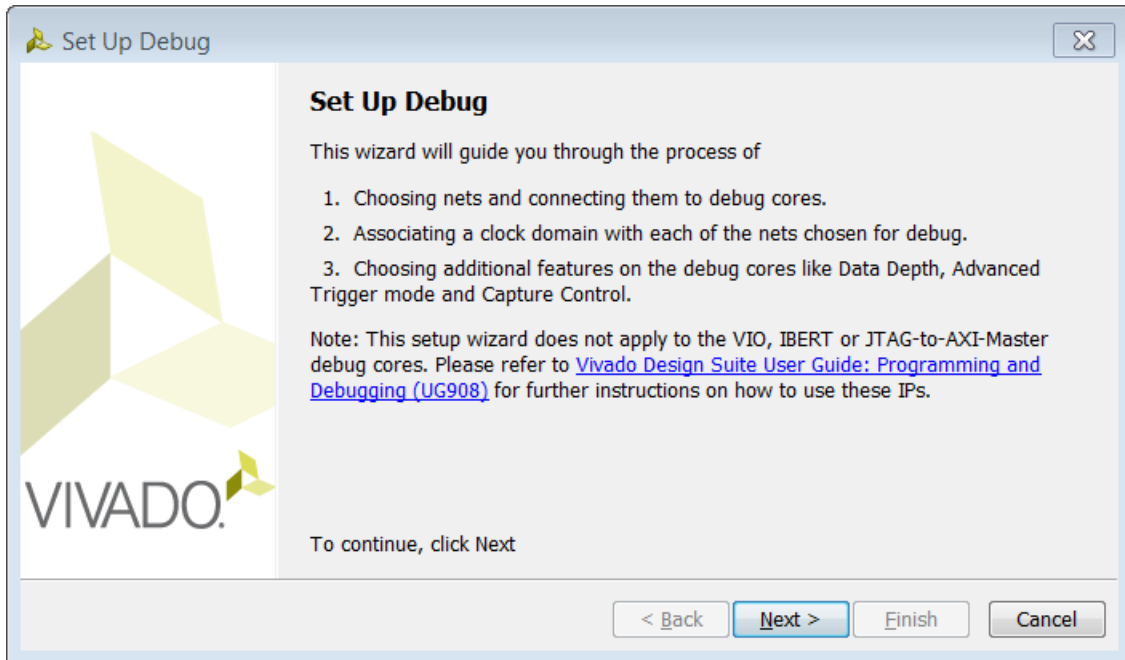


Figure 132: Setup Debug


- The Setup Debug dialog box opens. You then click **Next**.



**Figure 133: The Set up Debug Dialog Box**

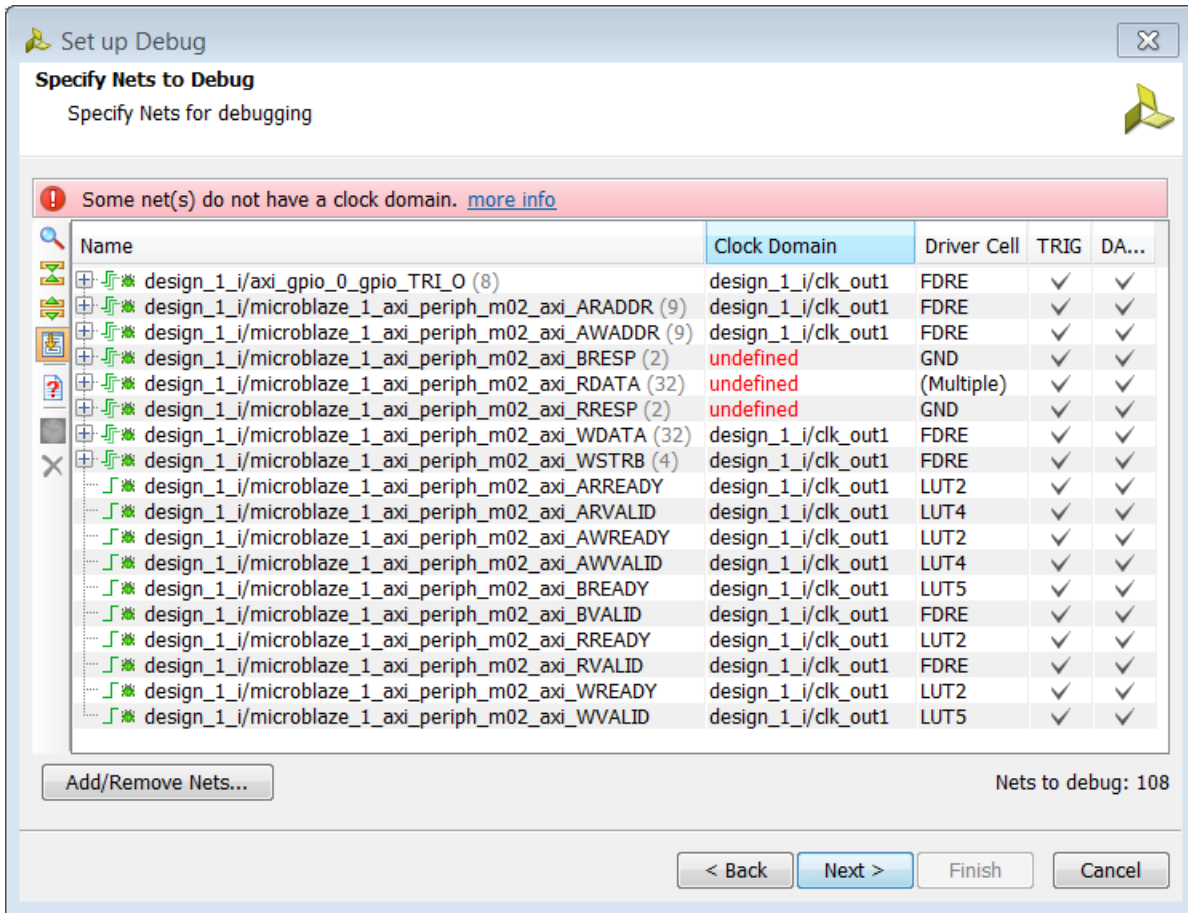
6. The Specify Nets to Debug page appears. In this page you can select a subset (or all) of the nets that you want to debug. Every signal must be associated with the same clock in an ILA. If the clock domain association cannot be found by the tool, you will manually associate those nets to a clock domain by selecting all the nets that have the Clock Domain column specified as **undefined**.

---

 **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.

---





**Figure 134: Selecting a Subset (or all) Nets Marked for Debug**

To associate a clock domain to the signals that have the Clock Domain column specified as **undefined**, select all the nets in question, right-click and choose **Select Clock Domain**.



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

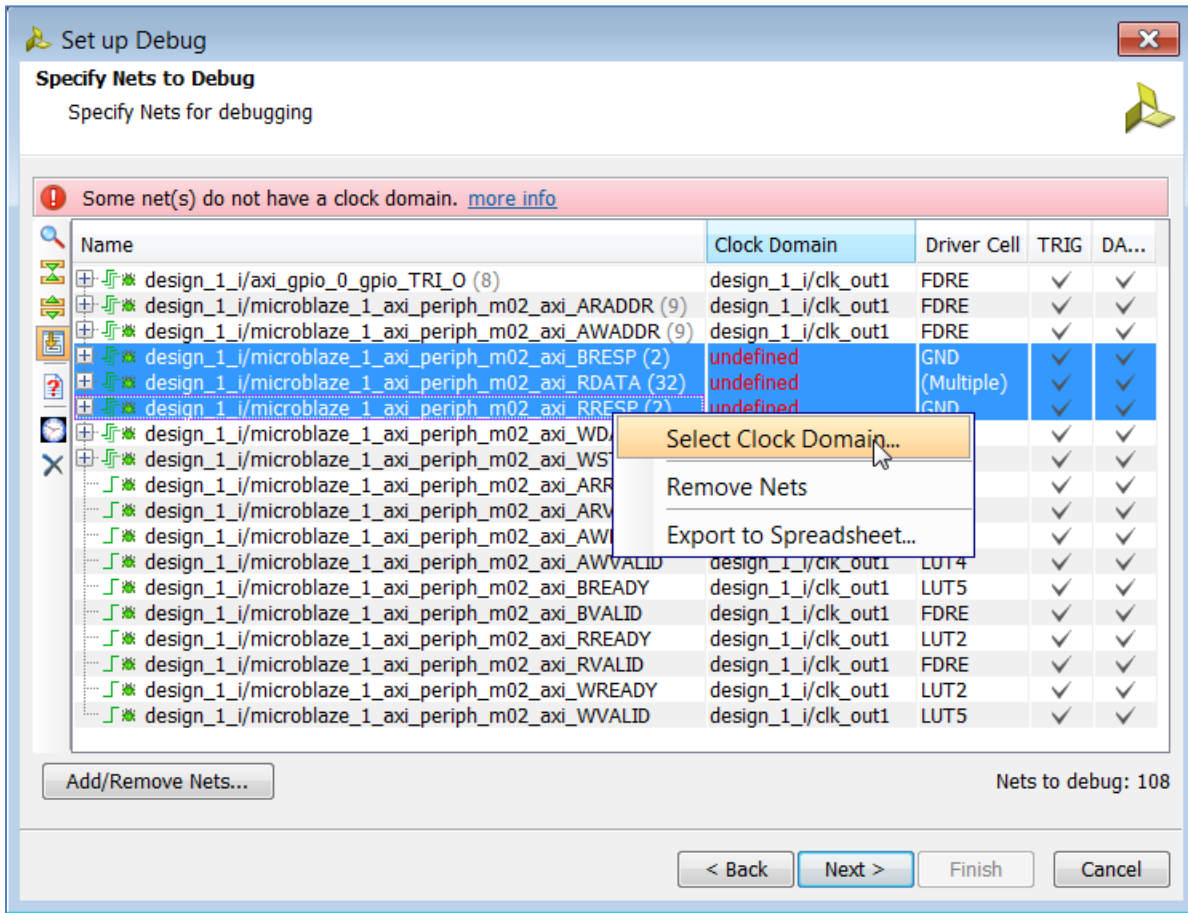
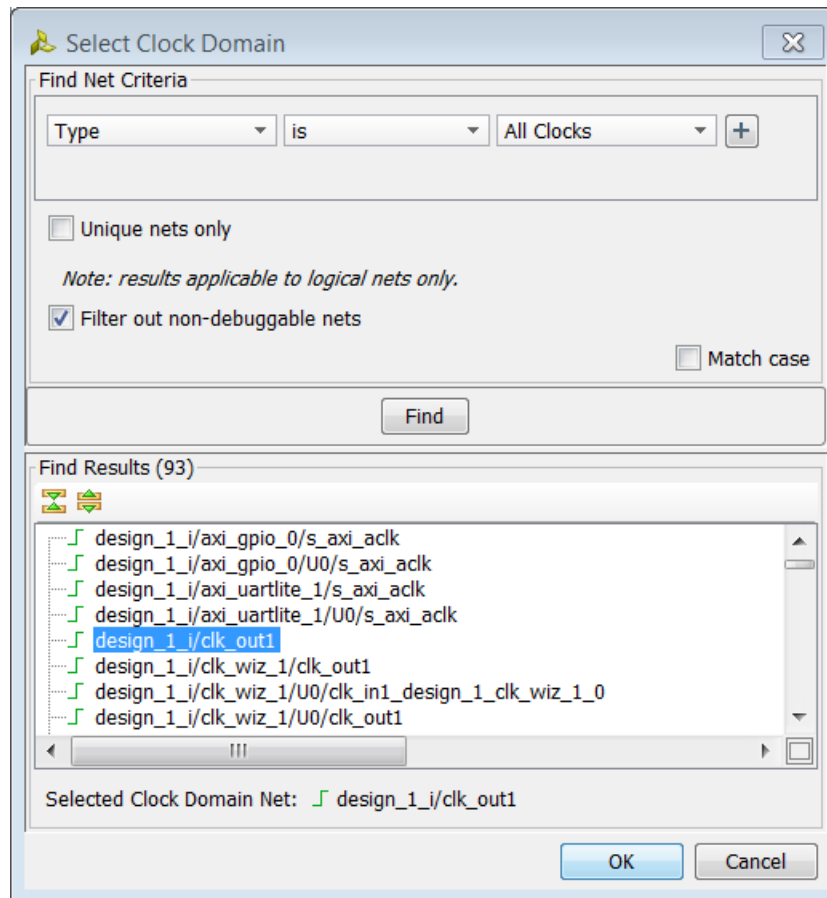


Figure 135: Select Clock Domain

7. In the Select Clock Domain dialog box, select the clock for the nets in question and click **OK**.



**Figure 136: Select Clock Domain Dialog Box**

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

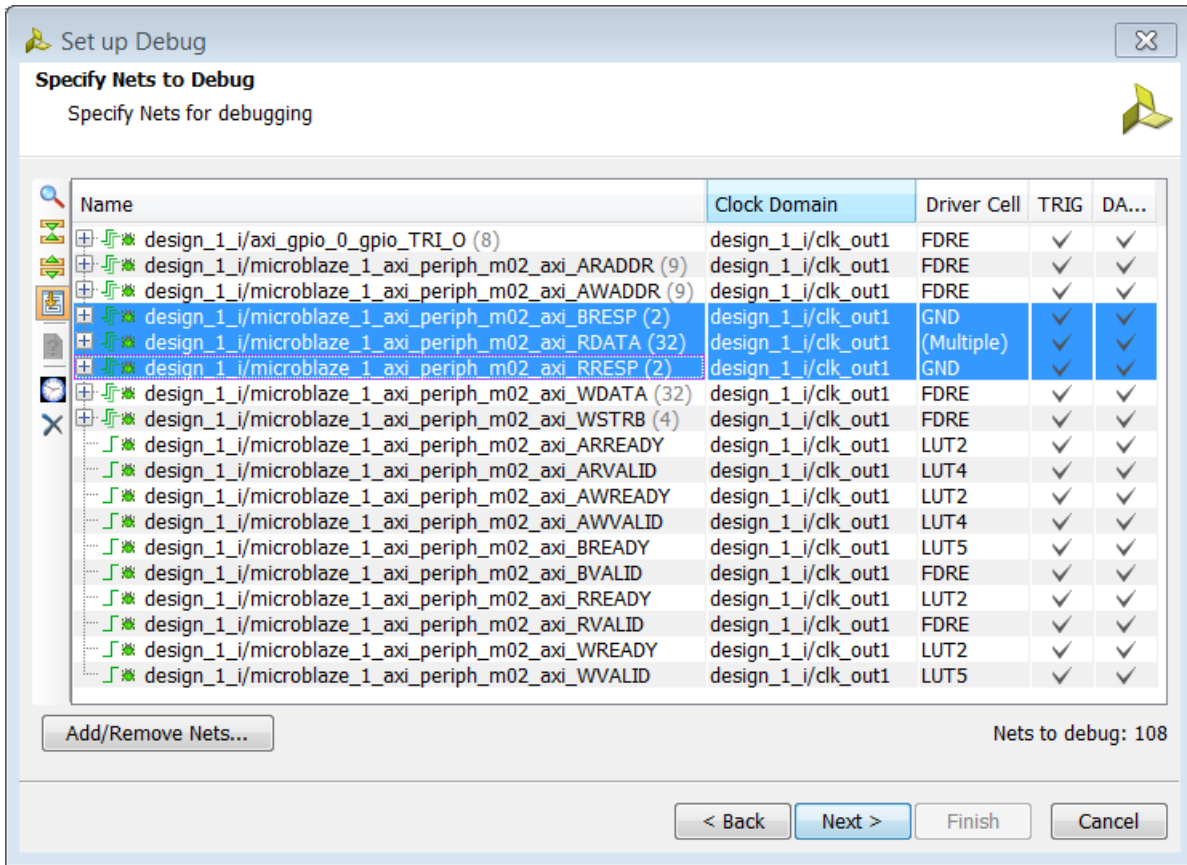


Figure 137: Specify Nets to Debug Dialog Box

9. In the ILA (Integrated Logic Analyzer) General Options page, select the appropriate options for triggering and capturing data and then click **Next**.

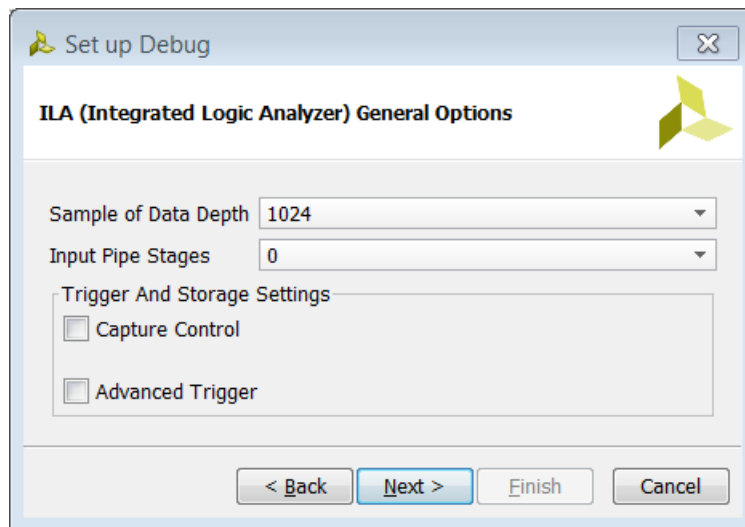
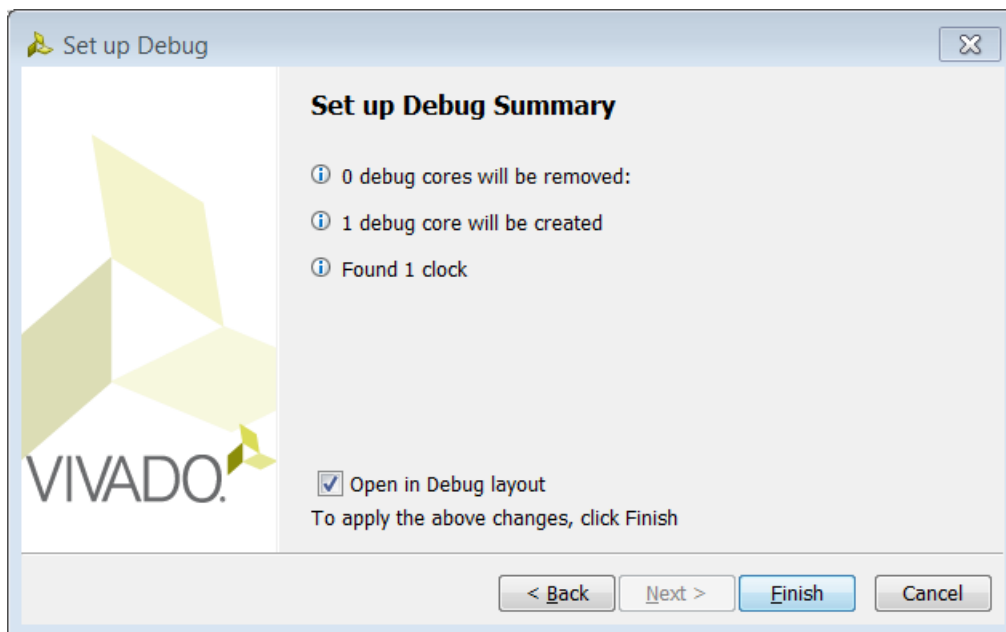


Figure 138: Setup the Trigger and Capture Modes in the ILA

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 3-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.

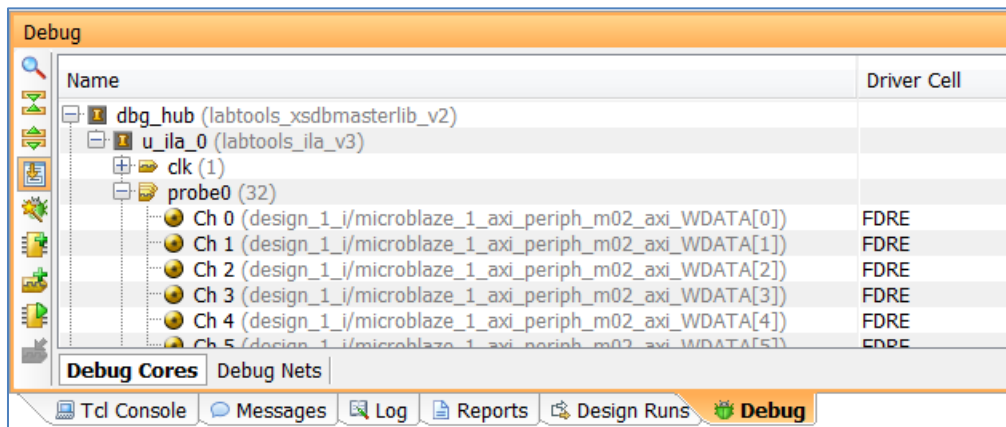
In addition to the basic capture of data, capture control capabilities allows you to only capture the data at the conditions where it matters. This will ensure that unnecessary BRAM space is not wasted and provides a highly efficient solution.

In the Summary page, you should verify that all the information looks correct and then click Finish.



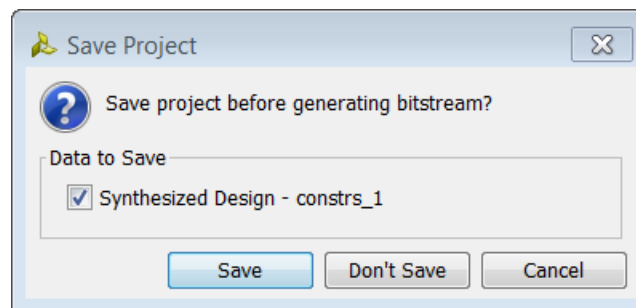
**Figure 139: Setup Debug Summary**

The Debug window looks as follows after the ILA core has been inserted. Note that 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 to it.



**Figure 140: Debug Window after ILA Insertion**

10. You are now ready to implement your design and generate a bitstream. You click on **Generate Bitstream** from the Program and Debug drop-down list in the Flow Navigator.
11. Since you have made changes to the netlist (by inserting an ILA core), a dialog box asking if the design should be saved prior to generating bitstream is displayed.



**Figure 141: Save Modified Constraints after ILA Insertion**

You can choose to save the design at this point, which will write the appropriate constraints in an active constraints file (if one exists) or will create a new constraints file. The constraints file contains all the commands to insert the ILA core in the synthesized netlist as shown below.

```

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/clock]
11 connect_debug_port u_ila_0/clock [get_nets [list design_1_i/clock_out1]]
12 set_property port_width 32 [get_debug_ports u_ila_0/probe0]
    
```

**Figure 142: XDC Constraints for ILA Core Insertion**

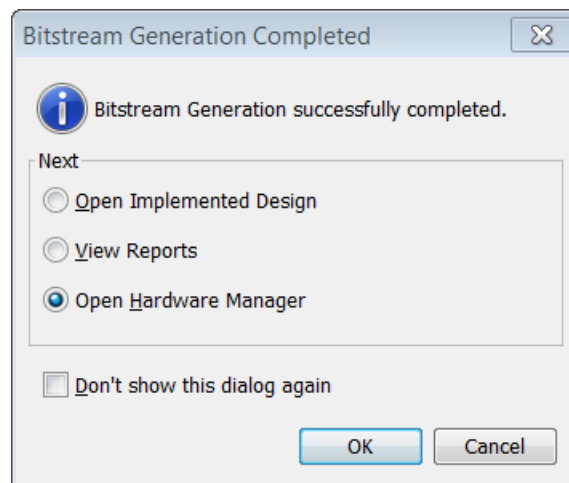
The benefit of saving the project is that if the signals marked for debug remain the same in the original block design, then there is no need to insert the ILA core after synthesis manually as these constraints will take care of it. Therefore, subsequent iteration of design changes will not require a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

If you add more nets for debug (or unmark some nets from debug) then you will still need to 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 will manually need to insert the ILA core in the synthesized netlist in subsequent iterations of the design.

## Connecting to the Target Hardware

1. Once the bitstream has been generated, the Bitstream Generation Completed dialog box pops up. You then select **Open Hardware Manager** and click **OK**.



**Figure 143: Bitstream Generation Completed**

- Click on the **Open target** link in the **Hardware Manager**.

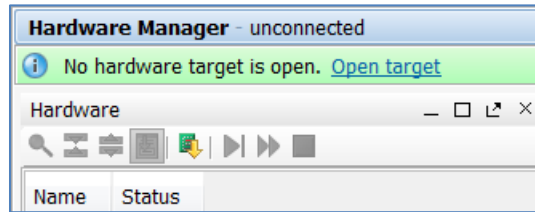


Figure 144: Open Target

- From the options presented, select any one of the three options. Selecting the **Auto Connect** option will attempt to connect to a target board connected locally to the machine. Selecting the **Recent Targets** will present a list of the recently opened targets, which can be connected to. Finally, **Open New Target** will present a dialog box to connect to the target hardware that may be connected locally or on a remote machine.

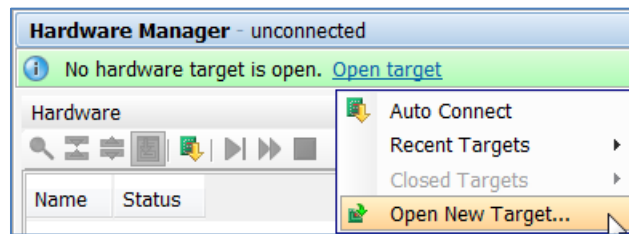


Figure 145: Connecting to a target board

- Selecting the **Open New Target** option opens the Open New Hardware Target dialog box. Next, in the Hardware Server Settings page you specify whether you want to connect to a **Local Server** or a **Remote Server**.

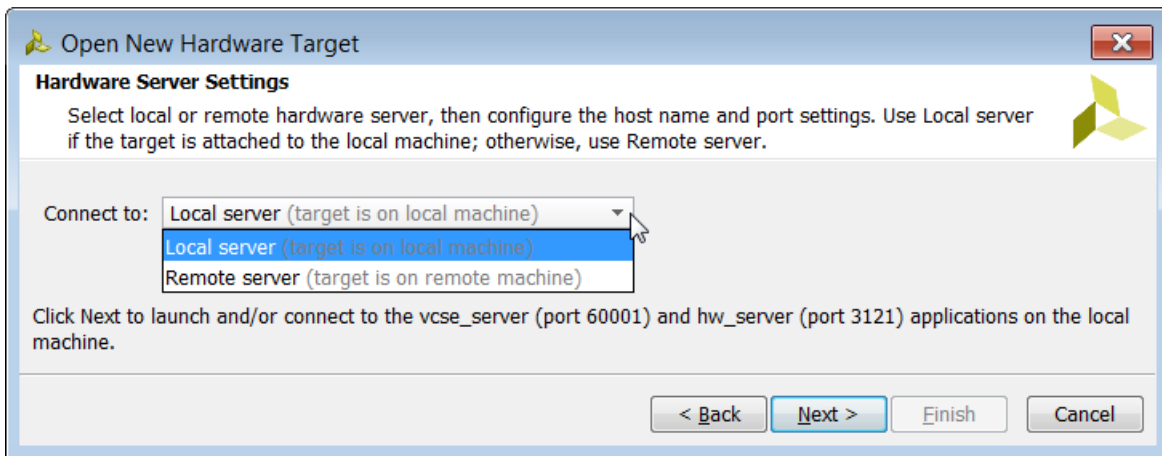
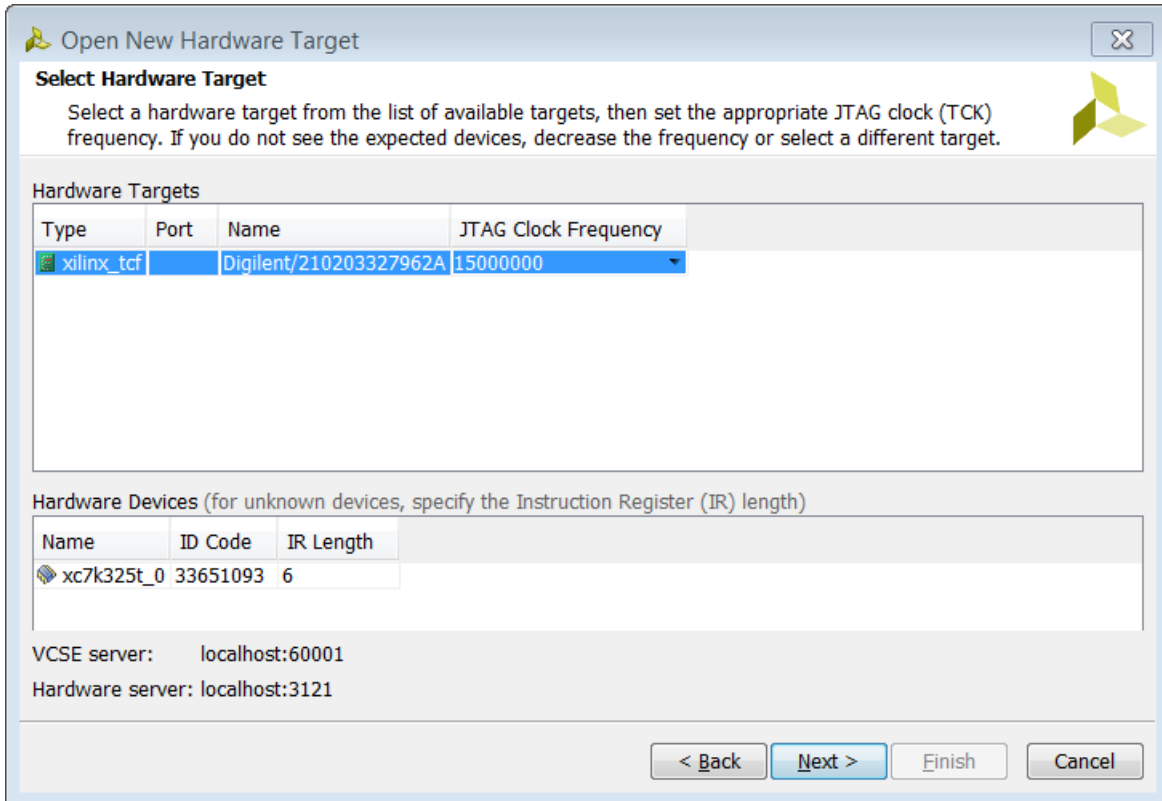


Figure 146: Connect to Hardware Server



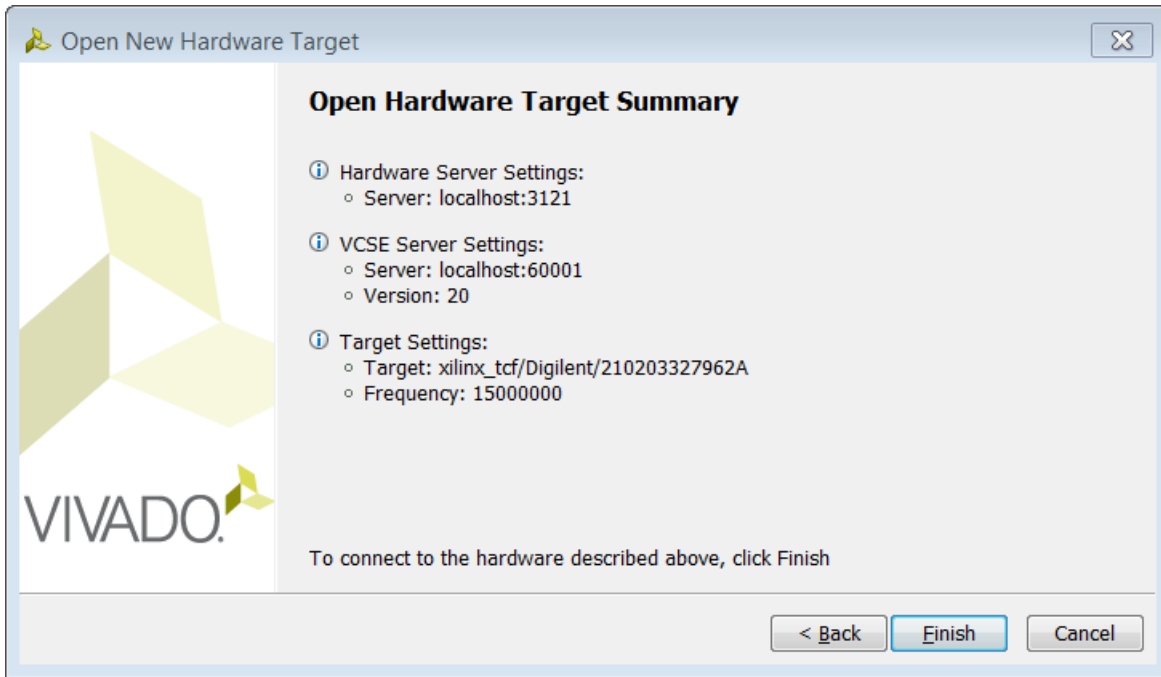
**Note:** Depending on your connection speed, this may take about 10~15 seconds.

5. If you are connecting to a Remote Server, you also need to specify the Host name and the Port. Click **Next**. Refer to *Vivado Design Suite User Guide: Programming and Debugging (UG908)* for more information on running and connecting to the hardware server.
6. If there is more than one target connected to the hw\_server, you will see multiple entries in the **Select Hardware Target** dialog box. In this case, there is only one target as shown in the following figure. You then click **Next**.



**Figure 147: Select Hardware Target**

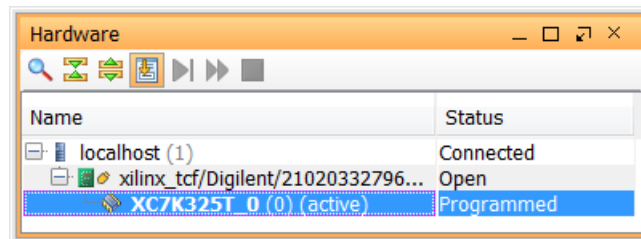
7. In the **Open Hardware Target Summary** page, click **Finish** as shown in the following figure.



**Figure 148: Open Hardware Summary**

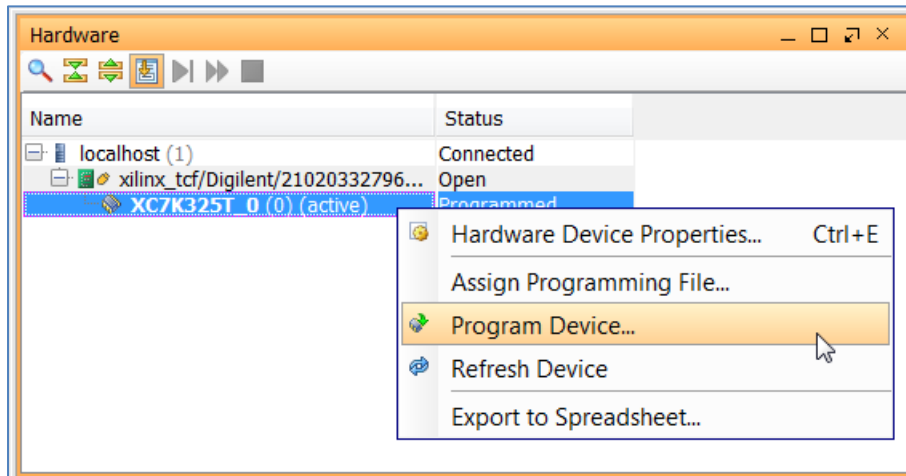
- Wait for the connection to the hardware to complete. The dialog in the following figure appears while the hardware is connecting.

Once the connection to the hardware target is made, the dialog shown in the following figure appears. The **Hardware** tab in the **Debug** view shows the hardware target and XC7K325T device that was detected in the JTAG chain.



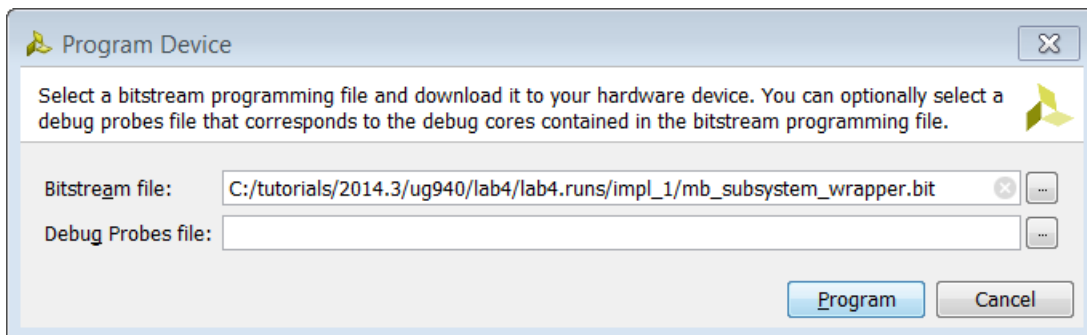
**Figure 149: Hardware Window shows Target and Device**

- Next, program the XC7K325T device using the .bit bitstream file that was created previously by right-clicking on the **XC7K325T** device and selecting **Program Device** as shown in the following figure.



**Figure 150: Program Device**

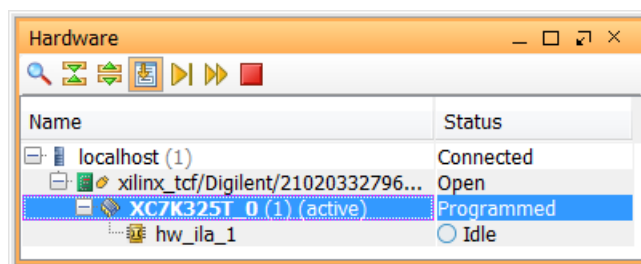
10. In the Program Device dialog box, you should verify that the BIT file is correct for the design that you are working on. Also, specify the correct probes file in the Debug Probes File field and click the **Program** button, as shown in the following figure, to program the device.



**Figure 151: Select Bitstream File to Download**

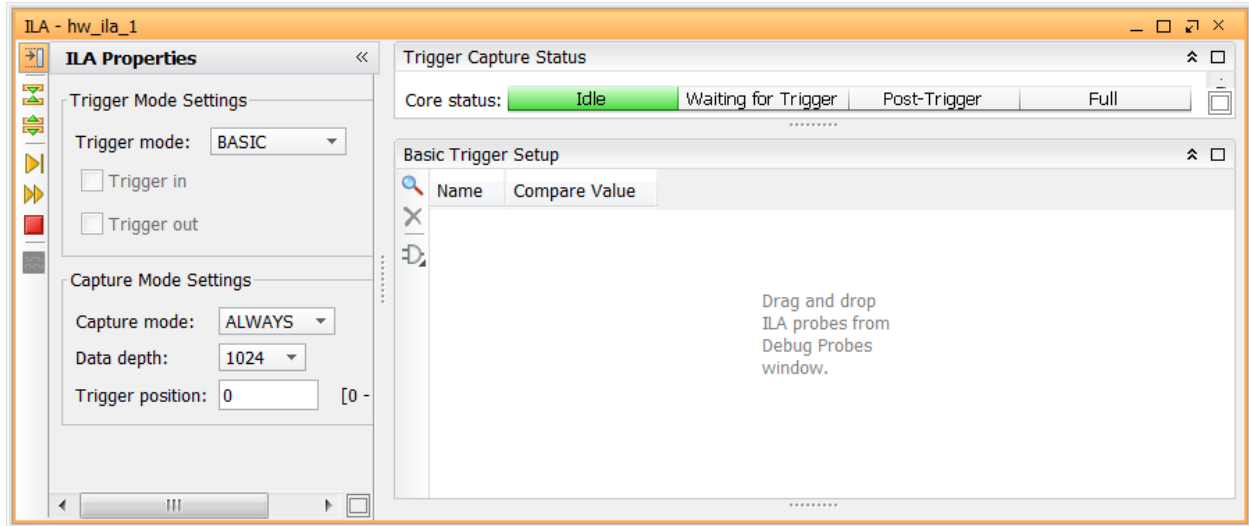
**Note:** Wait for the program device operation to complete. This may take few minutes.

Ensure that an ILA core was detected in the Hardware panel of the Debug view.



**Figure 152: ILA Core Detected**

The Integrated Logic Analyzer window opens.



**Figure 153: The Vivado Integrated Logic Analyzer Window**

For more information on Programming and Debug please refer to document *Vivado Design Suite User Guide: Programming and Debugging* ([UG908](#)).

## Overview

Typically you will create a new design in a project-based flow in the Vivado IDE GUI environment. Once the initial design has been put together, you may 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.

## Create a Design in the Vivado IDE GUI

Create a project and a new block design in the Vivado IDE GUI as mentioned in Chapter 2 of this user guide. Once the block design is complete, your canvas will contain a design like the example in the following figure.

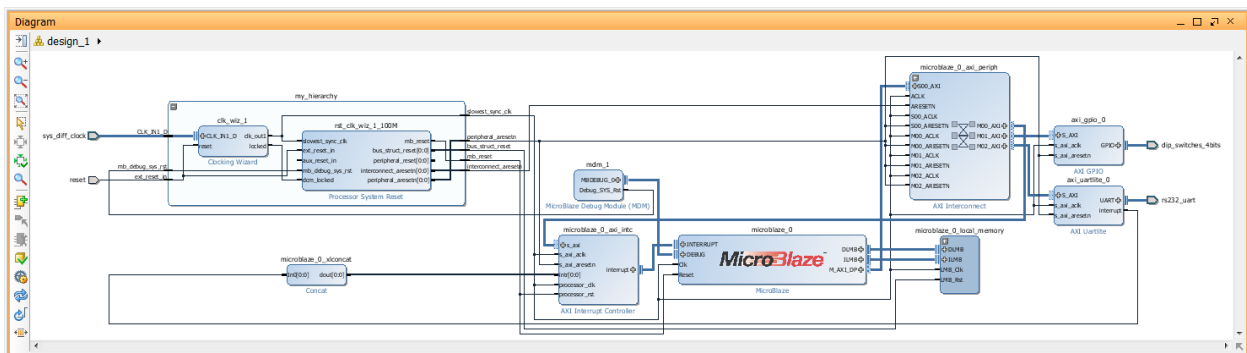


Figure 154: Block Design canvas

With the block design open, in the Tcl console type the following command:

```
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 information embedded in it about the version of the Vivado tools that it was created in and as such this file should not be used across different releases of the Vivado Design Suite. The Tcl file also contains information about all the IP present in the block design, their configuration and the connectivity.

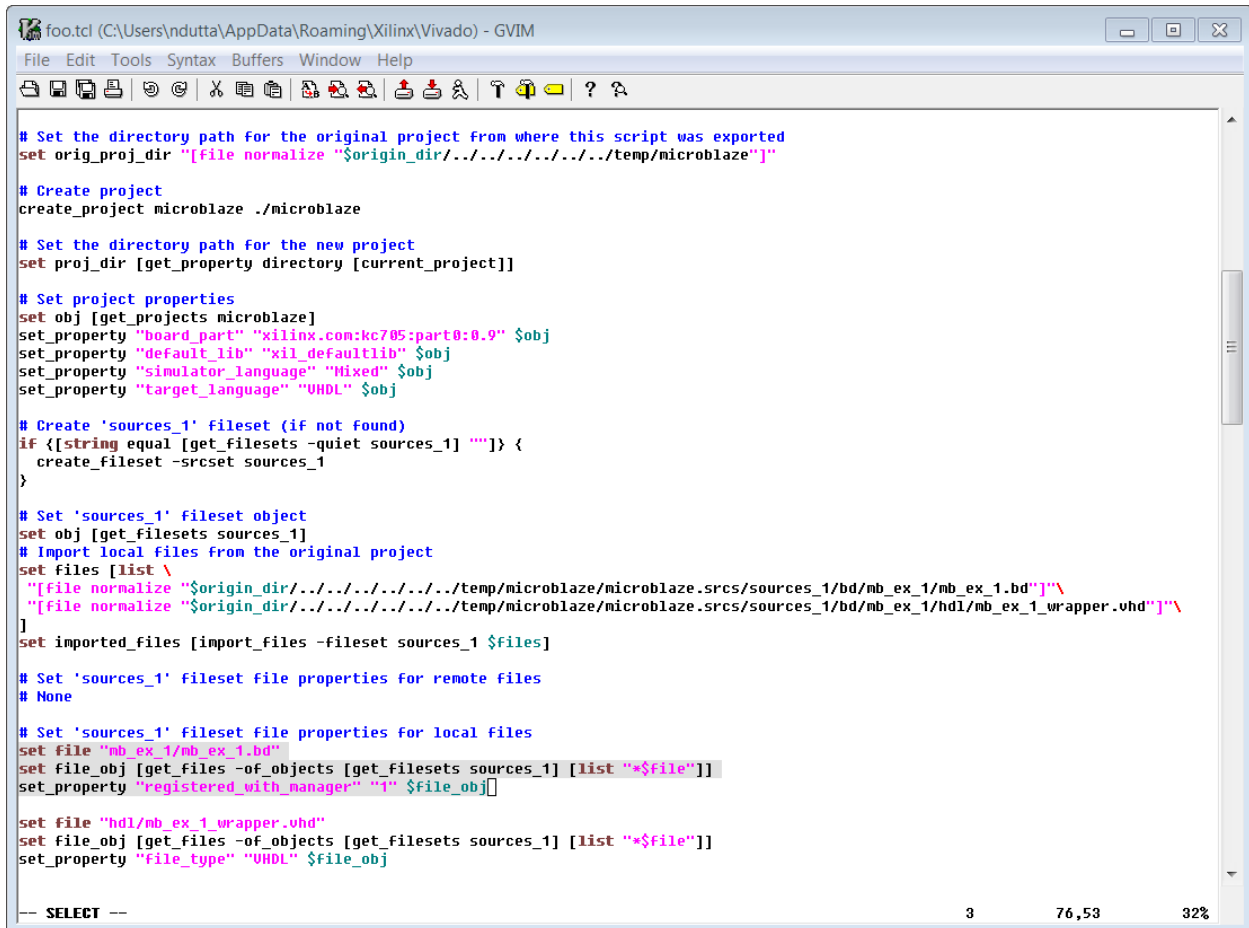
**CAUTION!** *The script produced by `write_bd_tcl` should only be used in the release it was created in. The script are not intended for use in versions of the Vivado Design Suite it was not produced in.*

## Save the Vivado Project Information in a Tcl File

The overall project settings can be saved by using the `write_project_tcl` command.

```
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 may look something as follows:



```
foo.tcl (C:\Users\ndutta\AppData\Roaming\Xilinx\Vivado) - GVIM
File Edit Tools Syntax Buffers Window Help
# Set the directory path for the original project from where this script was exported
set orig_proj_dir "[file normalize "$origin_dir/../../../../../../../../temp/microblaze"]"

# Create project
create_project microblaze ./microblaze

# Set the directory path for the new project
set proj_dir [get_property directory [current_project]]

# Set project properties
set obj [get_projects microblaze]
set_property "board_part" "xilinx.com:kc705:part0:0.9" $obj
set_property "default_lib" "xil_defaultlib" $obj
set_property "simulator_language" "Mixed" $obj
set_property "target_language" "UHDL" $obj

# Create 'sources_1' fileset (if not found)
if {[string equal [get_filesets -quiet sources_1] ""]} {
  create_fileset -srcset sources_1
}

# Set 'sources_1' fileset object
set obj [get_filesets sources_1]
# Import local files from the original project
set files [list \
  "[file normalize "$origin_dir/../../../../../../../../temp/microblaze/microblaze.srcs/sources_1/bd/mb_ex_1/mb_ex_1.bd"]" \
  "[file normalize "$origin_dir/../../../../../../../../temp/microblaze/microblaze.srcs/sources_1/bd/mb_ex_1/hdl/mb_ex_1_wrapper.uhd"]" \
]
set imported_files [import_files -fileset sources_1 $files]

# Set 'sources_1' fileset file properties for remote files
# None

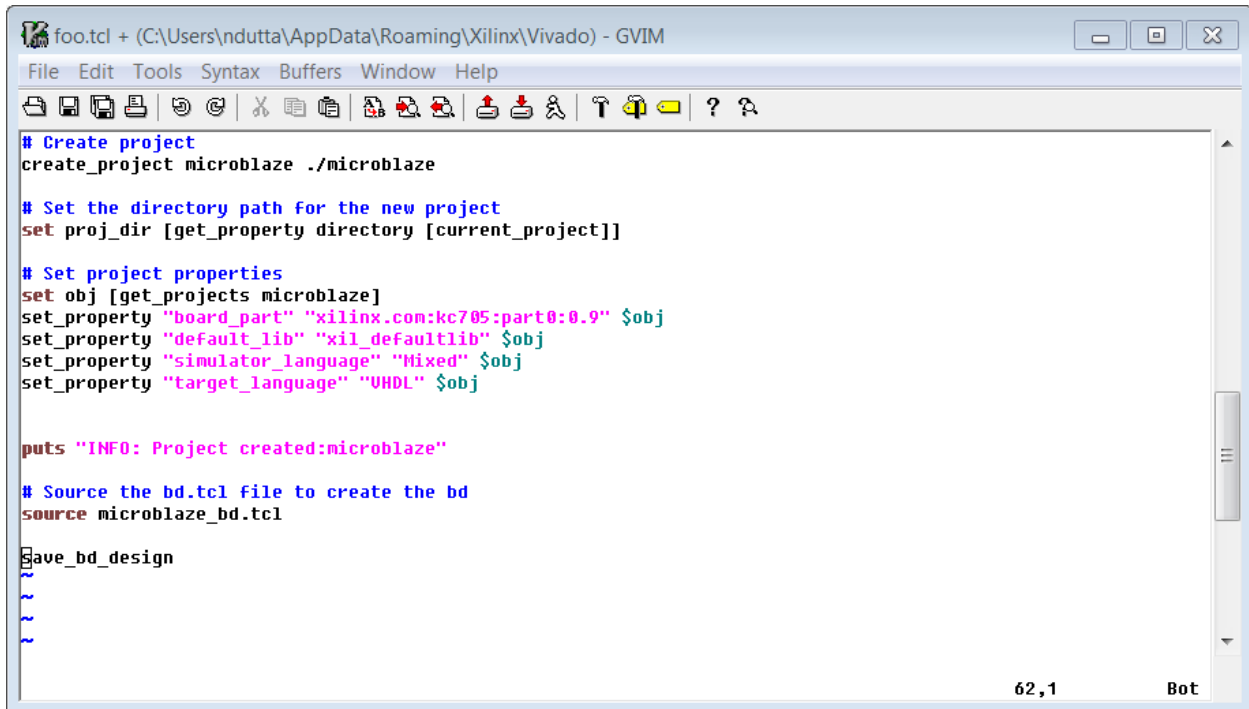
# Set 'sources_1' fileset file properties for local files
set file "mb_ex_1/mb_ex_1.bd"
set file_obj [get_files -of_objects [get_filesets sources_1] [list "$file"]]
set_property "registered_with_manager" "1" $file_obj

set file "hdl/mb_ex_1_wrapper.uhd"
set file_obj [get_files -of_objects [get_filesets sources_1] [list "$file"]]
set_property "file_type" "UHDL" $file_obj

-- SELECT --
3 76,53 32%
```

**Figure 155: Code Snippet from the Tcl File Generated by using the `write_project_tcl` Command**

In the above Tcl file, the block design file `.bd` is read explicitly as shown by the highlighted code. If you choose not to re-create the block design and just read the already created block design, then the above lines of code do not need to be modified. However, there are cases in which you may want to re-create the block design. For this purpose, the Tcl file generated by using the `write_project_tcl` command can be modified as follows:



```

foo.tcl + (C:\Users\ndutta\AppData\Roaming\Xilinx\Vivado) - GVIM
File Edit Tools Syntax Buffers Window Help
# Create project
create_project microblaze ./microblaze

# Set the directory path for the new project
set proj_dir [get_property directory [current_project]]

# Set project properties
set obj [get_projects microblaze]
set_property "board_part" "xilinx.com:kc705:part0:0.9" $obj
set_property "default_lib" "xil_defaultlib" $obj
set_property "simulator_language" "Mixed" $obj
set_property "target_language" "VHDL" $obj

puts "INFO: Project created:microblaze"

# Source the bd.tcl file to create the bd
source microblaze_bd.tcl

save_bd_design
~
~
~
62,1 Bot
    
```

**Figure 156: Code Snippet from the Tcl File to Recreate the Block Design using the Output file**

As can be seen from the above code snippet, the Tcl file from the write\_project\_tcl file has been modified to source the output file that was created using write\_bd\_tcl command. This will re-create the block design every time the Tcl file is run.

---

### Overview

Non-Project Mode is for users who want to manage their own design data or track the design state. In this flow, Vivado® tools read the various source files and implement the design through the entire flow in-memory. At any stage of the implementation process, you can generate a variety of reports based on your script. When running in Non-Project Mode, it is also important to note that this mode does not enable project-based features such as source file and run management, 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.

---

### Creating a Flow in Non-Project Mode

Vivado tools can be invoked in Tcl mode instead of the usual Project Mode by issuing following commands in the Tcl Console. The recommended approach in this mode is to create a Tcl script and source it from the Vivado prompt:

```
Vivado% vivado -mode tcl
```

In non-project mode, you have to create an in-memory project, and set your project options as shown below:

```
create_project -in_memory -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. Therefore, it is recommended that you specify the `create_project -in_memory` command to define the desired part settings before issuing the `read_bd` command.

Once 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 that has been created in a Project Mode with the entire directory structure of the block design intact, you can add the block design using the `read_bd` tcl command as follows:

```
Vivado% read_bd <absolute path to the bd file>
```



**CAUTION!** You need to have the project settings (board, part and user repository) match with the project settings of the original project in which the bd was created. Otherwise, the IP in the block design will be locked.



Once the block design is added successfully, you need to add your top-level RTL files and any top-level XDC constraints.

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

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

```
Vivado% make_wrapper -files [<path to bd>/<bd instance name>.bd] -top
add_files -norecurse <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.

If setting the block design as an out-of-context module is desired, then use the commands below to generate a synthesized design check point (dcp) for the block design.

```
create_fileset -blockset -define_from <block_design_name> <block_design_name>
reset_run <block_design_name>_synth_1 -from_step synth_design
```

For a MicroBlaze-based design, you should populate the I-LMB with either a Bootloop or your own executable in ELF format. You then need to add the ELF and associate it with the MicroBlaze instance. The following steps will do this.

```
vivado% add_files <ELF file Targeted to BRAM with .elf extension>
vivado% set_property MEMDATA.ADDR_MAP_CELLS {<bd instance name>/microblaze_0}
[get_files <BRAM Targeted ELF File>]
```

If the design has multiple levels of hierarchy, you need to ensure that the correct hierarchy is provided. After this, you need to go through the usual synthesis, Place and Route steps to get the design implemented. One aspect that needs to be kept in mind is that for the synthesis (synth\_design) step, you need to provide the target part as the default target part, which may not be the same as the desired one.

```
synth_design -top <path to top level wrapper file> -part <part>
opt_design
place_design
route_design write_bitstream top
```

Refer to the *Vivado Design Suite User Guide: Design Flows Overview* ([UG892](#)) for more details about working in a Non-Project Mode flow.

To export the implemented hardware system to SDK, you can use the following command:

```
write_sysdef -hwdef "C:/Data/ug940/lab1/zynq_design.hwdef" \
-bitfile "<project_name>/<project_name>.runs/impl_1/zynq_design.bit" \
-meminfo "<project_name>/<project_name>.runs/impl_1/zynq_design.bmm" \
-file "C:/Data/ug940/lab1/zynq_design.sysdef"
```

Refer to the *Vivado Design Suite Tcl Command Reference* ([UG835](#)) for more on the write\_sysdef or write\_hwdef commands.

### Overview

As you upgrade your Vivado® Design Suite to the latest release, you will need to upgrade the block designs created in 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 should not 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. If that is the case, then 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.

Upgrading can be done in two ways:

1. Using the Vivado IDE GUI in the Project Mode Flow
2. Using a Tcl script in the Non-Project Mode Flow

Both methods are described in this Chapter.

---

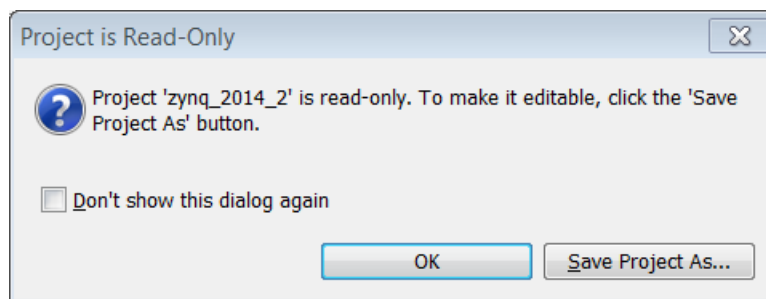
### Upgrading a Block Design in Project Mode

1. Launch the latest version of the Vivado Design Suite.
2. From the Vivado IDE main page, click on **Open Project** and navigate to the design that was created from a previous version of Vivado tools.
3. The **Older Project Version** pop-up opens. **Automatically upgrade for the current version** is selected by default. Although you can upgrade the design from a previous version by selecting the **Automatically upgrade for the current version**, it is highly recommended that you save your project with a different name before upgrading. To do this, select **Open project in read-only mode** and click **OK**.



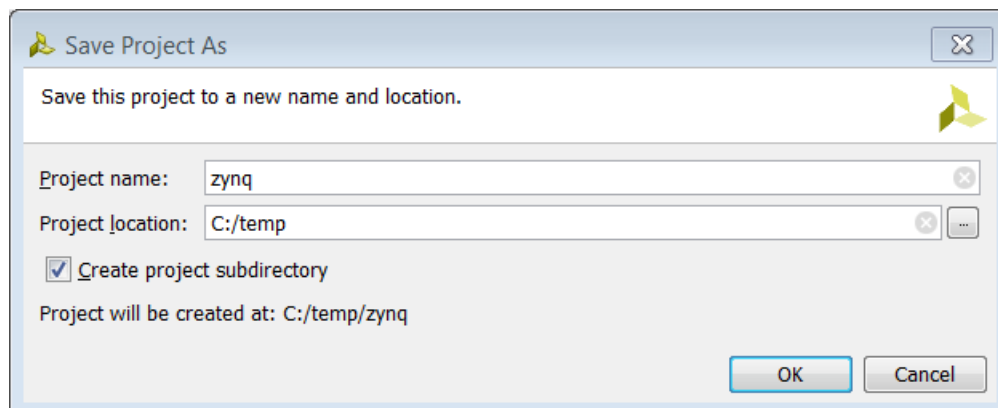
**Figure 157: Open Project in Read-Only Mode**

4. The Project is Read-Only dialog box pops-up. Select **Save Project As...**



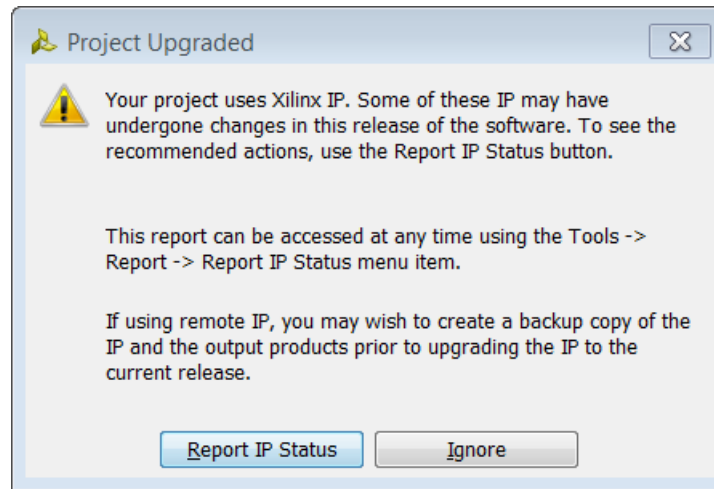
**Figure 158: Save Project As**

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



**Figure 159: Specify Project Name**

- The Project Upgraded dialog box opens, informing you that the IP used in the design may have changed and therefore need to be updated.

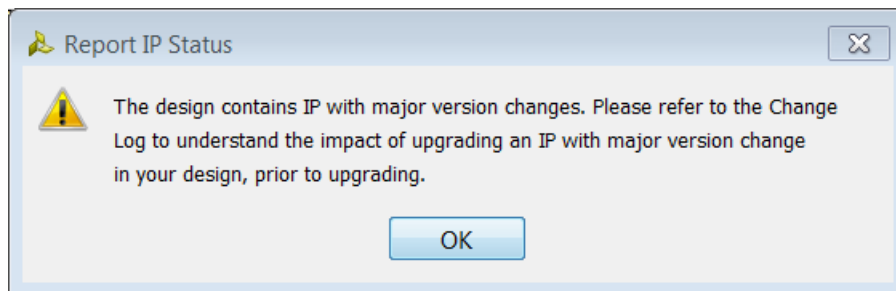


**Figure 160: Project Upgraded dialog box**

- Click on **Report IP Status**.

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

- If any of the IP in the design has gone through a major version change, then the following message will pop-up on the screen. Click **OK**.



**Figure 161: Report IP Status Results**

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 may have gone through. Also realize that you cannot individually select some IP of a block design for upgrade and let the others remain in their current versions.

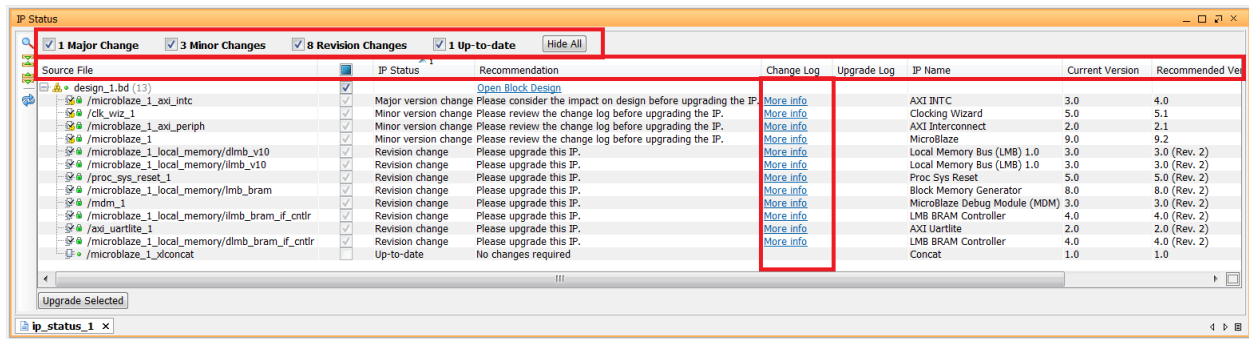


Figure 162: IP Status Report

The very 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 IP's current version is 5.0, and the upgraded version is 5.0 (Rev. 1)

You can click on the **More info...** link in the Change Log column to see a description of the change.

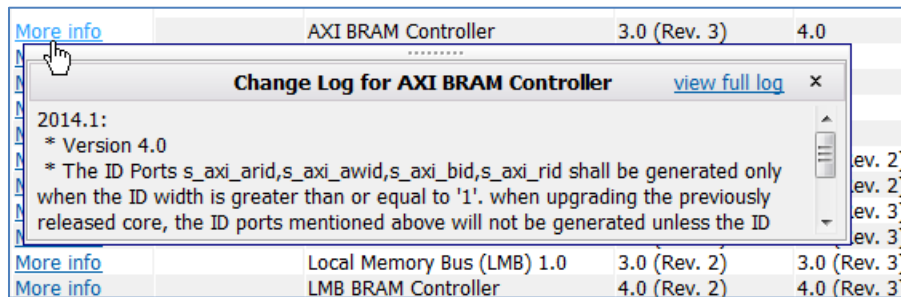


Figure 163: Inspect the Change Log by Clicking on More Info link

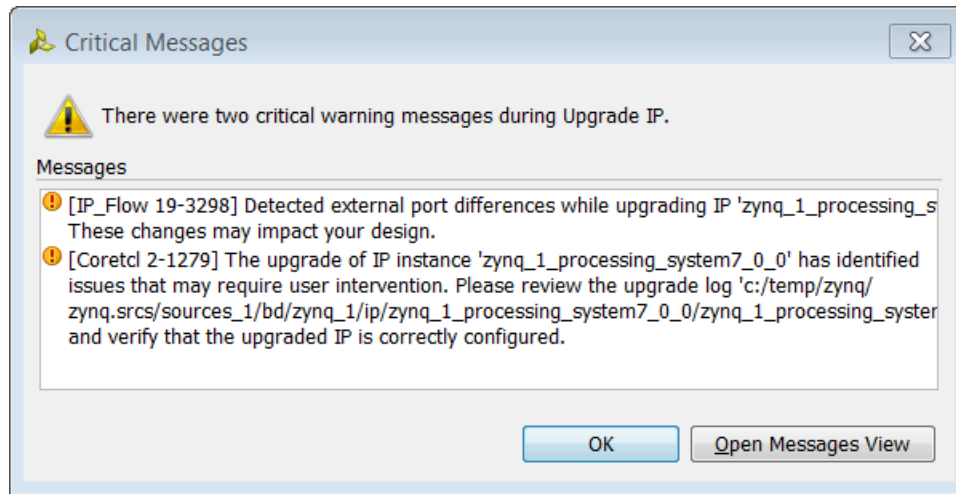
The Recommendation column also suggests that you need to understand what the changes are before selecting the IP for upgrade.

9. Once you understand the changes and the impact of them on your design, you should click on **Upgrade Selected**.

10. The Upgrade IP dialog box pops up to confirm that you want to proceed with upgrade.

11. Click **OK**.

12. When the upgrade process is complete, a Critical Messages dialog box may pop-up informing you about any critical issues that you need to pay attention to.



**Figure 164: Critical Messages dialog box**

13. You should review any critical warnings and other messages that may be flagged as a part of the upgrade. Click **Ok**.
14. If there are multiple diagrams in the design, the IP Status window will show the status of IP in all the diagrams as shown below.

| Source File                                  | Upgrade                             | IP Status                           | Recommendation  | Change Log   | IP Name                       | Current Version | Recommended Version | License  |
|--|-------------------------------------|-------------------------------------|---|--------------|-------------------------------|-----------------|---------------------|----------|
| mb_ex_1.bd (12)                              | <input checked="" type="checkbox"/> | <input checked="" type="checkbox"/> | Open Block Design...  |              |                               |                 |                     |          |
| /mb_1  | <input checked="" type="checkbox"/> | <input checked="" type="checkbox"/> | IP definition minor version change. Please review the chan... | More info... | MicroBlaze                    | 9.0             | 9.1                 | Included |
| /gpio_1                                      | <input checked="" type="checkbox"/> | <input checked="" type="checkbox"/> | IP definition revision change. Please upgrade this IP.        | More info... | AXI GPIO                      | 2.0             | 2.0 (Rev. 1)        | Included |
| /proc_reset_1                                | <input checked="" type="checkbox"/> | <input checked="" type="checkbox"/> | IP definition revision change. Please upgrade this IP.        | More info... | Proc Sys Reset                | 5.0             | 5.0 (Rev. 1)        | Included |
| /mb_1_local_memory/lmb_v10                   | <input checked="" type="checkbox"/> | <input checked="" type="checkbox"/> | IP definition revision change. Please upgrade this IP.        | More info... | Local Memory Bus (LMB) 1.0    | 3.0             | 3.0 (Rev. 1)        | Included |
| /mb_1_axi_periph                             | <input checked="" type="checkbox"/> | <input checked="" type="checkbox"/> | IP definition revision change. Please upgrade this IP.        | More info... | AXI Interconnect              | 2.0             | 2.0 (Rev. 1)        | Included |
| /mb_1_local_memory/lmb_bram_if_cntr          | <input checked="" type="checkbox"/> | <input checked="" type="checkbox"/> | IP definition revision change. Please upgrade this IP.        | More info... | LMB BRAM Controller           | 4.0             | 4.0 (Rev. 1)        | Included |
| /clk_wiz_1                                   | <input checked="" type="checkbox"/> | <input checked="" type="checkbox"/> | IP definition revision change. Please upgrade this IP.        | More info... | Clocking Wizard               | 5.0             | 5.0 (Rev. 1)        | Included |
| /mb_1_local_memory/dlmb_bram_if_cntr         | <input checked="" type="checkbox"/> | <input checked="" type="checkbox"/> | IP definition revision change. Please upgrade this IP.        | More info... | LMB BRAM Controller           | 4.0             | 4.0 (Rev. 1)        | Included |
| /mb_1_local_memory/dlmb_v10                  | <input checked="" type="checkbox"/> | <input checked="" type="checkbox"/> | IP definition revision change. Please upgrade this IP.        | More info... | Local Memory Bus (LMB) 1.0    | 3.0             | 3.0 (Rev. 1)        | Included |
| /mdm_1                                       | <input checked="" type="checkbox"/> | <input checked="" type="checkbox"/> | IP definition revision change. Please upgrade this IP.        | More info... | MicroBlaze Debug Module (MDM) | 3.0             | 3.0 (Rev. 1)        | Included |
| /uartlite_1                                  | <input checked="" type="checkbox"/> | <input checked="" type="checkbox"/> | IP definition revision change. Please upgrade this IP.        | More info... | AXI Uartlite                  | 2.0             | 2.0 (Rev. 1)        | Included |
| /mb_1_local_memory/lmb_bram                  | <input checked="" type="checkbox"/> | <input checked="" type="checkbox"/> | IP definition revision change. Please upgrade this IP.        | More info... | Block Memory Generator        | 8.0             | 8.0 (Rev. 1)        | Included |
| mb_ex_2.bd (11)                              | <input checked="" type="checkbox"/> | <input checked="" type="checkbox"/> | Open Block Design...  |              |                               |                 |                     |          |
| /microblaze_1                                | <input checked="" type="checkbox"/> | <input checked="" type="checkbox"/> | IP definition minor version change. Please review the chan... | More info... | MicroBlaze                    | 9.0             | 9.1                 | Included |
| /axi_gpio_1                                  | <input checked="" type="checkbox"/> | <input checked="" type="checkbox"/> | IP definition revision change. Please upgrade this IP.        | More info... | AXI GPIO                      | 2.0             | 2.0 (Rev. 1)        | Included |
| /microblaze_1_local_memory/lmb_bram_if_cntr  | <input checked="" type="checkbox"/> | <input checked="" type="checkbox"/> | IP definition revision change. Please upgrade this IP.        | More info... | LMB BRAM Controller           | 4.0             | 4.0 (Rev. 1)        | Included |
| /microblaze_1_local_memory/dlmb_v10          | <input checked="" type="checkbox"/> | <input checked="" type="checkbox"/> | IP definition revision change. Please upgrade this IP.        | More info... | Local Memory Bus (LMB) 1.0    | 3.0             | 3.0 (Rev. 1)        | Included |
| /microblaze_1_local_memory/dlmb_bram_if_cntr | <input checked="" type="checkbox"/> | <input checked="" type="checkbox"/> | IP definition revision change. Please upgrade this IP.        | More info... | LMB BRAM Controller           | 4.0             | 4.0 (Rev. 1)        | Included |
| /microblaze_1_local_memory/lmb_bram          | <input checked="" type="checkbox"/> | <input checked="" type="checkbox"/> | IP definition revision change. Please upgrade this IP.        | More info... | Block Memory Generator        | 8.0             | 8.0 (Rev. 1)        | Included |
| /mdm_1                                       | <input checked="" type="checkbox"/> | <input checked="" type="checkbox"/> | IP definition revision change. Please upgrade this IP.        | More info... | MicroBlaze Debug Module (MDM) | 3.0             | 3.0 (Rev. 1)        | Included |
| /microblaze_1_axi_periph                     | <input checked="" type="checkbox"/> | <input checked="" type="checkbox"/> | IP definition revision change. Please upgrade this IP.        | More info... | AXI Interconnect              | 2.0             | 2.0 (Rev. 1)        | Included |
| /proc_sys_reset_1                            | <input checked="" type="checkbox"/> | <input checked="" type="checkbox"/> | IP definition revision change. Please upgrade this IP.        | More info... | Proc Sys Reset                | 5.0             | 5.0 (Rev. 1)        | Included |
| /clk_wiz_1                                   | <input checked="" type="checkbox"/> | <input checked="" type="checkbox"/> | IP definition revision change. Please upgrade this IP.        | More info... | Clocking Wizard               | 5.0             | 5.0 (Rev. 1)        | Included |
| /microblaze_1_local_memory/lmb_v10           | <input checked="" type="checkbox"/> | <input checked="" type="checkbox"/> | IP definition revision change. Please upgrade this IP.        | More info... | Local Memory Bus (LMB) 1.0    | 3.0             | 3.0 (Rev. 1)        | Included |

**Figure 165: IP Status Window for Multiple Diagrams**

When you click on **Upgrade Selected**, all the block diagrams are updated in the design (if the block diagrams are selected for upgrade).

## Running Design Rule Checks

From the toolbar, click on **Validate Design**.

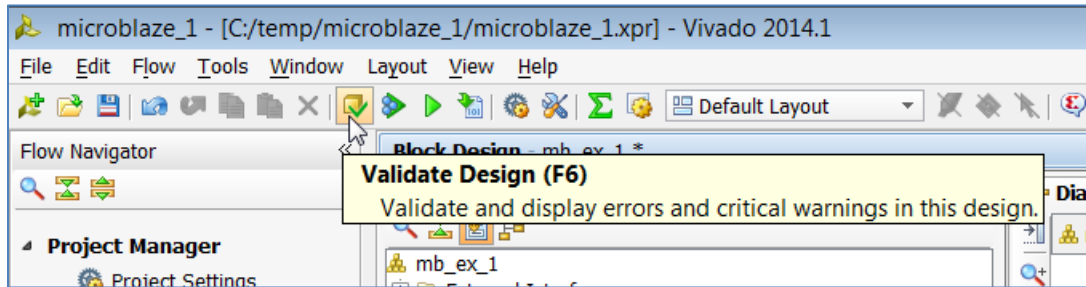



Figure 166: Validate Design

You can also do this by clicking the validate design icon  in the block design toolbar.

## Regenerating Output Products

1. In the Sources pane in Vivado, right-click on block diagram and select **Generate Output Products**.

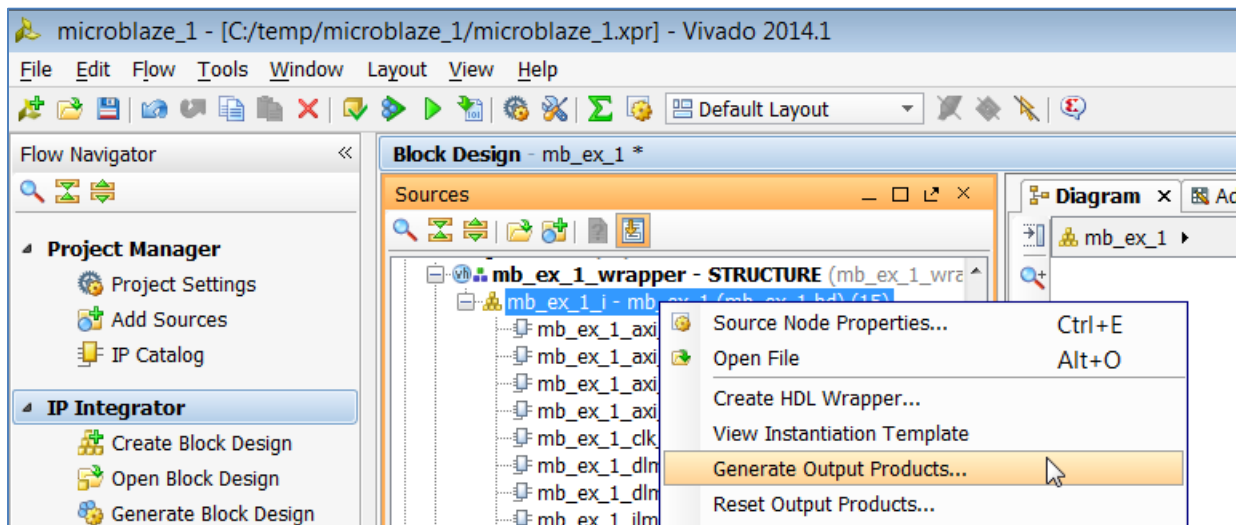
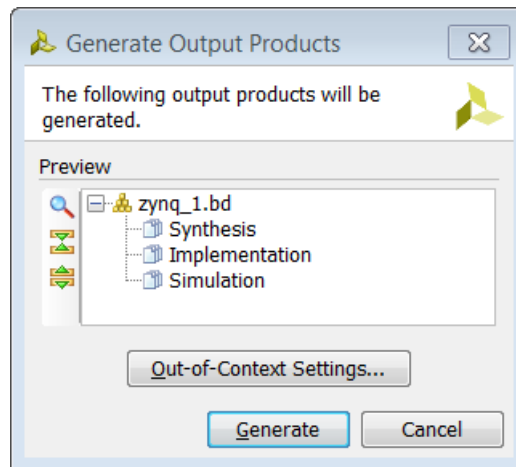


Figure 167: Generate Output Products

Alternately, you can also click on the **Generate Block Design** in the Flow Navigator under IP integrator drop-down list.



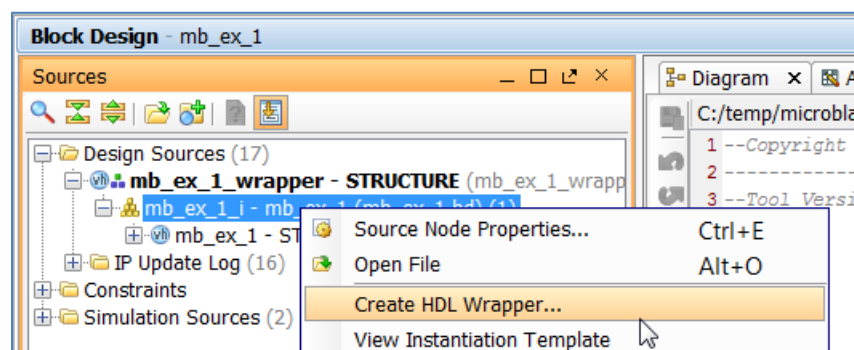
**Figure 168: Generate Output Products Dialog Box**

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

## Create/Change 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 pane in Vivado right-click on the block diagram and select **Create HDL Wrapper**.



**Figure 169: Create HDL Wrapper**

2. 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 else you can let the Vivado tools manage the wrapper file for you. Click **OK**.



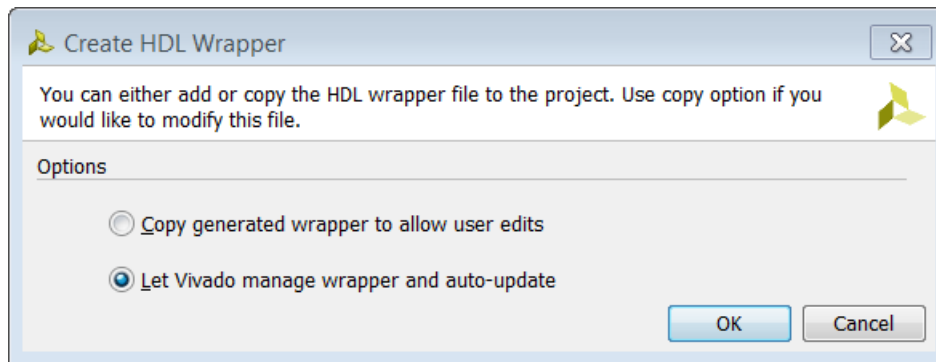


Figure 170: Create HDL Wrapper Dialog Box

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.srscs/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.srscs/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.srscs/sources_1/bd/bd_name/bd_name.bd] -top
# Overwrite any existing HDL wrapper from before
import_files -force -norecuse
<path_to_project>/project_name/project_name.srscs/sources_1/bd/bd_name/hdl/bd_name_wrap
per.v
update_compile_order -fileset sources_1
# Continue through implementation
...
```

### Overview

This chapter provides recommendations for using version control systems with IP integrator- based block designs in both Project as well as Non-Project Mode. The IP integrator tool in the Vivado® IDE is a powerful tool that enables the creation of complex IP subsystem designs. As designs get more complex, the challenge is to keep a track of different versions of the design to facilitate project management and collaboration in a team-design environment.

While a project may include multiple design sources and configuration files, only a subset of these files require revision control to recreate a project and reproduce implementation results. Some of these files applicable to block designs are:

- IP-XACT core files (.xci, .mem, .coe)
- Block diagram files (.bd)
- Embedded subsystems and files (.elf, .bmm)
- Xilinx Design constraints files (.xdc)
- Configuration files, including Vivado Simulator and Vivado Integrated Logic Analyzer configuration files (.wcfg)
- RTL file (wrappers for block designs if managed by user: .vhd, .v)

The Vivado Design Suite does not support any particular revision control system. Rather, it is designed to work with any revision control system. In order to make Vivado designs suitable for revision control, the Vivado Design Suite provides the following features:

- Updates the timestamps only when the files are modified. Accordingly, opening a project doesn't change the timestamp on it.
- Supports ASCII-based project files.
- Supports extensive Tcl scripting capabilities.

## Design Files Needed to be Checked In for Revision Control

A block design consists of several IP constructed graphically in a GUI environment. A block design directory structure in a Project Mode flow looks as shown in the following figure:




Figure 171: Vivado Project Directory Structure

In [Figure 171](#), the folders mean the following:

- project\_1 – Vivado Project folder (project\_1 is the name given to the project)
- project\_1.srcs – Sources folder containing project specific files
- sources\_1 – Folder containing all source file including the bd
- bd – contains block design specific data. May have multiple sub-directories corresponding to each block design
- design\_1 – folder containing data for the block design called design\_1. If there are multiple block designs in the project, then multiple folders will be present here.
- hdl – folder containing the top-level HDL file and the wrapper file.
- ip – contains subfolders pertaining each of the IP in the block design
- ui – folder containing GUI (IP integrator canvas) data for the block design

---

 **TIP:** *The recommendation is to put all the files contained in the bd folder including the entire directory structure under revision control.*

---

---

## Creating a Block Design for Use in a Different Project

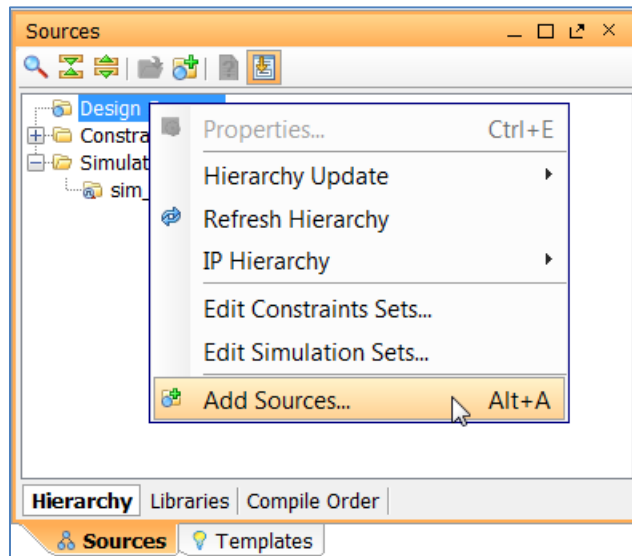
IP integrator provides the capability to re-use a block design created in a different project to be imported into other projects for re-use purpose. In order to do this, a block design must be created in a project-based flow. You also need to make sure that the design doesn't flag any DRC violations and synthesizes (and possibly implements) without any issues. Once you are satisfied with the block design, you can delete everything except for the **bd** directory and all the sub-directories included beneath it from the Vivado Project. This way all the block design data including the data for all the IP contained within the block design can be imported into a different Vivado project.

---

## Importing an Existing Block Design into a Different Vivado IDE Project

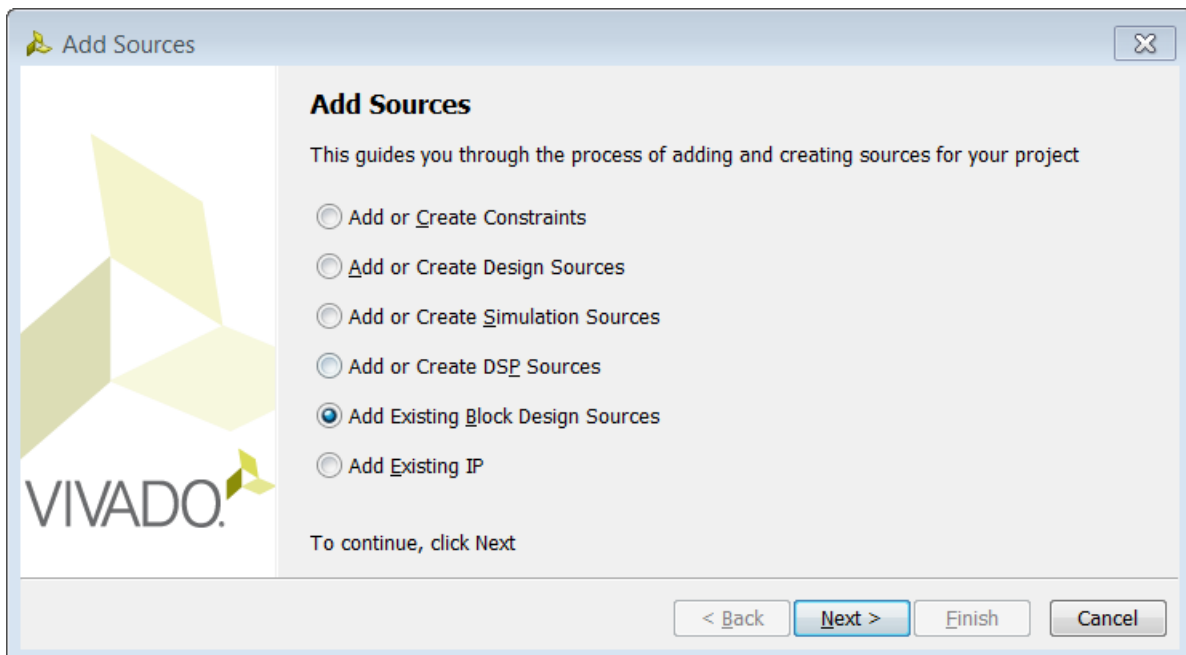
Assuming that a block design was created using a project-based flow, and all the directory structure including and within the bd folder is available, the block design can be opened in a different Vivado project. The only limitation in such a scenario is that the new project settings in which the existing block design is being imported must be the same as the original project in which the block design was created. If the target device of the projects (this includes devices even within the same family) are different then the IP will get locked and the design will have to be re-generated. In such a scenario there is a likelihood that the behavior of the design may not be the same as the original block design.

1. To import an existing block design, highlight Design Sources in the Sources window, right-click and select **Add Sources**.



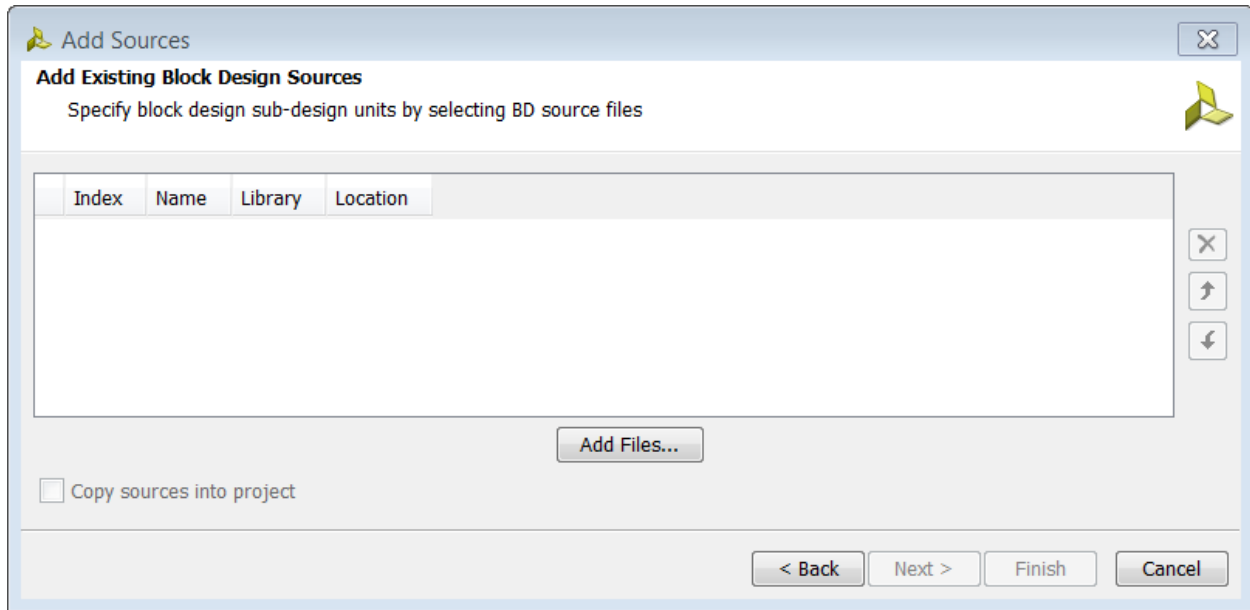
**Figure 172: Add Sources to a New Vivado Project**

2. The Add Sources dialog box appears. Select **Add Existing Block Design Sources**. Click **Next**.



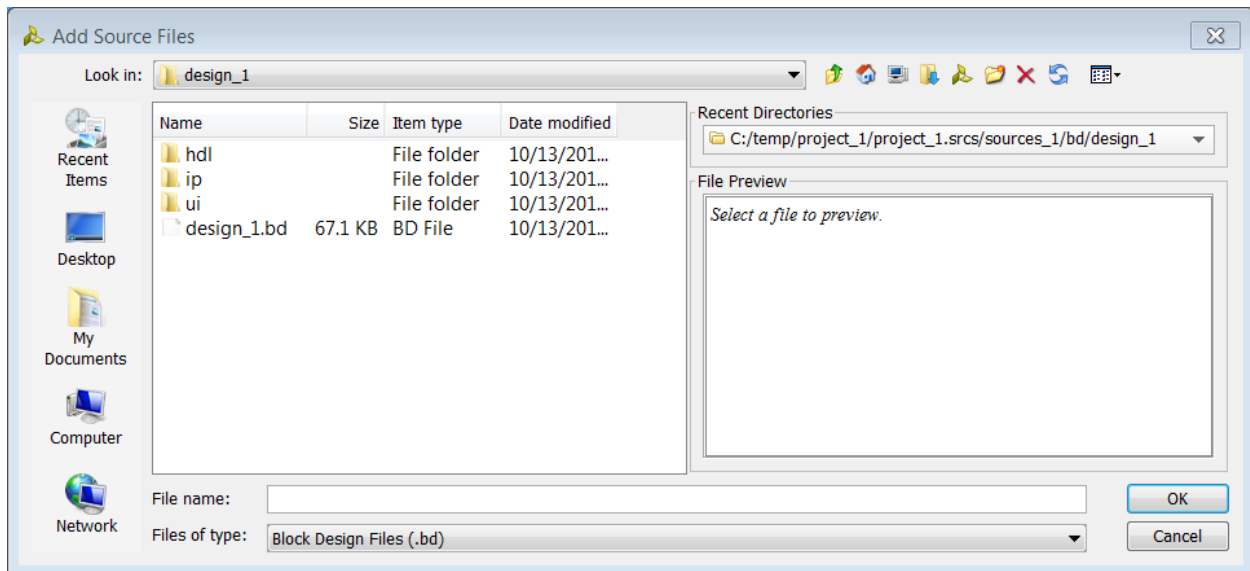
**Figure 173: Add Existing Block Design**

- In the Add Sources dialog box, click on **Add Files**.



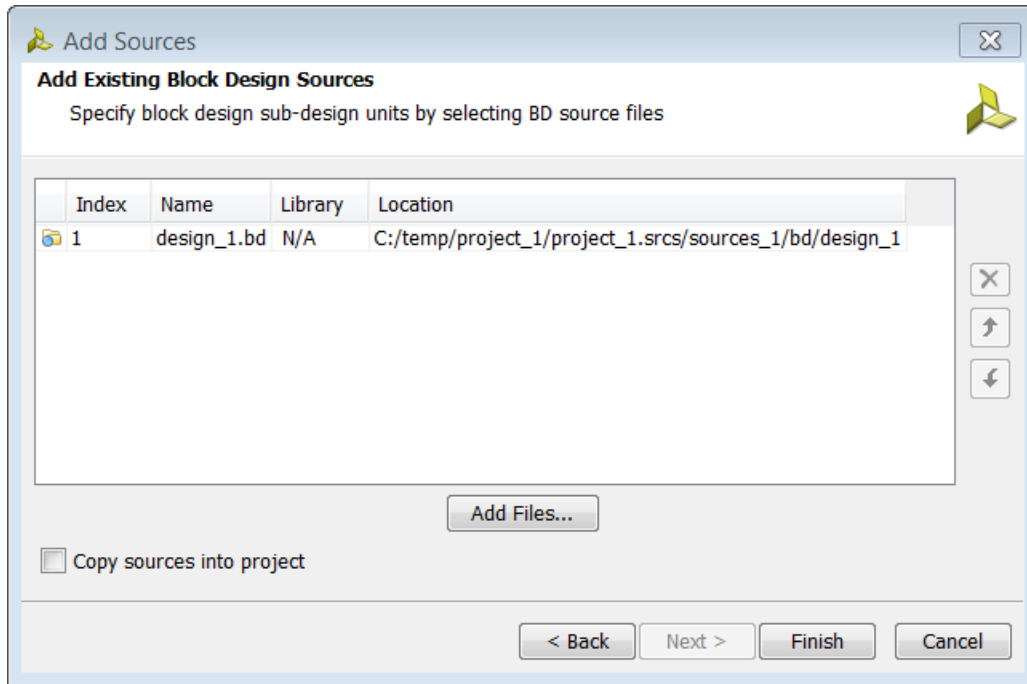
**Figure 174: Point to the Existing Block Design File by Clicking on Add Files**

- In the Add Source Files dialog box, browse to the **bd** folder where the block design is located, select the **.bd** file and click **OK**.



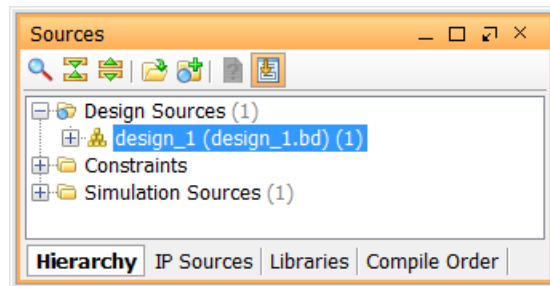
**Figure 175: Browse to the Folder Containing the Block Design**

- Click **Finish** once the existing block design has been added.



**Figure 176: Existing Block Design Sources Added to the Project**

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



**Figure 177: Imported Block Design in the Sources Window**

- Open the block design by double clicking on it.

You may need to update the IP used in the block design, or validate the block design, generate a wrapper and synthesize and implement the design. All of these topics are discussed in other sections of this document.

### Overview

Sometimes it may be necessary to use a third-party synthesis tool as a part of the design flow. In this case, you will need to incorporate the block design as a black-box to the top-level user design. You can then synthesize the rest of the design in a third-party synthesis tool, write out a HDL or EDIF netlist and implement the post-synthesis project in the Vivado environment.

This chapter describes the necessary 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 has been written for Synplify Pro synthesis tool.

### Creating a Design Check Point (DCP) File for a Block Design

A design check point can be created for a block design by setting the block design as an Out-of-Context module. To do this you can select the block design in the sources window, right-click and choose **Out-of-Context Settings**.

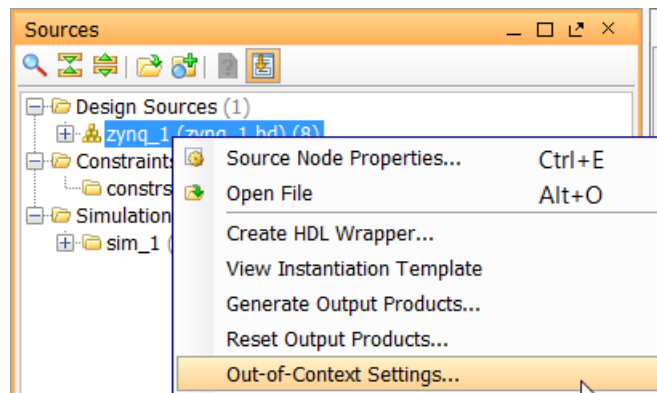
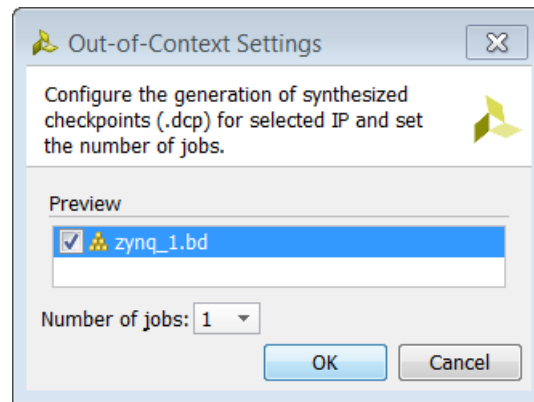


Figure 178: Set Block Design as an Out-of-Context Module

In the Out-of-Context Settings dialog box, click in the checkbox next to the block design that needs to be set as an out-of-context module.





**Figure 179: Out-of-Context Settings dialog box**

A square is placed against the block design in the Sources view to indicate that the block design has been set as an out-of-context module. The Design Runs window also shows an Out-of-Context module run for the block design.

Next, synthesize the design which will create a design-check-point file for the block design which can be found in the directory shown below.

```
<path_to_design>\<project_name>\<project_name>.runs\<block_design_name>_synth_1
```

Design checkpoints enable you to take a snapshot of your design in its current state. The current netlist, constraints, and implementation results are stored in the design checkpoint. Using design checkpoints, you can:

- Restore your design if needed
- Perform design analysis
- Define constraints
- Proceed with the design flow

---

## Create a Verilog or VHDL Stub File for the Block Design

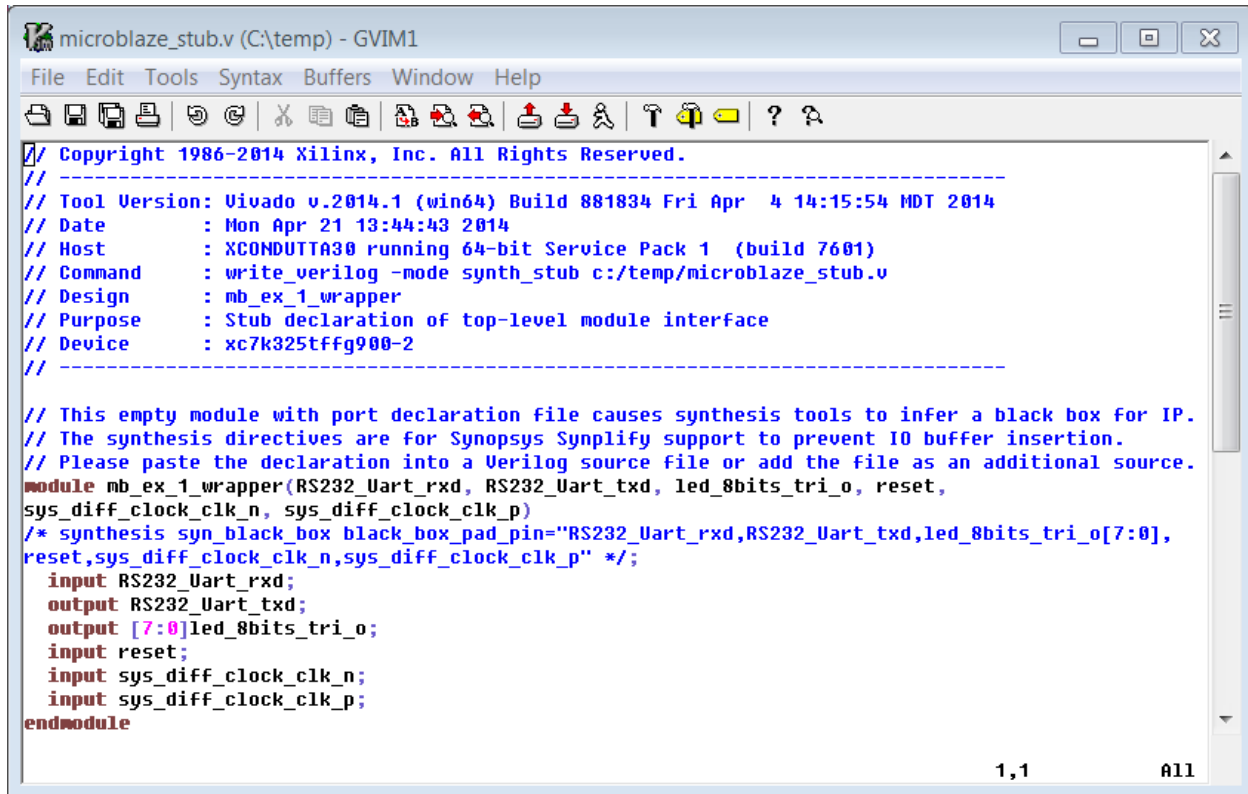
Once the check-point has been generated, you need to create a stub file which can be instantiated in the top-level HDL file to refer to the block design as a black-box. First open the synthesized design and then to create the HDL stub file using the following commands:

```
write_verilog -mode synth_stub <path_to_file>/<file_name>
write_vhdl -mode synth_stub <path_to_file>/<file_name>
```



**CAUTION!** The synthesized design must be open in order for the write\_verilog/write\_vhdl command to work.

An example stub file is shown in the following figure.



```

microblaze_stub.v (C:\temp) - GVIM1
File Edit Tools Syntax Buffers Window Help
[Icons]
// Copyright 1986-2014 Xilinx, Inc. All Rights Reserved.
//
// Tool Version: Uivado v.2014.1 (win64) Build 881834 Fri Apr  4 14:15:54 MDT 2014
// Date       : Mon Apr 21 13:44:43 2014
// Host       : XCONDUITA30 running 64-bit Service Pack 1 (build 7601)
// Command    : write_verilog -mode synth_stub c:/temp/microblaze_stub.v
// Design     : mb_ex_1_wrapper
// Purpose    : Stub declaration of top-level module interface
// Device     : xc7k325tffg900-2
//
// This empty module with port declaration file causes synthesis tools to infer a black box for IP.
// The synthesis directives are for Synopsys Synplify support to prevent IO buffer insertion.
// Please paste the declaration into a Verilog source file or add the file as an additional source.
module mb_ex_1_wrapper(RS232_Uart_rxd, RS232_Uart_txd, led_8bits_tri_o, reset,
sys_diff_clock_clk_n, sys_diff_clock_clk_p)
/* synthesis syn_black_box black_box_pad_pin="RS232_Uart_rxd,RS232_Uart_txd,led_8bits_tri_o[7:0],
reset,sys_diff_clock_clk_n,sys_diff_clock_clk_p" */;
  input RS232_Uart_rxd;
  output RS232_Uart_txd;
  output [7:0]led_8bits_tri_o;
  input reset;
  input sys_diff_clock_clk_n;
  input sys_diff_clock_clk_p;
endmodule
1,1 All

```

Figure 180: Example Stub File

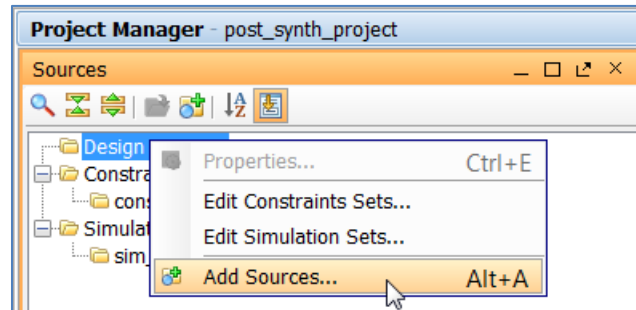
## Create a HDL or EDIF Netlist in the Synplify Project

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 will be treated as a black-box in Synplify. Once the project has been synthesized, an HDL or EDIF netlist for the project can be written out.

## Create a Post-Synthesis Project in Vivado and Implement

The next step is to create a post-synthesis project in the Vivado IDE. This can be done by selecting the **Post-synthesis Project** option in the Project type page while creating a New Project in Vivado.

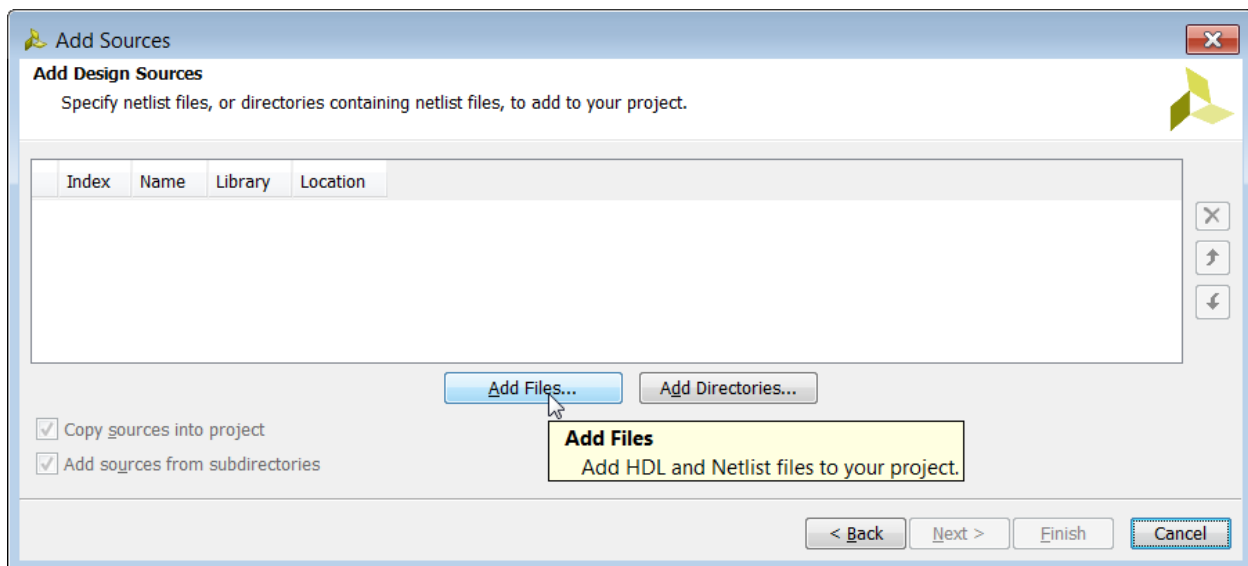
Once the project has been created, add the netlist file and the DCP file for the block design to the project by selecting and right-clicking on Design Sources and then choosing the **Add Sources** option.



**Figure 181: Add HDL Netlist from Synplify and the DCP File to the Project**

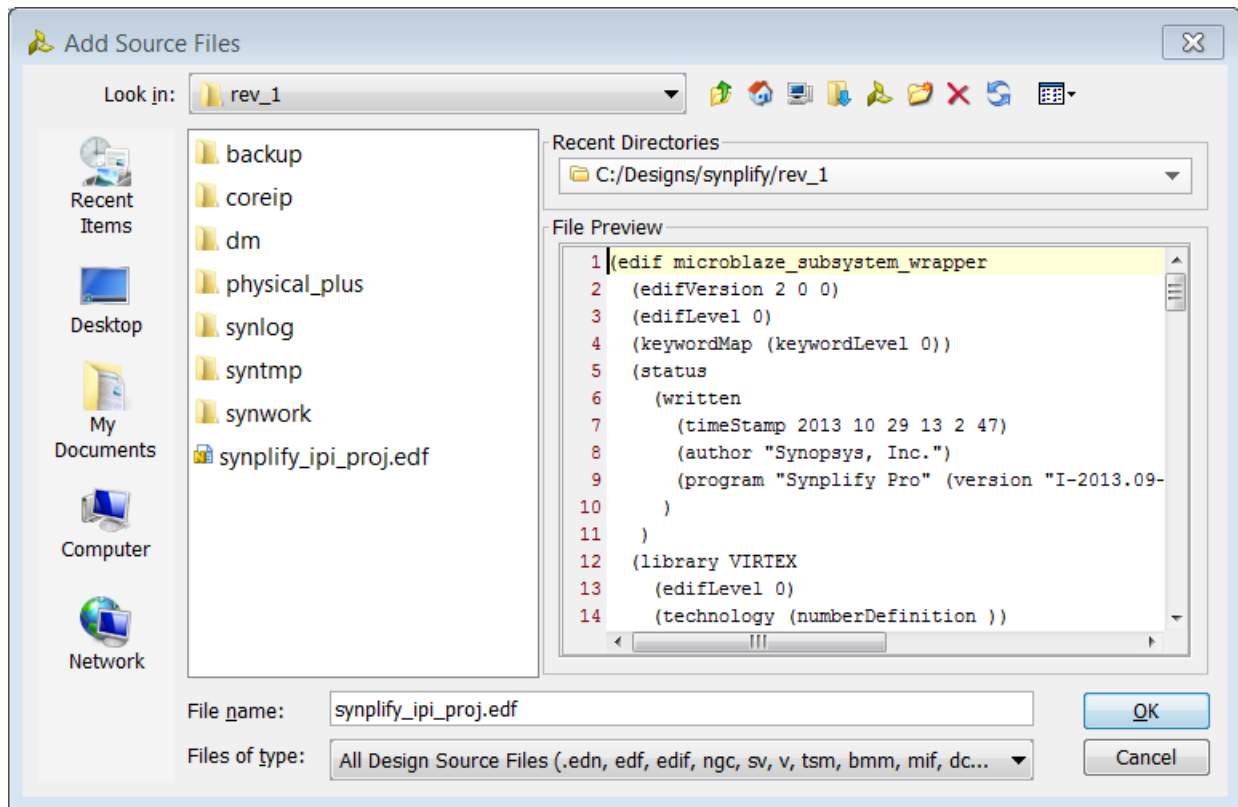
In the Add Sources dialog box **Add Design Sources** is selected by default. Click **Next**.

As shown below in the Add Design Sources page, click on **Add Files**.



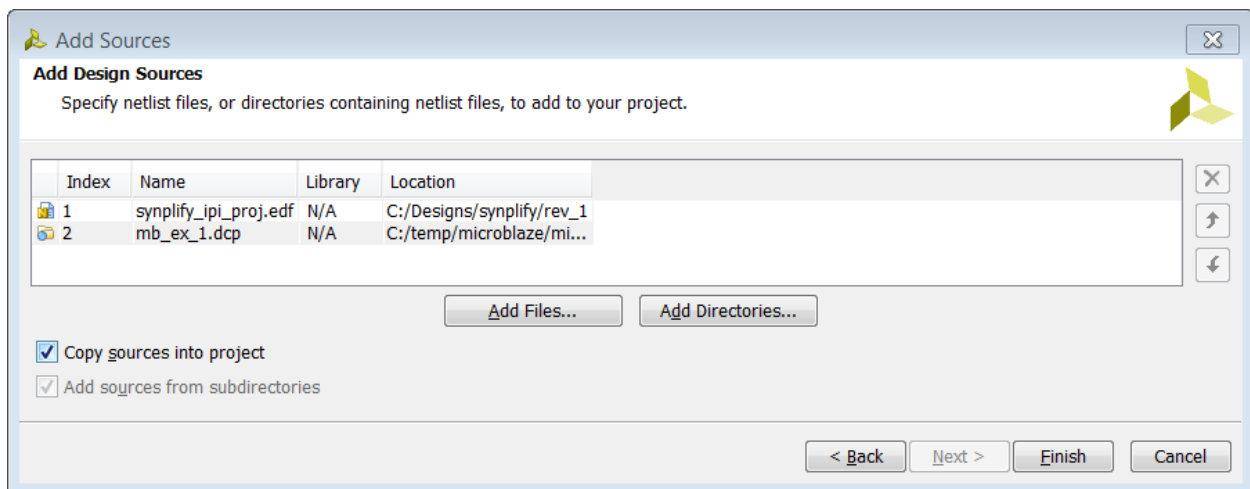
**Figure 182: Add Design Sources Page**

Select the netlist file by browsing to the right folder. Click **OK**.



**Figure 183: Browse to the Folder Containing the Netlist**

Repeat the above steps to add the DCP file as well and then click **Finish**.



**Figure 184: Add Netlist file and DCP File to the Project**

## Add Top-Level Constraints

Prior to implementing the design, you should add any necessary constraints to the project. The constraints file to the block design is contained with the DCP file. However, if you have changed the hierarchy of the block design, then the constraints file must be modified to make sure that hierarchical paths are properly scoped.

A constraints file can be added to the project just as the netlist and DCP file was added by right-clicking Design Sources in the Sources window, and choosing **Add Sources**. In the Add Sources dialog box select **Add or Create Constraints**.

## Add ELF File (if present)

If the block design has an ELF file associated to it, then you will need to add the ELF file to the Vivado project. In a post-synthesis project, adding an ELF file via the Vivado IDE GUI is not allowed. However, an ELF file can be added and associated to an embedded object using the following Tcl command:

```
add_files <path_to_elf_file>/<file_name>.elf
```

The added elf files can be seen in the Sources window.

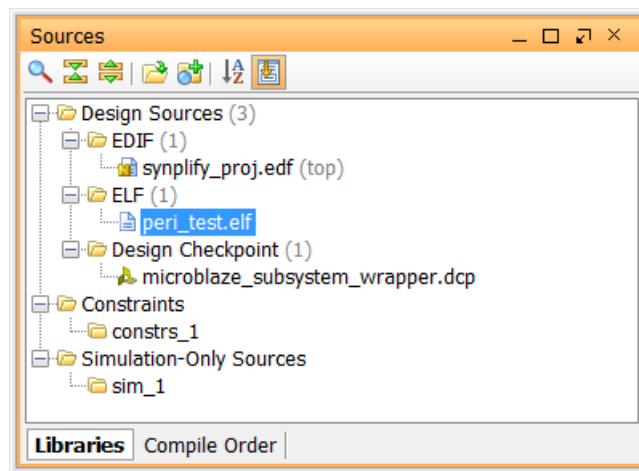


Figure 185: Check to make sure ELF file has been added to the project

Once the ELF file has been added to the project, it can be associated to an embedded object using the following command:

```
set_property SCOPED_TO_CELLS { <processor_instance> } \  
[get_files -all -of_objects [get_fileset sources_1] \  
<path_to_elf_file>/<file_name>.elf]
```

In the GUI you can do the same thing by selecting the ELF file in the Sources view and then editing the ADDR\_MAP\_CELLS field in the Source File Properties window.

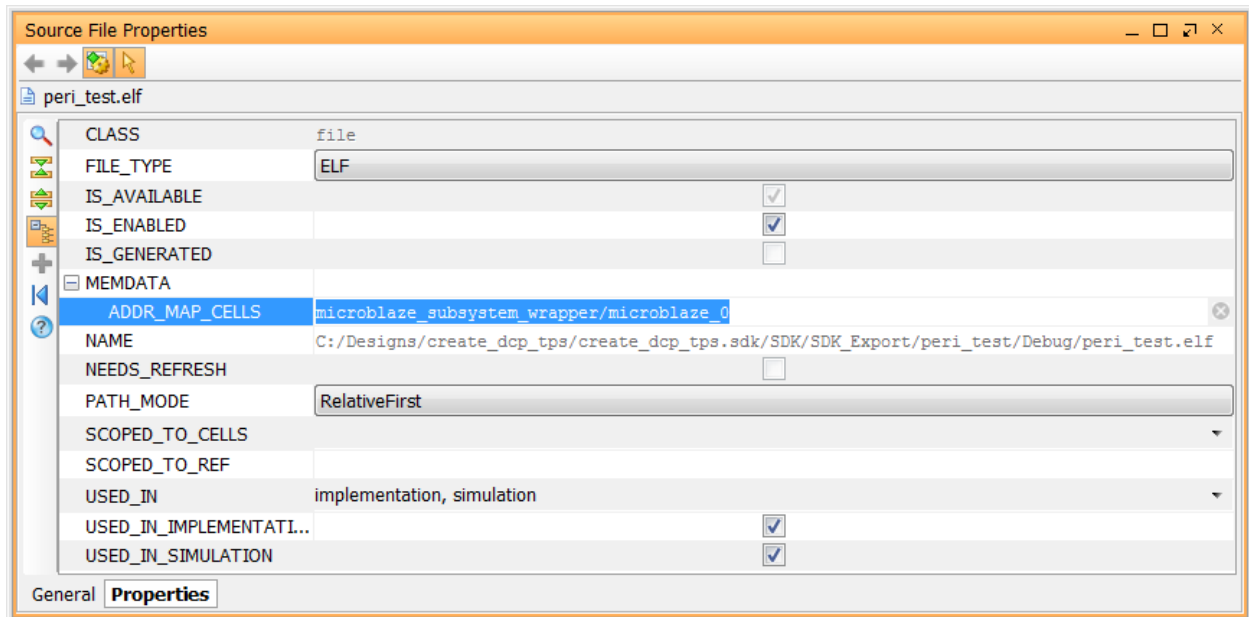


Figure 186: Specify the ADDR\_MAP\_CELLS field in the Source File Properties window

---

## Implement the Design

Next the design can be implemented and bitstream generated for the design.

### 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 interface. Several 7 series boards are currently supported, and support for boards with UltraScale™ parts is planned. Other boards, from third-party vendors such as Avnet, are also available now.

The IP integrator shows all the interfaces to the board in a separate tab called the Board tab. When you use this tab to select the desired interfaces and the Designer Assistance offered by IP integrator, you can easily connect your design to the board interfaces of your choosing. All the I/O constraints are automatically generated as a part of using this flow.

### 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 dialog box.

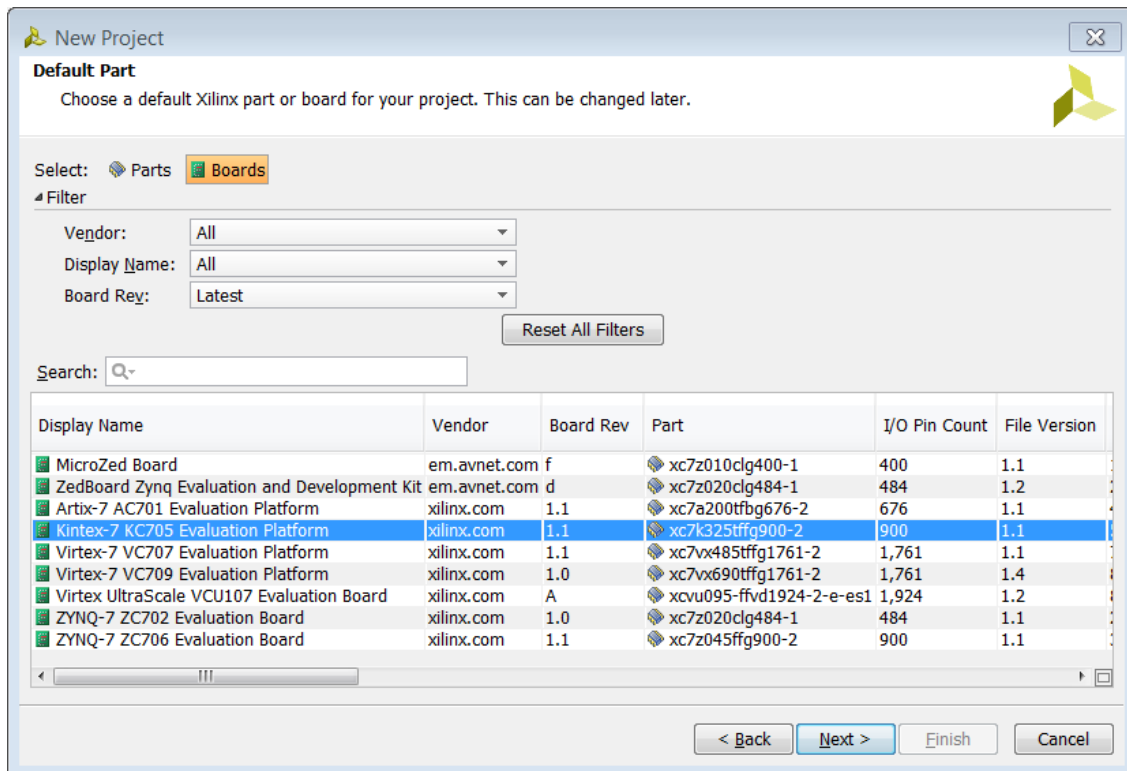
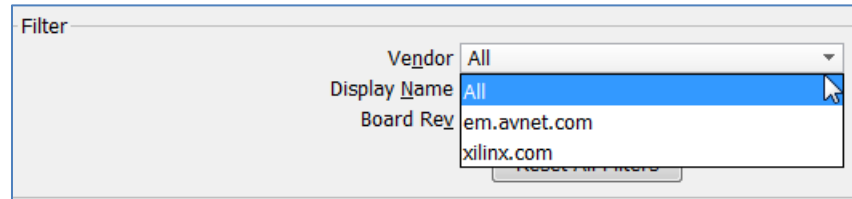


Figure 187: Select a target board

The list of available boards can be filtered based on Vendor, Display Name and Board Revision.



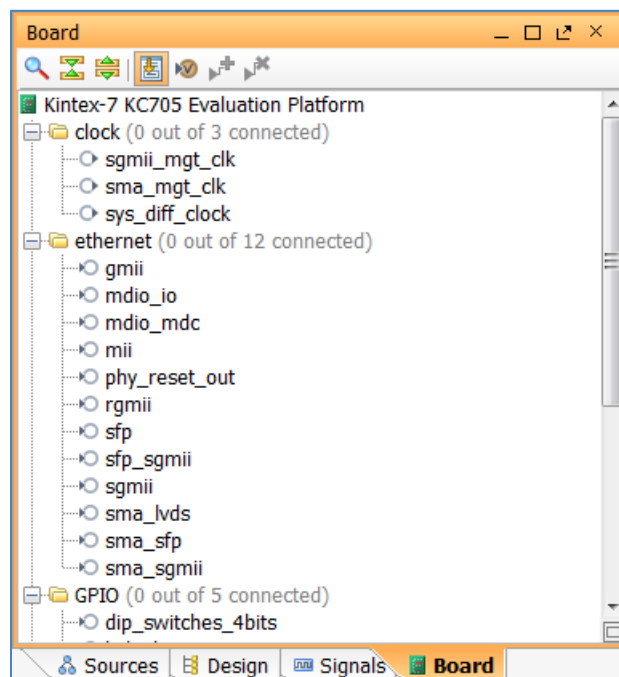
**Figure 188: Filter list of available boards**

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. Likewise, 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. Start a new block design by clicking on **Create Block Design** from the drop-down list IP integrator in the Flow Navigator. As the design canvas opens, you will notice a **Board** window, as shown below.

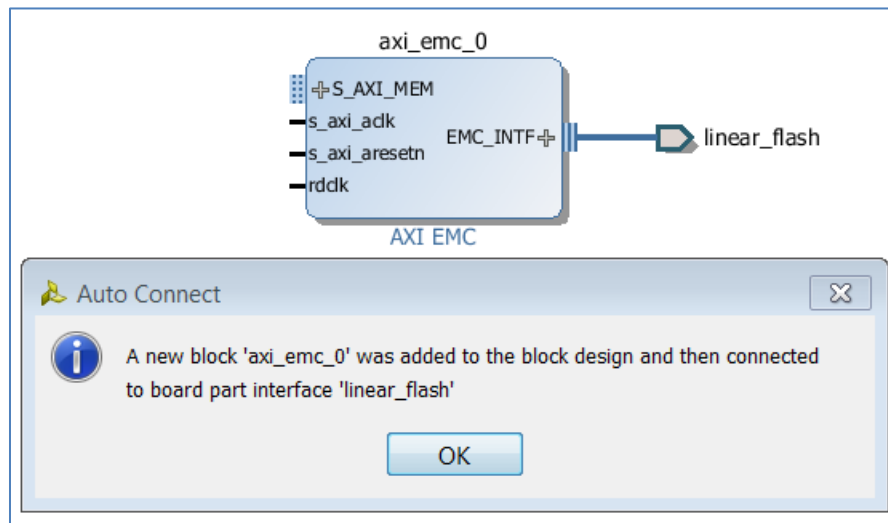


**Figure 189: Board window**



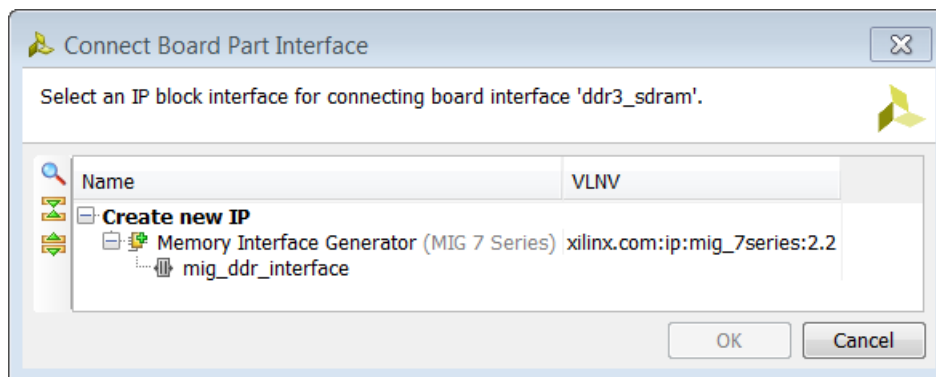
This Board window lists all the possible interfaces for an evaluation board (in the preceding figure the KC705 board is displayed). By selecting one of these interfaces, an IP can be quickly instantiated on the block design canvas.

The first way of using the Board interfaces, is to select an interface from the Board window and drag it on the block design canvas. This instantiates an IP that can connect to that interface and configures it appropriately for the interface in question. It then also connects the interface pin of the IP to an I/O port.



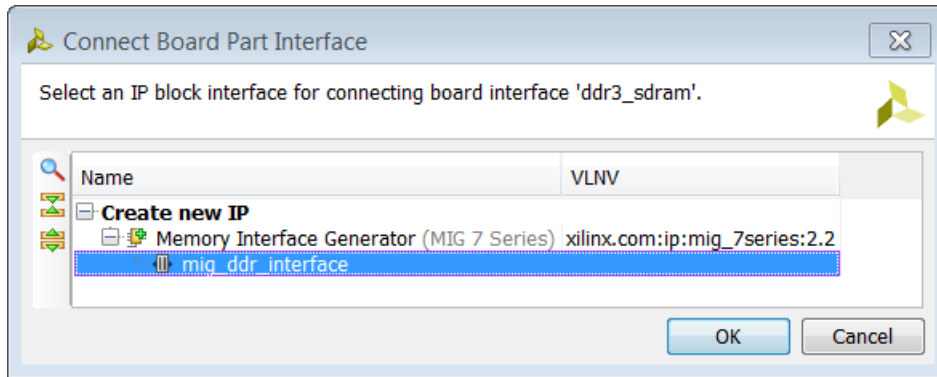
**Figure 190: Dragging and dropping an interface on the block design canvas**

The second way to use an interface on the target board, is to double-click the **ddr3\_sdr** interface from the Unconnected Interfaces folder. The Connect Board Part Interface dialog box opens.



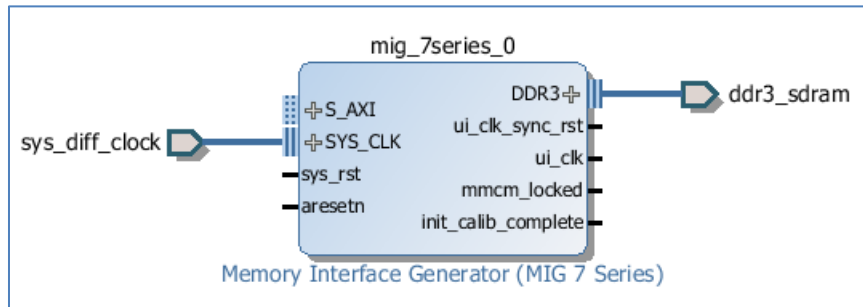
**Figure 191: Connect Board Part Interface Dialog Box**

Select the `mig_ddr_interface` and click **OK**.



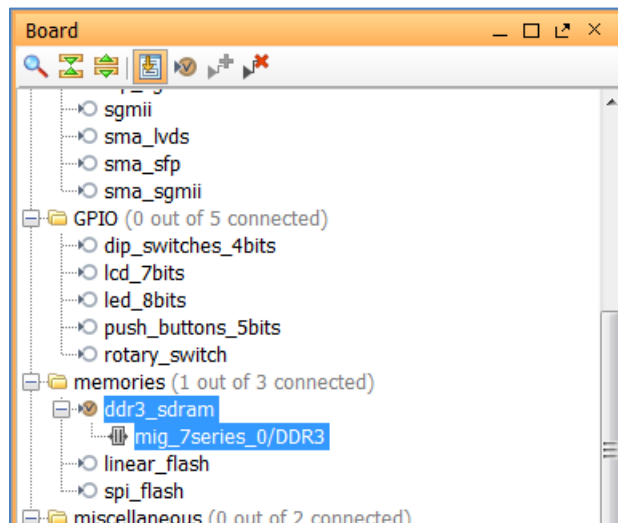
**Figure 192: IP Catalog Showing the List of IP that can be Connected to an Interface**

Notice that the IP is placed on the Diagram canvas and connections are made to the interface via I/O ports. The IP is all configured accordingly to connect to that interface.



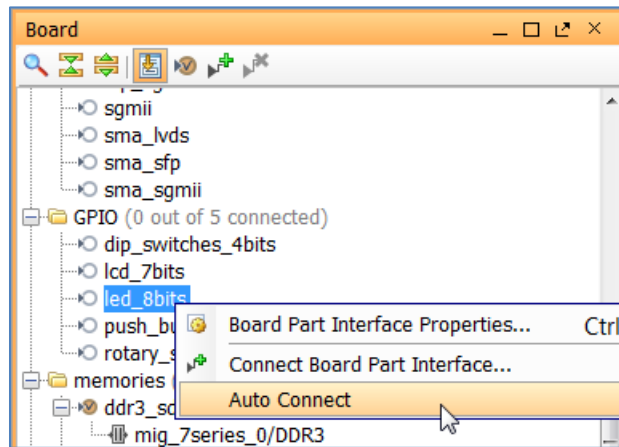
**Figure 193: IP instantiated, Configured and Connected to Interfaces on the Diagram Canvas**

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



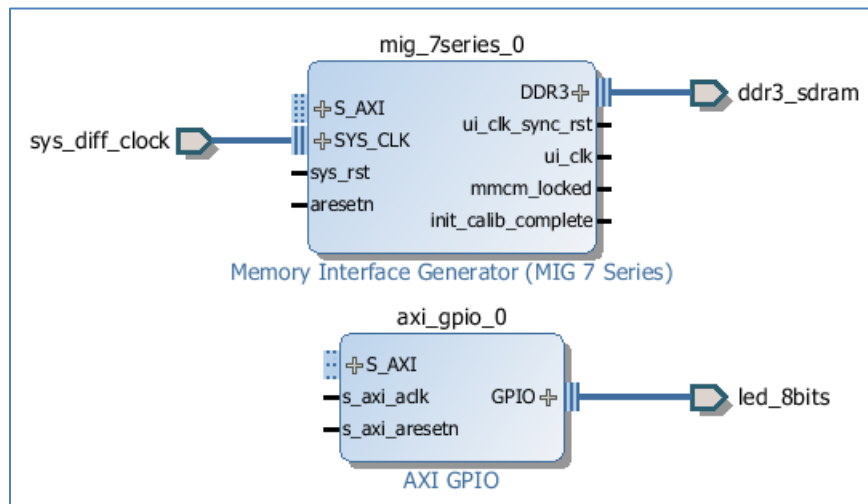
**Figure 194: Board Window after Connecting to an Interface**

An interface can also be connected using the Auto Connect option. To do this, select and right-click on the desired interface and from the menu select **Auto Connect**.



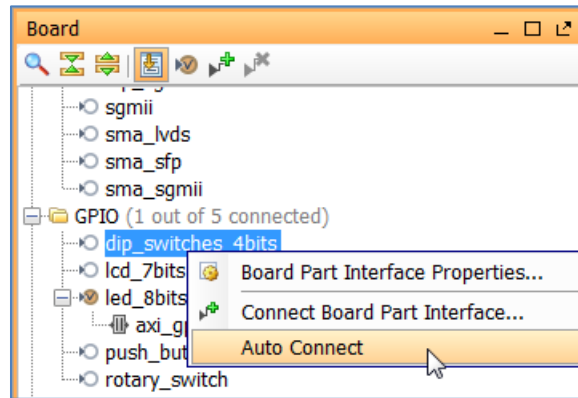
**Figure 195: Using the Preferred Connection Option to Connect to a Board Interface**

You will notice that the GPIO IP has been instantiated and the GPIO interface is connected to an I/O port.



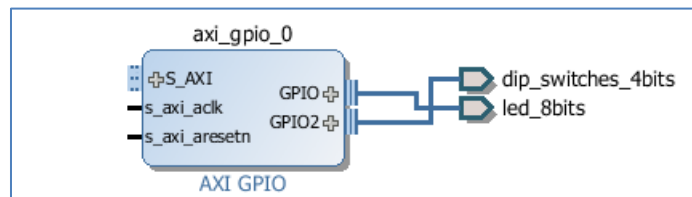
**Figure 196: Instantiating an IP using the Preferred Connection Option**

If another interface such as dip\_switches\_4bits is selected, then the board flow is smart enough to know that a GPIO already is instantiated in the design and it re-uses the second channel of the GPIO.



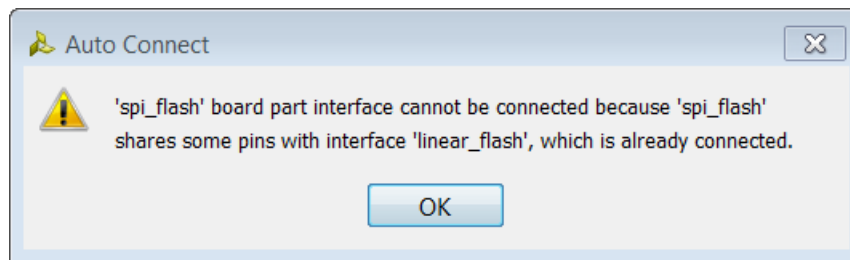
**Figure 197: Connecting an interface that can Share an Already Instantiated IP**

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



**Figure 198: GPIO IP Configured to Use the Second Channel**

If an external memory interface such as the linear\_flash or the spi\_flash is chosen, then as one of them is used the other interface 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 bd canvas.



**Figure 199: Auto-connect Warning**

## Complete Connections in the Block Design

Once the desired interfaces have been 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 MicroBlaze processor, as an example.

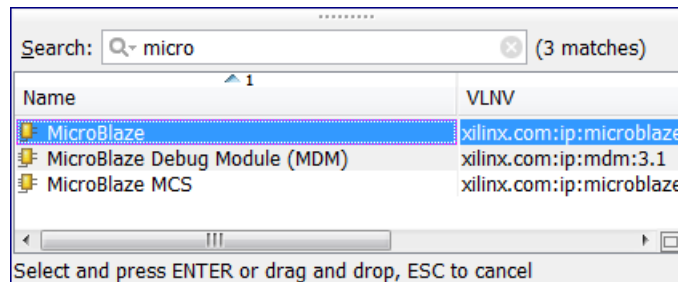


Figure 200: Instantiate a Processor to Complete the Design

As the processor is instantiated, Designer Assistance becomes available.

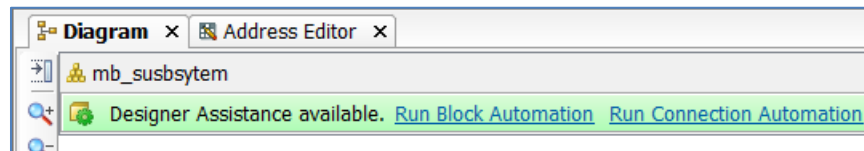
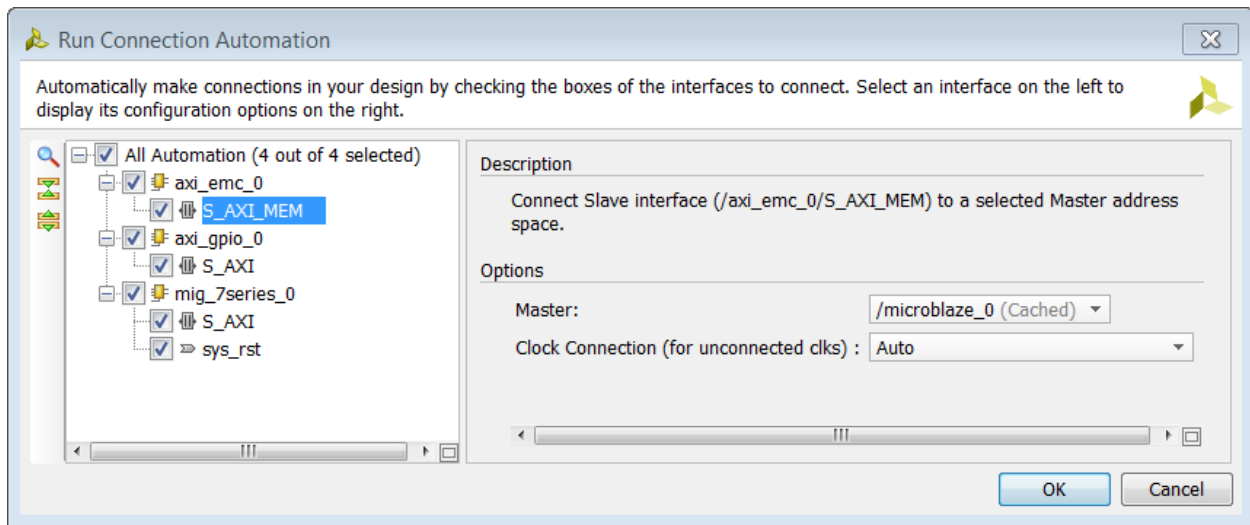


Figure 201: Use Designer Assistance to Complete Connections

Click on **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 202: Run Connection Automation to Complete Connections**

The rest of the process from here on after is the same as needed for designing in IP integrator as described in [Chapter 2 Creating a Block Design](#) of this document.

### Xilinx Resources

For support resources such as Answers, Documentation, Downloads, and Forums, see the Xilinx® Support website at:

[www.xilinx.com/support](http://www.xilinx.com/support)

For a glossary of technical terms used in Xilinx documentation, see:

[www.xilinx.com/company/terms.htm](http://www.xilinx.com/company/terms.htm)

---

### 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

#### Vivado® Design Suite Documentation

([www.xilinx.com/support/documentation/dt\\_vivado\\_vivado2014-1.htm](http://www.xilinx.com/support/documentation/dt_vivado_vivado2014-1.htm))

#### Vivado Design Suite User Guides

Vivado Design Suite User Guide: System-Level Design Entry ([UG895](#))

Vivado Design Suite User Guide: Design Flows Overview ([UG892](#))

Vivado Design Suite User Guide: Using the Vivado IDE ([UG893](#))

Vivado Design Suite User Guide: Using Tcl Scripting ([UG894](#))

Vivado Design Suite User Guide: Designing with IP ([UG896](#))

Vivado Design Suite User Guide: Model-Based DSP Design Using System Generator ([UG897](#))

Vivado Design Suite User Guide: Embedded Hardware Design ([UG898](#))

Vivado Design Suite User Guide: High-Level Synthesis ([UG902](#))

Vivado Design Suite User Guide: Using Constraints ([UG903](#))

Vivado Design Suite User Guide: Programming and Debugging ([UG908](#))

## Vivado Design Suite Tutorials

Vivado Design Suite Tutorial: Design Flows Overview ([UG888](#))

Vivado Design Suite Tutorial: Designing with IP ([UG939](#))

Vivado Design Suite Tutorial: Embedded Hardware Design ([UG940](#))

Vivado Design Suite Tutorial: Using Constraints ([UG945](#))

Vivado Design Suite Tutorial: Programming and Debugging ([UG936](#))

Other Vivado Design Suite Tutorials ([www.xilinx.com/training/vivado/index.htm](http://www.xilinx.com/training/vivado/index.htm))

## Other Vivado Design Suite Documents

Vivado Design Suite Tcl Command Reference Guide ([UG835](#))

AXI Reference Guide ([UG761](#))

Zynq-7000 All Programmable SoC PCB Design and Pin Planning Guide ([UG933](#))

Zynq-7000 All Programmable SoC Software Developers Guide ([UG821](#))

UltraFast Design Methodology Guide for the Vivado Design Suite ([UG949](#))

UltraFast Embedded Design Methodology Guide ([UG1046](#))

## Other Vivado Design Suite Documentation

([www.xilinx.com/support/documentation/dt\\_vivado\\_vivado2014-1.htm](http://www.xilinx.com/support/documentation/dt_vivado_vivado2014-1.htm))



---

### 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>.

© Copyright 2013-2014 Xilinx, Inc. Xilinx, the Xilinx logo, Artix, ISE, Kintex, Spartan, Virtex, Vivado, Zynq, and other designated brands included herein are trademarks of Xilinx in the United States and other countries. All other trademarks are the property of their respective owners.