Notice of Disclaimer

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 the Limited Warranties which can be viewed at http://www.xilinx.com/warranty.htm; 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 Critical Applications: http://www.xilinx.com/warranty.htm#critapps.

©Copyright 2012-2013 Xilinx, Inc. Xilinx, the Xilinx logo, Artix, ISE, Kintex, Spartan, Virtex, Vivado, Zynq, and other designated brands included herein are trademarks of Xilinx in the United States and other countries. All other trademarks are the property of their respective owners.

Revision History

The following table shows the revision history for this document.

<table>
<thead>
<tr>
<th>Date</th>
<th>Version</th>
<th>Revision</th>
</tr>
</thead>
<tbody>
<tr>
<td>10/16/12</td>
<td>2012.3</td>
<td>• Added section for Synplify EDIF flow.</td>
</tr>
<tr>
<td></td>
<td></td>
<td>• Various editorial fixes and updated screen shots to track changes to dialog boxes.</td>
</tr>
<tr>
<td>12/18/2012</td>
<td>2012.4</td>
<td>Revalidated for the current release. Editorial updates only; no technical content updates.</td>
</tr>
<tr>
<td>03/20/2013</td>
<td>2013.1</td>
<td>• Converted source to Word.</td>
</tr>
<tr>
<td></td>
<td></td>
<td>• Updated DDS IP and add VIO 2.0</td>
</tr>
<tr>
<td>06/19/2013</td>
<td>2013.2</td>
<td>Editorial updates only; no technical content updates.</td>
</tr>
<tr>
<td>07/07/2013</td>
<td>2013.2</td>
<td>Updated design source files.</td>
</tr>
<tr>
<td>10/02/2013</td>
<td>2013.3</td>
<td>• Added new lab on using Vivado® Serial Analyzer</td>
</tr>
<tr>
<td></td>
<td></td>
<td>• Updated design source files.</td>
</tr>
<tr>
<td>12/18/2013</td>
<td>2013.4</td>
<td>• Added new lab on Using Vivado® ILA core to debug JTAG-AXI Transactions.</td>
</tr>
</tbody>
</table>

Send Feedback
# Table of Contents

Vivado Design Suite Tutorial: ........................................................................................................................................................... 1

Programming and Debugging ........................................................................................................................................................ 1

  Revision History ................................................................................................................................................................. 2

Debugging in Vivado® ........................................................................................................................................................ 5

  Introduction ................................................................................................................................................................. 5

  Objectives ................................................................................................................................................................. 5

  Getting Started........................................................................................................................................................ 6

Lab 1: Using the Netlist Insertion Method for Debugging a Design in Vivado® ......................................................... 11

  Introduction .......................................................................................................................................................... 11

  Step 1: Creating a Project with the Vivado New Project Wizard ............................................................................ 11

  Step 2: Synthesizing the Design ....................................................................................................................... 12

  Step 3: Probing and Adding Debug IP ............................................................................................................ 14

  Step 4: Implementing and Generating Bitstream ............................................................................................. 21

Lab 2: Using the HDL Instantiation Method for Debugging a Design in Vivado® ............................................. 23

  Introduction .......................................................................................................................................................... 23

  Step 1: Creating a Project with the Vivado New Project Wizard ............................................................................ 23

  Step 2: Synthesize, Implement, and Generate Bitstream ................................................................................. 25

Lab 3: Using a VIO Core for Debugging a Design in Vivado® .................................................................................. 27

  Introduction .......................................................................................................................................................... 27

  Step 1: Creating a Project with the Vivado New Project Wizard ............................................................................ 28

  Step 2: Synthesize, Implement, and Generate Bitstream ................................................................................. 32

Lab 4: Using the Synplify Pro Synthesis Tool and Vivado® for Debugging a Design .......................................... 34

  Introduction .......................................................................................................................................................... 34

  Step 1: Create a Synplify Pro Project ............................................................................................................... 34

  Step 2: Synthesize the Synplify Project ........................................................................................................... 42

  Step 3: Create EDIF Netlists for the Black Box Created in Synplify Pro ....................................................... 43

  Step 4: Create a Post Synthesis Project in Vivado IDE ..................................................................................... 44
Step 5: Add (more) Debug Nets to the Project ....................................................................................... 46
Step 6: Implementing the Design and Generating the Bitstream .................................................... 48
Using the Vivado® Logic Analyzer to Debug the Hardware................................................................. 49
Introduction .......................................................................................................................................................... 49
Step 1: Verifying Operation of the Sine Wave Generator ................................................................. 49
Step 2: Debugging the Sine Wave Sequencer State Machine (Optional)...................................... 57
Verifying the VIO Core Activity (Only applicable to Lab 3)................................................................. 60
Lab 5: Using Vivado® Serial Analyzer to debug Serial Links............................................................ 67
Introduction .......................................................................................................................................................... 67
Design Description ............................................................................................................................................. 67
Step 1: Creating, Customizing, and Generating an IBERT Design .................................................... 68
Step 2: Adding an IBERT core to the Vivado Project ............................................................................. 69
Step 3: Synthesize, Implement and Generate Bitstream for the IBERT design ................................ 75
Step 4: Interact with the IBERT core using Serial I/O Analyzer .......................................................... 77
Lab 6: Using Vivado® ILA core to debug JTAG-AXI Transactions....................................................... 92
Introduction .......................................................................................................................................................... 92
Design Description ............................................................................................................................................. 93
Step 1: Opening the JTAG to AXI Master IP Example Design and Configuring the Debug Connections ......................................................................................................................... 93
Step 2: Program the KC705 Board and Interact with the JTAG to AXI Master Core ................ 111
Step 3: Using ILA 3.0 Advanced Trigger Feature to Trigger on an AXI Read Transaction..... 118
Debugging in Vivado®

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 choose 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 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 3.0, 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.

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 and the Integrated Logic Analyzer (ILA) core.
- Understand how to create an RTL project, probe your design, insert an ILA 3.0 core, and implement the design in the Vivado Integrated Design Environment.
- 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 2013.3

**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](image-url)
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 pulses from the two push button switches. Provides sine wave selection and indicator circuits, sequencing between 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.

Lab5 includes:

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

Board Support and Pinout Information

<table>
<thead>
<tr>
<th>Pinout Locations</th>
<th>Function</th>
<th>Pinout Locations</th>
</tr>
</thead>
<tbody>
<tr>
<td>CLK_N</td>
<td>AD11</td>
<td>Clock</td>
</tr>
<tr>
<td>CLK_P</td>
<td>AD12</td>
<td>Clock</td>
</tr>
<tr>
<td>GPIO_BUTTONS[0]</td>
<td>AA12</td>
<td>Reset</td>
</tr>
<tr>
<td>GPIO_BUTTONS[1]</td>
<td>AG5</td>
<td>Sine Wave Sequencer</td>
</tr>
<tr>
<td>GPIO_SWITCH</td>
<td>Y28</td>
<td>De-bounce Circuit Selector</td>
</tr>
<tr>
<td>LEDS_n[0]</td>
<td>AB8</td>
<td>Sine Wave Selection[0]</td>
</tr>
</tbody>
</table>
## Design Files

1. In your C: drive, create a folder called /Vivado_Debug.

2. Find the tutorial source files at the following location:
   

   **CAUTION!** The tutorial and design files may be updated or modified in 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 five 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). The files required for this lab are as shown below:

   - debounce.vhd
   - fsm.vhd
   - rt.tcl
   - sinegen.vhd
   - sinegen_demo.vhd
   - sine_high.xci
   - sine_low.xci
   - 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. The files required for this lab are as shown below:

- debounce.vhd
- fsm.vhd
- rt.tcl
- sinegen.vhd
- sinegen_demo_inst.vhd
- ila_0.xci
- sine_high.xci
- sine_low.xci
- 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 will walk you over the steps of instantiating and customizing a VIO core that you will hook to the I/Os of the design. The files needed for this lab are as follows:

- debounce.vhd
- fsm.vhd
- sinegen.vhd
- sinegen_demo_inst_vio.vhd
- sine_high.xci
- sine_low.xci
- sine_mid.xci
- 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. The following files are needed for this lab:

- dds_compiler_v6_0_viv.edn
- dds_compiler_v6_0_viv_parameterized1.edn
- dds_compiler_v6_0_viv_parameterized3.edn
- debounce.vhd
- fsm.vhd
- sine_high.xci
• sine_low.xci
• sine_mid.xci
• sinegen.edn
• sinegen_synplify.vhd
• synplify_1.sdc
• sinegen_demo_kc705.xdc

Lab 5: Debug high-speed serial I/O links using the Vivado Serial I/O Analyzer. This lab uses the Vivado IP example design.

Lab 6: Using Vivado ILA core to debug JTAG-to-AXI transactions. This lab uses the Vivado IP example design.

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.
Lab 1: Using the Netlist Insertion Method for Debugging a Design in Vivado®

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 screen, click Create New Project to start the New Project wizard. Click Next.
3. In the Project Name screen, name the new project proj_netlist and provide the project location (C:/Vivado_Debug). Ensure that Create Project Subdirectory is checked and click Next.
4. In the Project Type screen, specify the Type of Project to create as RTL Project and click Next.
5. In the Add Sources screen:
   a. Set Target Language to VHDL.
   b. Click the Add Files button.
   c. In the Add Source Files dialog box, navigate to the /src/Lab 1 directory.
   d. Select all VHD source files, and click OK.
   e. Verify that the files are added, and Copy Sources into Project is checked. Click Next.
6. In the Add Existing IP (optional) dialog box:
   a. Click the Add Files button.
b. In the Add Configurable IP dialog box, navigate to the /src/lab1 directory.

c. Select all XCI source files, and click OK.

d. Verify that the files are added and Copy Sources into Project is checked. Click Next.

7. In the Add Constraints (optional) dialog box, the provided XDC file, 
sinegen_demo_kc705.xdc should automatically appear in the main window. Click Next.

8. 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, and click Next.

9. Review the New Project Summary screen. Verify that the data appears as expected, per 
the steps above, click Finish.

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

---

Step 2: Synthesizing the Design

1. In the Project Manager, click Project Settings as shown Figure 2.

![Figure 2: Configuring the Project Settings](image)

2. In the Project Settings dialog box select Synthesis from the left pane and change 
flatten_hierachy option to none as shown in Figure 3. Click OK.
**Step 2: Synthesizing the Design**

**IMPORTANT:** The reason for changing this setting to **none** is to prevent the synthesis tool from performing any boundary optimizations for this tutorial.

3. In the Vivado Flow Navigator, expand the **Synthesis** drop-down list, and click **Run Synthesis**.
   
   **Note:** When synthesis runs, a progress indicator appears, showing that synthesis is occurring. This could take a few minutes.

4. In the **Synthesis Completed** dialog box, click **Cancel** as shown in **Figure 4**. You will implement the design later.
Step 3: Probing and Adding Debug IP

To add a Vivado ILA 2.0 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 examples of how to add debug nets using the Vivado IDE:

- Add mark_debug attribute to the target XDC file
  ```
  set_property mark_debug true [get_nets sine*]
  ```

**IMPORTANT:** Use these attributes in synthesized designs only. Do not use them with pre-synthesis or elaborated design netlists.

- Add mark_debug attribute to HDL files

  **VHDL**
  ```
  attribute mark_debug : string;
  attribute keep : 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;
  ```
Right-click and select **Mark Debug** or **Unmark Debug** on Synthesis netlist.

Use a Tcl prompt to set the mark_debug attribute. For example,

```tcl
set mark_debug true [get_nets sine*]
```

This applies the mark_debug on the current, open netlist.

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

1. In the Flow Navigator under the **Synthesis** drop-down list, click **Open Synthesized Design** as shown in Figure 5.

   ![Figure 5: Open Synthesized Design](image)

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

2. The Debug window opens. Click the **Debug** window if it is not already selected.

3. Expand **Unassigned Debug Nets** folder. Figure 6 shows those debug nets that were tagged in `sinegen_demo.vhd` with mark_debug attributes, as shown in Figure 7.

   ![Figure 6: VHDL Example Using MARK_DEBUG Attributes](image)
4. Select the **Netlist** tab and expand **Nets**. Select the following nets for debugging as shown in Figure 8.

- **GPIO_BUTTONS_IBUF[0]** and **GPIO_BUTTONS_IBUF[1]** - Nets folder under the top-level hierarchy
- **sine(20)** - Nets folder under the **U_SINEGEN** hierarchy
- **sel(2)** - Nets folder under the **U_SINEGEN** hierarchy

**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 **Figure 9**.

![Figure 9: Adding Nets from the Netlist Tab](image)

6. In the **Confirm Debug Net(s)** dialog box click **OK**.

**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 **Figure 10** and **Figure 11**.
Running the Set Up Debug Wizard

7. From the Debug window or Tools menu, select Set Up Debug. The Set Up Debug wizard opens.
8. The **Set up Debug** dialog box opens. Click **Next**.

9. In the **Specify Nets to Debug** dialog box, ensure that all the nets have been added for debug and click **Next**.
10. In the **Trigger and Capture Modes** dialog box, check both the **Enable advanced trigger mode** and **Enable basic capture mode**. Click **Next**.

11. In the **Setup Debug Summary** dialog box, make sure that all the information is correct and as expected. Click **Finish**.
Step 4: Implementing and Generating Bitstream

1. Click on **Generate Bitstream** from the **Program and Debug** drop-down list in the **Flow Navigator**.

2. In the **Save Project** dialog box click on **Save**. This applies the mark debug attributes on the newly marked nets. You can see those constraints can by inspecting the `sinegen_demo_kc705.xdc` file.

3. When the **No Implementation Results Available** dialog box pops up, click **Yes**.

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. In the **Implementation is Out-of-date** dialog box, click **Yes**.

7. In the **Flow Navigator**, under **Implementation**, expand the **Implemented Design** drop-down list and select **Report Timing Summary**.

8. In the **Report Timing Summary** dialog box, click **OK**.

9. Ensure that all the specified timing constraints are met.
10. Proceed to Using the Vivado® Logic Analyzer to Debug the Hardware on page 49 to complete the rest of the steps for debugging the design.
Lab 2: Using the HDL Instantiation Method for Debugging a Design in Vivado®

Introduction

The HDL Instantiation method is one of the two methods supported in Vivado® Debug Probing. For this flow, you will generate an ILA 3.0 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 Getting Started screen, click Create New Project to start the New Project wizard. Click Next.
3. In the Project Name screen, name the new project proj_hdl and provide the project location (C:/Vivado_Debug). Ensure that Create Project Subdirectory is checked and click Next.
4. In the Project Type screen, specify the Type of Project to create as RTL Project and click Next.
5. In the Add Sources screen:
   a. Set Target Language to VHDL.
   b. Click the Add Files button.
   c. In the Add Source Files dialog box, navigate to the /src/Lab2 directory.
   d. Select all VHD source files, and click OK.
   e. Verify that the files are added, and Copy Sources into Project is checked. Click Next.
6. In the Add Existing IP (optional) dialog box:
   a. Click the Add Files button.
   b. In the Add Configurable IP dialog box, navigate to the /src directory.
   c. Select all XCI source files, and click OK.
Step 1: Creating a Project with the Vivado New Project Wizard

d. Verify that the files are added, and **Copy Sources into Project** is checked. Click **Next**.

7. In the **Add Constraints (optional)** dialog box, the provided XDC file, `sinegen_demo_kc705.xdc` should automatically appear in the main window. Click **Next**.

8. 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**.

9. Review the **New Project Summary** screen. Verify that the data appears as expected, per the steps above, click **Finish**.

---

**CAUTION!** It might take a moment for the project to initialize.

---

10. In the **Sources** window in Vivado, expand **sinegen_demo_inst** to see the source files for this lab. Note that **ila_0** core has been added to the project.

    ![ILA instantiation in HDL](image.png)

Figure 19: ILA instantiation in HDL

Double-click the `sinegen_demo_inst.vhd` file to open it and inspect the instantiation and port mapping of the ILA core in the HDL code. Note that attributes have been placed in the source file to preserve the net names.

    -- Attributes for keeping the net names preserved
    attribute keep : string;
    attribute keep of GPIO_BUTTONS_db : signal is "true";
    attribute keep of GPIO_BUTTONS_dly : signal is "true";
    attribute keep of GPIOBUTTONS_re : signal is "true";
    attribute keep of sine : signal is "true";
    attribute keep of sine : signal is "true";

    ![Using “keep” attribute to preserve net names](image.png)

Figure 20: Using “keep” attribute to preserve net names
Step 2: Synthesize Implement and Generate Bitstream

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

2. The No Implementation Results Available dialog box appears. Click Yes.

3. After bitstream generation completes, the Bitstream Generation Completed dialog box appears. Open Implemented Design is selected by default. Click OK.

Step 2: Synthesize Implement and Generate Bitstream

5. In the **Report Timing Summary** dialog box, click **OK**. Make sure that all timing constraints are met.

6. Proceed to Using the Vivado® Logic Analyzer to Debug the Hardware on page 49 to complete the rest of this lab.
**Lab 3: Using a VIO Core for Debugging a Design in Vivado®**

**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 3.0 core.

![Figure 25: VIO Block Diagram](image)

This lab will walk you through the steps of instantiating and configuring a VIO core and connecting the I/Os of the design to it. 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 **Getting Started** screen, click **Create New Project** to start the New Project wizard. Click **Next**.
3. In the **Project Name** screen, name the new project *proj_hdl_vio* and provide the project location (*C:/Vivado_Debug*). Ensure that **Create project subdirectory** is checked and click **Next**.
4. In the **Project Type** screen, specify the **Type of Project** to create as **RTL Project** and click **Next**.
5. In the **Add Sources** screen:
   a. Set **Target Language** to **VHDL**.
   b. Click the **Add Files** button.
   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 checked. Click **Next**.
6. In the **Add Existing IP (optional)** dialog box:
   a. Click the **Add Files** button.
   b. In the **Add Configurable IP** dialog box, navigate to the */src/Lab3* directory.
   c. Select all XCI source files, and click **OK**.
   d. Verify that the files are added and **Copy sources into project** is checked. Click **Next**.
7. In the **Add Constraints (optional)** dialog box, the provided XDC file, *sinegen_demo_kc705.xdc*, should automatically appear in the main window. Click **Next**.
8. 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**, and click **Next**.
9. Review the **New Project Summary** screen. Verify that the data appears as expected, per the steps above, click **Finish**.
   
   **Note:** It might take a moment for the project to initialize.
10. 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.
Step 1: Creating a Project with the Vivado New Project Wizard

From the Flow Navigator, click IP Catalog, expand Debug & Verification, then expand Debug, and double-click on VIO 3.0. The Customize IP dialog box opens.

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 check the Enable Input Probe Activity Detectors checkbox, if it is not checked already.

On the PROBE_IN Ports tab, set Probe Width to 4 bits wide.
Step 1: Creating a Project with the Vivado New Project Wizard

Figure 28: Configure the PROBE_IN Ports of the VIO core

14. On the PROBE_OUT Ports, set Probe Width to 2 bits wide with an initial value of 0 in hex format.

Figure 29: Configure the PROBE_OUT Ports of the VIO core

15. Click OK to generate the IP. The Generate Output Products dialog box will pop-up. Click Generate.
Output product generation should take less than a minute. At this point, you have finished customizing the VIO 3.0. This core has already been instantiated in the top level design as shown in Figure 31:

```
-- VIO
U_VIO : vio_0
  port map
  (  
    CLK => clk,
    PROBE_INO => (DONT_EAT & GPIO_BUTTONS_re(1) & sine5el),
    PROBE_OUTO(0) => push_button_react,
    PROBE_OUTO(1) => push_button_vio
  );
```

At this point, the Sources window should look as shown in Figure 32.

16. 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. Also, note that attributes have been placed in the source file to preserve the net names.
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.

2. The **No Implementation Results Available** dialog box pops up. Click **Yes**.

3. After bitstream generation completes, the **Bitstream Generation Completed** dialog box pops up. **Open Implemented Design** is selected by default. Click **OK**.

4. From the Implementation drop-down list in **Flow Navigator**, expand **Implemented Design** and select **Report Timing Summary**.

---

**Figure 33**: Using “keep” attribute to preserve net names

```vhdl
-- Attributes for keeping the net names preserved
attribute keep : string;
attribute keep of GPIO_BUTTONS_ab : signal is "true";
attribute keep of GPIO_BUTTONS_dly : signal is "true";
attribute keep of GPIO_BUTTONS_re : signal is "true";
attribute keep of sineSel : signal is "true";
attribute keep of sine : signal is "true";
attribute keep of push_button_vio : signal is "true";
attribute keep of push_button_reset : signal is "true";
```

**Figure 34**: Hook signals that need to be debugged in the ILA

```vhdl
-- ILA

U_ILA : ila_0
port map
(
  CLK => clk,
  PROBE0 => sineSel,
  PROBE1 => sine,
  PROBE2 => (push_button_reset & push_button_vio),
  PROBE3 => GPIO_BUTTONS_re,
  PROBE4 => GPIO_BUTTONS_dly
);  
```

**Figure 35**: Report timing summary for Lab 3
5. In the **Report Timing Summary** dialog box, click **OK**. Make sure that all timing constraints have been met.

![Figure 36: Review Timing Summary](image_url)

6. Proceed to Using the Vivado® Logic Analyzer to Debug the Hardware on page 49. Skip forward to **Verifying the VIO Core Activity (Only applicable to Lab 3)** to complete the rest of this lab.
Lab 4: Using the Synplify Pro Synthesis Tool and Vivado® for Debugging a Design

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 (Version 2013-3 SP1).

Step 1: Create a Synplify Pro Project

1. Launch Synplify Pro and select File > New. Set File Type to Project File (Project) as highlighted in Figure 37. In the New File Name box, enter synplify_1. Click OK.
Step 1: Create a Synplify Pro Project

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

3. In the left panel of the Synplify Pro window, click **Add File** as shown in **Figure 39**.
4. 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

Click Add.
Figure 40: Adding VHDL Source Files to the Synplify Pro Project
5. In the same dialog box set **Files of type** to **Constraints File**. This shows the `synplify_1.sdc` file. Select the file and click **Add** as shown in **Figure 41**.
6. In the same dialog box set **Files of type** to **Compiler Directives File**. This shows the `synplify_1.cdc` file. Select the file and click **Add** as shown in Figure 42. Click **OK**.

![Figure 42: Adding CDC Constraints File to the Synplify Pro Project](image-url)
7. Now, you need to set the implementation options. Click **Implementation Options** in the Synplify Pro window as shown in Figure 43.

![Figure 43: Opening Implementation Options in Synplify Pro](image)

8. This brings up the **Implementation Options** dialog box as shown in Figure 44. 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 44: Specifying Implementation Options in Synplify Pro](image)
Mark Nets for Debug in the Constraints Files

9. 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 45: Specifying attributes to preserve net names in Synplify

10. You also can specify the `mark_debug` attributes in the source HDL files to mark the signals for debug, as shown in the snippet code 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 GPIOBUTTONS_dly : signal is "true";
attribute mark_debug of GPIO_BUTTONS_re : signal is "true";
```

Figure 46: Add mark_debug attribute in HDL file

11. The `synplify_1.sdc` file contains various kinds of constraints such as pin location, I/O standard, and clock definition. The `synplify_1.cdc` 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 Figure 47.

```
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_no_prune) {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 47: Synplify Pro Constraints in CDC 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 project settings in place, click the Run button in the left panel of the Synplify Pro window to start synthesizing the design.

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. The mark_debug attributes can be seen in this netlist.

5. Click File > Save All to save the project, then click File > Exit.
Step 3: Create EDIF Netlists 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 New 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 checked. Click Next.
6. Under Add Existing IP, 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 Add Constraints, the .sdc files are automatically added to the project. These files are not needed for this step. Remove them from this project by clicking the Remove Selected File button on the right of the dialog box. Click Next.
8. Under Default Part, select Boards and then select the Kintex-7 KC705 Evaluation Platform and correct version for your hardware. Click Next.
9. Under New Project Summary, ensure that all the settings are correct and click Finish.
10. Once the project has been created, in Vivado Flow Navigator, under the Project Manager folder, click Project Settings. In the pop-up dialog box, in the left panel, click Synthesis. From the pull down menu on the right panel, set –flatten_hierarchy to none. Click OK.
12. When synthesis completes the Synthesis Completed dialog box appears. Select Open Synthesized Design and click OK.
13. Now you need to write the netlist file for all the components used in the sinegen block. The four netlist files used in this tutorial are already provided as a part of the source files. However, you can overwrite them by using your own netlist files. To do this use the following Tcl command in the Tcl console of Vivado IDE.

   write_edif -force ../Vivado_Debug/src/Lab4/sinegen.edn

   Ensure that the path specified to the src folder is correct. At this point, you should see four .edn files in the Vivado_Debug/src folder as shown below:

   • dds_compiler_v6_0_viv.edn
   • dds_compiler_v6_0_viv_parameterized1.edn
   • dds_compiler_v6_0_viv_parameterized3.edn
Step 4: Create a Post Synthesis Project in Vivado IDE

1. Launch Vivado IDE.
2. Click Create New 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 four netlist files created in the previous section. Click Add Files again, navigate to the Vivado_Debug/src/Lab4 folder and select the following files:
   - sinegen.edn
   - dds_compiler_v6_0_viv.edn
   - dds_compiler_v6_0_viv_parameterized1.edn
   - dds_compiler_v6_0_viv_parameterized3.edn
   Click OK in the Add Source Files dialog box. In the Add Netlist Sources dialog box ensure that Copy Sources into Project is checked. Click Next.
7. Under Add Constraints, a .sdc file, should be automatically populated. Remove this file by selecting it and clicking the Remove Selected File button on the right of the dialog box. 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 checked. Click Next.
8. Under Default Part, select Boards and then select the 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, select sinegen_demo.edf and select Specify Top Module.
Step 4: Create a Post Synthesis Project in Vivado IDE

Figure 49: Specifying the top-level module

11. In the Specify Top Module dialog box, click on the browse button.

![Specify Top Module dialog box](image)

Figure 50: Browse to the top module

12. In the Select Top Module dialog box, select `sinegen_demo` and then click OK.

![Select Top Module dialog box](image)
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)
   - sineSel (2)

   After selecting all the nets mentioned above click **Mark Debug**.
Step 5: Add (more) Debug Nets to the Project

3. In the Confirm Debug Net(s) dialog box, click OK.
4. You should be able to see all the nets that are marked for debug as shown in Figure 54.

Running the Set up Debug Wizard

5. 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.

6. Click through the wizard to create Vivado logic analyzer debug cores, keeping the default settings.
Step 6: Implementing the Design and Generating the Bitstream

**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 Using the Vivado Logic Analyzer to Debug the Hardware on page 49 to complete the rest of this lab.
Using the Vivado® Logic Analyzer to Debug the Hardware

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.

Setting Up

Connecting to the target board remotely

If you plan to connect remotely, you will need to make sure you have KC705 hardware plugged into a machine and you are running a cse_server application on that machine. If you plan to connect locally, skip steps 1-3 below.

1. Connect the Digilent USB JTAG cable of your KC705 board to a Windows machine’s USB port.
2. Ensure that the board is plugged in and powered on.
3. Assuming you are connecting your KC705 board to a 64-bit Windows machine and you will be running the cse_server from the network instead of your local drive, open a cmd prompt and type the following:

   `<Xilinx_Install>\Vivado\2013.3\bin\ vcse_server`
Step 1: Verifying Operation of the Sine Wave Generator

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

Connecting to the target board locally

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

1. Connect the Digilent USB JTAG cable of your KC705 board to a Windows machine’s USB port.
2. Ensure that the board is plugged in and powered on.
3. 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, from the Program and Debug drop-down list, select Open Hardware Manager.

![Open Hardware Manager](image)

Figure 56: Open Hardware Manager

2. Click on the Open a new hardware target link in the Hardware Session view.

![Connect to a Hardware Target](image)

Figure 57: Connect to a Hardware Target

3. The Hardware Manager window opens. Click Open a new hardware target. The Open New Hardware Target dialog box opens.
4. Type the name of the server (for e.g. `localhost:60001`) in the text field and click Next.
Step 1: Verifying Operation of the Sine Wave Generator

Figure 58: CSE Server Name

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

5. If there is more than one target connected to the cse_server you will see multiple entries in the Select Hardware Target dialog box. In this tutorial, there is only one target as shown in Figure 59. Click Next.

Figure 59: Select Hardware Target
6. Leave these settings at their default values as shown in Figure 60. Click Next.

Figure 60: CSE Hardware Target Parameter Settings

7. In the Open Hardware Target Summary page, click Finish as shown in Figure 61.

Figure 61: Open Hardware Summary

8. Wait for the connection to the hardware to complete. The dialog in Figure 62 appears while hardware is connecting.

Figure 62: Open Hardware Target

Once the connection to the hardware target is made, the dialog shown in Figure 63 appears.
Step 1: Verifying Operation of the Sine Wave Generator

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

9. Next, program the XC7K325T device using the .bit bitstream file that was created previously by right-clicking on the XC7K325T device and selecting Program Device as shown in Figure 64.

10. In the Program Device dialog box verify that the .bit file is correct for the lab that you are working on and click the Program button to program the device as shown in Figure 65.

CAUTION! The file paths of the bitstream to be programmed will be different for different labs. Ensure that the relative paths are correct.
Step 1: Verifying Operation of the Sine Wave Generator

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

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

![Figure 66: ILA Core Detection](image)

12. The **Integrated Logic Analyzer** window opens.

![Figure 67: The Vivado Integrated Logic Analyzer window](image)

**Verifying Sine Wave Activity**

13. Click the **Run Trigger Immediate** button to trigger and capture data immediately as shown in **Figure 68**.

![Figure 68: Run Trigger Immediate Button](image)

14. In the **Waveform** window, verify that there is activity on the 20-bit sine signal as shown in **Figure 69**.
Step 1: Verifying Operation of the Sine Wave Generator

Displaying the Sine Wave

15. Right-click \texttt{U\_SINEGEN/sine[19:0]} signals, and select \textbf{Waveform Style > Analog} as shown in \textbf{Figure 70}.

\textbf{TIP:} Notice that 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.

16. Right-click on \texttt{U\_SINEGEN/sine[19:0]} signals, and select \textbf{Radix > Signed Decimal}. You should now be able to see the high frequency sine wave as shown in \textbf{Figure 71} instead of the square wave.
Correcting Display of the Sine Wave

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

17. Cycle the sine wave sequential circuit by pressing the GPIO_SW_E push button as shown in Figure 72.

![Figure 72: Sine Wave Sequencer Push Button](image)

18. Click **Run Trigger Immediately** again to see the new sine selected sine wave. You should see the mid frequency as shown in Figure 73. Notice that the sel signal also changed from 0 to 1 as expected.

![Figure 73: Output Sine Wave Displayed in Analog Format – Mid Frequency](image)

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

![Figure 74: Output Sine Wave Displayed in Analog Format – Low Frequency](image)
Step 2: Debugging the Sine Wave Sequencer State Machine (Optional)

As you were correcting 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, you will 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. Figure 76 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 (shown in Figure 1). The output signal Y connects to the top-level signal, sineSel, which selects the sine wave.

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 Figure 73, Figure 74, and Figure 75.
Step 2: Debugging the Sine Wave Sequencer State Machine (Optional)

Viewing the State Machine Glitch

You cannot troubleshoot the issue you 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_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 menu.
2. In the ILA Properties window set the following:
   a. Trigger Mode to BASIC
   b. Capture Mode to BASIC
   c. Data Depth to 1024
   d. Trigger position to 512
   e. Drag and drop the GPIO_BUTTONS_IBUF_1 [1:1] bit from the **Debug Probes** window, in the Basic Trigger Setup box. Change the Compare Value field to R by clicking in the Compare Value column and typing the value R in the Value field.
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 we are using bit 1 of this bus to set up our trigger condition. For example in case of a two-bit bus, you will set up the Value Field in the Compare Value dialog box to RX.

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

3. Select the hw_ila_data_1.wcfg tab or the waveform tab. Source the rt.tcl file in the Tcl command prompt. This Tcl command performs the following tasks:
   - 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_IBUF signal (this could take 10 or more tries). This is a visualization of the glitch that is occurring on the input. An example of the glitch is shown in Figure 78 and Figure 79.

CAUTION! You may have to repeat steps 3 and 4 repeatedly to see the glitch. Once the glitch can be seen, you may observe signal glitches are not at exactly the same location as shown in the figure below.

Figure 78: GPIO_BUTTONS_BUF1 Signal Glitch
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.

5. Enable the de-bouncer circuit by setting DIP switch position on the KC705 board (labeled De-bounce Enable in Figure 1) to the ON or UP position.

6. Source the rt.tcl file again and:
   - Ensure that you no longer see multiple transitions on the GPIOBUTTON_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 on Open Hardware Manager.

2. The Hardware Manager window opens. Click on Open a new hardware target.
Verifying the VIO Core Activity (Only applicable to Lab 3)

3. The **Open New Hardware Target** dialog box opens. Click **Next**.

4. In the **Server Name** field type in the name and port of the remote server that you want to connect to (for e.g. `localhost:60001`).

5. Ensure that you are connected to the right target by selecting the target from the Hardware Targets pane. If there is only one target then that will be selected by default. Click **Next**.

6. On the **Set Hardware Target Properties** page, click **Next**.

7. On 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**.

9. In the Program Device dialog box, ensure that the bit file to be programmed is correct and click **OK**.
10. Once the FPGA has been programmed, you will see the VIO and the ILA core in the **Hardware** window.

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

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

16. If the Debug Probes window is not open already, open it by selecting **Window => Debug Probes** from the menu.

The **Debug Probes** window shows the probes available in the ILA as well as the VIO cores.
Also note that separate tabs are available for the ILA as the VIO cores as shown below.

![Tabs for ILA and VIO](image)

Figure 89: Tabs for ILA and VIO

17. In the **Debug Probes** window, from the hw_vio_1, select all the probes, then drag-and-drop these probes in the VIO core tab. Note the initial values of all the probes.

![Drag-and-drop VIO probes from the Debug Probes window into VIO core tab](image)

Figure 90: Drag-and-drop VIO probes from the Debug Probes window into VIO core tab

Note that PROBE_IN [3] is assigned to the signal DONT_EAT, PROBE_IN [2] is assigned to GPIO_BUTTONS_re and PROBE_IN [1:0] are assigned to sineSel.

18. Set the “push_button_reset” output probe by right-clicking on **push_button_reset** and select **Toggle Button** type. 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.
Verifying the VIO Core Activity (Only applicable to Lab 3)

Figure 91: Toggle the push_button_reset signal

The **Value** field for push_button_reset is highlighted. Click in the **Value** field to change its value to **1**.

Figure 92: Toggle the value of push_button_reset

19. Follow the step above to change the push_button_vio to Toggle button as well.
20. The “sineSel” signal was connected to the PROBE_IN0 [1:0]. Set these two bits of the input probe by right-clicking on PROBE_IN0[0] and PROBE_IN0[1] and selecting **LED**.

Figure 93: Change sineSel to LED
21. 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.

![Select LED Colors dialog box](image)

Figure 94: Pick the Low Value and High Value color of the LEDs

22. When finished, your VIO Probes window in the Hardware Manager should look similar to Figure 95.

![VIO Probes window](image)

Figure 95: Input and Output VIO Signals Displayed

23. To cycle through each different sine wave output frequency using the virtual “push_button_vio” from the VIO core, follow 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...

   b. Select hw_ila_1 in the Hardware window and click Run Trigger to capture and display the selected sine wave signal from the previous step.
Lab 5: Using Vivado® Serial Analyzer to debug Serial Links

Introduction

The Serial I/O analyzer is used to interact with IBERT Version 3.0 or later 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 will:

- 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.

Figure 96 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 tool.

- **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 96: IBERT Design Flow](image)

**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 **Getting Started** screen, click **Create New Project** to start the New Project wizard. Click **Next**.
3. In the **Project Name** screen, name the new project **ibert_tutorial** and provide the project location (**C:/ibert_tutorial**). Ensure that **Create Project Subdirectory** is checked and click **Next**.
4. In the **Project Type** screen, specify the **Type of Project** to create as **RTL Project** and click **Next**.

5. In the **Add Sources** screen click **Next**.

6. In the **Add Existing IP** screen, click **Next**.

7. In the **Add Constraints** screen, click **Next**.

8. In the **Default Part** dialog box, select **Boards** and then select **Kintex-7 KC705 Evaluation Platform**. Click **Next**.

9. Review the **New Project Summary** screen. 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. Click **IP Catalog** in the **Flow Navigator**. The IP Catalog opens up.

   ![Opening the Vivado IP Catalog](image)

   **Figure 97: Opening the Vivado IP Catalog**

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

   ![Instantiating the IBERT IP from the Vivado IP Catalog](image)

   **Figure 98: Instantiating the IBERT IP from the Vivado IP Catalog**

3. Select and double-click the **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:
Step 2: Adding an IBERT core to the Vivado Project

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** field is set to 1.

d. Change the **LineRate (Gbps)** to 8, **DataWidth** to 40, and **Refclk (MHz)** to 125. Also ensure that the **Quad Count** is set to 2 and **Quad PLL** box is checked.

![Setting Protocol Definition on the IBERT Core](image)

Figure 99: Setting the Protocol Definition on the IBERT Core

5. Under the **Protocol Selection** tab, make the following selections:

a. In the **Protocol Selected** field, from the pull-down menu select **Custom 1 / 8 Gbps** for GTX Location QUAD_117. This should automatically populate the **Refclk Selection** field to **MGTREFCLK0 117** and the **TXUSRCLK Source** to **Channel 0**.

b. In the **Protocol Selected** field, from the pull-down menu select **Custom 1 / 8 Gbps** for GTX Location QUAD_118. Change the **Refclk Selection** for QUAD_118 to **MGTREFCLK0 117** and **TXUSRCLK Source** to **Channel 0** as well.
Step 2: Adding an IBERT core to the Vivado Project

Figure 100: Setting the Protocol Selection on the IBERT Core

6. Click on the **Clock Settings** tab and make the following changes:
   a. Leave the **Source** field to its default value of **External**.
   b. Change the **I/O Standard** field to **DIFF SSTL15**.
   c. Change the **P Package Pin** to **AD12** by typing in the box.
   d. Change the **N Package Pin** to **AD11** by typing in the box.
   e. Leave the **Frequency(MHz)** to its default value of **200.00**.
Step 2: Adding an IBERT core to the Vivado Project

7. Click on the **Summary** tab and make sure that it looks as follows and then click **OK**.

![Figure 101: Specifying clock settings for the IBERT Core](image1)

![Figure 102: IBERT Core Summary Page](image2)
8. The **Generate Output Products** dialog box will pop-up. Click **Generate**.

![Figure 103: Generate Output Products](image-url)
9. In the **Project Manager** window right-click the IP and select **Open IP Example Design**.

![Figure 104: Open Example IP Design Menu Item](image.png)
Step 3: Synthesize, Implement and Generate Bitstream for the IBERT design

10. In the **Open IP Example Design** dialog box, ensure that the **Overwrite existing example project** is checked and click **OK**.

![Open IP Example Design dialog](image)

**Figure 105 Open IP Example Design dialog**

---

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

1. Click on **Generate Bitstream** in the Flow Navigator. The **No Implementation Results Available** dialog box pops up. Click **Yes**.

![No Implementation Results Available dialog box](image)

**Figure 106: No Implementation Results Available dialog box**

2. When the bitstream generation is complete, the **Bitstream Generation Completed** dialog box pops up. Select **Open Hardware Manager** and click **OK**.
Step 3: Synthesize, Implement and Generate Bitstream for the IBERT design

Figure 107: Bitstream Generation Completed dialog box

3. The **Hardware Manager** window appears as shown.

Figure 108: 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 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.

1. Click Open a new hardware target. The Open New Hardware Target dialog box opens. Click Next.

2. In the Vivado VCSE Server Name field, enter the default value, localhost:60001. Click Next.

Figure 109: Open a new hardware target

Figure 110: Open New Hardware Target dialog box
3. In the **Select Hardware Target** dialog box, click **Next**. There is only one target board in this case to connect to, so the default is selected.

4. In the **Set Hardware Target Properties** dialog box, leave the **Frequency** fields at its default values and click **Next**.
5. In the **Open Hardware Target Summary**, review the options that have been selected and click **Finish**.

6. The **Hardware** window in Vivado IDE should show the status of the target FPGA device on the KC705 board.
Step 4: Interact with the IBERT core using Serial I/O Analyzer

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

![Figure 116: Program target device](image)

8. The Program Device dialog box opens. Make sure that the correct bitfile is selected and click OK.

![Figure 117: Program Device dialog box](image)

9. 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 4 GTX transceivers. These components of the IBERT were detected while scanning the FPGA after downloading the bitstream. If you do not see the QUADS then select the XC7K325 device, right-click and select Refresh Device.
10. Next, we will 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, we will simply link the TX and RX of the same channel. To create a link click **Create Links** in the Links tab.

The **Create Links** dialog box opens. Make sure the first transceiver pairs (MGT_X0Y8/TX and MGT_X0Y8/RX) are selected.
Step 4: Interact with the IBERT core using Serial I/O Analyzer

Figure 120: Selecting the transceiver pairs for creating new links

Then click on the **Add a new link** (+) button. In the **Link group description** field, type **Link Group SMA**. Click **OK**.
Step 4: Interact with the IBERT core using Serial I/O Analyzer

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

Click on 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 (add new link) repeatedly, until all the links have been added to the new link group called Link Group Internal Loopback. Click OK.
Step 4: Interact with the IBERT core using Serial I/O Analyzer

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

![Create Link dialog box to create the second link group](image)

**Figure 123: Create Link dialog box to create the second link group**

As expected, only Link1 indicates a link with **8.0 Gbps** line rate and the rest of the GTX channels show **No Link** status.

![Links window after all the link groups have been created](image)

**Figure 124: Links window after all the link groups have been created**

For more information about the different columns of the Links windows, refer to [Vivado User Guide: Programming and Debugging, (UG908)](https://www.xilinx.com).
12. We can fix the **No Link** status for all the links by changing the GT properties of these channels to one of the loopback modes. To do this, select the appropriate transceiver in the **Hardware** window and modify the GT property in the **GT Properties** window.

![Figure 125: Select the transceiver whose property needs to be modified](image1)

In the **GT Properties** window, under the **Properties** tab, select **LOOPBACK** pull-down menu and select **Near-End PCS**.

![Figure 126: Selecting appropriate Loopback mode for transceivers](image2)

This changes the Status in the Link window for the transceiver in question as shown.
Step 4: Interact with the IBERT core using Serial I/O Analyzer

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

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

The **Create Scan** dialog box opens. In this dialog box, you can change the various scan properties. In this case, we will leave everything to its default value and click **OK**. For more information on the scan properties see [Vivado User Guide: Programming and Debugging](https://www.xilinx.com), (UG908).
Step 4: Interact with the IBERT core using Serial I/O Analyzer

Figure 129: The Create Scan dialog box

The Scan Plot window opens as shown below.

Figure 130: 2D Scan Plot

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

14. You can also perform a Sweep test on the links that you created earlier. In the Links window, highlight Link 0 under the Link called Link Group SMA, right-click and select Create Sweep.
Step 4: Interact with the IBERT core using Serial I/O Analyzer

15. 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**.
Step 4: Interact with the IBERT core using Serial I/O Analyzer

Figure 132: Create Sweep dialog box

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

Figure 133: 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 134: Displaying Scan Plots

The **Scan Plots** window opens showing the details of the scan performed.
Step 4: Interact with the IBERT core using Serial I/O Analyzer

Figure 135: Analyzing the results of individual scans
Lab 6: Using Vivado® ILA core to debug JTAG-AXI Transactions

Introduction

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

Purpose of This Tutorial

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.

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 3.0 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 3.0 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 Getting Started screen, click Create New Project to start the New Project wizard. Click Next.
3. In the Project Name screen, name the new project jtag_2_axi_tutorial and provide the project location (C:/jtag_2_axi_tutorial). Ensure that Create Project Subdirectory is checked and click Next.
4. In the Project Type screen, specify the Type of Project to create as RTL Project. Click Next.
5. In the Add Sources screen click Next.
6. In the Add Existing IP screen click Next.
7. In the Add Constraints screen click Next.
Step 1: Opening the JTAG to AXI Master IP Example Design and Configuring the Debug Connections

8. In the **Default Part** screen choose **Boards** and choose the **Kintex-7 KC705 Evaluation Platform** board. Click **Next**.

![Figure 136: Choosing the Kintex-7 KC705 Evaluation Platform board](image-url)
Step 1: Opening the JTAG to AXI Master IP Example Design and Configuring the Debug Connections

9. Click **Finish** in the **New Project Summary** screen.

![New Project Summary](image)

**Figure 137: New Project Summary**
Step 1: Opening the JTAG to AXI Master IP Example Design and Configuring the Debug Connections

10. In the leftmost panel of the Flow Navigator click IP Catalog under Project Manager.

![Image of Flow Navigator with IP Catalog selected]

Figure 138: IP Catalog
Step 1: Opening the JTAG to AXI Master IP Example Design and Configuring the Debug Connections

11. In the **Search** field of the **IP Catalog** tab on the right, type in **JTAG to AXI**.

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

![Figure 139: JTAG to AXI Master IP Core](image)
12. Double-click **JTAG to AXI Master** core. The **Customization** dialog of the core appears. Accept the default core settings by clicking **OK**.

![Figure 140: JTAG to AXI Master Customization Dialog](image-url)
Step 1: Opening the JTAG to AXI Master IP Example Design and Configuring the Debug Connections

13. Click **Generate** on the **Generate Output Products** dialog.

![Generate Output Products dialog](image1.png)

**Figure 141: Generate Output Products dialog**

14. The `jtag_axi_0` IP core is inserted into the design.

![Project Manager](image2.png)

**Figure 142: Generated JTAG to AXI Master IP in the design**
15. Right-click **jtag_axi_0** and select **Open IP Example Design**.

![Open IP Example Design menu item](image)

**Figure 143: Open IP Example Design menu item**

16. In the **Open IP Example Design** dialog ensure that **Overwrite existing example project** is checked. Click **OK**.

![Open IP Example Design dialog](image)

**Figure 144: Open IP Example Design dialog**

17. 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.
18. 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 76-112:

```
(* 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 76-112 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 [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_awid;
(* mark_debug *) wire 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;
```

19. Save changes to `example_jtag_axi_0.v` file.

20. In the **Flow Navigator** on the left side of the Vivado window, click **Run Synthesis**.
Step 1: Opening the JTAG to AXI Master IP Example Design and Configuring the Debug Connections

21. Open the synthesized design by selecting **Open Synthesized Design** and clicking **OK**.

![Figure 145: Open Synthesized Design](image)
22. Once the synthesized design opens, do the following:
   a. Select the **Debug** layout in the main toolbar of the Vivado IDE.

![Figure 146: Debug Layout in the Vivado IDE toolbar](image-url)
b. Select the **Debug** window near the bottom of the Vivado IDE.

![Figure 147: Debug window in the Vivado IDE](image)

---

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

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

![Figure 148: Set up Debug Wizard]

23. Once the Set up Debug wizard pops up, click **Next**.
Step 1: Opening the JTAG to AXI Master IP Example Design and Configuring the Debug Connections

24. In the next panel 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 149: Missing Clock Domain dialog](image)

25. In the resulting pop-up, click **Assign All Clock Domains**.
26. In the resulting pop-up, select the `aclk` clock net, then click **OK**.

27. Note that all of the nets now have an assigned clock domain. Click **Next**.
28. In the **Trigger and Capture Modes** screen, ensure that **the Enable advanced trigger mode** and **Enable basic capture mode** are checked. Click **Next**.

29. Click **Finish**
Step 1: Opening the JTAG to AXI Master IP Example Design and Configuring the Debug Connections

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

![Figure 152: ILA core inserted into the design](image)

30. Save the constraints by clicking the Save button.
Step 1: Opening the JTAG to AXI Master IP Example Design and Configuring the Debug Connections

31. The insertion of debug cores and changing of properties on those debug cores adds constraints to your target XDC constraint file. This modification of your target constraints file currently sets your synthesis out of date. You can force the design up to date by clicking more info in the upper-right corner of the Vivado IDE, then clicking force up-to-date.

![Figure 153: Forcing Synthesis up-to-date](image)

32. In the Flow Navigator on the left side of the Vivado IDE, click Generate Bitstream.

33. Click Yes to implement the design.

34. Wait until the Vivado status shows write_bitstream complete.

35. In the Bitstream Generation Completed pop-up dialog, select Open Hardware Manager and click OK.

![Figure 154: Open Hardware Manager](image)
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.
Step 2: Program the KC705 Board and Interact with the JTAG to AXI Master Core

2. The **Hardware Manager** window opens. Click **Open a new hardware target**. The **Open New Hardware Target** dialog box opens.

![Figure 155: Connect to a Hardware Target](image1)

3. Type the name of the server (for e.g. **localhost:60001**) in the text field and click **Next**.

![Figure 156: CSE Server Name](image2)

**Note**: Depending on your connection speed, this may take about 10~15 seconds.
Step 2: Program the KC705 Board and Interact with the JTAG to AXI Master Core

4. If there is more than one target connected to the cse_server you will see multiple entries in the Open Hardware Target dialog box. In this tutorial, there is only one target as shown in Figure 157. Click Next.

![Image of Open Hardware Target dialog box]

**Figure 157: Select Hardware Target**
5. Leave these settings at their default values as shown in Figure 158. Click **Next**.

![Figure 158: CSE Hardware Target Parameter Settings](image)

6. In the **Open Hardware Target Summary** page, click **Finish** as shown in **Figure 159**.

![Figure 159: Open Hardware Summary](image)

7. Wait for the connection to the hardware to complete. The dialog in **Figure 160** appears while hardware is connecting.

![Figure 160: Open Hardware Target](image)

Once the connection to the hardware target is made, the dialog shown in **Figure 161** appears.
Step 2: Program the KC705 Board and Interact with the JTAG to AXI Master Core

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

![Figure 161: Active Target Hardware](image1)

8. Next, program the XC7K325T device using the `.bit` bitstream file that was created previously by right-clicking the **XC7K325T** device and selecting **Program Device** as shown in **Figure 162**.

![Figure 162: Program Active Target Hardware](image2)

9. In the **Program Device** dialog box verify that the `.bit` file is correct for the lab that you are working on and click the **OK** button to program the device.

![Figure 163: Select Bitstream file to download](image3)

**Note:** Wait for the program device operation to complete. This may take few minutes.
Step 2: Program the KC705 Board and Interact with the JTAG to AXI Master Core

10. 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 164: ILA core instances in the Hardware window](image)

11. The JTAG to AXI Master core can only be communicated with by using Tcl commands. 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. Since 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 165: Reset JTAG to AXI core](image)

12. 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 4"` sets the AXI burst length to 128 words
- `"-data {44444444_33333333_22222222_11111111}"` is the data to be written.
Step 2: Program the KC705 Board and Interact with the JTAG to AXI Master Core

**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.

13. 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

14. 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] vcse_server: WRITE DATA is :
   4444444333333332222222211111111...
   ```

15. 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] vcse_server: READ DATA is :
   4444444333333332222222211111111...
   ```
Step 3: Using ILA 3.0 Advanced Trigger Feature to Trigger on an AXI Read Transaction

1. Open the ILA core’s dashboard by right-clicking hw_ila_1 in the Hardware Manager window and selecting Open Dashboard.

![Figure 166: Opening the ILA Dashboard](image-url)
2. In the ILA – hw ila_1 dashboard, locate the **Trigger Mode Settings** area and set **Trigger mode** to **ADVANCED**.

3. In the **Capture Mode Settings** area set the **Trigger position** to **512**.

4. In the **Trigger State Machine** area click the **Create new trigger state machine** link.

![Figure 167: Setting Trigger mode to ADVANCED and Trigger Position to 512 in the ILA Dashboard](image)
Step 3: Using ILA 3.0 Advanced Trigger Feature to Trigger on an AXI Read Transaction

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

![New Trigger State Machine File dialog box](image)

**Figure 168: Creating a new Trigger State Machine script**

6. 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.

```plaintext
# 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.
```

Send Feedback
Step 3: Using ILA 3.0 Advanced Trigger Feature to Trigger on an AXI Read Transaction

```haskell
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.

7. Arm the ILA core’s trigger by right-clicking the `hw_ila_1` core in the **Hardware Manager** window and selecting **Run Trigger**.

![Image of Hardware Manager window with Run Trigger selected](image.png)

*Figure 169: Run Trigger*
8. 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_arvalid** state. Note that the pre-trigger capture of 512 samples has completed successfully:

![Trigger Capture Status window](image)

*Figure 167: Trigger Capture Status window*
9. In the Tcl console, run the read transaction that you set up in the previous section of this tutorial.
   
   ```tcl
   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.

   ![Waveform window]

   Figure 170: Waveform window