Vivado Design Suite Tutorial

Programming and Debugging

UG936 (v2018.2) June 6, 2018

Revision History

The following table shows the revision history for this document.

Section	Revision Summary
06/06/2018 Version 2018.2	
General updates	Editorial updates only. No technical content updates.
04/27/2018 Version 2018.1	
General updates	General updates

Table of Contents

Introduction

This document contains a set of tutorials designed to help you debug complex FPGA designs. The first four labs explain different kinds of debug flows that you can chose to use during the course of debug. These labs introduce the Vivado® debug methodology recommended to debug your FPGA designs. The labs describe the steps involved in taking a small RTL design and the multiple ways of inserting the Integrated Logic Analyzer (ILA) core to help debug the design. The fifth lab is for debugging high-speed serial I/O links in Vivado. The sixth lab is for debugging JTAG-AXI transactions in Vivado. The first four labs converge at the same point when connected to a target hardware board.

Example RTL designs are used to illustrate overall integration flows between Vivado logic analyzer, ILA, and Vivado Integrated Design Environment (IDE). In order to be successful using this tutorial, you should have some basic knowledge of Vivado Design Suite tool flow.

TRAINING: Xilinx provides training courses that can help you learn more about the concepts presented in this document. Use these links to explore related courses:

- Vivado Design Suite Hands-on Introductory Workshop Training Course
- Vivado Design Suite Tool Flow Training Course
- Essentials of FPGA Design Training Course
- Designing FPGAs Using the Vivado Design Suite 1
- Designing FPGAs Using the Vivado Design Suite 2
- Designing FPGAs Using the Vivado Design Suite 3
- Designing FPGAs Using the Vivado Design Suite 4
- Vivado Design Suite User Guide: Programming and Debugging (UG908)

Objectives

These tutorials:

- Show you how to take advantage of integrated Vivado logic analyzer features in the Vivado design environment that make the debug process faster and simpler.

- Provide specifics on how to use the Vivado IDE and the Vivado logic analyzer to debug common problems in FPGA logic designs.
- Provide specifics on how to use the Vivado Serial I/O Analyzer to debug high-speed serial links.

After completing this tutorial, you will be able to:

- Validate and debug your design using the Vivado Integrated Design Environment (IDE) and the Integrated Logic Analyzer (ILA) core.
- Understand how to create an RTL project, probe your design, insert an ILA core, and implement the design in the Vivado IDE.
- Generate and customize an IP core netlist in the Vivado IDE.
- Debug the design using Vivado logic analyzer in real-time, and iterate the design using the Vivado IDE and a KC705 Evaluation Kit Base Board that incorporates a Kintex®-7 device.
- Analyze high-speed serial links using the Serial I/O Analyzer.

Getting Started

Setup Requirements

Before you start this tutorial, make sure you have and understand the hardware and software components needed to perform the labs included in this tutorial as listed below.

Software

- Vivado Design Suite 2018.1

Hardware

- Kintex-7 FPGA KC705 Evaluation Kit Base Board
- Digilent Cable
- Two SMA (Sub-miniature version A) cables

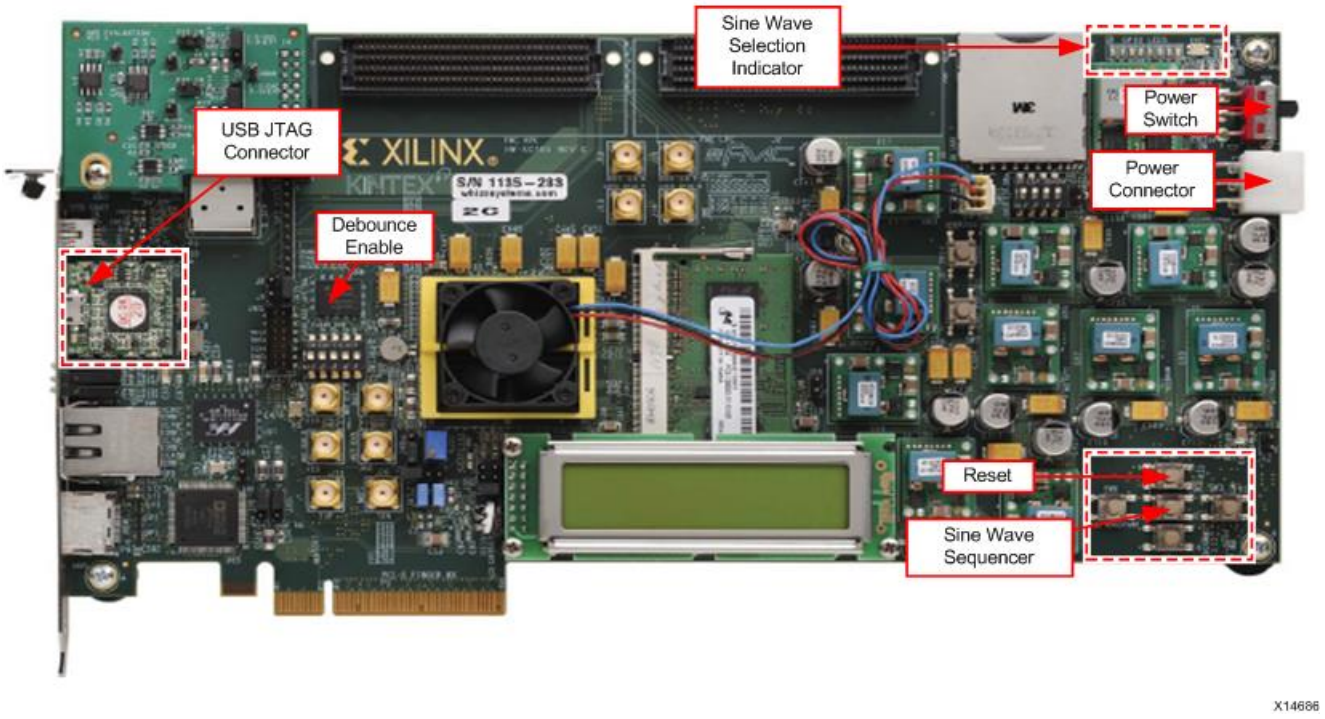

Figure 1: KC705 Board Showing Key Components

Tutorial Design Components

Labs 1 through 4 include:

- A simple control state machine
- Three sine wave generators using AXI-Streaming interface, native DDS Compiler
- Common push buttons (GPIO_BUTTON)
- DIP switches (GPIO_SWITCH)
- LED displays (GPIO_LED) VIO Core (Lab 3 only)

Push Button Switches: Serve as inputs to the de-bounce and control state machine circuits. Pushing a button generates a high-to-low transition pulse. Each generated output pulse is used as an input into the state machine.

DIP Switch: Enables or disables a de-bounce circuit.

De-bounce Circuit: In this example, when enabled, provides a clean pulse or transition from high to low. Eliminates a series of spikes or glitches when a button is pressed and released.

Sine Wave Sequencer State Machine: Captures and decodes input from the two push buttons. Provides sine wave selection and indicator circuits, sequencing among 00, 01, 10, and 11 (zero to three).

LED Displays: GPIO_LED_0 and GPIO_LED_1 display selection status from the state machine outputs, each of which represents a different sine wave frequency: high, medium, and low.

Lab 5 includes:

- An IBERT core
- A top-level wrapper that instantiates the IBERT core.

Board Support and Pinout Information

Table 1: Pinout Information for the KC705 Board

Pin Name	Pin Location	Description
CLK_N	AD11	Clock
CLK_P	AD12	Clock
GPIO_BUTTONS[0]	AA12	Reset
GPIO_BUTTONS[1]	AG5	Sine Wave Sequencer
GPIO_SWITCH	Y28	De-bounce Circuit Selector
LEDS_n[0]	AB8	Sine Wave Selection[0]
LEDS_n[1]	AA8	Sine Wave Selection[1]
LEDS_n[2]	AC9	Reserved
LEDS_n[3]	AB9	Reserved

Design Files

1. In your `C:` drive, create a folder called `/Vivado_Debug`.
2. Download the Reference Design Files from the Xilinx website.

CAUTION! *The tutorial and design files may be updated or modified between software releases. You can download the latest version of the material from the Xilinx website.*

3. Unzip the tutorial source file to the `/Vivado_Debug` folder. There are six labs that use different methodologies for debugging your design. Select the appropriate lab and follow the steps to complete them

Lab 1: This lab walks you through the steps of marking nets for debug in HDL as well as the post-synthesis netlist (Netlist Insertion Method). Following are the required files:

- `debounce.vhd`
- `fsm.vhd`
- `sinegen.vhd`
- `sinegen_demo.vhd`
- `sine_high/sine_high.xci`
- `sine_low/sine_low.xci`
- `sine_mid/sine_mid.xci`
- `sinegen_demo_kc705.xdc`

Lab 2: This lab goes over the details of marking nets for debug in the source HDL (HDL instantiation method) as well as instantiating an ILA core in the HDL. Following are the required files:

- `debounce.vhd`
- `fsm.vhd`
- `sinegen.vhd`
- `sinegen_demo_inst.vhd`
- `ila_0/ila_0.xci`
- `sine_high/sine_high.xci`
- `sine_low/sine_low.xci`
- `sine_mid/sine_mid.xci`
- `sinegen_demo_kc705.xdc`

Lab 3: You can test your design even if the hardware is not physically accessible, using a VIO core. This lab walks you through the steps of instantiating and customizing a VIO core that you will hook to the I/Os of the design. Following are the required files:

- `debounce.vhd`
- `fsm.vhd`
- `sinegen.vhd`
- `sinegen_demo_inst_vio.vhd`
- `sine_high/sine_high.xci`
- `sine_low/sine_low.xci`
- `sine_mid/sine_mid.xci`
- `ila_0/ila_0.xci`
- `sinegen_demo_kc705.xdc`

Lab 4: Nets can also be marked for debug in a third-party synthesis tool using directives for the synthesis tool. This lab walks you through the steps of marking nets for debug in the Synplify tool and then using Vivado to perform the rest of the debug. Following are the required files:

- `debounce.vhd`
- `fsm.vhd`
- `sign_high.dcp`
- `sign_low.dcp`
- `sine_mid.dcp`
- `sine_high.xci`
- `sine_low.xci`
- `sine_mid.xci`
- `sinegen.edn`
- `sinegen_synplify.vhd`
- `synplify_1.sdc`
- `synplify_1.fdc`
- `sinegen_demo_kc705.xdc`

Lab 5: Take designs created from Lab 1, Lab 2, Lab 3, and Lab 4 and load them onto the KC705 board.

Lab 6: Enhance post implementation debugging by using the ECO flow to replace debug probes.

Lab 7: Use the Incremental Compile flow to enable faster debugging flows. Using the results from a previous implementation run, this flow allows you to make debug modifications and rerun implementation.

Lab 8: Debug high-speed serial I/O links using the Vivado Serial I/O Analyzer. This lab uses the Vivado IP example design.

Lab 9: Use Vivado ILA core to debug JTAG-to-AXI transactions. This lab uses the Vivado IP example design.

Lab 10: Evaluate and Monitor IBERT UltraScale+™ GTR (IBERT GTR) transceivers in Zynq® UltraScale+ MPSoC™ devices. This lab takes you through the steps of configuring the GTR, generating the FSBL (First Stage Boot Loader file) and using the Vivado Serial Analyzer tool to debug the links.

Connecting the Boards and Cables

1. Connect the Digilent cable from the Digilent cable connector to a USB port on your computer.
2. Connect the two SMA cables (for lab 5 only) as follows:
   a. Connect one SMA cable from J19 (TXP) to J17 (RXP).
   b. Connect the other SMA cable from J20 (TXN) to J66 (RXN).

The relative locations of SMA cables on the board are shown in Figure 1: KC705 Board Showing Key Components.

Introduction

In this lab, you will mark signals for debug in the source HDL as well as the post synthesis netlist. Then you will create an ILA core and take the design through implementation. Finally, you will use Vivado® to connect to the KC705 target board and debug your design using Vivado Integrated Logic Analyzer.

Step 1: Creating a Project with the Vivado New Project Wizard

To create a project, use the New Project wizard to name the project, to add RTL source files and constraints, and to specify the target device.

1. Invoke the Vivado IDE.
2. In the **Getting Started** page, click **Create Project** to start the New Project wizard. Click **Next**.
3. In the **Project Name** page, name the new project **proj_netlist** and provide the project location (`C:/Vivado_Debug`). Ensure that **Create Project Subdirectory** is selected and click **Next**.
4. In the **Project Type** page, specify the type of project to create as **RTL Project**. Click **Next**.
5. In the **Add Sources** page:
   a. Set **Target Language** to **VHDL**.
   b. Click the green "**+**" sign, and then click **Add Files**.
   c. In the **Add Source Files** dialog box, navigate to the `/src/lab1` directory.
   d. Select all VHD source files, and click **OK**.
   e. Verify that the files are added, and **Copy Sources into project** is selected.
6. Click **Add**.
7. In the **Add Directories** dialog box, navigate to the `/src/lab1` directory.
8. Select sine_high, sine_low, and sine_mid directories and click **Select**.
9. Verify that the directories are added. Click **Next**.
10. In the **Add Constraints** dialog box, click the "**+**" sign, and then click **Add Files**.
11. Navigate to /src/lab1 directory and select `sinegen_demo_kc705.xdc`. Click **Next**.

12. In the **Default Part** dialog box, specify the **xc7k325tffg900-2** part for the KC705 platform. You can also select **Boards** and then select **Kintex-7 KC705 Evaluation Platform**. Click **Next**.
13. Review the **New Project Summary** page. Verify that the data appears as expected, per the steps above, and click **Finish**.

    Note: *It could take a moment for the project to initialize.*

Step 2: Synthesizing the Design

1. In the Project Manager, click **Settings** as shown in the following figure.

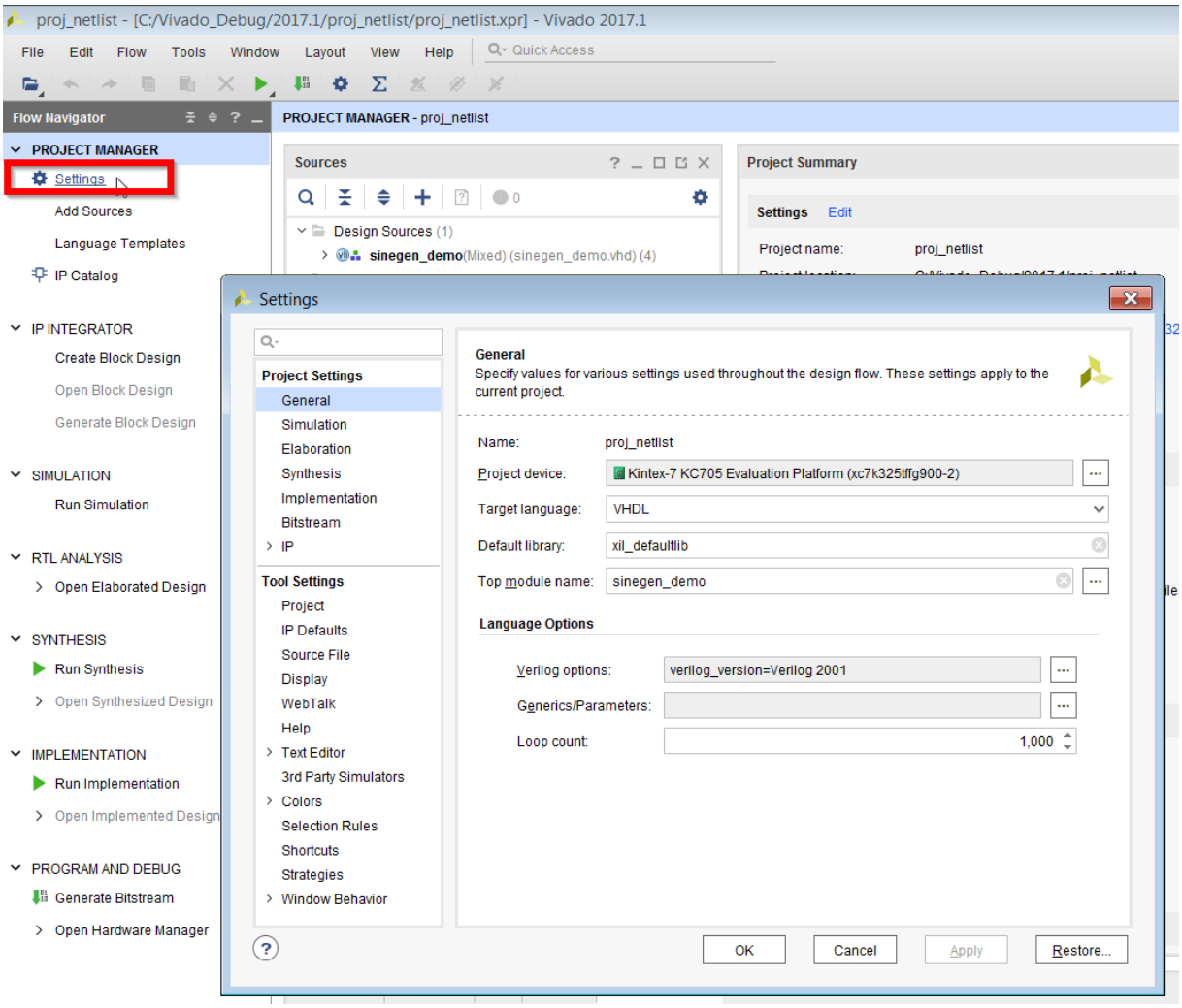

Figure 2: Configuring the Settings

IMPORTANT: *As an optional step, in the* ***Settings*** *dialog box, select Synthesis from the left and change* ***flatten hierarchy*** *to* ***none****. The reason for changing this setting to* ***none*** *is to prevent the synthesis tool from performing any boundary optimizations for this tutorial.*

2. In the Vivado **Flow Navigator**, expand the **Synthesis** drop-down list, and click **Run Synthesis**. In the **Launch Runs** dialog box, accept all of the default settings (Launch runs on local host), and click **OK**.

   Note: *When synthesis runs, a progress indicator appears, showing that synthesis is occurring. This could take a few minutes.*

3. In the **Synthesis Completed** dialog box, click **Cancel** as shown in the following figure. You will implement the design later.

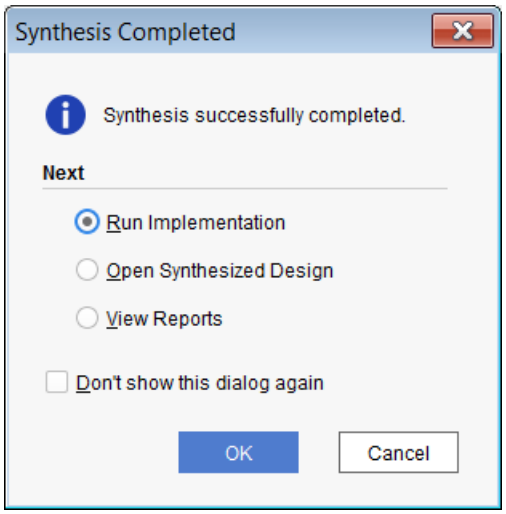

Figure 3: Synthesis Completed Dialog Box

Step 3: Probing and Adding Debug IP

To add a Vivado ILA core to the design, take advantage of the integrated flows between the Vivado IDE and Vivado logic analyzer.

In this step, you will accomplish the following tasks:

- Add debug nets to the project.
- Run the Set Up Debug wizard.
- Implement and open the design.
- Generate the bitstream.

Adding Debug Nets to the Project

Following are some ways to add debug nets using the Vivado IDE:

- Add MARK_DEBUG attribute to HDL files.

VHDL

```
attribute mark_debug : string;
attribute mark_debug of sine     : signal is "true";
attribute mark_debug of sineSel  : signal is "true";
```

Verilog

```
(* mark_debug = "true" *) wire sine;
(* mark_debug = "true" *) wire sineSel;
```

This method lets you probe signals at the HDL design level. This can prevent optimization that might otherwise occur to that signal. It also lets you pick up the signal tagged for post synthesis, so you can insert these signals into a debug core and observe the values on this signal during FPGA operation. This method gives you the highest probability of preserving HDL signal names after synthesis.

- Right-click and select **Mark Debug** or **Unmark Debug** on a synthesized netlist.

This method is flexible since it allows probing the synthesized netlist in the Vivado IDE and allows you to add/remove MARK_DEBUG attributes at any hierarchy in the design. In addition, this method does not require HDL source modification. However, there may be situations where synthesis may not preserve the signals due to netlist optimization involving absorption or merging of design structures.

- Use a Tcl prompt to set the MARK_DEBUG attribute on a synthesized netlist.

```
set_property mark_debug true [get_nets -hier [list {sine[*]}]]
```

This applies the MARK_DEBUG on the current, open netlist.

This method is flexible since you can turn MARK_DEBUG on and off by modifying the Tcl command. In addition, this method does not require HDL source modification. However, there may be situations where synthesis does not preserve the signals due to netlist optimization involving absorption or merging of design structures.

In the following steps, you learn how to add debug nets to HDL files and the synthesized design using Vivado IDE.

***TIP:** Before proceeding, make sure that the **Flow Navigator** on the left panel is enabled. Use **Ctrl-Q** to toggle it off and on.*

1. In the **Flow Navigator** under the **Synthesis** drop-down list, click **Open Synthesized Design** as shown in the following figure.

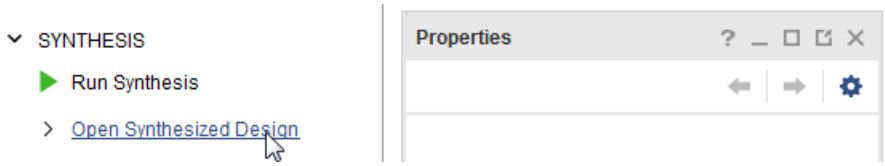

Figure 4: Open Synthesized Design

2. In the main toolbar drop-down menu, select **Debug**. When the **Debug** window opens. Click the window if it is not already selected.
3. Expand the **Unassigned Debug Nets** folder. The following figure shows those debug nets that were tagged with MARK_DEBUG attributes in `sinegen_demo.vhd`.

```
61
62    -- Add mark_debug attributes to show debug nets in the synthesized netlist
63    attribute mark_debug : string;
64    attribute mark_debug of GPIO_BUTTONS_db : signal is "true";
65    attribute mark_debug of GPIO_BUTTONS_dly : signal is "true";
66    attribute mark_debug of GPIO_BUTTONS_re : signal is "true";
67    attribute mark_debug of DONT_EAT : signal is "true";
68
69
70    component sinegen
71      port
72      (
73        clk   : in    std_logic;
74        reset : in    std_logic;
75        sel   : in    std_logic_vector(1 downto 0);
76        sine  : out   std_logic_vector(19 downto 0)
77      );
78
79    end component;
80
```

Figure 5: VHDL Example Using MARK_DEBUG Attributes

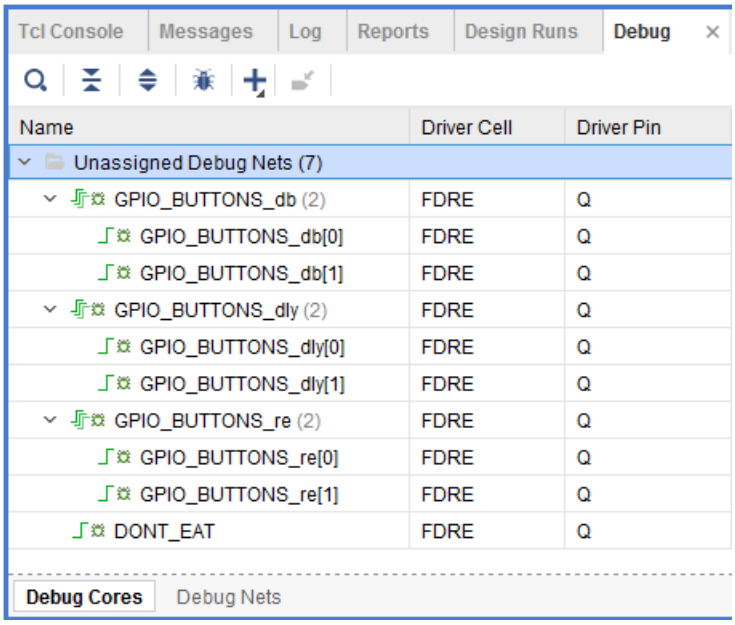

Figure 6: Unassigned Debug Nets Post-Synthesis

4. In the **Netlist** window, elect the **Netlist** tab and expand **Nets**. Select the following nets for debugging as shown in the following figure.
   - `GPIO_BUTTONS_IBUF[0] and GPIO_BUTTONS_IBUF[1]` - Nets folder under the top-level hierarchy
   - `sel(2)` - Nets folder under the U_SINEGEN hierarchy
   - `sine(20)` - Nets folder under the U_SINEGEN hierarchy

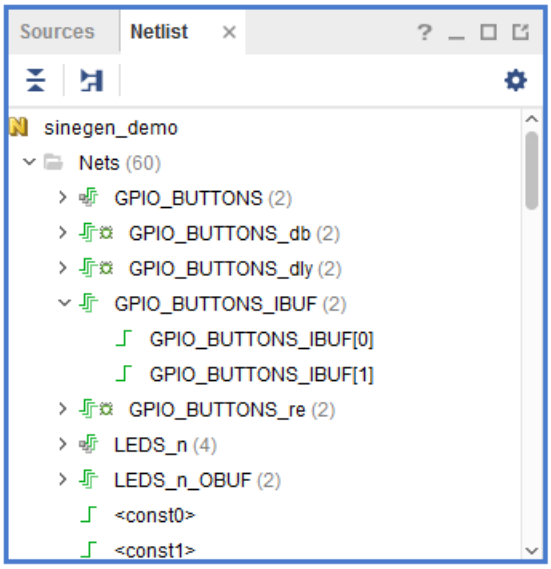

Figure 7: Add Nets for Debug from the Synthesized Netlist

***Note**: These signals represent the significant behavior of this design and are used to verify and debug the design in subsequent steps.*

5. Right-click the selected nets and select **Mark Debug** as shown in the following figure.

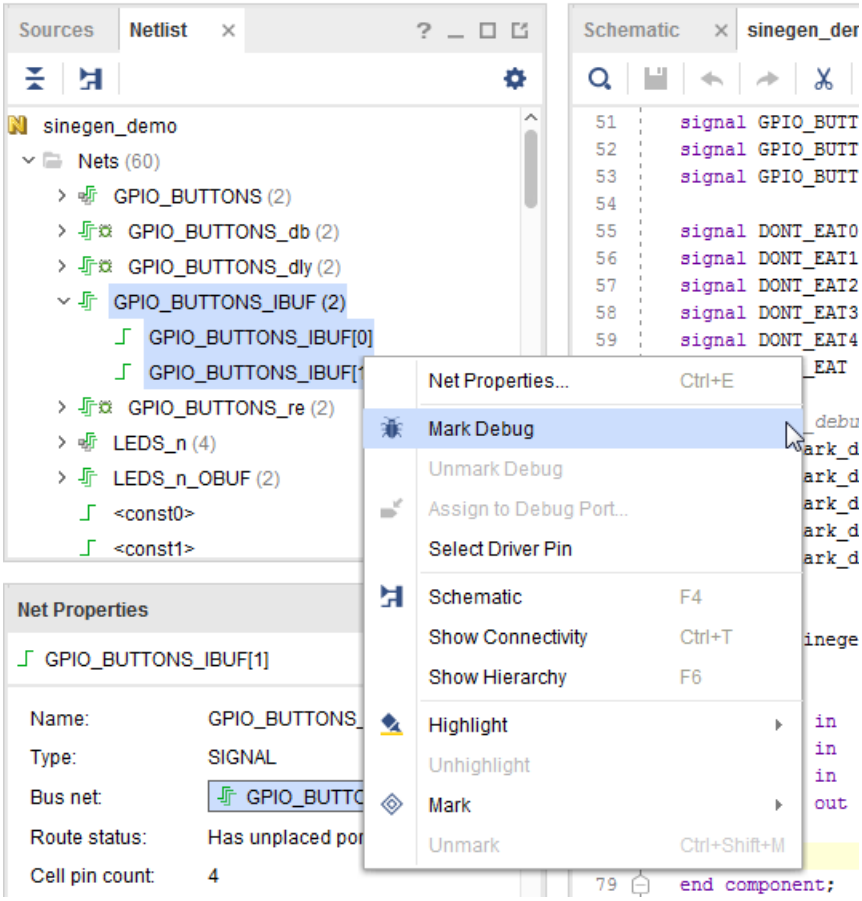

Figure 8: Adding Nets from the Netlist Tab

6. Next, mark nets for debug in the Tcl console. Mark nets "`sine(20)`" under the U_SINEGEN hierarchy for debug by executing the following Tcl command.

```
set_property mark_debug true [get_nets -hier [list {sine[*]}]]
```

***TIP:** In the **Debug** window, you can see the unassigned nets you just selected. In the **Netlist** window, you can also see the green bug icon next to each scalar or bus, which indicates that a net has the attribute mark_`debug = true` as shown the following two figures.*

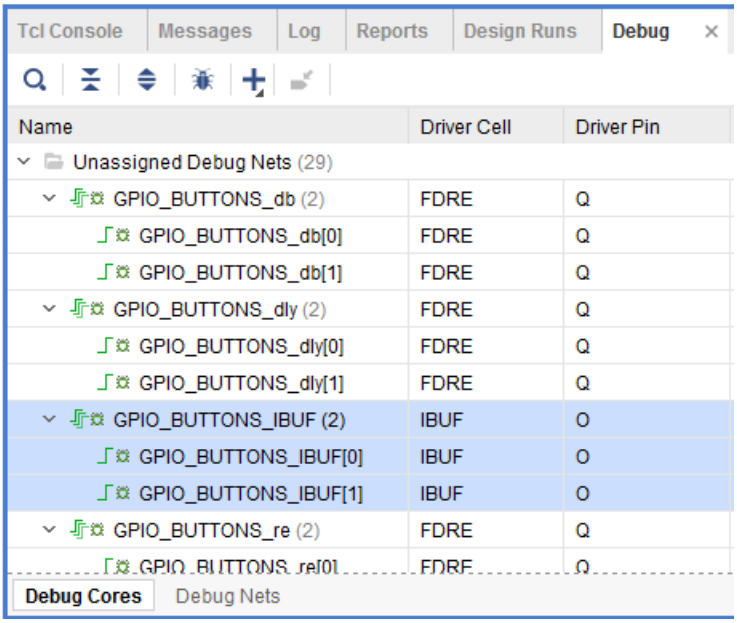

Figure 9: Newly Added Nets for Debug from the Synthesized Netlist

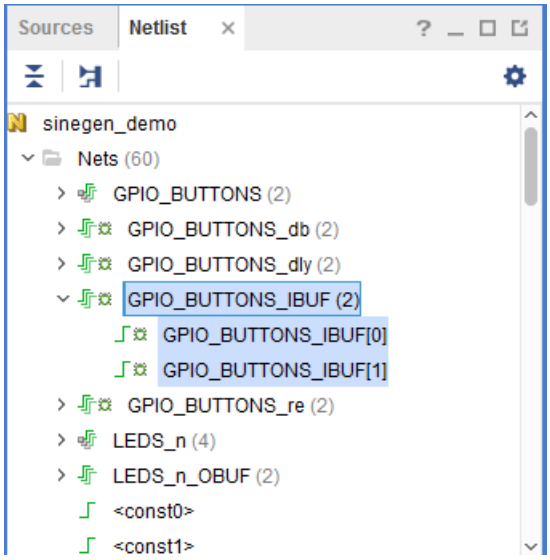

Figure 10: Netlist View of Nets Marked for Debug

Running the Set Up Debug Wizard

1. From the **Debug** window tool bar or **Tools** drop-down menu, select **Set Up Debug**. The **Set up Debug** wizard opens.

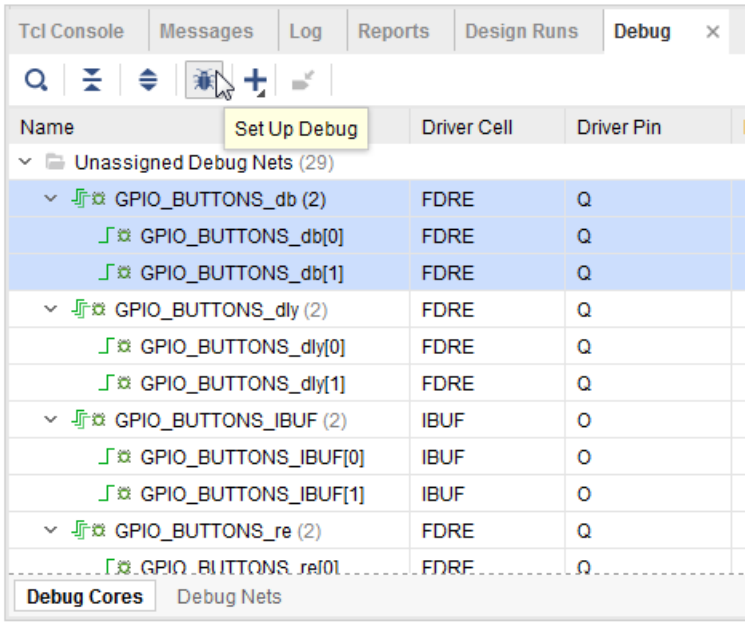

Figure 11: Launching the Set up Debug Wizard

2. When the **Set up Debug** wizard opens, click **Next.**

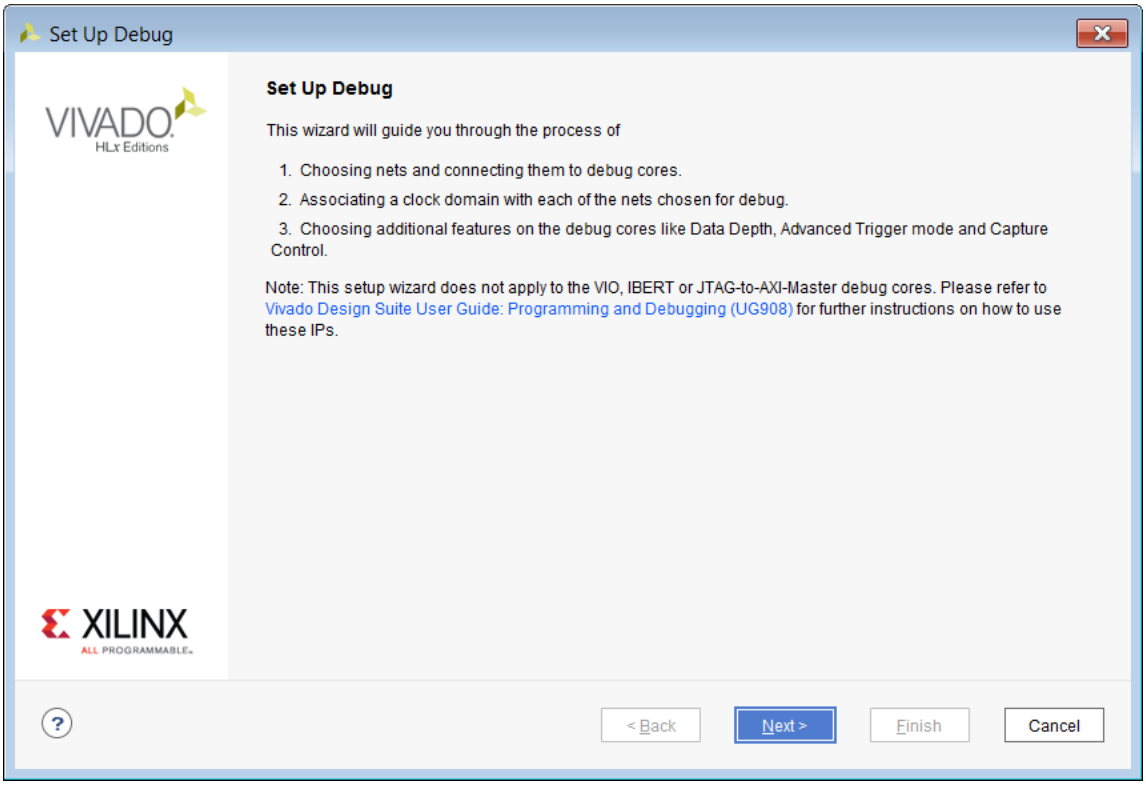

Figure 12: Set up Debug Wizard

3. In the **Nets to Debug** page, shown in the following figure, ensure that all the nets have been added for debug and click **Next**.

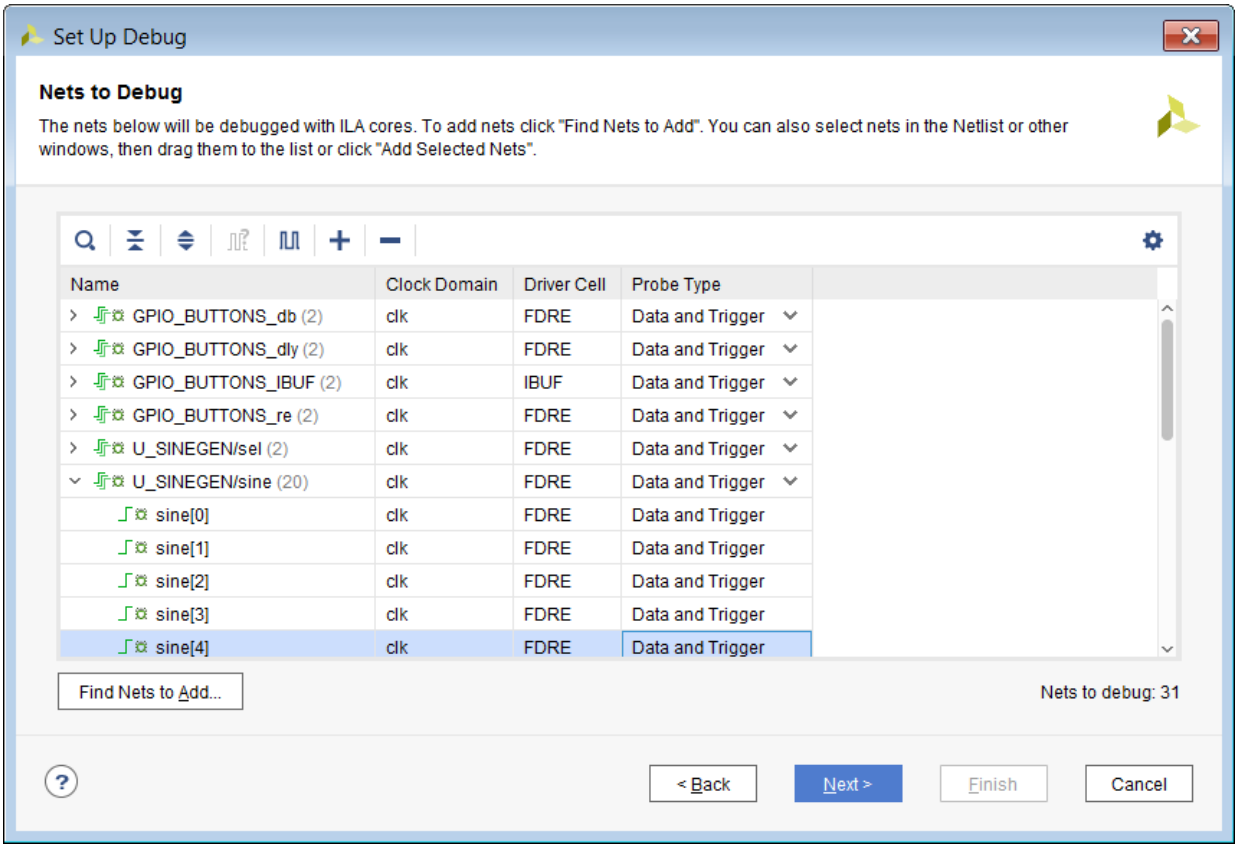

Figure 13: Specify Nets to Debug

4. In the **ILA Core Options** page, go to **Trigger and Storage Settings** section and select both **Capture Control** and **Advanced Trigger**. Click **Next**.

5. In the **Setup Debug Summary** page, make sure that all the information is correct and as expected. Click **Finish**.

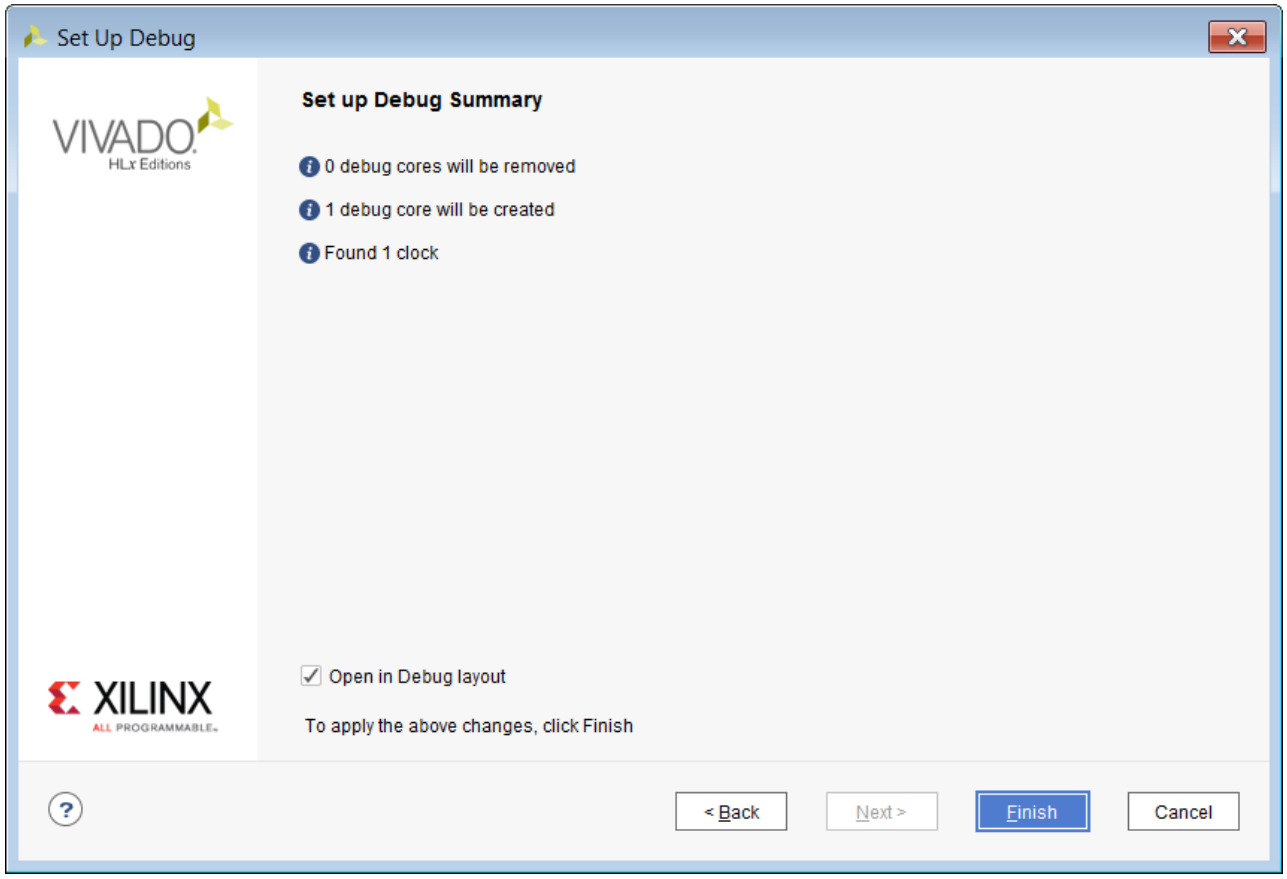

Figure 14: Set up Debug Summary

Upon clicking **Finish**, the relevant XDC commands that insert the ILA core(s) are generated.

Step 4: Implementing and Generating Bitstream.

1. In the **Flow Navigator**, under **Program and Debug**, click **Generate Bitstream**.

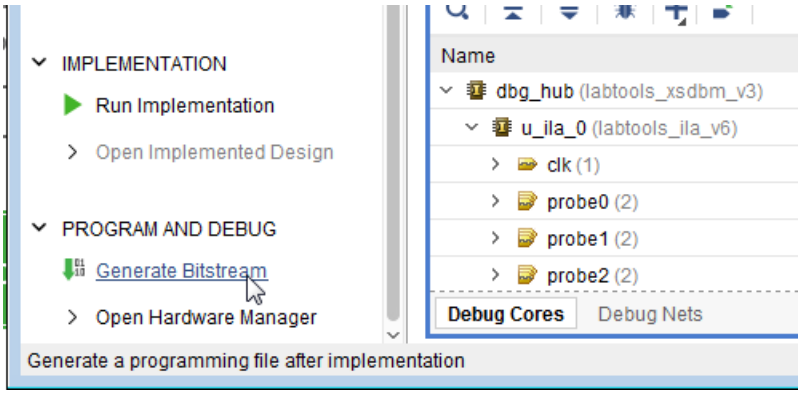

Figure 15: Implement Design and Generate Bitstream

2. In the **Save Project** dialog box click **Save**. This applies the MARK_DEBUG attributes on the newly marked nets. You can see those constraints by inspecting `the sinegen_demo_kc705.xdc` file.
3. When the **No Implementation Results Available** dialog box pops up, click **Yes**. In the **Launch Runs** dialog box, accept all of the default settings (Launch runs on local host) and click **OK**.
4. When the bitstream generation completes, the **Bitstream Generation Completed** dialog box pops up. Click **OK**.
5. In the dialog box asking to close synthesized design before opening implemented design. Click **Yes**.
6. Examine the **Timing Summary** report to ensure that all the specified timing constraints are met.

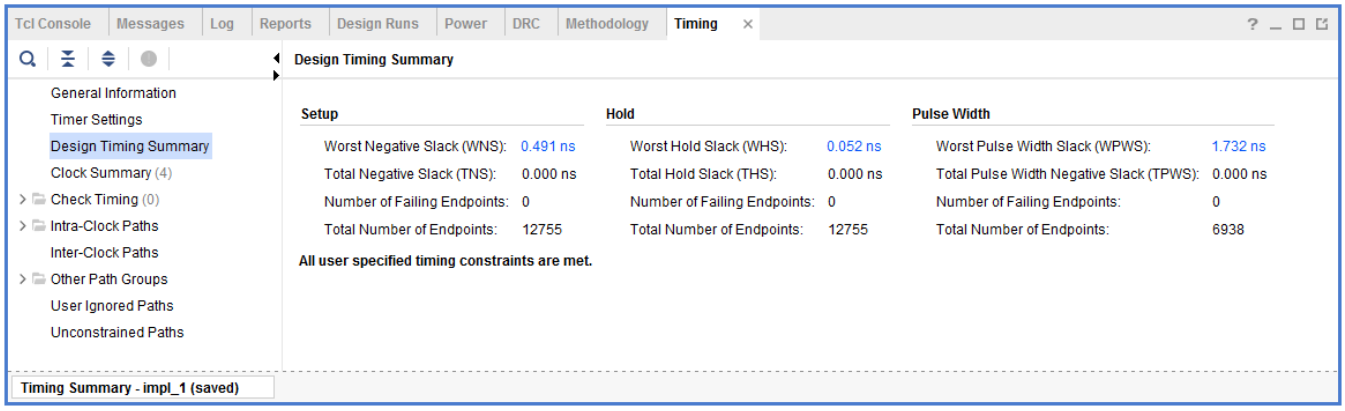

Figure 16: View the Timing Summary Report

Proceed to Lab 5: Using Vivado Logic Analyzer to Debug Hardware to complete the rest of the steps for debugging the design.

Introduction

The HDL Instantiation method is one of the two methods supported in Vivado® Debug Probing. For this flow, you will generate an ILA IP using the Vivado IP Catalog and instantiate the core in a design manually as you would with any other IP.

Step 1: Creating a Project with the Vivado New Project Wizard

To create a project, use the New Project wizard to name the project, to add RTL source files and constraints, and to specify the target device.

1. Invoke the Vivado IDE.
2. In the **Quick Start** tab, click **Create Project** to start the New Project wizard. Click **Next**.
3. In the **Project Name** page, name the new project **proj_hdl** and provide the project location (`C:/Vivado_Debug`). Ensure that **Create project subdirectory** is selected. Click **Next**.
4. In the **Project Type** page, specify the **Type of Project** to create as **RTL Project**. Click **Next**.
5. In the **Add Sources** page:
   a. Set **Target Language** to **VHDL**.
   b. Click the "+" sign, and then click **Add Files**.
   c. In the **Add Source Files** dialog box, navigate to the `/src/lab2` directory, and choose the `sine_high`, `sine_low`, `sine_mid`, and `ila_0` directories. Click **Select**.
   d. Select all VHD source files, and click **OK**.
   e. Verify that the files are added, and **Copy Sources into Project** is selected.
6. Click the "+" sign, and then click **Add Files**.
7. Navigate to the `/src/lab2/sine_high` directory.
8. Verify that the directories are added, and **Copy Sources into Project** is selected. Click **Next**.
9. In the **Add Constraints** dialog box, click the green "**+**" sign, and then click **Add Files**.
10. Navigate to `/src/lab1` directory and select `sinegen_demo_kc705.xdc`. Click **Next**.

11. In the **Default Part** page, specify the **xc7k325tffg900-2** part for the KC705 platform. You can also select **Boards** and then select **Kintex-7 KC705 Evaluation Platform**. Click **Next**.
12. Review the **New Project Summary** page. Verify that the data appears as expected, per the steps above. Click **Finish**.
13. In the **Sources** window in Vivado IDE, expand **sinegen_demo_inst** to see the source files for this lab. Note that **ila_0** core has been added to the project.

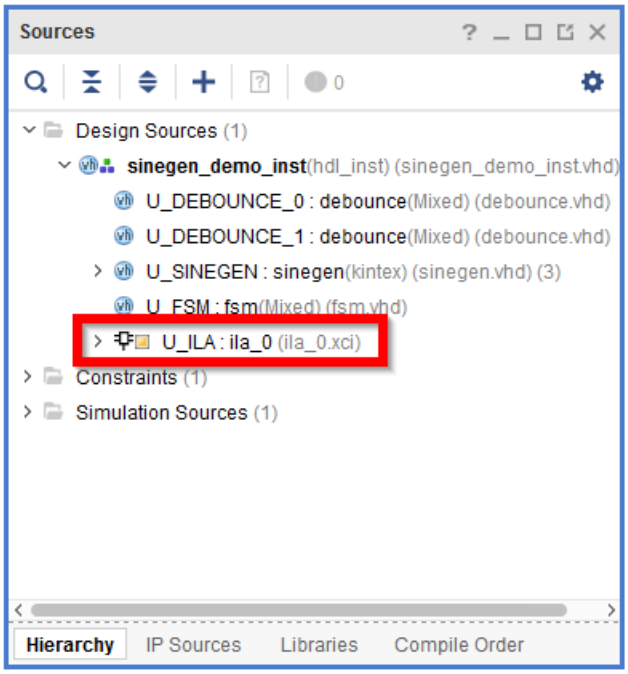

Figure 17: ILA Instantiation in HDL

14. Double-click the **sinegen_demo_inst.vhd** file, shown in the following figure to open it and inspect the instantiation and port mapping of the ILA core in the HDL code.

```
------------------------------------------------------------
-- ILA
------------------------------------------------------------
U_ILA : ila_0
  port map
  (
    CLK => clk,
    PROBE0 => sineSel,
    PROBE1 => sine,
    PROBE2 => GPIO_BUTTONS_db,
    PROBE3 => GPIO_BUTTONS_re,
    PROBE4 => GPIO_BUTTONS_dly,
    PROBE5 => GPIO_BUTTONS
  );
```

Figure 18: Hook Signals that Require Debugging in the ILA

Step 2: Synthesize Implement and Generate Bitstream

1. From the **Program and Debug** drop-down list, in **Flow Navigator**, click **Generate Bitstream**. This will synthesize, implement and generate a bitstream for the design.

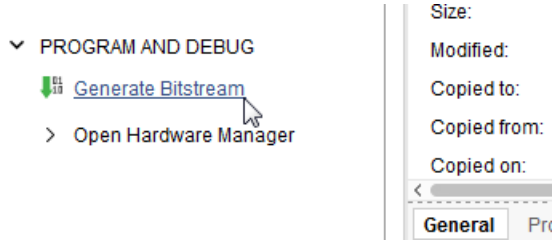

Figure 19: Generate Bitstream

2. The **No Implementation Results Available** dialog box appears. Click **Yes**. In the **Launch Runs** dialog box, accept all of the default settings (Launch runs on local host) and click **OK**.
3. After bitstream generation completes, the **Bitstream Generation Completed** dialog box appears. **Open Implemented Design** is selected by default. Click **OK**.
4. In the **Design Timing Summary** window, ensure that all timing constraints are met.

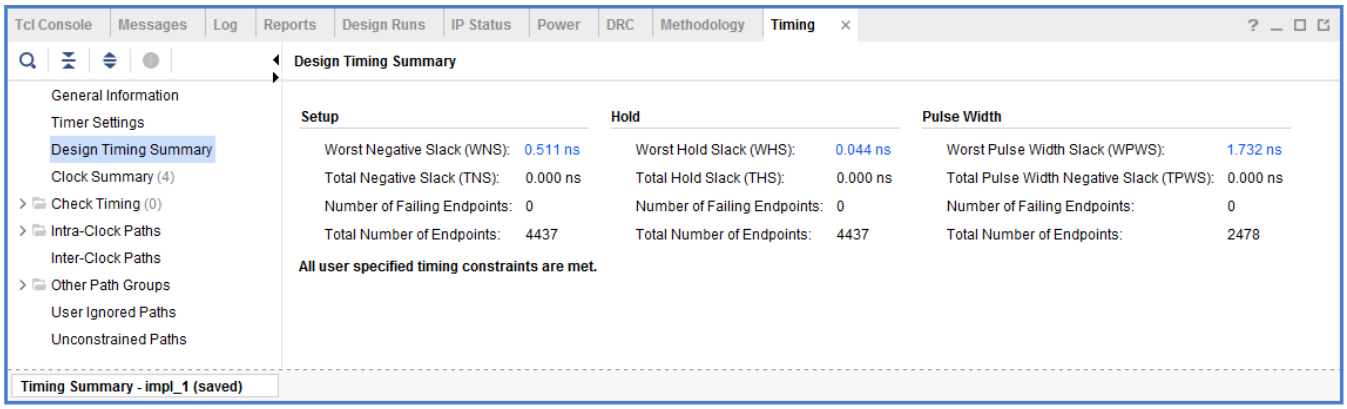

Figure 20: Review Design Timing Summary

5. Proceed to Lab 5: Using Vivado Logic Analyzer to Debug Hardware chapter to complete the rest of this lab.

Introduction

The Virtual Input/Output (VIO) core is a customizable core that can both monitor and drive internal FPGA signals in real time. The number and width of the input and output ports are customizable in size to interface with the FPGA design. Because the VIO core is synchronous to the design being monitored and/or driven, all design clock constraints that are applied to your design are also applied to the components inside the VIO core. Run time interaction with this core requires the use of the Vivado® logic analyzer feature. The following figure is a block diagram of the new VIO core.

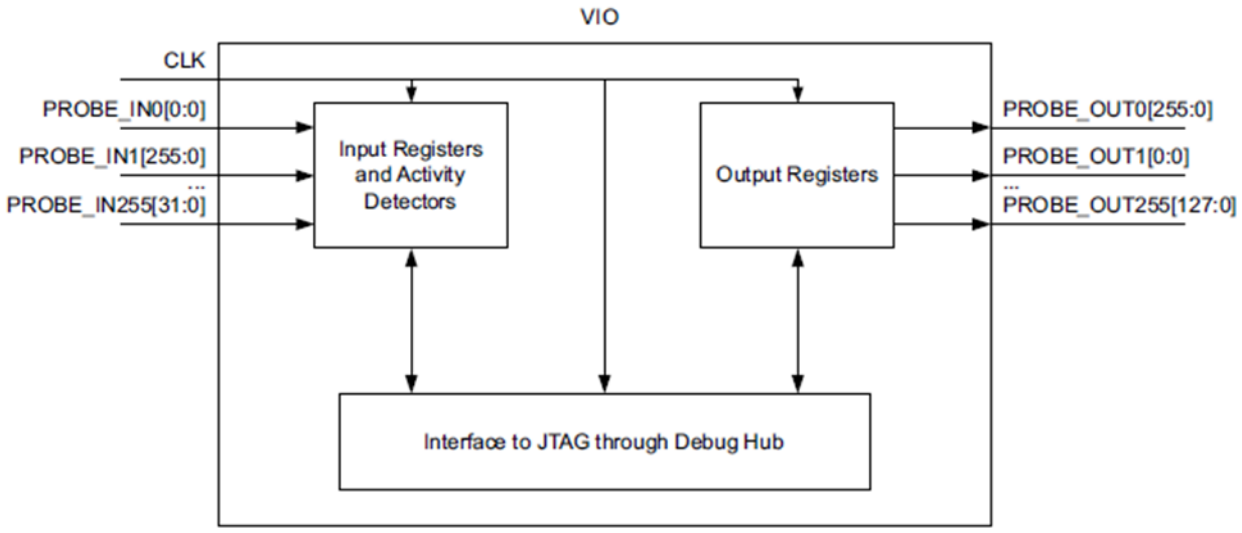

Figure 21: VIO Block Diagram

This lab walks you through the steps of instantiating and configuring the VIO core. It walks you through the steps of connecting the I/Os of the design to the VIO core. This way, you can debug your design when you do not have access to the hardware or the hardware is remotely located.

The following ports are created:

- One 4-bit PROBE_IN0 port. This has two bits to monitor the 2-bit Sine Wave selector outputs from the finite state machine (FSM) and other two bits to mimic the state of the other two LEDs on the board. We will configure these 4-bit signals as LEDs during run time to mimic the LEDs displayed on the KC705 board.
- One 2-bit PROBE_OUT0 port to drive the input buttons on the FSM. We will configure it so one bit can be used as a toggle switch during run time to mimic the "PUSH_BUTTON", SW3, and second bit will be used as the "PUSH_BUTTON", SW6.

Step 1: Creating a Project with the Vivado New Project Wizard

To create a project, use the New Project wizard to name the project, to add RTL source files and constraints, and to specify the target device.

1. Invoke Vivado IDE.
2. In the **Quick Start** tab, click **Create Project** to start the New Project wizard. Click **Next**.
3. In the **Project Name** page, name the new project **proj_hdl_vio** and provide the project location (`C:/Vivado_Debug`). Ensure that **Create project subdirectory** is selected. Click **Next**.
4. In the **Project Type** page, specify the **Type of Project** to create as **RTL Project**. Click **Next**.
5. In the **Add Sources** page:
   a. Set **Target Language** to **VHDL**.
   b. Click **Add Files**.
   c. In the **Add Source** Files dialog box, navigate to the `/src/lab3 directory`.
   d. Select all VHD source files, and click **OK**.
   e. Verify that the files are added, and **Copy Sources into Project** is selected. Click **Next**.
6. Click the green "+" sign, and then click **Add Files**.
7. In the **Add Source Directories** dialog box, navigate to the `/src/lab3` directory and choose the `sine_high`, `sine_low`, `sine_mid`, and `ila_0` directories. Click **Select**.
8. Verify that the files are added and **Copy sources into project** is selected. Click **Next**.
9. In the **Add Constraints** dialog box, click the "**+**" sign, and then click **Add Files**.
10. Navigate to `/src/lab3` directory and select `sinegen_demo_kc705.xdc`. Click **Next**.
11. In the **Default Part** page, specify the **xc7k325tffg900-2** part for the KC705 platform. You can also select **Boards** and then select **Kintex-7 KC705 Evaluation Platform**. Click **Next**.
12. Review the **New Project Summary** page. Verify that the data appears as expected, per the steps above. Click **Finish**.

    ***Note**: It might take a moment for the project to initialize.*

13. In the **Sources** window in Vivado IDE, expand sinegen_demo_inst_vio to see the source files for this lab. Note that `ila_0` core has been added to the project. However, vio_0 (the VIO core) is missing.

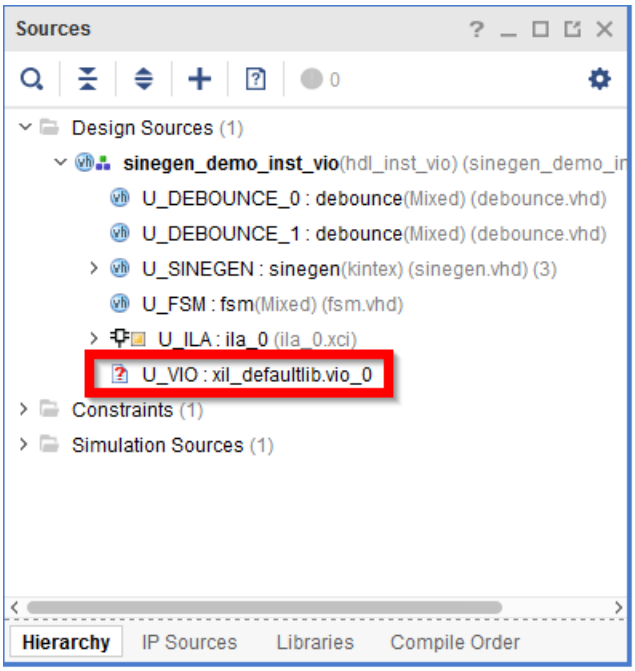

Figure 22: Missing Source for VIO Core

14. In this step, you will instantiate and configure this VIO core. From the **Flow Navigator**, click **IP Catalog**, expand **Debug & Verification**, then expand **Debug**, and double-click **VIO**. The **Customize IP** dialog box opens.
15. On the **General Options** tab, leave the **Component Name** to its default value of **vio_0**, **set Input Probe Count** to **1**, **Output Probe Count** to **1**, and select the **Enable Input Probe Activity Detectors** check box.

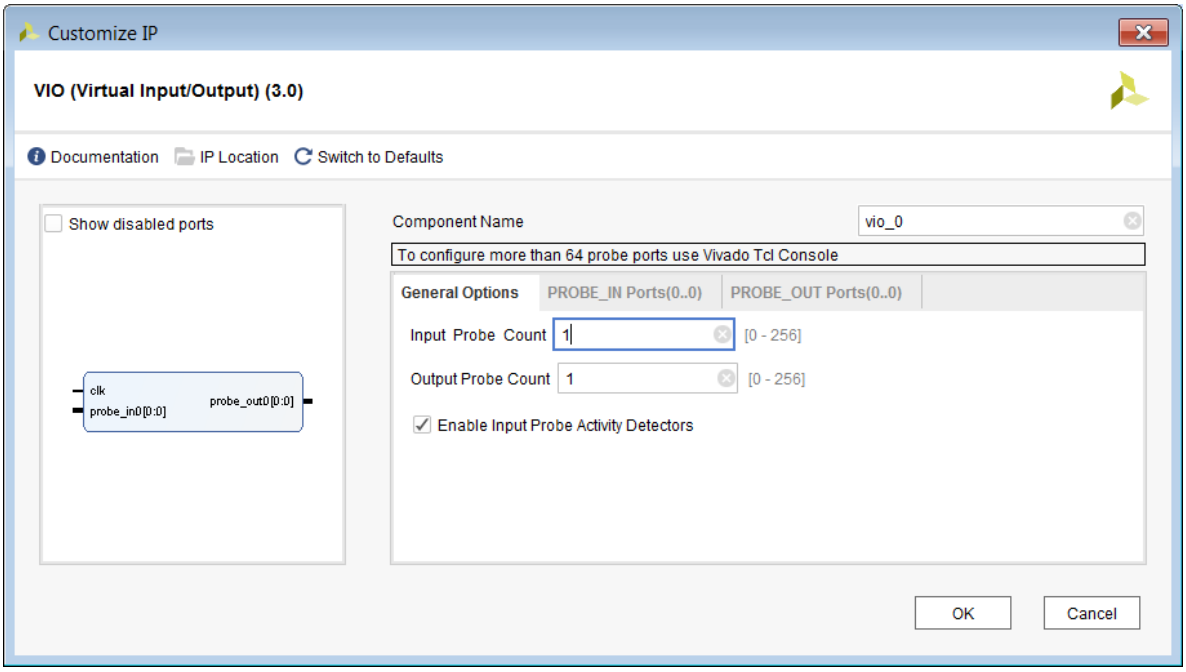

Figure 23: Configure General Options of the VIO Core

16. On the **PROBE_IN Ports** tab, set **Probe Width** to **4** bits wide.

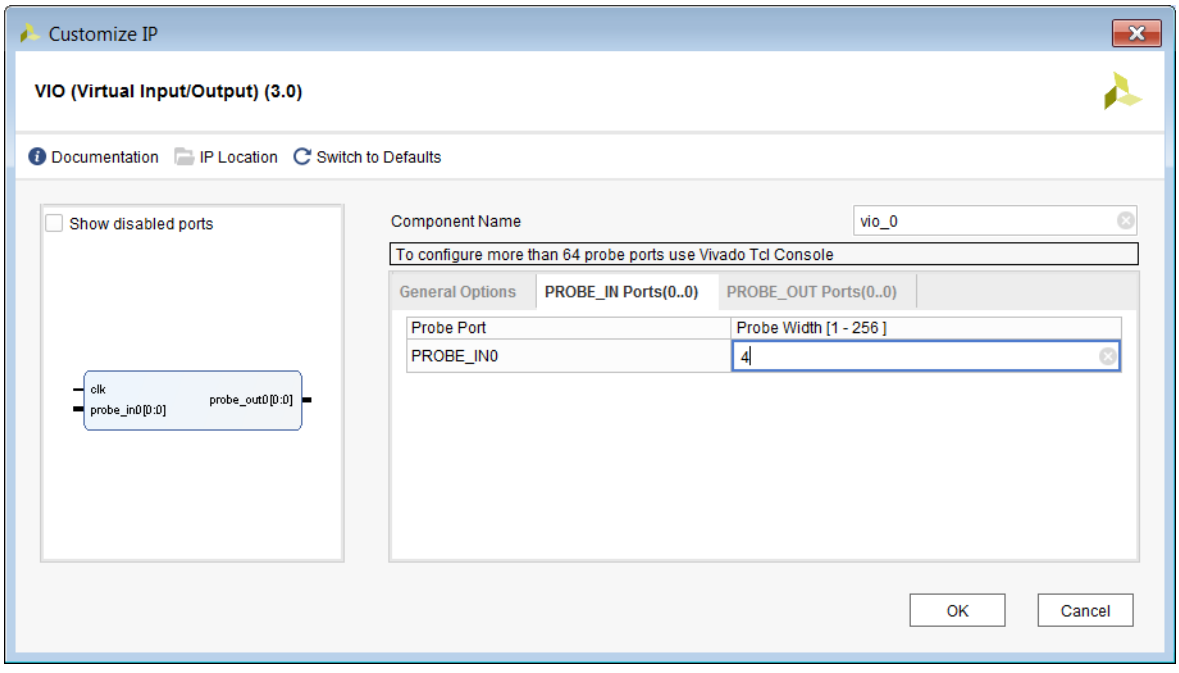

Figure 24: Configure PROBE_IN Ports of the VIO Core

17. On **the PROBE _OUT Ports**, set **Probe Width** to **2** bits wide with an initial value of **0** in hex format.

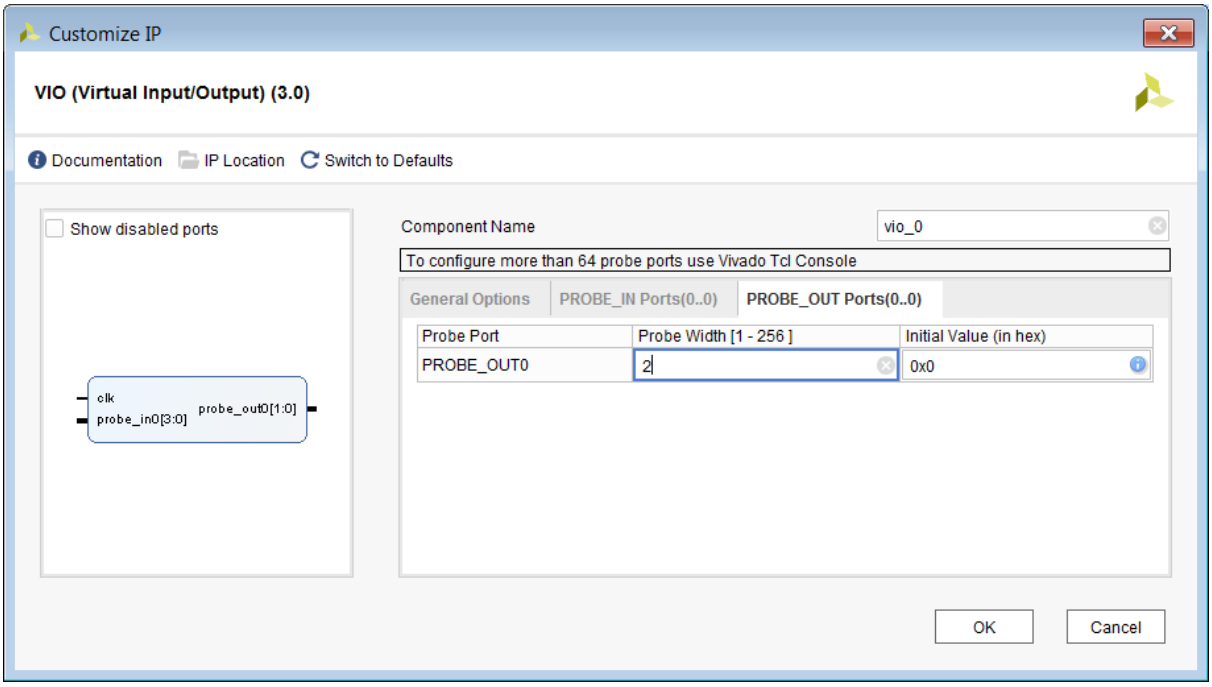

Figure 25: Configure the PROBE_OUT Ports of the VIO Core

18. Click **OK** to generate the IP. The **Generate Output Products** dialog box will appear. Click **Generate**.

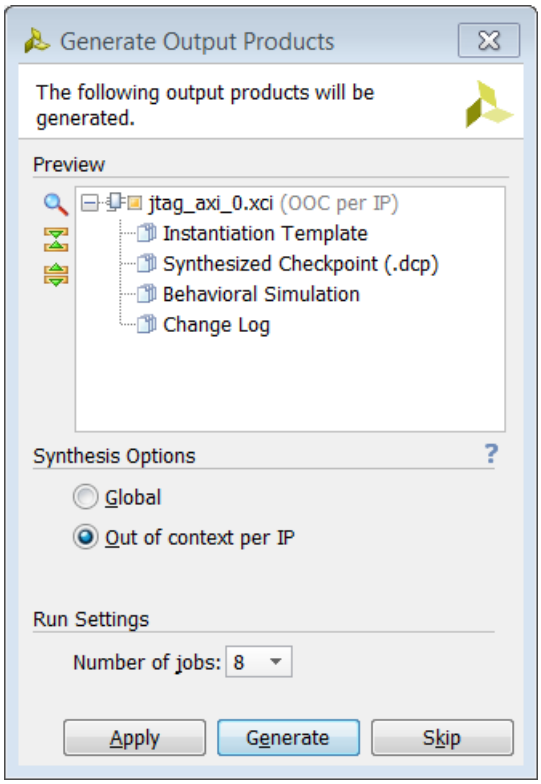

Figure 26: Generate Output Products for the VIO Core

Output product generation should take less than a minute. At this point, you have finished customizing the VIO. This core has already been instantiated in the top level design as shown in the following figure.

```
-------------------------------------------------------------
-- VIO
-------------------------------------------------------------
U_VIO : vio_0
  port map
  (
    CLK => clk,
    PROBE_IN0(3) => DONT_EAT,
    PROBE_IN0(2) => GPIO_BUTTONS_re(1),
    PROBE_IN0(1 downto 0) => sineSel,
    PROBE_OUT0(1) => push_button_reset,
    PROBE_OUT0(0) => push_button_vio
  );
```

Figure 27: VIO Instantiation in the Top Level Design

At this point, the **Sources** window should look as shown in the following figure.

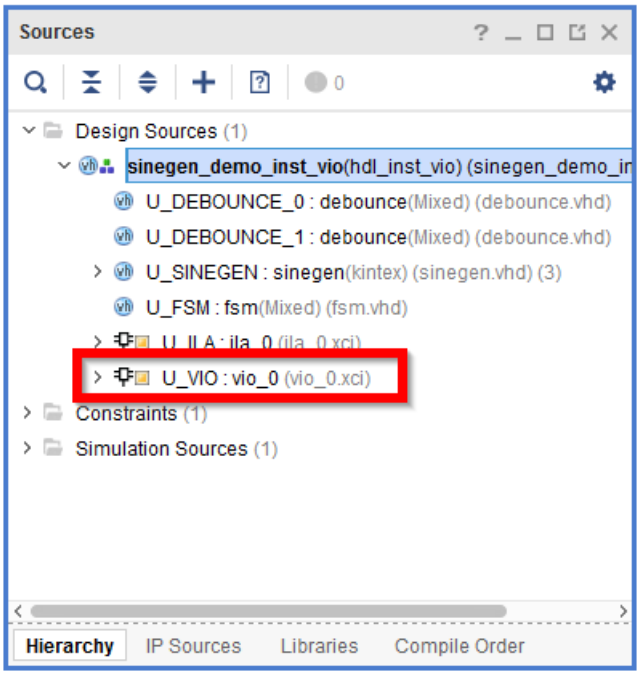

Figure 28: Instantiated VIO Core in the Sources Window

19. Double-click **sinegen_demo_inst.vhd** in the **Sources** window, to open it and inspect the instantiation and port mapping of the ILA core in the HDL code.

```
------------------------------------------------------------
-- ILA
------------------------------------------------------------
U_ILA : ila_0
  port map
  (
    CLK => clk,
    PROBE0 => sineSel,
    PROBE1 => sine,
    PROBE2 => GPIO_BUTTONS_db,
    PROBE3 => GPIO_BUTTONS_re,
    PROBE4 => GPIO_BUTTONS_dly,
    PROBE5 => GPIO_BUTTONS
  );
```

Figure 29: Hook signals that need to be debugged in the ILA

Step 2: Synthesize, Implement, and Generate Bitstream

1. From the **Program and Debug** drop-down list, in **Flow Navigator**, click **Generate Bitstream**. This synthesizes, implements, and generates a bitstream for the design.
2. The **Missing Implementation Results** dialog box appears. Click **OK**.
3. After bitstream generation completes, the **Bitstream Generation Completed** dialog box appears. **Open Implemented Design** is selected by default. Click **OK**.
4. Inspect the Timing Summary report and make sure that all timing constraints have been met.

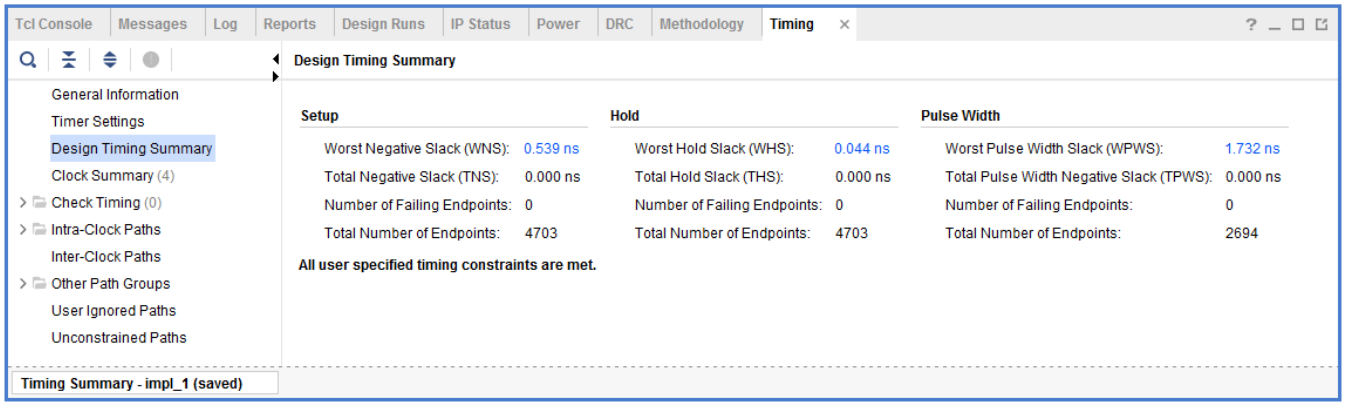

Figure 30: Report Timing Summary Dialog Box

5. Proceed to Lab 5: Using Vivado Logic Analyzer to Debug Hardware chapter to complete the rest of the steps for debugging the design. Skip forward to **Verifying the VIO Core Activity (Only applicable to Lab 3)** section to complete the rest of this lab.

Introduction

This simple tutorial shows how to do the following:

- Create a Synplify Pro project for the wave generator design.
- Mark nets for debug in the Synplify Pro constraints file as well as VHDL source files.
- Synthesize the Synplify Pro project to create an EDIF netlist.
- Create a Vivado® project based on the Synplify Pro netlist.
- Use the Vivado IDE to setup and debug the design from the synthesized design using Synplify Pro.

Step 1: Create a Synplify Pro Project

1. Launch Synplify Pro and select **File > New**.
2. Set **File Type** to **Project File (Project)** as highlighted in the following figure.
3. In the **New File Name** box, enter **synplify_1**.
4. Click **OK**.

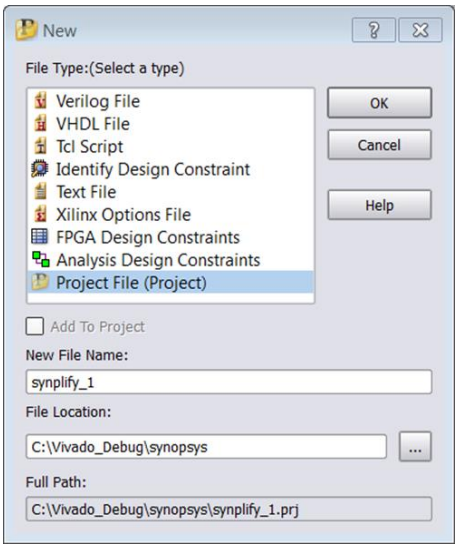

Figure 31: Synplify Pro New Project Dialog Box

5. If you get a dialog box asking you to create a non-existing directory, click **OK**.

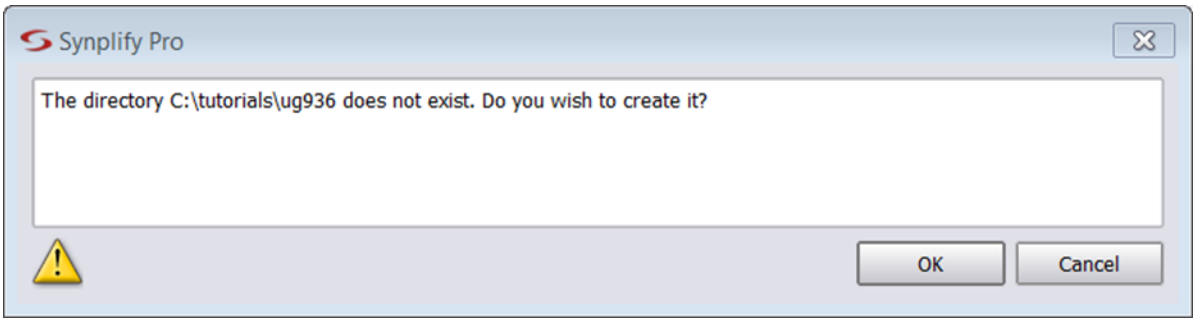

Figure 32: Synplify Pro project Confirmation Dialog Box

6. In the left panel of the **Synplify Pro** window, click **Add File** as shown in the following figure.

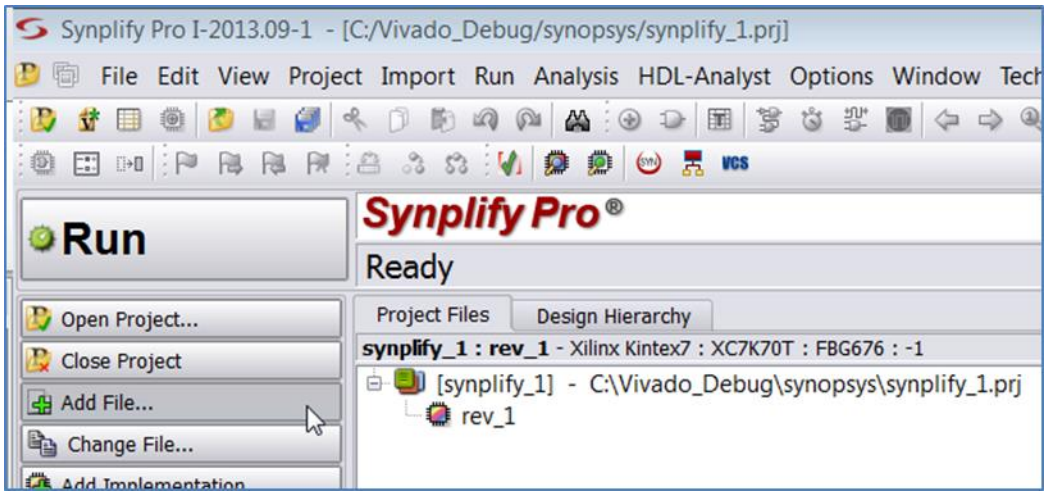

Figure 33: Adding Files to a Synplify Pro Project

7. In the **Add Files to Project** dialog box, change the **Files of Type** to **HDL File**. Navigate to `C:\Vivado_Debug\src\lab4`, which shows all the VHDL source files needed for this lab. Select the following three files by pressing the **Ctrl** key and clicking on them.
   - `debounce.vhd`
   - `fsm.vhd`
   - `sinegen_demo.vhd`
8. Click **Add**.

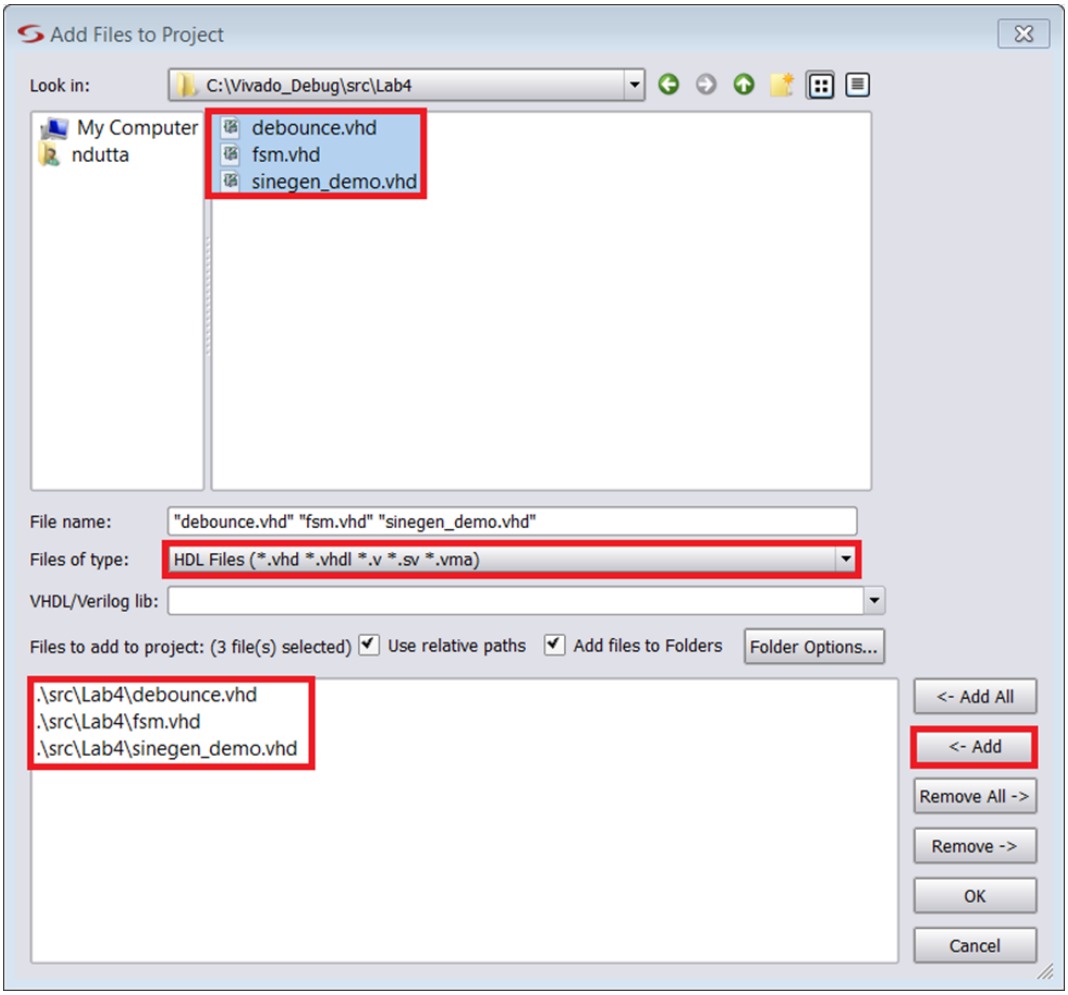

Figure 34: Adding VHDL Source Files to the Synplify Pro Project

10. In the same dialog box set **Files of type** to **Constraints Files**. This shows the `synplify_1.sdc` file. Select the file and click **Add** as shown in the following figure.

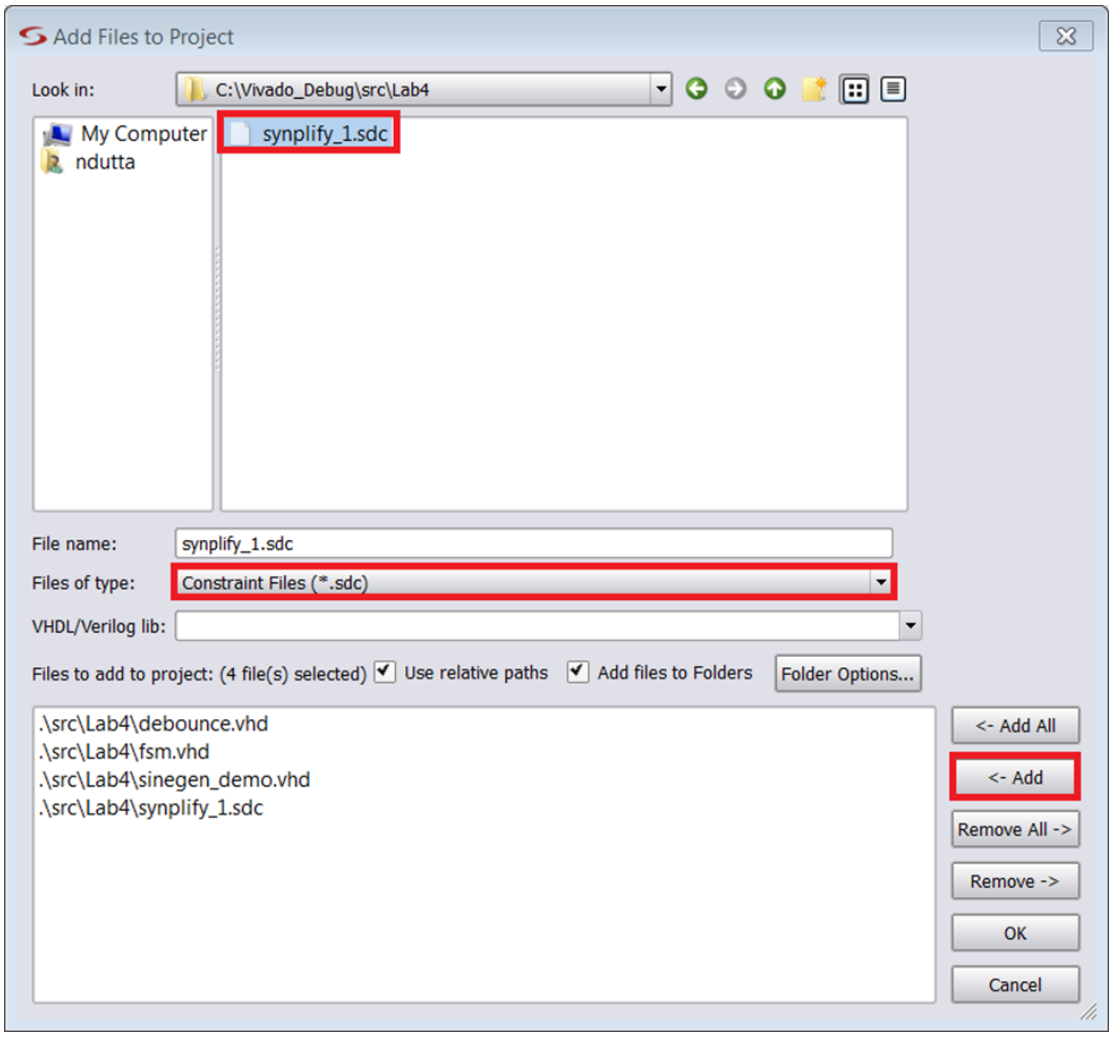

Figure 35: Adding SDC Constraints File to the Synplify Pro Project

11. In the same dialog box, set **Files of type** to **FPGA Constraint Files**. This shows the `synplify_1.fdc` file. Select the file and click **Add** as shown in the following figure. Click **OK**.

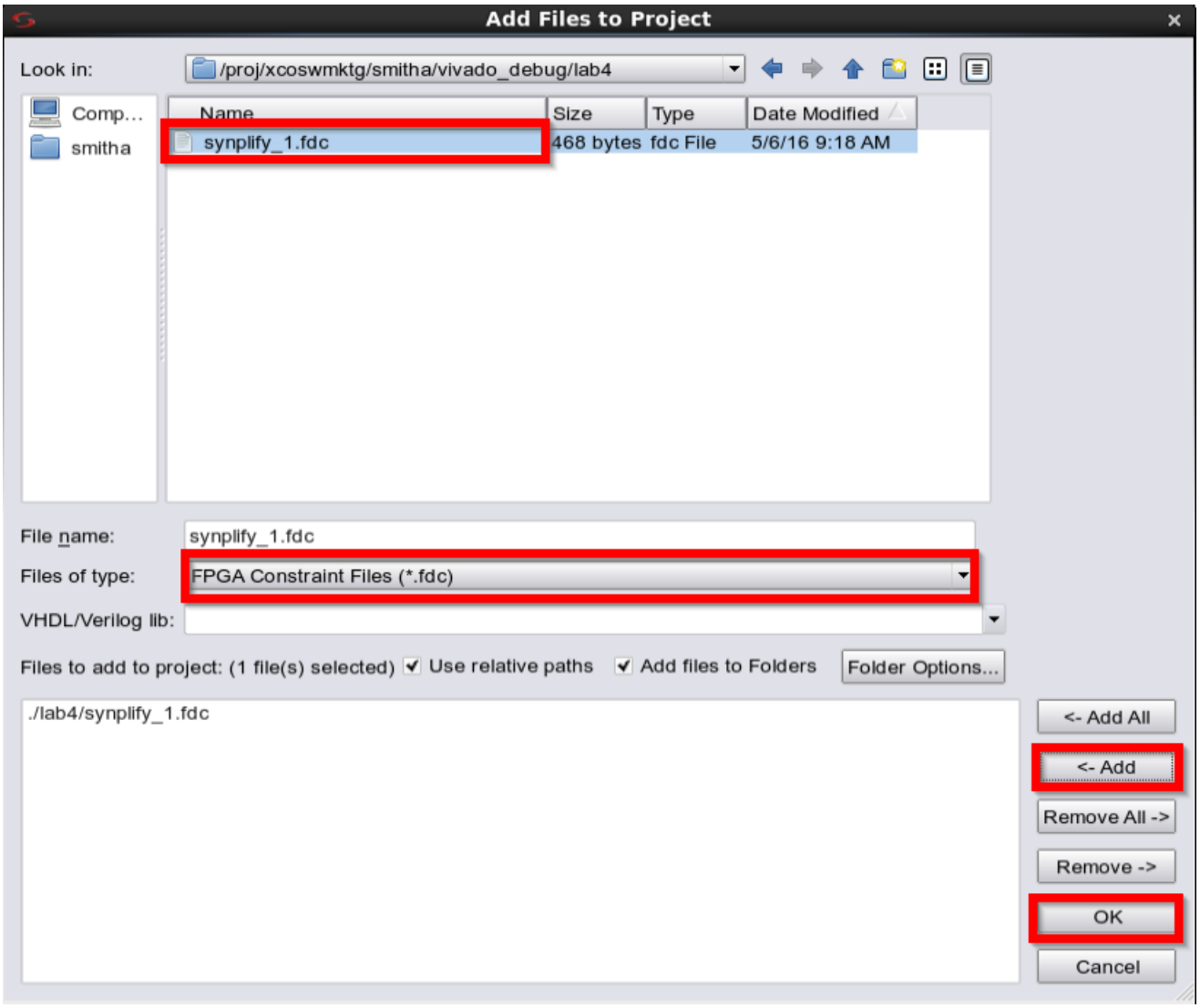

Figure 36: Adding FPGA Constraints File to the Synplify Pro Project

12. Now, you need to set the implementation options.
13. Click **Implementation Options** in the **Synplify Pro** window as shown in the following figure.

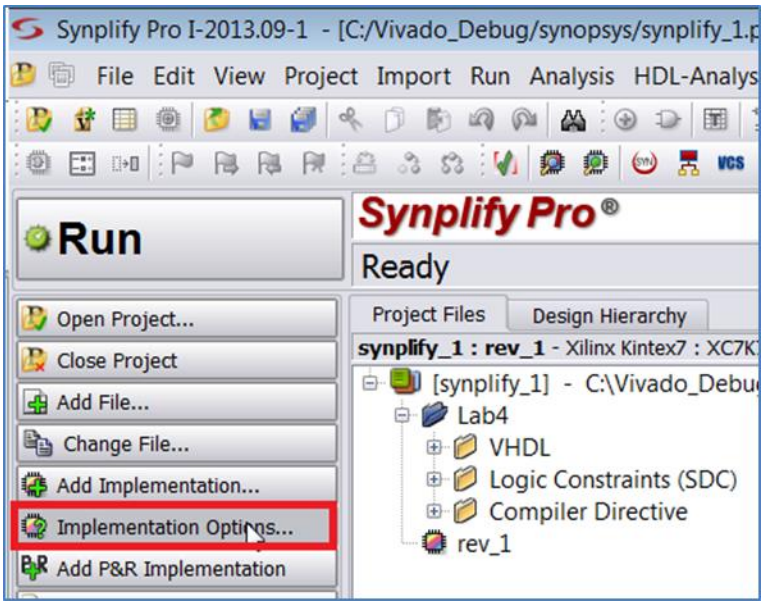

Figure 37: Opening Implementation Options in Synplify Pro

14. This brings up the **Implementation Options** dialog box as shown in the following figure. In the **Device** tab, set **Technology** to **Xilinx Kintex7**, **Part** to **XC7K325T**, **Package** to **FFG900** and **Speed** to **-2**. Leave all the other options at their default values. Click **OK**.

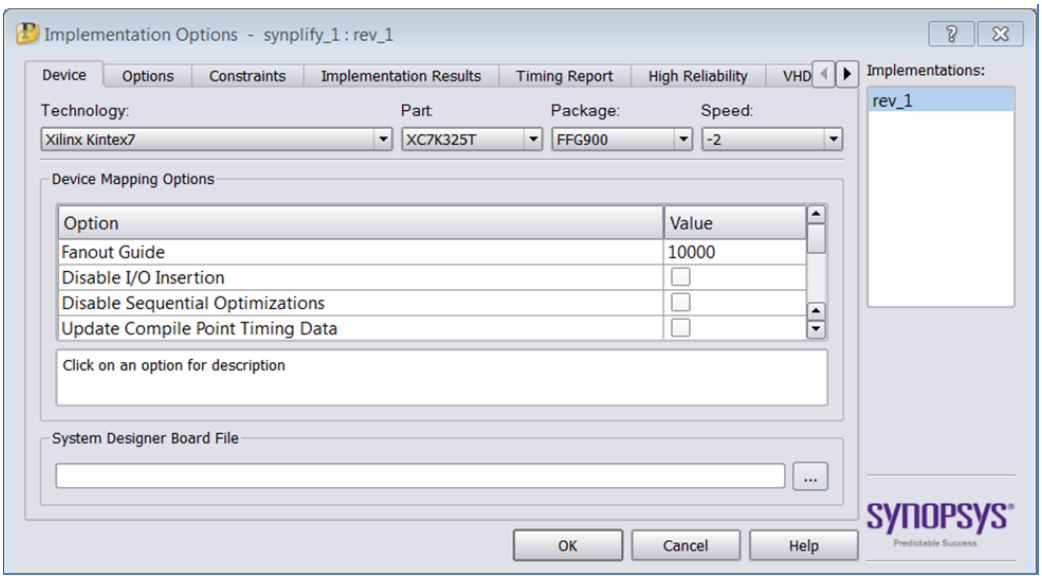

Figure 38: Specifying Implementation Options in Synplify Pro

15. You need to preserve the net names that you want to debug by putting attributes in the HDL files. These attributes are already placed in the `sinegen_demo.vhd`, file of this tutorial. Open the `sinegen_demo.vhd` file and inspect the lines shown.

```
-- Attributes for Synplify Pro
attribute syn_keep : boolean;
attribute syn_keep of GPIO_BUTTONS_db    : signal is true;
attribute syn_keep of GPIO_BUTTONS_dly   : signal is true;
attribute syn_keep of GPIO_BUTTONS_re    : signal is true;
```

Figure 39: Specifying Attributes to Preserve Net Names in Synplify

16. You also can specify the MARK_DEBUG attributes in the source HDL files to mark the signals for debug, as shown in the code snippet from `singen_demo.vhd` file.

```
-- Add mark_debug attributes to show debug nets in the synthesized netlist
attribute mark_debug : string;
attribute mark_debug of GPIO_BUTTONS_db : signal is "true";
attribute mark_debug of GPIO_BUTTONS_dly : signal is "true";
attribute mark_debug of GPIO_BUTTONS_re : signal is "true";
```

Figure 40: Add MARK_DEBUG Attribute in HDL File

17. The `synplify_1.sdc` file contains various kinds of constraints such as pin location, I/O standard, and clock definition. The `synplify_1.fdc` file contains directives for the compiler. Here is where the nets of interest to us that are marked for debug are located. The attribute and the nets selected for debug are shown in the following figure.

```
# Attributes that are needed to mark_debug the nets that are needed to be viewed in ILA

define_attribute  -comment {Mark sinegen as black box} {v:work.sinegen} {syn_black_box} {1}
define_attribute  -comment {Set no_prune on sinegen} {v:work.sinegen} {syn_noprune} {1}
define_attribute  -comment {Mark entire bus for debug} {i:sinegen.sine[*]} {mark_debug} {"true"}
define_attribute  -comment {Mark entire bus for debug} {i:sinegen.sel[*]} {mark_debug} {"true"}
~
```

Figure 41: Synplify Pro Constraints in FDC Files

In the above constraints, sinegen has been defined as a black box by using the syn_black_box attribute. Second, the syn_no_prune attribute has been used so that the I/Os of this block are not optimized away. Finally, two nets, `sine[20:0]` and `sel[1:0]`, have been assigned the MARK_DEBUG attribute such that these two nets should show up in the synthesized design in Vivado IDE for further debugging. For further information on these attributes, please refer to the Synplify Pro User Manual and Synplify Pro Reference Manual.

Step 2: Synthesize the Synplify Project

1. Before implementing the project, you need to set the name for the output netlist file. By default, the name of the output netlist file is synplify_1.edf. To change the name of the output file, type the following command at the Tcl command prompt:

   ```
   %project -result_file "./rev_1/sinegen_demo.edf"
   ```

   You will use this file in Vivado IDE.

2. With all the settings in place, click the **Run** button in the left panel of the **Synplify Pro** window to start synthesizing the design.

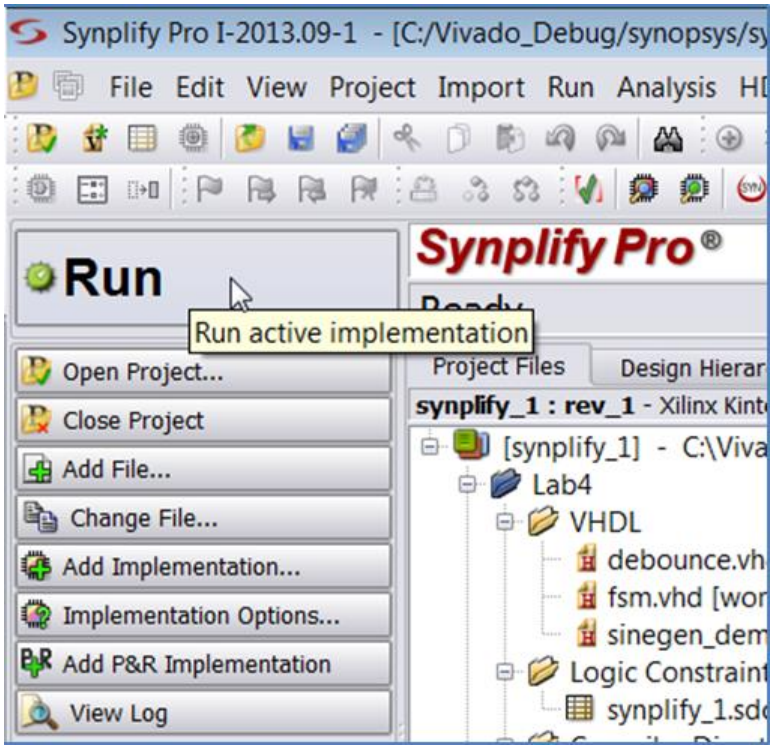

Figure 42: Synthesize the Design in Synplify

3. During synthesis, status messages appear in the **Tcl Script** tab. Warning messages are expected, but there should not be any Error messages. To see detailed messages, click the **Messages** tab in the bottom left-hand corner of the Synplify Pro console.

4. When synthesis completes, the output netlist is written to the file: `rev_1/sinegen_demo.edf`

   [Optional] To view the netlist select **View > View Result File**.

5. Click **File > Save All** to save the project, then click **File > Exit**.

Step 3: Create DCPs for the Black Box Created in Synplify Pro

The black box, sinegen, created in the Synplify Pro project, contains the Direct Digital Synthesizer IP. You need to create a synthesized design for this block. To do this, create an RTL type project in Vivado IDE by following the steps outlined below.

1. Launch Vivado IDE.
2. Click **Create Project**. This opens up the **New Project** wizard. Click **Next**.
3. Under **Project Name**, set the project name to **proj_synplify_netlist**. Click **Next**.
4. Under **Project Type**, select **RTL Project**. Click **Next**.
5. Under **Add Sources**, click **Add Files**, navigate to the `Vivado_Debug/src/lab4` folder and select the `sinegen.vhd` file. Set **Target Language** to **VHDL**. Ensure that **Copy sources into project box** is selected. Click **Next**.
6. Click **Add Files**, navigate to the `Vivado_Debug/src/lab4` folder and select the `sine_high.xci`, `sine_low.xci`, and `sine_mid.xci` files. Click **Next**.
7. Under **Default Parts**, select **Boards** and then select the **Kintex-7 KC705 Evaluation Platform** and correct version for your hardware. Click **Next**.
8. Under **New Project Summary**, ensure that all the settings are correct. Click **Finish**.
9. Once the project has been created, in Vivado **Flow Navigator**, under the **Project Manager** folder, click **Settings**. In the dialog box, in the left panel, click **Synthesis**. From the pull-down menu on the right panel, set **-flatten_hierarchy** to **none**. Click **OK**.
10. In Vivado IDE **Flow Navigator**, under **Synthesis Folder**, click **Run Synthesis**.
11. When synthesis completes the **Synthesis Completed** dialog box appears. Select **Open Synthesized Design** and click **OK**.
12. Click **File > Exit** in Vivado IDE. When the **OK to exit** dialog box pops up, click **OK**.

Step 4: Create a Post Synthesis Project in Vivado IDE

1. Launch Vivado IDE.
2. Click **Create Project**. This opens up the New Project wizard. Click **Next**.
3. Set the **Project Name** to **proj_synplify**. Click **Next**.
4. Under **Project Type**, select **Post-synthesis Project**. Click **Next**.
5. Under **Add Netlist Sources**, click **Add Files**, navigate to the `Vivado_Debug/synopsys/rev_1` folder, and select **sinegen_demo.edf**. Click **OK**.

6. Add the netlist file created in the previous section. Click **Add Files** again, navigate to the `proj_synplify_netlist/proj_synplify_netlist.runs/synth1` folder and select the following file:
   - `sinegen.dcp`

   Add the DCP files created for the sub-module IPs in the previous section. Click **Add Directories** again, navigate to the `proj_synplify_netlist/proj_synplify_netlist.srcs/sources_1/ip` folder and select the following:
   - `sine_high`
   - `sine_mid`
   - `sine_low`

   Click **OK** in the **Add Source Files** dialog box. In the **Add Netlist Sources** dialog box ensure that **Copy Sources into Project** is selected. Click **Next**.
7. Click **Add Files**, navigate to the `Vivado_Debug/src` folder, and select the **sinegen_demo_kc705.xdc** file. This file has the appropriate constraints needed for this Vivado project. Click **OK** in the **Add Constraints File** dialog box. In the **Add Constraints (optional)** dialog box ensure that **Copy Constraints into Project** is selected. Click **Next**.
8. Under **Default Part**, select **Boards** and then select **Kintex-7 KC705 Evaluation Platform** and the right version number for your hardware. Click **Next**.
9. Under **New Project Summary**, ensure that all the settings are correct and click **Finish**.
10. In the **Sources** window, ensure **sinegen_demo.edf** is selected as the top module.

Step 5: Add More Debug Nets to the Project

1. In Vivado IDE, in the **Flow Navigator**, select **Open Synthesized Design** from the **Netlist Analysis** folder.
2. Select the **Netlist** tab in the **Netlist** window to expand Nets. Select the following nets for debugging:
   - `GPIO_BUTTONS_c(2)`
   - `sine (20)`

   After selecting all the specified nets, right-click the nets and click **Mark Debug**, as shown in the following figure.

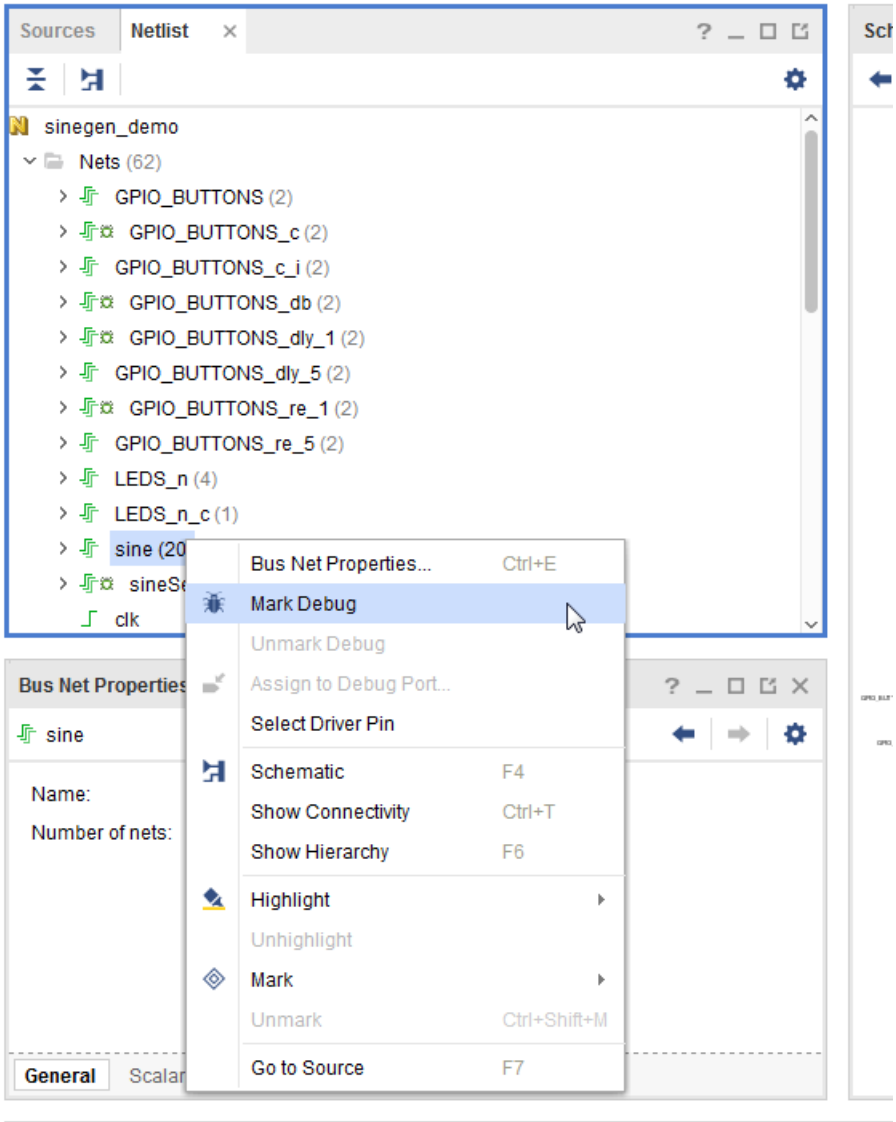

Figure 43: Mark Additional Signals for Debug

3. You should be able to see all the nets that are marked for debug, as shown in the following figure.

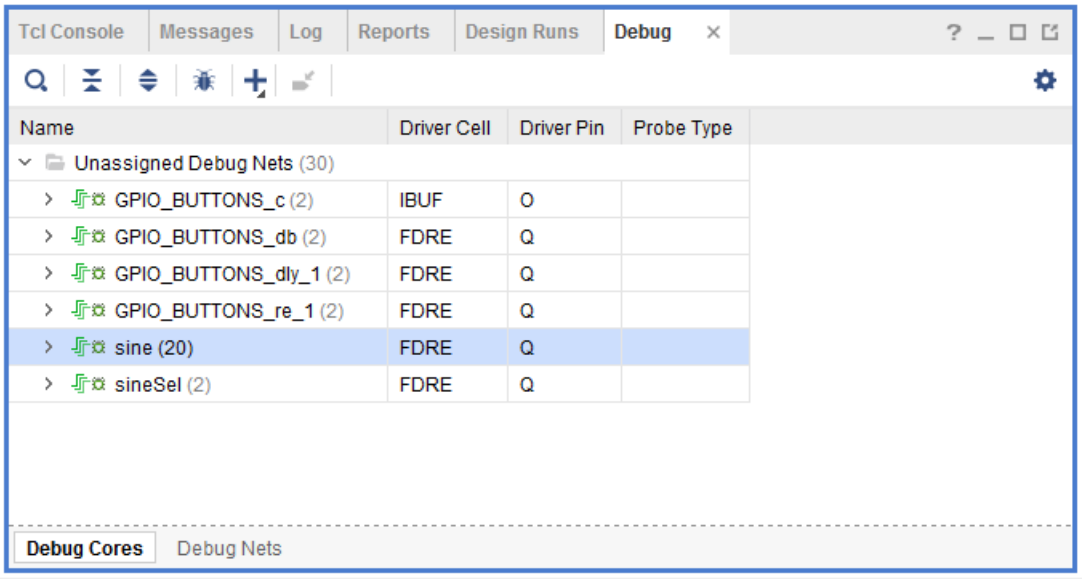

Figure 44: Nets Added for Debug through the Synplify Pro Flow in Vivado IDE

Running the Set up Debug Wizard

1. Click the **Set up Debug** icon in the **Debug** window or select the **Tools** menu, and select **Set up Debug**. The **Set up Debug** wizard opens.

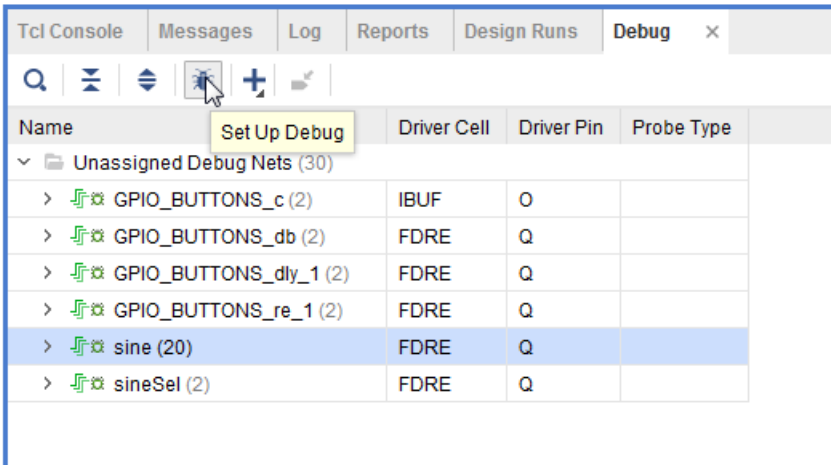

Figure 45: Run the Set up Debug Wizard

2. Click through the wizard to create Vivado logic analyzer debug cores, keeping the default settings.

   ***Note**: In the **Specify Nets to Debug** dialog box, ensure that all the nets marked for debug have the same clock domain.*

Step 6: Implementing the Design and Generating the Bitstream

1. In the **Flow Navigator**, under the **Program and Debug** drop-down list, click **Generate Bitstream**.
2. In the **Save Project** dialog box, click **Save**.
3. When the Bitstream generation finishes, the **Bitstream Generation Completed** dialog box pops-up and **Open Implemented Design** is selected by default. Click **OK**.
4. If you get a dialog box asking to close the synthesized design before opening the implemented design, click **Yes**.
5. Proceed to Lab 5: Using Vivado Logic Analyzer to Debug Hardware to complete the rest of this lab.

Introduction

The final step in debugging is to connect to the hardware and debug your design using the Integrated Logic Analyzer. Before continuing, make sure you have the KC705 hardware plugged into a machine.

In this step, you learn:

- How to debug the design using the Vivado® logic analyzer.
- How to use the currently supported Tcl commands to communicate with your target board (KC705).
- How to discover and correct a circuit problem by identifying unintended behaviors of the push button switch.
- Some useful techniques for triggering and capturing design data.

Step 1: Verifying Operation of the Sine Wave Generator

After doing some setup work, you will use Vivado logic analyzer to verify that the sine wave generator is working correctly. Your two primary objectives are to verify that:

- All sine wave selections are correct.
- The selection logic works correctly.

Target Board and Server Set Up

Connecting to the target board remotely

If you plan to connect remotely, you need to make sure that the KC705 board is plugged into a machine and you are running an hw_server application on that machine. If you plan to connect locally, skip steps 1-5 below and go directly to the Connecting to the Target Board Locally section.

1. Connect the Digilent USB JTAG cable of your KC705 board to a USB port on a Windows system.
2. Ensure that the board is plugged in and powered on.
3. Power cycle the board to clear the device.
4. Turn DIP switch positions (pin 1 on SW13, De-bounce Enable) to the OFF position.

5. Assuming you are connecting your KC705 board to a 64-bit Windows machine and you will be running the hw_server from the network instead of your local drive, open a `cmd` prompt and type the following:

   ```
   <Xilinx_Install>\Vivado\2018.x\bin\hw_server
   ```

   Leave this `cmd` prompt open while the hw_server is running. Note the machine name that you are using, you will use this later when opening a connection to this instance of the hw_server application.

Connecting to the Target Board Locally

If you plan to connect locally, ensure that the KC705 board is plugged into a Windows machine and then perform the following steps:

1. Connect the Digilent USB JTAG cable of your KC705 board to a USB port on a Windows system.
2. Ensure that the board is plugged in and powered on.
3. Power cycle the board to clear the device.
4. Turn DIP switch positions (pin 1 on SW13, De-bounce Enable) to the OFF position.

Using the Vivado Integrated Logic Analyzer

1. In the **Flow Navigator**, under **Program and Debug**, select **Open Hardware Manager**.

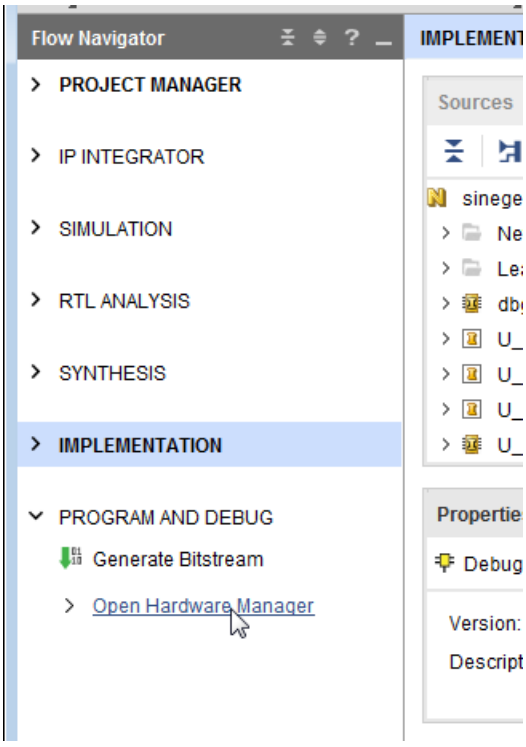

Figure 46: Open Hardware Manager

2. The **Hardware Manager** window opens. Click **Open Target > Open New Target**.

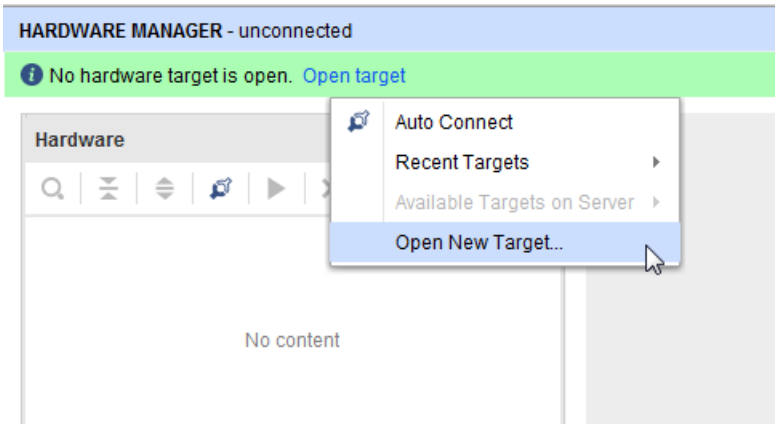

Figure 47: Connect to a Hardware Target

3. The **Open New Hardware Target** wizard opens. Click **Next**.
4. In the **Hardware Server Settings** page, type the name of the server (or select **Local server** if the target is on the local machine) in the **Connect to** field. Click **Next**.

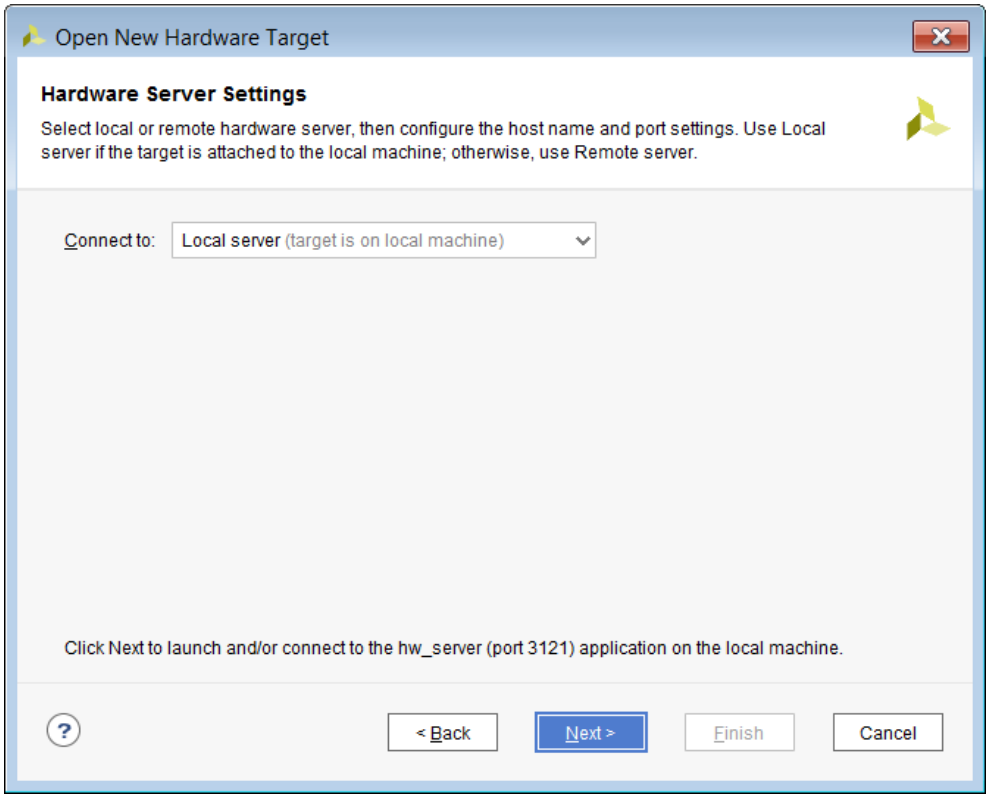

Figure 48: Hardware Server Settings

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

5. If there is more than one target connected, you will see multiple entries in the **Select Hardware Target** page. In this tutorial, there is only one target, as shown in the following figure. Click **Next**.

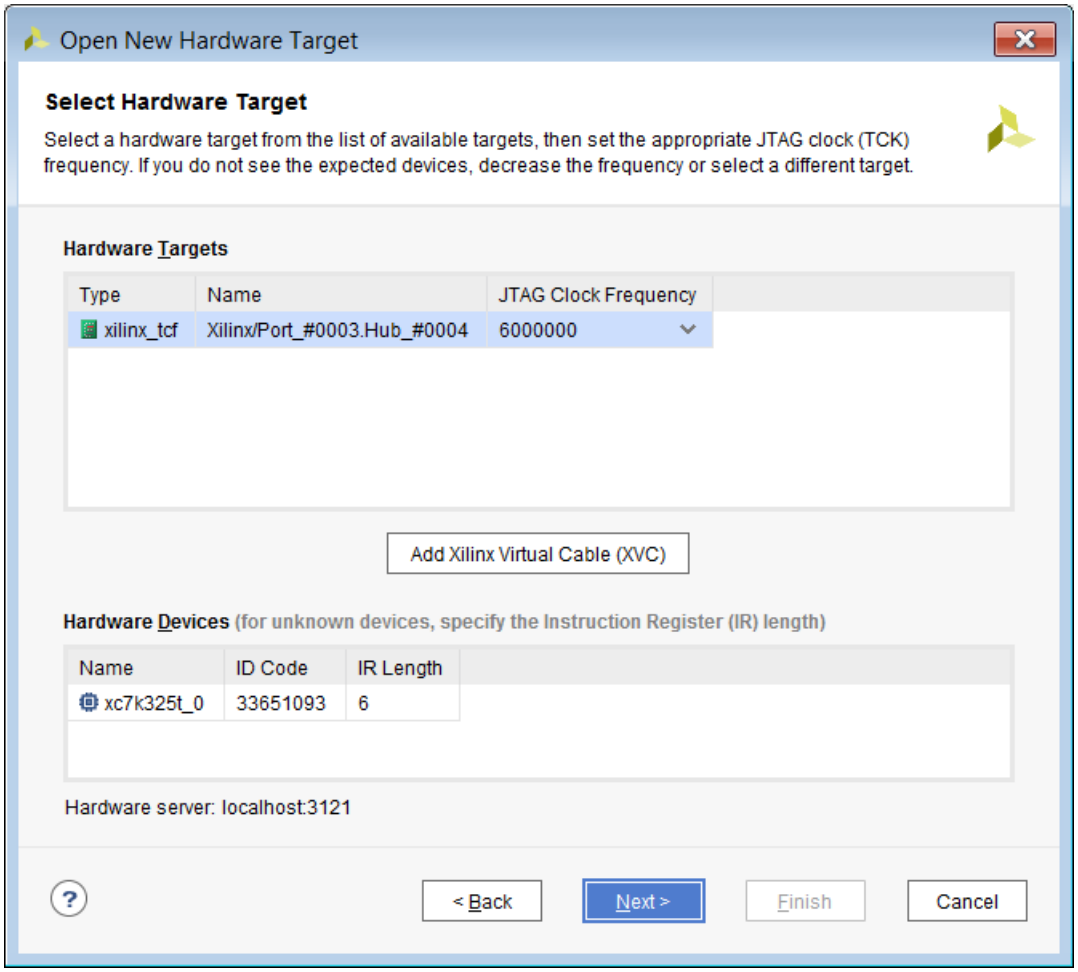

Figure 49: Select Hardware Target

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

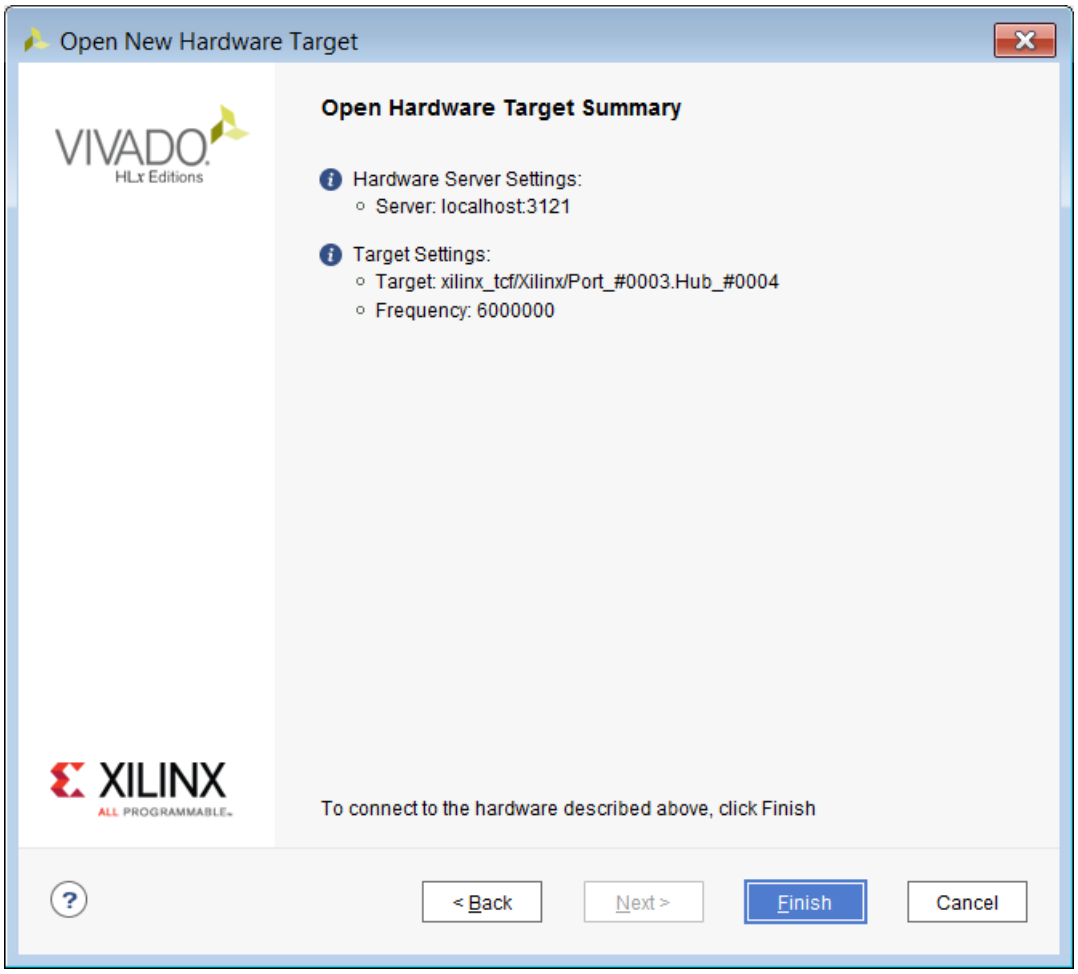

Figure 50: Hardware Target Summary

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

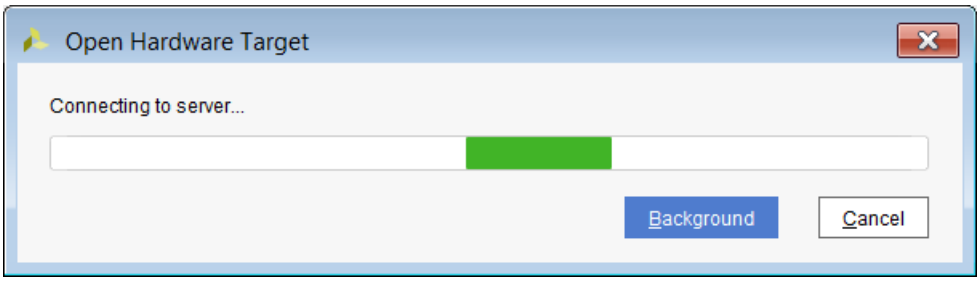

Figure 51: Open Hardware Target

After the connection to the hardware target is made, the Hardware window appears as in the following figure.

***Note**: The **Hardware** tab in the **Debug** view shows the hardware target and XC7K325T device detected in the JTAG chain.*

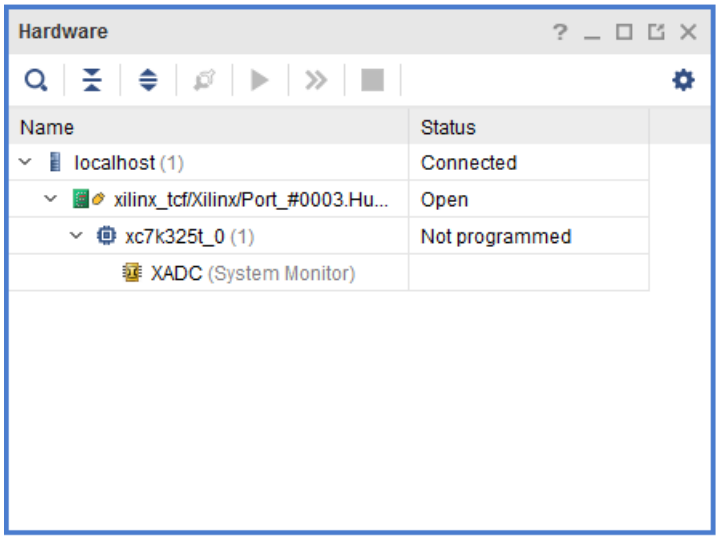

Figure 52: Active Target Hardware

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

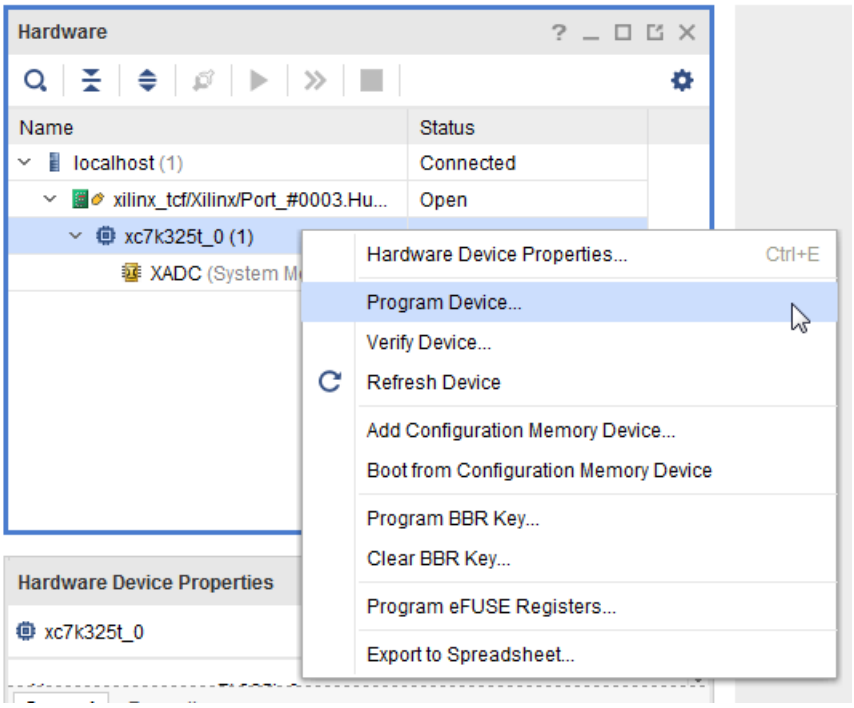

Figure 53: Program Active Target Hardware

9. In the **Program Device** dialog box verify that the `.bit` and `.ltx` files are correct for the lab that you are working on and click **Program** to program the device as shown in the following figure.

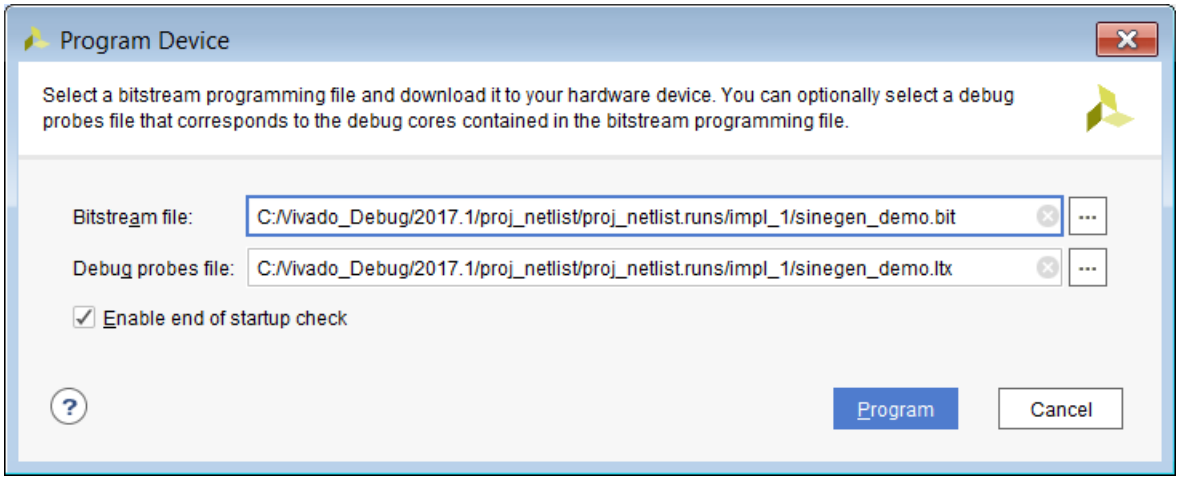

Figure 54: Select Bitstream File to Download for Lab 1

CAUTION! *The file paths of the bitstream and debug probes to be programmed will be different for different labs. Ensure that the relative paths are correct.*

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

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

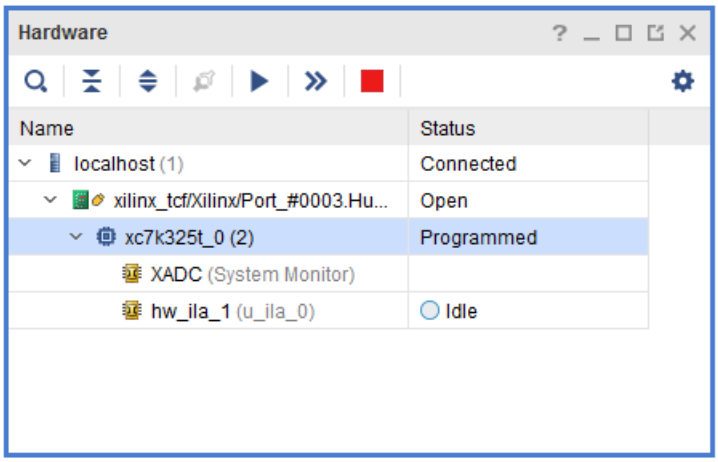

Figure 55: ILA Core Detection

11. The Integrated Logic Analyzer dashboard opens, as shown in the following figure.

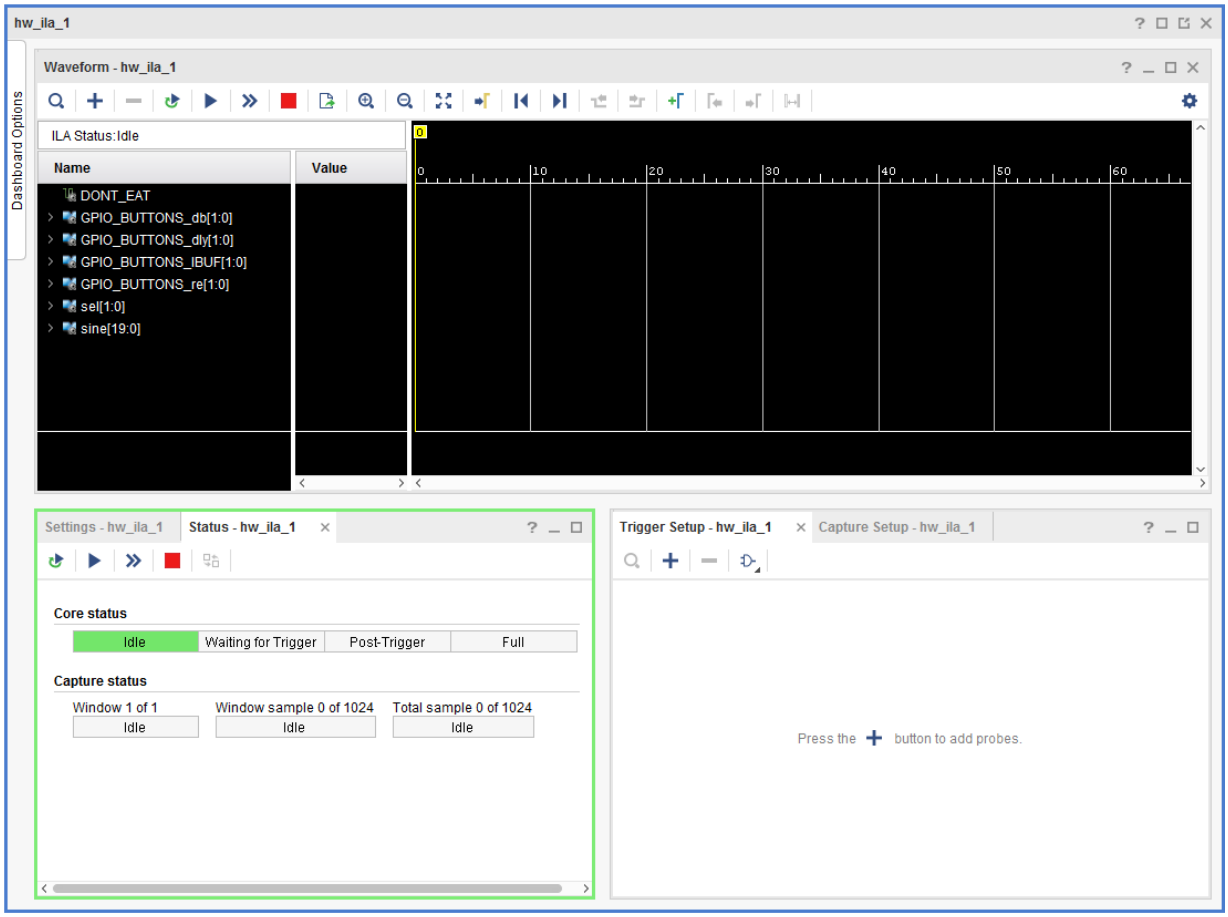

Figure 56: Vivado Integrated Logic Analyzer window

Verifying Sine Wave Activity

12. In the **Hardware** window, click **Run Trigger Immediate** to trigger and capture data immediately as shown in shown in the following figure.

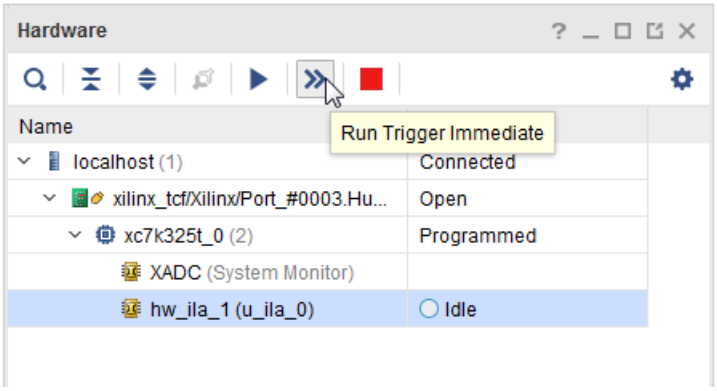

Figure 57: Run Trigger Immediate Button

13. In the **Waveform** window, verify that there is activity on the 20-bit sine signal as shown in the following figure.

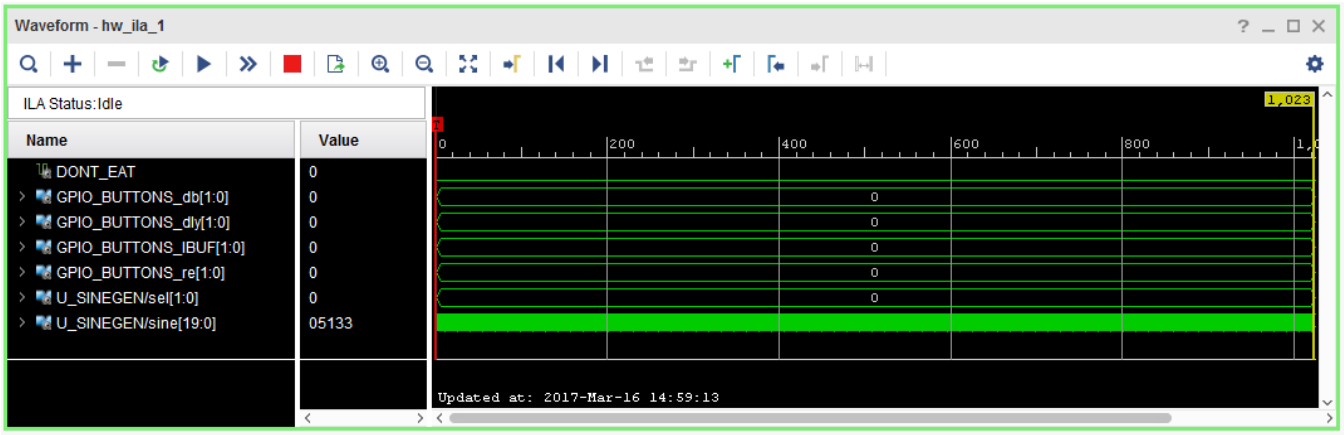

Figure 58: Output Sine Wave Displayed in Digital Format

Displaying the Sine Wave

14. Right-click **U_SINEGEN/sine[19:0]** signals, and select **Waveform Style > Analog** as shown in the following figure.

TIP: *The waveform does not look like a sine wave. This is because you must change the radix setting from Hex to Signed Decimal, as described in the following subsection.*

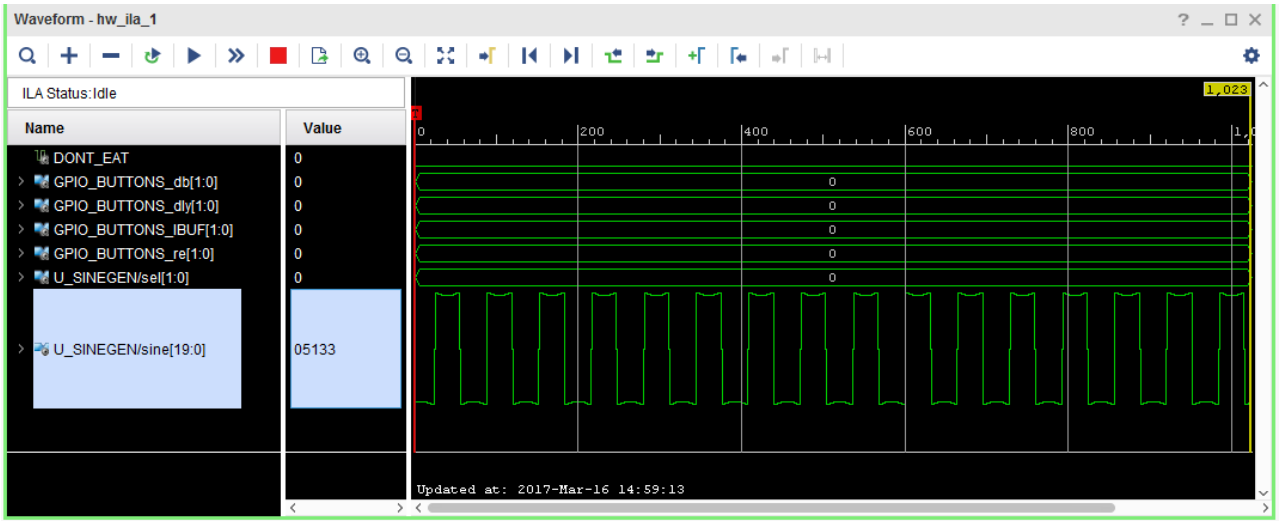

Figure 59: Output Sine Wave Displayed in Analog Format - High Frequency 1

15. Right-click **U_SINEGEN/sine[19:0]** signals, and select **Radix > Signed Decimal**.

    You should now be able to see the high frequency sine wave as shown in the following figure instead of the square wave.

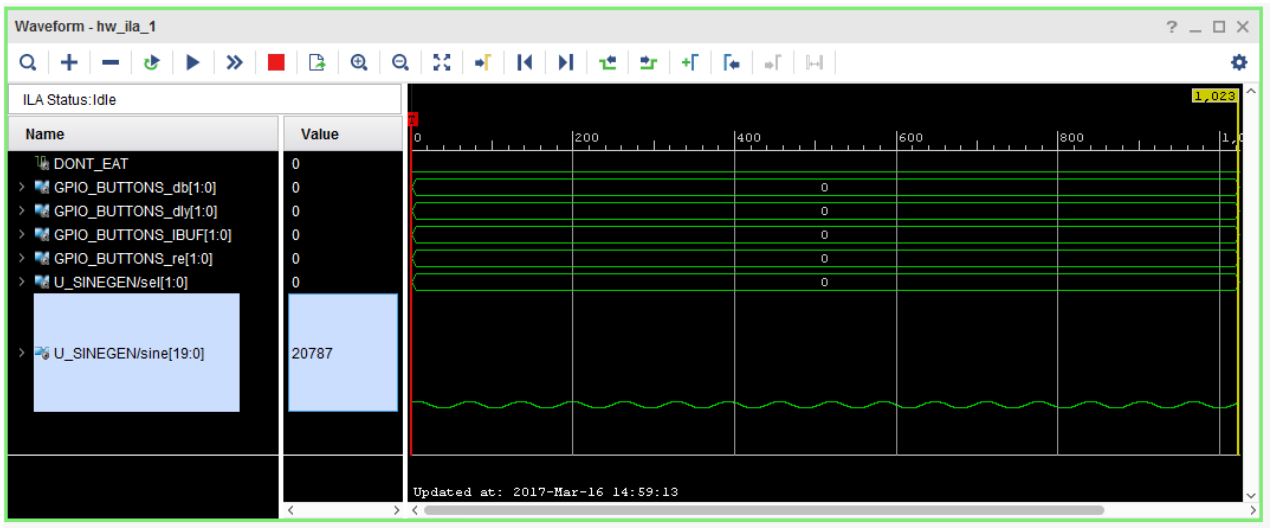

Figure 60: Output Sine Wave Displayed in Analog Format - High Frequency 2

Correcting Display of the Sine Wave

To view the mid, and low frequency output sine waves, perform the following steps:

16. Cycle the sine wave sequential circuit by pressing the GPIO_SW_E push button as shown in the following figure.

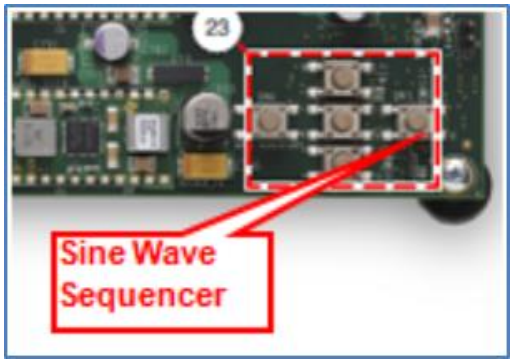

Figure 61: Sine Wave Sequencer Push Button

17. Click **Run Trigger Immediately** again to see the new sine selected sine wave. You should see the mid frequency as shown in the following figure. Notice that the `sel` signal also changed from 0 to 1 as expected.

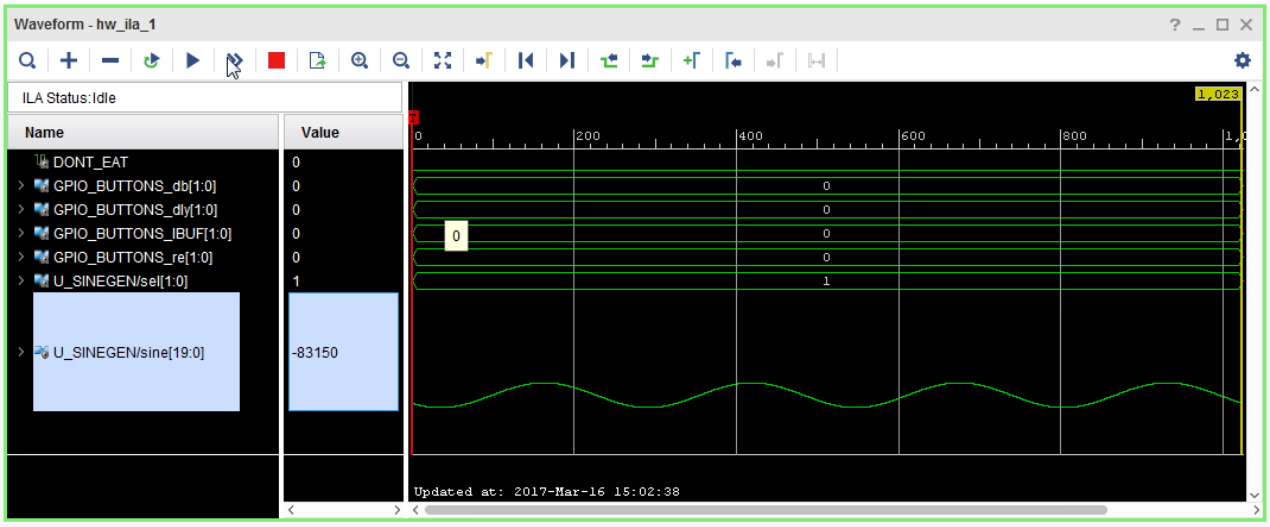

Figure 62: Output Sine Wave Displayed in Analog Format - Mid Frequency

18. Repeat step 17 and 18 to view other sine wave outputs.

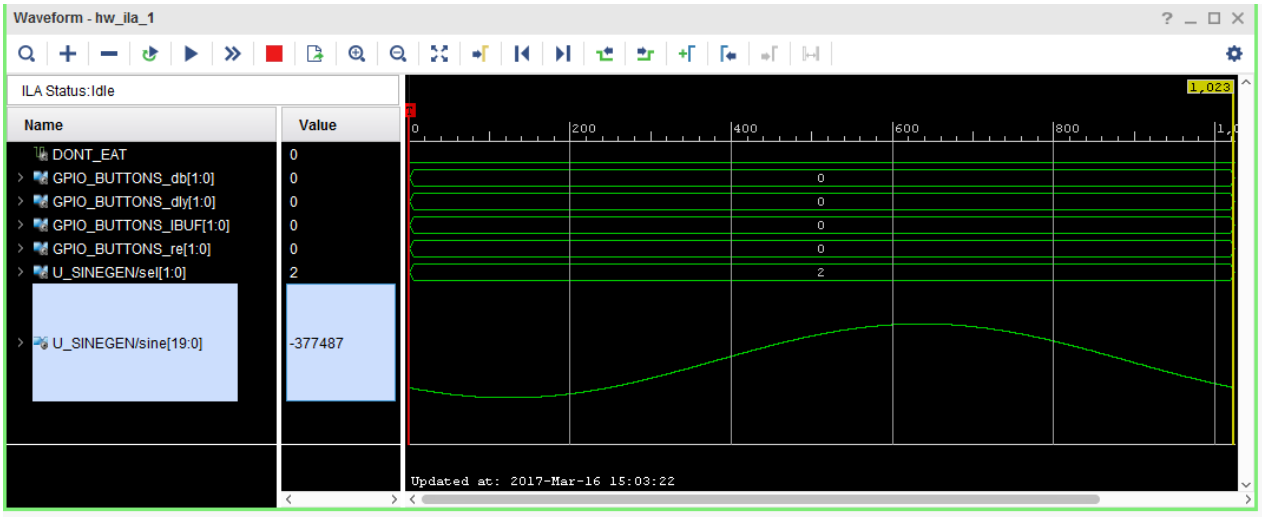

Figure 63: Output Sine Wave Displayed in Analog Format - Low Frequency

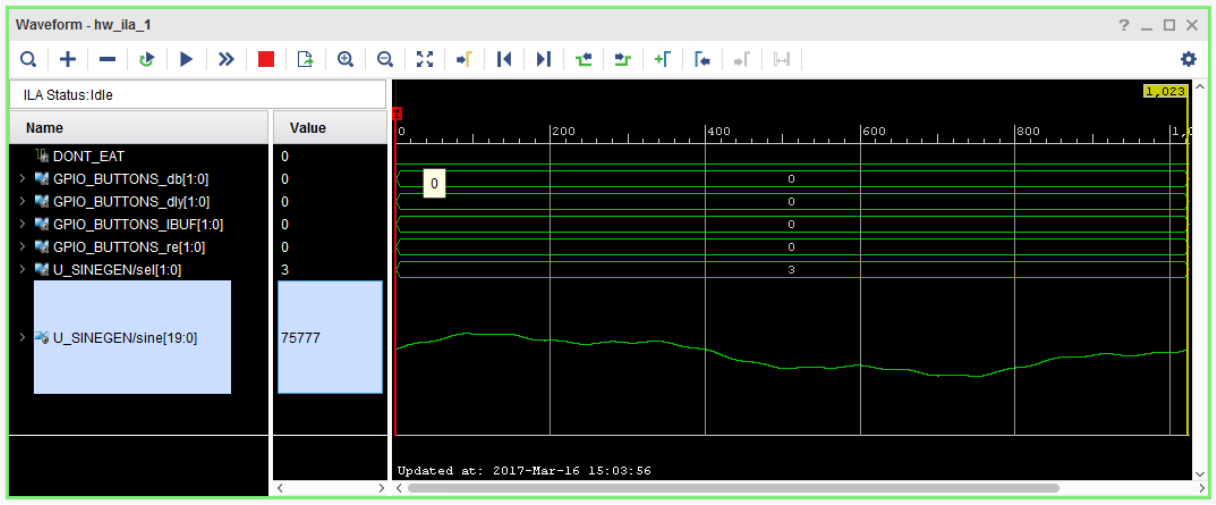

Figure 64: Output Sine Wave Displayed in Analog Format - Mixed Frequency

***Note**: As you sequence through the sine wave selections, you may notice that the LEDs do not light up in the expected order. You will debug this in the next section of this tutorial. For now, verify for each LED selection, that the correct sine wave displays. Also, note that the signals in the **Waveform** window have been re-arranged in the previous three figures.*

Step 2: Debugging the Sine Wave Sequencer State Machine (Optional)

As you corrected the sine wave display, the LEDs might not have lit up in sequence as you pressed the Sine Wave Sequencer button. With each push of the button, there should be a single, cycle-wide pulse on the GPIO_BUTTONS_re[1] signal. If there is more than one, the behavior of the LEDs becomes irregular. In this section of the tutorial, use Vivado logic analyzer to probe the sine wave sequencer state machine, and to view and repair the root cause of the problem.

Before starting the actual debug process, it is important to understand more about the sine wave sequencer state machine.

Sine Wave Sequencer State Machine Overview

The sine wave sequencer state machine selects one of the four sine waves to be driven onto the sine signal at the top-level of the design. The state machine has one input and one output. The following figure shows the schematic elements of the state machine. Refer to this diagram as you read the following description and as you perform the steps to view and repair the state machine glitch.

- The input is a scalar signal called "button". When the button input equals "1", the state machine advances from one state to the next.
- The output is a 2-bit signal vector called "Y", and it indicates which of the four sine wave generators is selected.

The input signal button connects to the top-level signal GPIO_BUTTONS_re[1], which is a low-to-high transition indicator on the Sine Wave Sequencer button. The output signal Y connects to the top-level signal, sineSel, which selects the sine wave.

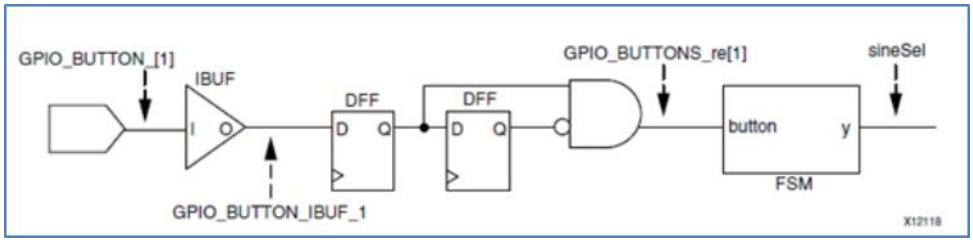

Figure 65: Sine Wave Sequencer Button Schematic

Viewing the State Machine Glitch

You cannot troubleshoot the issue identified above by connecting a debug probe to the GPIO_BUTTON [1] input signal itself. The GPIO_BUTTON [1] input signal is a PAD signal that is not directly accessible from the FPGA fabric. Instead, you must trigger on low-to-high transitions (rising edges) on the GPIO_BUTTON_IBUF signal, which is connected to the output of the input buffer of the GPIO_BUTTON [1] input signal.

As described earlier, the glitch reveals itself as multiple low-to-high transitions on the GPIO_BUTTONS_1_IBUF signal, but it occurs intermittently. Because it could take several button presses to detect it, you will now set up the Vivado logic analyzer tool to Repetitive Trigger Run Mode. This setting makes it easier to repeat the button presses and look for the event in the Waveform viewer.

1. Open the **Debug Probes** window if not already open by selecting **Window > Debug Probes** from the Vivado main menu.
2. In the **ILA Core Properties** window scroll down to the link marked **To view editable ILA Properties: Open ILA Dashboard** and set the following:
   a. **Trigger Mode** to **BASIC_ONLY**
   b. **Capture Mode** to **BASIC**
   c. **Window Data Depth** to **1024**
   d. **Trigger position** to **512**
   e. Press the **+** button in the **Trigger Setup** window and add probe **GPIO_BUTTONS_IBUF_1**. Change the **Value** field to **RX** by selecting the value **RX** in the **Value** field, as shown in the following figure.

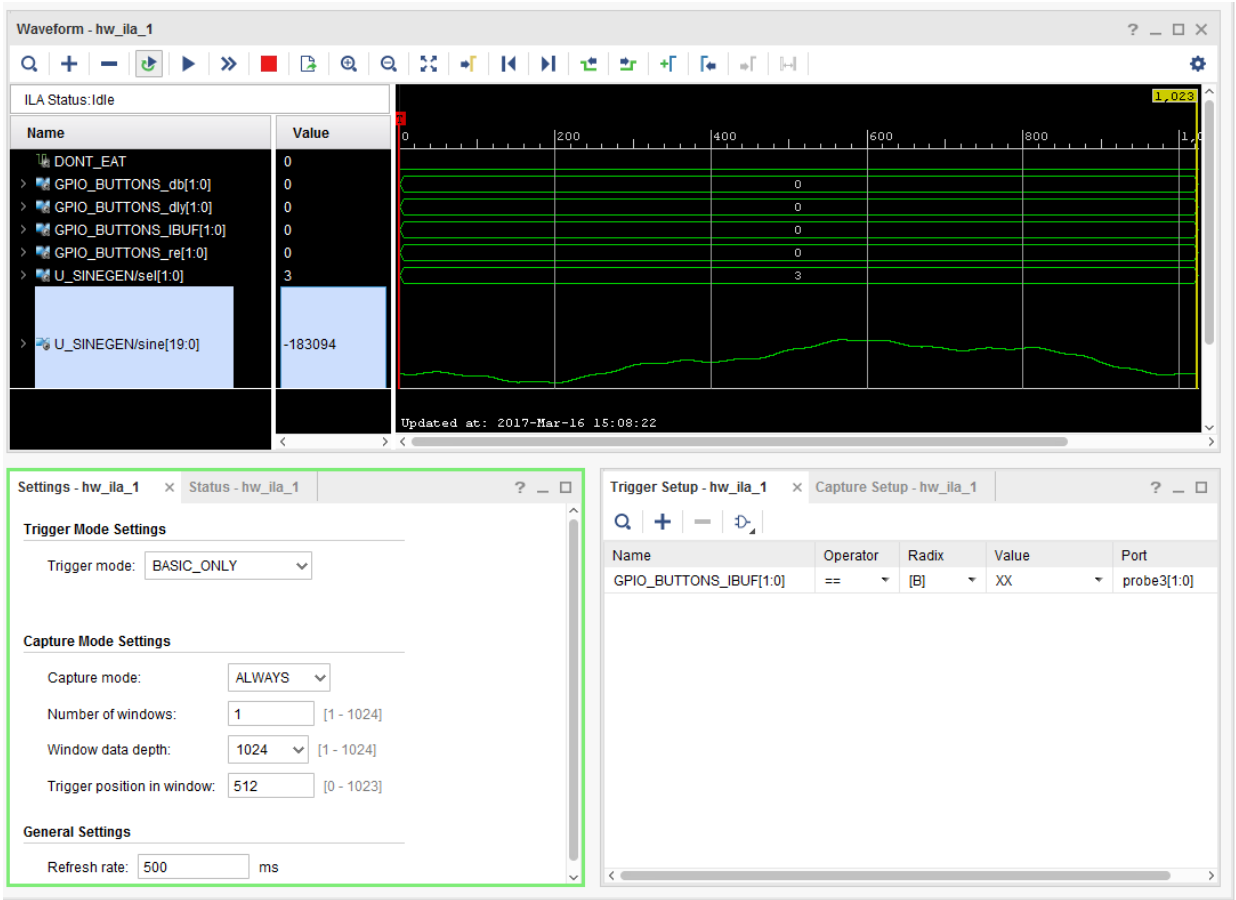

Figure 66: Trigger Setup Window

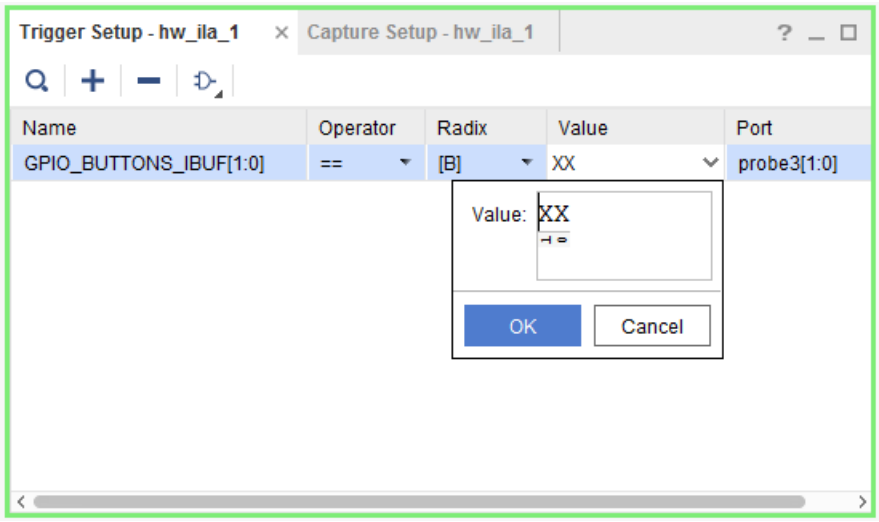

Figure 67: Setting Trigger Conditions

CAUTION! *For different labs the GPIO_BUTTONS_IBUF may show up differently. This may show up as two individual bits or two bits lumped together in a bus. Ensure that you are using bit 1 of this bus to set up your trigger condition. For example in case of a two-bit bus, you will set the* ***Value*** *field in the* ***Compare Value*** *dialog box to* ***RX****.*

3. Select **Enable Auto Re-trigger** mode on the ILA debug core as shown below.

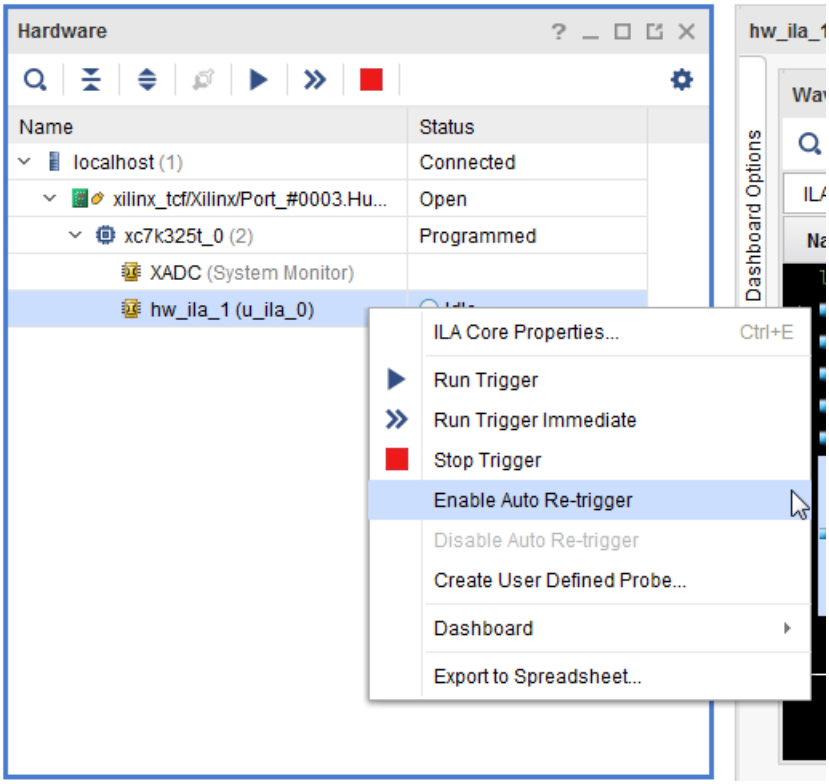

Figure 68: Enable Auto Re-trigger

CAUTION! *The **ILA properties** window may look slightly different for different labs.*

When you issue a **Run Trigger** or a **Run Trigger Immediate** command after setting the **Auto Retrigger** mode, the ILA core does the following repetitively until you disable the **Auto Retrigger** mode option.

- Arms the trigger.
- Waits for the trigger.
- Uploads and displays waveforms.

4. On the KC705 board, press the Sine Wave Sequencer button until you see multiple transitions on the GPIO_BUTTONS_1_IBUF signal (this could take 10 or more tries). This is a visualization of the glitch that occurs on the input. An example of the glitch is shown in the following two figures.

CAUTION! *You may have to repeat the previous two steps repeatedly to see the glitch. Once you can see the glitch, you may observe that the signal glitches are not at exactly the same location as shown in the figure below.*

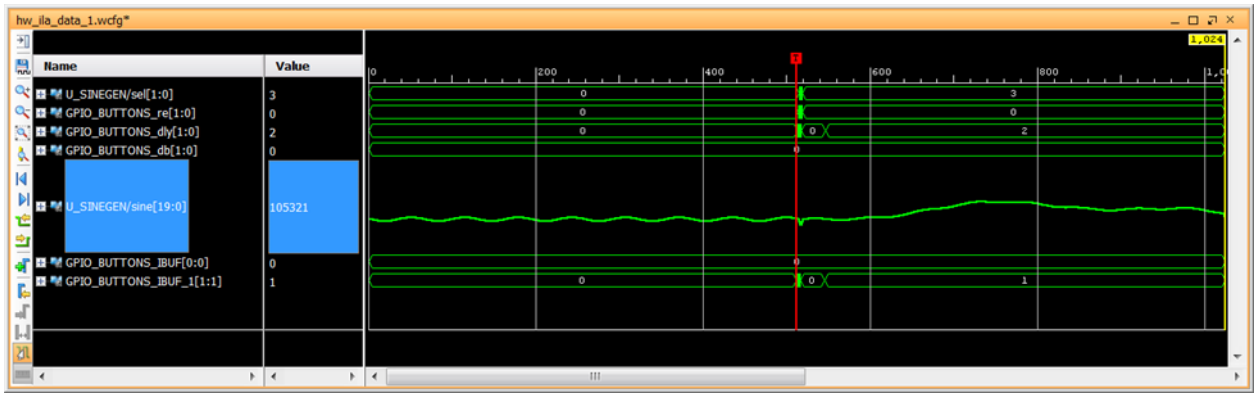

Figure 69: GPIO_BUTTONS_BUF1 Signal Glitch

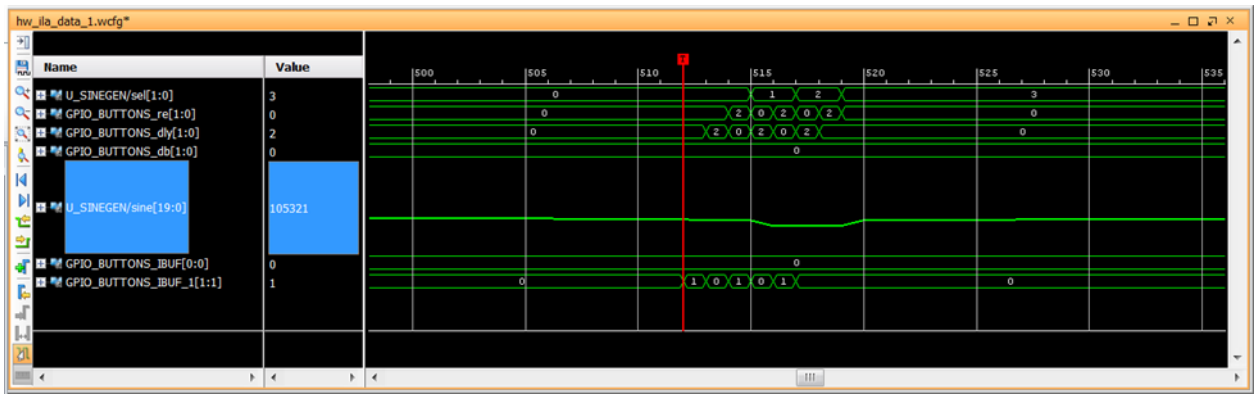

Figure 70: GPIO Buttons_1_re Signal Glitch magnified

Fixing the Signal Glitch and Verifying the Correct State Machine Behavior

The multiple transition glitch or "bounce" occurs because the mechanical button is making and breaking electrical contact just as you press it. To eliminate this signal bounce, a "de-bouncer" circuit is required.

1. Enable the de-bouncer circuit by setting DIP switch position on the KC705 board (labeled De-bounce Enable in Figure 1: KC705 Board Showing Key Components) to the **ON** or **UP** position.
2. Enable the **Auto-Retrigger** mode on the ILA debug core and click **RunTrigger** on the ILA core, and:
    - Ensure that you no longer see multiple transitions on the GPIO_BUTTON_re[1] signal on a single press of the Sine Wave Sequencer button.
    - Verify that the state machine is working correctly by ensuring that the sineSel signal transitions from 00 to 01 to 10 to 11 and back to 00 with each successive button press.

Verifying the VIO Core Activity (Only applicable to Lab 3)

1. From the **Program and Debug** section in **Flow Navigator**, click **Open Hardware Manager**.

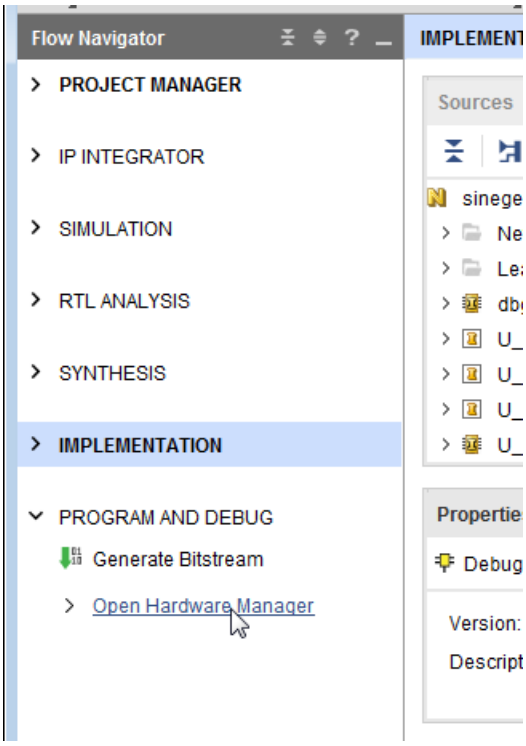

Figure 71: Open Hardware Manager

The **Hardware Manager** window opens.

2. Click **Open a new hardware target**.

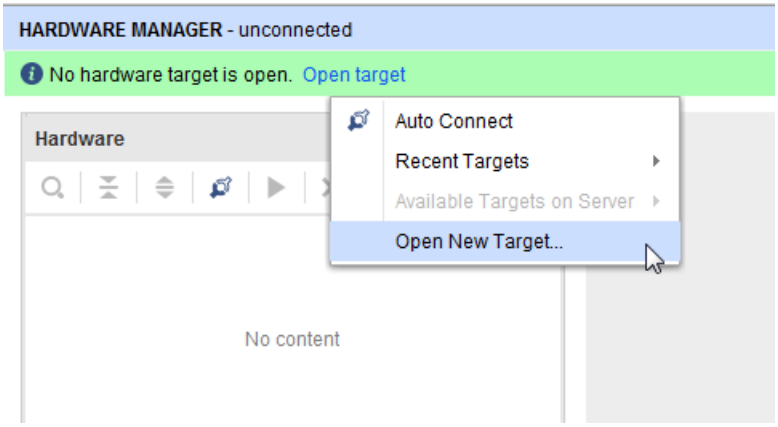

Figure 72: Connect to a New Hardware Target

3. The **Open New Hardware Target** wizard opens. Click **Next**.
4. In the **Hardware Server Settings** page, type the name of the server (or select **Local server** if the target is on the local machine) in the **Connect to** field.
5. Ensure that you are connected to the right target by selecting the target from the **Hardware Targets** page. If there is only one target, that target is selected by default. Click **Next**.
6. In the **Set Hardware Target Properties** page, click **Next**.
7. In the **Open Hardware Target Summary** page, verify that all the information is correct, and click **Finish**.
8. Program the device by selecting and right-clicking the device in the **Sources** window and then selecting **Program Device**.

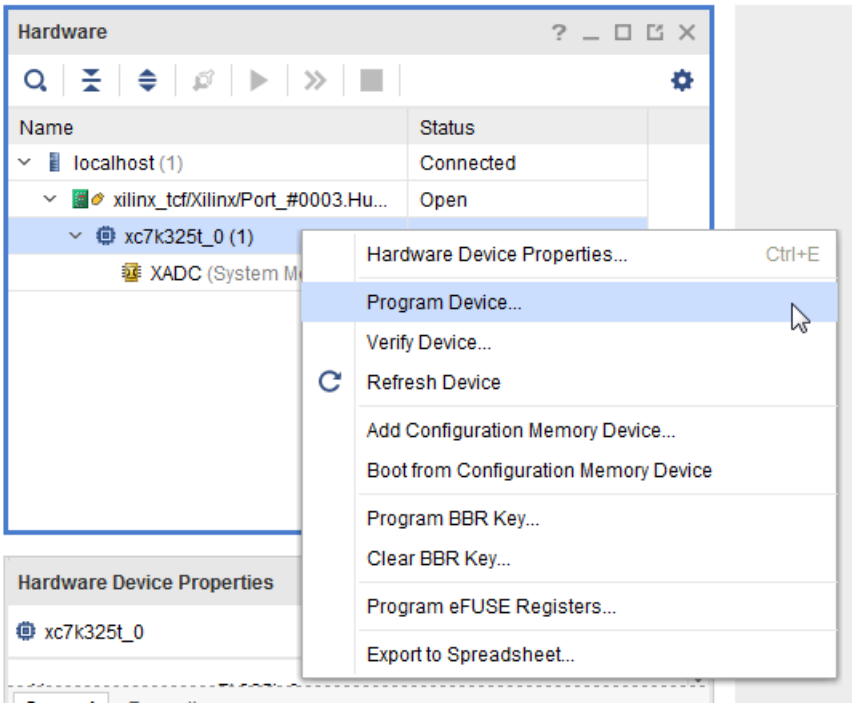

Figure 73: Program FPGA

9. In the **Program Device** dialog box, ensure that the bit file to be programmed is correct. Click **OK**.

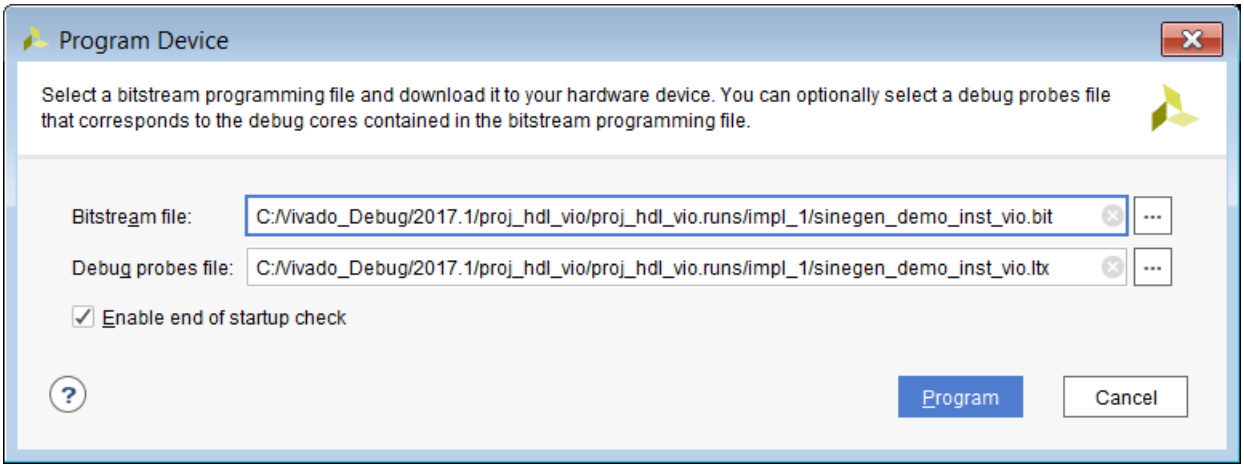

Figure 74: Program Device with the sinegen_demo_inst_vio.bit File

10. After the FPGA device is programmed, you see the VIO and the ILA core in the **Hardware** window.

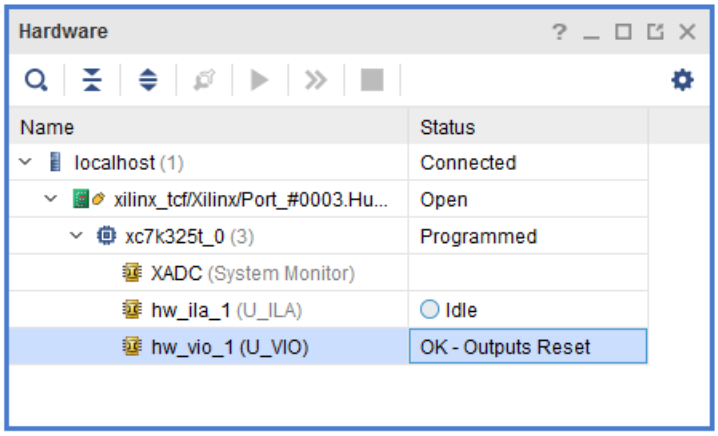

Figure 75: The ILA and VIO Cores in the Hardware Window

You now have a debug dashboard for the ILA core as shown in the following figure.

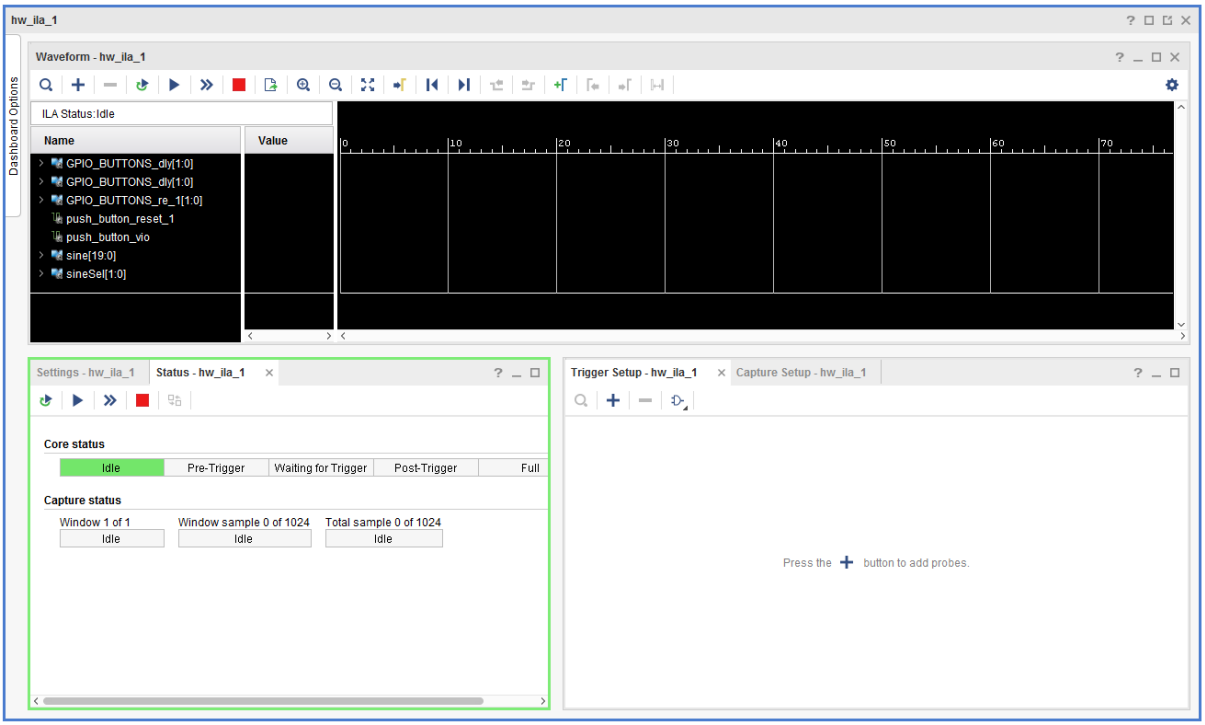

Figure 76: ILA Core and VIO Core Dashboards

11. Click **Run Trigger Immediate** to capture the data immediately.

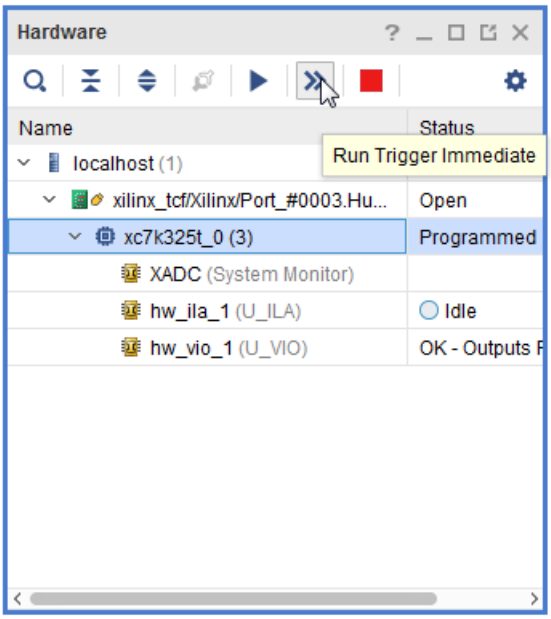

Figure 77: Run Trigger Immediate

12. Make sure that there is activity on the sine [19:0] signal.
13. Select the sine signal in the **Waveform** window, right-click and select **Waveform Style > Analog**.
14. Select the sine signal in the **Waveform** window again, right-click and select **Radix > Signed Decimal**. You should be able to see the sine wave in the **Waveform** window.

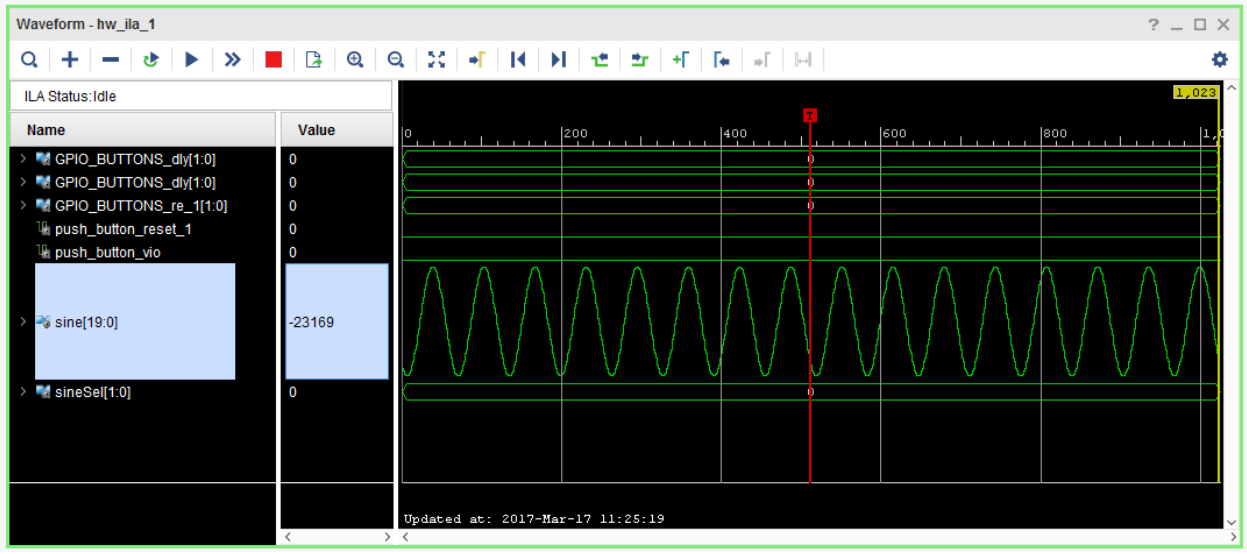

Figure 78: Sine Wave after Modifying the Properties of the sine [19:0] Signal

15. Instead of using the GPIO_SW push button to cycle through each different sine wave output frequency, you are going to use the virtual "push_button_vio" toggle switch from the VIO core.

16. You can now customize the ILA dashboard options to include the VIO window. This allows you to toggle the VIO output drivers and observe the impact on the ILA waveform window all in one dashboard. Slide out the **Dashboard Options** window.

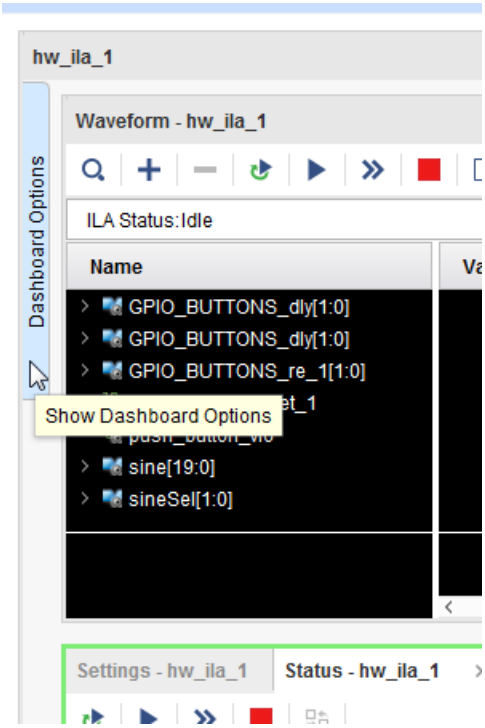

Figure 79: Invoking Dashboard Options

17. Add the VIO window to the ILA dashboard by selecting **hw_vio_1**.

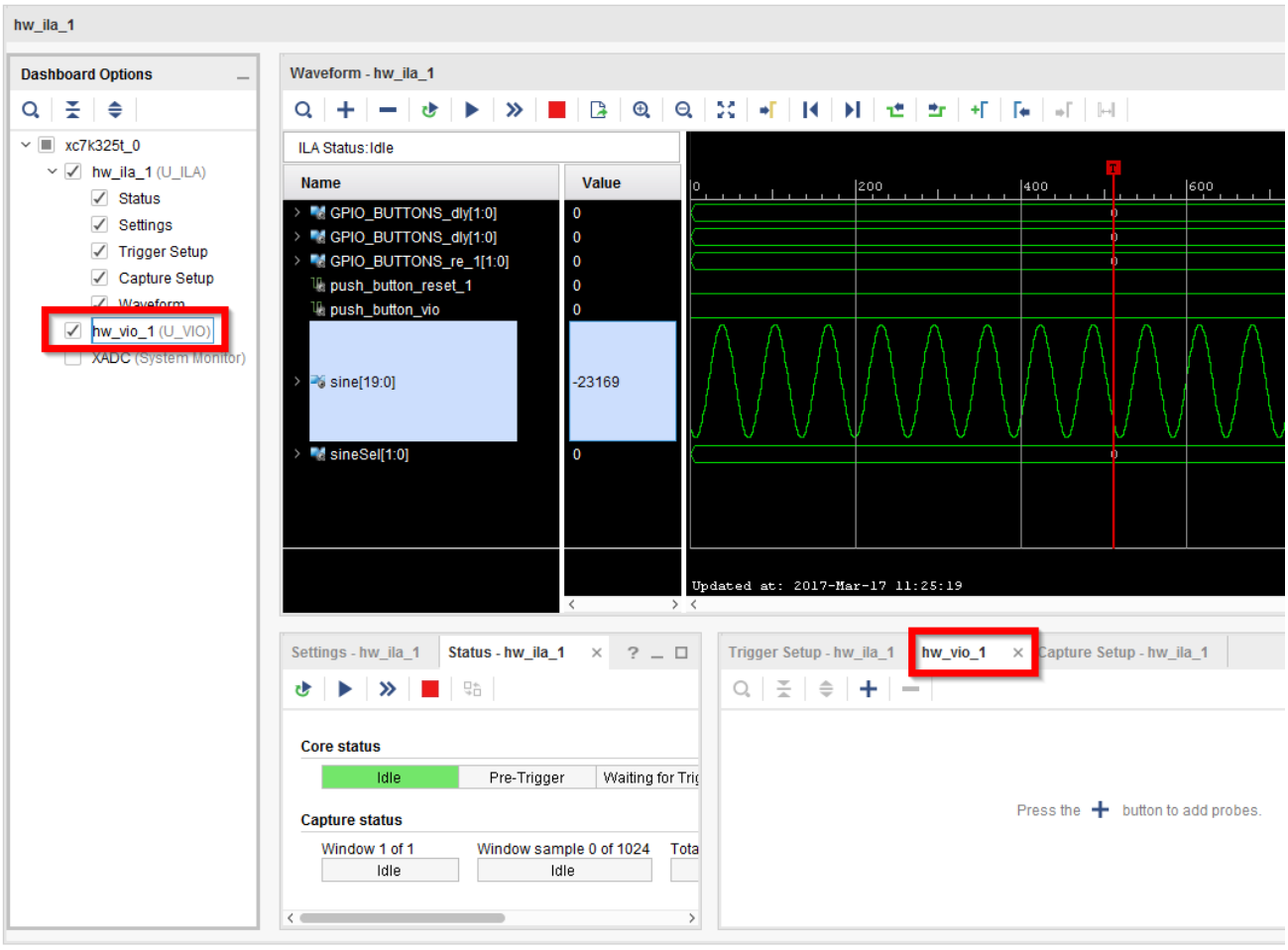

Figure 80: Dashboard Options Adding VIO

Note: *The ILA dashboard now contains the VIO window as well.*

18. Adjust the **Trigger Setup – hw_ila_1** window and the **hw_vio_1** window so that they are side by side as shown in the following figure.

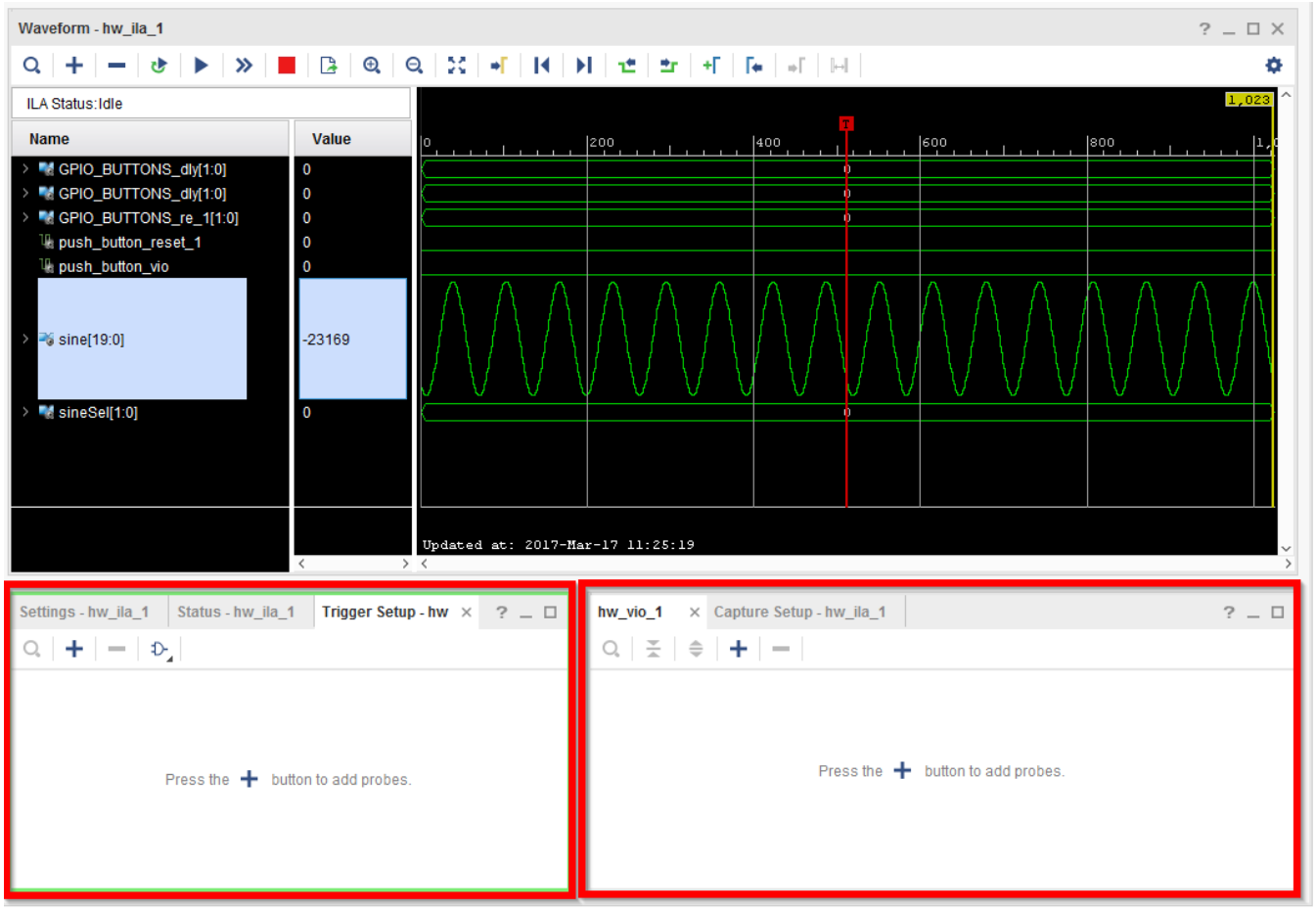

Figure 81: ILA Basic Trigger Window and VIO Window Adjustment

19. In the **hw_vio_1** window, select the "**+**" button, and select all the probes under hw_vio_1.
20. Click **OK**.

***Note*:** The initial values of all the probes.

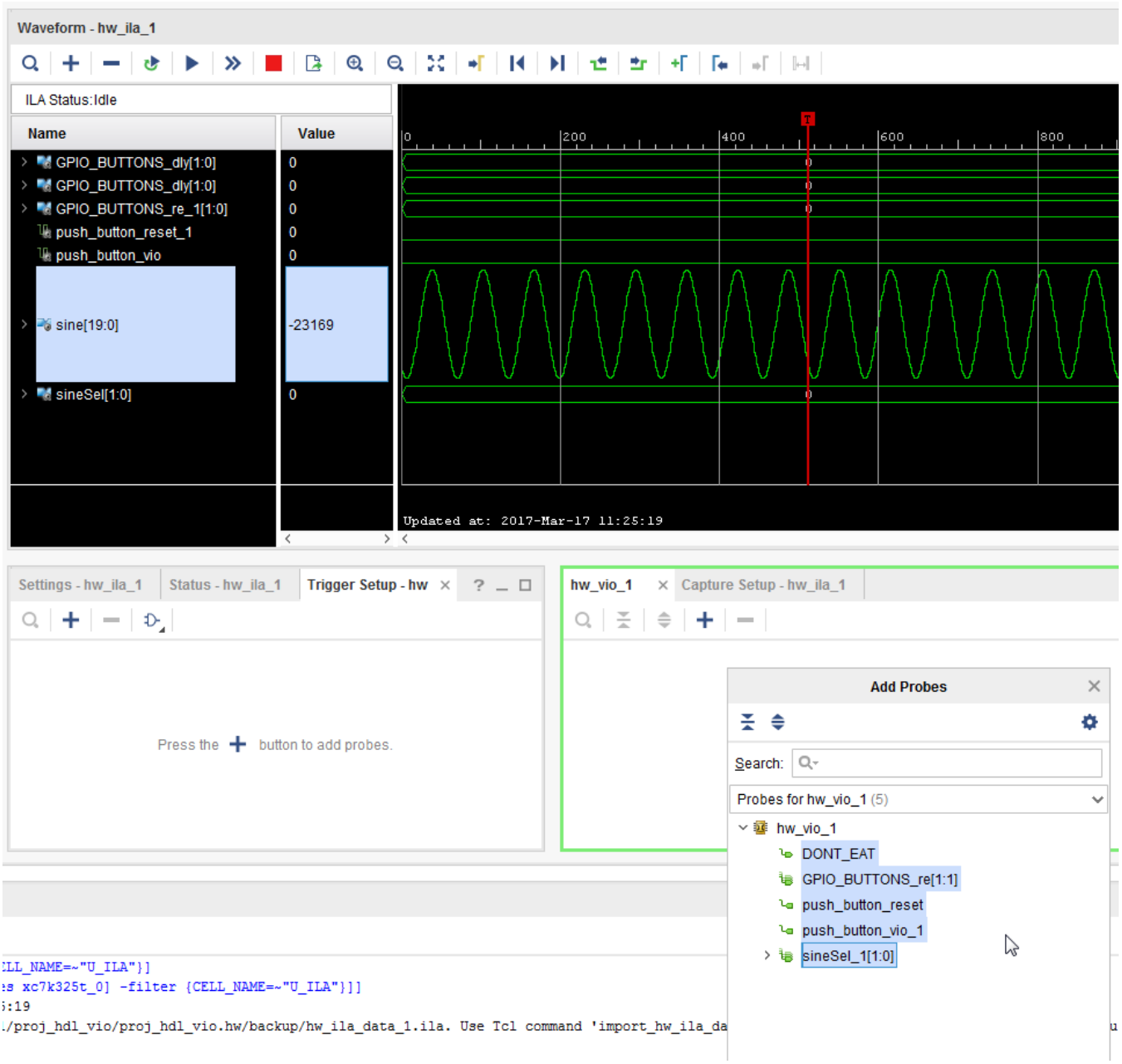

Figure 82: VIO Add Probes Window

21. Note the values on all probes in the **hw_vio_1** window.

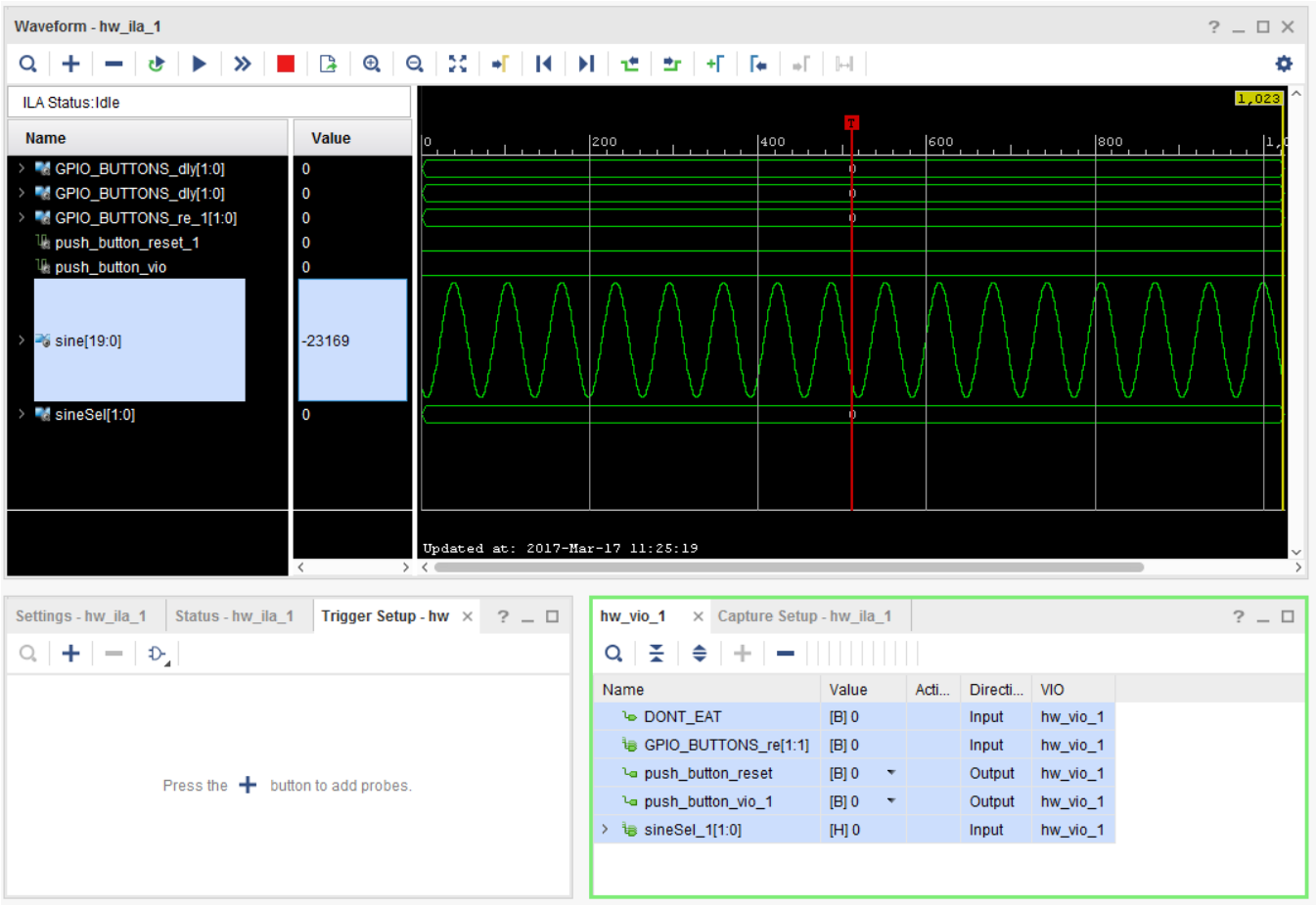

Figure 83: VIO Probes Added to hw_vio_1 Window

22. Set the push_button_reset output probe by right-clicking **push_button_reset** and select **Toggle Button**.

    This will toggle the output driver from logic from '0' to '1' to '0' as you click. It is similar to the actual push button behavior, though there is no bouncing mechanical effect as with a real push button switch.

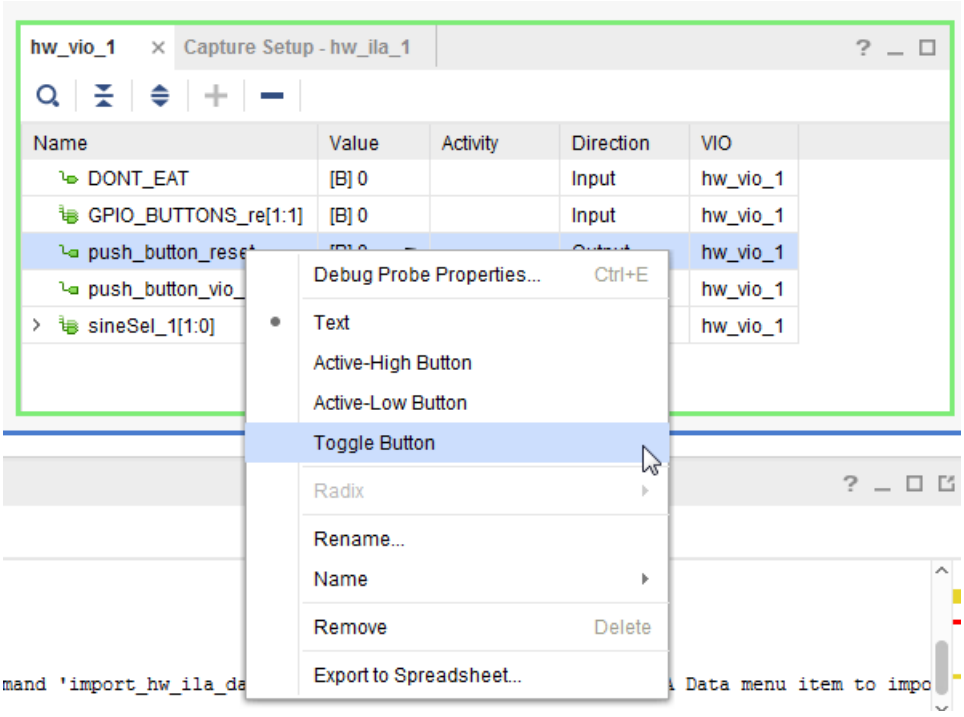

Figure 84: Toggle the push_button_reset Signal

The **Value** field for push_button_reset is highlighted.

23. Click in the **Value** field to change its value to **1**.

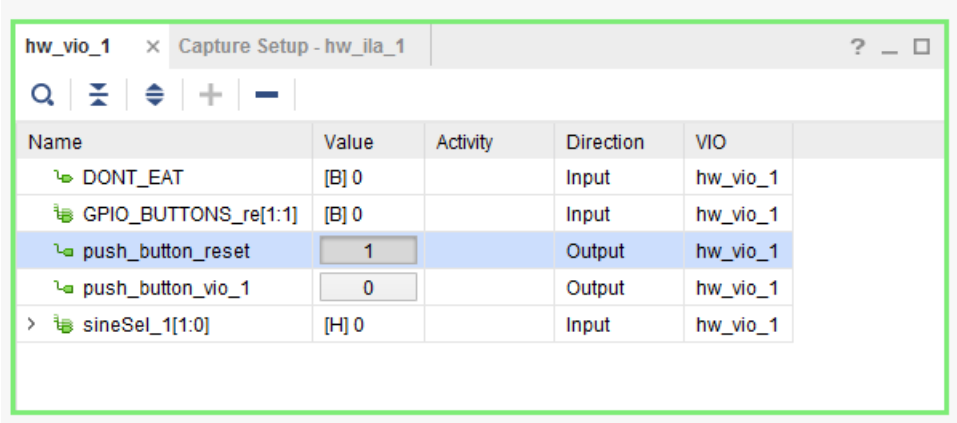

Figure 85: Toggle the Value of push_button_reset

24. Follow the step above to change the push_button_vio to Toggle button as well.

25. Set these two bits of the "sineSel" input probe by right-clicking **PROBE_IN0[0] and PROBE_IN0[1]** and selecting **LED**.

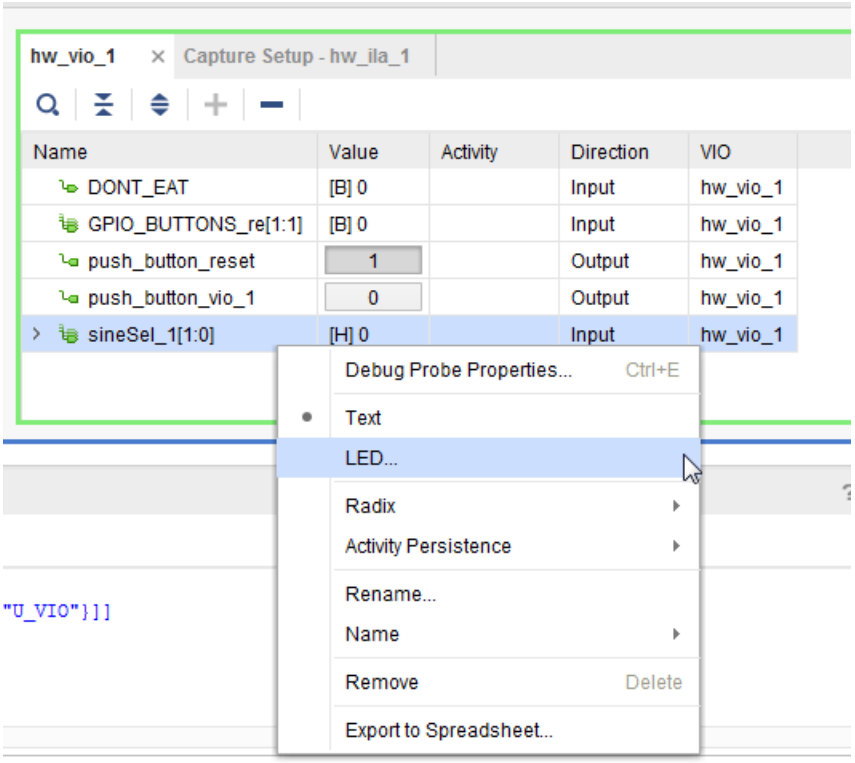

Figure 86: Change sineSel to LED

26. In the **Select LED Colors** dialog box, pick the **Low Value Color** and the **High Value Color** of the LEDs as you desire and click **OK**.

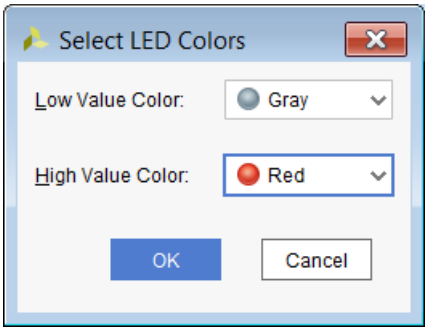

Figure 87: Pick the Low Value and High Value Color of the LEDs

27. When finished, your **VIO Probes** window in the Hardware Manager should look similar to the following figure.

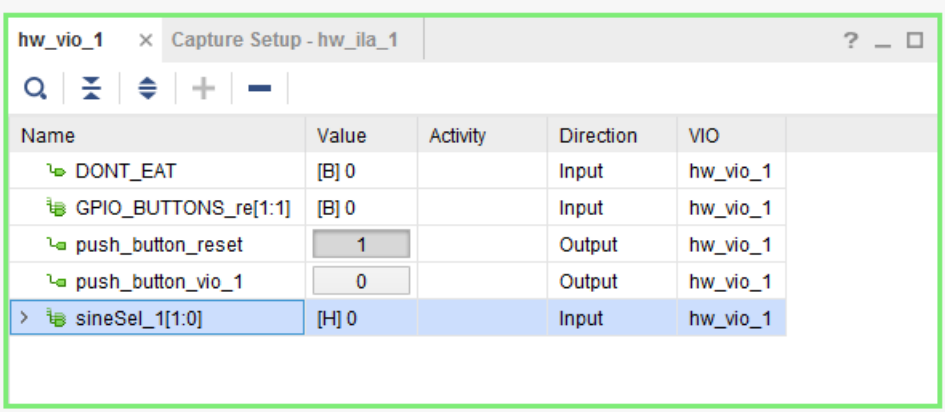

Figure 88: Input and Output VIO Signals Displayed

28. To cycle through each different sine wave output frequency using the virtual "push_button_vio" from the VIO core, perform the following simple steps:

    a. Toggle the value of the "push_button_vio" output driver from 0 to 1 to 0 by clicking on the logic displayed under the **Value** column. You will notice the sineSel LEDs changed accordingly – 0, 1, 2, 3, 0, etc…

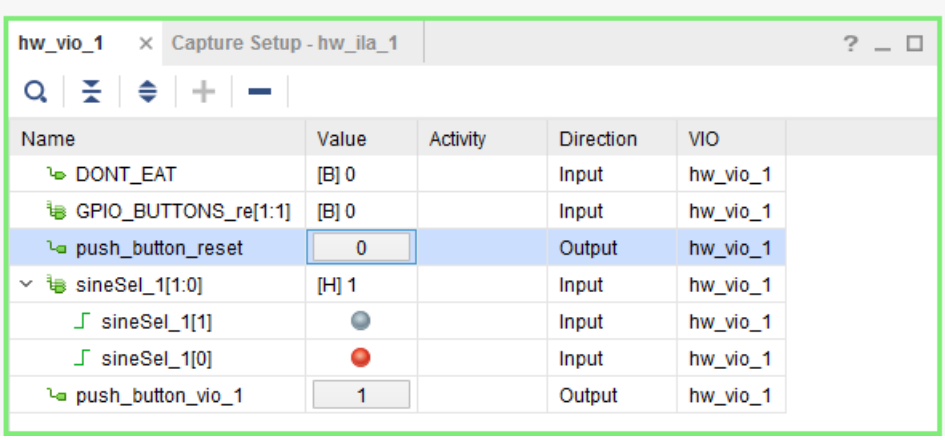

Figure 89: Toggle push_button_reset

    b. Click **Run Trigger** for hw_ila_1 to capture and display the selected sine wave signal from the previous step.

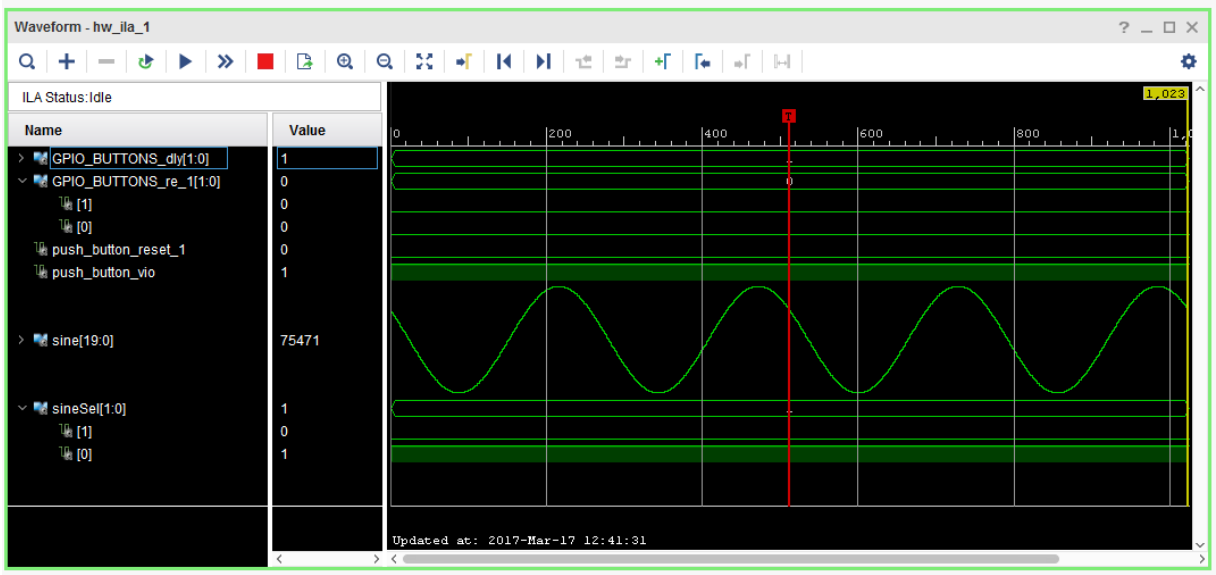

Figure 90: Run Trigger for hw_ila_1

Lab 6: Using ECO Flow to Replace Debug Probes Post Implementation

This simple tutorial shows you how to replace nets connected to an ILA core in a placed and routed design checkpoint using the Vivado® Design Suite Engineering Change Order (ECO) flow.

TIP: *To learn more about using the ECO flow, refer to the Debugging Designs Post Implementation Chapter in the Vivado Design Suite User Guide: Programming and Debugging (UG908).*

1. Open the Vivado Design Suite, and select **File > Open Checkpoint.**

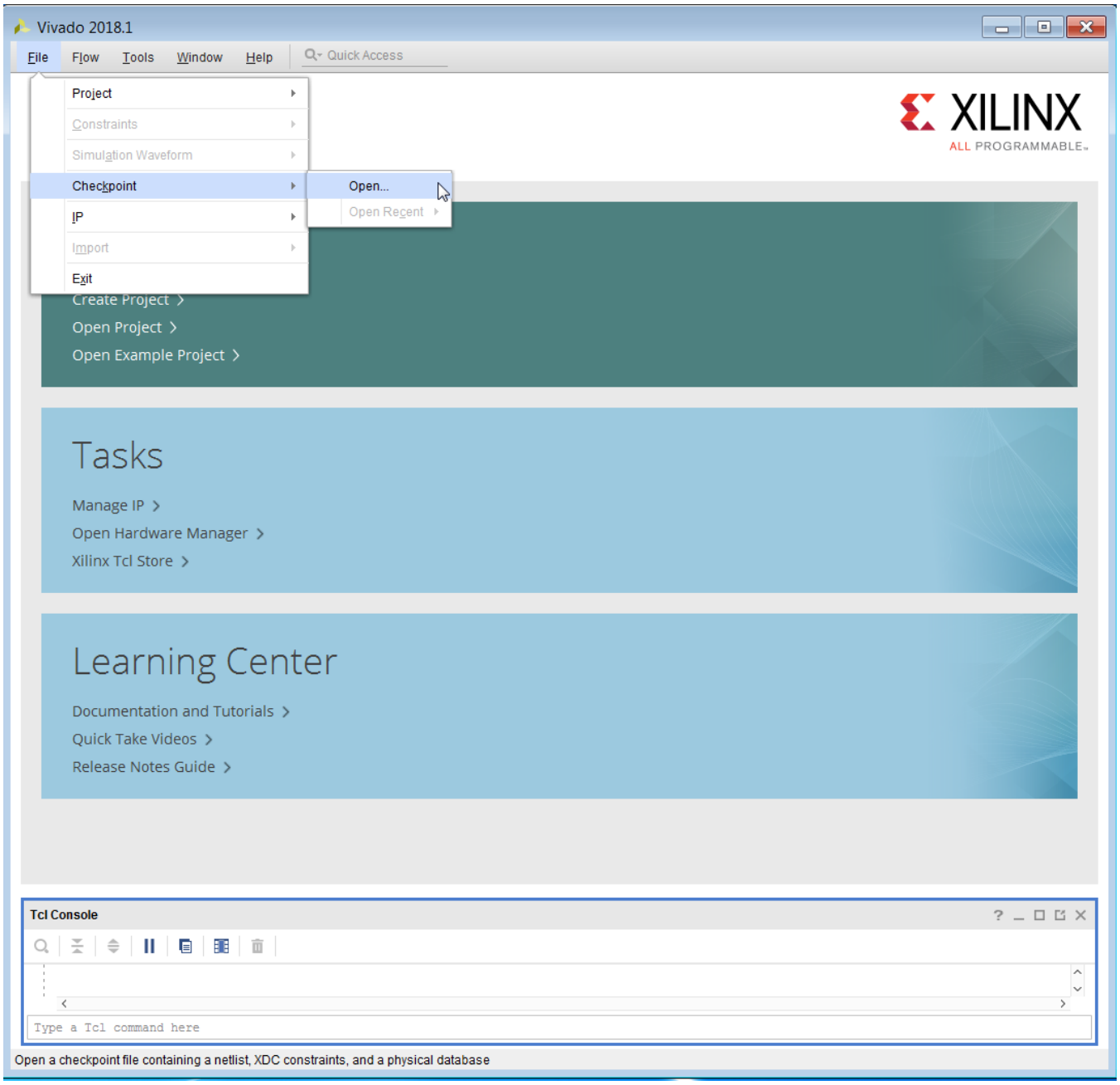

Figure 91: Opening a Checkpoint in Vivado IDE

2. Open the routed checkpoint that you created in Lab 2: Using the HDL Instantiation Method for Debugging a Design in Vivado.

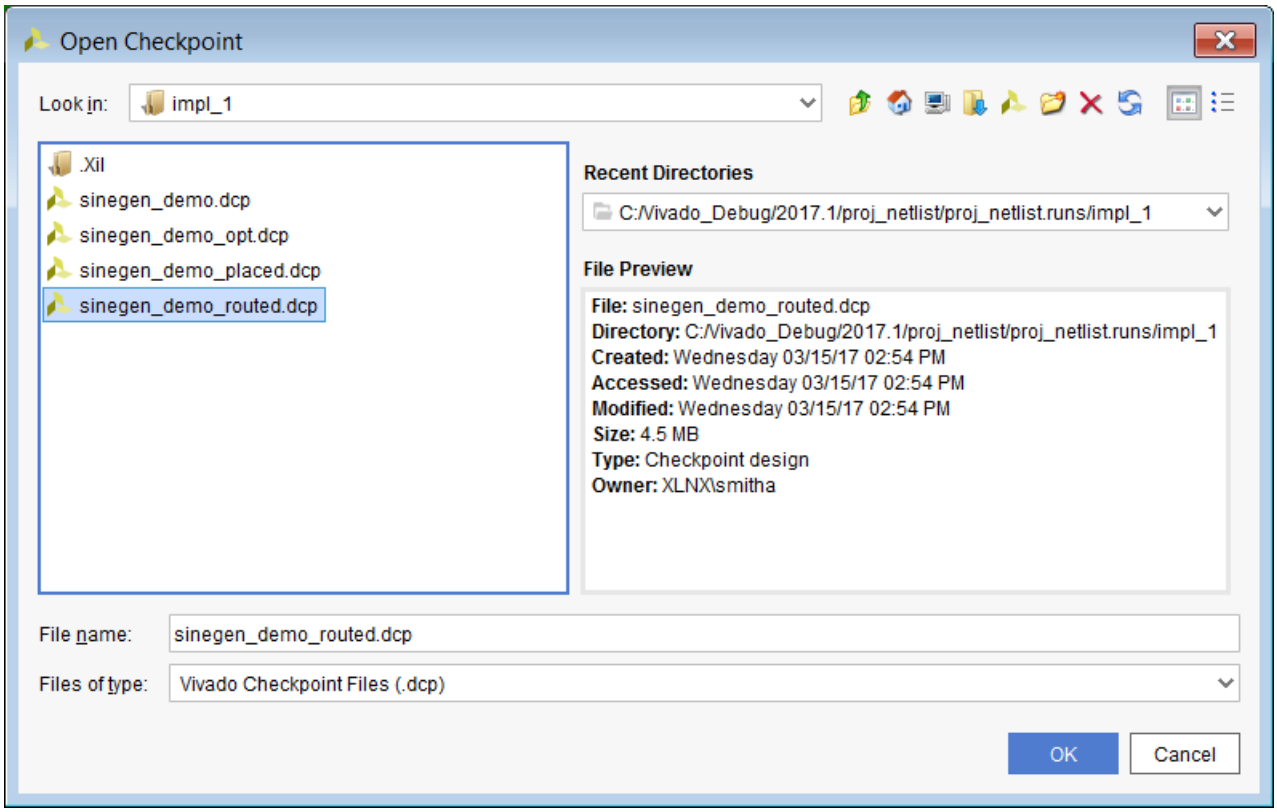

Figure 92: Open Checkpoint Dialog Box

Change the layout in the Vivado Design Suite toolbar dropdown to **ECO**.

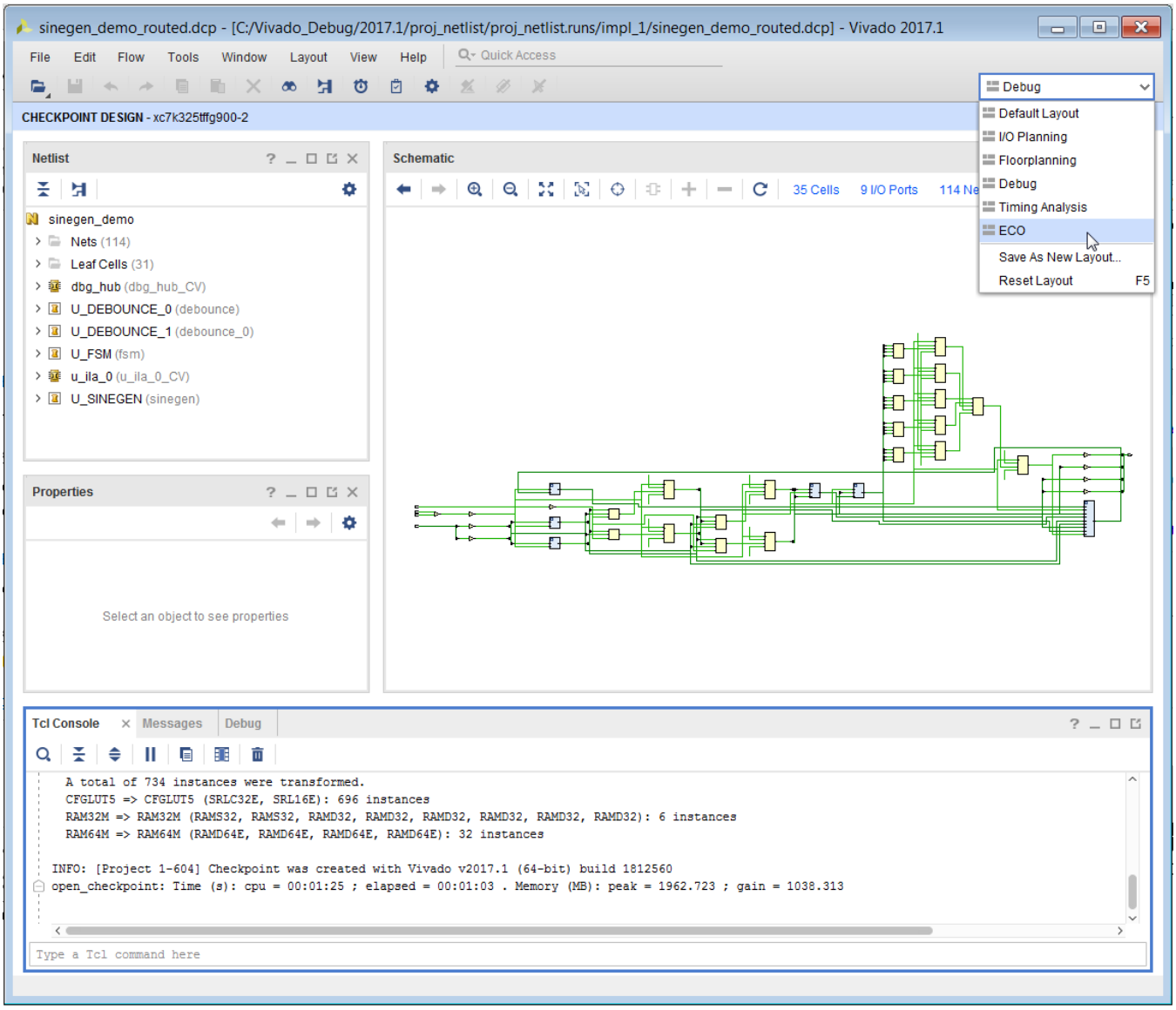

Figure 93: Change Layout to ECO

Note: *The* ***Flow Navigator*** *window now changes to* ***ECO Navigator*** *with a different set of options.*

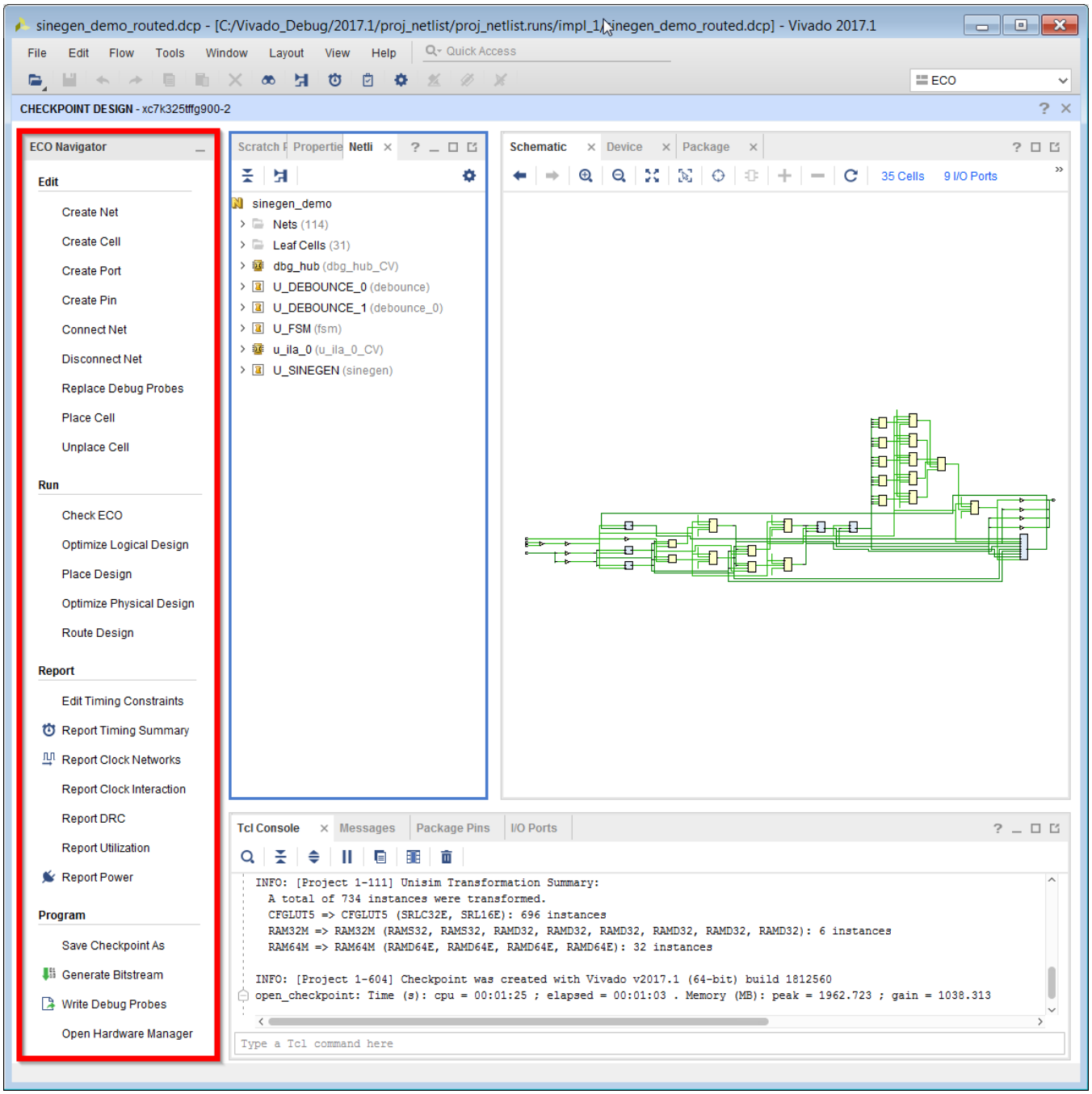

Figure 94: ECO Navigator Window

3. In the **ECO Navigator** window, click **Replace Debug Probes** to bring up the **Replace Debug Probes** dialog box. Note the Debug Hub and ILA cores in the design.

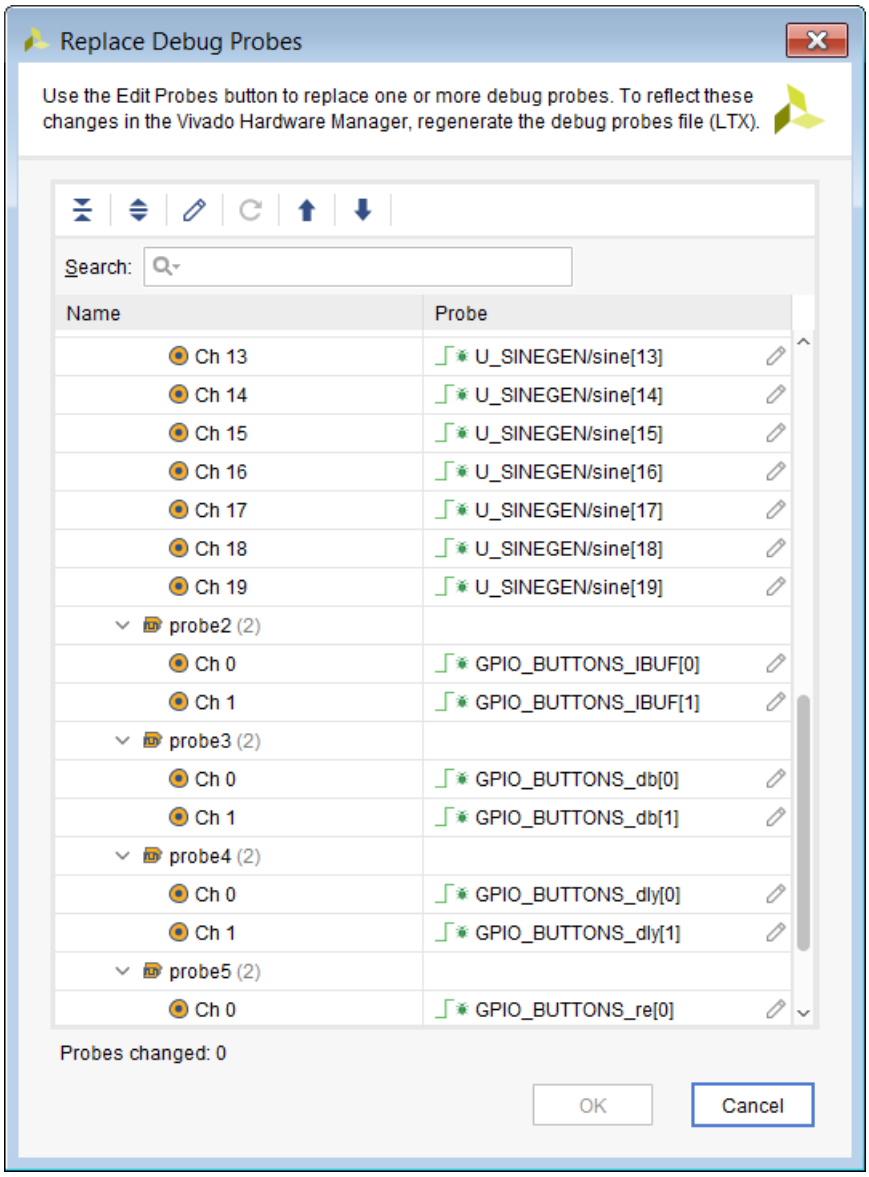

Figure 95: Replace Debug Probes Dialog Box

IMPORTANT: *Xilinx strongly recommends that you do not replace the clock nets associated with ILA and Debug Hub cores.*

4. In the **Replace Debug Probes** dialog box, highlight the probes whose nets you want to change. In this lab we will replace the **GPIO_BUTTONS_dly[0]** net that is being probed.
5. Click the **Edit Probes** button to the right of the **GPIO_BUTTONS_dly[0]** probe net to bring up the **Replace Debug Probes** dialog box.

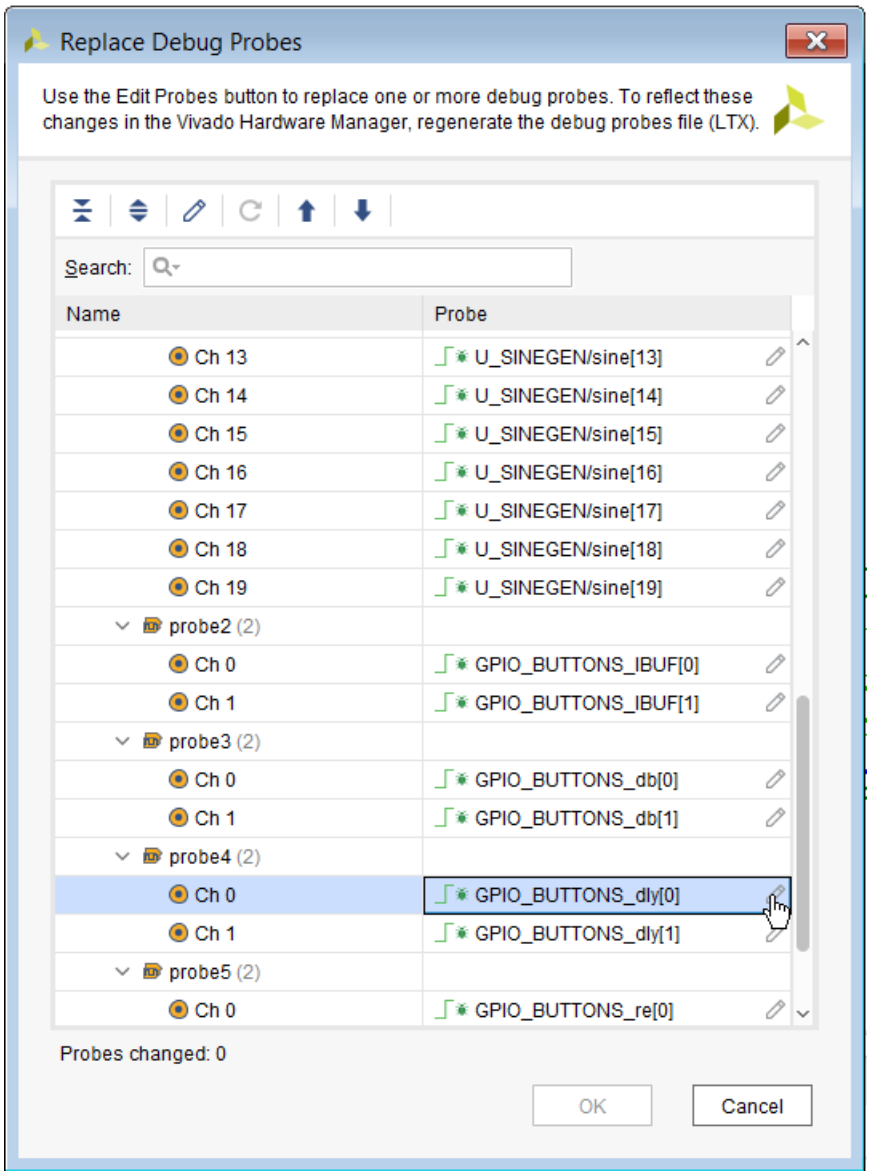

Figure 96: Edit Probes Button

6. In the **Choose Nets** dialog box, choose the **U_DEBOUNCE_0/clear** net to replace the existing **GPIO_BUTTONS_dly[0]** probe net. Click **OK**.

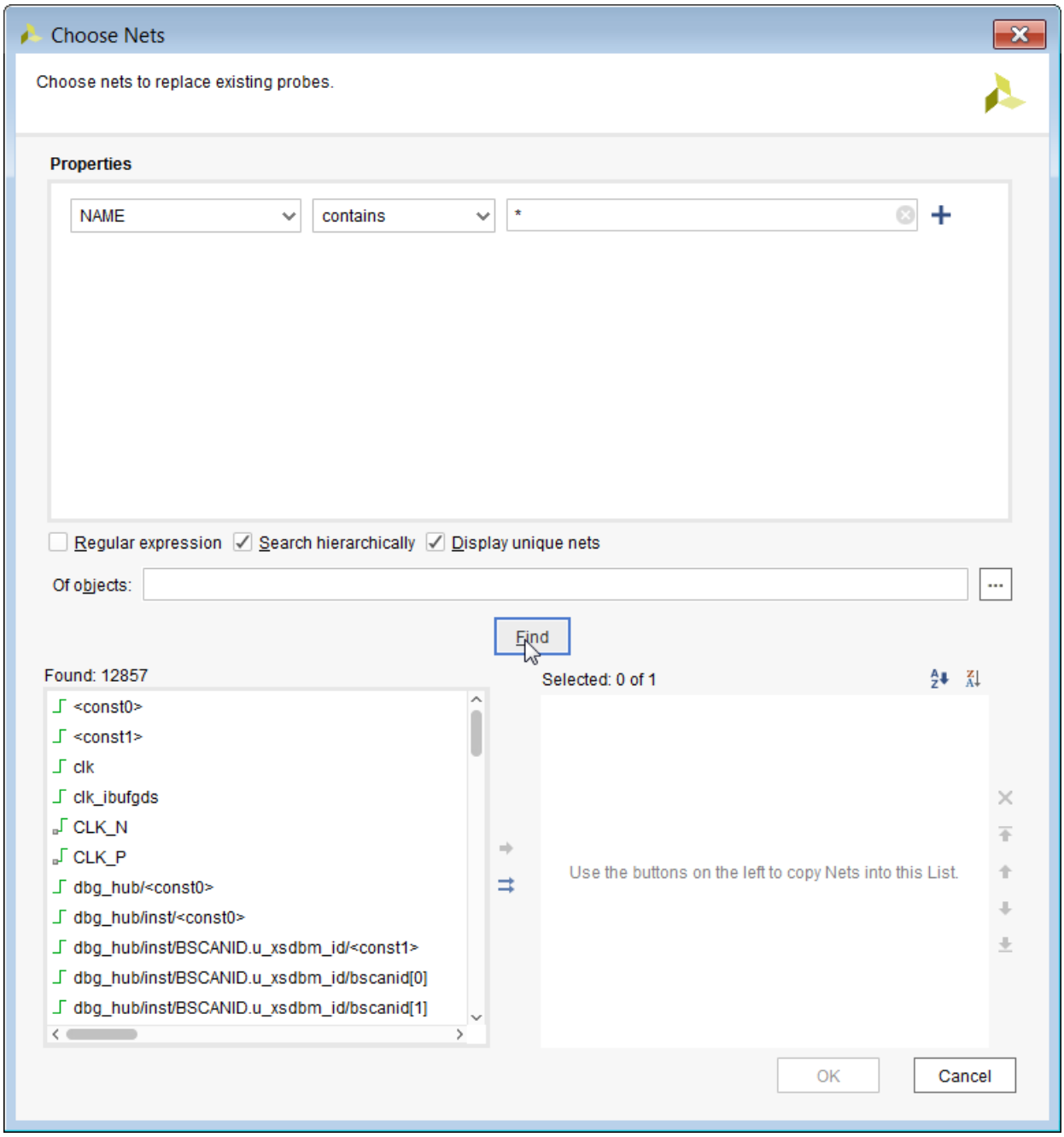

Figure 97: Choose Nets

7. Type for "*clear net" in the **Name** field and Click **Find**. Notice the **U_DEBOUNCE_0** net in the **Found nets** area. Select **U_DEBOUNCE_0/clear net** using the "**->**" arrow and click **OK**. The **U_DEBOUNCE_0/clear** net to replaces the existing **GPIO_BUTTONS_dly[0]** probe net.

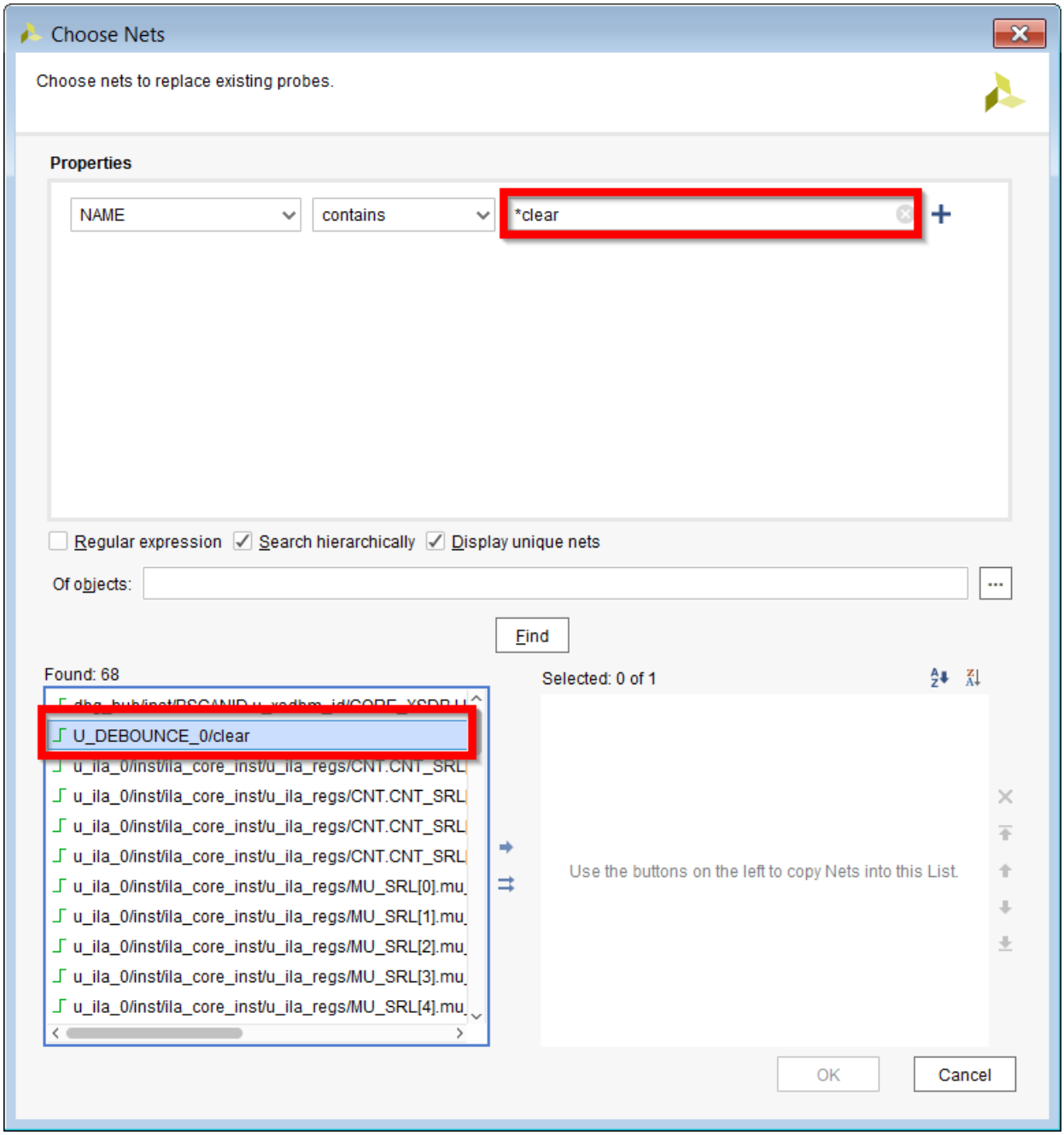

Figure 98: Choose Nets – Clear

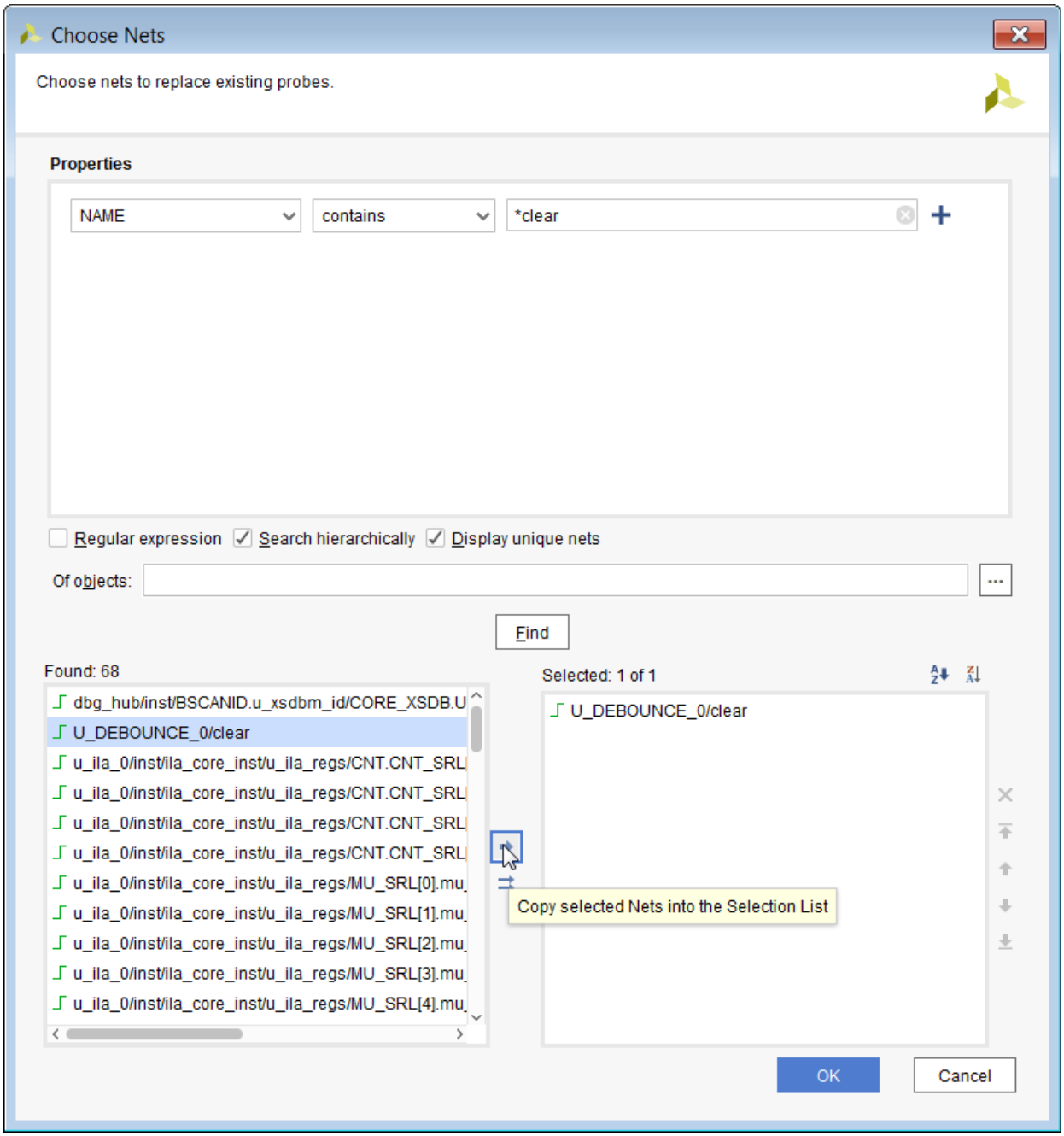

Figure 99: Choose Nets - Copy

8. Now click **OK** in the **Replace Debug Probes** dialog.

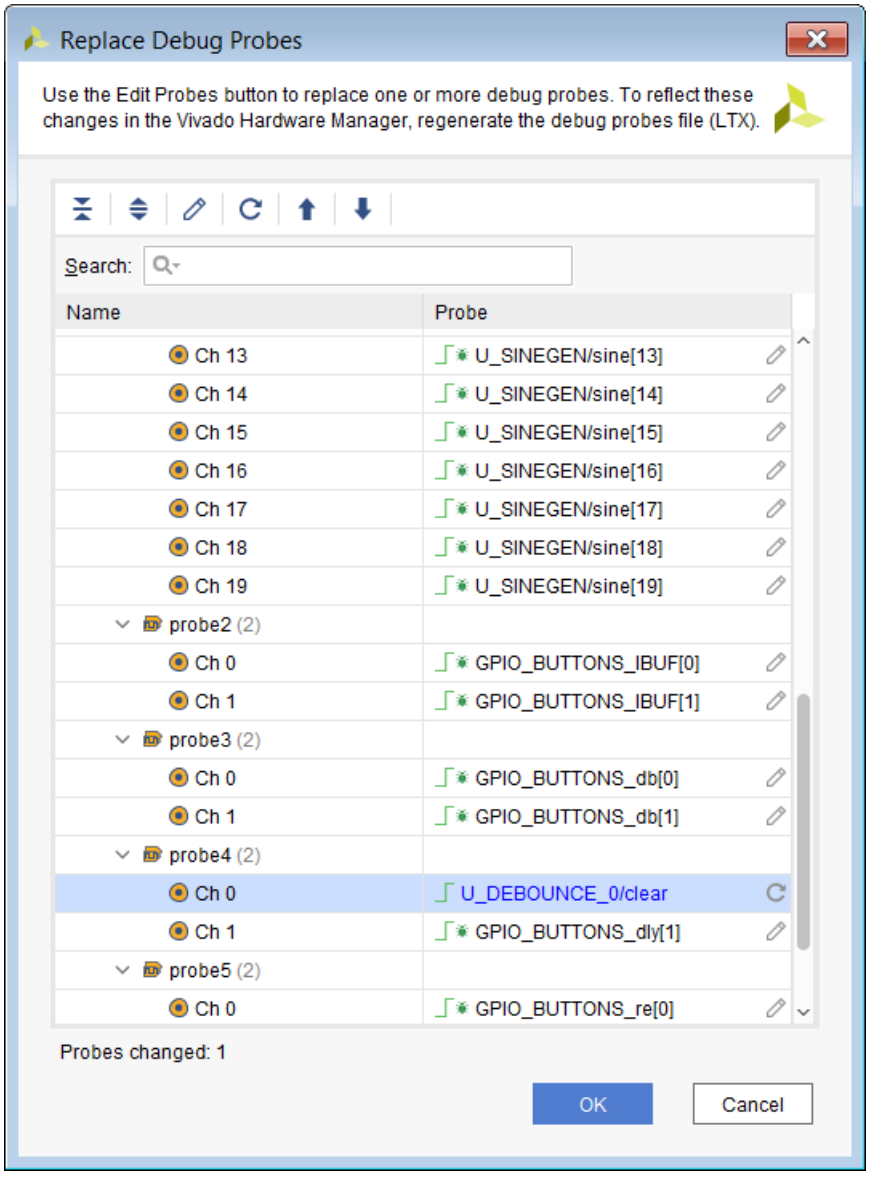

Figure 100: Finish Replace Debug Probes

IMPORTANT: *Check the Tcl Console to ensure that there are no Warnings/Errors.*

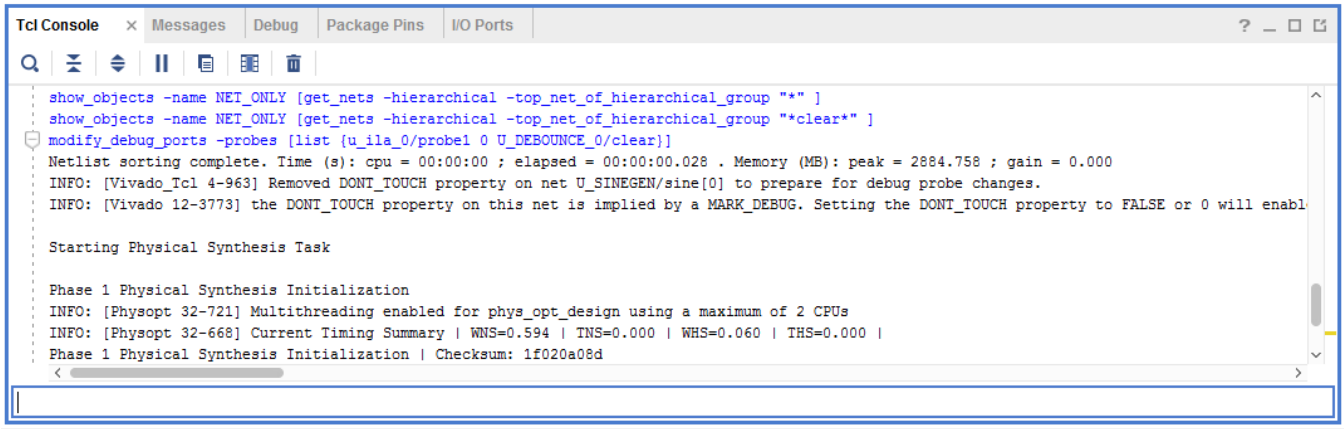

Figure 101: Checking the Tcl Console for Warnings/Errors

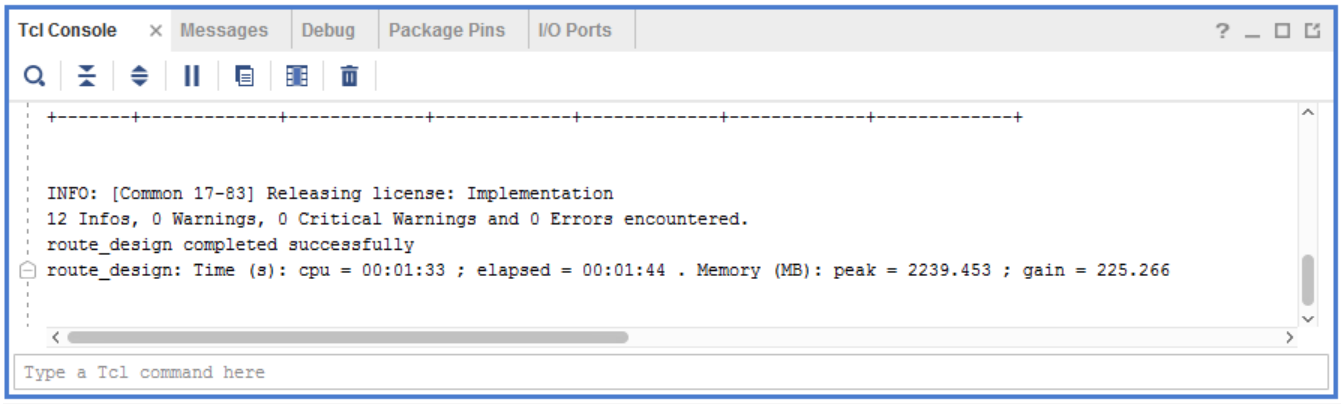

Figure 102: Route Design Messages

9. Save your modifications to a new checkpoint. Use the **Save Checkpoint As** option in the ECO Navigator to bring up the **Save Checkpoint As** dialog box. Specify a file name for the `.dcp` file and click **OK**.

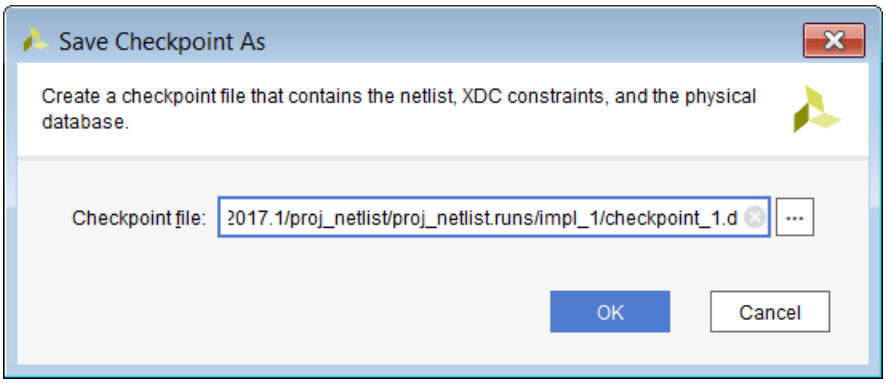

Figure 103: Save Checkpoint As Dialog Box

10. Click **Write Debug Probes** in the ECO Navigator. When the **Write Debug Probes** dialog appears, click **OK** to generate a new `.ltx` file for the debug probes.

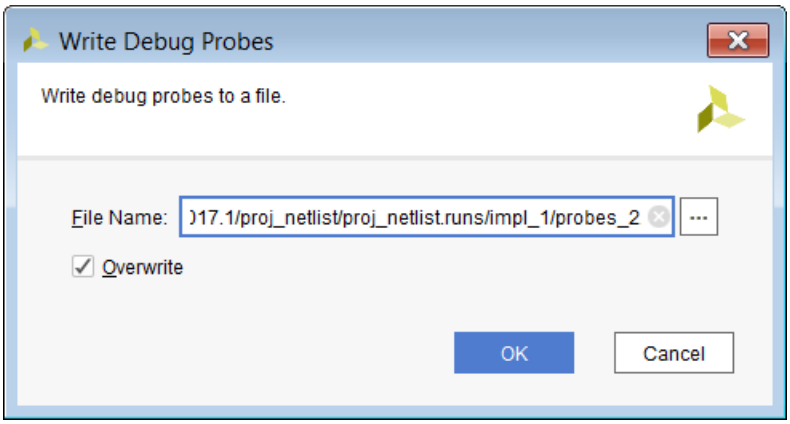

Figure 104: Write Debug Probes Dialog Box

11. When the **Generate Bitstream** dialog appears, change the bit file name to `project_sinegen_demo_routed_debug_changes.bit` in the **Bit File** field and click **OK** to generate a new `.bit` file that reflects the debug probe changes.

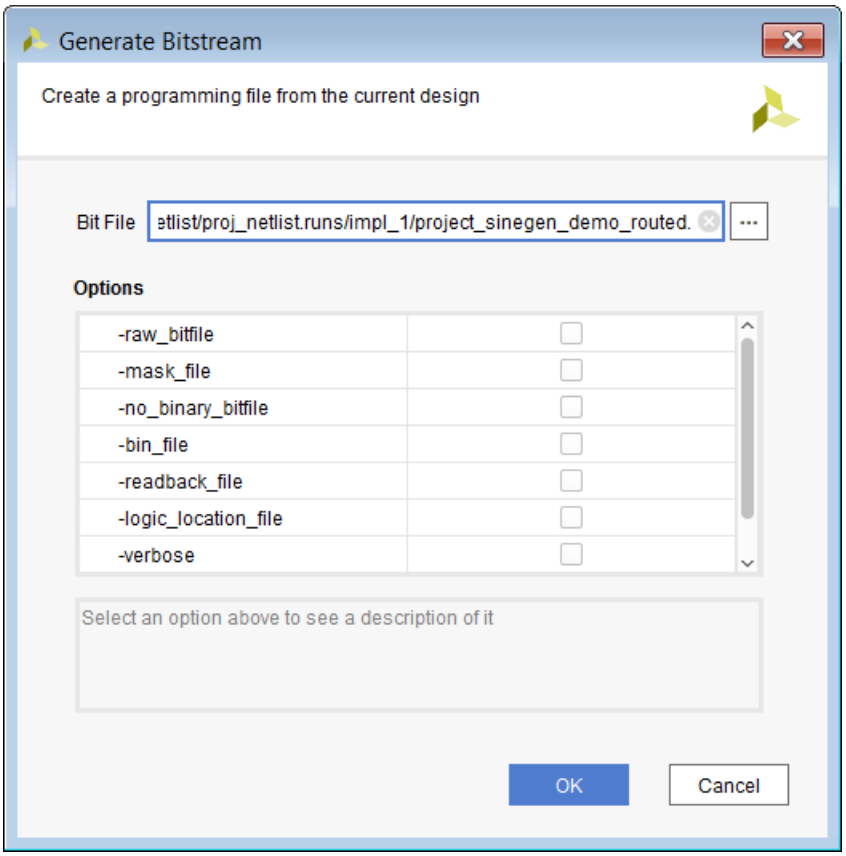

Figure 105: Generate Bitstream Dialog Box

12. Connect to the Vivado Hardware Manager by selecting **Open Hardware Manager** in the ECO Navigator.
13. Connect to the local hardware server by following the steps in the Target Board and Server Set Up section in Lab 5: Using Vivado Logic Analyzer to Debug Hardware.

Program the device using the `.bit` file and `.ltx` files that you created in the previous steps.

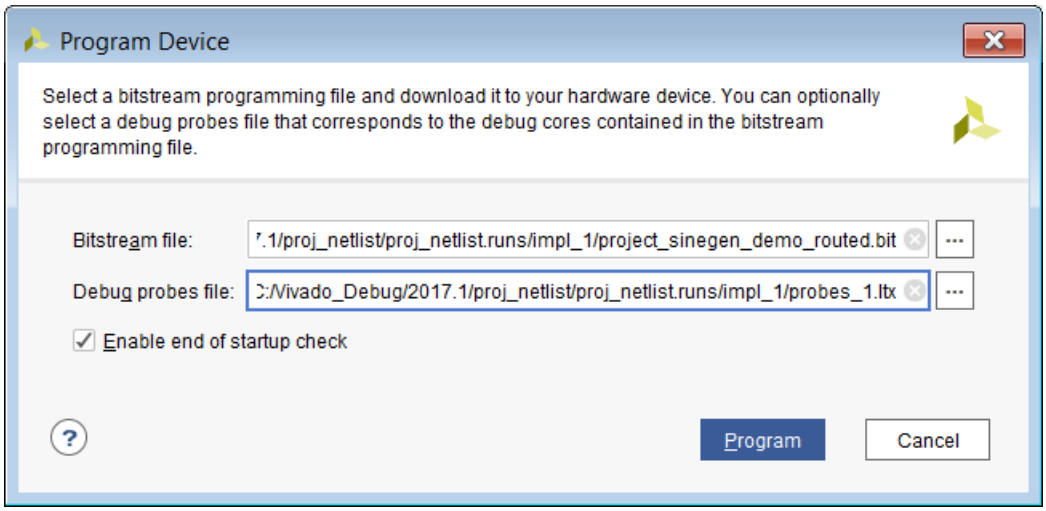

Figure 106: Program Device Dialog Box

14. Select **Window > Debug Probes** from the Vivado Design Suite toolbar. Ensure that the probes that were replaced in step 8 and 9 above are reflected in the probes associated with **hw_ila_1**.

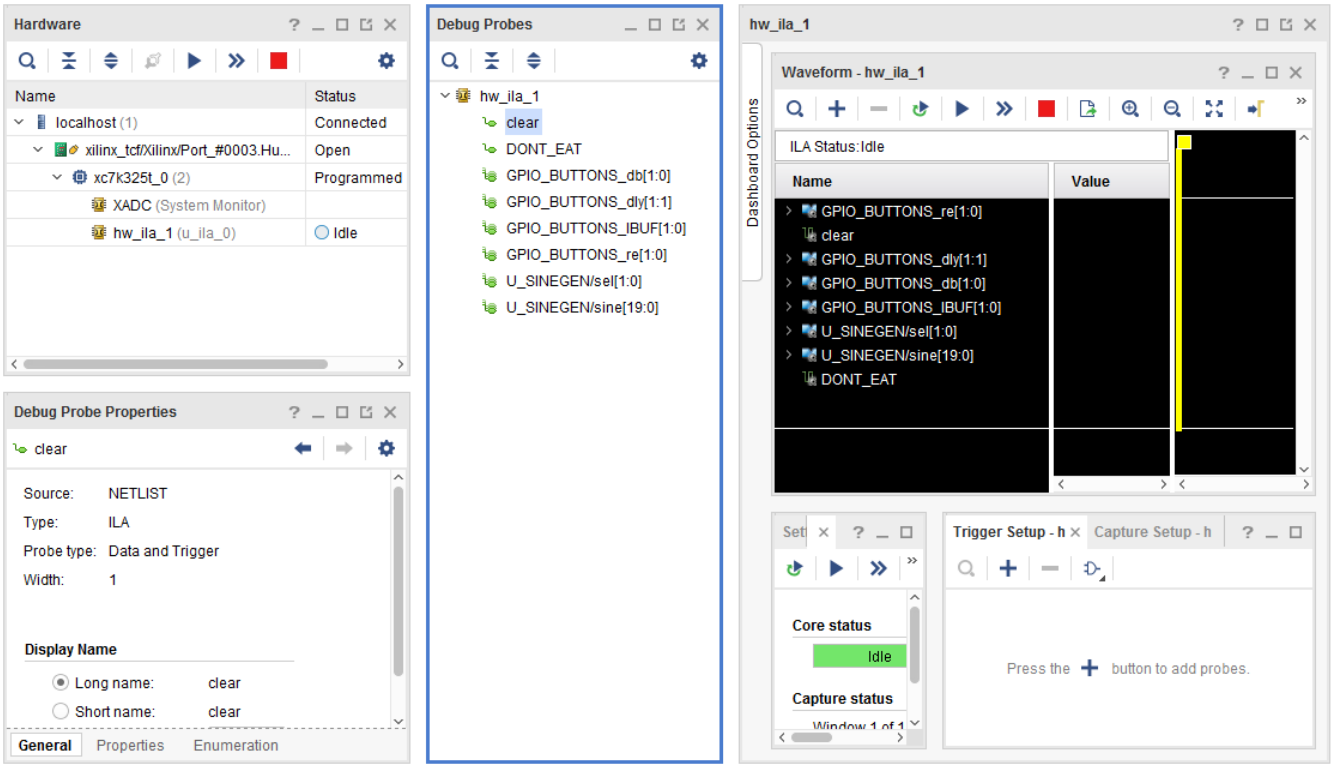

Figure 107: Verifying Debug Probes

15. Run the Trigger on the ILA. Ensure the probes that were replaced in step 8 and 9 above are reflected in the **Waveform** window as well.

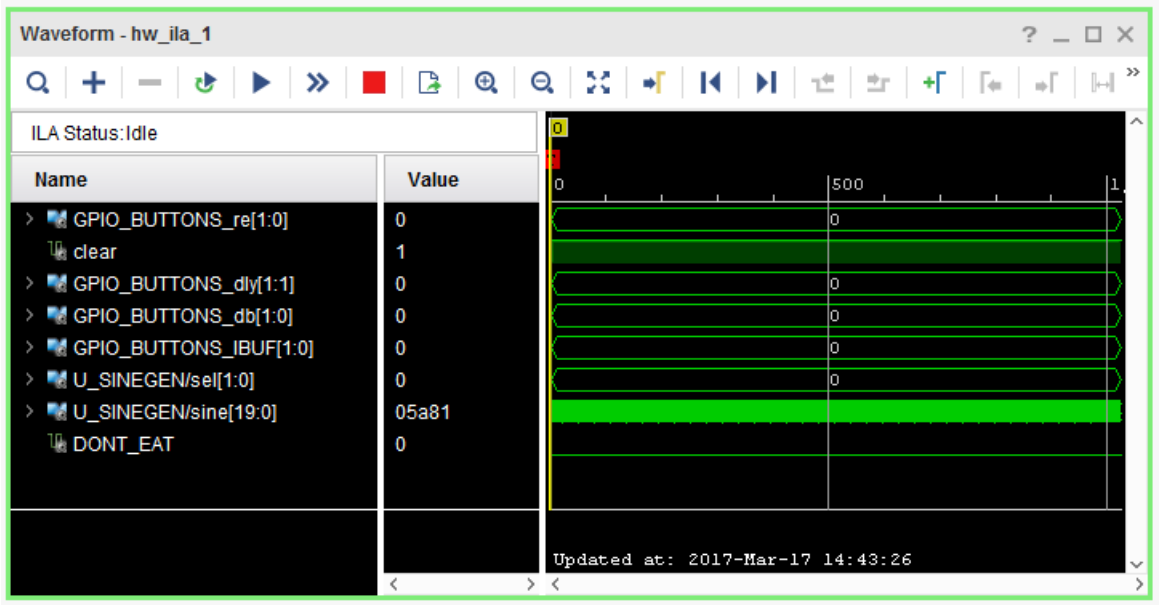

Figure 108: Running the Trigger on the ILA

Introduction

This lab introduces the Vivado® Incremental Compile Flow to add/edit/delete debug cores to an earlier implementation of the design.

Procedure

This lab consists of five generalized steps followed by general instructions and supplementary detailed steps that allow you to make choices based on your skill level as you progress through the lab.

If you need help completing a general instruction, go to the detailed steps below it, or if you are ready, simply skip the step-by-step directions and move on to the next general instruction.

The lab has five primary steps as follows:

Step 1: Opening the Example Design and Adding a Debug Core

1. Start Vivado IDE

   Load Vivado IDE by doing one of the following:

   - Double-click the Vivado IDE icon on the Windows desktop
   - Type `vivado` in a command terminal.

   From the **Getting Started** page, click **Open Example Project**.
2. In the **Open Example Project** dialog box, click **Next**.
3. Select the **CPU (Synthesized)** design template, and click **Next**.
4. In the **Project Name** dialog box, specify the following:

- Project name: `project_cpu_incremental`
- Project location: `<Project_Dir>`

Click **Next**.

5. In the **Default Part** screen, select **xc7k70tfbg676-2** and click **Next**.
6. The **New Project Summary** screen appears, displaying project details. Reviewed these and click **Finish**
7. When Vivado IDE opens with the default view, open the Synthesized design
8. In the **Netlist** window, select the set of signals specified below in the `cpuEngine` hierarchy and apply the MARK_DEBUG property by right-clicking and selecting **Mark Debug** from the dialog.

```
cpuEngine/dcqmem_dat_qmem[*],
cpuEngine/dcpu_dat_qmem[*],
cpuEngine/dcqmem_adr_qmem[*],
cpuEngine/du_dsr[*],
cpuEngine/dvr0__0[*],
cpuEngine/du_dsr[*],
cpuEngine/dcqmem_sel_qmem[*]
```

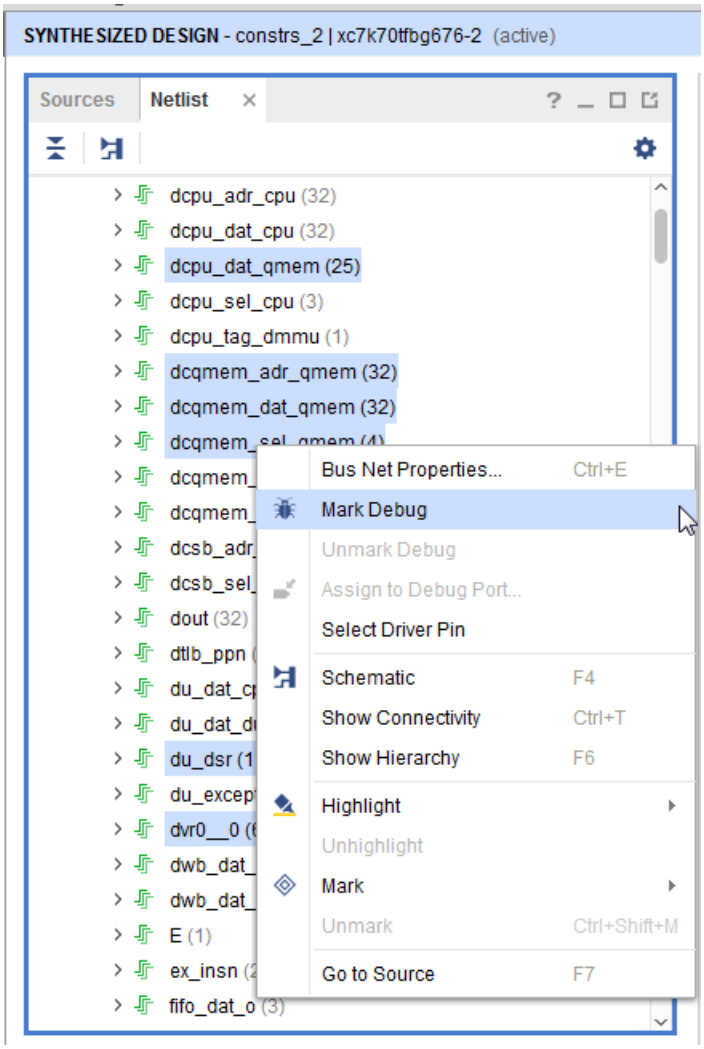

Figure 109: Set MARK_DEBUG Property

Alternatively use can use the Tcl command below to set the MARK_DEBUG property on the signals specified.

```
set_property mark_debug true [get_nets [list {cpuEngine/dcqmem_dat_qmem[*]}
{cpuEngine/dcpu_dat_qmem[*]} {cpuEngine/dcqmem_adr_qmem[*]}
{cpuEngine/du_dsr[*]} {cpuEngine/dvr0__0[*]} {cpuEngine/du_dsr[*]}
{cpuEngine/dcqmem_sel_qmem[*]}]]
```

9. In the **Flow Navigator**, click **Set Up Debug** to invoke the Set Up Debug wizard.

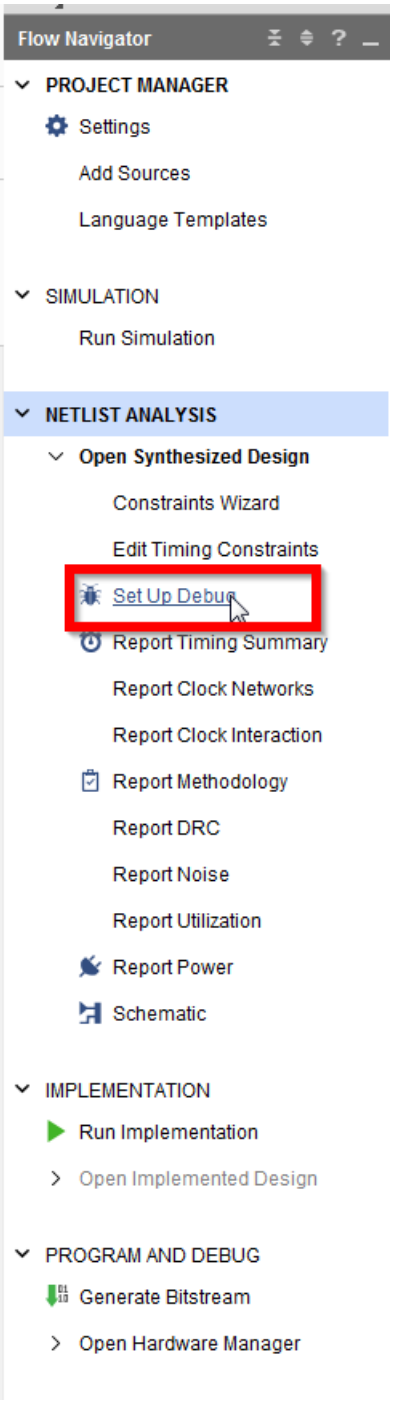

Figure 110: Setup Debug from Flow Navigator

10. When the Set Up Debug Wizard appears, click **Next**.

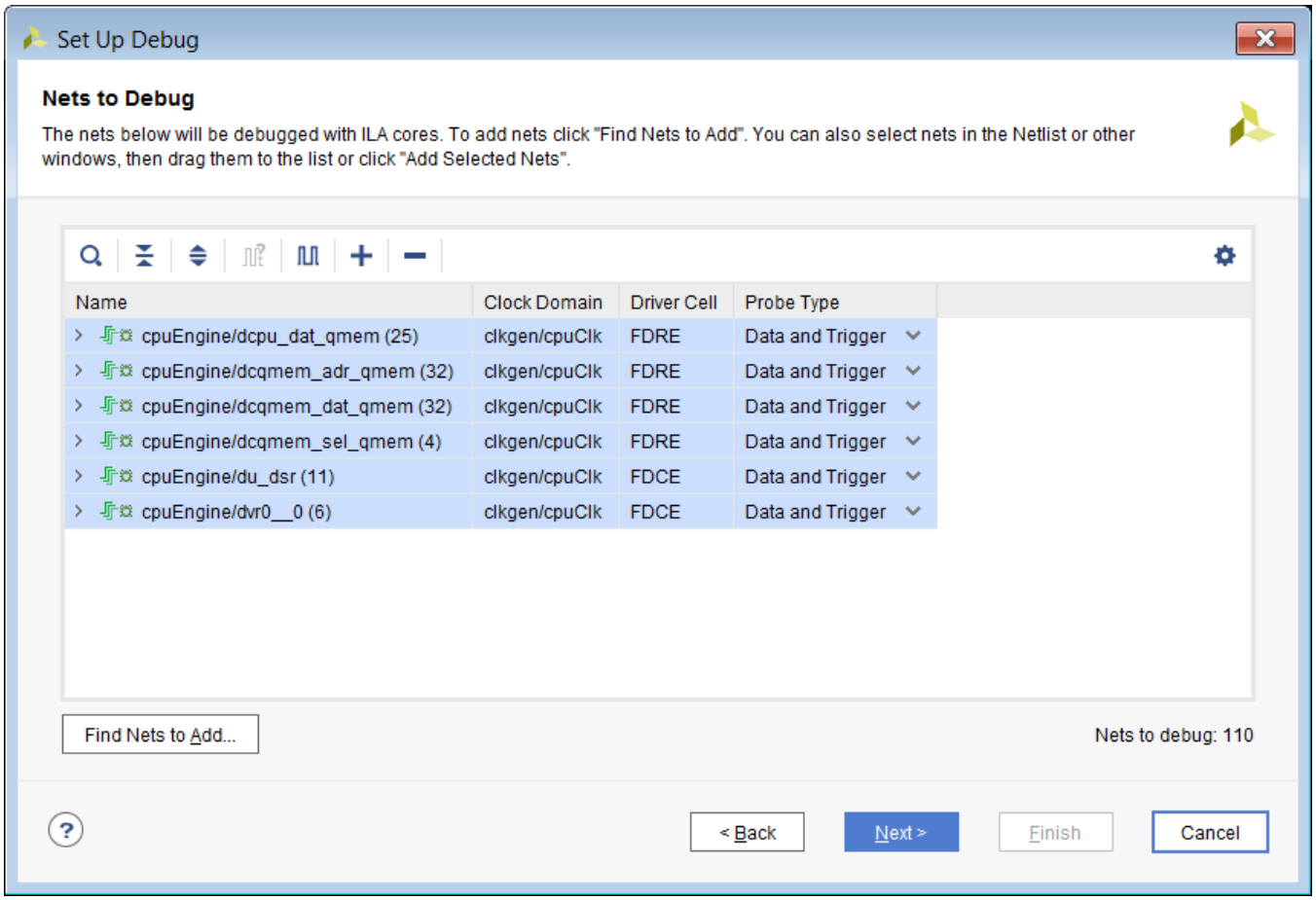

Figure 111: Set Up Debug Nets to Debug

11. When ILA Core Options screen appears, click **Next** again.
12. When **Set Up Debug Summary** screen appears, ensure that **1 debug core** is created and click **Finish**.

13. Check the Debug widow to ensure that **u_ila_0** core has been inserted into the design.

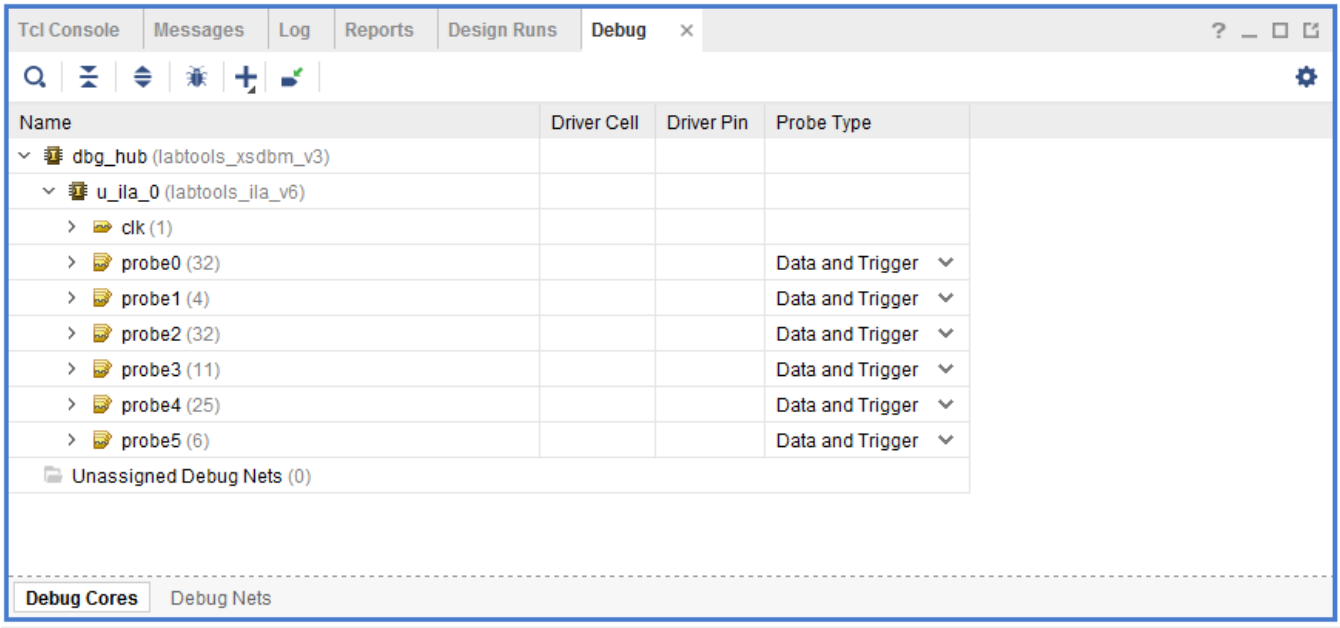

Figure 112: Check u_ila_0 Core

14. Save the new debug XDC commands by selecting **File > Save Constraints** or clicking the **Save Constraints** button.

Step 2: Compiling the Reference Design

The following are the steps to run implementation on the reference design.

1. From the **Flow Navigator**, select **Run Implementation**.
2. After implementation finishes, the **Implementation Complete** dialog box opens. Click **Cancel**.
3. In a project-based design, the Vivado Design Suite saves intermediate implementation results as design checkpoints in the implementation runs directory. You will use one of the saved design checkpoints from the implementation in the incremental compile flow.

TIP: *When you re-run implementation, the previous results will be deleted. Save the intermediate implementation results to a new directory or create a new implementation run for your incremental compile to preserve the reference implementation run directory.*

4. In the **Design Runs** window, right click **impl_1** and select **Open Run Directory** from the popup menu. This opens the run directory in a file browser as seen in the figure below. The run directory contains the routed checkpoint (`top_routed.dcp`) to be used later for the incremental compile flow. The location of the implementation run directory is a property of the run.
5. Get the location of the current run directory in the Tcl Console by typing:

```
get_property DIRECTORY [current_run]
```

This returns the path to the current run directory that contains the design checkpoint. You can use this Tcl command, and the DIRECTORY property, to locate the DCP files needed for the incremental compile flow.

Step 3: Create New Runs

In this step, you define new synthesis and implementation runs to preserve the results of the current runs. Then you make debug related changes to the design and rerun synthesis and implementation. If you do not create new runs, Vivado overwrites the current results.

1. From the Vivado tool bar, select **Flow > Create Runs** to invoke the Create New Runs wizard.
2. In the **Create New Runs** screen, click **Next**.
3. The **Configure Implementation Runs** screen opens, as shown in the figure below. Select the **Make Active** check box, and click **Next**.

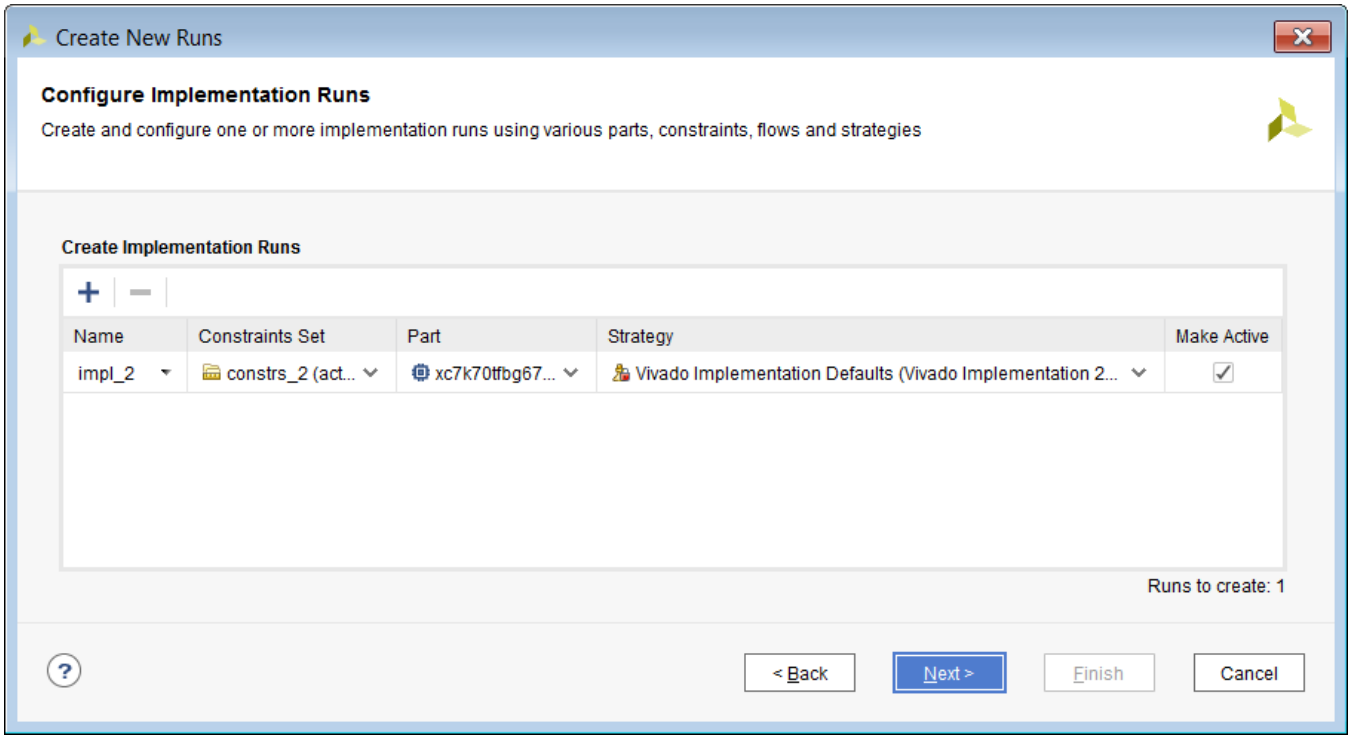

Figure 113: Configure Implementation Runs

4. From the **Launch Options** window, select **Do not launch now** and click **Next**.

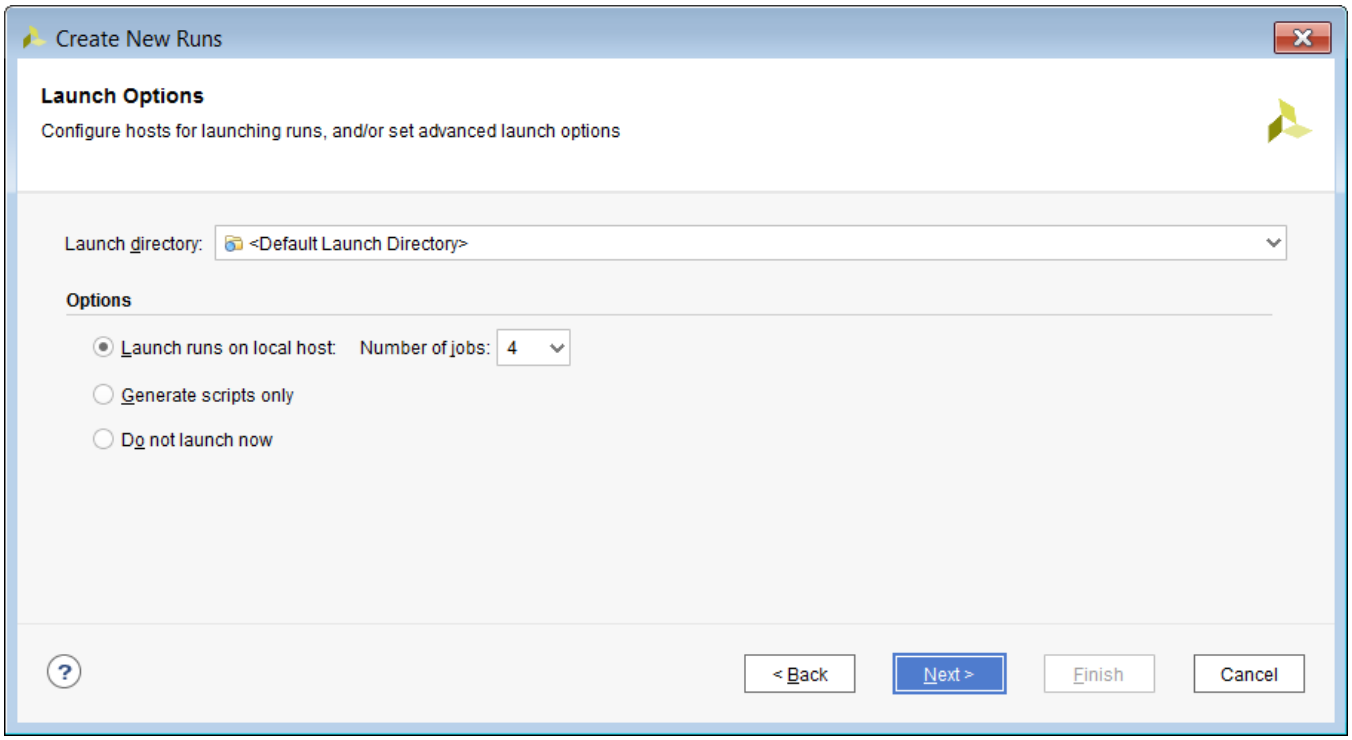

Figure 114: Launch Options

5. In the **Create New Runs Summary** screen, click **Finish** to create the new runs.

   The Design Runs window displays the new active runs in bold.

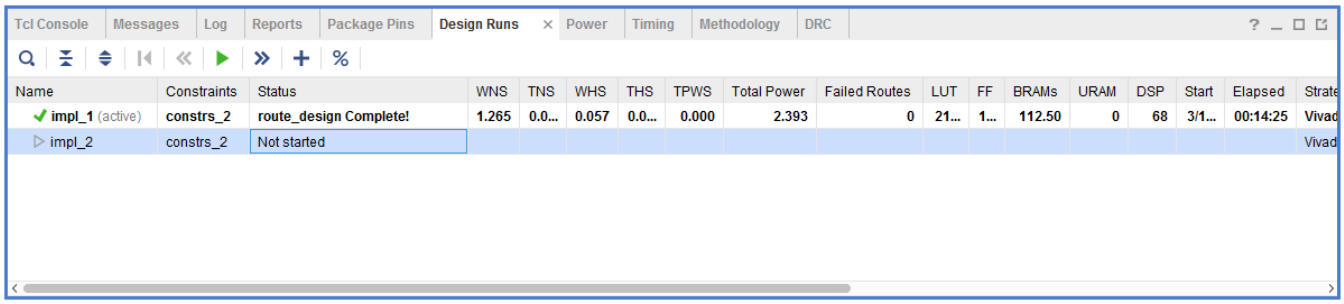

Figure 115: New Design Runs

Step 4: Making Incremental Debug Changes

In this step, in order to add/delete/edit debug cores, you need to reopen the synthesized netlist. Make debug related changes to the design using the Set Up Debug wizard.

1. If you have closed the synthesized netlist, go back to the synthesized design using the **Flow Navigator**.
2. For this tutorial, assume that you now need to debug some other nets in addition to the ones already being debugged. However, you want to reuse the previous place and route results. So now, you will debug the nets `fftEngine/fifo_out[*]`.
3. Apply the MARK_DEBUG property to this bus in the netlist window.

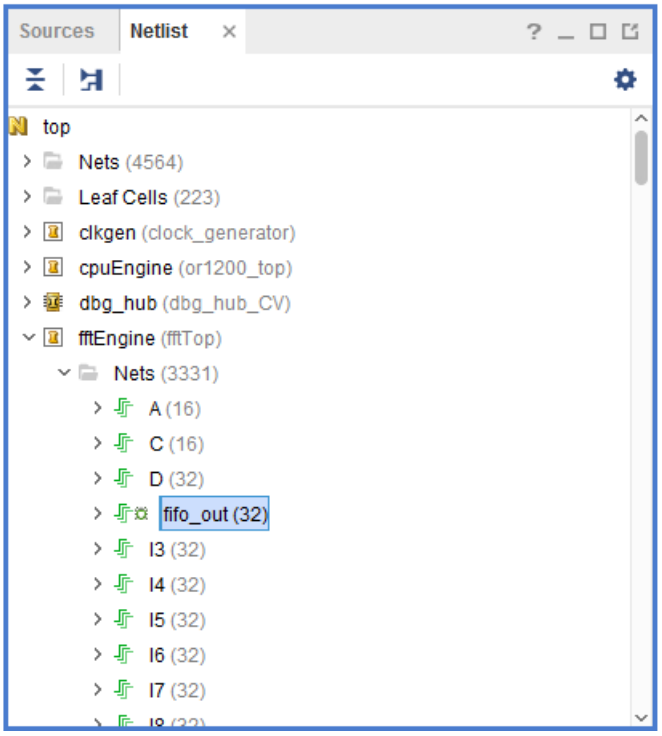

Figure 116: Netlist fifo_out

4. Click **Set Up Debug** to invoke the Set Up Debug wizard in the **Flow Navigator**.

5. In the **Existing Debug Nets** tab, select **Continue debugging 110 nets connected to existing debug cores**.

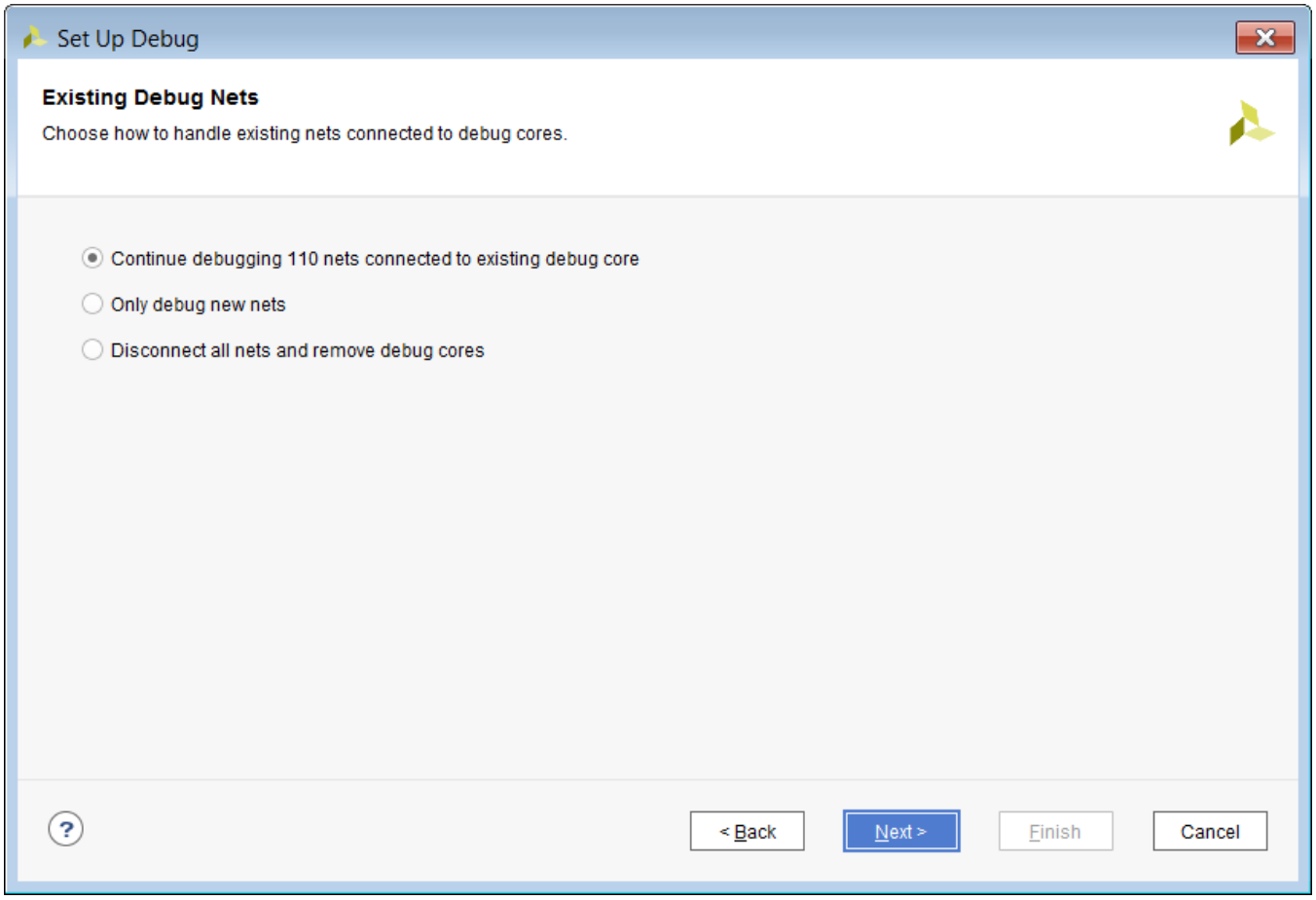

Figure 117: Existing Debug Nets

6. Click **Next** to debug the new unassigned debug nets.

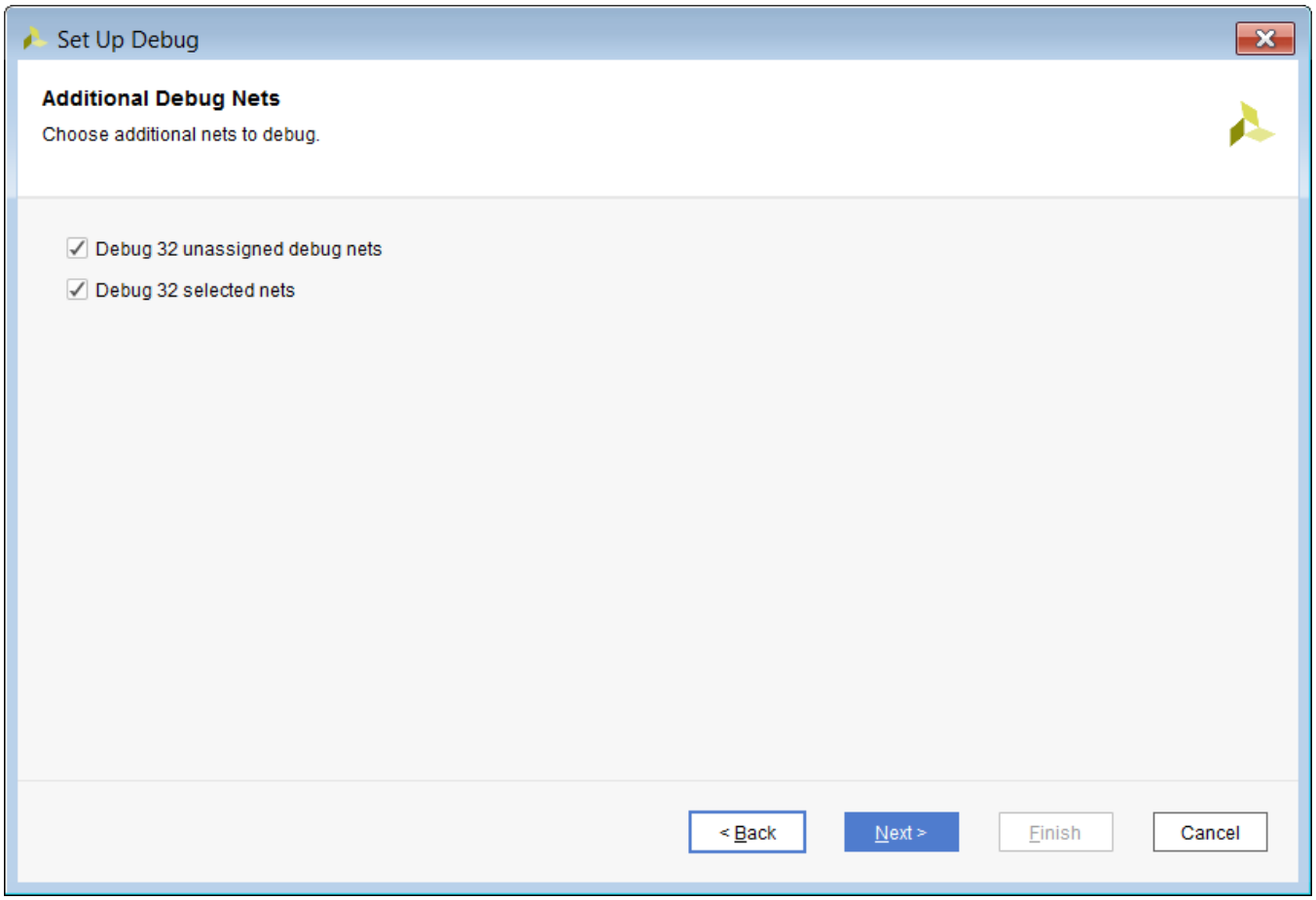

Figure 118: Set Up Debug Additional Debug Nets

7. Click **Next** and ensure the new nets are in the list of **Nets to Debug**.

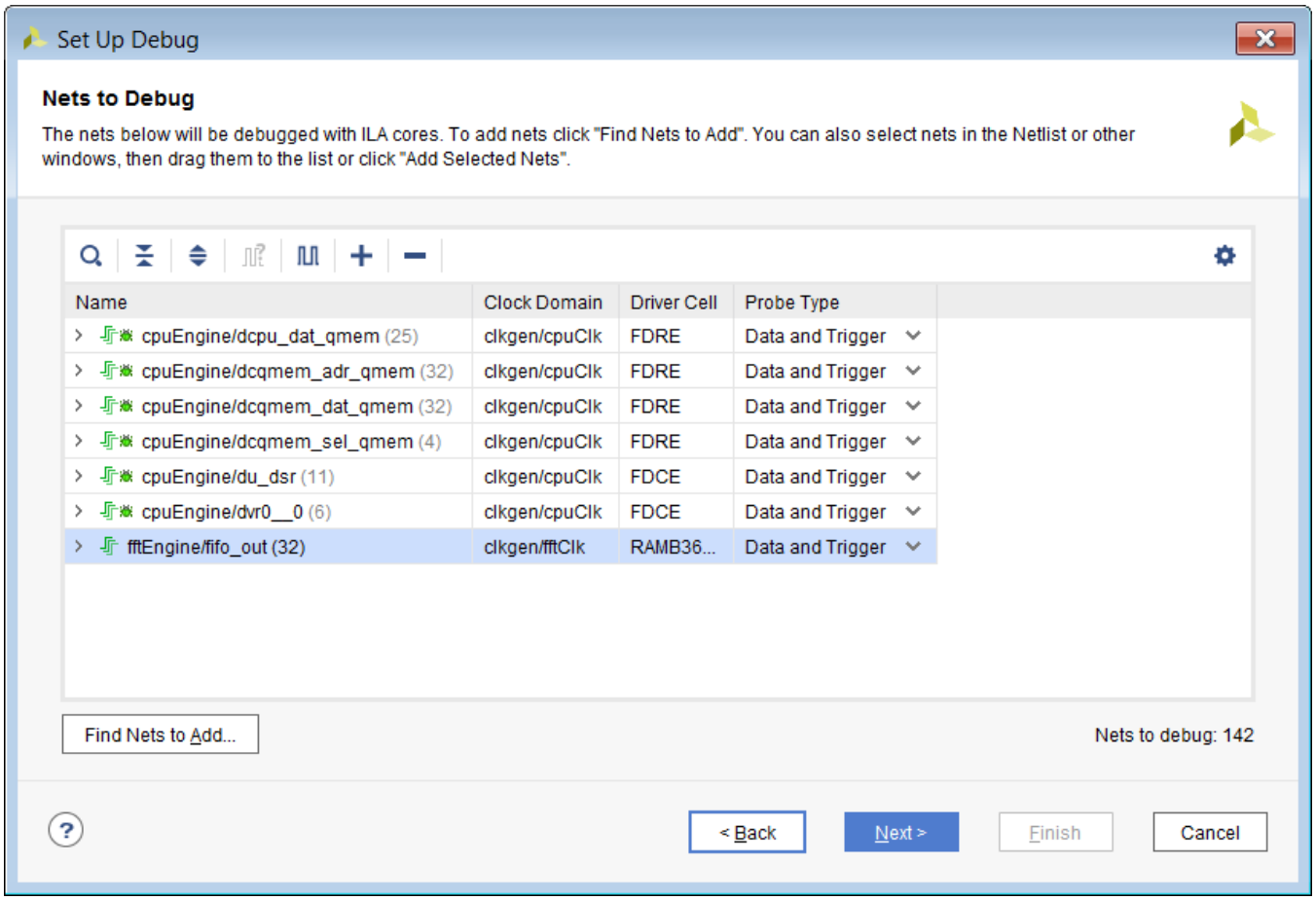

Figure 119: Viewing Additional Debug Nets

8. Click **Next** and ensure that two debug cores are created and click **Finish**.
9. Save the new debug XDC commands by clicking the **Save Constraints** button or selecting **File > Save Constraints** from the main Vivado toolbar.

Step 5: Running Incremental Compile

In the previous steps, you have updated the design with debug changes. You could run implementation on the new netlist, to place and route the design and work to meet the timing requirements. However, with only minor changes between this iteration and the last, the incremental compile flow lets you reuse the bulk of your prior debug, placement and routing efforts. This can greatly reduce the time it takes to meet timing on design iterations. For more information, refer to *Vivado Design Suite User Guide: Implementation* (UG904).

1. Start by defining the design checkpoint (DCP) file to use as the reference design for the incremental compile flow. This is the design from which the Vivado Design Suite draws placement and routing data.
2. In the **Design Runs** window, right-click the **impl_2 run** and select **Set Incremental Compile** from the popup menu. The Set Incremental Compile dialog box opens.
3. Click the **Browse** button in the **Set Incremental Compile** dialog box, and browse to the `./project_cpu_incremental.runs/impl_1` directory.
4. Select `top_routed.dcp` as the incremental compile checkpoint.

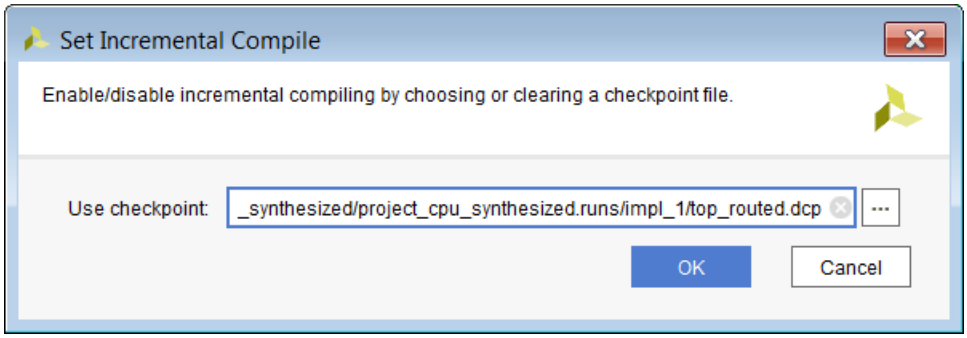

Figure 120: Set Incremental Compile

5. Click **OK**. This information is stored in the INCREMENTAL_CHECKPOINT property of the selected run. Setting this property tells the Vivado Design Suite to run the incremental compile flow during implementation.
6. You can check this property on the current run using the following Tcl command:

   ```
   get_property INCREMENTAL_CHECKPOINT [current_run]
   ```

   This returns the full path to the top_routed.dcp checkpoint.

TIP: *To disable Incremental Compile for the current run, clear the INCREMENTAL_CHECKPOINT property. This can be done using the Set Incremental Compile dialog box, or by editing the property directly through the Properties window of the design run, or through the reset_property command.*

7. From the **Flow Navigator**, select **Run Implementation**.

   This runs implementation on the current run, using the `top_routed.dcp` file as the reference design for the incremental compile flow. When the run is finished, the **Implementation Completed** dialog box opens.

8. Select **Open Implemented Design** and click **OK**. As shown in the following figure, the **Design Runs** window shows the elapsed time for implementation run **impl_2** versus **impl_1**.

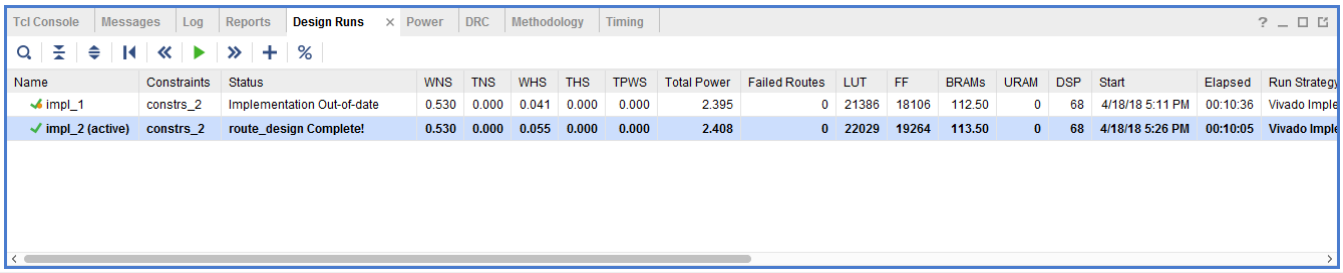

Figure 121: Design Runs

> ***Note:*** *This is an extremely small design. The advantages of the incremental compile flow are greater and significant with larger, more complex designs.*

9. Select the **Reports** tab in the **Results** window area and under **Place Design**, double-click **Incremental Reuse Report** as shown in the following figure.

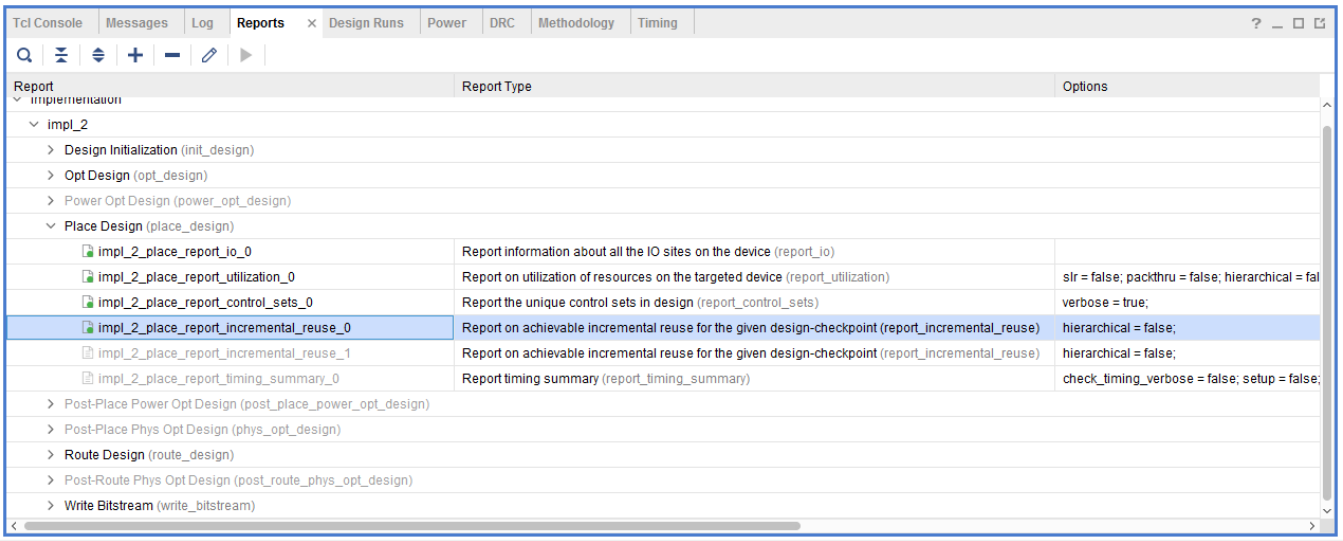

Figure 122: Opening Incremental Reuse Report

The Incremental Reuse Report opens in the Vivado IDE text editor. This report shows the percentage of reused Cells, Ports, and Nets. A higher percentage indicates more effective reuse of placement and routing from the incremental checkpoint.

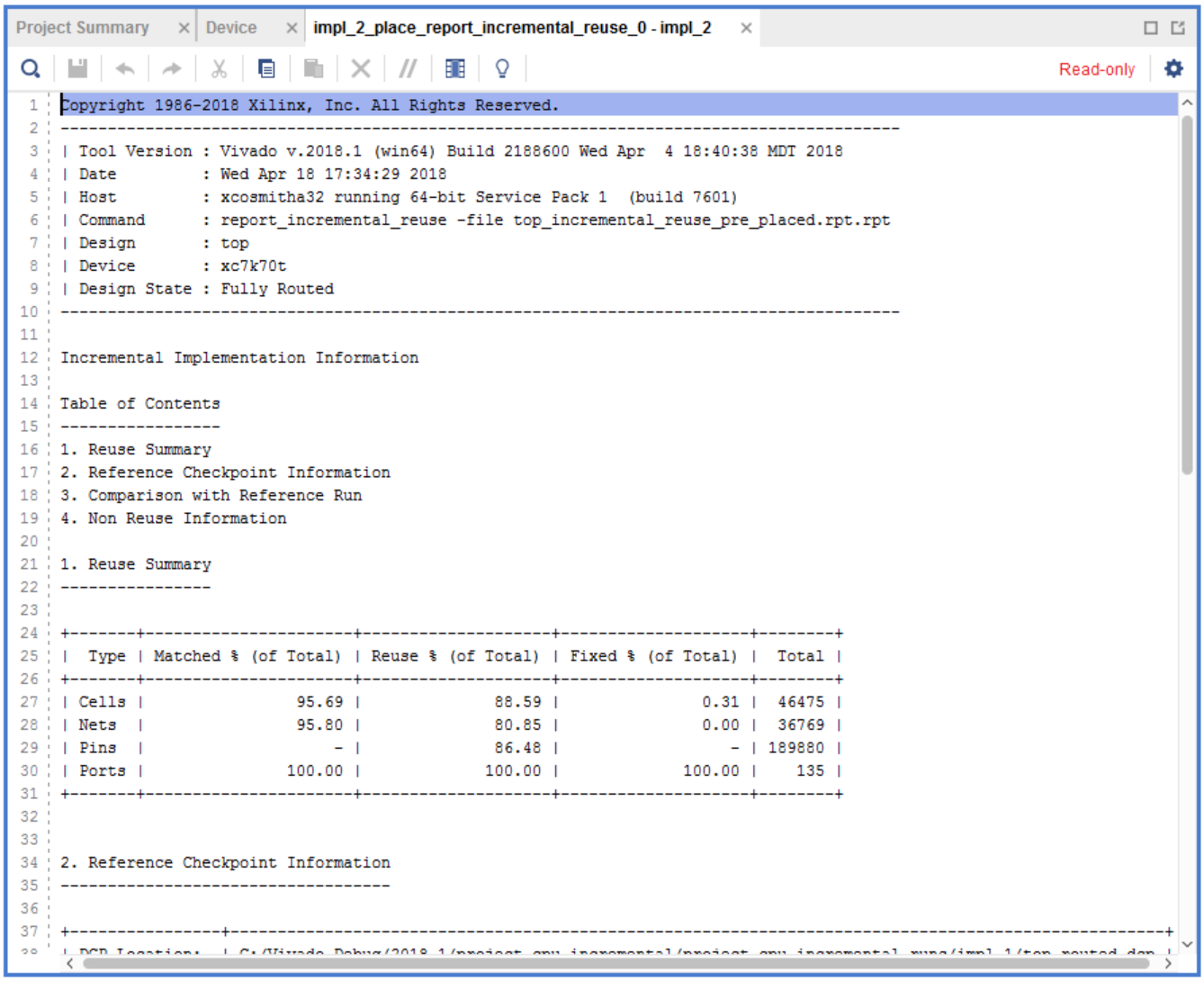

Figure 123: Incremental Reuse Report Sample

In the report, fully reused nets indicate that the entire routing of the nets is reused from the reference design. Partially reused nets indicate that some of the routing of the nets reuses routing from the reference design. Some segments re-route due to changed cells, changed cell placements, or both. Non-reused nets indicate that the net in the current design was not matched in the reference design.

Conclusion

This concludes the lab. You can close the current project and exit the Vivado IDE.

In this lab, you learned how to run the Incremental Compile Debug flow, using a checkpoint from a previously implemented design. You inserted a new debug core using the Set Up Debug wizard on the

synthesized netlist. You examined the similarity between a reference design checkpoint and the current design by examining the Incremental Reuse Report.

Lab 8: Using Vivado Serial Analyzer to Debug Serial Links

Introduction

The Serial I/O analyzer is used to interact with IBERT debug IP cores contained in a design. It is used to debug and verify issues in high speed serial I/O links.

The Serial I/O Analyzer has several benefits as listed below:

- Tight integration with Vivado® IDE.
- Ability to script during netlist customization/generation and serial hardware debug.
- Common interface with the Vivado Integrated Logic Analyzer.

The customizable LogiCORE™ IP Integrated Bit Error Ratio Tester (IBERT) core for 7 series FPGA GTX transceivers is designed for evaluating and monitoring the GTX transceivers. This core includes pattern generators and checkers that are implemented in FPGA logic, and provides access to ports and the dynamic reconfiguration port attributes of the GTX transceivers. Communication logic is also included to allow the design to be run time accessible through JTAG.

In the course of this tutorial, you:

- Create, customize, and generate an Integrated Bit Error Ratio Tester (IBERT) core design in the Vivado Integrated Design Suite.
- Interact with the design using Serial I/O Analyzer. This includes connecting to the target KC705 board, configuring the device, and interacting with the IBERT/Transceiver IP cores.
- Perform a sweep test to optimize your transceiver channel and to plot data using the IBERT sweep plot GUI feature.

Design Description

You can customize the IBERT core and use it to evaluate and monitor the functionality of transceivers for a variety of Xilinx devices. The focus for this tutorial is on Kintex®-7 GTX transceivers. Accordingly, the KC705 target board is used for this tutorial.

The following figure shows a block diagram of the interface between the IBERT Kintex-7 GTX core interfaces with Kintex-7 transceivers.

- **DRP Interface and GTX Port Registers**: IBERT provides you with the flexibility to change GTX transceiver ports and attributes. Dynamic reconfiguration port (DRP) logic is included, which allows the runtime software to monitor and change any attribute in any of the GTX transceivers included in the IBERT core. When applicable, readable and writable registers are also included. These are connected to the ports of the GTX transceiver. All are accessible at run time using the Vivado logic analyzer.
- **Pattern Generator**: Each GTX transceiver enabled in the IBERT design has both a pattern generator and a pattern checker. The pattern generator sends data out through the transmitter.
- **Error Detector**: Each GTX transceiver enabled in the IBERT design has both a pattern generator and a pattern checker. The pattern checker takes the data coming in through the receiver and checks it against an internally generated pattern.

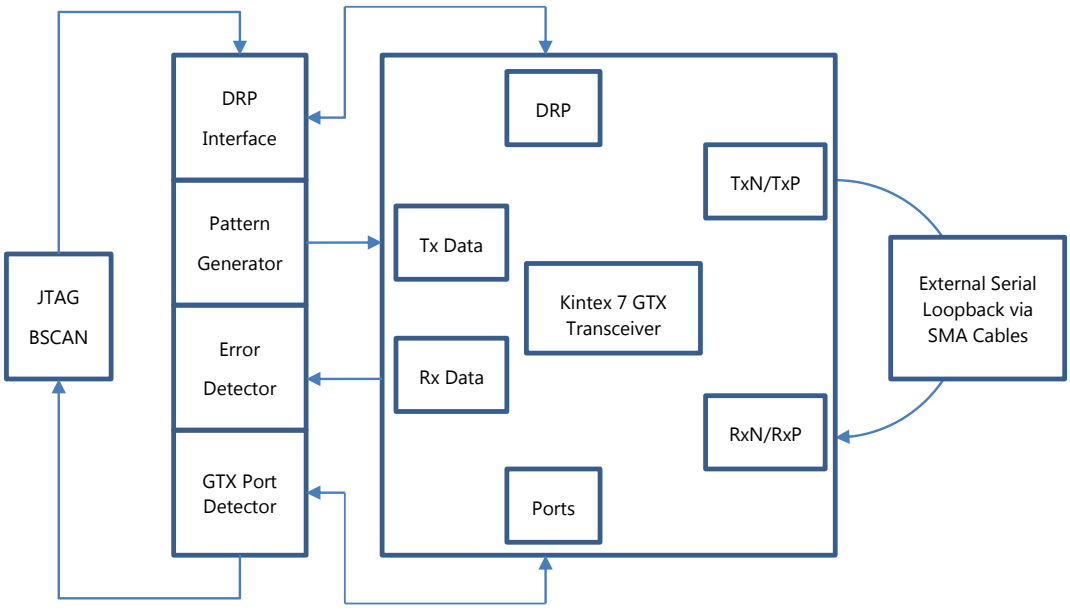

Figure 124: IBERT Design Flow

Step 1: Creating, Customizing, and Generating an IBERT Design

To create a project, use the New Project wizard to name the project, to add RTL source files and constraints, and to specify the target device.

1. Invoke the Vivado IDE.
2. In the **Quick Start** screen, click **Create Project** to start the **New Project** wizard, and click **Next**.
3. In the **Project Name** page, name the new project **ibert_tutorial** and provide the project location (`C:/ibert_tutorial`). Ensure that **Create Project Subdirectory** is selected. Click **Next**.
4. In the **Project Type** page, specify the **Type of Project** to create as **RTL Project.** Click **Next**.
5. In the **Add Sources** page, click **Next**.
6. In the **Add Existing IP** page, click **Next**.
7. In the **Add Constraints** page, click **Next**.
8. In the **Default Part** page, select **Boards** and then select **Kintex-7 KC705 Evaluation Platform**. Click **Next**.
9. Review the **New Project Summary** page. Verify that the data appears as expected, per the steps above. Click **Finish**.

   ***Note**: It might take a moment for the project to initialize.*

Step 2: Adding an IBERT core to the Vivado Project

1. In the **Flow Navigator** click **IP Catalog**.

   The IP Catalog opens.

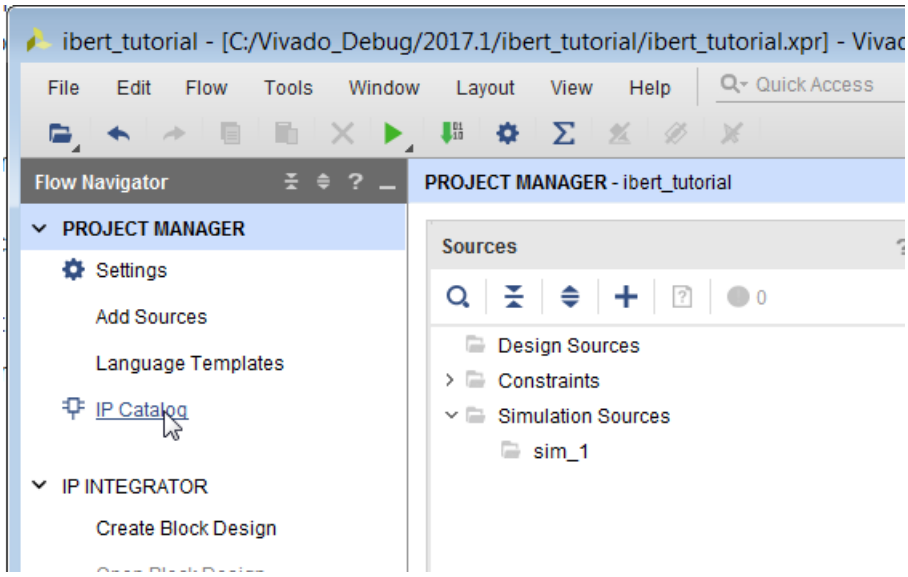

Figure 125: Opening the Vivado IP Catalog

2. In the search field of the IP Catalog type **IBERT**, to display the IBERT 7 Series GTX IP.

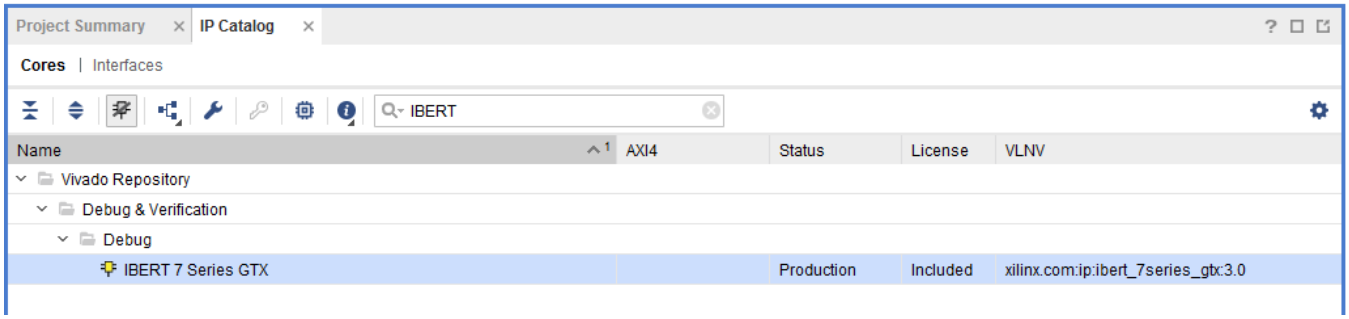

Figure 126: Instantiating the IBERT IP from the Vivado IP Catalog

3. Double-click **IBERT 7 Series GTX IP**. This brings up the customization GUI for the IBERT.

4. In the **Customize IP** dialog box, choose the following options in the **Protocol Definition** tab:
   a. Type the name of the component in the **Component Name** field. In this case, leave the name as the default name, **ibert_7series_gtx_0**.
   b. Ensure that the **Silicon Version** is selected as **General ES/Production**.
   c. Ensure that the **Number of Protocols** option is set to **1**.
   d. Change the **LineRate (Gbps)** to **8**.
   e. Change **DataWidth** to **40**.
   f. Change **Refclk (MHz)** to **125**.
   g. Ensure that the **Quad Count** is set to **2**.
   h. Ensure **Quad PLL** box is selected.

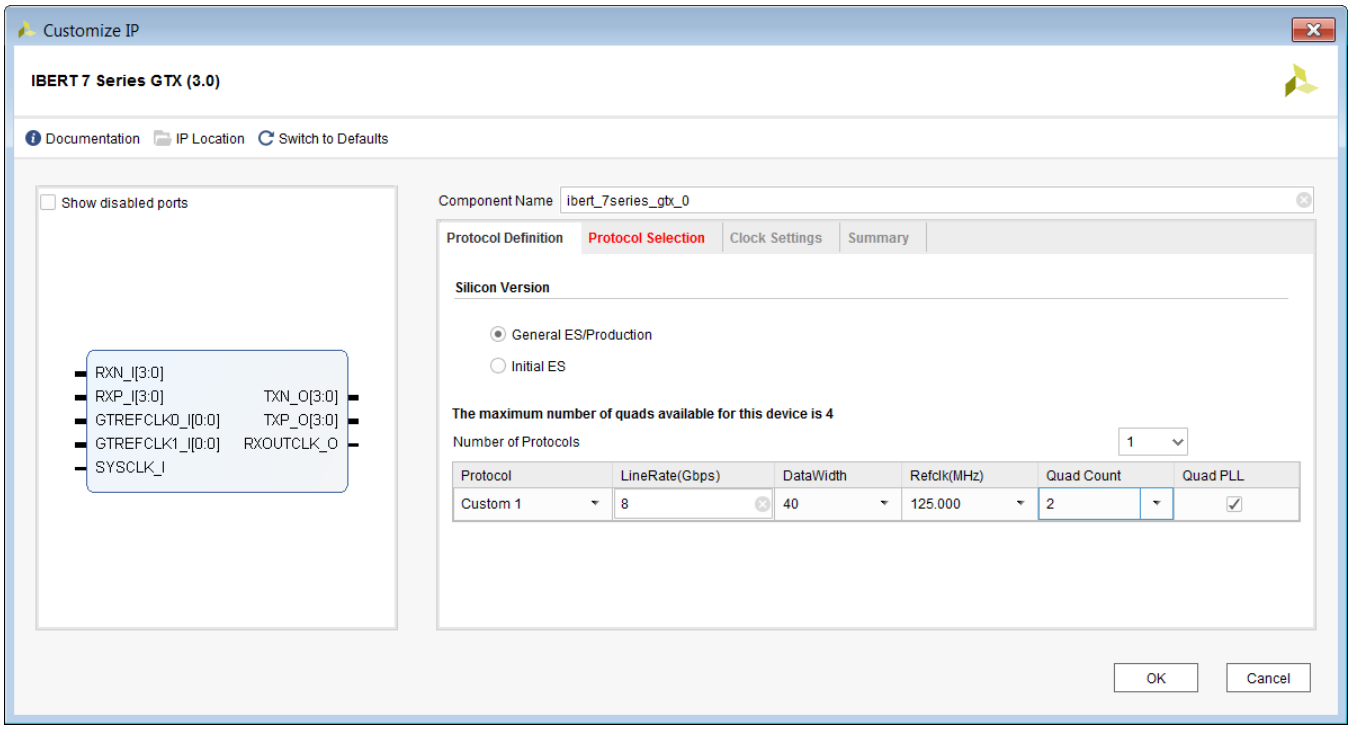

Figure 127: Setting the Protocol Definition on the IBERT Core

5. Under the **Protocol Selection** tab, update the following selections:
   a. For GTX Location QUAD_117, in the **Protocol Selected** column, click the pull-down menu and select **Custom 1 / 8 Gbps**. This should automatically populate **Refclk Selection** to **MGTREFCLK0 117** and **TXUSRCLK Source** to **Channel 0**.

b. For GTX Location QUAD_118, do the following:

   i. In the **Protocol Selected** column, click the pull-down menu and select **Custom 1 / 8 Gbps.**

   ii. In the **Refclk Selection** column, change the value to **MGTREFCLK0 117**.

   iii. In the **TXUSRCLK Source** column, change the value to **Channel 0**.

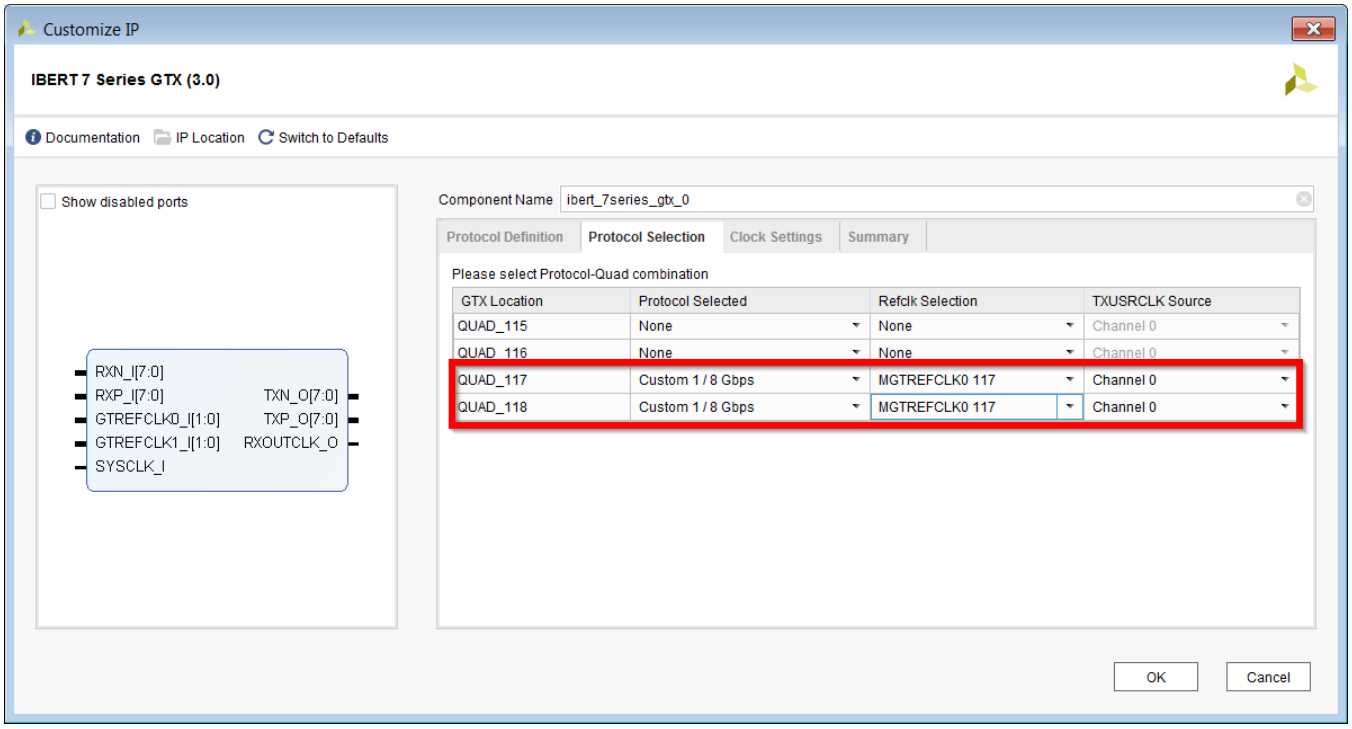

Figure 128: Setting the Protocol Selection on the IBERT Core

6. Click the **Clock Settings** tab and make the following changes for both QUAD_117 and QUAD_118:

   a. Leave the **Source** column at its default value of **External**.

   b. Change the **I/O Standard** column to **DIFF SSTL15**.

   c. Change the **P Package Pin** to **AD12**.

   d. Change the **N Package Pin** to **AD11**.

   e. Leave the **Frequency(MHz)** at its default value of **200.00**.

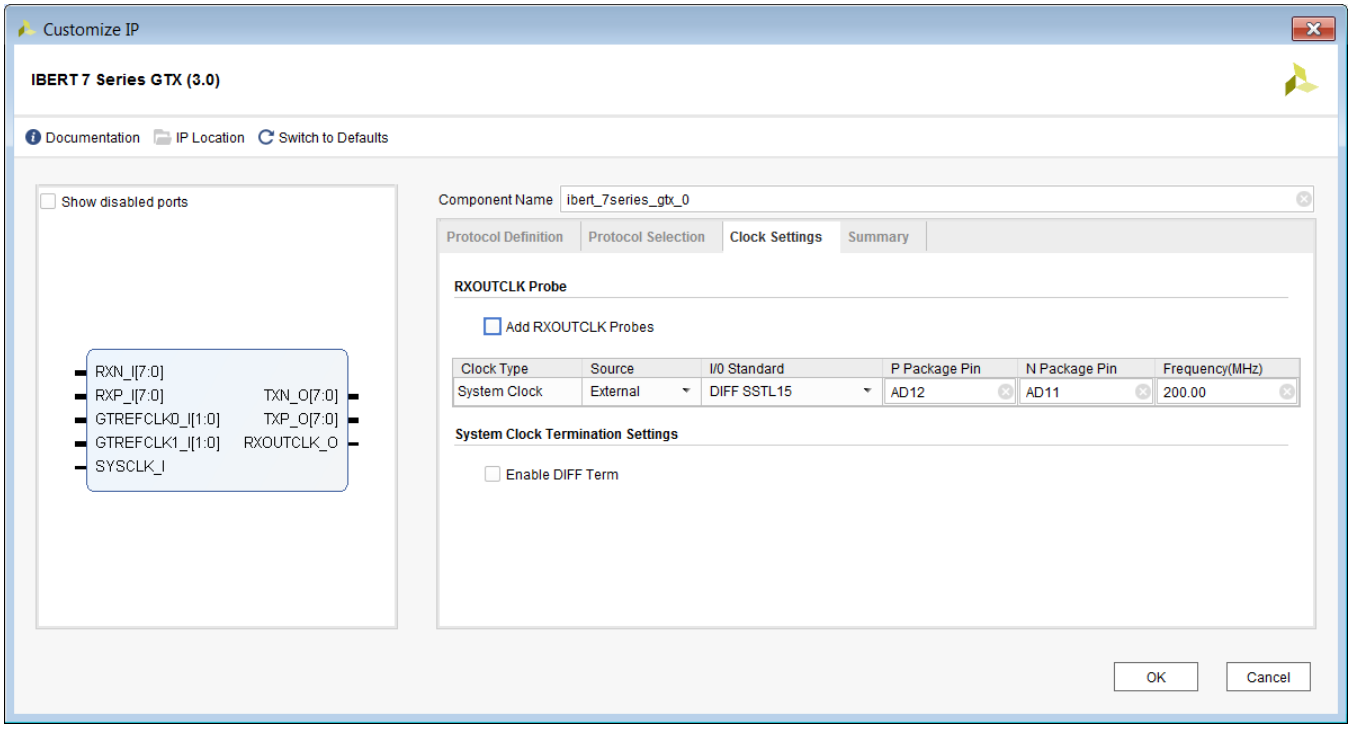

Figure 129: Specifying Clock Settings for the IBERT Core

7. Click the **Summary** tab and ensure that the content matches the following figure, then click **OK**.

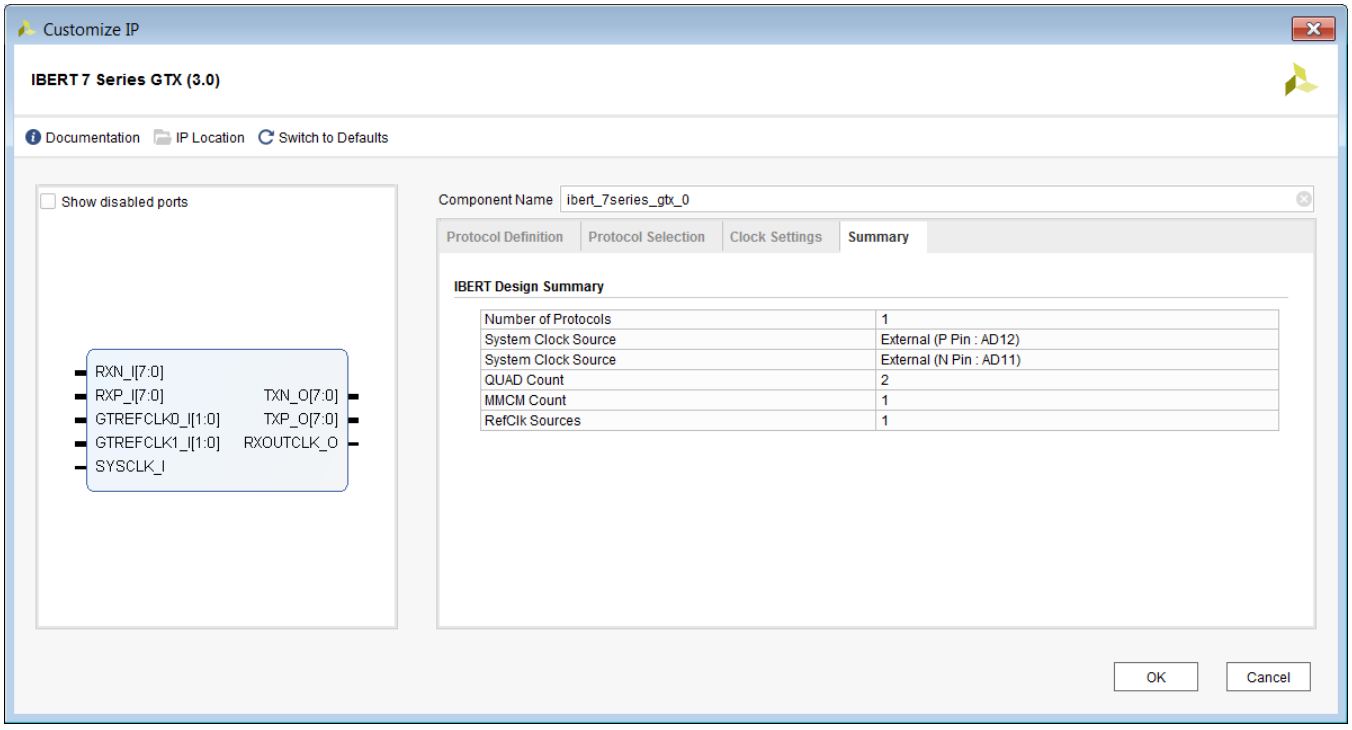

Figure 130: IBERT Core Summary Page

8. When the **Generate Output Products** dialog box opens, click **Generate**.

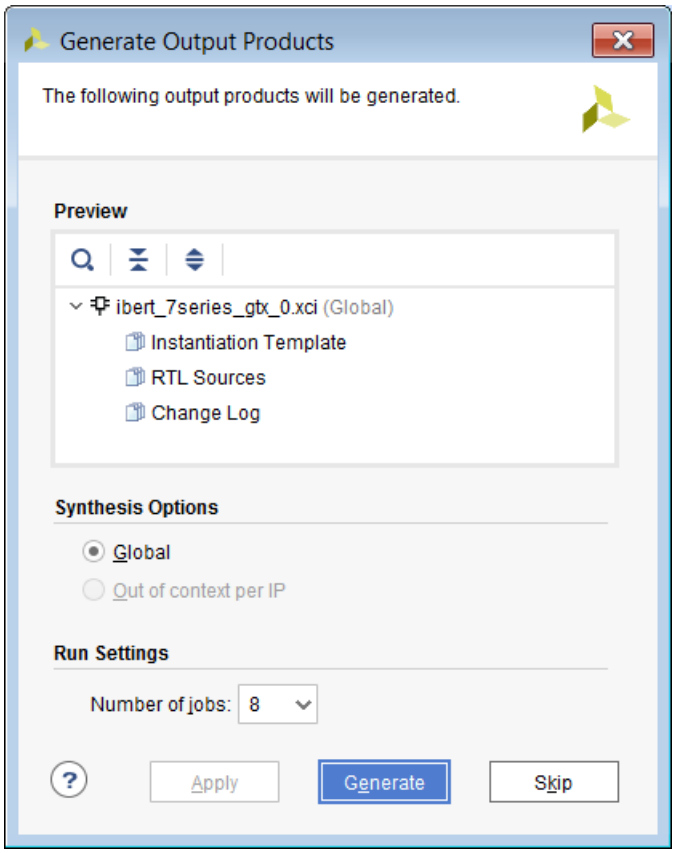

Figure 131: Generate Output Products

9. In the **Sources** window, right-click the IP, and select **Open IP Example Design**.

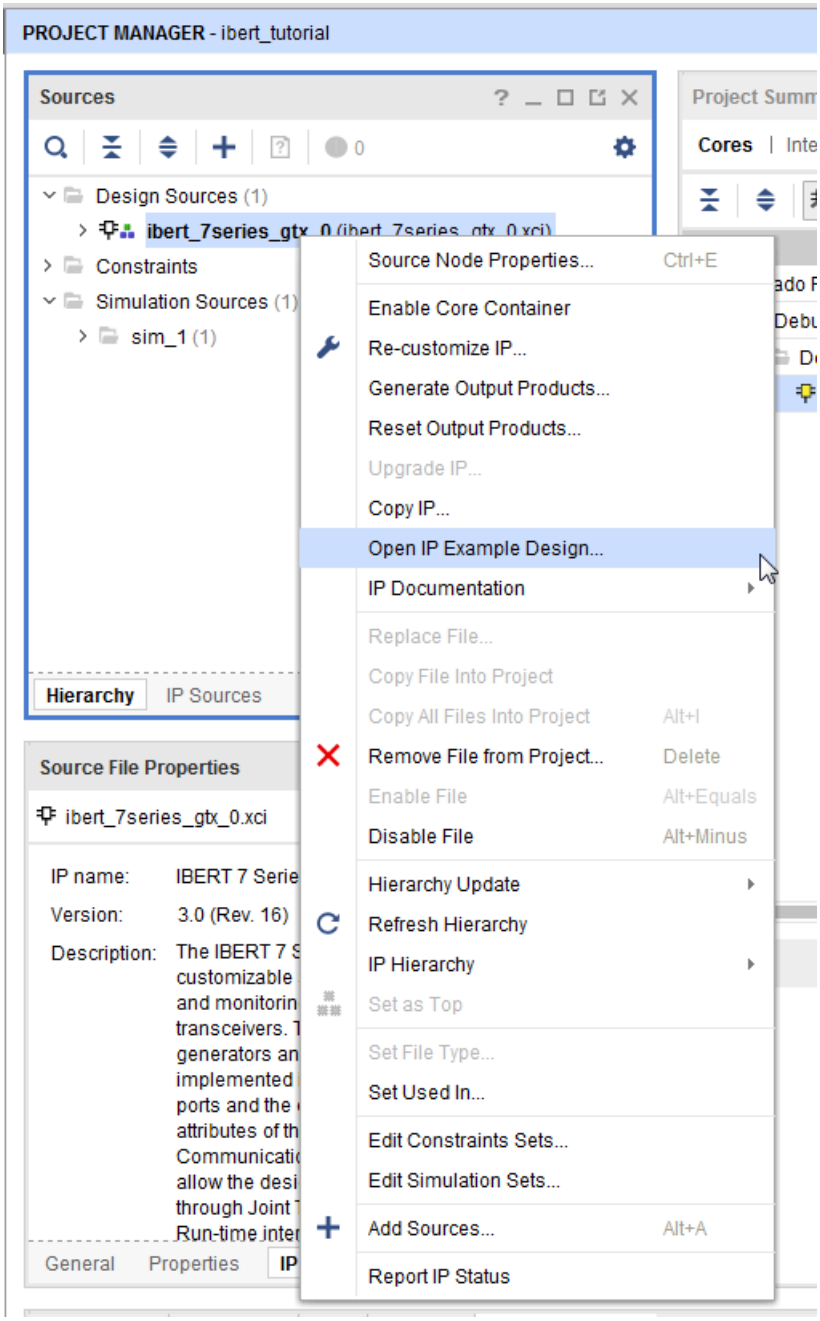

Figure 132: Open Example IP Design Menu Item

10. In the **Open IP Example Design** dialog box, and specify the location of your project directory. Ensure that the **Overwrite existing example project** is selected and click **OK**.

    Note: This opens a new instance of Vivado IDE with the new example design opened.

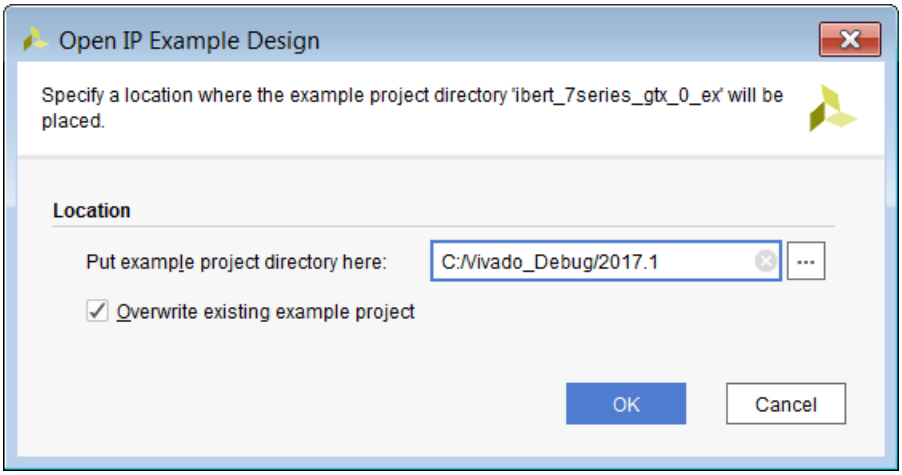

Figure 133: Open IP Example Design Dialog Box

Step 3: Synthesize, Implement and Generate Bitstream for the IBERT design

1. In the newly opened instance of Vivado IDE, click **Generate Bitstream** in the **Flow Navigator**. When the **No Implementation Results Available** dialog box appears. Click **Yes**.

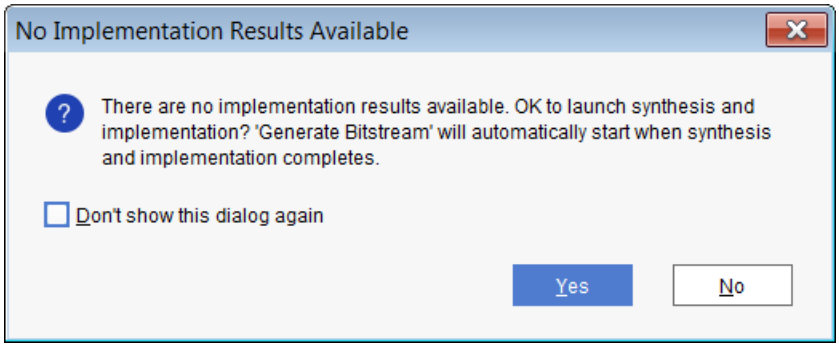

Figure 134: No Implementation Results Available Dialog Box

When the bitstream generation is complete, the **Bitstream Generation Completed** dialog box opens.

2. Select **Open Hardware Manager**, and click **OK**.

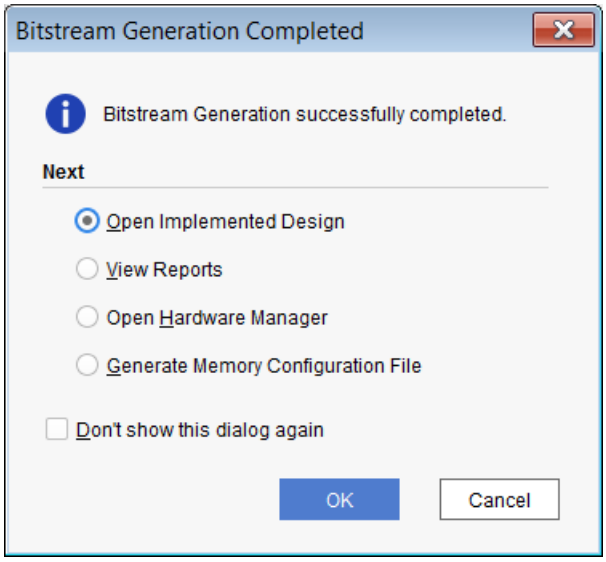

Figure 135: Bitstream Generation Completed Dialog Box

3. The **Hardware Manager** window appears as shown in the following figure.

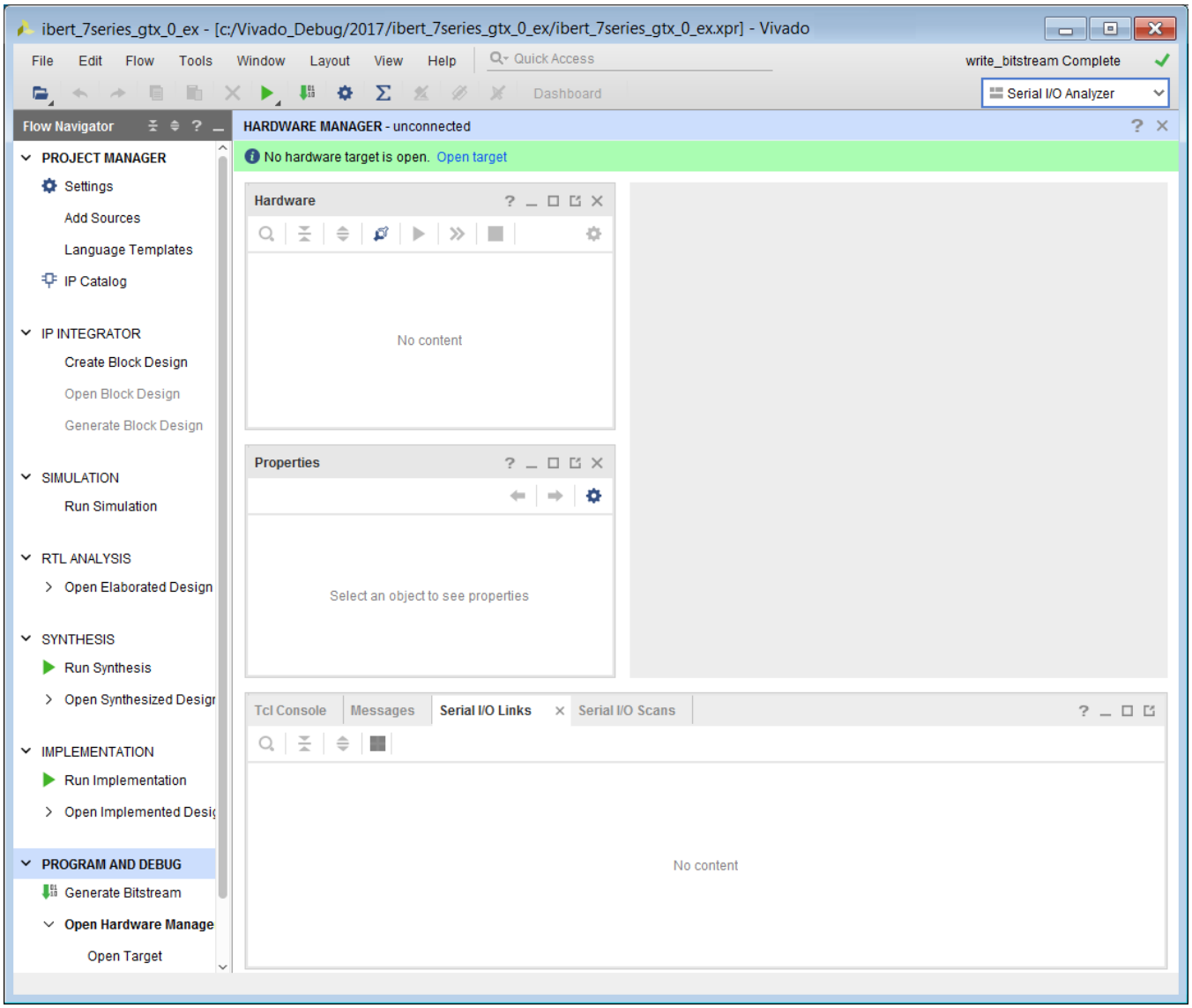

Figure 136: Hardware Manager Window

Step 4: Interact with the IBERT core using Serial I/O Analyzer

In this tutorial step, you connect to the KC705 target board, program the bitstream created in the previous step, and then use the Serial I/O Analyzer to interact with the IBERT design that you created in Step 1. You perform some analysis using various input patterns and loopback modes, while observing the bit error count.

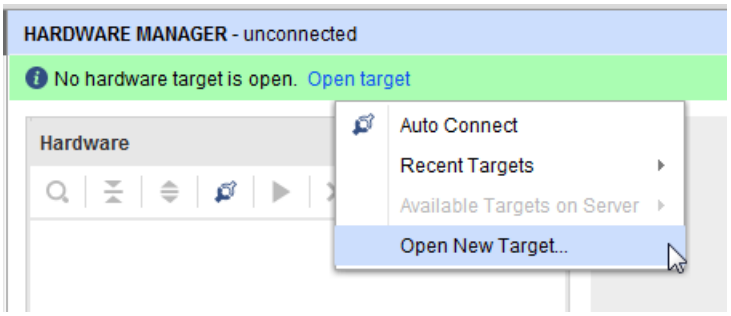

Figure 137: Open a New Hardware Target

1. Click **Open New Target**. When the **Open Hardware Target** wizard opens, click **Next**.

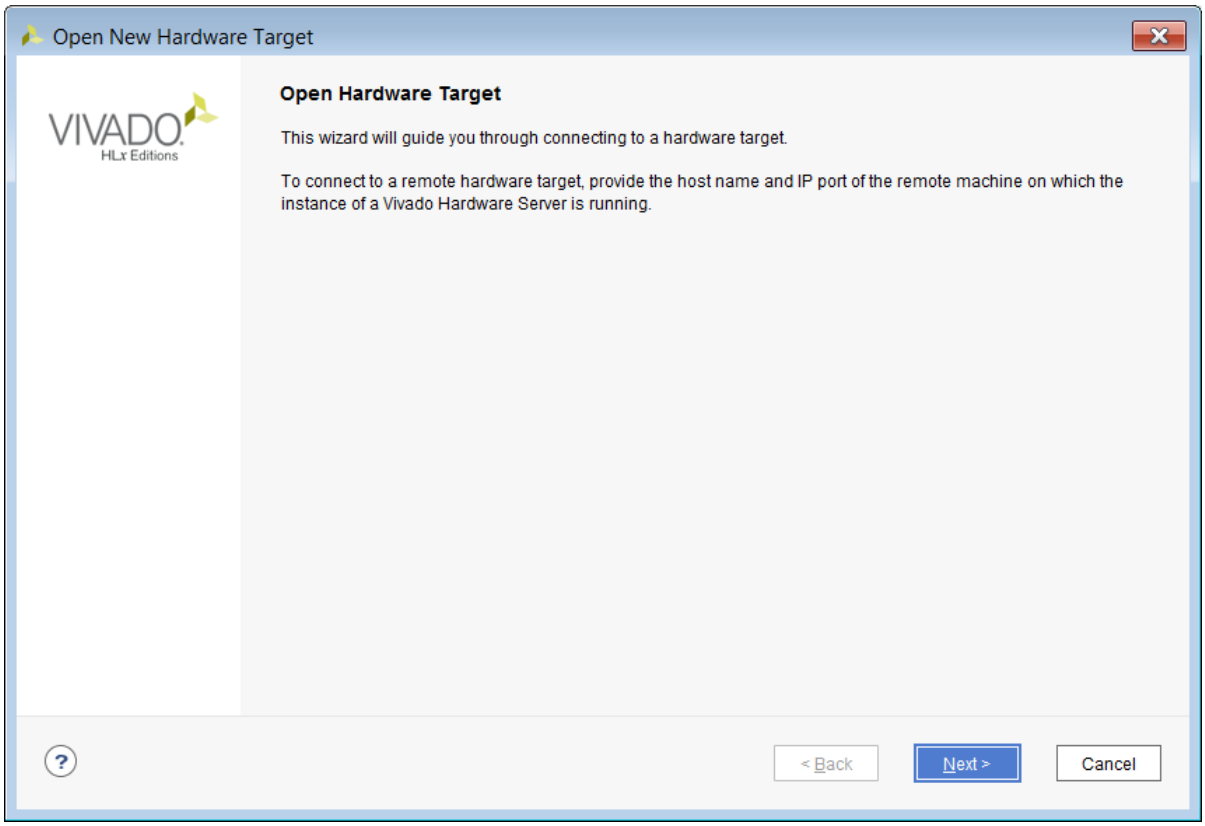

Figure 138: Open New Hardware Target Wizard

2. In the **Connect to** field, choose **Local server**. Click **Next**.

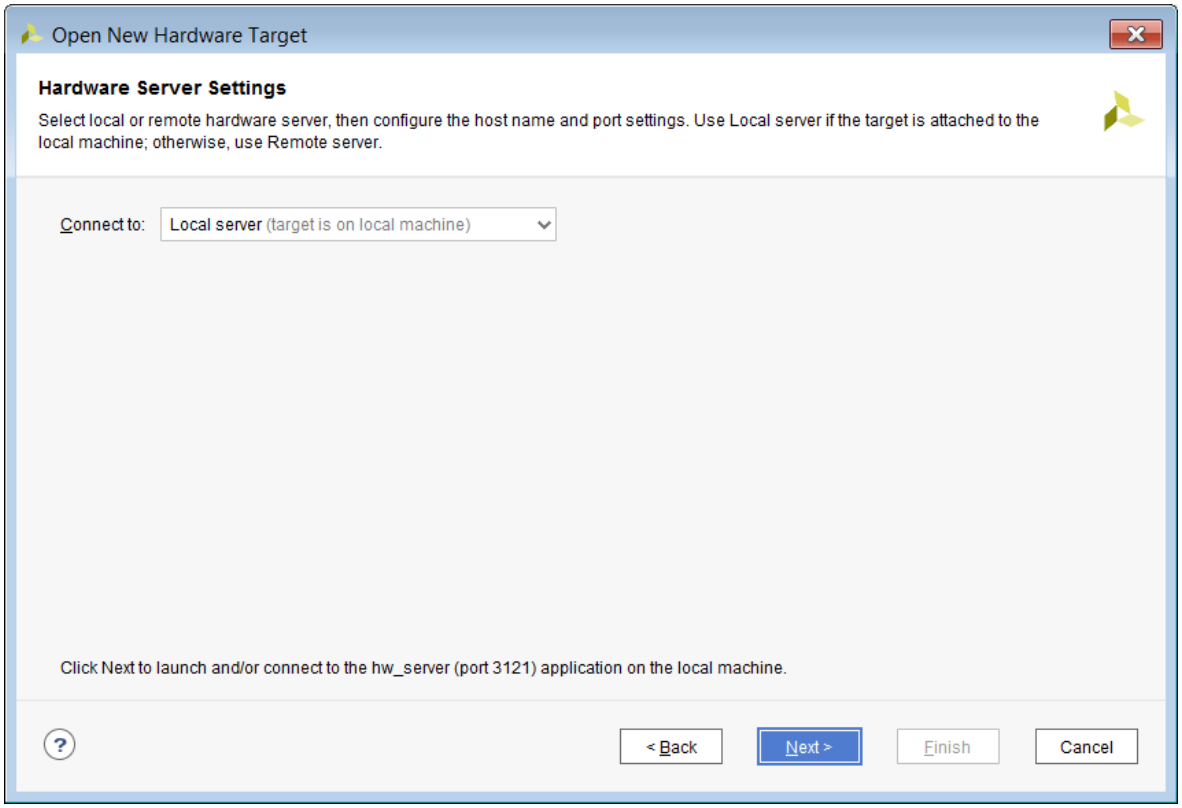

Figure 139: Vivado CSE Server Name Page

3. In the **Select Hardware Target** page, and click **Next**.

   There is only one target board in this case to connect to, so that the default is selected.

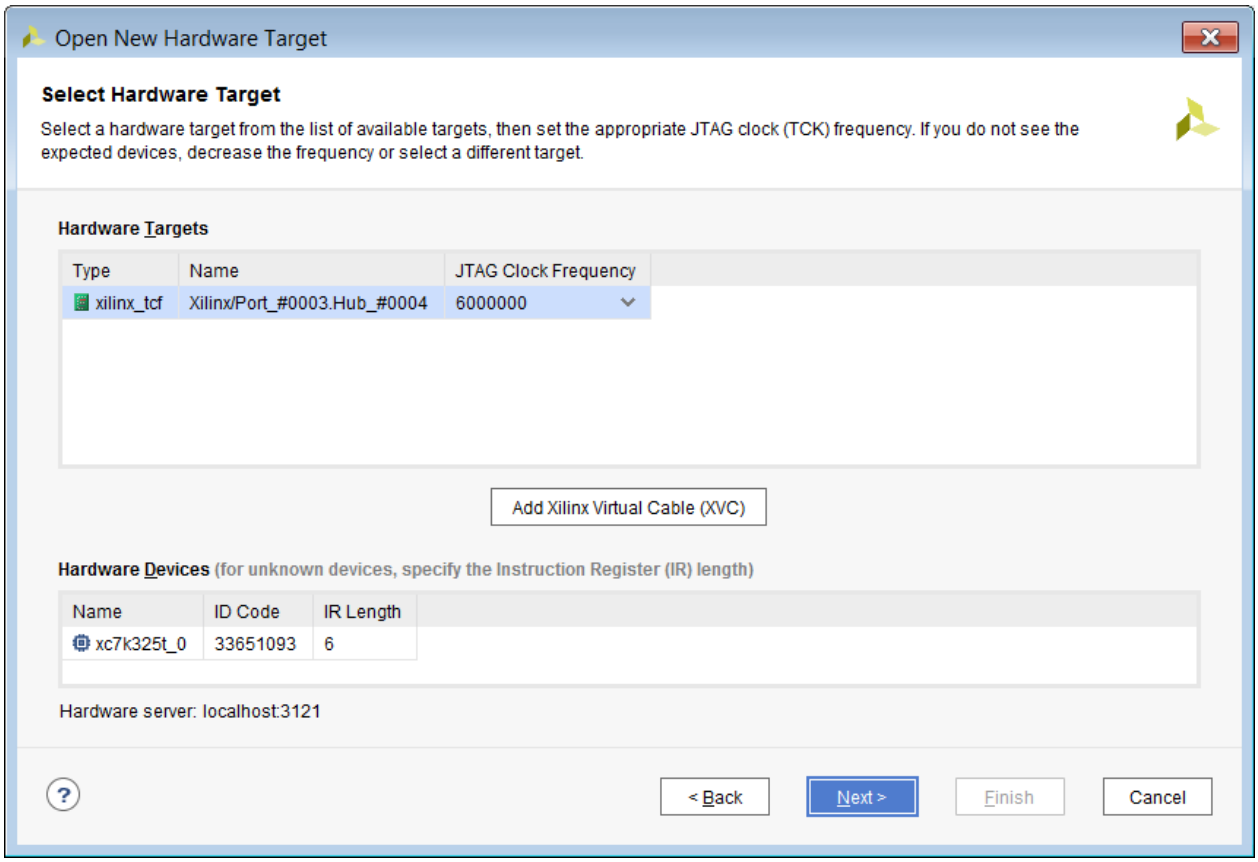

Figure 140: Select Hardware Target Page

4. In the **Open Hardware Target Summary** page, review the options that you selected. Click **Finish**.

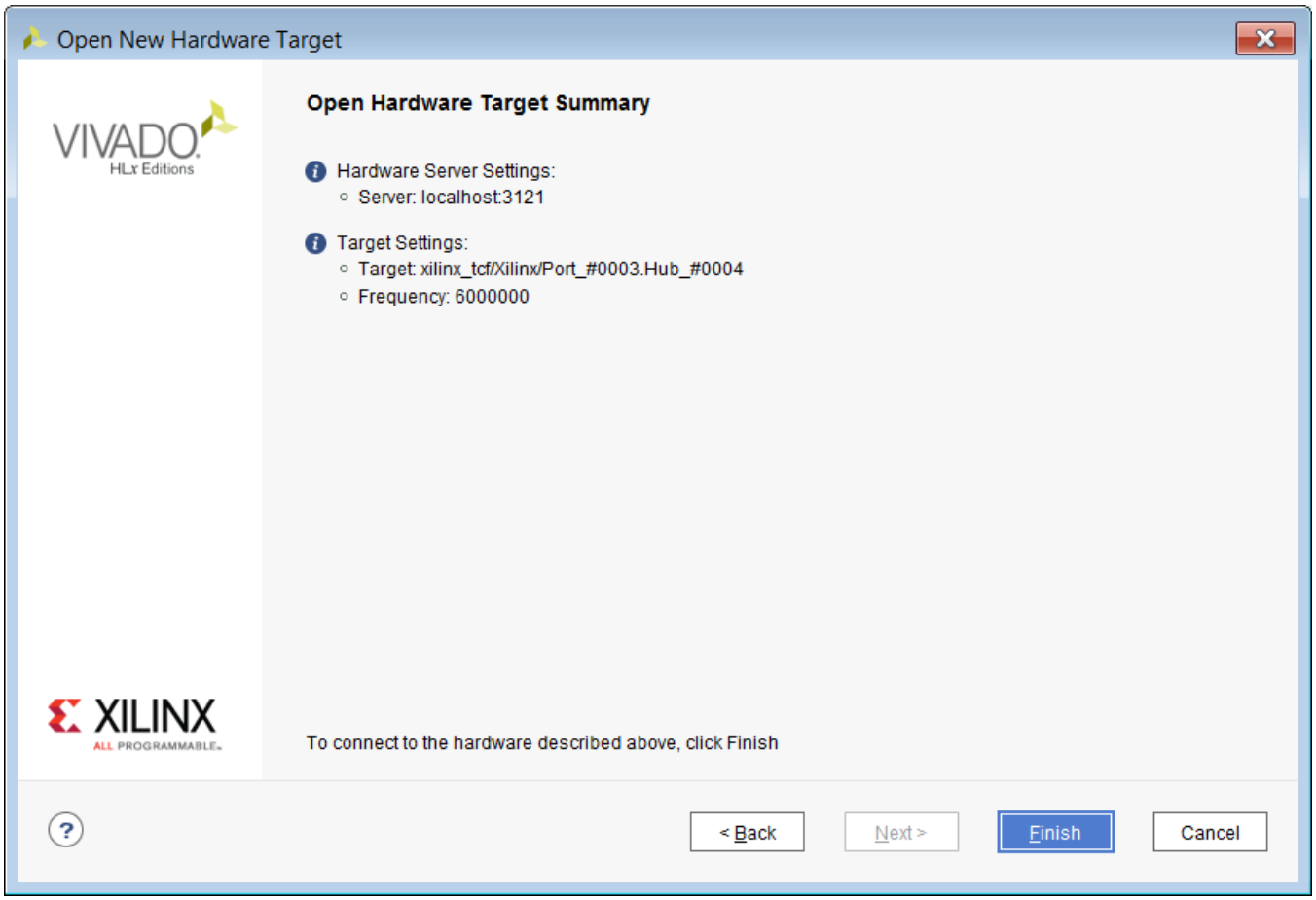

Figure 141: Open Hardware Target Summary Dialog Box

5. The **Hardware** window in Vivado IDE should show the status of the target FPGA device on the KC705 board.

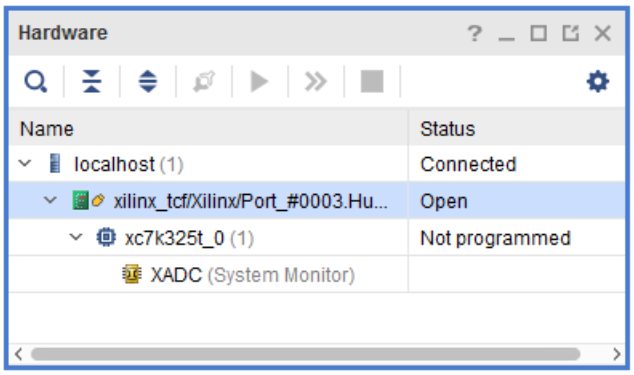

Figure 142: Hardware Window Showing the XC7K325T Device on the KC705 Board

6. Select **XC7K325T_0(0)** in the **Hardware** window, right-click and select **Program Device**.

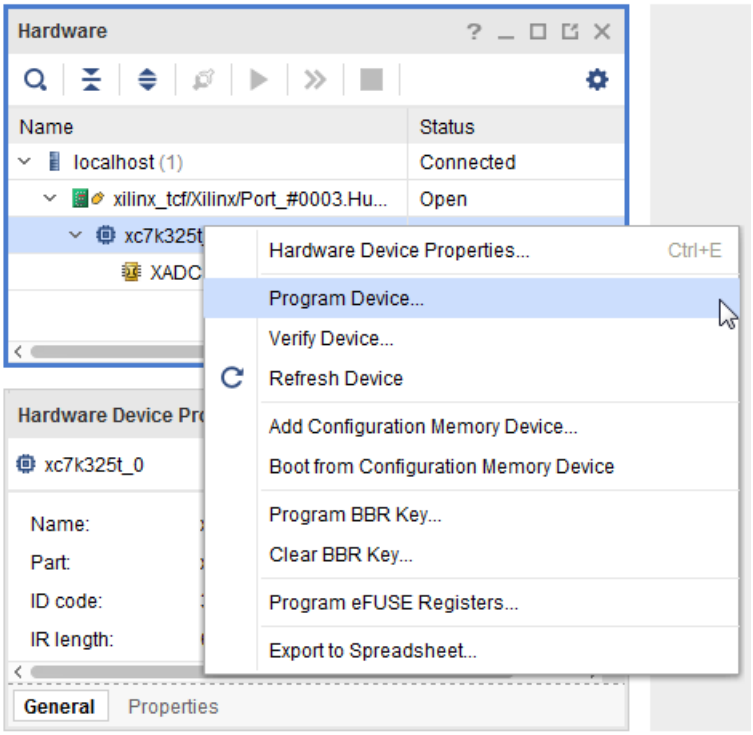

Figure 143: Program Target Device

7. The **Program Device** dialog box opens. Make sure that the correct `.bit` file is selected, and click **Program**.

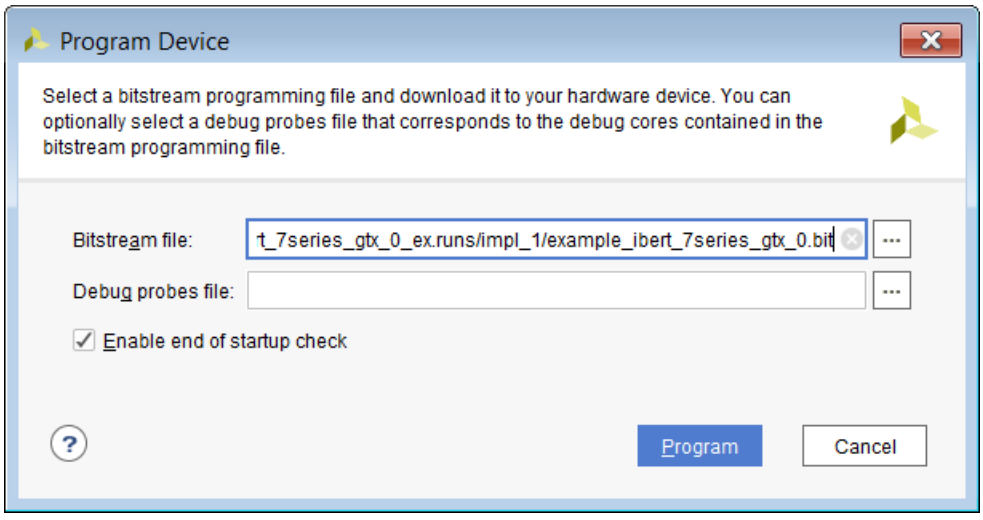

Figure 144: Program Device Dialog Box with .bit File

8. The **Hardware** window now shows the IBERT IP that you customized and implemented from the previous steps. It contains two QUADS each of which has four GTX transceivers. These components of the IBERT were detected while scanning the device after downloading the bitstream. If you do not see the QUADS then select the **XC7K325** device, right-click and select **Refresh Device**.

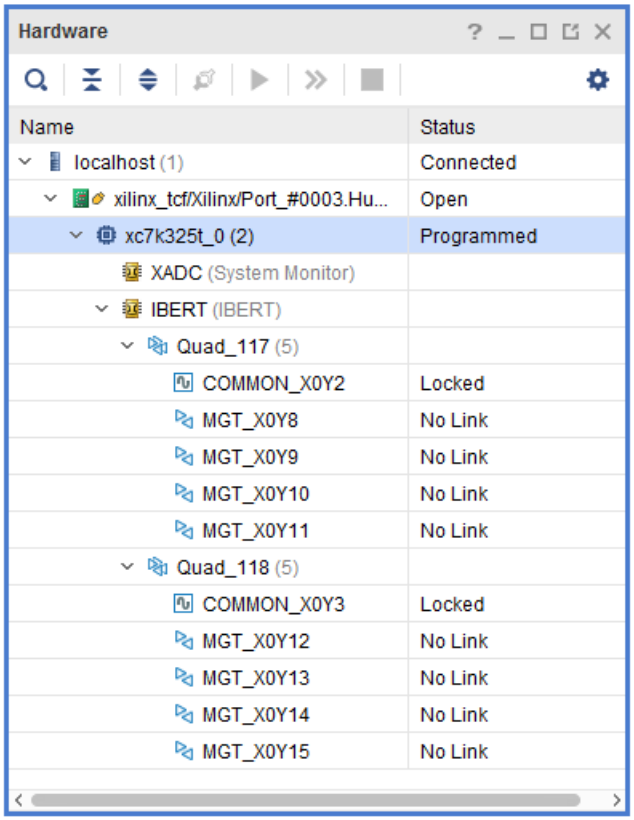

Figure 145: The Hardware Window Showing the QUADS after Device Programming

9. Next, create links for all eight transceivers. Vivado Serial I/O analyzer is a link-based analyzer, which allows users to link between any transmitter and receiver GTs within the IBERT design. For this tutorial, simply link the TX and RX of the same channel. To create a link, right-click the **IBERT Core** in the **Hardware** window and click **Create Links**.

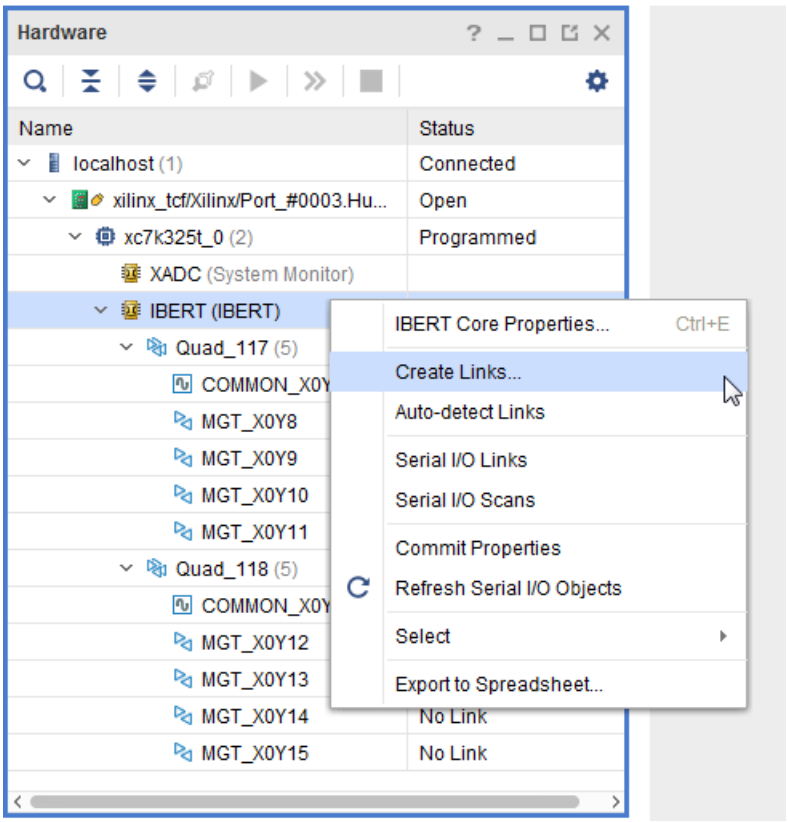

Figure 146: Create Links

The **Create Links** dialog box opens.

10. Ensure the first transceiver pairs (MGT_X0Y8/TX and MGT_X0Y8/RX) are selected.

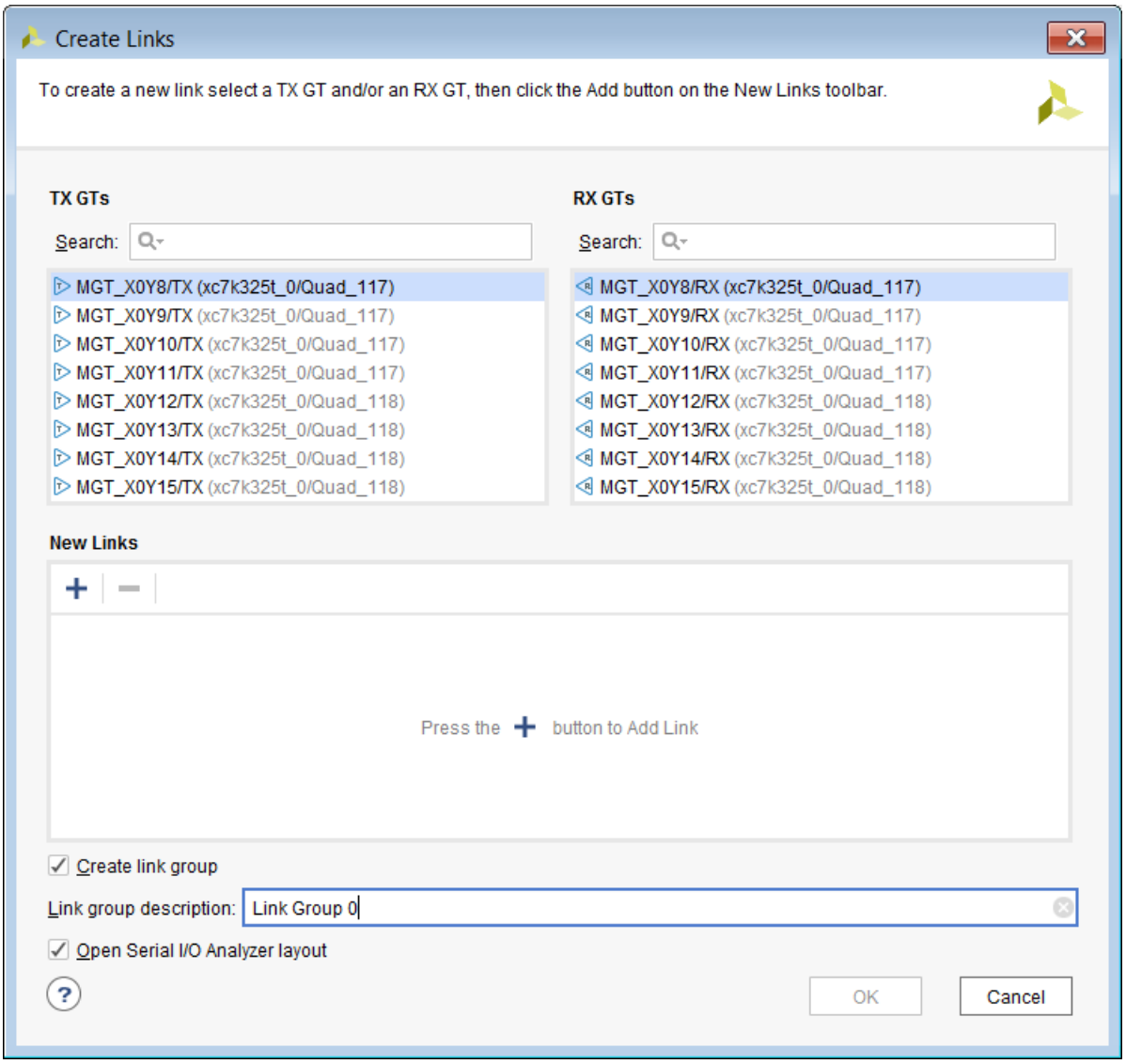

Figure 147: Selecting the Transceiver Pairs for Creating New Links

11. Click the "+" button add a new link. In the **Link group description** field, type **Link Group SMA**. Select the **Internal Loopback** check box.

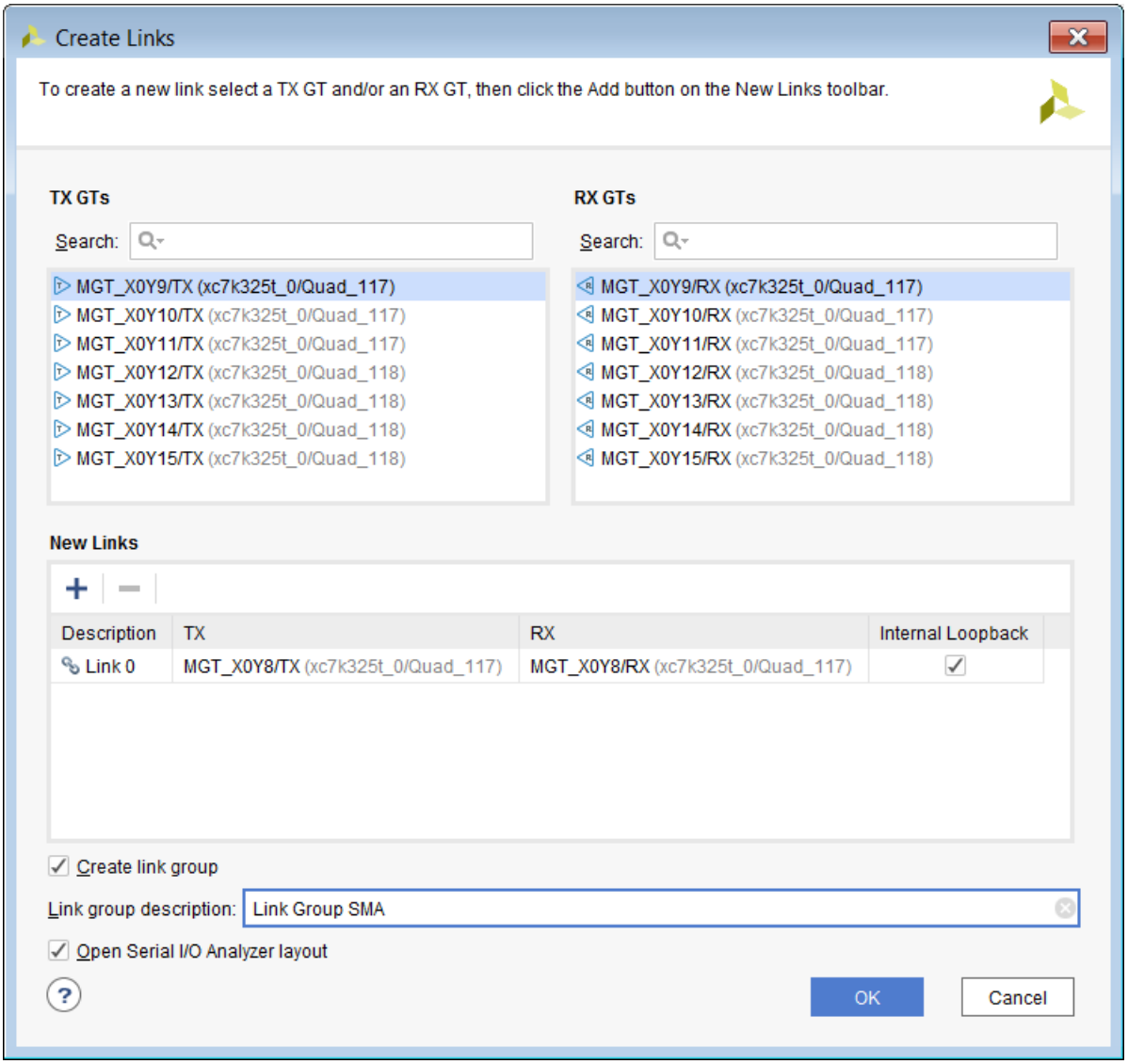

Figure 148: Create Links Dialog Box

For the first link group, call this Link Group SMA as this is the only transceiver channel that is linked through the SMA cables. The new link shows up in the **Links** window.

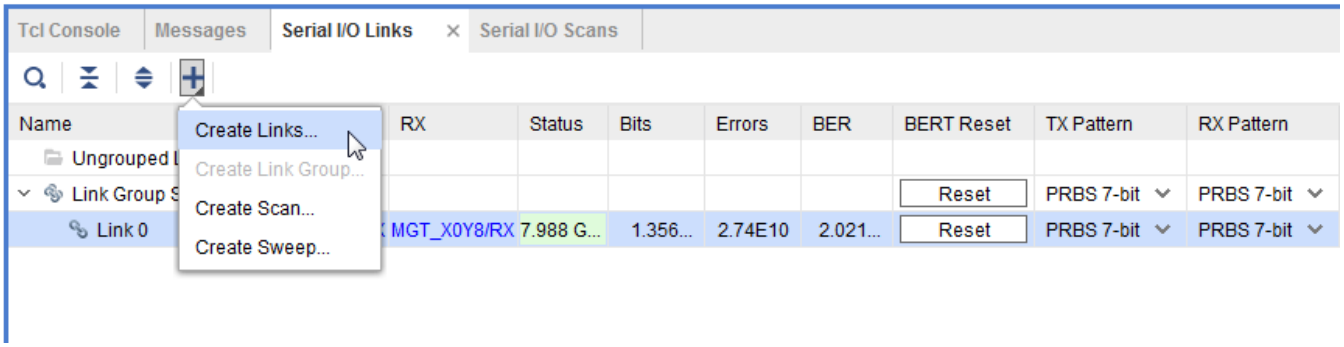

Figure 149: Create Link Groups for Other Transceiver Pairs

12. Click **Create Link** again to create link groups for the rest of the transceiver pairs. To do this ensure that the transceiver pairs are selected, and click the **+** sign icon (add new link) repeatedly, until all the links have been added to the new link group called **Link Group Internal Loopback**. Click **OK**.

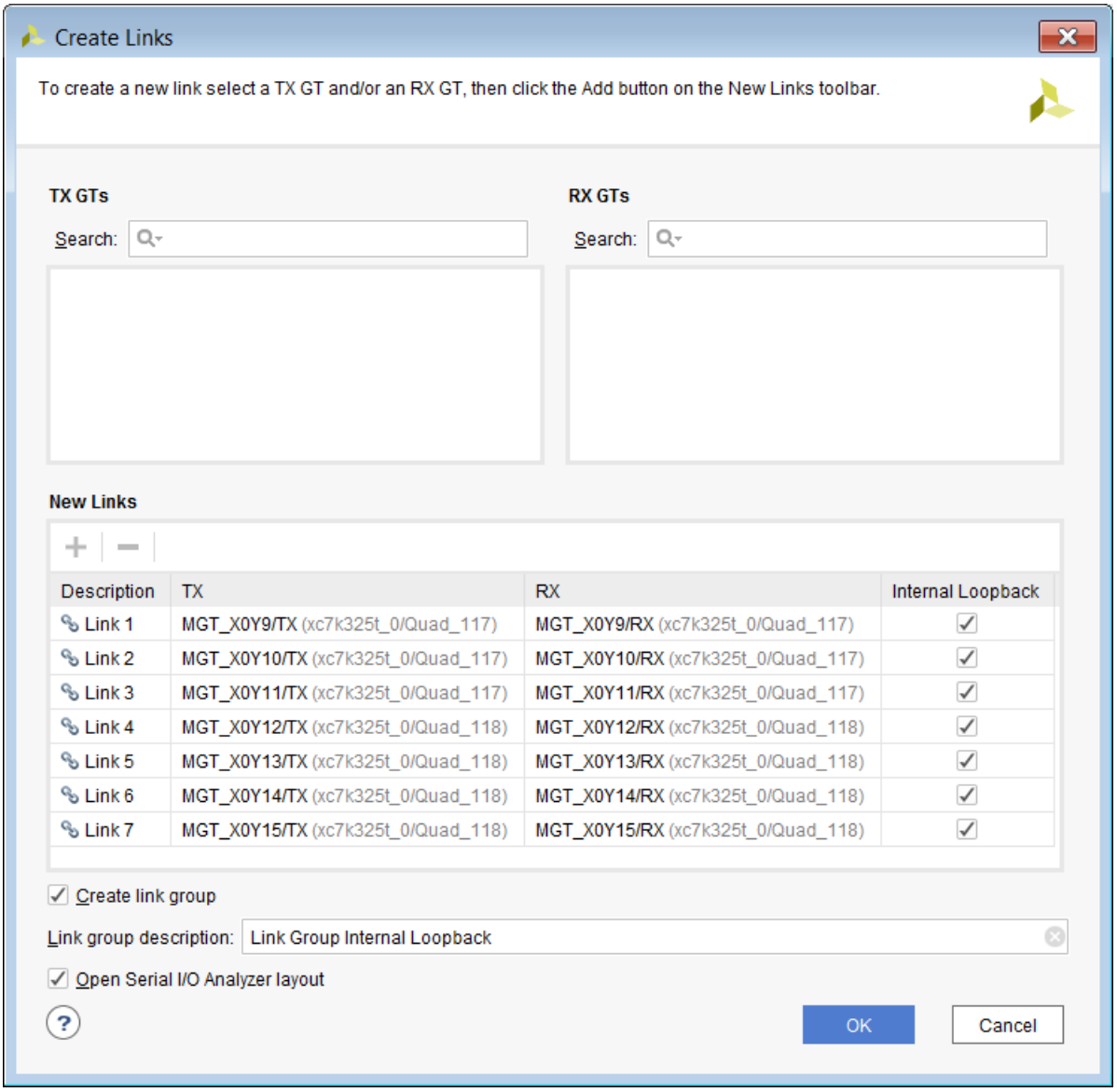

Figure 150: Create Link Dialog Box to Create the Second Link Group

13. After the links have been created, they are added to the **Links** window as shown.

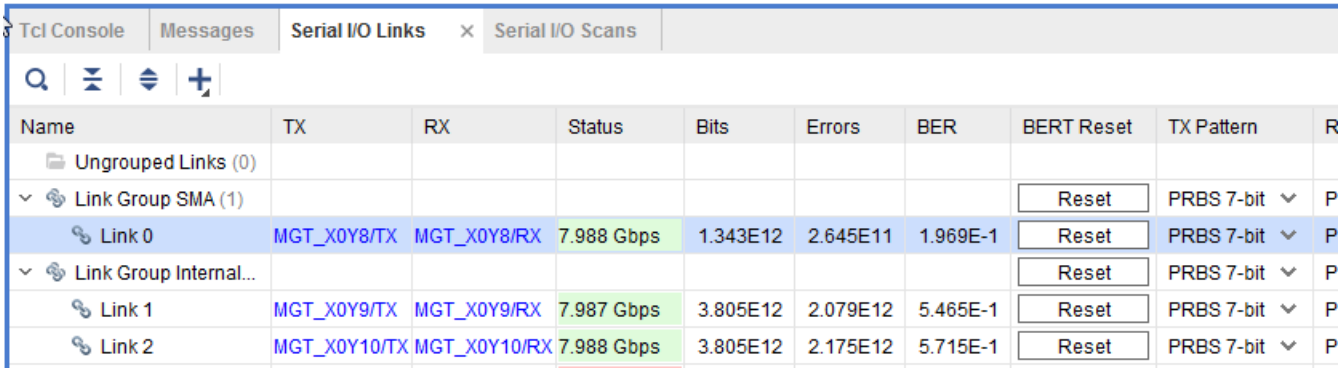

Figure 151: Links Window after Link Groups are Created

The status of the links indicate an **8.0 Gbps** line rate.

For more information about the different columns of the **Links** windows, see the *Vivado Design Suite User Guide: Programming and Debugging (UG908)*.

14. Change the GT properties of the rest of the transceivers as described above.

15. Next, create a 2D scan. Click **Create Scan** in the **Links** window.

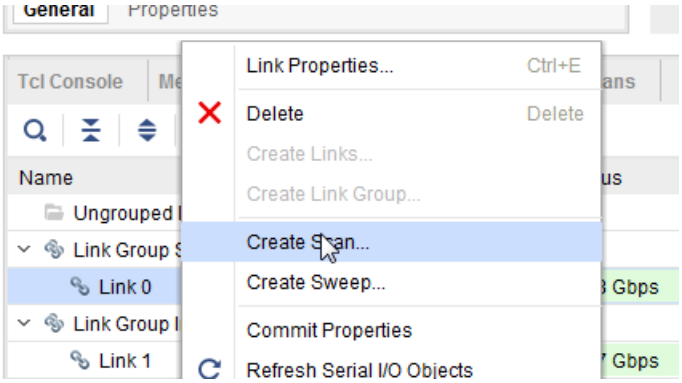

Figure 152: Creating a 2D Scan for Link 1

The **Create Scan** dialog box opens. In this dialog box, you can change the various scan properties. In this case, leave everything to its default value and click **OK**. For more information on the scan properties, see *Vivado Design Suite User Guide: Programming and Debugging (UG908)*.

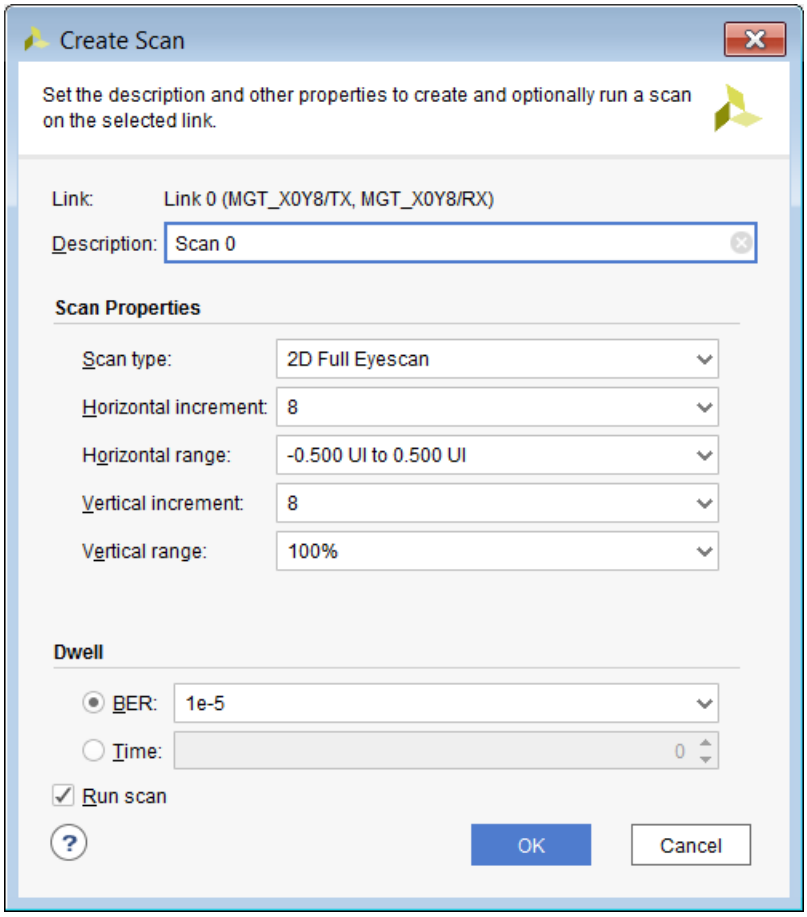

Figure 153: The Create Scan Dialog Box

The **Scan Plot** window opens as shown in the following figure.

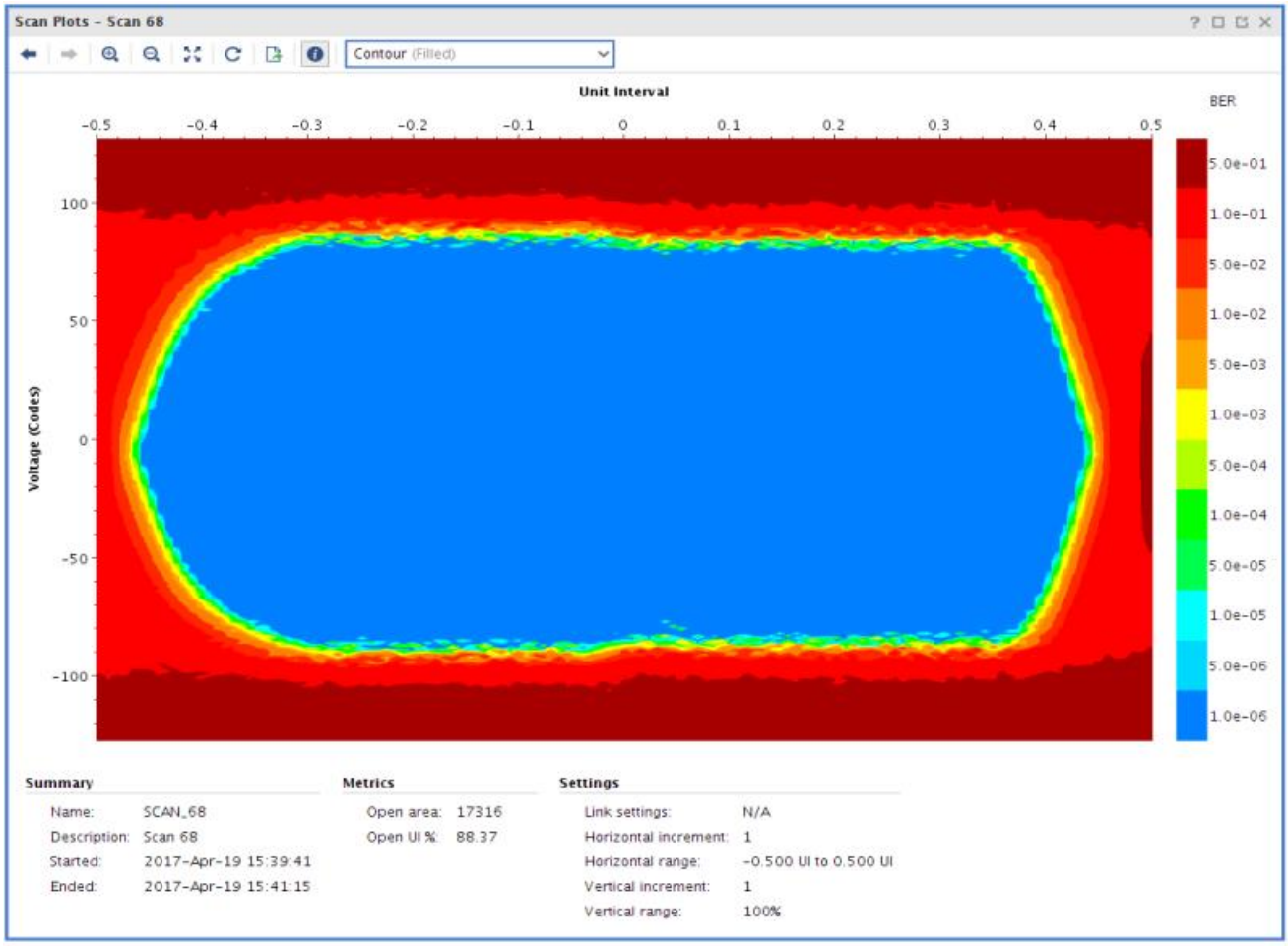

Figure 154: 2D Scan Plot

The 2D Scan Plot is a heat map of the BER value.

You can also perform a Sweep test on the links that you created earlier.

16. In the **Links** window, highlight **Link 0** under the Link called Link **Group SMA**, right-click and select **Create Sweep**.

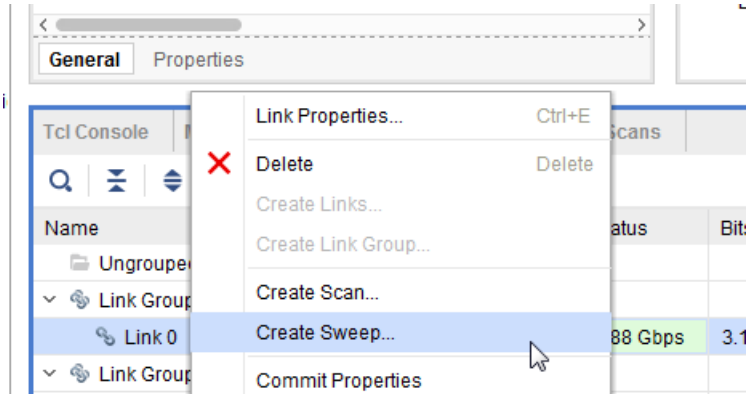

Figure 155: Create a Sweep Test

17. The **Create Sweep** dialog box opens, as shown below. Various properties for the Sweep test can be changed in this dialog box. Leave all the values to its default state and click **OK**.

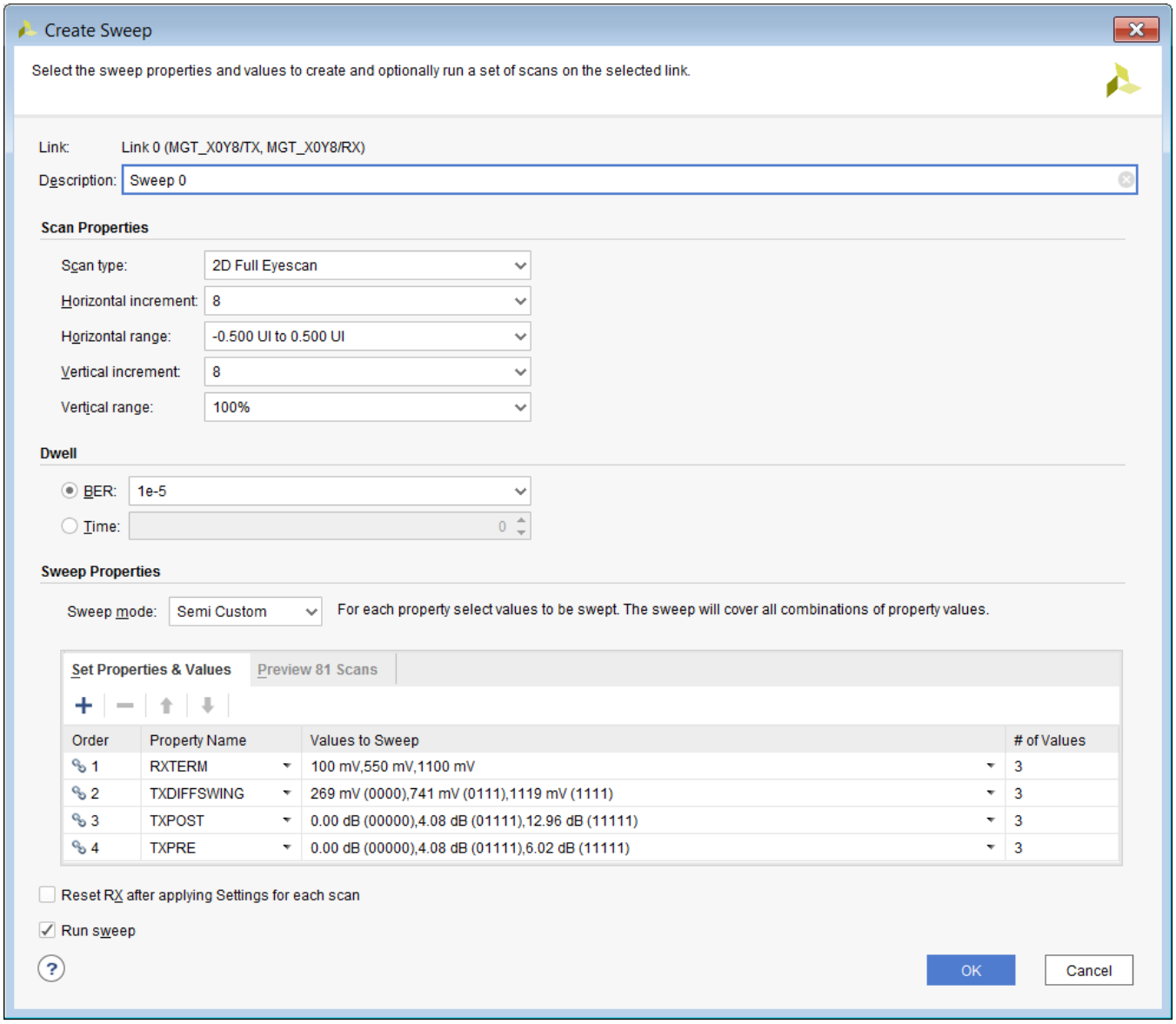

Figure 156: Create Sweep Dialog Box

Because here are four different Sweep Properties and each of these properties has three different values (as seen in the **Values to Sweep** column), a total number of 81 sweep tests are carried out. The **Scans** window shows the results of all the scans that have been done for the selected link.

CAUTION! *Since there are 81 scans to be done, it could be a few minutes before all the scans are complete.*

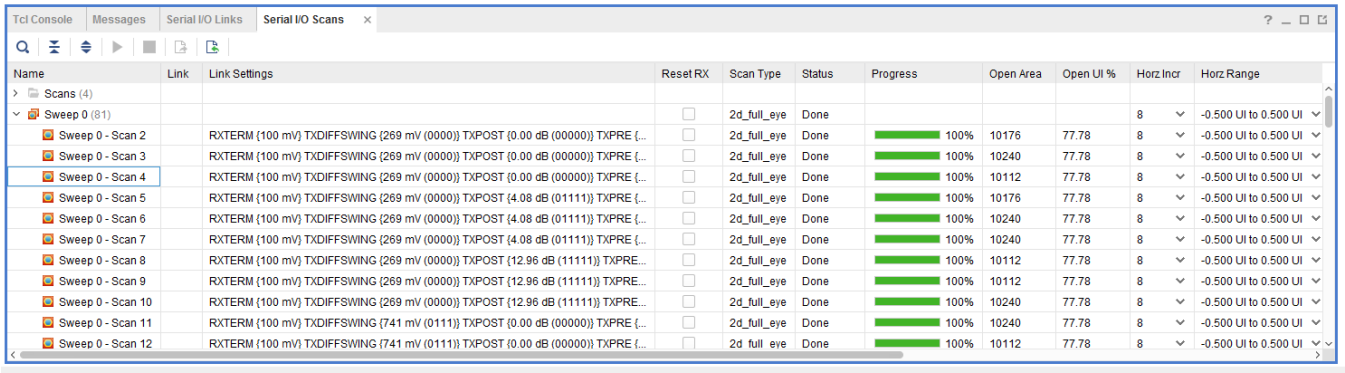

Figure 157: Sweep Test Results in the Scans Window

To see the results of any of the scans that have been performed, highlight the scan, right-click, and select **Display Scan Plots**.

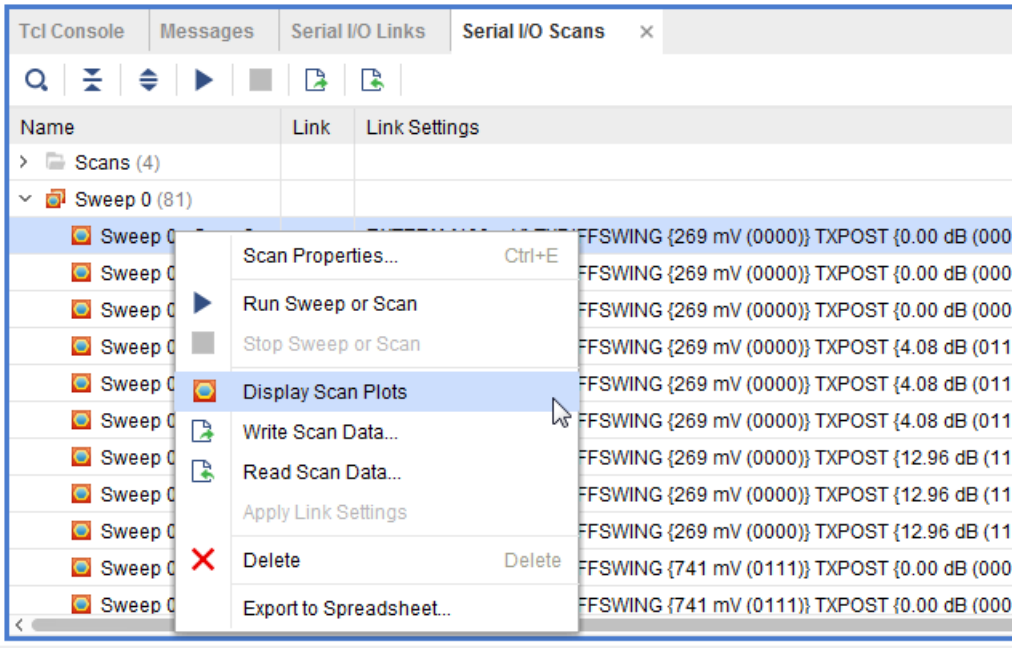

Figure 158: Displaying Scan Plots

The **Scan Plots** window opens showing the details of the scan performed.

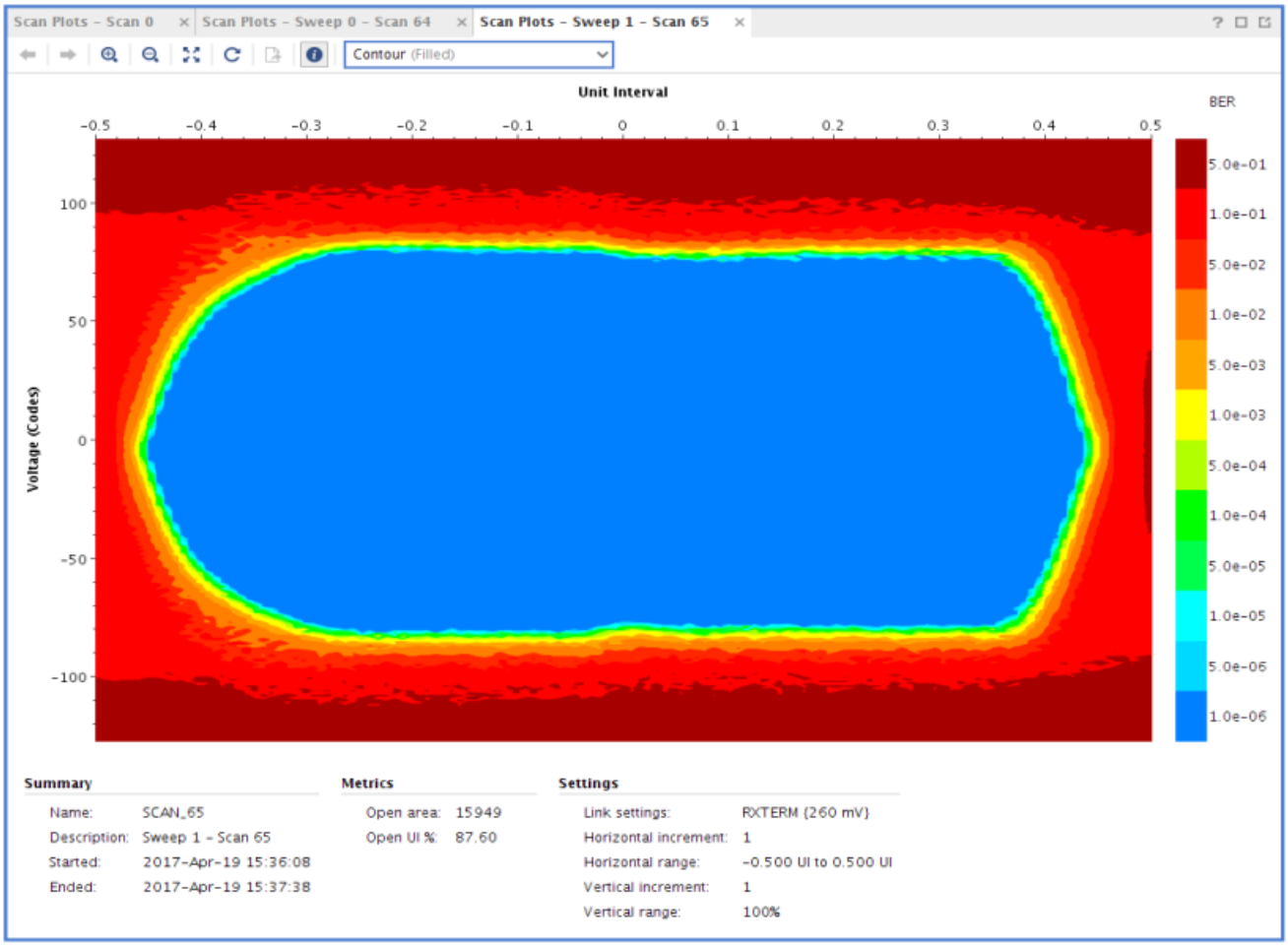

Figure 159: Analyzing the Results of Individual Scans

Lab 9: Using Vivado ILA Core to Debug JTAG-AXI Transactions

Introduction

The purpose of this tutorial is to provide a very quick and easy to reproduce introduction to inserting an ILA core into the JTAG to AXI Master IP core example design, and using the ILA's advanced trigger and capture capabilities.

What is the JTAG to AXI Master IP core?

The LogiCORE™ IP JTAG-AXI core is a customizable core that can generate AXI transactions and drive AXI signals internal to FPGA at run-time. This supports all memory-mapped AXI interfaces (except AXI4-Stream) and Lite protocol and can be selected using a parameter. The width of AXI data bus is customizable. This IP can drive any AXI4-Lite or Memory Mapped Slave directly. This can also be connected as master to the interconnect. Run-time interaction with this core requires the use of the Vivado® logic analyzer feature.

Key Features

- AXI4 master interface
- Option to select AXI4-Memory Mapped and AXI4-Lite interfaces
- User controllable AXI read and write enable
- User Selectable AXI datawidth : 32 and 64
- User Selectable AXI ID width up to four bits
- Vivado logic analyzer Tcl Console interface to interact with hardware

Additional Documentation

LogiCORE IP JTAG AXI Master v1.0 Product Guide (AXI) (PG174) contains more information the JTAG to AXI Master IP core.

Design Description

This section has three steps as follows:

1. Opening the JTAG to AXI Master IP Example Design project and adding MARK_DEBUG to the AXI interface connection. Inserting an ILA core into the design and configuring it for advanced trigger is also included in this step.
2. Programming the KC705 board and interacting with the JTAG to AXI Master IP core.
3. Using the ILA Advanced Trigger Feature to Trigger on an AXI Read Transaction.

Step 1: Opening the JTAG to AXI Master IP Example Design and Configuring the AXI Interface Debug Connections

To create a project, use the New Project wizard to name the project, add RTL source files and constraints, and specify the target device.

1. Invoke the Vivado IDE.
2. In the **Quick Start** tab, click **Create Project** to start the New Project wizard. Click **Next**.
3. In the **Project Name** page, name the new project **jtag_2_axi_tutorial** and provide the project location (`C:/jtag_2_axi_tutorial`). Ensure that **Create Project Subdirectory** is selected. Click **Next**.
4. In the **Project Type** page, specify the **Type of Project** to create as **RTL Project**. Click **Next**.
5. In the **Add Sources** page, click **Next**.
6. In the **Add Constraints** page, click **Next**.

7. In the **Default Part** page, shown in the following figure, choose **Boards** and choose the **Kintex-7 KC705 Evaluation Platform**. Click **Next**.

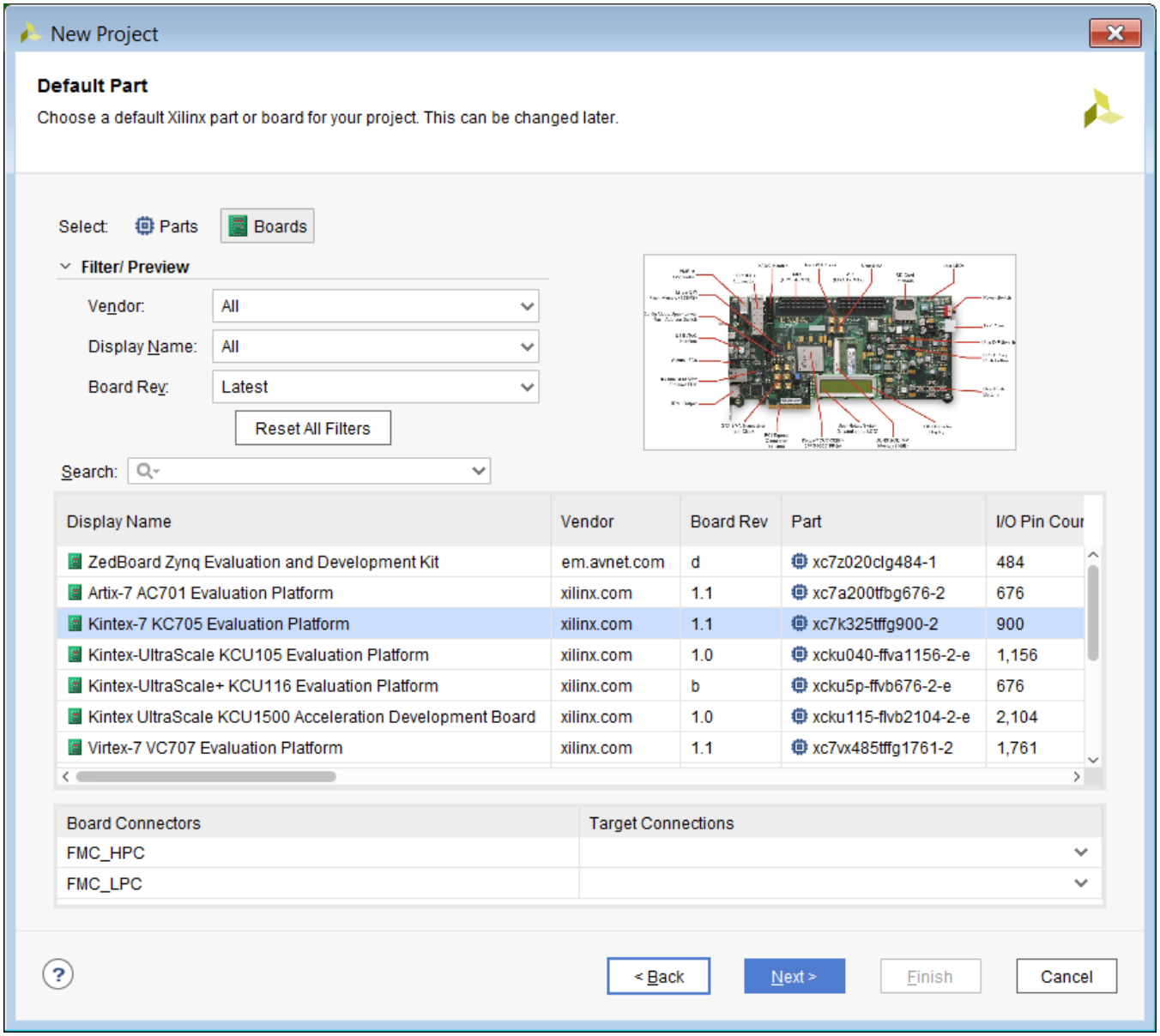

Figure 160: Choosing the Kintex-7 KC705 Evaluation Platform Board

8. In the **New Project Summary** page, shown in the following figure, click **Finish**.

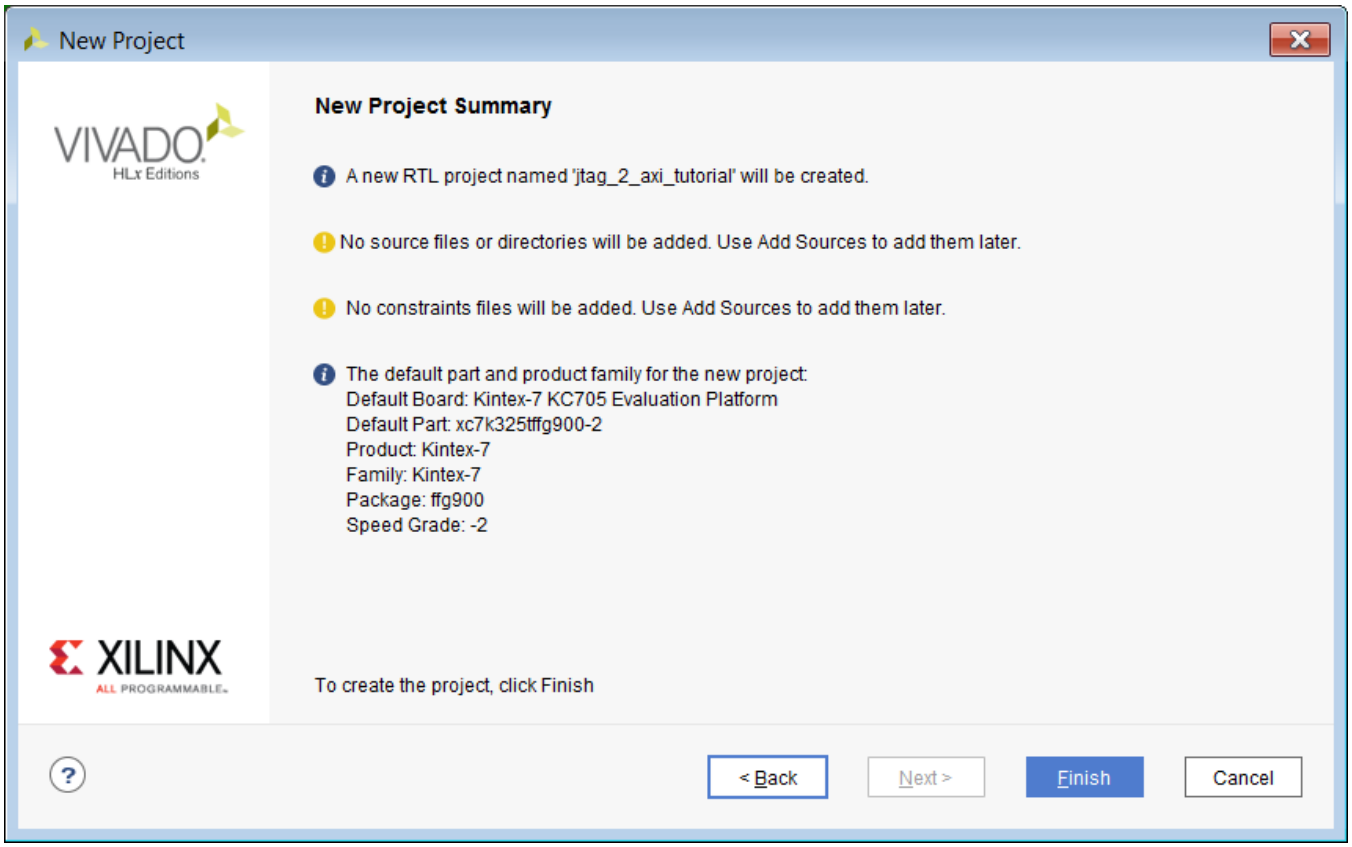

Figure 161: New Project Summary

9. In the leftmost panel of the **Flow Navigator** under **Project Manager**, click **IP Catalog.**

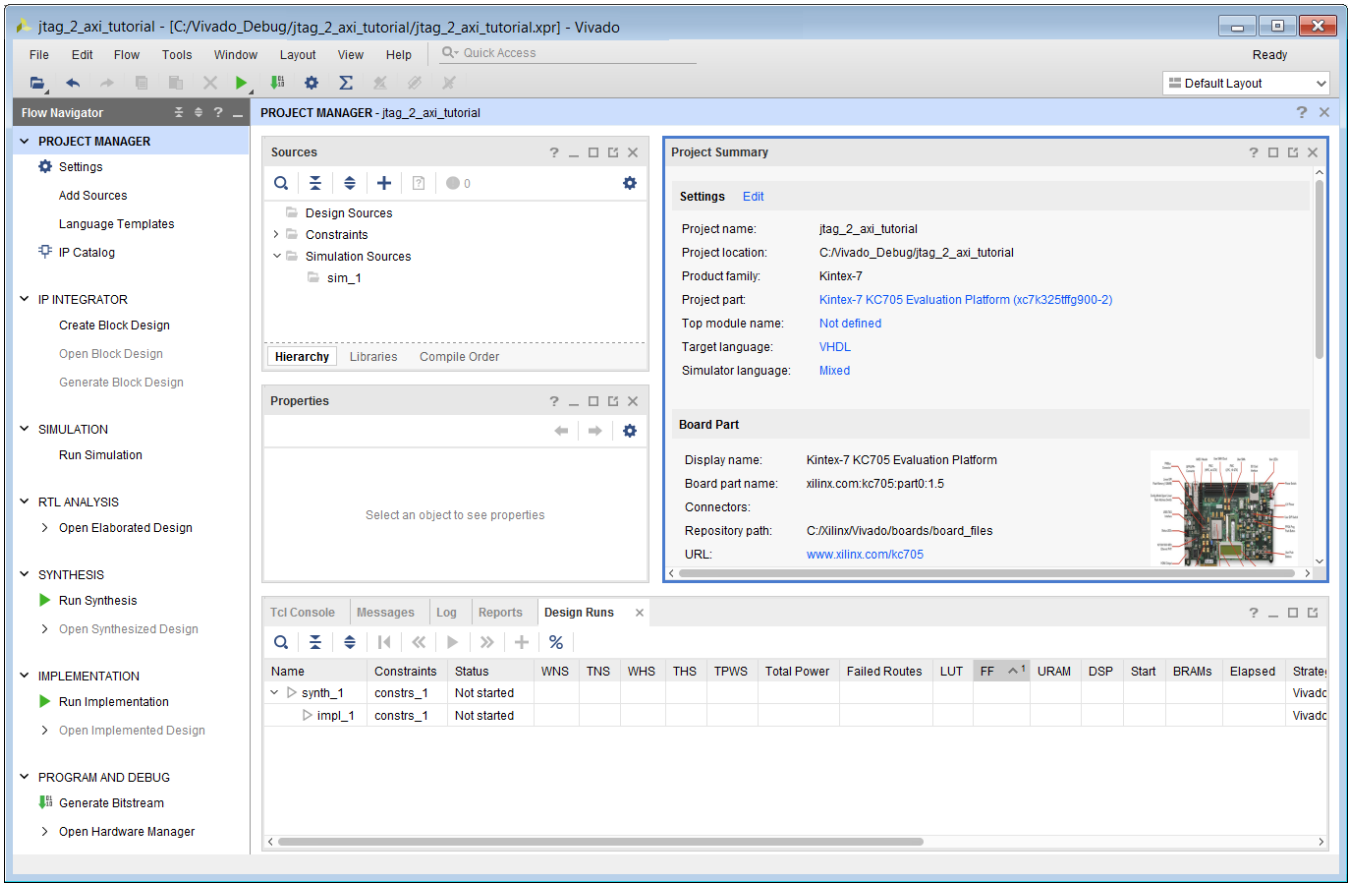

Figure 162: IP Catalog from Flow Navigator

10. In the **Search** field on the upper left of the **IP Catalog** tab, type in **JTAG to AXI**.

***Note**: The JTAG to AXI Master core shows up under the **Debug & Verification** > **Debug** category.*

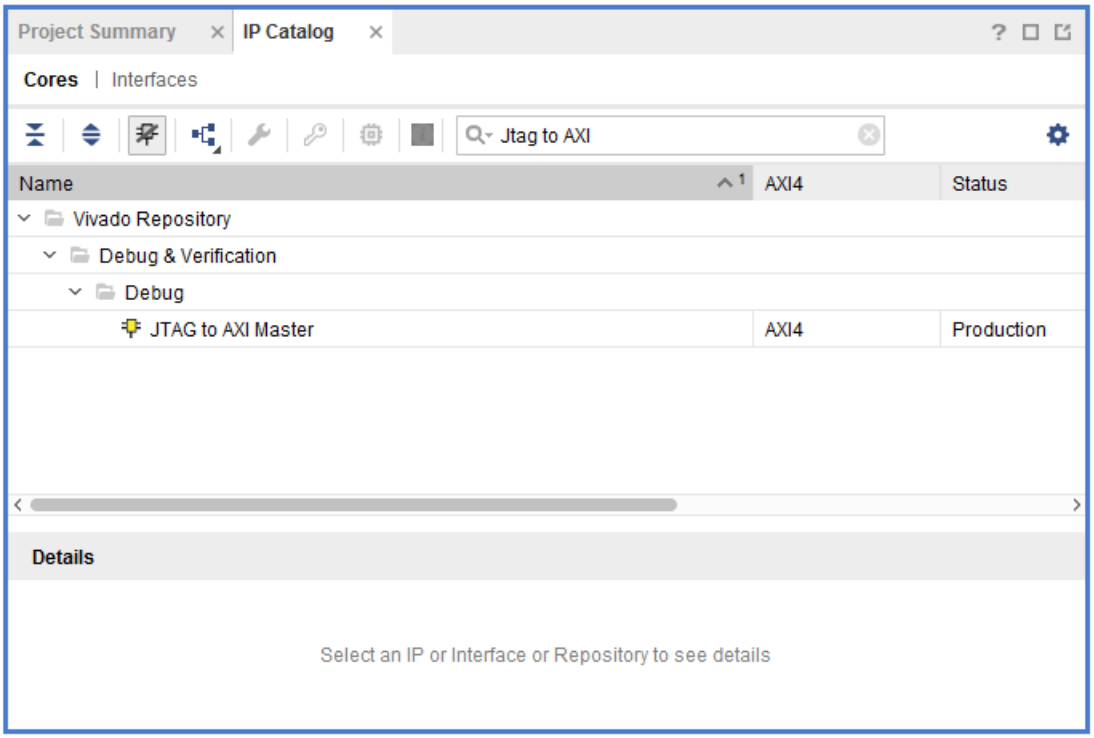

Figure 163: JTAG to AXI Master IP Core

11. Double-click **JTAG to AXI Master** core. The **Customization** dialog of the core appears. Accept the default core settings by clicking **OK**.

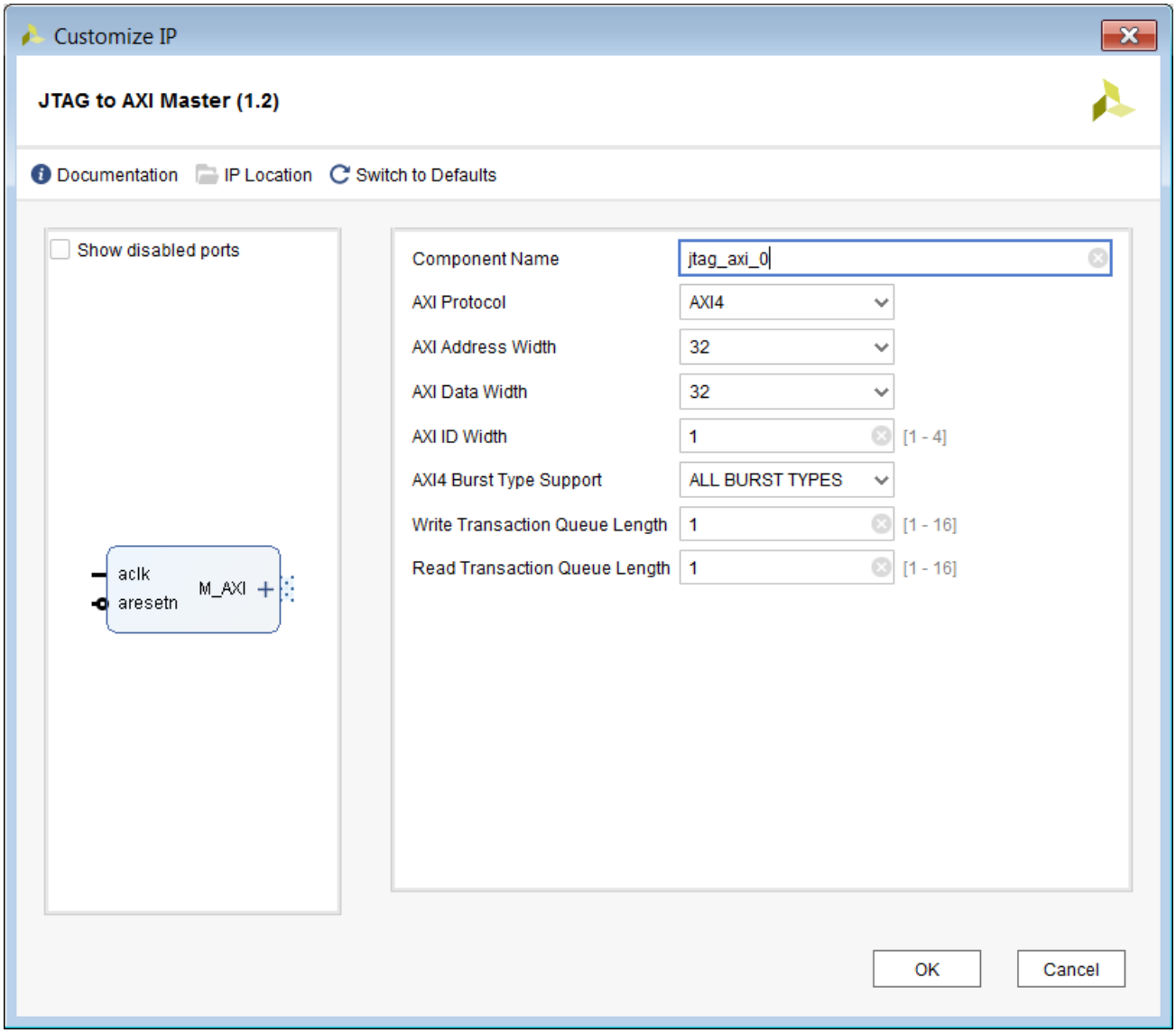

Figure 164: JTAG to AXI Master Customization Dialog

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

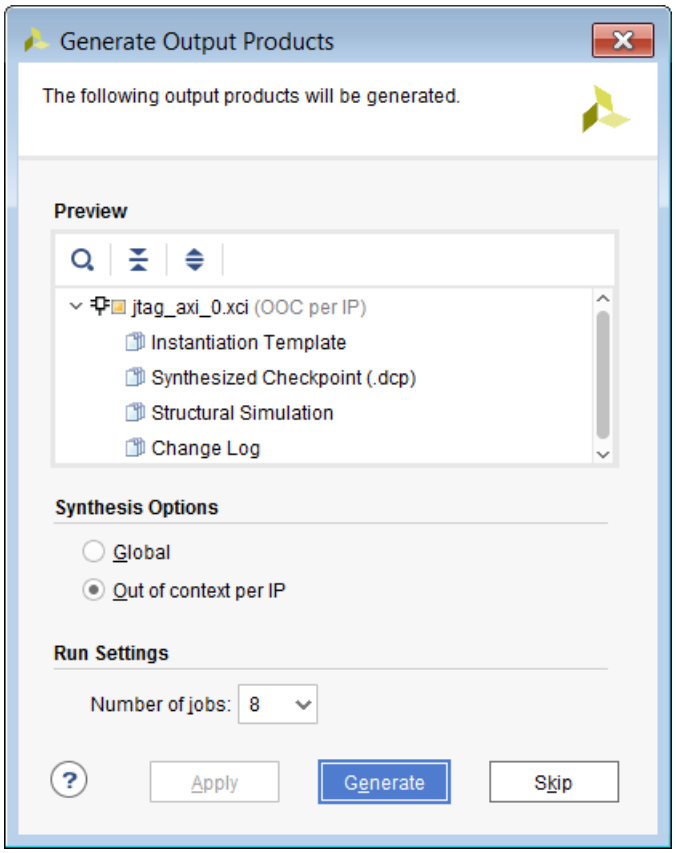

Figure 165: Generate Output Products Dialog Box

13. The **jtag_axi_0 IP** core is inserted into the design.

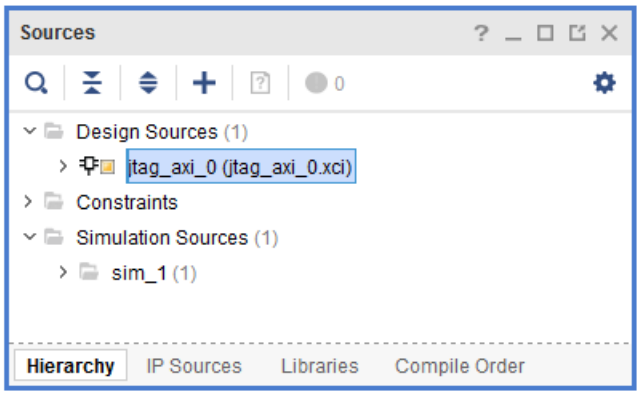

Figure 166: Generated JTAG to AXI Master IP in the Design

14. Right-click **jtag_axi_0** and select **Open IP Example Design**.

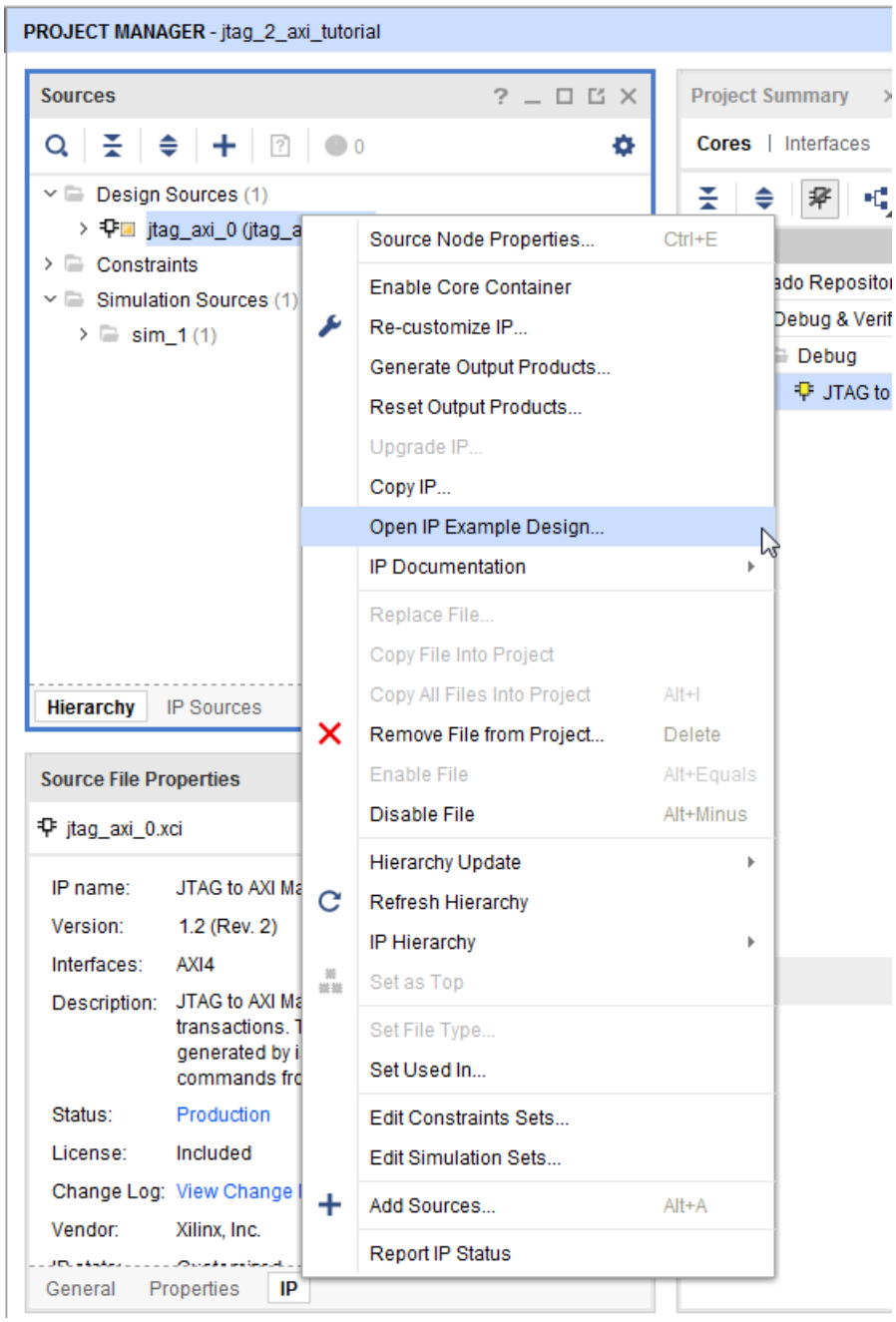

Figure 167: Open IP Example Design Menu Item

15. In the **Open IP Example Design** dialog, ensure that **Overwrite existing example project** is selected. Click **OK**.

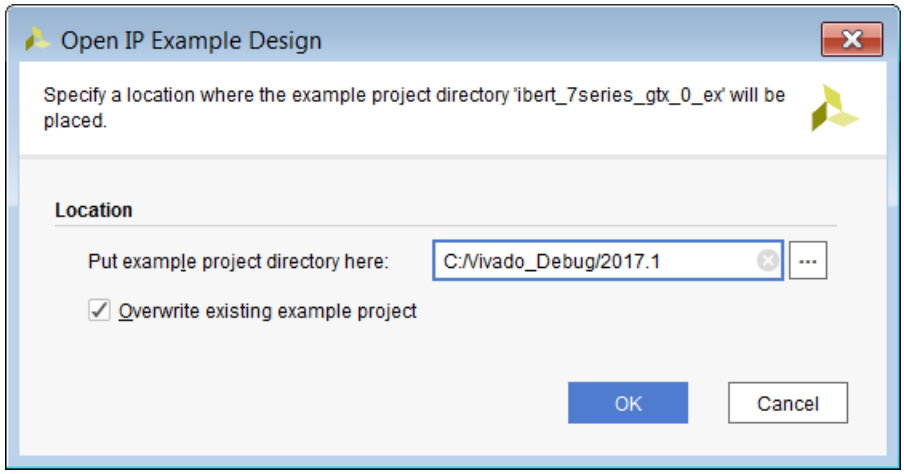

Figure 168: Open IP Example Design Dialog Box

16. Open the `example_jtag_axi_0.v` file and notice that the jtag_axi_0 module is connected to an axi_bram_ctrl_0 (AXI-BRAM block memory) module.
17. In the example_jtag_axi_0.v file, add the following string to the beginning of the wire declaration for each `axi_*` signal from lines 72-108:

```
(* mark_debug *)
```

***Note**: Do not put mark_debug on the `axi_aclk` signal since this might result in Vivado Synthesis adding a LUT1 to the clock path, which could possibly cause you to not meet timing.*

Lines 72-108 should look like this:

```
(* mark_debug *) wire [31:0]axi_araddr;
(* mark_debug *) wire [1:0]axi_arburst;
(* mark_debug *) wire [3:0]axi_arcache;
(* mark_debug *) wire [0 :0]axi_arid;
(* mark_debug *) wire [7:0]axi_arlen;
(* mark_debug *) wire axi_arlock;
(* mark_debug *) wire [2:0]axi_arprot;
(* mark_debug *) wire [3:0]axi_arqos;
(* mark_debug *) wire axi_arready;
(* mark_debug *) wire [2:0]axi_arsize;
(* mark_debug *) wire axi_arvalid;
(* mark_debug *) wire [31:0]axi_awaddr;
(* mark_debug *) wire [1:0]axi_awburst;
(* mark_debug *) wire [3:0]axi_awcache;
(* mark_debug *) wire [0 :0]axi_awid;
(* mark_debug *) wire [7:0]axi_awlen;
(* mark_debug *) wire axi_awlock;
(* mark_debug *) wire [2:0]axi_awprot;
(* mark_debug *) wire [3:0]axi_awqos;
(* mark_debug *) wire axi_awready;
(* mark_debug *) wire [2:0]axi_awsize;
(* mark_debug *) wire axi_awvalid;
(* mark_debug *) wire [0 :0]axi_bid;
(* mark_debug *) wire axi_bready;
(* mark_debug *) wire [1:0]axi_bresp;
(* mark_debug *) wire axi_bvalid;
(* mark_debug *) wire [31 :0]axi_rdata;
(* mark_debug *) wire [0 :0]axi_rid;
(* mark_debug *) wire axi_rlast;
(* mark_debug *) wire axi_rready;
(* mark_debug *) wire [1:0]axi_rresp;
(* mark_debug *) wire axi_rvalid;
(* mark_debug *) wire [31 :0]axi_wdata;
(* mark_debug *) wire axi_wlast;
(* mark_debug *) wire axi_wready;
(* mark_debug *) wire [3 :0]axi_wstrb;
(* mark_debug *) wire axi_wvalid;
```

18. Save changes to `example_jtag_axi_o.v` file.

19. In the **Flow Navigator** on the left side of the Vivado window, click **Run Synthesis**.
20. Open the synthesized design by selecting **Open Synthesized Design** and clicking **OK**.

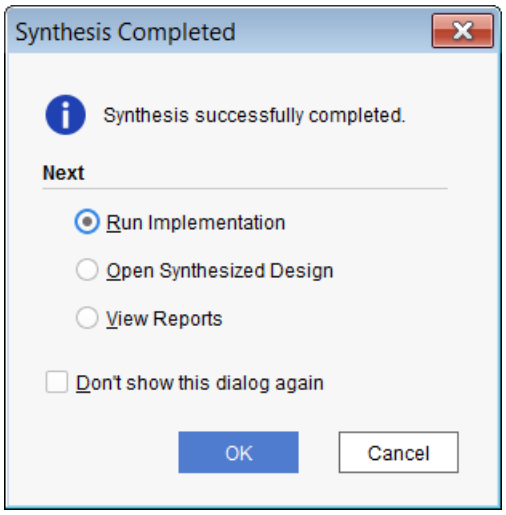

Figure 169: Open Synthesized Design Dialog Box

21. After the synthesized design opens, do the following:
    a. Select the **Debug** layout in the main toolbar Layout drop-down of the Vivado IDE.

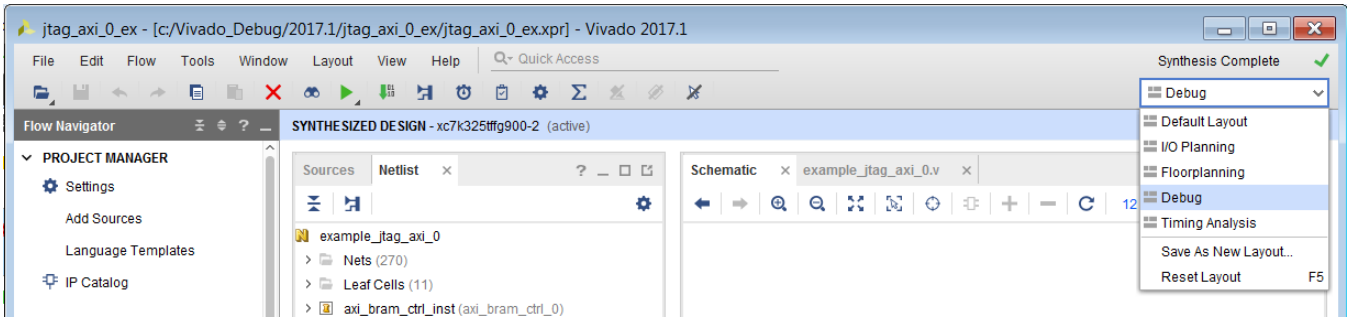

Figure 170: Debug Layout in the Vivado IDE Toolbar

b. Select the **Debug** window near the bottom of the Vivado IDE.

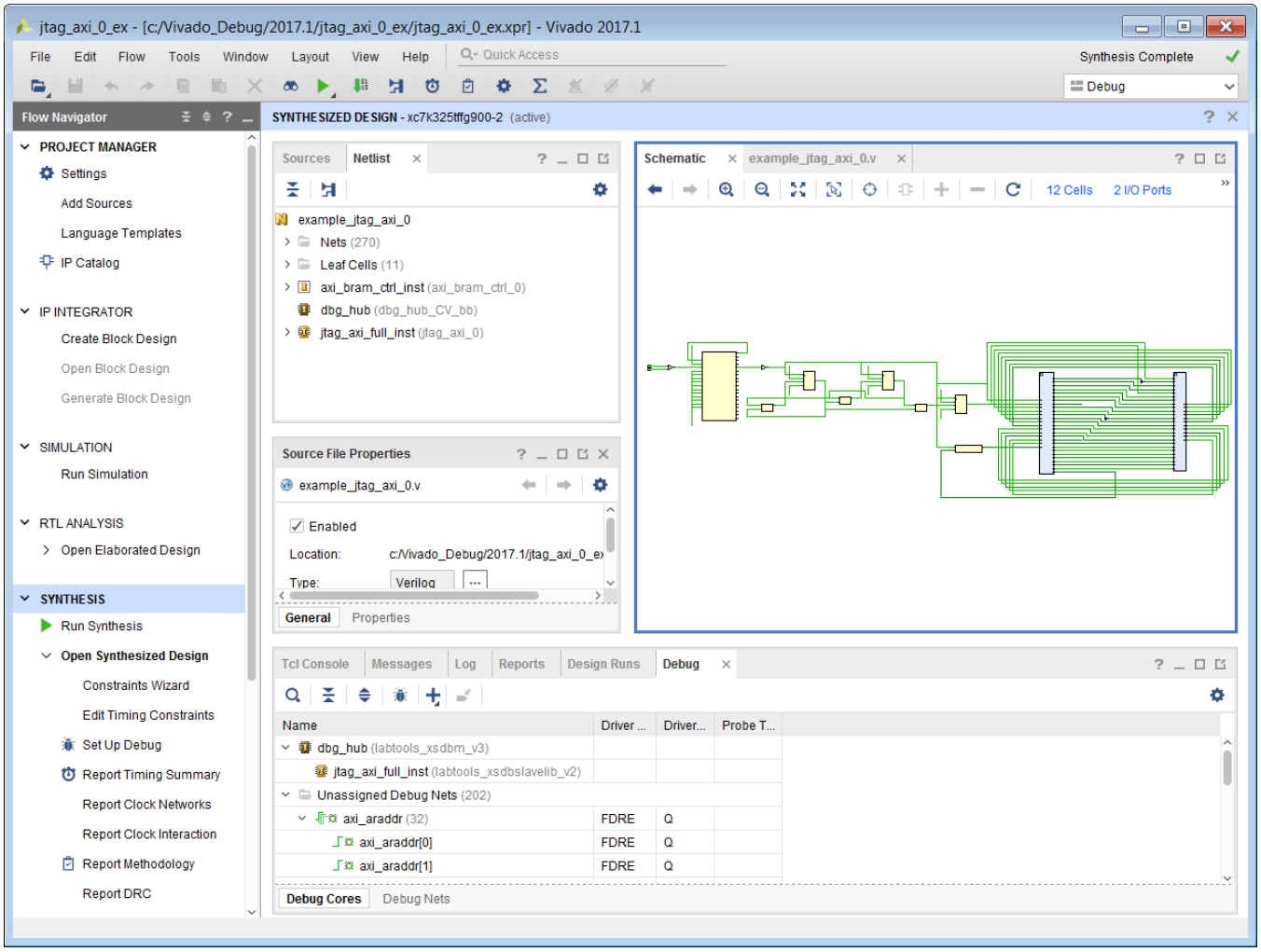

Figure 171: Debug Window in the Vivado IDE

c. Click the **Set Up Debug** toolbar button to launch the **Set up Debug** wizard.

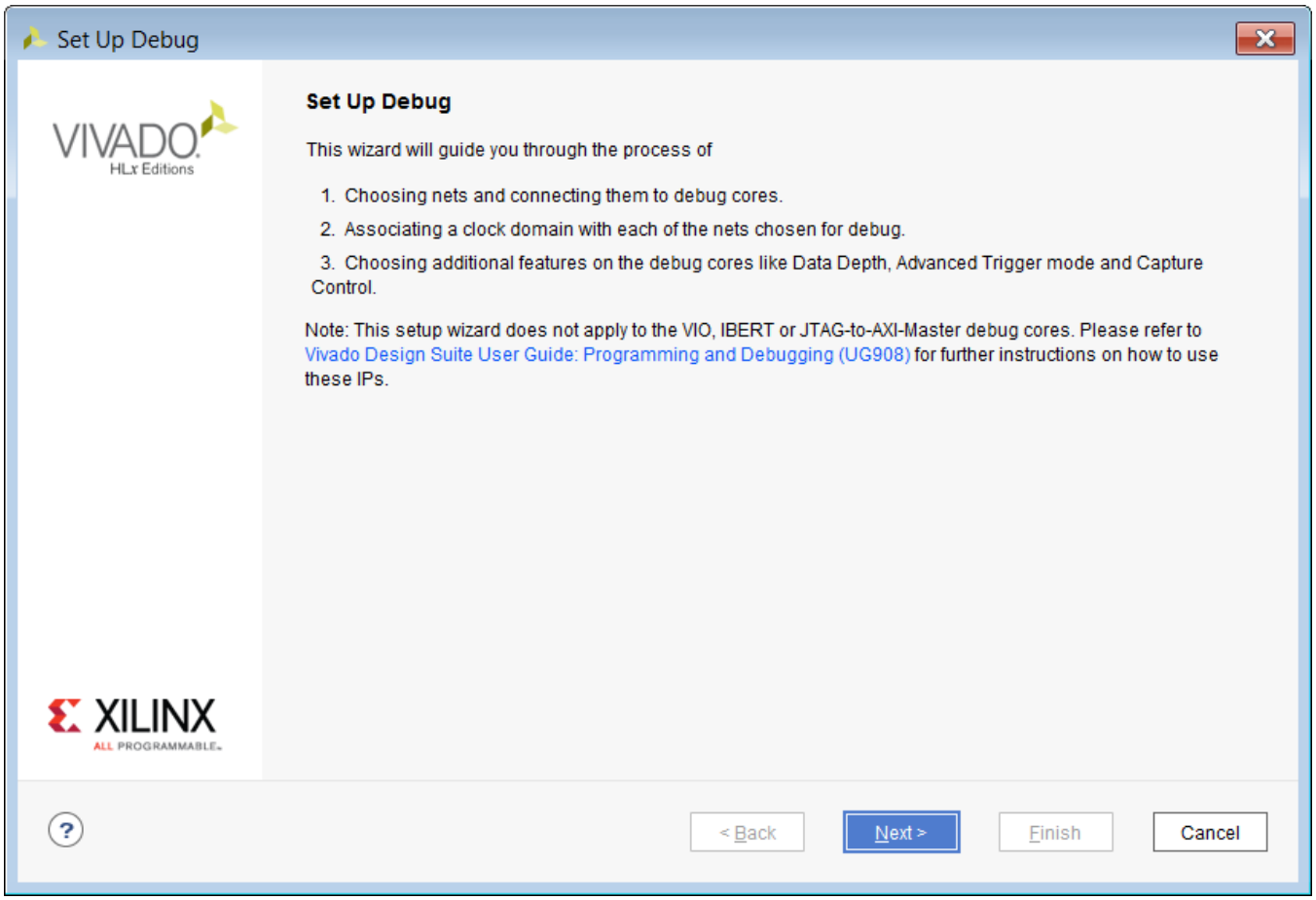

Figure 172: Set Up Debug Wizard

22. The **Set Up Debug** wizard opens, click **Next**.

23. In the next page of the **Setup Debug** wizard, note that some of the nets that you would like to debug have no detectable clock domains selected. Click the **more info** link in the message banner.

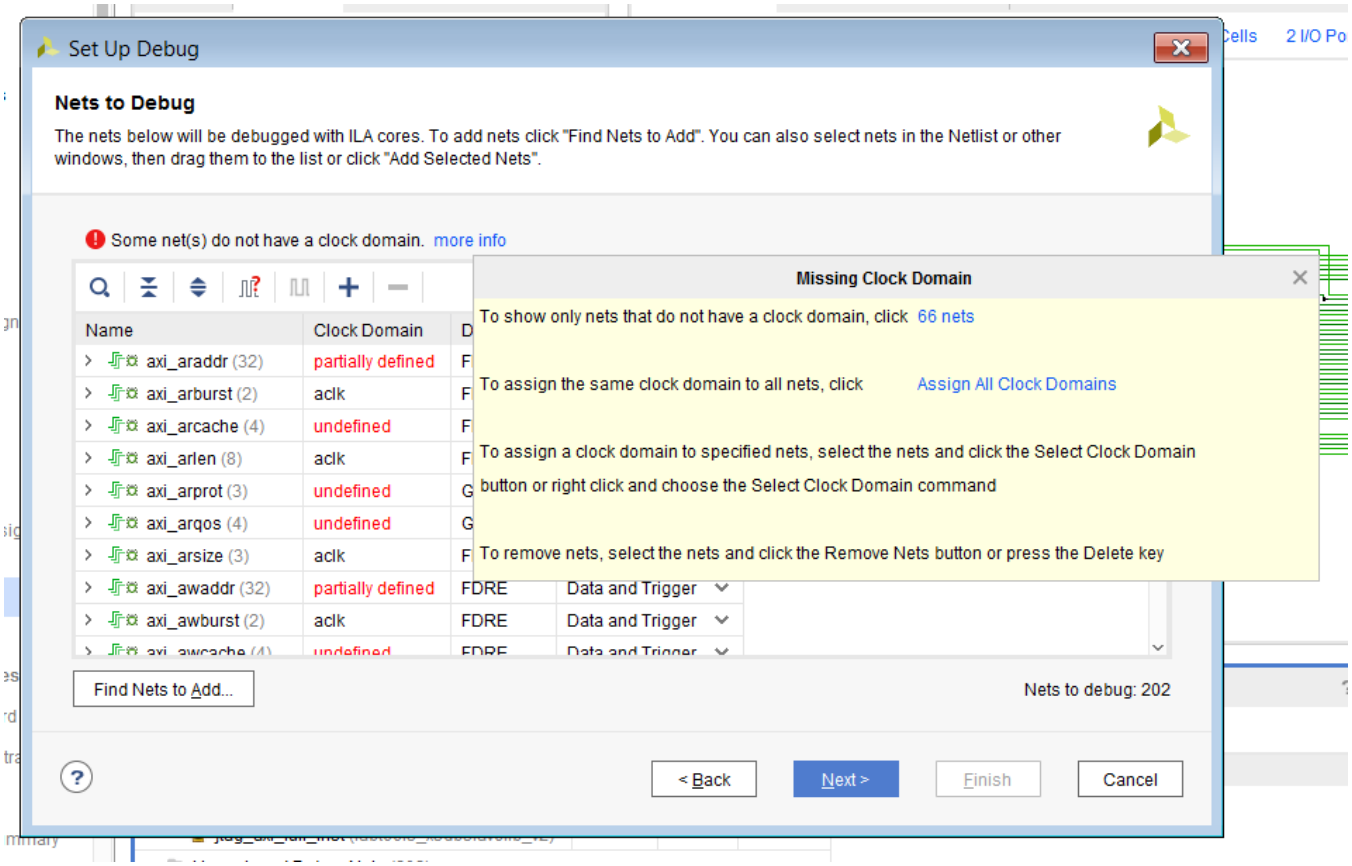

Figure 173: Missing Clock Domain Dialog Box

24. In the resulting pop-up, click **Assign All Clock Domains**.

25. In the **Select Clock Domain** dialog box, select the **aclk** clock net, then click **OK**.

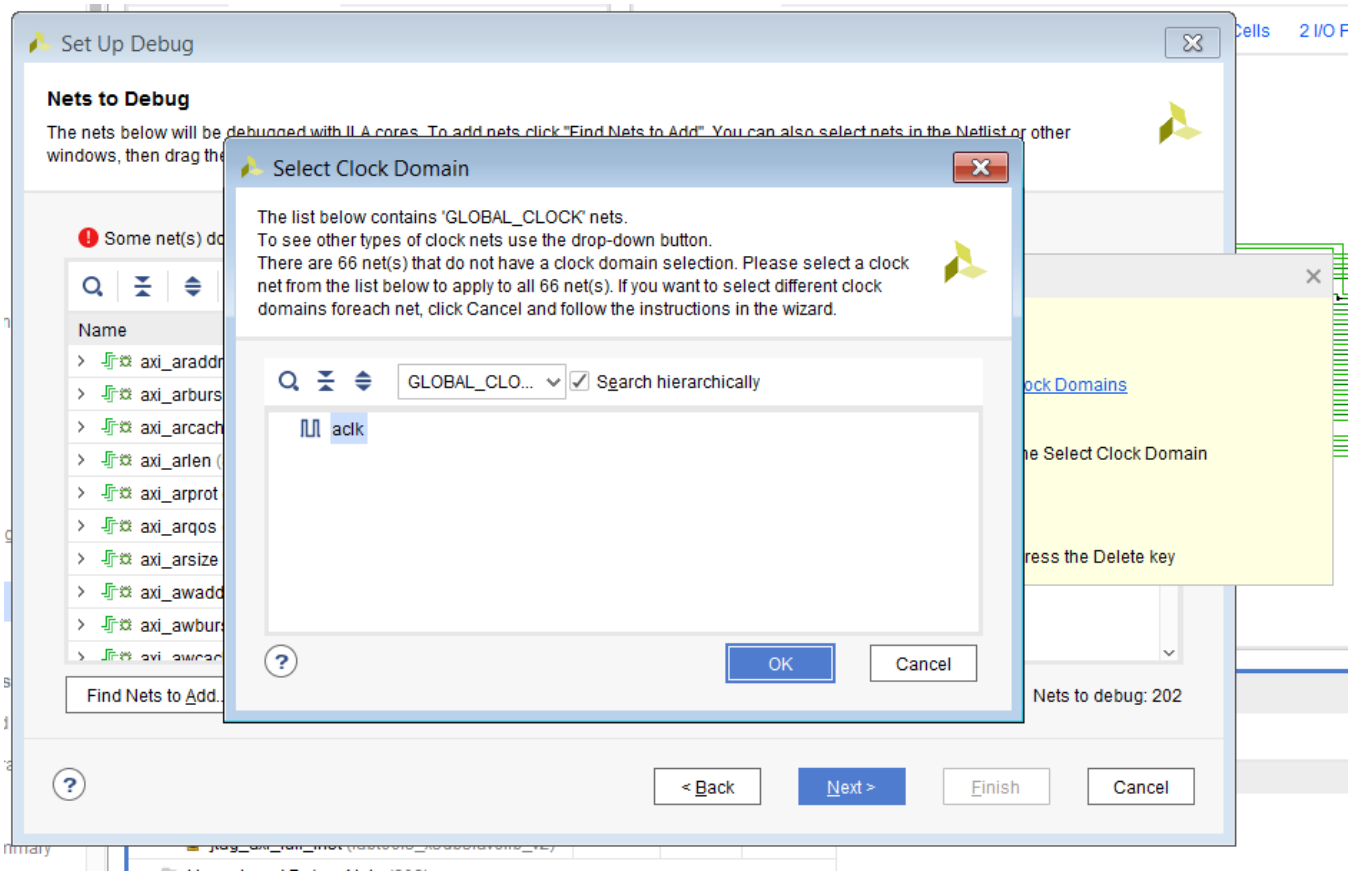

Figure 174: Select Clock Domain Dialog Box

26. Observe that all of the nets now have an assigned clock domain. Click **Next**.

27. In the **Trigger and Storage Settings** area of the **ILA General Options** page, ensure that **Advanced Trigger** and **Capture Control** are selected. Click **Next**.

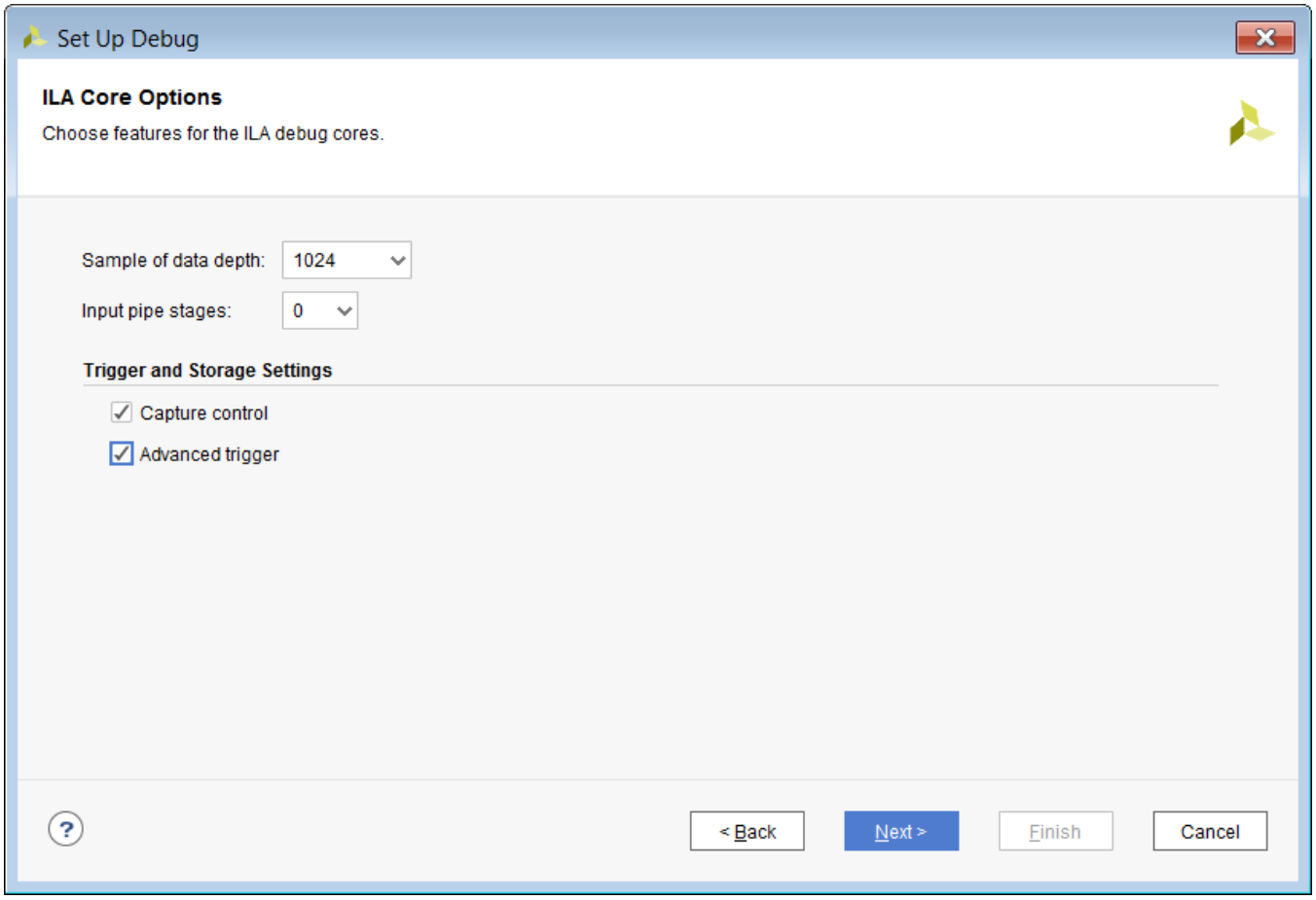

Figure 175: Trigger and Capture Modes Page

28. When **Set up Debug Summary** page appears, ensure that summary is correct and click **Finish**.

    Note: *See that the ILA core was inserted and attached to the* `dbg_hub` *core.*

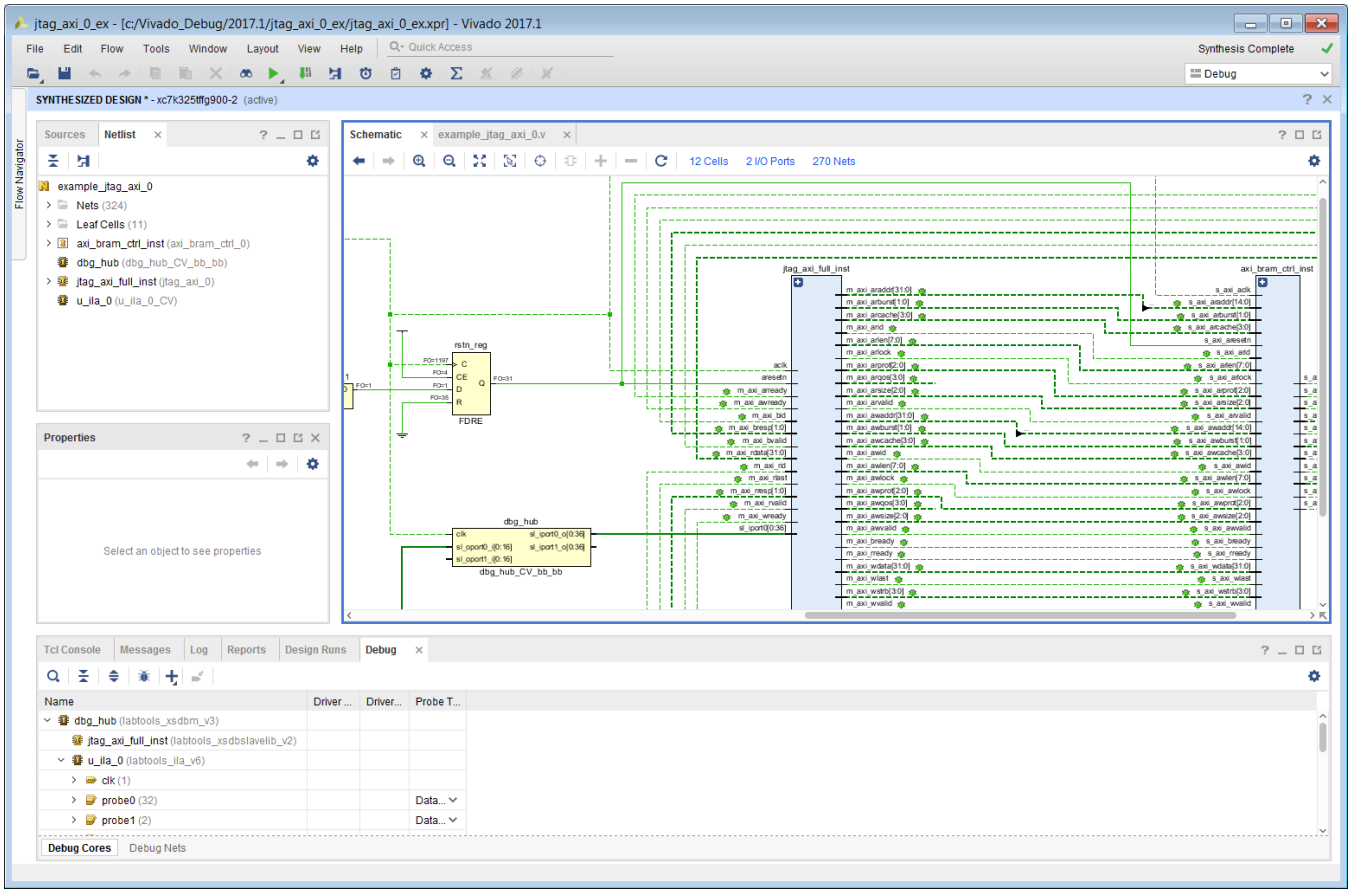

Figure 176: ILA Core Inserted into the Design

29. Save the constraints by clicking **Save**.

30. In the **Flow Navigator** on the left side of the Vivado IDE, click **Generate Bitstream**.
31. Click **Yes** to implement the design.
32. Wait until the Vivado status shows **write_bitstream complete**.
33. In the **Bitstream Generation Completed** dialog box, select **Open Hardware Manager** and click **OK**.

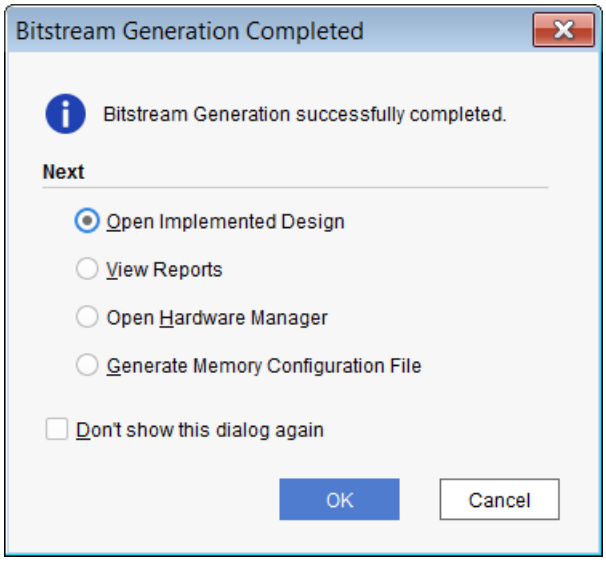

Figure 177: Select Open Hardware Manager

Step 2: Program the KC705 Board and Interact with the JTAG to AXI Master Core

1. Connect your KC705 board's USB-JTAG interface to a machine that has Vivado IDE and cable drivers installed on it and power up the board.
2. The **Hardware Manager** window opens. Click **Open New Target**. The **Open New Hardware Target** dialog box opens.

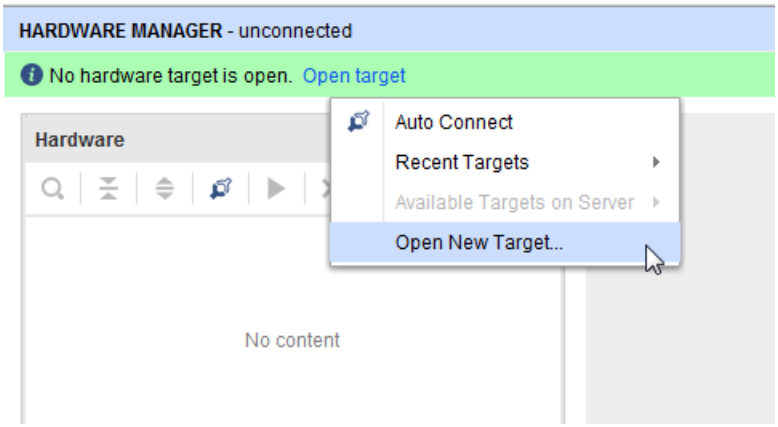

Figure 178: Connect to a Hardware Target

3. In the **Connect to** field choose **Local server,** and click **Next**.

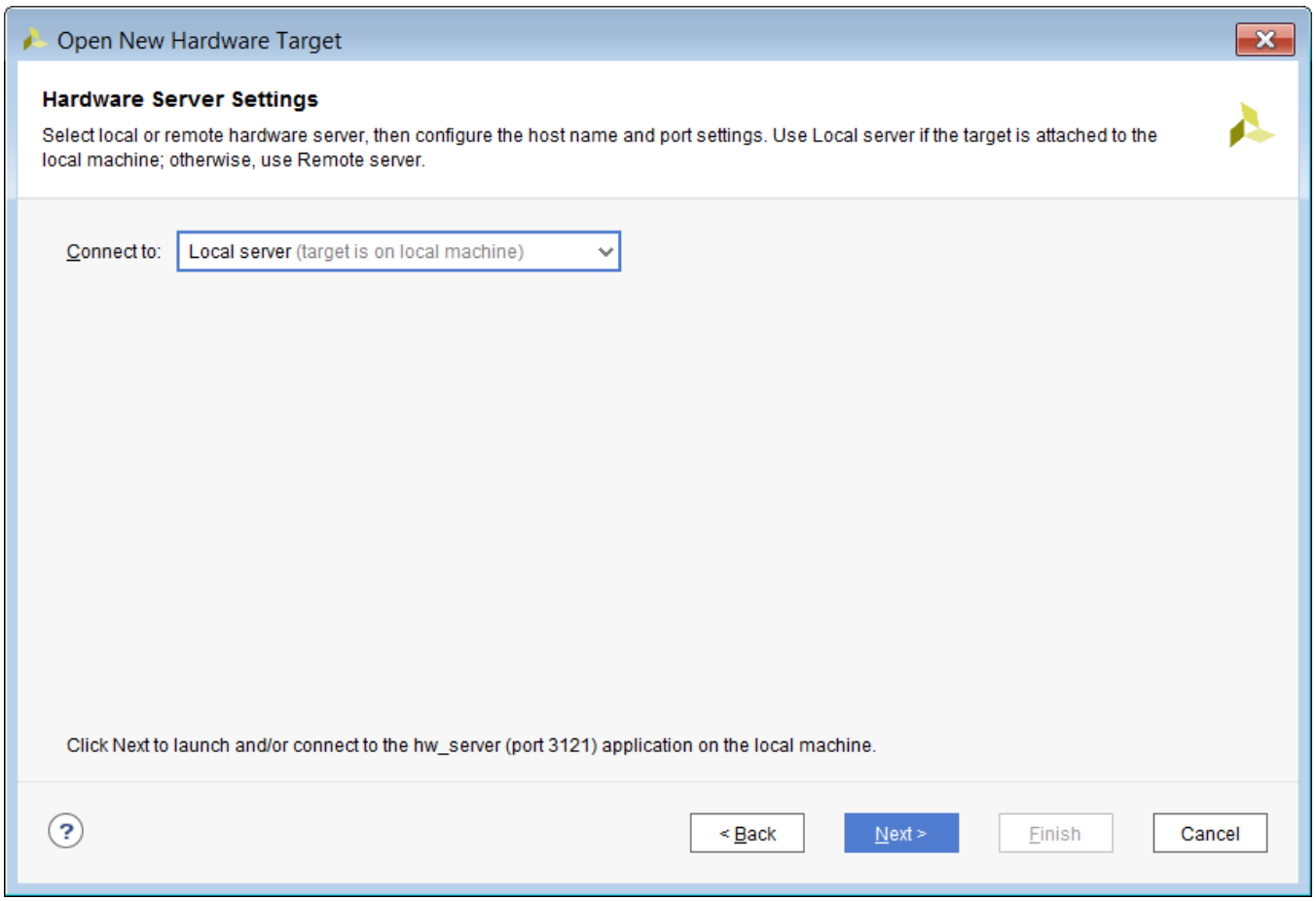

Figure 179: Hardware Server Name

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

4. If there is more than one target connected to the hardware server, you will see multiple entries in the **Select Hardware Target** page. In this tutorial, there is only one target as shown in the following figure. Leave these settings at their default values and click **Next**.

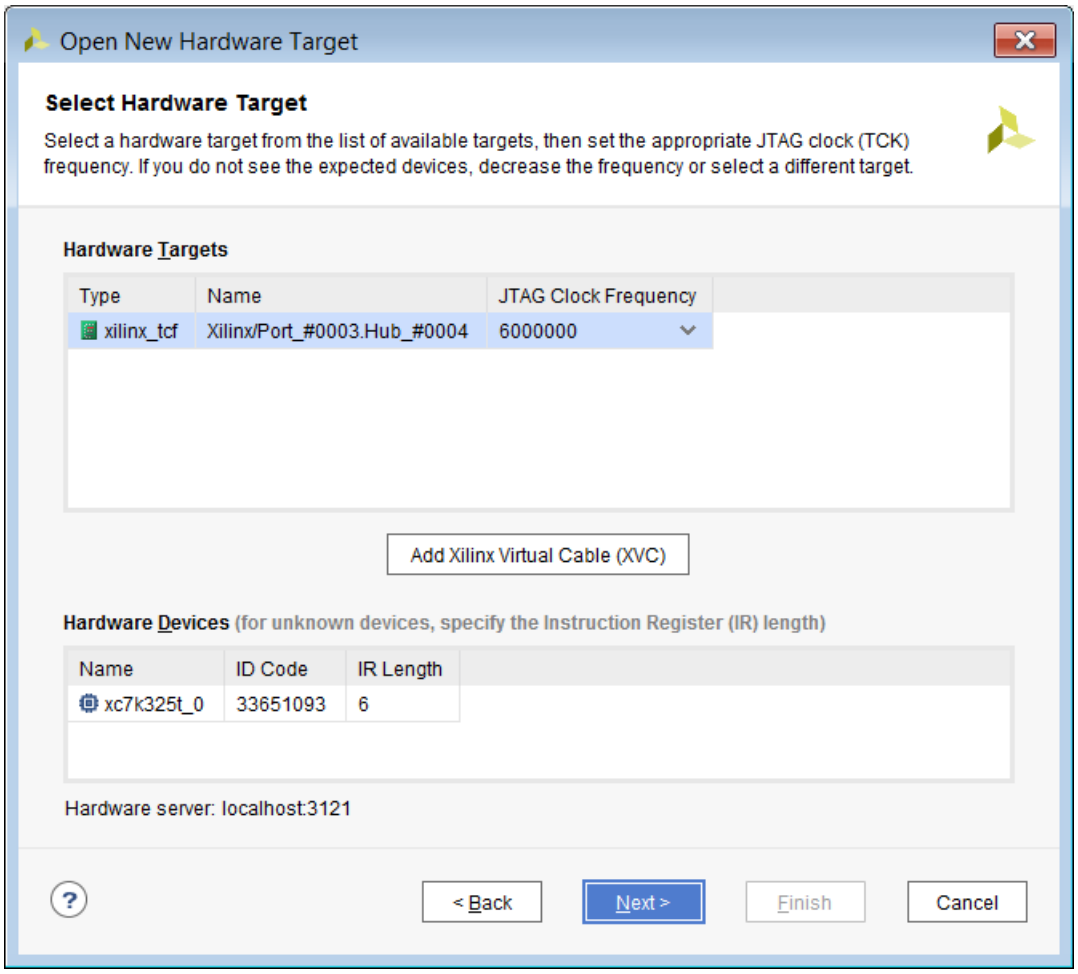

Figure 180: Select Hardware Target

5. Leave these settings at their default values as shown. Click **Next**.

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

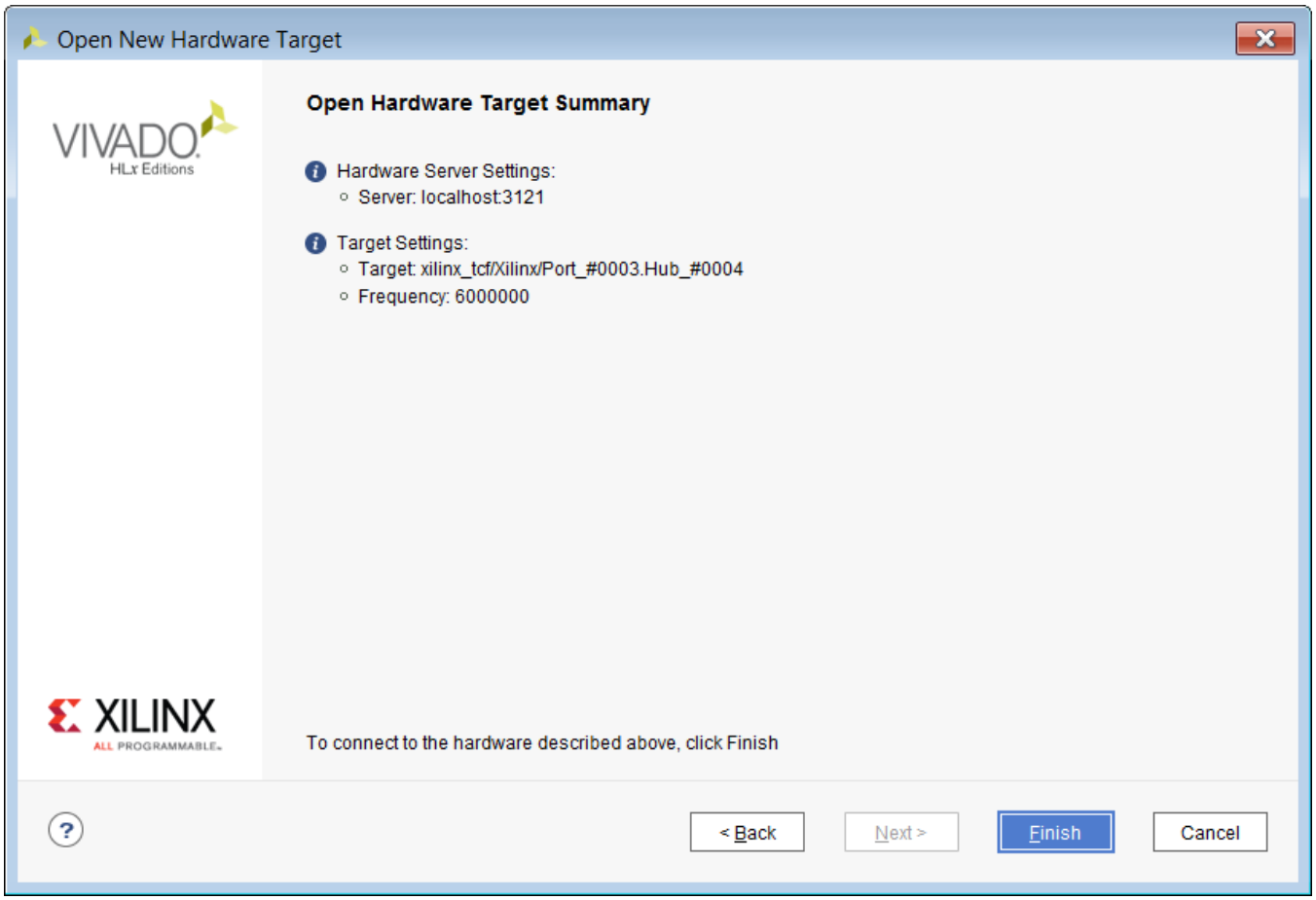

Figure 181: Open Hardware Summary

Wait for the connection to the hardware to complete. After the connection to the hardware target is made, the dialog shown in the following figure opens.

> ***Note**: The **Hardware** tab in the **Debug** view shows the hardware target and XC7K325T device that was detected in the JTAG chain.*

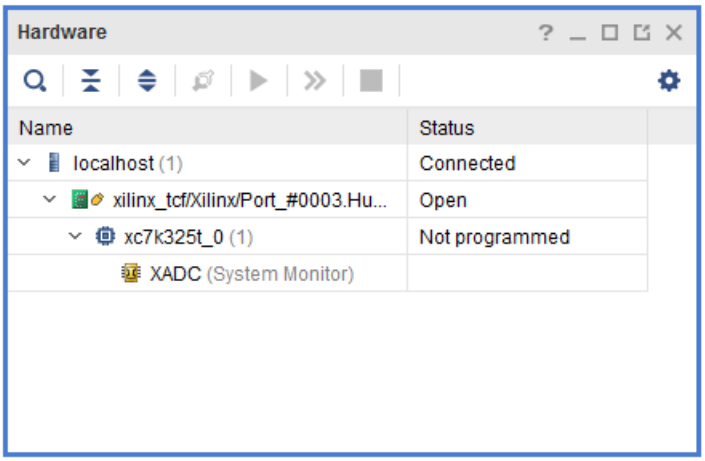

Figure 182: Hardware Target and XC7K325T Device

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

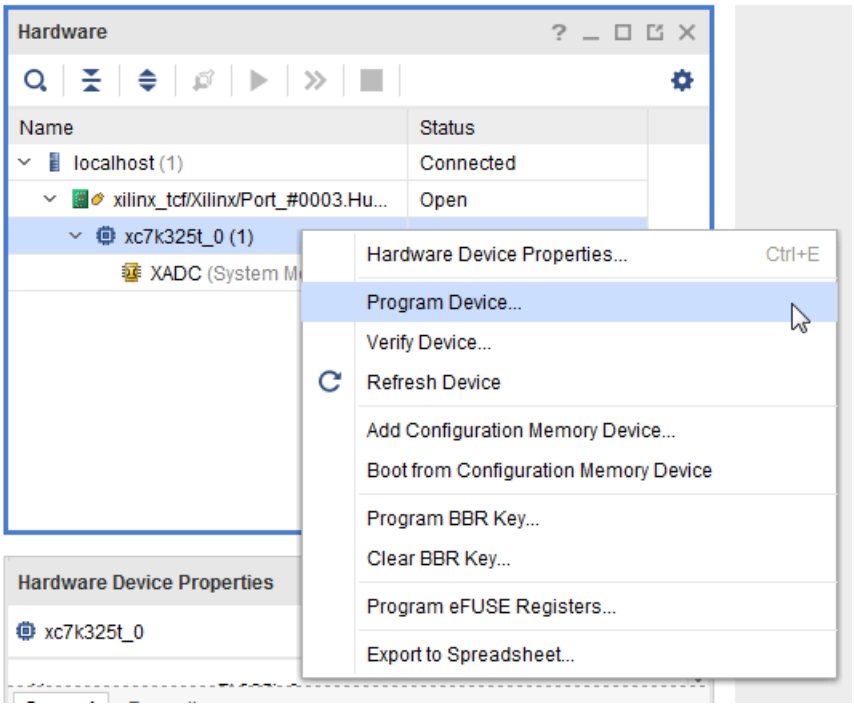

Figure 183: Program Active Target Hardware

8. In the **Program Device** dialog box verify that the `.bit` file is correct for the lab that you are working on. Click **Program** to program the device.

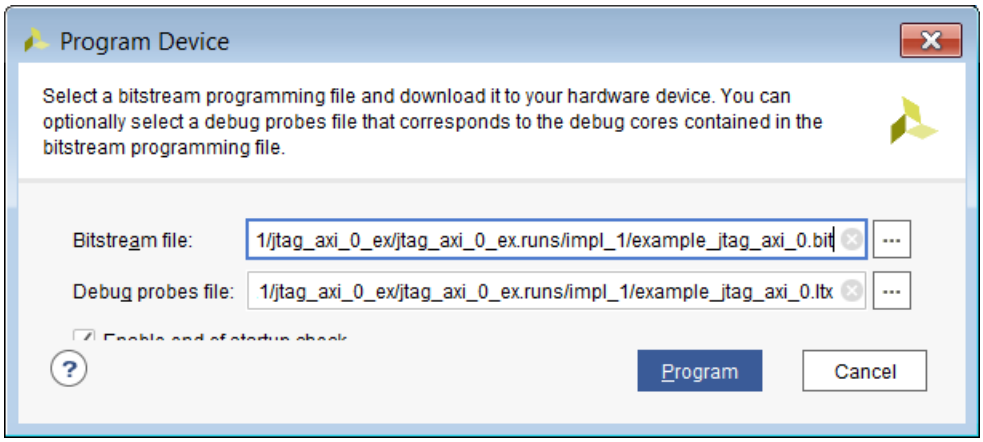

Figure 184: Select Bitstream File to Download

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

9. Verify that the JTAG to AXI Master and ILA cores are detected by locating the **hw_axi_1** and **hw_ila_1** instances in the **Hardware Manager** window.

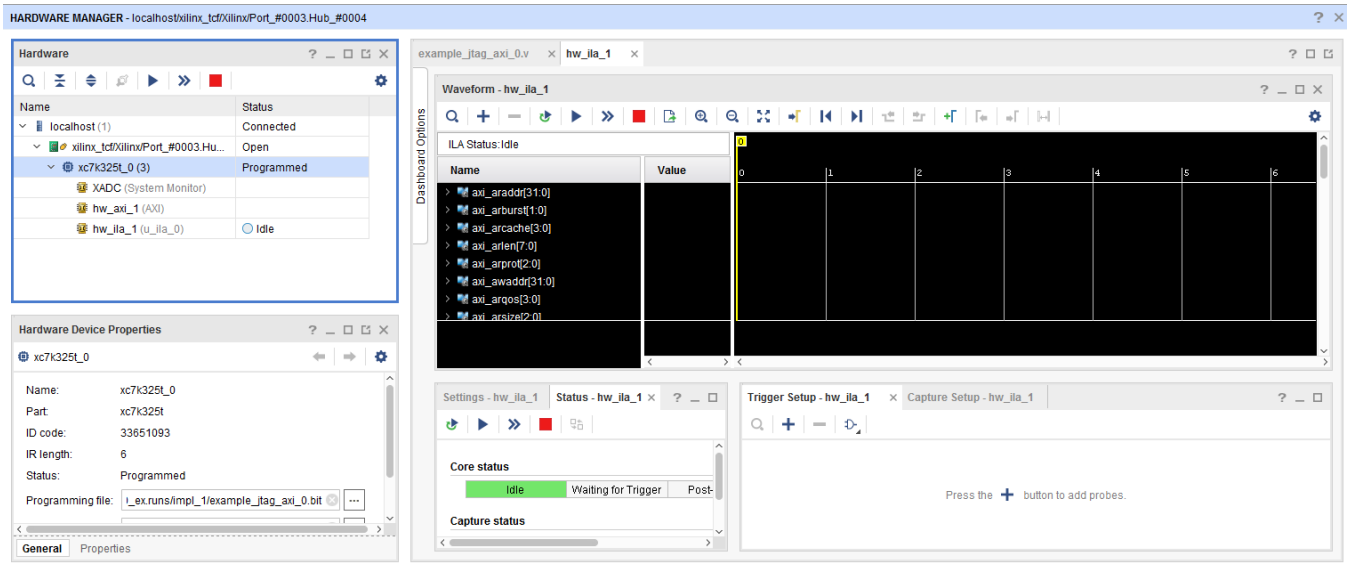

Figure 185: ILA Core Instances in the Hardware Window

10. You can communicate with the JTAG to AXI Master core with Tcl commands only. You can issue AXI read and write transactions using the `run_hw_axi` command. However, before issuing these transactions, it is important to reset the JTAG to AXI Master core. Because the `aresetn` input port of the `jtag_axi_0` core instance is not connected to anything, you need to use the following Tcl commands to reset the core:

    ```
    reset_hw_axi [get_hw_axis hw_axi_1]
    ```

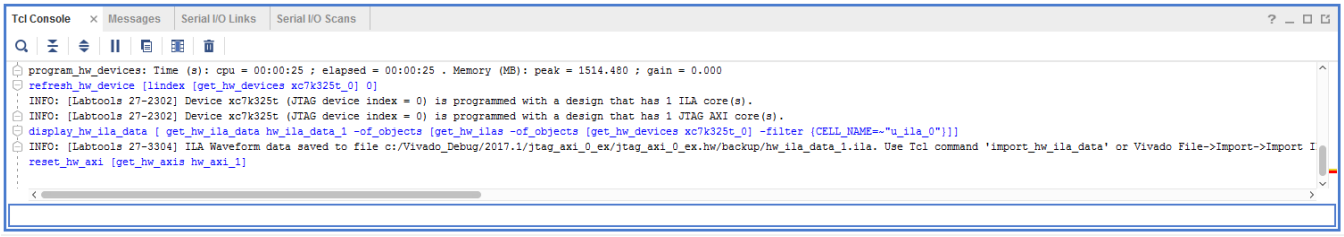

Figure 186: Reset JTAG to AXI core

11. The next step is to create a 4-word AXI burst transaction to write to the first four locations of the BRAM:

    ```
    set wt [create_hw_axi_txn write_txn [get_hw_axis hw_axi_1] -type WRITE -address
    00000000 -len 128 -data {44444444_33333333_22222222_11111111}]
    ```

    where:

    - write_txn is the name of the transaction
    - [get_hw_axis hw_axi_1] returns the hw_axi_1 object
    - -address 00000000 is the start address
    - -len 128 sets the AXI burst length to 128 words
    - -data {44444444_33333333_22222222_11111111} is the data to be written.

    ***Note**: The data direction is MSB to the left (i.e., address 3) and LSB to the right (i.e., address 0). Also note that the data will be repeated from the LSB to the MSB to fill up the entire burst.*

12. The next step is to set up a 128-word AXI burst transaction to read the contents of the first four locations of the AXI-BRAM core:

```
set rt [create_hw_axi_txn read_txn [get_hw_axis hw_axi_1] -type READ -address 00000000 -len 128]
```

    where:

    - read_txn is the name of the transaction
    - [get_hw_axis hw_axi_1] returns the hw_axi_1 object
    - -address 00000000 is the start address
    - -len 128 sets the AXI burst length to 4 words

13. After creating the transaction, you can run it as a write transaction using the run_hw_axi command:

```
run_hw_axi $wt
```

This command should return the following:

```
INFO: [Labtools 27-147] : WRITE DATA is : 44444444333333332222222211111111…
```

14. After creating the transaction, you can run it as a read transaction using the run_hw_axi command:

```
run_hw_axi $rt
```

This command should return the following:

```
INFO: [Labtools 27-147] : READ DATA is : 44444444333333332222222211111111…
```

Step 3: Using ILA Advanced Trigger Feature to Trigger on an AXI Read Transaction

1. In the **ILA – hw_ila_1** dashboard, locate the **Trigger Mode Settings** area and set **Trigger mode** to **ADVANCED_ONLY**.
2. In the **Capture Mode Settings** area set the **Trigger position** to **512**.
3. In the **Trigger State Machine** area click the **Create new trigger state machine** link.

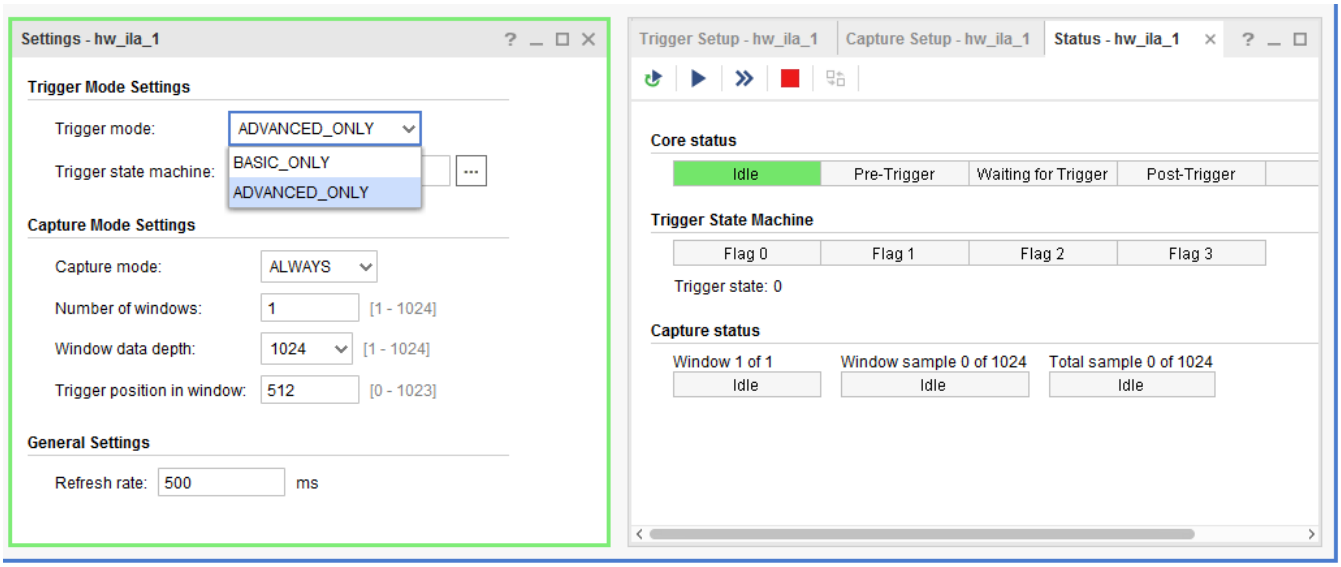

Figure 187: Setting Trigger Mode to ADVANCED and Trigger Position to 512 in the ILA Dashboard

4. In the **New Trigger State Machine File** dialog box set the name of the state machine script to **txns.tsm**.

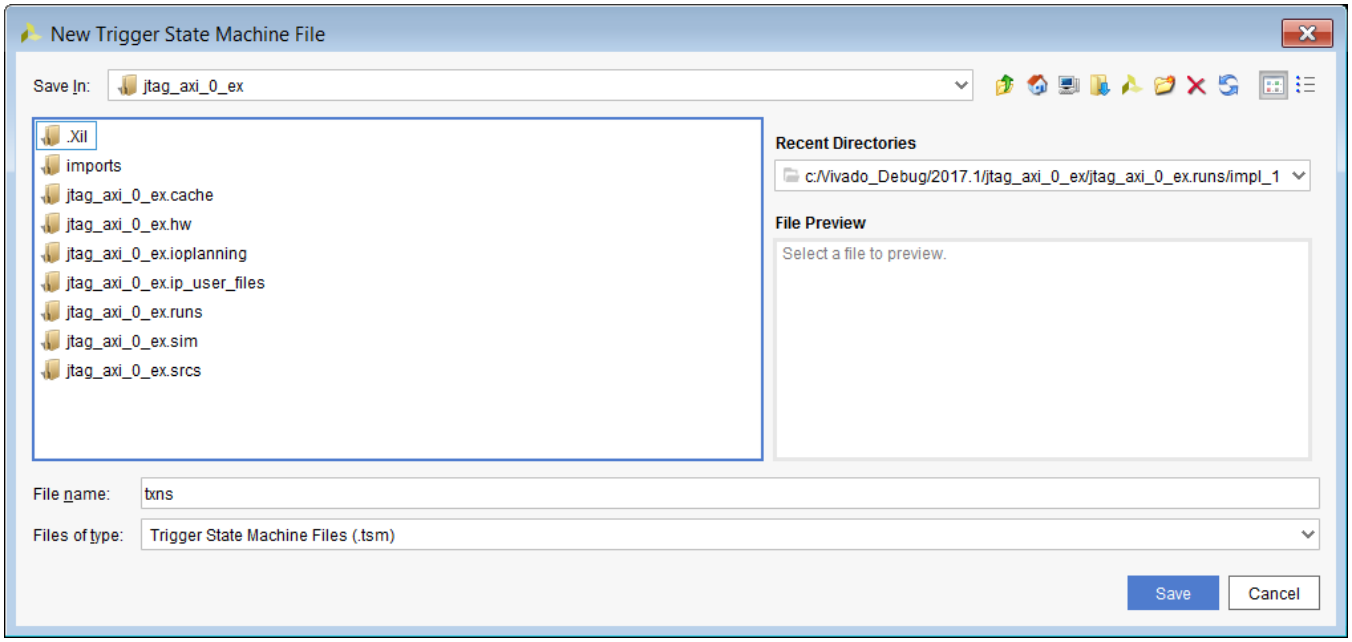

Figure 188: Creating a New Trigger State Machine Script

5. A basic template of the trigger state machine script is displayed in the Trigger State Machine gadget. Expand the trigger state machine gadget in the ILA dashboard. Copy the script below after line 17 of the state machine script and save the file.

```
# The "wait_for_arvalid" state is used to detect the start
# of the read address phase of the AXI transaction which
# is indicated by the axi_arvalid signal equal to '1'
#
state wait_for_arvalid:
    if (axi_arvalid == 1'b1) then
      goto wait_for_rready;
    else
      goto wait_for_arvalid;
    endif
#
# The "wait_for_rready" state is used to detect the start
# of the read data phase of the AXI transaction which
# is indicated by the axi_rready signal equal to '1'
#
state wait_for_rready:
  if (axi_rready == 1'b1) then
    goto wait_for_rlast;
  else
    goto wait_for_rready;
  endif

#
# The "wait_for_rlast" state is used to detect the end
```

```
# of the read data phase of the AXI transaction which
# is indicated by the axi_rlast signal equal to '1'.
# Once the end of the data phase is detected, the ILA core
# will trigger.
#
state wait_for_rlast:
  if (axi_rlast == 1'b1) then
    trigger;
  else
    goto wait_for_rlast;
  endif
```

***Note**: The state machine is used to detect the various phases of an AXI read transaction:*

- Beginning of the read address phase.
- Beginning of the read data phase.
- End of the read data phase.

6. Arm the trigger of the ILA by right-clicking the **hw_ila_1** core in the **Hardware Manager** window and selecting **Run Trigger**.

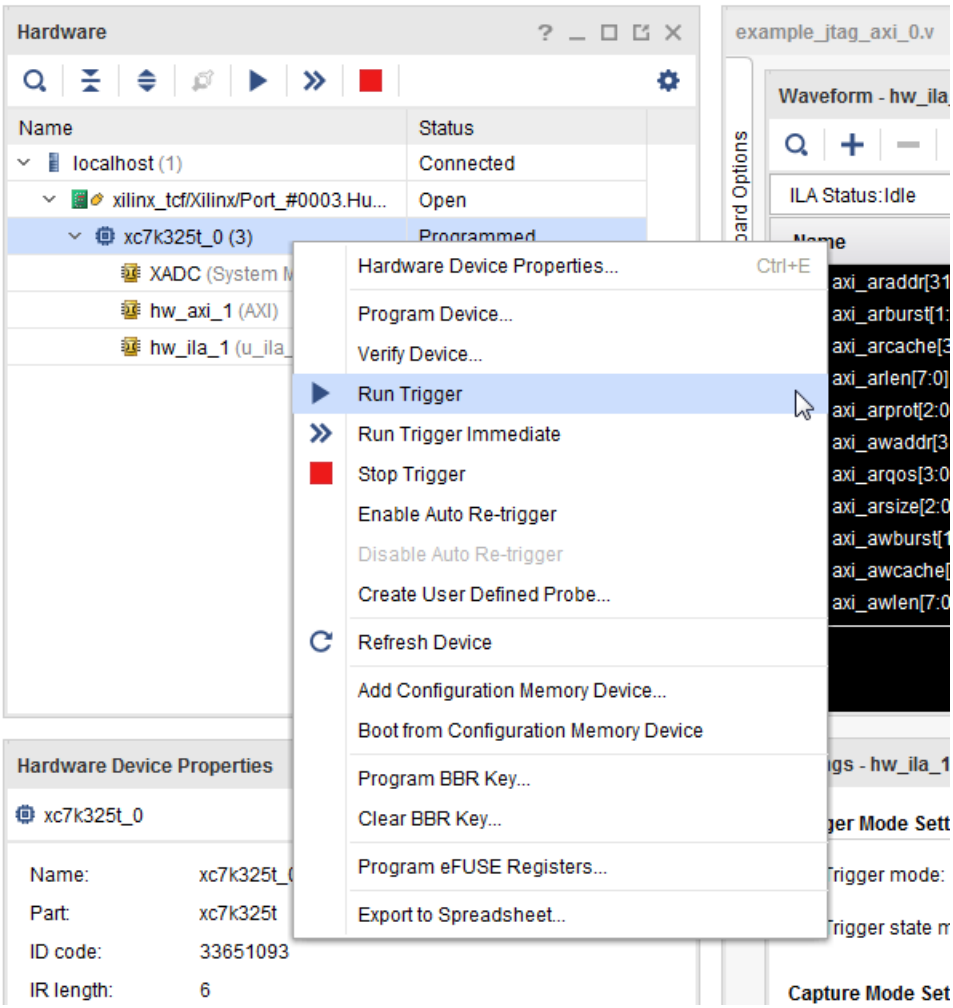

Figure 189: Run Trigger

7. In the **Trigger Capture Status** window, note that the ILA core is waiting for the trigger to occur, and that the trigger state machine is in the **wait_for_a_valid** state. Note that the pre-trigger capture of 512 samples has completed successfully:

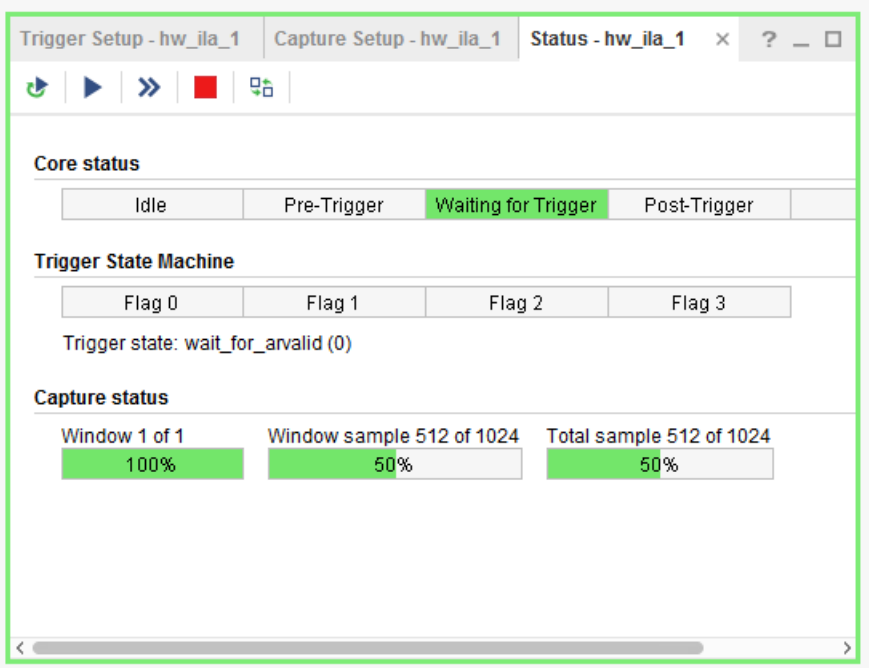

Figure 190: Trigger Capture Status Window

8. In the Tcl console, run the read transaction that you set up in the previous section of this tutorial.

```
run_hw_axi $rt
```

***Note**: The ILA core has triggered and the trigger mark is on the sample where the `axi_rlast` signal is equal to '1', just as the trigger state machine program intended.*

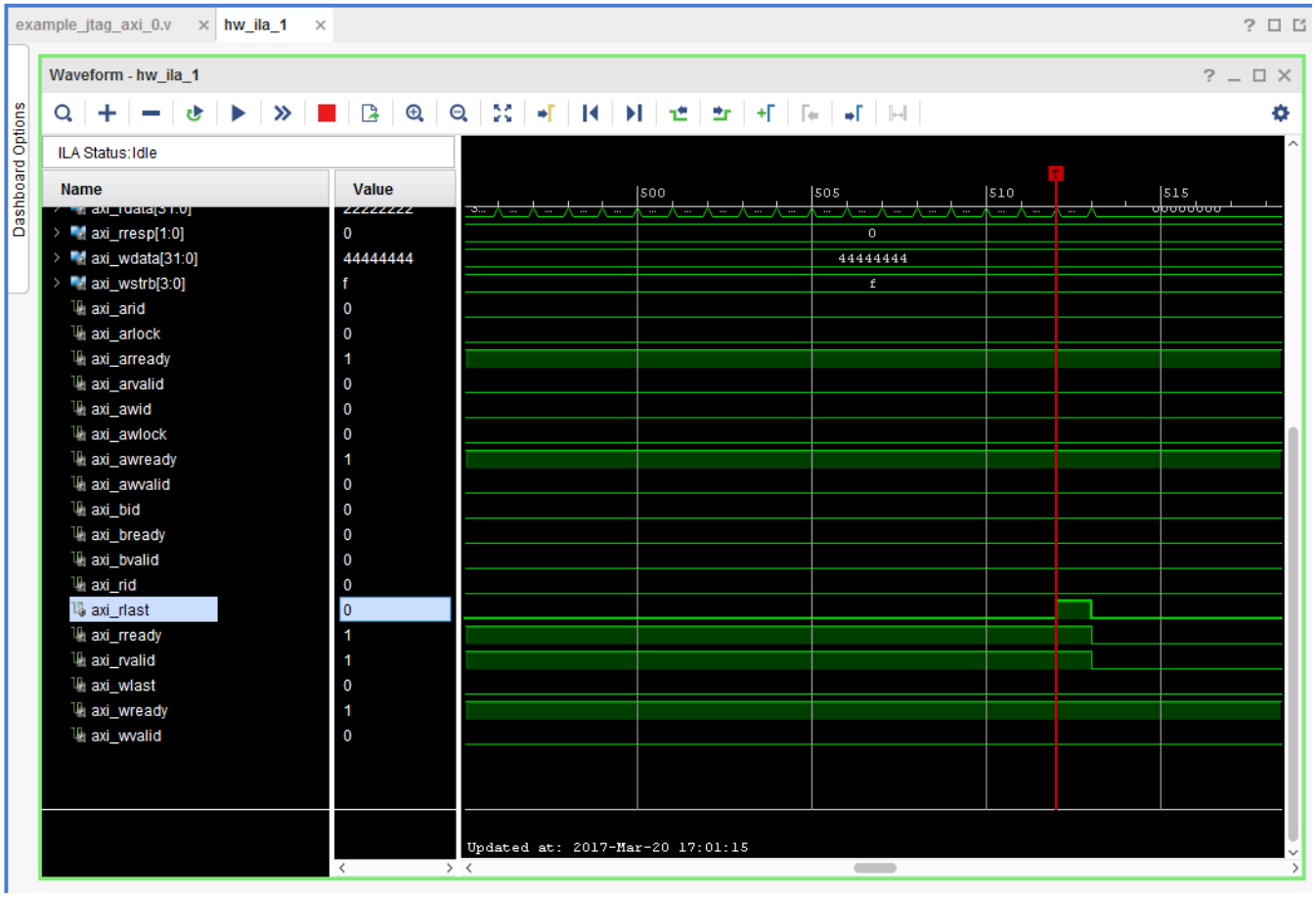

Figure 191: Waveform window

Lab 10: Using Vivado Serial Analyzer to Debug GTR Serial Links

Introduction

IBERT UltraScale+™ GTR (IBERT GTR) can be used to evaluate and monitor GTR transceivers in Zynq® UltraScale+ MPSoC devices. With this feature, a user can accomplish the following tasks:

- Perform eye scans with user data
- Change GTR settings
- View link status
- Check the "lock" status of all PLLs used by all GTR lanes

IBERT GTR does NOT provide the following capabilities:

- Perform eye scans with raw PRBS data patterns
- Measure Bit Error Ratio (no bit or error counters)

Note that this solution is purely software based, meaning that no IP or logic is required in the programmable logic of the device. This documentation will guide you through the setup of the GTR Transceivers by creating a First Stage Boot Loader (FSBL). It will then demonstrate how to load the FSBL into the Zynq UltraScale+ and use IBERT GTR.

IMPORTANT: *This is a supported feature in Vivado® 2017.2 and beyond.*

IBERT GTR Flow

The IBERT GTR Bring-up and subsequent EyeScan involves three different components:

1. Generating Zynq UltraScale+ MPSoC PS Hardware Definition File (HDF) from Vivado after configuring the GTR
2. Using SDK XSCT flow to generate First Stage Boot Loader file by using the Hardware Definition File (HDF)
3. Using First Stage Boot Loader file with Vivado Serial I/O Analyzer to bring up IBERT GTR.

Tools Required:

- Vivado

- SDK
- XSCT (Part of SDK)

Board/Part/Components required:

- ZCU102 Rev 1.0 board
- XCZU9EG-FFVB1156 production device
- PCIe®:
  - A PCIe card which has at least x4 lanes
  - PCI Express® 4x Male to PCI-E 16x Female Riser Cable if PCIe card is larger than x4
- SATA:
  - SanDisk 128 GB SATA SSD Drive
  - SATA connector cable
  - 4 Pin Molex to SATA Power Cable Adapter
- USB:
  - SanDisk Ultra 32 GB USB 3.0 Flash Drive
  - USB 3.0 Type A Female to Micro Male Adapter

Required Files

- First Stage Boot Loader ELF File (Created using instructions below) which configures the GTR
- Configuration Bitstream File (Optional file that may be needed to custom configure the FPGA depending on the board setup)
- Tcl Script to generate the FSBL and modify C-source for USB Support (when available)

Assumptions

1. FSBL should always target A53 processor as R5 (psu_cortexr5_0) is exclusively used by IBERT GTR.
2. Physical devices such as SATA drive, PCIE card, etc. are needed for validation.

Step 1: Generating Zynq UltraScale+ MPSoC PS Hardware Definition File (HDF)

1. Open Vivado.
2. Click on **Create Project** and click **Next**.

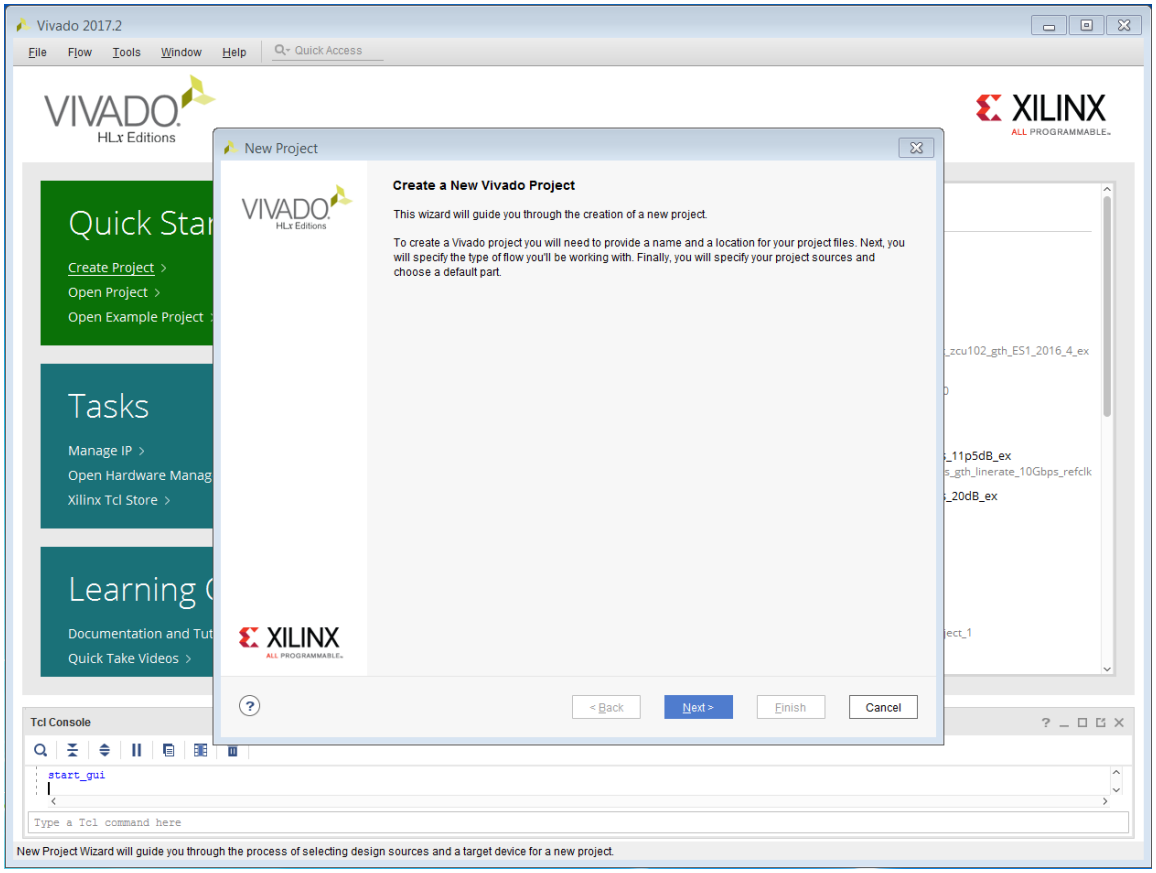

Figure 192: New Project Window

3. Set your project name and specify the project directory. Click **Next**.
4. Select Project type as **RTL project**
5. To generate default HDF, keep **Do not specify sources at this time** checked, then click **Next**.
6. To choose the board, click on the board icon and select **Zynq UltraScale+ ZCU102 Evaluation board**, with **Board Rev 1.0**. Click **Next**.

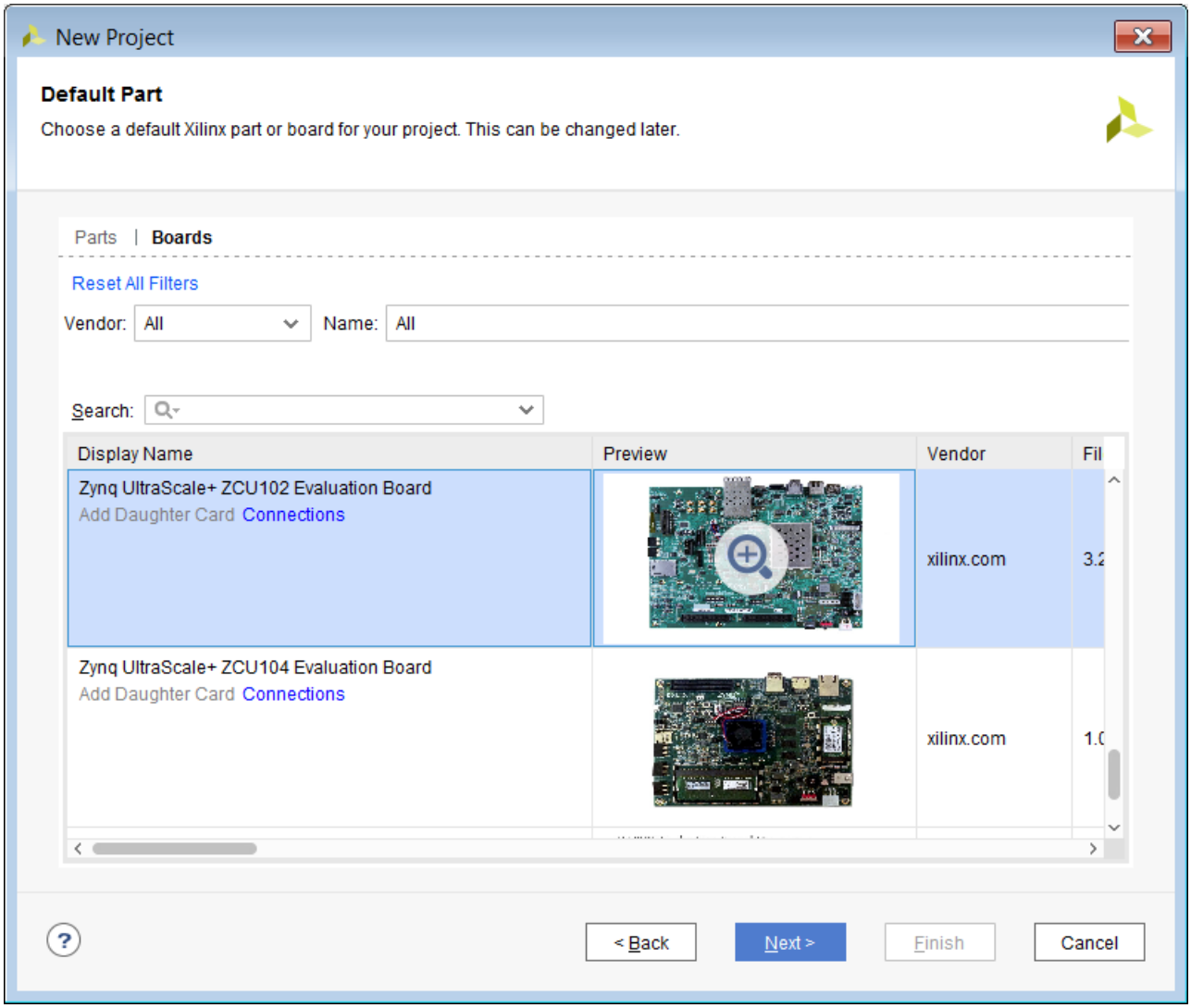

Figure 193: Board Selection Window

7. The project summary displays. To create the project, click **Finish**.

8. Select **Create Block Design** in the Flow Navigator. You can specify the design name and directory, but it is not necessary for a local project directory. Click **OK** to create the block design.

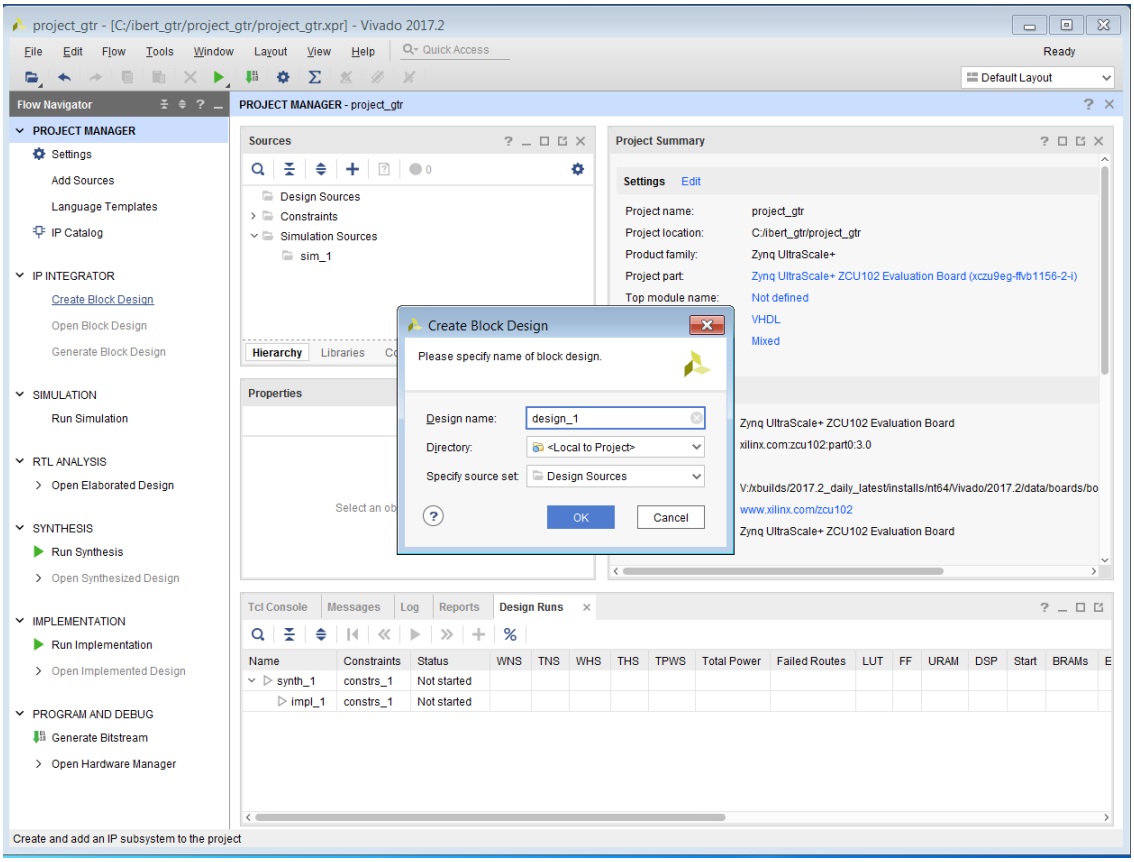

Figure 194: Create Block Design

9. An empty design diagram displays. Click on **Add IP** button to add IP. Select IP based on selected board (for ZCU102 evaluation board, search for 'Zynq UltraScale+ MPSoC') and double click on the selected IP.

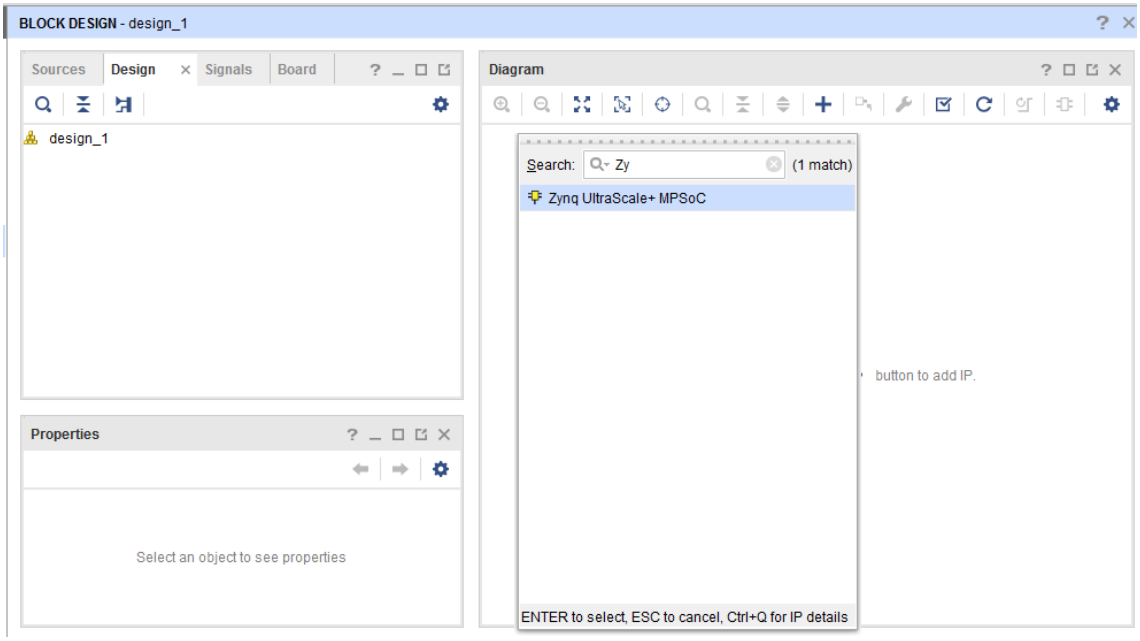

Figure 195: Add IP Window

10. Select **Run Block Automation** in the design diagram window. Click **OK** to continue creating the ZCU102 design.

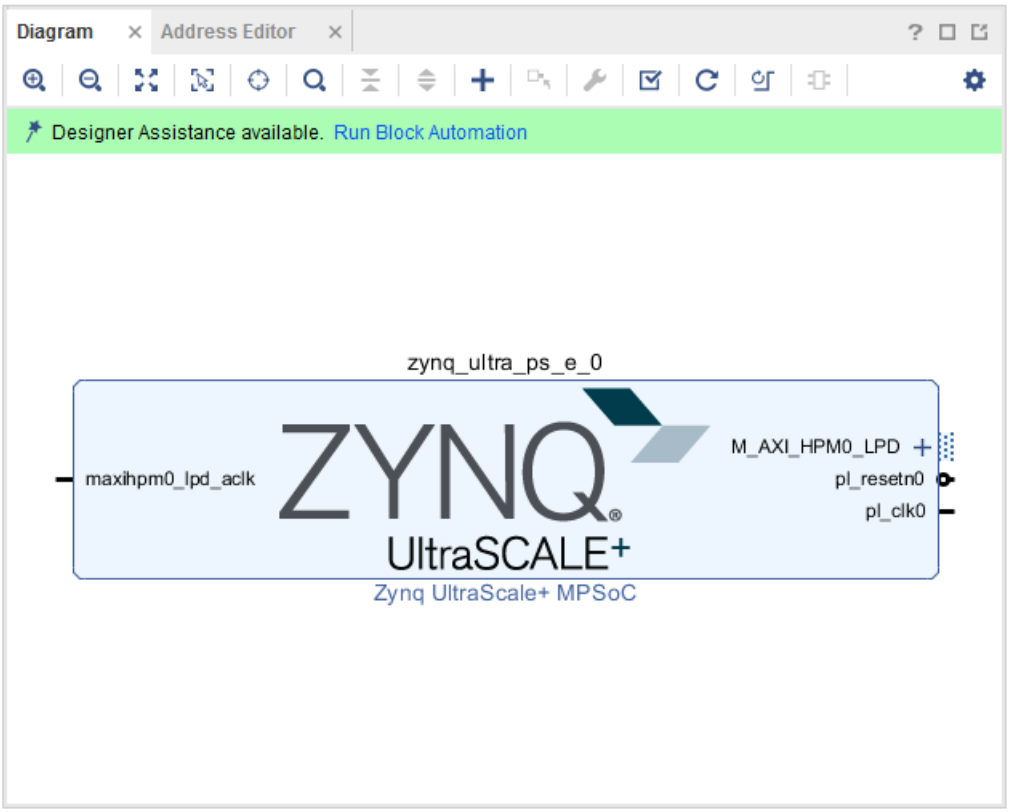

Figure 196: Run Block Automation

11. When the design diagram appears, follow the below steps to validate design:

    a. Connect `maxihpm0_fpd_aclk` and `maxihpm1_fpd_aclk` together to `pl_clk0`, as shown in the figure below.

        i. Select `maxihpm0_fpd_aclk` and drag it to `maxihpm1_fpd_aclk`.

        ii. Select `maxihpm1_fpd_aclk` and drag it to `pl_clk0`.

    b. Right-click on the Zynq UltraScale+ MPSoC block and select **Validate Design** to validate the design. It will say validation successful. Click **OK**.

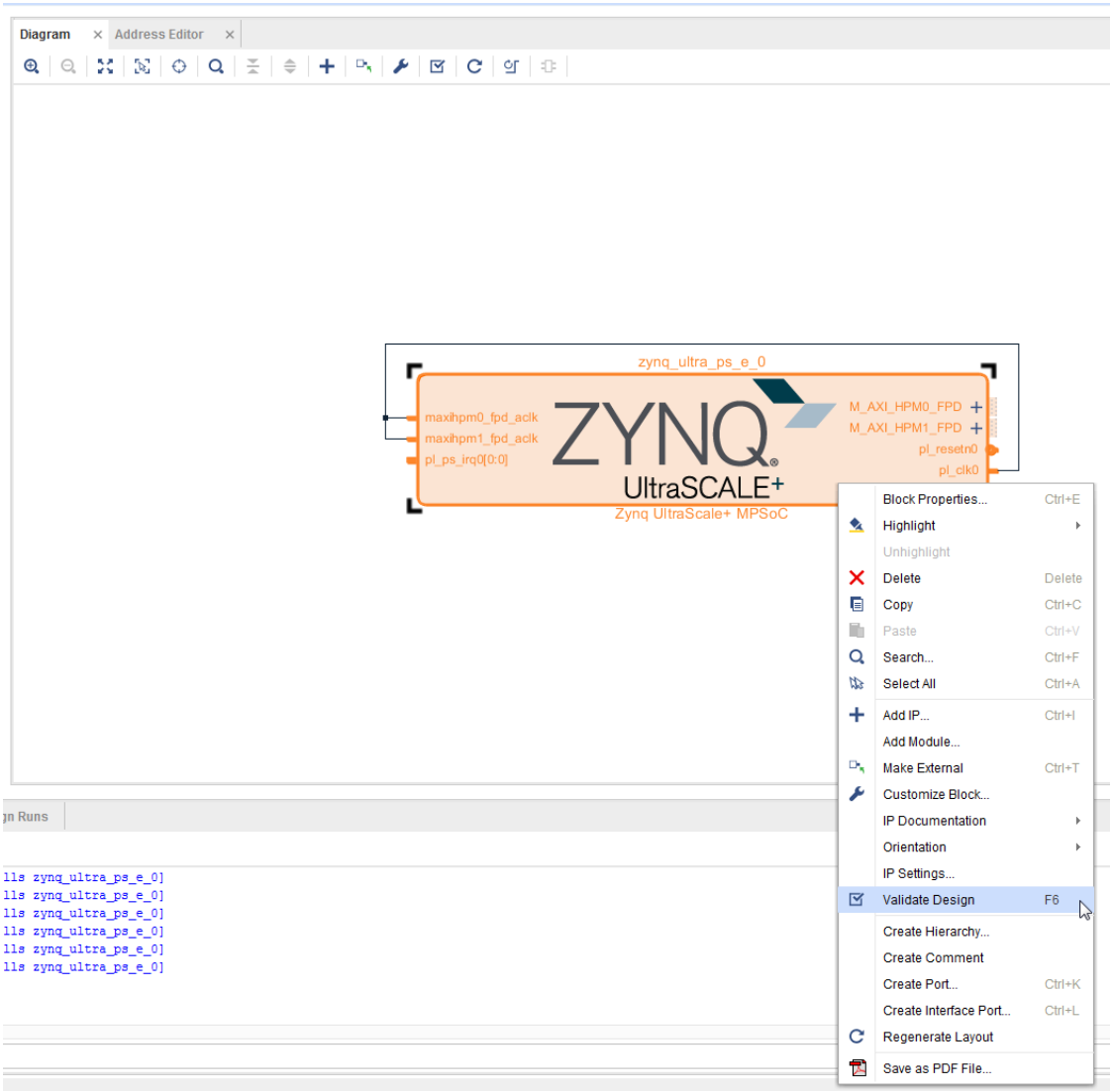

Figure 197: Validating the Design

12. Customize the design by double-clicking on the Zynq UltraScale+ MPSoC block and configuring the parameters. There are 4 valid GT configurations for ZCU102 board as shown below:

Table 2: Supported GTR Connector Functionality

SEL (S3,2,1,0)	ICM Settings (Lane 0,1,2,3)	PCIe Connector	DP Connector	USB Connector	SATA Connector
0 0 0 0	PCIe.o, PCIe.1, PCIE.2, PCIe.3	PCIe Gen2 x4	N.C.	N.C.	N.C.
1 1 1 1	DP.1, DP.0, USB, SATA	N.C.	DP.0, DP.1	USB0	SATA1
1 1 0 0	PCIe.0, PCIe.1, USB, SATA	PCIe Gen2 x2	N.C.	USB0	SATA1
1 1 1 0	PCIe.0, DP.0, USB, SATA	PCIe Gen2 x1	DP.0	USB0	SATA1

13. Select the settings based on your requirements by double-clicking on the Zynq UltraScale+ MPSoC block to customize GT Lane configuration.

14. Select **I/O Configuration** > **High Speed**. Select one of the four combinations using the settings in the screenshots below:

    a. PCIe – Display Port - USB - SATA (Default Vivado preset)

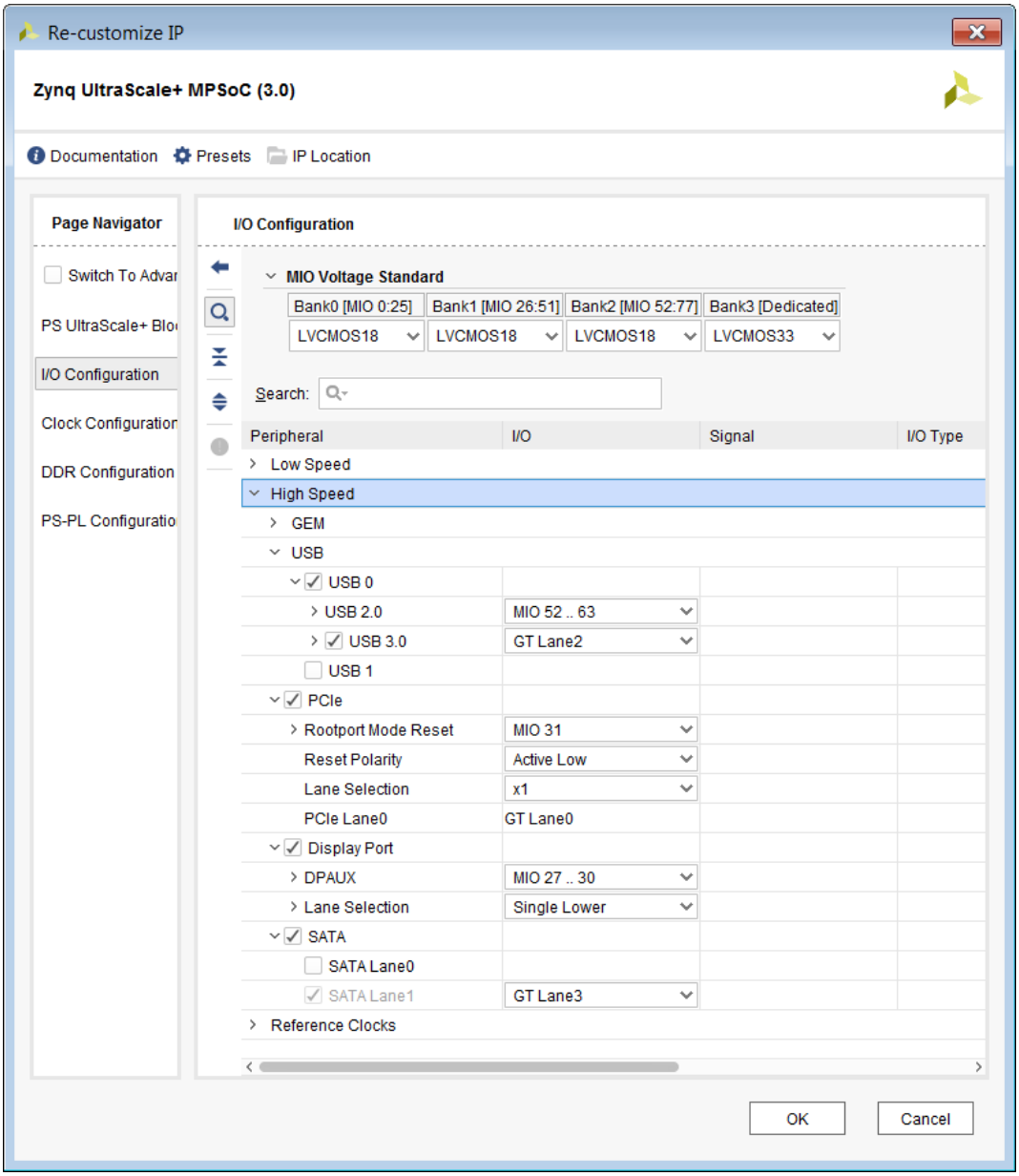

Figure 198: I/O Customization Window: PCIe – Display Port – USB – SATA (Default)

b. PCIe - PCIe - USB – SATA

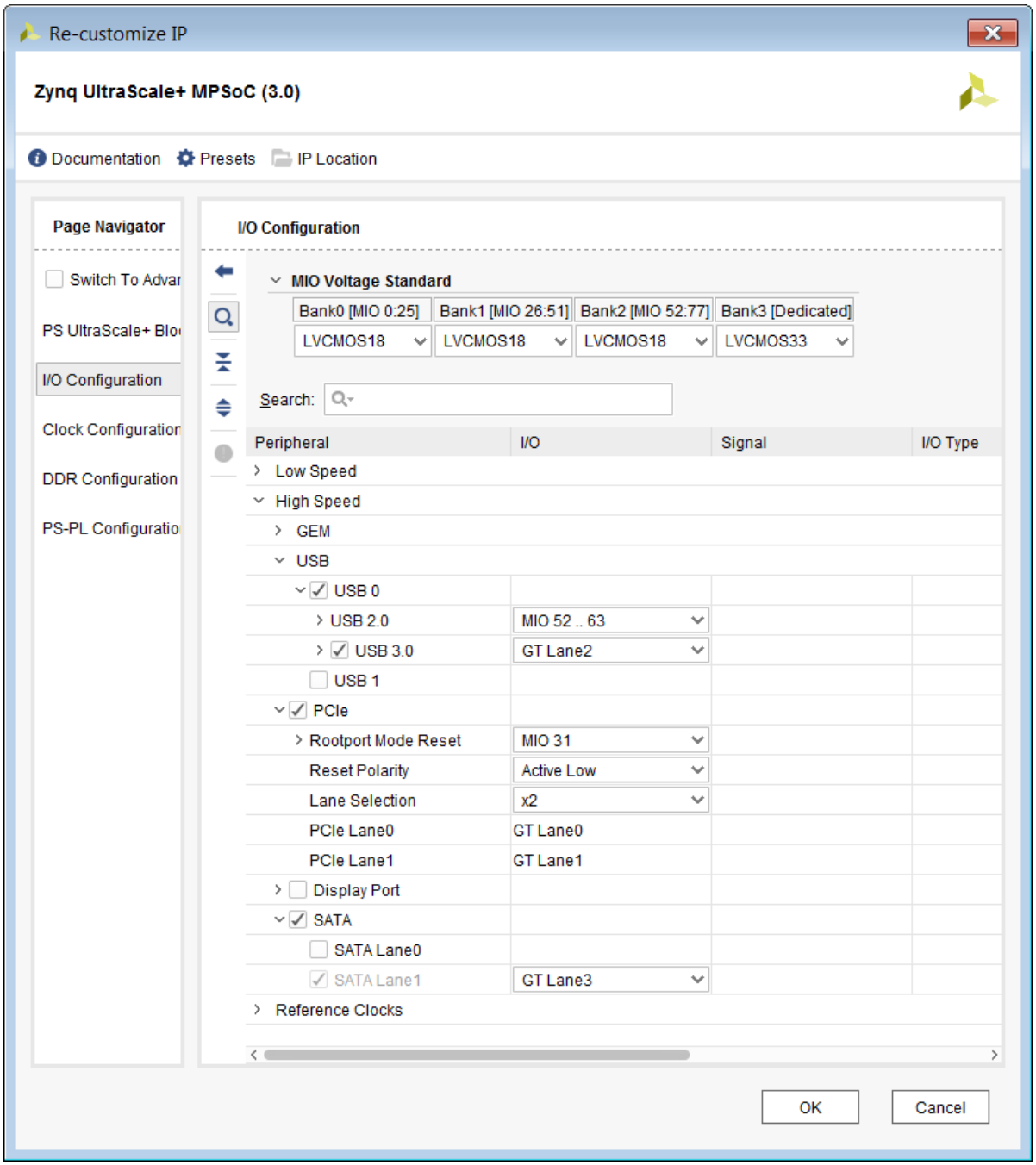

Figure 199: I/O Configuration Window: PCIe – PCIe – USB – SATA

c. Display Port - Display Port - USB – SATA

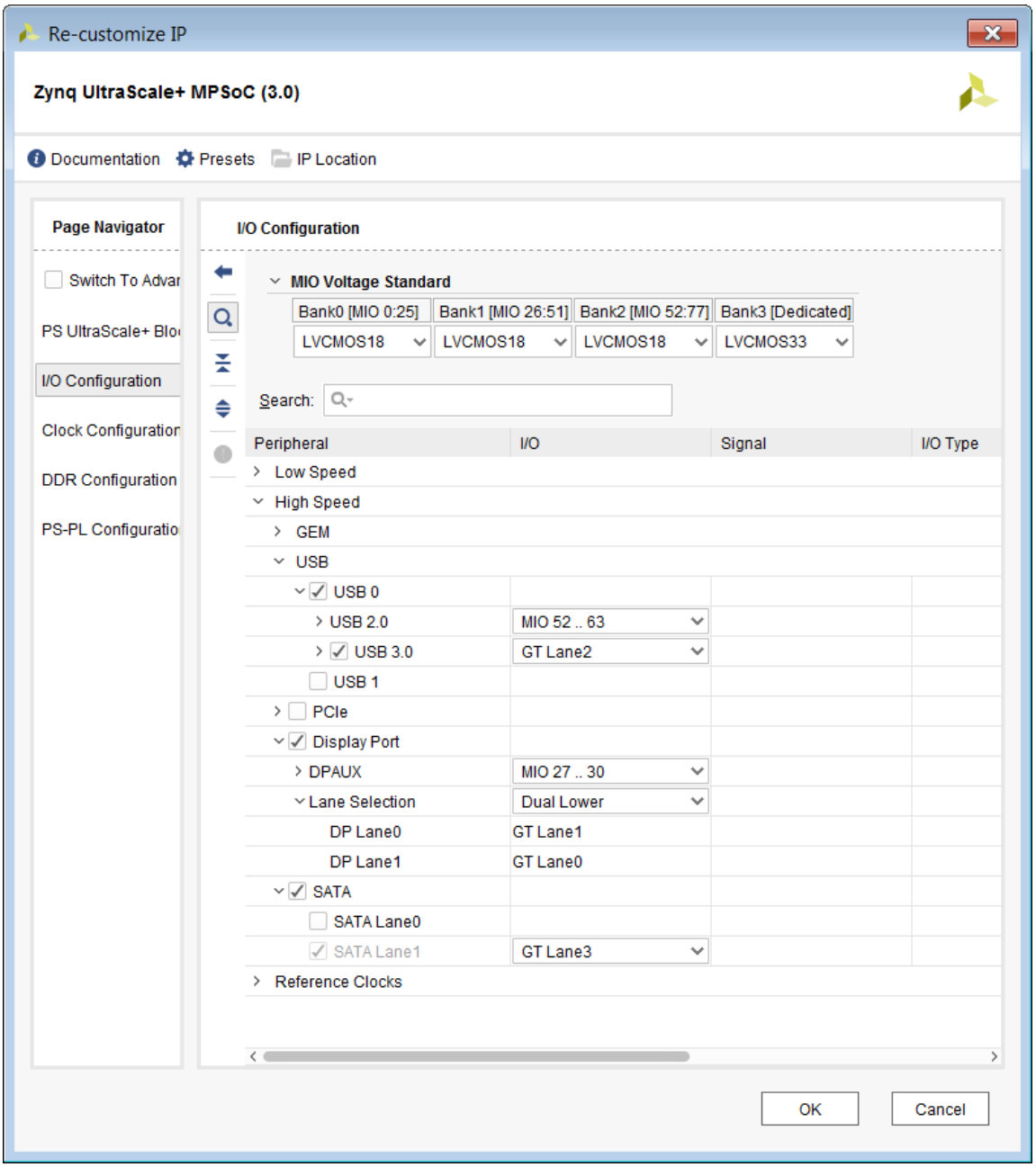

Figure 200: I/O Configuration Window: Display Port - Display Port - USB – SATA

d. PCIe - PCIe - PCIe – PCIe (PCIe x4)

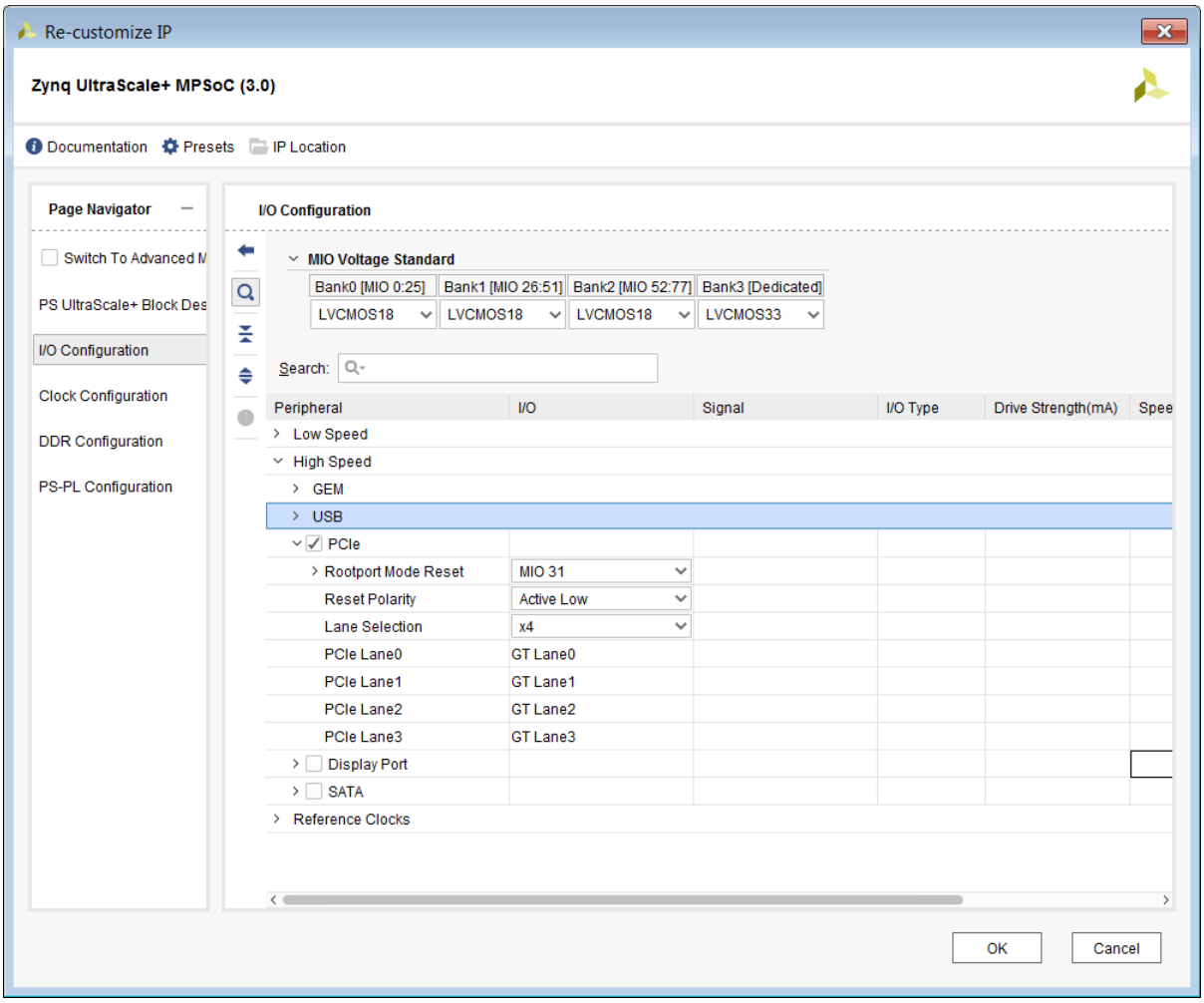

Figure 201: PCIe - PCIe - PCIe – PCIe

15. You can save customized presets using preset by selecting **Presets > Save Configuration** at the top of the window. This is useful if you want to save various configurations for future use.

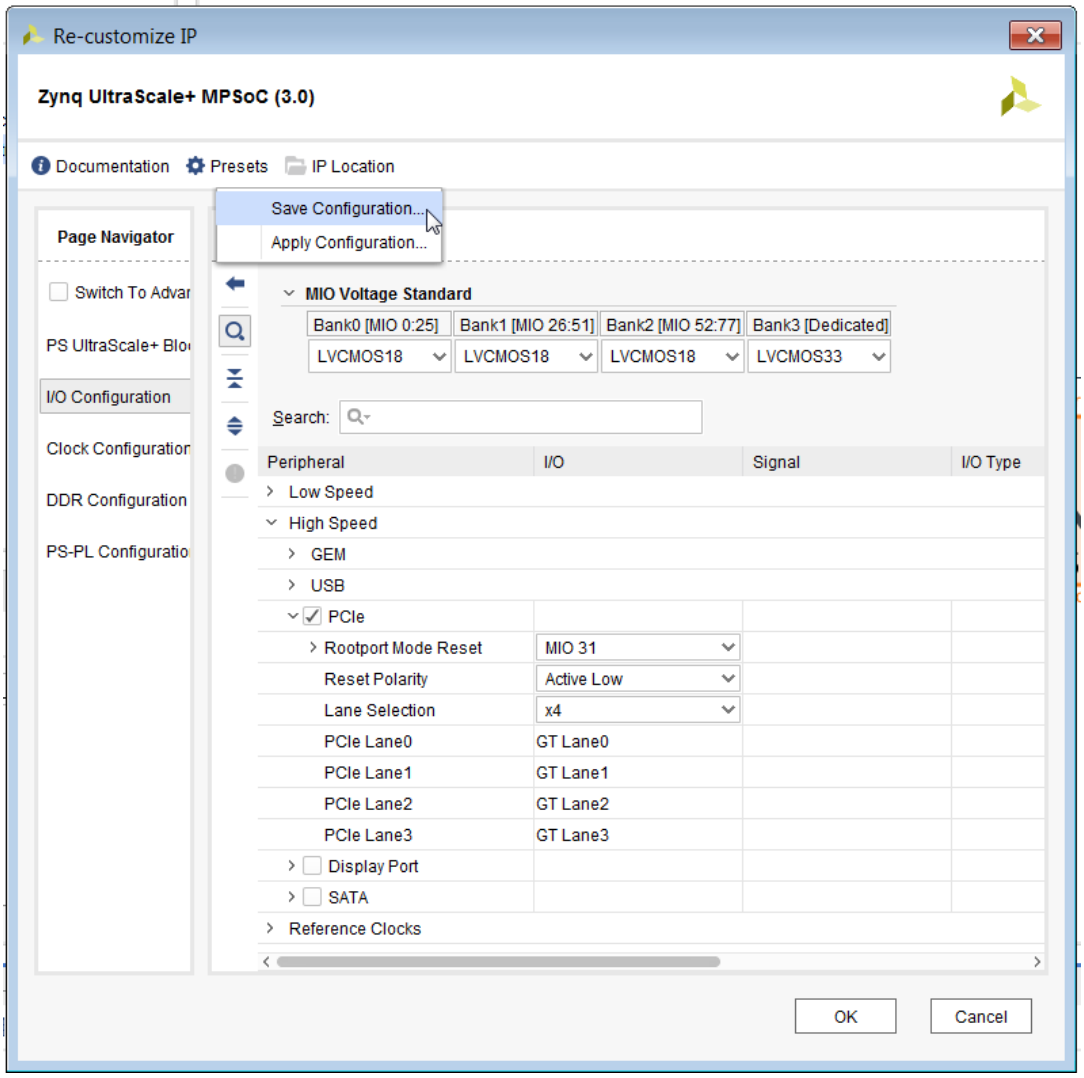

Figure 202: Selecting Save Configuration

16. Saved preset can be applied by selecting **Preset > Apply Configuration**. A file selection dialogue will appear as shown below. Choose the preset **PCIE-PCIE-USB-SATA** and click **OK**.

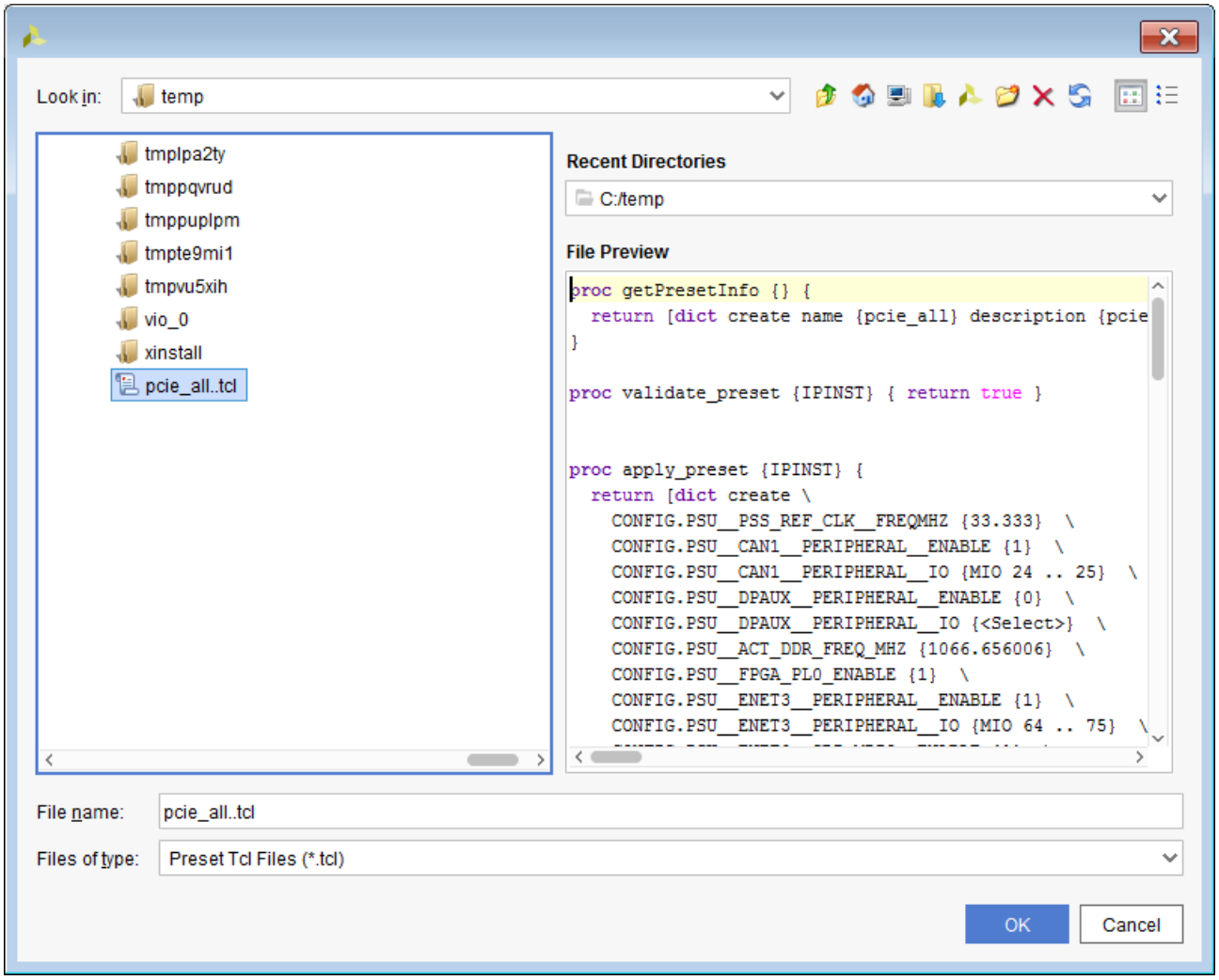

Figure 203: Applying a Preset Configuration

17. Click **OK** when finished customizing GT Lane configuration.
18. Do not click **Run Block Automation** again, even though the banner will reappear. It will reset the customized values if used.
19. Click on the **Sources** tab on the top left of the Block Design window. Under the Block Designs group, click on **IP Sources.** Right click on **design_1** and then click on **Create HDL Wrapper.**

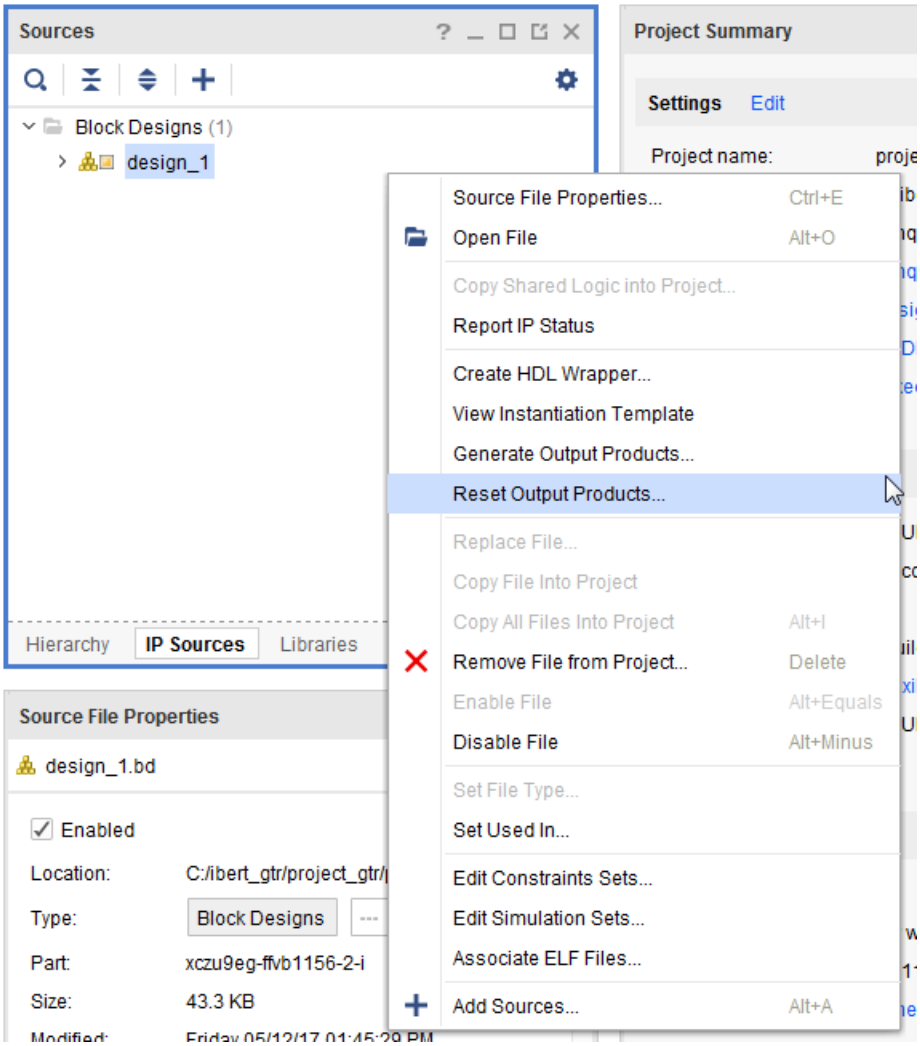

Figure 204: Create HDL Wrapper

20. Leave the option **Let Vivado manage wrapper and auto-update** selected. Click on **OK** in the dialogue to create HDL Wrapper.
21. Right click on **design_1_i** in **IP Sources** tab and click on **Generate Output Products**.
22. Click on **Generate** to generate with the default options in the panel.
23. Click **OK** after the generation is complete.

24. Select **File > Export > Export Hardware**.

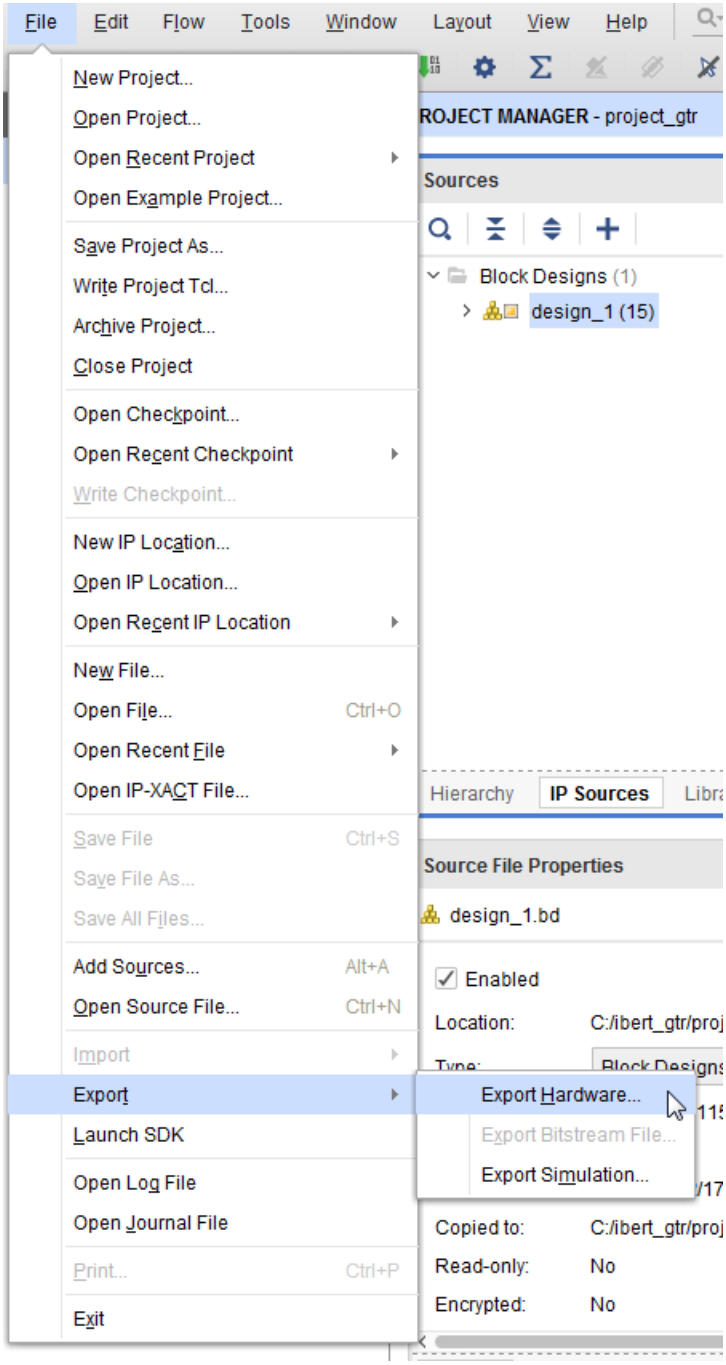

Figure 205: Export Hardware

25. You can specify an export directory or use `Local to Project`. Click **OK** to export the hardware.

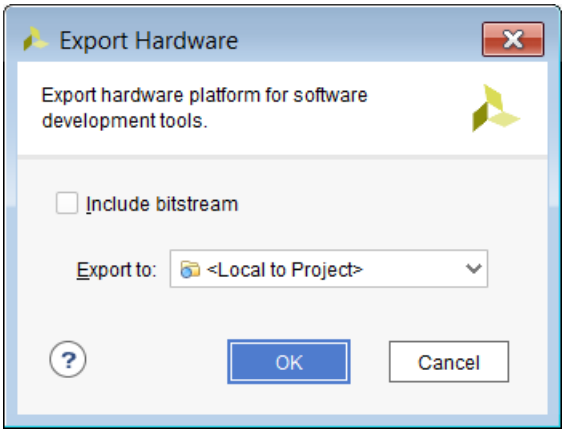

Figure 206: Export Hardware Window

26. You can see generated HDF path in the Tcl Console after the `write_hwdef` function call.

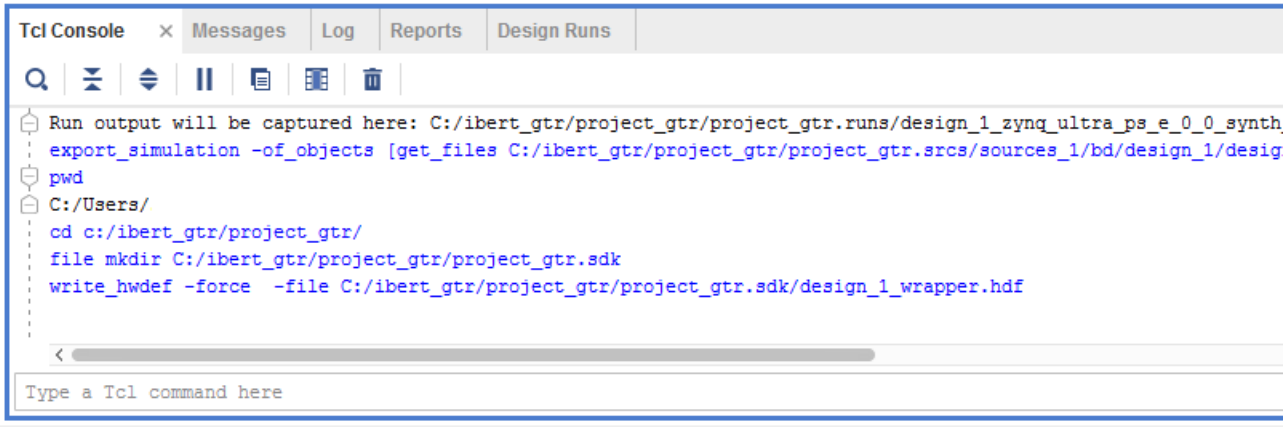

Figure 207: Running write_hwdef

Step 2: Using XSCT flow to generate FSBL by using HDF

Xilinx Software Command-line Tool (XSCT) is an interactive and scriptable command-line interface to Xilinx Software Development Kit (Xilinx SDK). The XSCT flow should be used for a more automated flow that is Tcl based and only requires running a Tcl script. There is an SDK flow that should be used for a more interactive setup of the FSBL which uses a GUI. You can use the SDK flow if you need to make changes/customizations to the FSBL. SDK flow could also be used for custom board. See "Appendix A" for the SDK or XSCT Manual flow.

Generating using XSCT Automated Flow:

To create an FSBL for the A53 #0 (64 bit) automatically (and modify the xfsbl_main.c/h files if USB is present) using the provided script, use the follow steps:

1. Copy the `src/lab10/xsct_create_fsbl.tcl` script to the directory where the HDF file is located (e.g. `project_gtr/project_gtr.sdk`). You can modify the Tcl script if you changed the default name of the HDF file in Vivado. You can also change the script if the compiler options need to be different.
2. Open a terminal on Linux or command prompt on Windows.
3. Cd into the directory where the HDF file is located.
4. Call `xsct` from the SDK install area:

   ```
   % xsct xsct_create_fsbl.tcl
   ```

5. It prints out the location of the generated `.elf` file when the script completes.

Step 3: ZCU102 Board Settings

USB jumper setting requirements for HOST mode on ZCU102:

1. Make sure below jumpers are correctly set for USB to be in HOST mode (refer to UG1182).
   e. J7 – ON
   f. J113 – 1-2
   g. J110 – 2-3
2. Refer to the below image for jumper settings on a ZCU102 Rev 1.0 board:

Figure 208: Jumper Settings for the ZCU102 Board

Using FSBL with Serial I/O Analyzer to bring up IBERT GTR

1. Connect all the physical devices such as SATA Drive, PCIE card, and USB device based on your selection from the four valid GT configurations for ZCU102 prior to loading the FSBL. Hot swap or hot plug is not supported.
2. Open Vivado.

3. Open hardware manager and connect to a board with a Zynq UltraScale+ device. The example below shows connecting to a board on a remote machine, so `hw_server` needs to be running on the remote machine before it can connect.

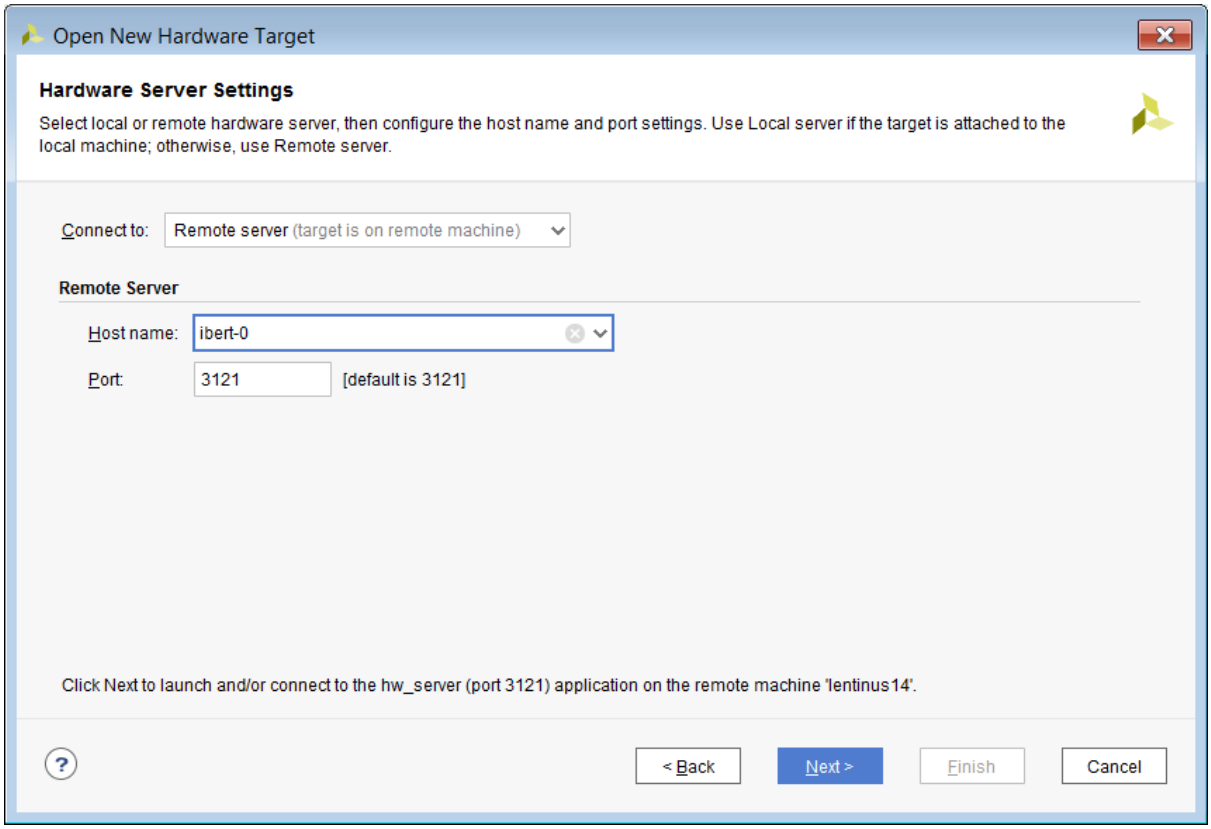

Figure 209: Hardware Server Settings

4. Verify the **ARM_DAP** is visible in the hardware device list and click **Next**, and then click **Finish**.

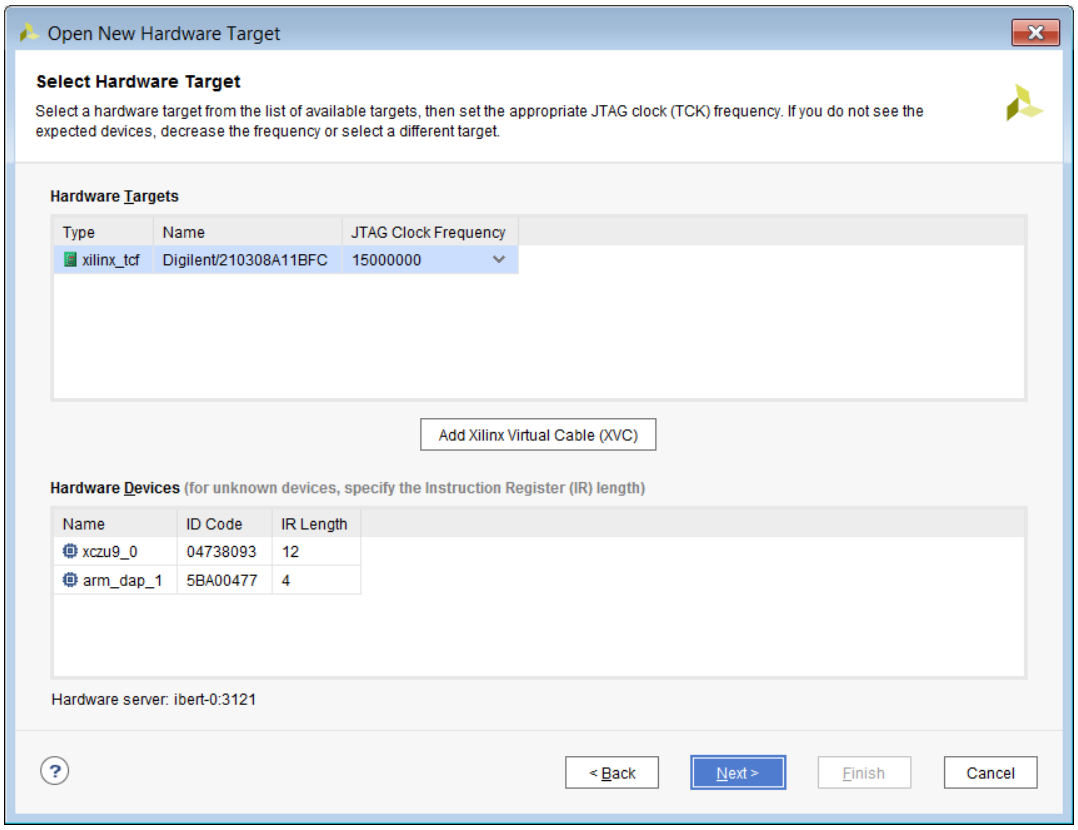

Figure 210: Selecting the Device

5. Right-click on the **ARM_DAP** device in the hardware tree and select **Configure IBERT GTR**.

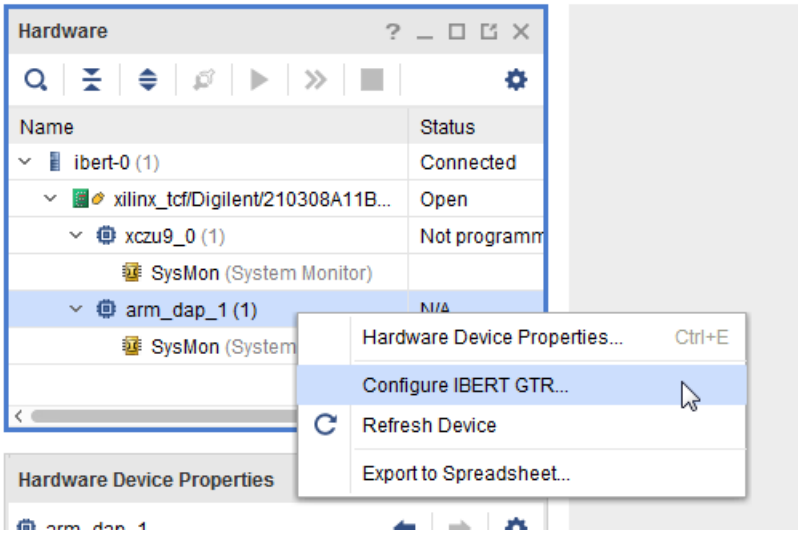

Figure 211: Selecting Configure IBERT GTR

6. When the dialog box opens, you must provide the FSBL ELF file created in the previous steps and optionally a configuration file (a bitstream, if your design requires one). You can also reset the system before configuring with the **Reset Zynq** option checked. Click **OK** when done.

   Note: The **Reset Zynq** option leaves the ARM_DAP in a bad state on early versions of Zynq UltraScale+ devices (e.g. ZU9EG es1). If that occurs, power cycle the board and keep the **Reset Zynq** option unchecked.

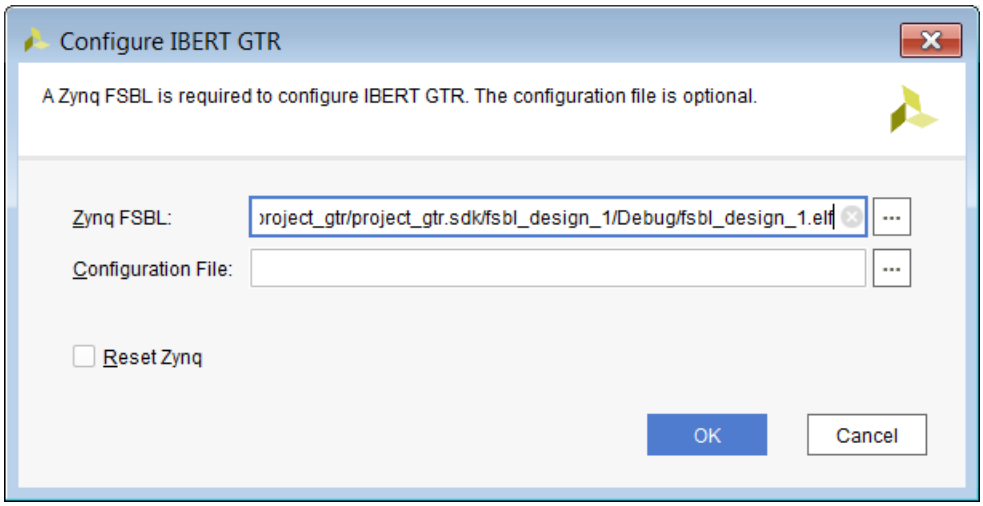

Figure 212: Selecting the FSBL File

7. `config_hw_sio_gts` is executed with the selected settings. `refresh_hw_device` is then called to rescan the device for new debug cores. The IBERT should be configured as shown in the example below:

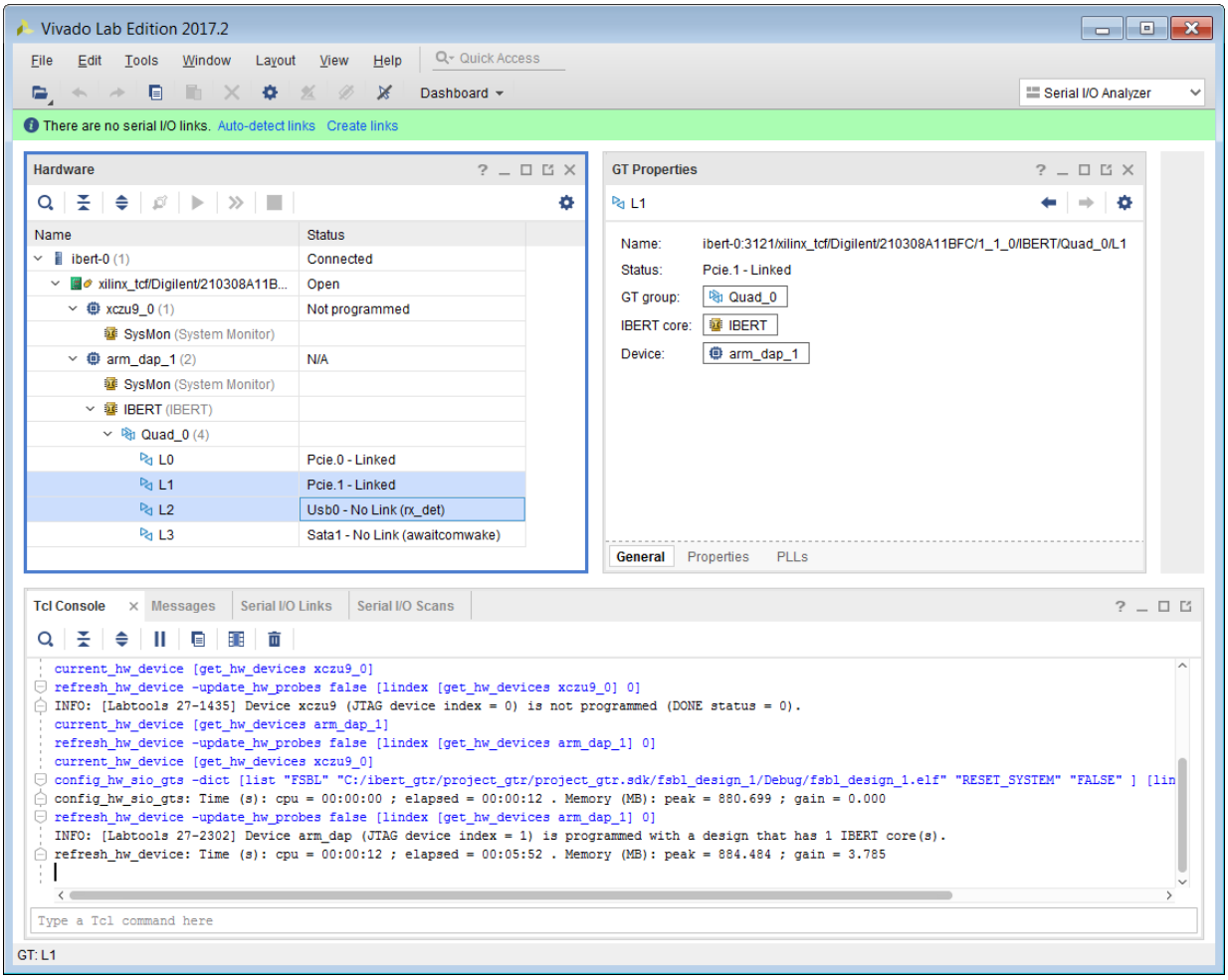

Figure 213: IBERT Configuration

8. The **Auto-detect links** option does not work for GTR. You can manually create links by using **Create Links** as shown below:

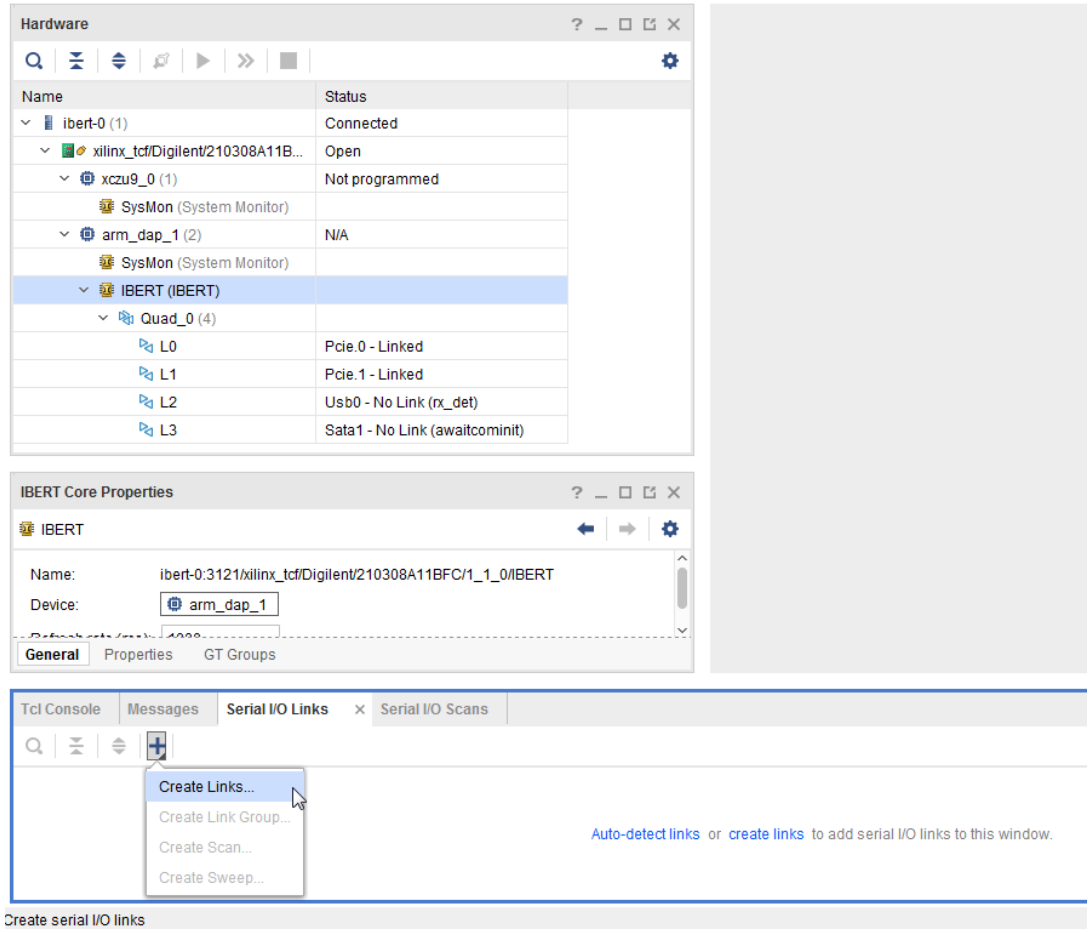

Figure 214: Selecting Create Links

9. Create links for all four lanes with each lane's **TX** connected to the same lane's **RX**, as shown in the figure below.

   Click **OK** when done.

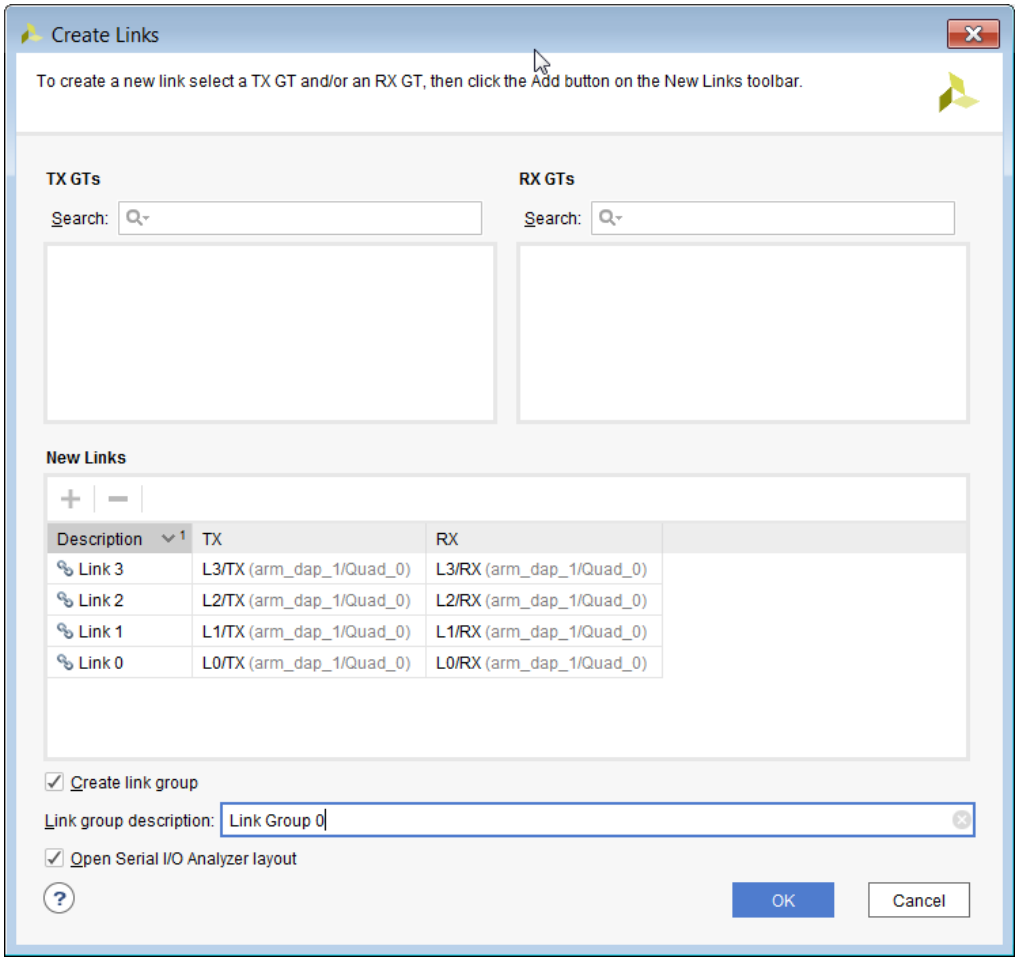

Figure 215: Creating Links

10. The figure below shows the Serial I/O Links view where **Status** shows all the four lanes as linked.

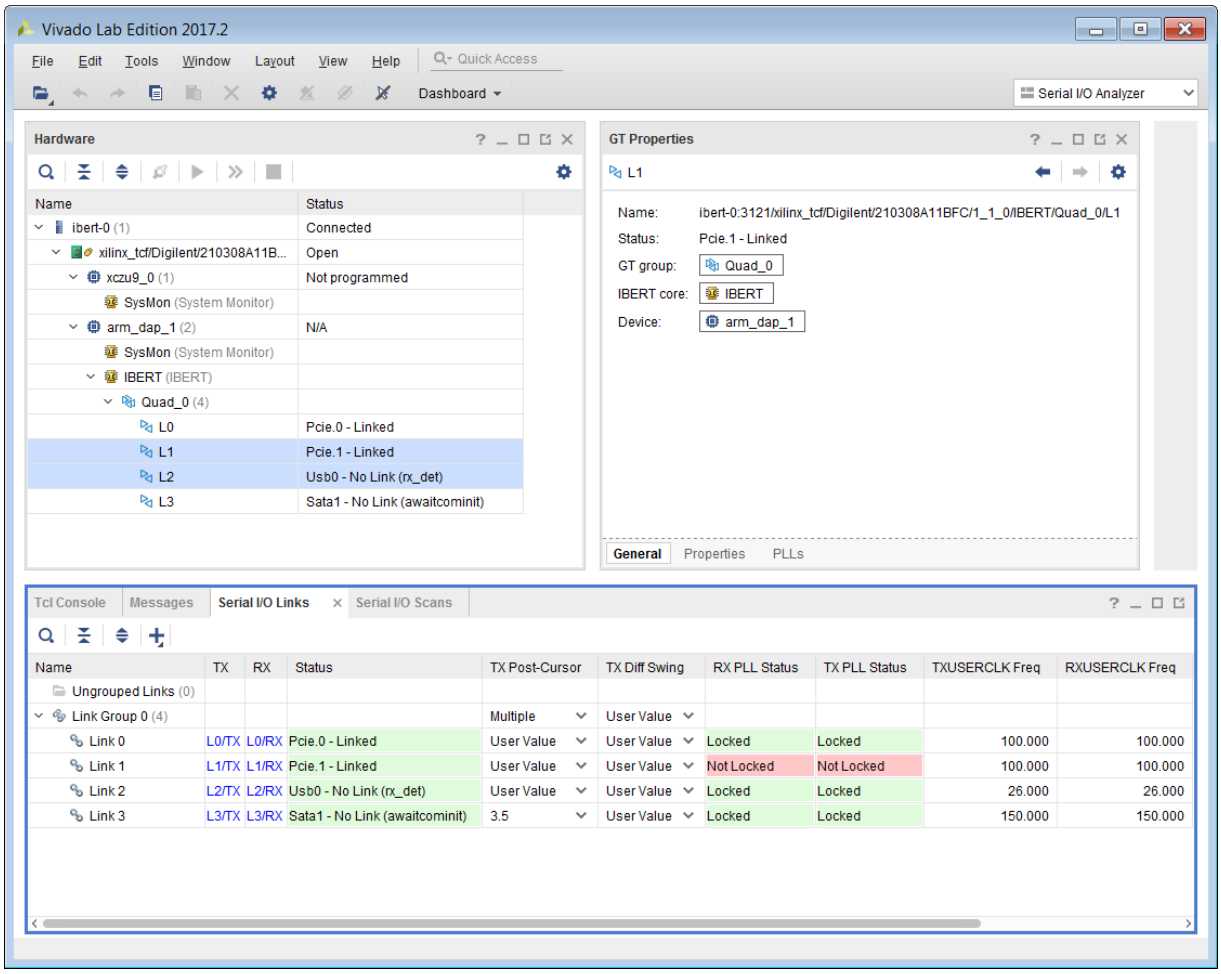

Figure 216: Serial I/O Links View

Note: The **Link 1** PLL Status shows **Not Locked**, because it uses the **Link 0** PLL as required by PCIe protocol.

11. Right-click on any link and select **Create Scan**.

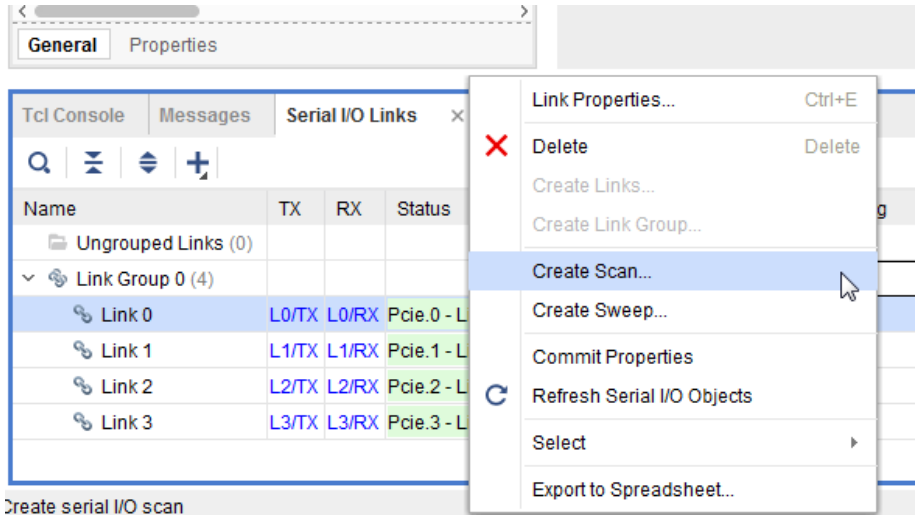

Figure 217: Selecting Create Scan

12. Select the appropriate parameters for EyeScan and perform the EyeScan. For example, the figure below is performing EyeScan on Lane L1 (Link 1). Once the EyeScan completes, the eye from -1UI to +1UI will be displayed.

    Note: Although the **Create Scan** pop up shows -0.5UI to +0.5UI, the EyeScan displayed is from -1UI to +1UI.

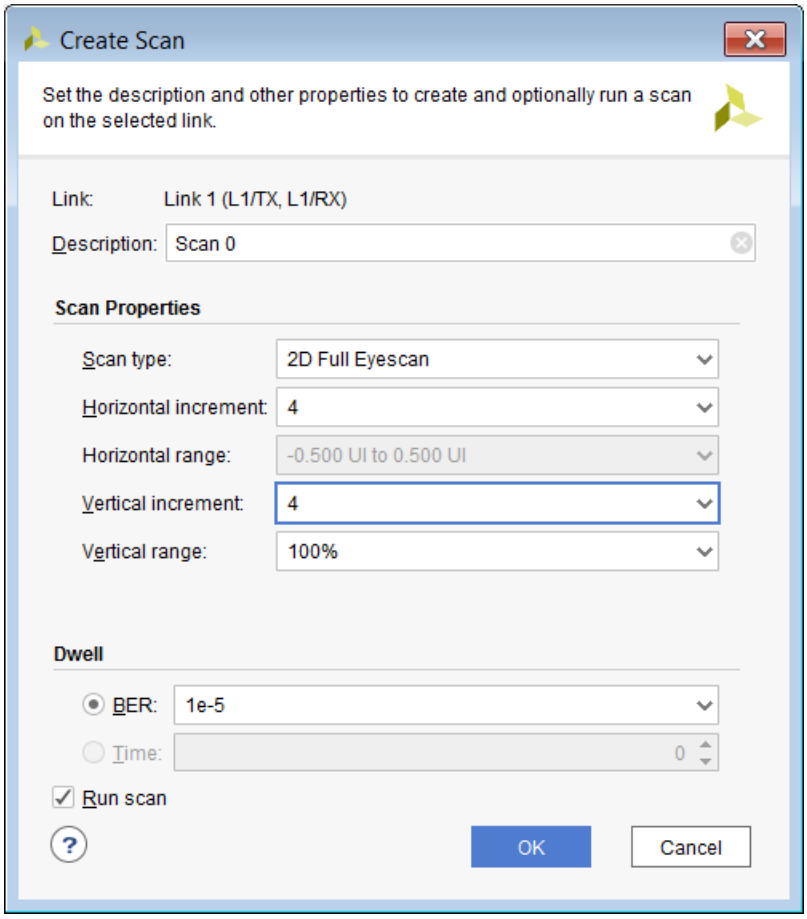

Figure 218: Selecting EyeScan Parameters

13. Below is a sample EyeScan performed on Lane L1:

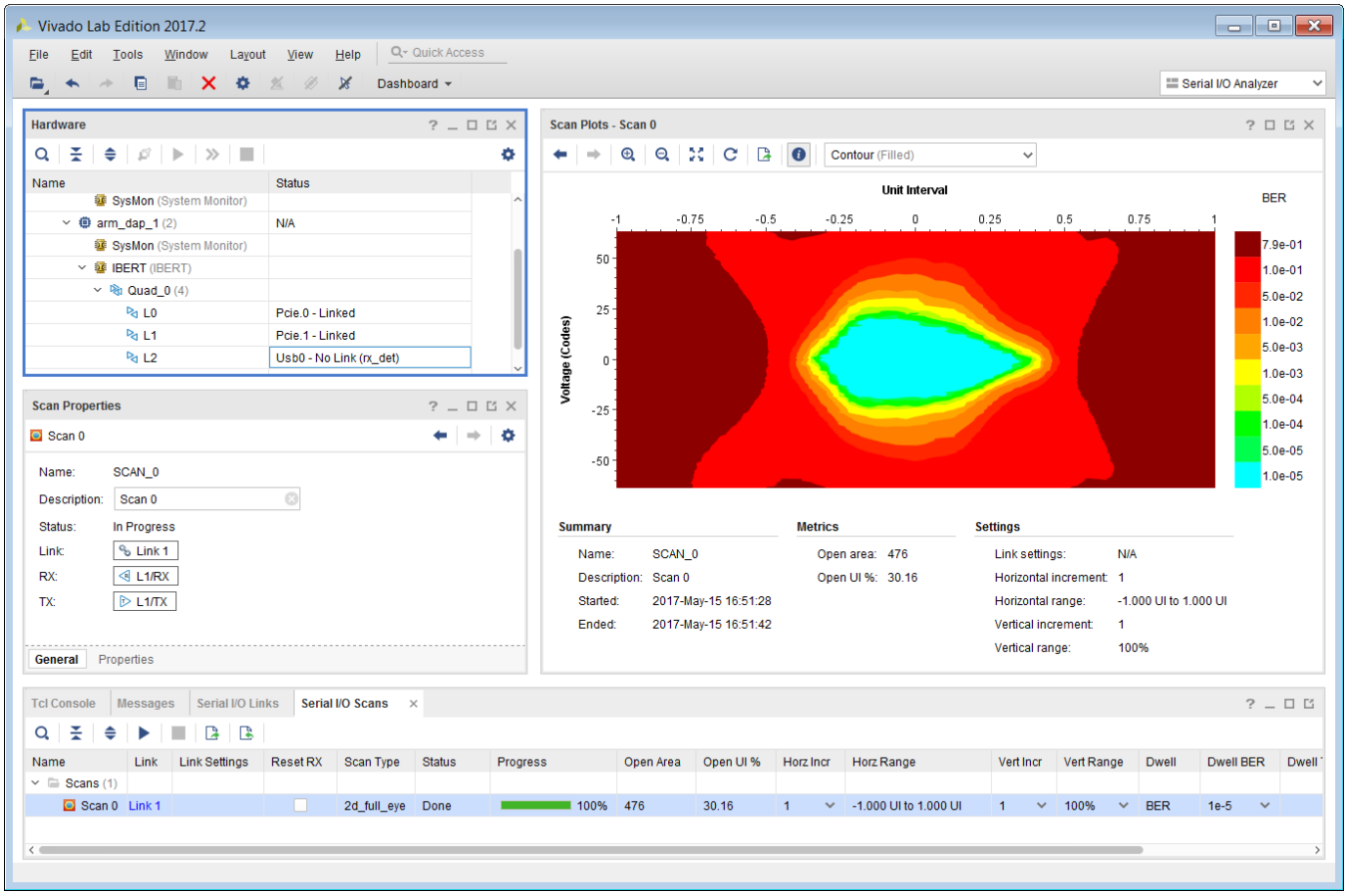

Figure 219: Sample EyeScan

Note: The value reported by **Open UI %** is a percentage of the entire horizontal axis, which is 2UI wide for GTR.

Troubleshooting

Known Issues

1. By default, FSBL does not enumerate USB as that is something Linux drivers would do. To put USB in link state without Linux, a small modification is required in the FSBL C-code. This modification still does not enumerate the device, it only brings the USB into link state.
2. The EyeScan does not have a built-in time-out mechanism. If your link is poor (for example, if `L*_TM_DIG_8.EYESURF_ENABLE != 1`), then the EyeScan will hang without providing a user. No results are returned in this case.
3. If the EyeScan progress is not moving, make sure the below parameters for all lanes are set for EyeScan to function correctly. Note that `*` represents the lane number (as in, for Lane 0 the parameter would be `L0`).

   Click on the lane in the hardware tree and then click on the properties tab. There's a search button you can use to find the properties below.

   a. `L*_TM_MISC3.CDR_EN_FPL = 0`

   b. `L*_TM_MISC3.CDR_EN_FFL = 0`

   c. `L*_TM_DIG_8.EYESURF_ENABLE = 1`

   Also check below parameters values which ensures Eye Scan circuit is operational.

   d. `L*_PLL_LOCK = 1`

   e. `L*_TM_SAMP_STATUS4.E_SAMP_PH0_CALIB_CODE` is non-zero value

   f. `L*_TM_SAMP_STATUS5.E_SAMP_PH180_CALIB_CODE` is non-zero value

Notes

1. As mentioned in Assumptions, IBERT GTR uses the psu_cortexr5_0 core, so no other applications should use this core.
2. TCM0 and TCM1 memory are combined to form a unified memory for IBERT GTR. Any other processor core should not access this memory while IBERT GTR is running.
3. The error counter is 16 bits and the sample counter is 32 bits. Each sample can have 8 bits of error count. Therefore on the edges, the error counter can saturate with a sample count value of 8192. GTR does not stop the sample counter even if the error counter saturates. A prescale=0 produces 8192 samples and thus a total samples of 8192 *8 (65536) and thus the outside edges of eye could show a BER of e-01 or less depending on the prescale selected.
4. The EyeScan assumes there is link present. If there is no link, then the EyeScan may not complete. Cancelling the EyeScan stops the command sequence, but the state of the previous point scan will be unknown.

5. If you run EyeScan and because of no link the EyeScan does not complete, set the register `L*_TM_MISC_ST_0.EYE_SURF_RUN` to `0` for the given lane before you run the EyeScan again.
6. If you run EyeScan on a lane that is either powered down or Display Port, it will immediately stop and the scan will be marked as incomplete. EyeScan will not work in either scenario.

Using SDK Flow or XSCT Flow to Generate FSBL by Using HDF

The SDK flow should be used for a more interactive setup of the FSBL which uses a GUI as well as when creating FSBL for custom board. The XSCT flow should be used for a Tcl based flow.

Generating using SDK Flow:

1. Launch SDK from Vivado with the opened Zynq project from earlier with the HDF file generated. This could be either for ZCU102 board or custom Zynq board. Launch the SDK by selecting **File > Launch SDK**.

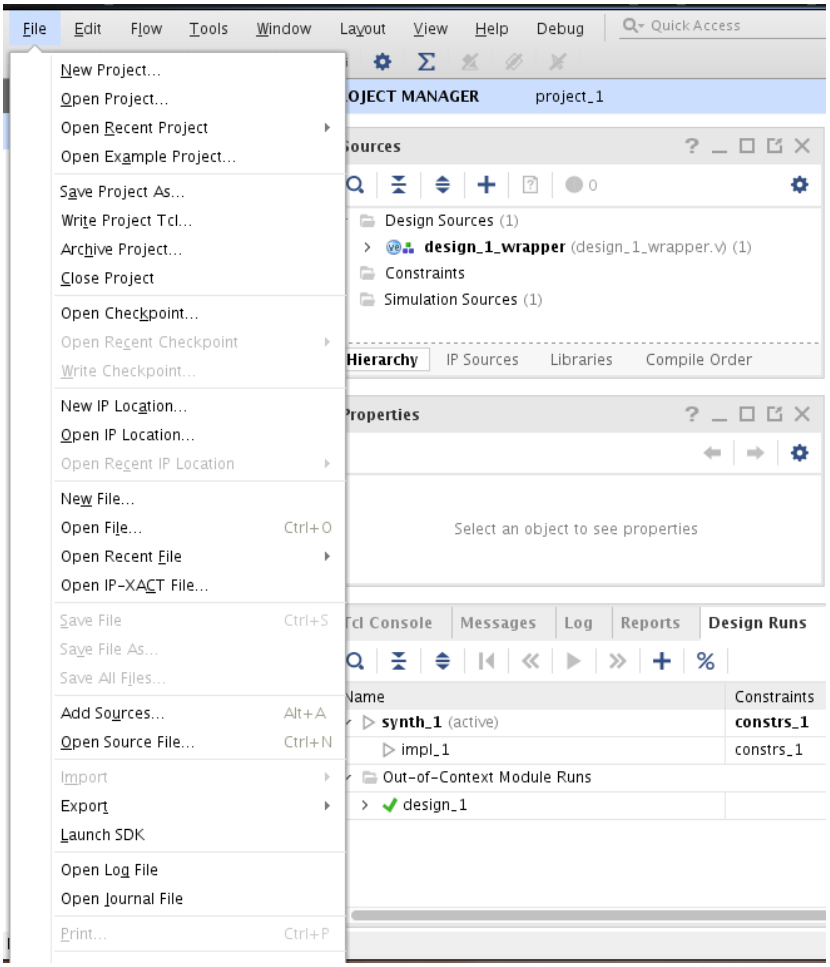

Figure 220: Launching the SDK

2. Provide the **Exported Location** path (where your `*.hdf` file was exported). The SDK **Workspace** could also be the same path. This creates the SDK workspace. Click **OK**.

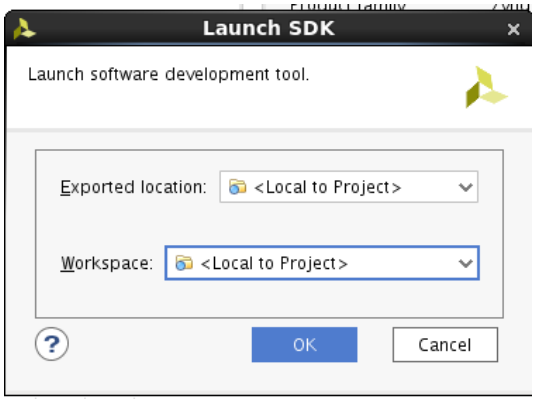

Figure 221: Creating the SDK Workspace

3. Once the SDK is open, select **File > New > Application Project** to open the New Project window. Provide a name for the FSBL project
4. The Target Hardware section is be populated by your generated hardware platform from Vivado along with the processor **psu_cortexa53_0**. Keep this processor for FSBL generation.

   Note: The psu_cortexr5_0 is used by the IBERT GTR. No other applications should use this core.

5. For **psu_cortexa53_***, select **64-bit** (default) for **Compiler**.

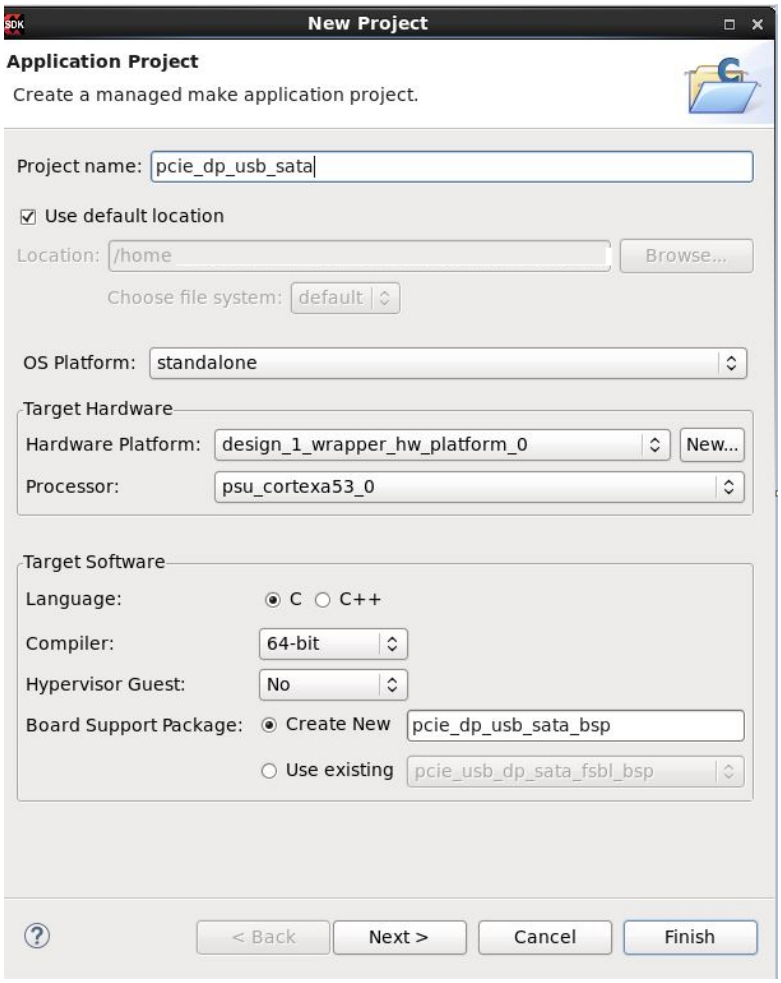

Figure 222: Creating a New Project

6. Leave other options set to their defaults and click **Next**.

7. Select the **Zynq MP FSBL** template.

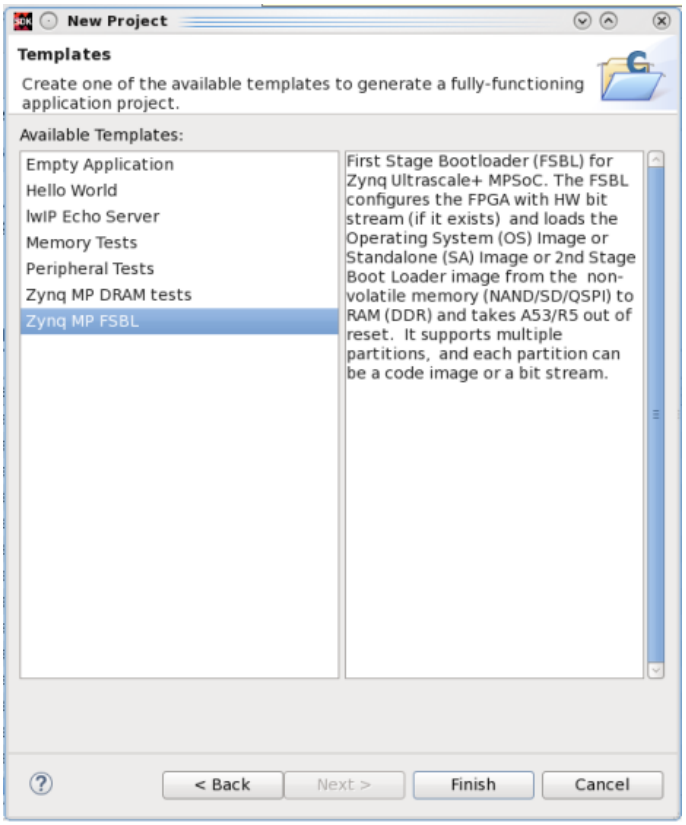

Figure 223: Selecting the New Project Template

8. Click **Finish** to generate the A53 FSBL. This populates the FSBL code and also builds it, along with the BSP.
9. Change the compiler optimization flag from `O0` to `O2`. To change this in the SDK, follow these steps:
   a. Right-click the FSBL application project and select **C/C++ Build Settings**.
   b. On the **Tool Settings** tab, select **Optimization** (under the ARM A53 gcc compiler).
   c. On the **Optimization Level** drop-down menu, change **None (-O0)** to **Optimize more (-O2)**.
   d. Click **OK** to close the Properties window.
10. Debug prints in FSBL are now disabled by default (except for FSBL banner). To enable debug prints, you must define the symbol FSBL_DEBUG_INFO. To change this in the SDK, follow these steps:
   a. Right-click the FSBL application project and select **C/C++ Build Settings**.
   b. On the **Tool Settings** tab, select **Symbols** (under the ARM A53 gcc compiler).
   c. Click the **Add** button in the **Defined symbols (-D)** section and enter the value: `FSBL_DEBUG_INFO`
   d. Click **OK** to close the Properties window.

11. Skip this step if you are building FSBL for a custom board.

    For building the FSBL for a ZCU102, define the additional symbol XPS_BOARD_ZCU102. Note that in this flow, the Hardware Platform (HDF) was created using the ZCU102 board preset, so the XPS_BOARD_ZCU102 symbol may be defined by default in the FSBL. If it is present, it does not need to be redefined.

    To define the symbol in the SDK, follow these steps:

    a. Right-click the FSBL application project and select **C/C++ Build Settings**.

    b. On the **Tool Settings** tab, select **Symbols** (under ARM A53 gcc compiler).

    c. Click the **Add** button in the **Defined symbols (-D)** section and enter the value: `XPS_BOARD_ZCU102`

    d. Click **OK** to close the Properties window.

12. If the current configuration has **USB** set on one of the GTR lanes, then the FSBL source files need to be modified. This is needed to set the USB in a linked state. This step should only be performed when using an "FSBL only" setup with IBERT GTR. When Linux setup is used, USB drivers set the link appropriately and this step is not needed.

13. Open the files in SDK mentioned in next step, located in the applications `src` folder.

    Make the appropriate changes and save the file, which automatically builds the project and generates `<application_name>.elf` in the `Debug` folder inside the applications folder.

    If building for a custom board, make sure you do the appropriate changes to bring up the USB in a linked state. After this step the FSBL should be ready to be used in Vivado.

Bringing the USB into a Linked State

xfsbl_main.h

After the first line below, add the EyeScan changes as highlighted to `xfsbl_main.h`:

```
#define XFSBL_IMAGE_SEARCH_OFFSET      (0x8000U) /**< 32KB offset */
/**
 * EyeScan
 */
#undef USB3_0_XHCI_GCTL_OFFSET
#define USB3_0_XHCI_GCTL_OFFSET   0XFE20C110
#define USB3_0_XHCI_GCTL_U0_LTSSM_MASK    0XFFFFDFFFU /*GCTL[13:12]=01*/
```

xfsbl_main.c

After the lines below, add the EyeScan changes as highlighted to `xfsbl_main.c`:

```
XFsbl_Printf(DEBUG_INFO,
   "================= In Stage 4 ============ \n\r");
//EyeScan
u32 RegValue;
RegValue = Xil_In32(USB3_0_XHCI_GCTL_OFFSET) &
   USB3_0_XHCI_GCTL_U0_LTSSM_MASK;
Xil_Out32(USB3_0_XHCI_GCTL_OFFSET, RegValue);
```

Generating using XSCT Manual Flow:

1. To create an FSBL for the A53 #0 (64 bit), type `xsct` from shell in the directory where the HDF file is located.
2. From the xsct console, issue the following commands:

```
setws
createhw -name hw0 -hwspec <path to hdf generated by you>
createapp -name fsbl_design_1 -app {Zynq MP FSBL} -proc psu_cortexa53_0 -
   hwproject hw0 -os standalone -arch 64 -lang C
configapp -app fsbl_design_1 -set compiler-optimization {Optimize more (-O2)}
configapp -app fsbl_design_1 -add define-compiler-symbols FSBL_DEBUG_INFO
```

3. If using the ZCU102, also run this command:

```
configapp -app fsbl_design_1 -add define-compiler-symbols XPS_BOARD_ZCU102
```

4. Make changes to `xfsbl_main.c` and `xfsbl_main.h` as shown above in Bringing the USB into a Linked State if the USB is in one of the GTR lanes. Save these files.
5. Run the following command:

```
projects -build
```

The FSBL ELF file, `fsbl_design_1.elf`, will be generated in the `fsbl_design_1/Debug/` directory.

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 https://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 https://www.xilinx.com/legal.htm#tos.

AUTOMOTIVE DISCLAIMER

AUTOMOTIVE PRODUCTS (IDENTIFIED AS “XA” IN THE PART NUMBER) ARE NOT WARRANTED FOR USE IN THE DEPLOYMENT OF AIRBAGS OR FOR USE IN APPLICATIONS THAT AFFECT CONTROL OF A VEHICLE (“SAFETY APPLICATION”) UNLESS THERE IS A SAFETY CONCEPT OR REDUNDANCY FEATURE CONSISTENT WITH THE ISO 26262 AUTOMOTIVE SAFETY STANDARD (“SAFETY DESIGN”). CUSTOMER SHALL, PRIOR TO USING OR DISTRIBUTING ANY SYSTEMS THAT INCORPORATE PRODUCTS, THOROUGHLY TEST SUCH SYSTEMS FOR SAFETY PURPOSES. USE OF PRODUCTS IN A SAFETY APPLICATION WITHOUT A SAFETY DESIGN IS FULLY AT THE RISK OF CUSTOMER, SUBJECT ONLY TO APPLICABLE LAWS AND REGULATIONS GOVERNING LIMITATIONS ON PRODUCT LIABILITY.