

# **Vivado Design Suite User Guide**

## ***Programming and Debugging***

UG908 (v2013.3) October 2, 2013



#### 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.

Date	Version	Revision
07/25/12	2012.2	Initial Xilinx release.
10/16/12	2012.3	Update to Bitstream settings.
12/18/12	2012.4	Added section for Waveform Configuration. Removed Waveform Viewer Limitations.
03/20/13	2013.1	<ul style="list-style-type: none"><li>• General updates for 2013.1.</li><li>• New section on In-System Serial I/O Debugging. New section on Description of hw_tcl Commands.</li><li>• New table for New Debugging Cores in Vivado® IP for use in HDL Instantiation Flow</li></ul>
06/19/2013	2013.2	<ul style="list-style-type: none"><li>• Updated support for hw_server including TCF Agent.</li><li>• Made various corrections and updates to reflect changes to the Vivado user interface.</li></ul>
10/02/2013	2013.3	<ul style="list-style-type: none"><li>• Updates to 2013.3 software features.</li><li>• Added new ILA Dashboard user interface.</li><li>• Added new ILA advanced trigger and capture control features.</li><li>• Added new VIO Dashboard user interface.</li><li>• Added new JTAG to AXI Master feature.</li></ul>

# Table of Contents

Revision History .....	2
<b>Chapter 1: Introduction</b>	
Getting Started .....	7
<b>Chapter 2: Programming the Device</b>	
Introduction .....	8
Generating the Bitstream .....	8
Changing the Bitstream File Format Settings .....	9
Changing Device Configuration Bitstream Settings .....	10
Programming the FPGA Device .....	12
Using a Vivado Hardware Manager to Program an FPGA Device .....	13
Launching iMPACT .....	20
<b>Chapter 3: Debugging the Design</b>	
Introduction .....	22
RTL-level Design Simulation .....	22
Post-Implemented Design Simulation .....	23
In-System Logic Design Debugging .....	23
In-System Serial I/O Design Debugging .....	23
<b>Chapter 4: In-System Logic Design Debugging Flows</b>	
Introduction .....	24
Probing the Design for In-System Debugging .....	24
Using the Netlist Insertion Debug Probing Flow .....	25
HDL Instantiation Debug Probing Flow Overview .....	36
Using the HDL Instantiation Debug Probing Flow .....	38
Implementing the Design Containing the Debug Cores .....	42
<b>Chapter 5: Debugging Logic Designs in Hardware</b>	
Introduction .....	43
Launching ChipScope Pro Analyzer to Debug the Design .....	43
Using Vivado Logic Analyzer to Debug the Design .....	44

Connecting to the Hardware Target and Programming the FPGA Device .....	44
Setting up the ILA Core to Take a Measurement .....	45
Writing ILA Probes Information .....	60
Reading ILA Probes Information .....	61
Viewing Captured Data from the ILA Core in the Waveform Viewer .....	61
Saving and Restoring Captured Data from the ILA Core .....	61
Setting Up the VIO Core to Take a Measurement .....	62
Viewing the VIO Core Status .....	63
Interacting with VIO Core Output Probes .....	69
Using Vivado Logic Analyzer in a Lab Environment .....	73
Using Vivado Logic Analyzer and ChipScope™ Pro Analyzer Simultaneously .....	75
Description of Hardware Manager Tcl Objects and Commands .....	78
Using Hardware Manager Tcl Commands .....	82

## Chapter 6: Viewing ILA Probe Data Using Waveform Viewer

Introduction .....	83
About Wave Configurations and Windows .....	83
Waveform Viewer Configuration Signals and Buses .....	85
ILA Probes in Waveform Configuration .....	86
Customizing the Waveform Configuration .....	87
Renaming Objects .....	92
About Radixes and Analog Waveforms .....	94
Zoom Gestures .....	98

## Chapter 7: In-System Serial I/O Debugging Flows

Introduction .....	99
Generating an IBERT Core using the Vivado IP Catalog .....	99
Generating and Implementing the IBERT Example Design .....	100

## Chapter 8: Debugging the Serial I/O Design in Hardware

Introduction .....	102
Using Vivado® Serial I/O Analyzer to Debug the Design .....	102

## Appendix A: Device Configuration Bitstream Settings

## Appendix B: Trigger State Machine Language Description

States .....	121
Goto Action .....	121
Conditional Branching .....	122
Counters .....	122

Flags .....	123
Conditional Statements .....	123

## **Appendix C: Additional Resources**

Xilinx Resources .....	129
Solution Centers .....	129
References .....	129



# Introduction

## Getting Started

After successfully implementing your design, the next step is to run it in hardware by programming the FPGA device and debugging the design in-system. All of the necessary commands to perform programming of FPGA devices and in-system debugging of the design are in the **Program and Debug** section of the **Flow Navigator** window in the Vivado® Integrated Design Environment (IDE) (see [Figure 1-1](#))

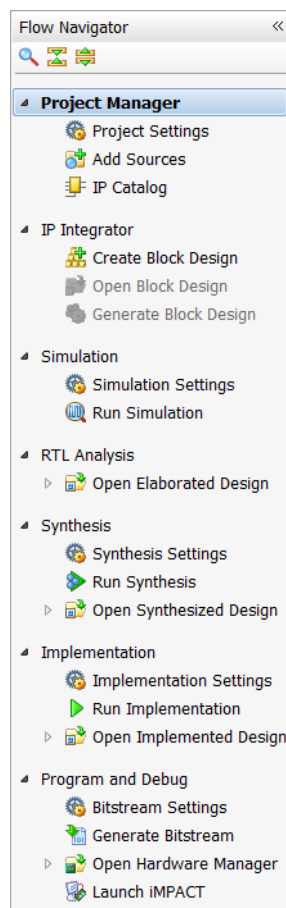


Figure 1-1: Program and Debug section of the Flow Navigator panel

# Programming the Device

---

## Introduction

The hardware programming phase is broken into two steps:

1. Generating the bitstream data programming file from the implemented design.
  2. Connecting to hardware and downloading the programming file to the target FPGA device.
- 

## Generating the Bitstream

Before generating the bitstream data file, it is important to review the bitstream settings to make sure they are correct for your design.

There are two types of bitstream settings in Vivado® IDE:

1. Bitstream file format settings.
2. Device configuration settings.

The **Bitstream Settings** button in the **Flow Navigator** and the **Flow > Bitstream Settings** menu selection opens the **Bitstream** section in the **Project Settings** popup window (see [Figure 2-1](#)). Once the bitstream settings are correct, the bitstream data file can be generated using the `write_bistream` Tcl command or by using the **Generate Bitstream** button in the **Flow Navigator**.



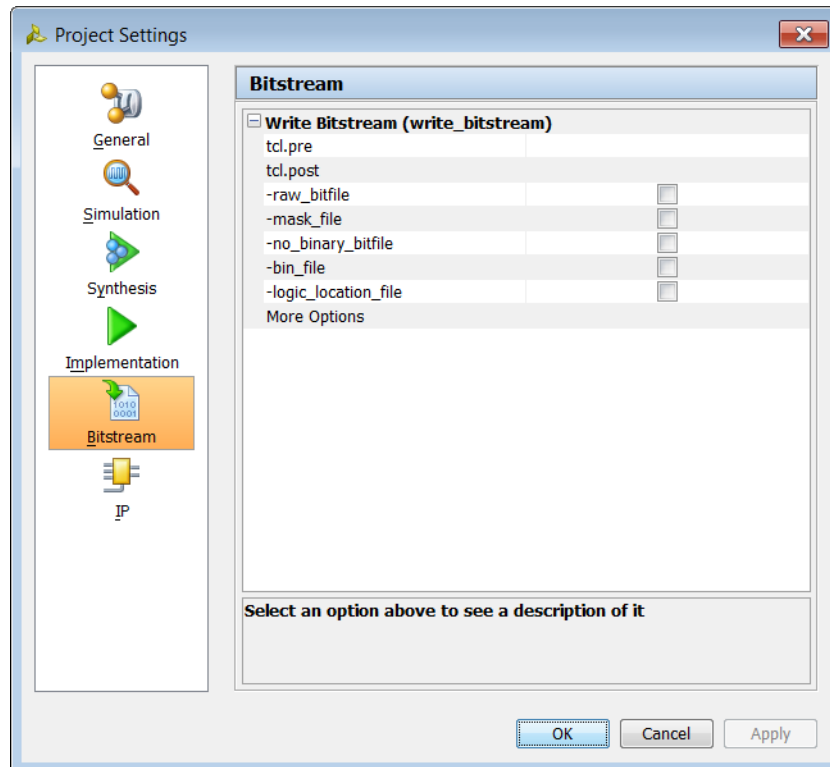


Figure 2-1: Bitstream settings panel

## Changing the Bitstream File Format Settings

By default, the `write_bitstream` Tcl command generates a binary bitstream (`.bit`) file only. You can optionally change the file formats written out by the `write_bitstream` Tcl command by using the following command switches:

- `-raw_bitfile`: (Optional) Causes `write_bitstream` to write a raw bit file (`.rbt`) which contains the same information as the binary bitstream file, but is in ASCII format. The output file is named `filename.rbt`
- `-mask_file`: (Optional) Write a mask file (`.msk`), which has mask data where the configuration data is in the bitstream file. This file determines which bits in the bitstream should be compared to readback data for verification purposes. If a mask bit is 0, that bit should be verified against the bitstream data. If a mask bit is 1, that bit should not be verified. The output file is named `file.msk`.
- `-no_binary_bitfile`: (Optional) Do not write the binary bitstream file (`.bit`). Use this command when you want to generate the ASCII bitstream or mask file, or to generate a bitstream report, without generating the binary bitstream file.
- `-logic_location_file`: (Optional) Creates an ASCII logic location file (`.ll`) that shows the bitstream position of latches, flip-flops, LUTs, Block RAMs, and I/O block inputs and

outputs. Bits are referenced by frame and bit number in the location file to help you observe the contents of FPGA registers.

- `-bin_file`: (Optional) Creates a binary file (`.bin`) containing only device programming data, without the header information found in the standard bitstream file (`.bit`).
- `-reference_bitfile <arg>`: (Optional) Read a reference bitstream file, and output an incremental bitstream file containing only the differences from the specified reference file. This partial bitstream file can be used for incrementally programming an existing device with an updated design.

---

## Changing Device Configuration Bitstream Settings

The most common configuration settings that you can change fall into the device configuration settings category. These settings are properties on the device model and you change them by using the Properties window for the selected synthesized design netlist. The following steps describe how to set various bitstream properties using this method:

1. In the synthesized design, select the top-level design in the **Netlist** window.
2. In the **Properties** window, click the green “plus” sign to open the **Add Properties** dialog box.
3. In the **Add Properties** dialog box, type **BITSTREAM** into the **Search** text field. Find and select the desired **BITSTREAM** properties, then click **OK**. (see [Figure 2-2](#)).
4. In the **Properties** window, locate the **BITSTREAM** properties and set them to the desired values (see [Figure 2-3](#)).

You can also set the bitstream properties using the `set_property` command in an XDC file. For instance, here is an example on how to change the start-up DONE cycle property:

```
set_property BITSTREAM.STARTUP.DONE_CYCLE 4 [current_design]
```

Additional examples and templates are provided in the Vivado Templates. [Appendix A, Device Configuration Bitstream Settings](#) describes all of the device configuration settings.

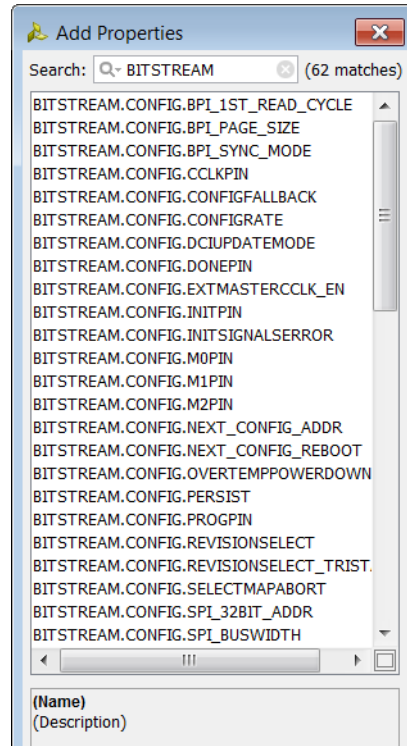


Figure 2-2: Add Bitstream Properties

5. In the **Properties** window, locate the **BITSTREAM** properties and set them to the desired values (see [Figure 2-3](#)).

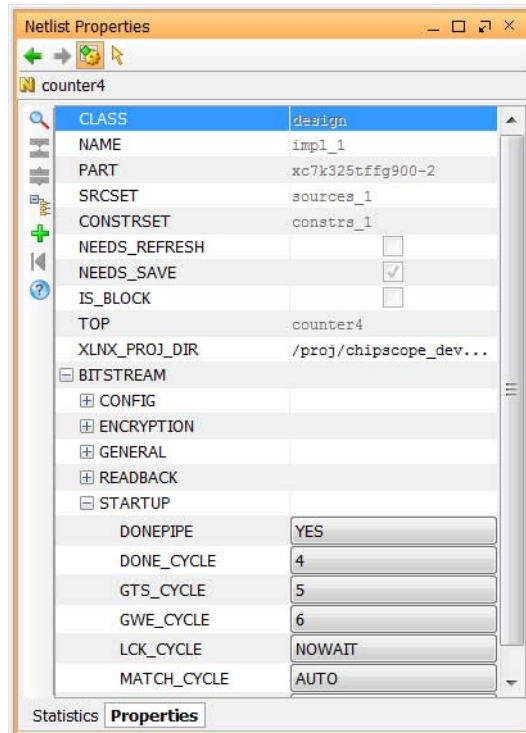


Figure 2-3: Netlist Bitstream Properties

You can also set the bitstream properties using the `set_property` command in an XDC file. For instance, here is an example on how to change the start-up DONE cycle property:

```
set_property BITSTREAM.STARTUP.DONE_CYCLE 4 [current_design]
```

Additional examples and templates are provided in the Vivado Templates. [Appendix A, Device Configuration Bitstream Settings](#) describes all of the device configuration settings.

## Programming the FPGA Device

The next step after generating the bitstream data programming file is to download it into the target FPGA device. The Vivado tool has native in-system device programming capabilities built in.

## Using a Vivado Hardware Manager to Program an FPGA Device

The Vivado IDE tool includes functionality that allows you to connect to hardware containing one or more FPGA devices to program and interact with those FPGA devices. Connecting to hardware can be done from either the Vivado IDE graphical user interface or by using Tcl commands. In either case, the steps to connect to hardware and program the target FPGA device are the same:

1. Open the hardware manager.
2. Open a hardware target that is managed by a hardware server running on a host computer.
3. Associate the bitstream data programming file with the appropriate FPGA device.
4. Program or download the programming file into the hardware device.

### Opening the Hardware Manager

Opening the hardware Manager is the first step in programming and/or debugging your design in hardware. To open the hardware Manager, do one of the following:

- If you have a project open, click the **Open Hardware Manager** button in the **Program and Debug** section of the **Flow Navigator**.
- Select the **Flow > Hardware Manager** menu option.
- In the **Tcl Console** window, run the `open_hw` command

### Opening Hardware Target Connections

The next step in opening a hardware target (for instance, a hardware board containing a JTAG chain of one or more FPGA devices) is connecting to the hardware server (also called the Vivado CSE server or `vcse_server` application) that is managing the connection to the hardware target. You can do this one of three ways:

- Use the **Open Target** selection under **Hardware Manager** in the Program and Debug section of the flow navigator of the Vivado IDE to open new or recent hardware targets (see Figure 2-4).

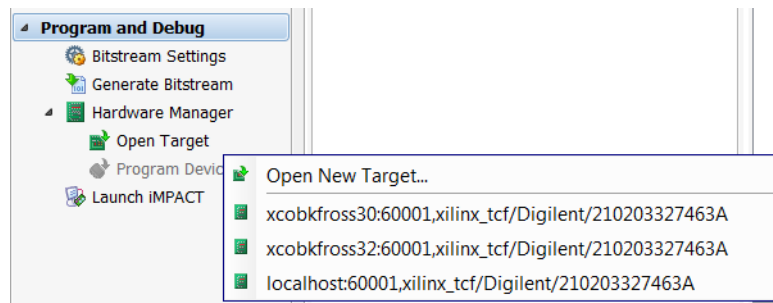


Figure 2-4: Using the Flow Navigator to Open a Hardware Target

- Use the **Open recent target** or **Open a new hardware target** selections on the green user assistance banner across the top of the **Hardware Manager** window to open recent or new hardware targets, respectively (see Figure 2-5).

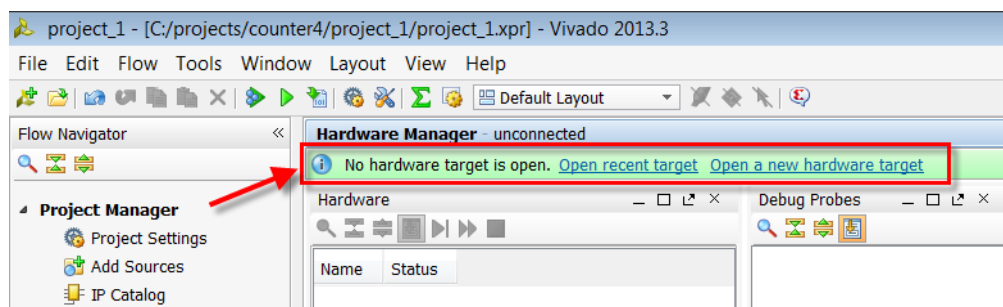


Figure 2-5: Using the User Assistance Bar to Open a Hardware Target

- Use Tcl commands to open a connection to a hardware target.

## Connecting to Hardware Target Using hw\_server TCF Agent

Vivado supports two methods of connecting to hardware targets:

- Using Target Communication Framework (TCF) Agent plug-ins [Default]
- Using legacy CSE plug-ins

The Xilinx TCF Agent is called Hardware Server (`hw_server`). The list of compatible JTAG download cables and devices that are supported by `hw_server` are:

- Xilinx Platform Cable USB II (DLC10)
- Xilinx Platform Cable USB (DLC9G, DLC9LP, DLC9)
- Digilent JTAG-HS2

- Digilent JTAG-SMT2
- Digilent JTAG-HS1
- Digilent JTAG-SMT1

The `hw_server` TCF agent is automatically started by `vcse_server`. However, you can also start the `hw_server` manually. For instance, on Windows platforms, at a `cmd` prompt run the following command:

```
C:\Xilinx\Vivado\vivado_release.version\bin\hw_server.bat
```



---

**IMPORTANT:** *Vivado and `vcse_server` automatically try to connect to compatible JTAG targets using the `hw_server` TCF agent.*

---

*If you want to connect to JTAG targets using the legacy CSE plug-ins, follow these instructions:*

- In Vivado: Before using the **Open New Hardware Target** wizard in the **Hardware Manager**, run the following command:  

```
set_param labtools.use_hw_server false
```
- If launching `vcse_server` manually, use the following command line to disable the launching of the `hw_server` TCF agent:

```
vcse_server -disable_tcf
```

Follow the steps in the next section to open a connection to a new hardware target using this agent.

## Opening a New Hardware Target

The **Open New Hardware Target** wizard provides an interactive way for you to connect to a hardware server and target. The wizard is a three-step process:

1. Specify or select the host name and port of the Vivado CSE server (also called `vcse_server`) that is managing the hardware targets on the machine to which the target board is connected (see [Figure 2-6](#)).

**Note:** If you use “localhost” or the hostname/IP address of the machine on which you are running the Vivado tool as the host name, a `vcse_server` process automatically starts on that machine. This `vcse_server` is used in the subsequent panels of the wizard.

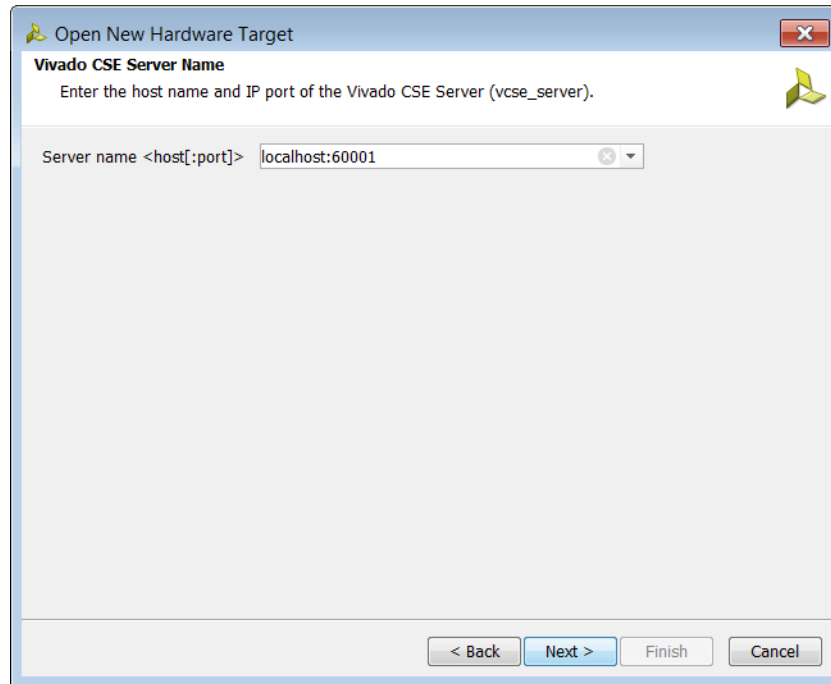


Figure 2-6: Specifying the Vivado CSE Server Name

2. Select the appropriate hardware target from the list of targets that are managed by the hardware server. Note that when you select a target, you see the various hardware devices that are available on that hardware target (see Figure 2-7).

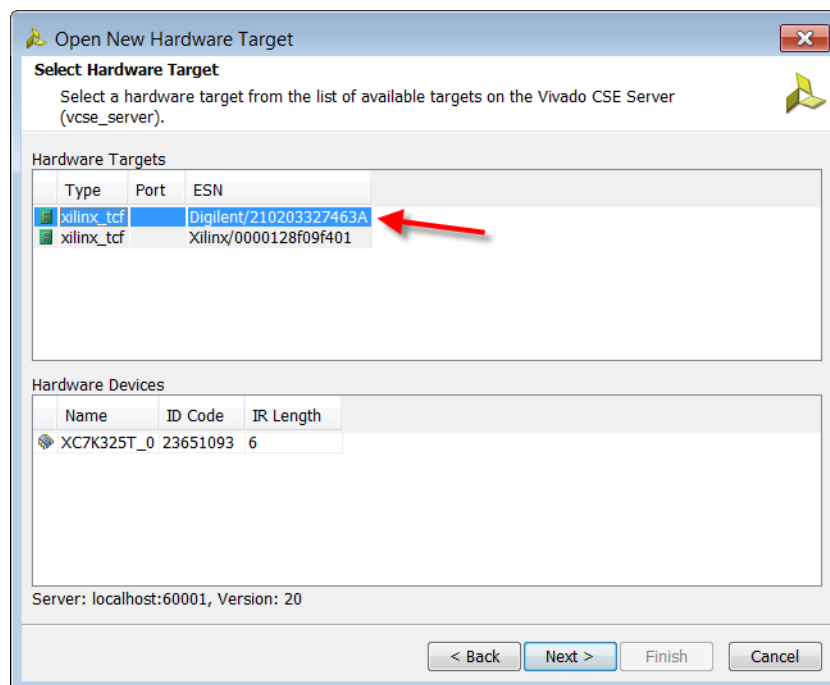


Figure 2-7: Selecting the Hardware Target



**Note:** If one or more of the devices is unknown to Vivado, you can provide the instruction register (IR) length directly in the **Hardware Devices** table of the Open New Hardware Target wizard (Figure 2-8).

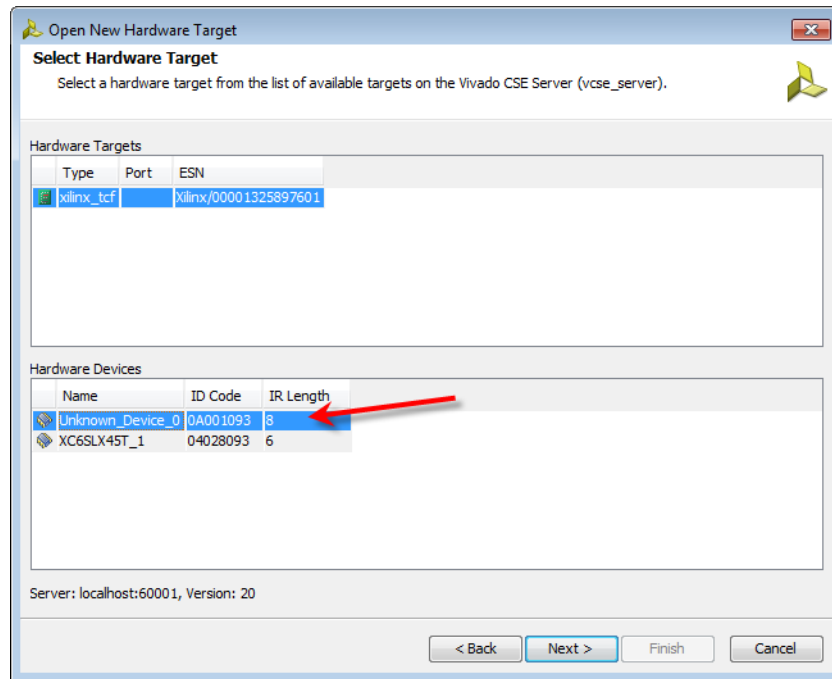


Figure 2-8: Instruction Register Length



**IMPORTANT:** Vivado and *vcse\_server* automatically try to connect to compatible JTAG targets using the *hw\_server* TCF agent.

If you want to connect to JTAG targets using the legacy CSE plug-ins, follow these instructions:

- In Vivado: Before using the **Open New Hardware Target** wizard in the **Hardware Manager**, run the following command:  
`set_param labtools.use_hw_server false`
- If launching *vcse\_server* manually, use the following command line to disable the launching of the *hw\_server* TCF agent:  
`vcse_server -disable_tcf`
- 3. Set the properties of the hardware target, such as the frequency of the TCK clock pin, etc. Note that each type of hardware target may have different properties. Refer to the [documentation](#) of each hardware target for more information about these properties.

## Opening a Recent Hardware Target

The **Open New Hardware Target** wizard is also what populates a list of previously connected hardware targets. Instead of connecting to a hardware target by going through

the wizard, you can re-open a connection to a previously connected hardware target by selecting the **Open recent target** link in the **Hardware Manager** window and selecting one of the recently connected hardware server/target combinations in the list. You can also access this list of recently used targets through the **Open Target** selection under **Hardware Manager** in the Program and Debug section of the Vivado IDE flow navigator.

## Opening a Hardware Target Using Tcl Commands

You can also use Tcl commands to connect to a hardware server/target combination. For instance, to connect to the `digilent_plugin` target (serial number 210203339395) that is managed by the `vcse_server` running on `localhost:60001`, use the following Tcl commands:

```
connect_hw_server -host localhost -port 60001
current_hw_target [get_hw_targets */xilinx_tcf/Digilent/210203327463A]
open_hw_target
```

Once you finish opening a connection to a hardware target, the **Hardware** window is populated with the hardware server, hardware target, and various hardware devices for the open target (see [Figure 2-9](#)).

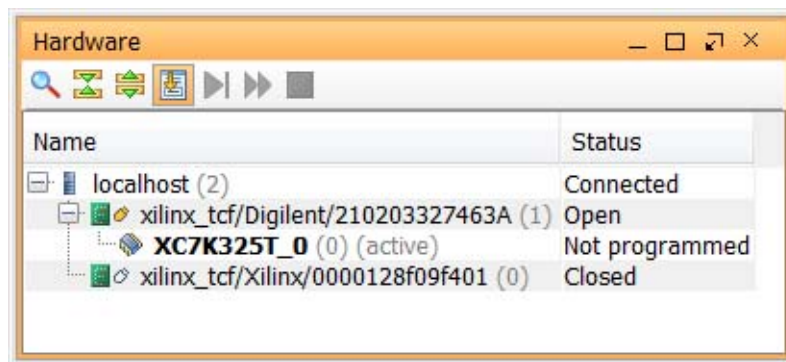


Figure 2-9: Hardware View after Opening a Connection to the Hardware Target

## Associating a Programming File with the Hardware Device

After connecting to the hardware target and before you program the FPGA device, you need to associate the bitstream data programming file with the device. Select the hardware device in the **Hardware** window and make sure the **Programming file** property in the **Properties** window is set to the appropriate bitstream data (.bit) file.

**Note:** As a convenience, Vivado IDE automatically uses the .bit file for the current implemented design as the value for the **Programming File** property of the first matching device in the open hardware target. This feature is only available when using the Vivado IDE in project mode. When using the Vivado IDE in non-project mode, you need to set this property manually.

You can also use the `set_property` Tcl command to set the `PROGRAM.FILE` property of the hardware device:

```
set_property PROGRAM.FILE {C:/design.bit} [lindex [get_hw_devices] 0]
```

## Programming the Hardware Device

Once the programming file has been associated with the hardware device, you can program the hardware device using by right-clicking on the device in the **Hardware** window and selecting the **Program Device** menu option. You can also use the `program_hw_device` Tcl command. For instance, to program the first device in the JTAG chain, use the following Tcl command:

```
program_hw_devices [lindex [get_hw_devices] 0]
```

Once the progress dialog has indicated that the programming is 100% complete, you can check that the hardware device has been programmed successfully by examining the DONE status in the Hardware Device Properties view (see [Figure 2-10](#)).

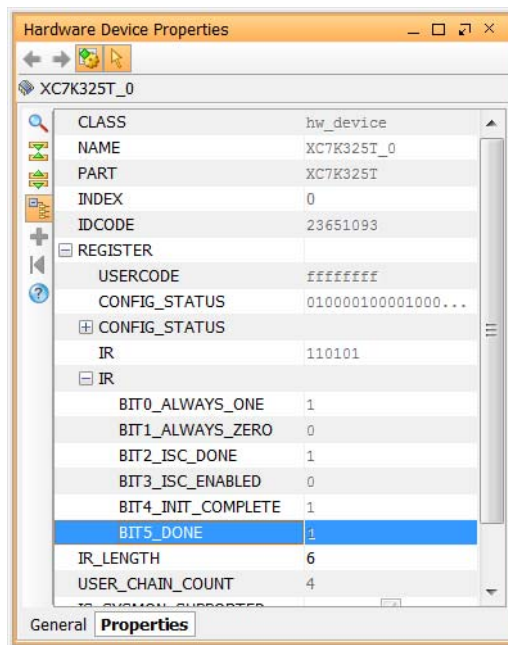


Figure 2-10: Checking the DONE status of an FPGA device

You can also use the `get_property` Tcl command to check the DONE status. For instance, to check the DONE status of a Kintex™-7 device that is the first device in the JTAG chain, use the following Tcl command:

```
get_property REGISTER.IR.BIT5_DONE [lindex [get_hw_devices] 0]
```

If you use another means to program the hardware device (for instance, a flash device or external device programmer such as the iMPACT tool), you can also refresh the status of a hardware device by right-clicking the **Refresh Device** menu option or by running the `refresh_hw_device` Tcl command. This refreshes the various properties for the device, including but not limited to the DONE status.

## Closing the Hardware Target

You can close a hardware target by right-clicking on the hardware target in the **Hardware** window and selecting **Close Target** from the popup menu. You can also close the hardware target using a Tcl command. For instance, to close the xilinx\_platformusb/USB21 target on the localhost server, use the following Tcl command:

```
close_hw_target {localhost/xilinx_tcf/Digilent/210203327463A}
```

## Closing a Connection to the Hardware Server

You can close a hardware server by right-clicking on the hardware server in the **Hardware** window and selecting **Close Server** from the popup menu. You can also close the hardware server using a Tcl command. For instance, to close the connection to the localhost server, use the following Tcl command:

```
disconnect_hw_server localhost
```

---

## Launching iMPACT

The iMPACT tool lets you perform device configuration and file generation.

- Device Configuration lets you directly configure Xilinx FPGAs and PROMs with the JTAG download cables (Xilinx Parallel Cable IV, Xilinx Platform Cable USB, Xilinx Platform Cable USB II, or Digilent JTAG cables).
- Operating in Boundary-Scan mode, iMPACT can configure or program Xilinx FPGAs, CPLDs, and PROMs.
- File generation enables you to create the following programming file types: System ACE™ CF, PROM, SVF, STAPL, and XSVF files.

iMPACT also lets you:

- Readback and verify design configuration data.
- Debug configuration problems.
- Execute SVF and XSVF files.

You can launch the iMPACT software tool directly from the Vivado IDE on any implemented design on which the Generate Bitstream command has been run. To invoke iMPACT, in the **Flow Navigator**, select **Launch iMPACT**.

The BIT bitstream file is passed automatically to iMPACT when launched from the Vivado tool. For more information on using iMPACT, see the iMPACT Help.



**IMPORTANT:** To launch the iMPACT tool, the ISE Design Suite or ISE Lab Tools software needs to be installed on your system and the iMPACT tool needs to be accessible in your PATH environment variable. If you do not meet these two requirements, you will see the error dialog shown in [Figure 2-11](#) when you try to launch the iMPACT tool.

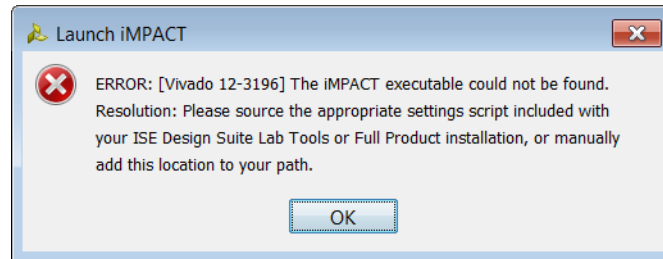


Figure 2-11: Problem locating\_iMPACT executable

# Debugging the Design

---

## Introduction

Debugging an FPGA design is a multistep, iterative process. Like most complex problems, it is best to break the FPGA design debugging process down into smaller parts by focusing on getting smaller sections of the design working one at a time rather than trying to get the whole design to work at once. Iterating through the design flow by adding one module at a time and getting it to function properly in the context of the whole design is one example of a proven design and debug methodology. You can use this design and debug methodology in any combination of the following design flow stages:

- RTL-level design simulation
  - Post-implemented design simulation
  - In-system debugging
- 

## RTL-level Design Simulation

The design can be functionally debugged during the simulation verification process. Xilinx provides a full design simulation feature in the Vivado® IDE. The Vivado design simulator can be used to perform RTL simulation of your design. The benefits of debugging your design in an RTL-level simulation environment include full visibility of the entire design and ability to quickly iterate through the design/debug cycle. The limitations of debugging your design using RTL-level simulation includes the difficulty of simulating larger designs in a reasonable amount of time in addition to the difficulty of accurately simulating the actual system environment. For more information about using the Vivado simulator, refer to the *Vivado Design Suite User Guide: Logic Simulation (UG937)* [Ref 1].

---

## Post-Implemented Design Simulation

The Vivado simulator can also be used to simulate the post-implemented design. One of the benefits of debugging the post-implemented design using the Vivado simulator includes having access to a timing-accurate model for the design. The limitations of performing post-implemented design simulation include those mentioned in the previous section: long run-times and system model accuracy.

---

## In-System Logic Design Debugging

The Vivado IDE also includes a logic analysis feature that enables you to perform in-system debugging of the post-implemented design on an FPGA device. The benefits for debugging your design in-system include debugging your timing-accurate, post-implemented design in the actual system environment at system speeds. The limitations of in-system debugging includes somewhat lower visibility of debug signals compared to using simulation models and potentially longer design/implementation/debug iterations, depending on the size and complexity of the design.

In general, the Vivado tool provides several different ways to debug your design. You can use one or more of these methods to debug your design, depending on your needs. [Chapter 4, In-System Logic Design Debugging Flows](#) focuses on the in-system logic debugging capabilities of the Vivado IDE.

---

## In-System Serial I/O Design Debugging

To enable in-system serial I/O validation and debug, the Vivado IDE includes a serial I/O analysis feature. This allows you to measure and optimize your high-speed serial I/O links in your FPGA-based system. The Vivado serial I/O analyzer features are designed to help you address a range of in-system debug and validation problems from simple clocking and connectivity issues to complex margin analysis and channel optimization issues. The main benefit of using the Vivado serial I/O analyzer over some other external instrumentation techniques is that you are measuring the quality of the signal after the receiver equalization has been applied to the received signal. This ensures that you are measuring at the optimal point in the TX-to-RX channel thereby ensuring real and accurate data.

The Vivado tool provides the means to generate the design used to exercise the gigabit transceiver endpoints as well as the run-time software to take measurements and help you optimize your high-speed serial I/O channels. [Chapter 7, In-System Serial I/O Debugging Flows](#) guides you through the process of generating the IBERT design. [Chapter 8, Debugging the Serial I/O Design in Hardware](#) guides you through the use of the run time Vivado serial I/O analyzer feature.

## In-System Logic Design Debugging Flows

---

### Introduction

The Vivado® tool provides many features to debug a design in-system in an actual hardware device. The in-system debugging flow has three distinct phases:

1. **Probing phase:** Identifying what signals in your design you want to probe and how you want to probe them.
2. **Implementation phase:** Implementing the design that includes the additional debug IP that is attached to the probed nets.
3. **Analysis phase:** Interacting with the debug IP contained in the design to debug and verify functional issues.

This in-system debug flow is designed to work using the iterative design/debug flow described in the previous section. If you choose to use the in-system debugging flow, it is advisable to get a part of your design working in hardware as early in the design cycle as possible. The rest of this chapter describes the three phases of the in-system debugging flow and how to use the Vivado® logic debug feature to get your design working in hardware as quickly as possible.

---

### Probing the Design for In-System Debugging

The probing phase of the in-system debugging flow is split into two steps:

1. Identifying what signals or nets you want to probe
2. Deciding how you want to add debug cores to your design

In many cases, the decision you make on what signals to probe or how to probe them can affect one another. It helps to start by deciding if you want to manually add the debug IP component instances to your design source code (called the HDL instantiation probing flow) or if you want the Vivado tool to automatically insert the debug cores into your post-synthesis netlist (called the netlist insertion probing flow). [Table 4-1](#) describes some of the advantages and trade-offs of the different debugging approaches.



Table 4-1: Debugging Strategies

Debugging Goal	Recommended Debug Programming Flow
Identify debug signals in the HDL source code while retaining flexibility to enable/disable debugging later in the flow.	<ul style="list-style-type: none"> <li>Use mark_debug property to tag signals for debugging in HDL.</li> <li>Use the <b>Set up Debug</b> wizard to guide you through the Netlist Insertion probing flow.</li> </ul>
Identify debug nets in synthesized design netlist without having to modify the HDL source code.	<ul style="list-style-type: none"> <li>Use the <b>Mark Debug</b> right-click menu option to select nets for debugging in the synthesized design netlist.</li> <li>Use the <b>Set up Debug</b> wizard to guide you through the Netlist Insertion probing flow.</li> </ul>
Automated debug probing flow using Tcl commands.	<ul style="list-style-type: none"> <li>Use set_property Tcl command to set the mark_debug property on debug nets.</li> <li>Use Netlist Insertion probing flow Tcl commands to create debug cores and connect to them to debug nets.</li> </ul>
Explicitly attach signals in the HDL source to an ILA debug core instance.	<ul style="list-style-type: none"> <li>Identify HDL signals for debugging.</li> <li>Use the HDL Instantiation probing flow to generate and instantiate an Integrated Logic Analyzer (ILA) core and connect it to the debug signals in the design.</li> </ul>

## Using the Netlist Insertion Debug Probing Flow

Insertion of debug cores in the Vivado tool is presented in a layered approach to address different needs of the diverse group of Vivado users:

- The highest level is a simple wizard that creates and configures Integrated Logic Analyzer (ILA) cores automatically based on the selected set of nets to debug.
- The next level is the main **Debug** window allowing control over individual debug cores, ports and their properties. The **Debug** window can be displayed when the Synthesized Design is open by selecting the **Debug layout** from the **Layout Selector** or the **Layout** menu, or can be opened directly using **Window > Debug**.
- The lowest level is the set of Tcl XCD debug commands that you can enter manually into an XDC constraints file or replay as a Tcl script.

You can also use a combination of the modes to insert and customize debug cores.

## Marking HDL Signals for Debug

You can identify signals for debugging at the HDL source level prior to synthesis by using the mark\_debug constraint. Nets corresponding to signals marked for debug in HDL are automatically listed in the **Debug** window under the **Unassigned Debug Nets** folder.

**Note:** In the **Debug** window, the **Debug Nets** view is a more net-centric view of nets that you have selected for debug. The **Debug Cores** view is a more core-centric view where you can view and set core properties.

The procedure for marking nets for debug depends on whether you are working with an RTL source-based project or a synthesized netlist-based project. For an RTL netlist-based project:

- Using the Vivado synthesis feature you can optionally mark HDL signals for debug using the `mark_debug` constraint in VHDL and Verilog source files. The valid values for the `mark_debug` constraint are "TRUE" or "FALSE". The Vivado synthesis feature does not support the "SOFT" value.
- Using Xilinx Synthesis Technology (XST) you can optionally mark nets for debug using the `mark_debug` constraint in VHDL and Verilog sources. In addition to the boolean string values of, "TRUE" or "FALSE," a value of "SOFT" allows the software to optimize the specified net, if possible.

For a synthesized netlist-based project:

- Using the Synopsys® Synplify® synthesis tool, you can optionally mark nets for debug using the `mark_debug` and `syn_keep` constraints in VHDL or Verilog, or using the `mark_debug` constraint alone in the Synopsys Design Constraints (SDC) file. Synplify does not support the "SOFT" value, as this behavior is controlled by the `syn_keep` attribute.
- Using the Mentor Graphics® Precision® synthesis tool, you can optionally mark nets for debug using the `mark_debug` constraint in VHDL or Verilog.

The following subsections provide syntactical examples for Vivado synthesis, XST, Synplify, and Precision source files.

## Vivado Synthesis `mark_debug` Syntax Examples

The following are examples of VHDL and Verilog syntax when using Vivado synthesis.

- VHDL Syntax Example

```
attribute mark_debug : string;  
attribute mark_debug of char_fifo_dout: signal is "true";
```

- Verilog Syntax Example

```
(* mark_debug = "true" *) wire [7:0] char_fifo_dout;
```

## XST mark\_debug Syntax Examples

The following are examples of VHDL and Verilog syntax when using XST.

- VHDL Syntax Example

```
attribute mark_debug : string;
attribute mark_debug of char_fifo_dout: signal is "true";
```

- Verilog Syntax Example

```
(* mark_debug = "true" *) wire [7:0] char_fifo_dout;
```

## Synplify mark\_debug Syntax Examples

The following are examples of Synplify syntax for VHDL, Verilog, and SDC.

- VHDL Syntax Example

```
attribute syn_keep : boolean;
attribute mark_debug : string;
attribute syn_keep of char_fifo_dout: signal is true;
attribute mark_debug of char_fifo_dout: signal is "true";
```

- Verilog Syntax Example

```
(* syn_keep = "true", mark_debug = "true" *) wire [7:0] char_fifo_dout;
```

- SDC Syntax Example

```
define_attribute {n:char_fifo_din[*]} {mark_debug} {"true"}
```




---

**IMPORTANT:** Net names in an SDC source must be prefixed with the "n:" qualifier.

---

**Note:** Synopsys Design Constraints (SDC) is an accepted industry standard for communicating design intent to tools, particularly for timing analysis. A reference copy of the SDC specification is available from Synopsys by registering for the TAP-in program at:

<http://www.synopsys.com/Community/Interoperability/Pages/TapinSDC.aspx>

## Precision mark\_debug Syntax Examples

The following are examples of VHDL and Verilog syntax when using Precision.

- VHDL Syntax Example

```
attribute mark_debug : string;
attribute mark_debug of char_fifo_dout: signal is "true";
```

- Verilog Syntax Example

```
(* mark_debug = "true" *) wire [7:0] char_fifo_dout;
```

## Synthesizing the Design

The next step is to synthesize the design containing the debug cores by clicking **Run Synthesis** in the Vivado IDE or by running the following Tcl commands:

```
launch_runs synth_1
wait_on_run synth_1
```

You can also use the synth\_design Tcl command to synthesize the design. Refer to the *Vivado Design Suite User Guide: Synthesis (UG901)* [Ref 2] for more details on the various ways you can synthesize your design.

## Marking Nets for Debug in the Synthesized Design

Open the synthesized design by clicking **Open Synthesized Design** in the **Flow Navigator** and select the **Debug** window layout to see the **Debug** window. Any nets that correspond to HDL signals that were marked for debugging are shown in the **Unassigned Debug Nets** folder in the **Debug** window (see Figure 4-1).

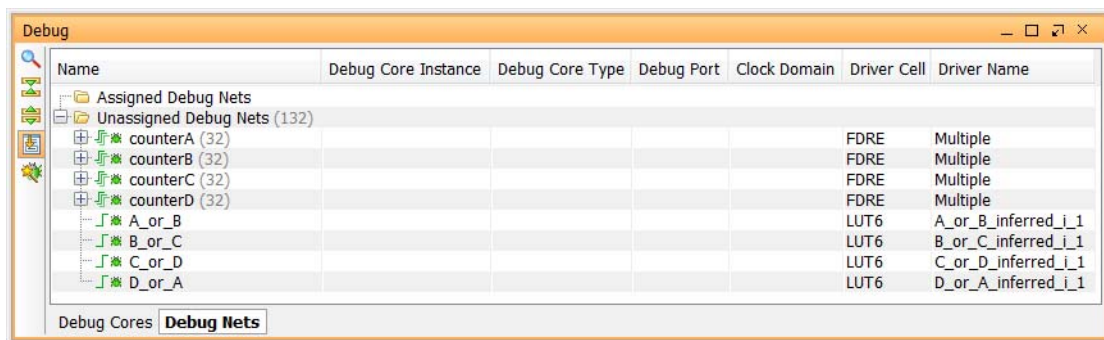


Figure 4-1: Unassigned Debug Nets

- Selecting a net in any of the design views (such as the **Netlist** or **Schematic** windows), then right-click select the **Mark Debug** option.
- Selecting a net in any of the design views, then dragging and dropping the nets into the **Unassigned Debug Nets** folder.
- Using the net selector in the **Set up Debug** wizard (see [Using the Set Up Debug Wizard to Insert Debug Cores](#) for details).

## Using the Set Up Debug Wizard to Insert Debug Cores

The next step after marking nets for debugging is to assign them to debug cores. The Vivado IDE provides an easy to use **Set up Debug** wizard to help guide you through the process of automatically creating the debug cores and assigning the debug nets to the inputs of the cores.

To use the **Set up Debug** wizard to insert the debug cores:

1. Optionally, select a set of nets for debugging either using the unassigned nets list or direct net selection.
2. Select **Tools > Set up Debug** from the Vivado IDE main menu.
3. Click **Next** to get to the **Specify Nets to Debug** panel (see [Figure 4-2](#)).
4. Optionally, click **Add/Remove Nets** to add more nets or remove existing nets from the table. You can also right-click a debug net and select **Remove Nets** to remove nets from the table.
5. Right-click a debug net and select **Select Clock Domain** to change the clock domain to be used to sample value on the net.

**Note:** The **Set up Debug** wizard attempts to automatically select the appropriate clock domain for the debug net by searching the path for synchronous elements. Use the **Select Clock Domain** dialog window to modify this selection as needed, but be aware that each clock domain present in the table results in a separate ILA core instance.

6. Once you are satisfied with the debug net selection, click **Next**.

**Note:** The **Set up Debug** wizard inserts one ILA core per clock domain. The nets that were selected for debug are assigned automatically to the probe ports of the inserted ILA cores. The last wizard screen shows the core creation summary displaying the number of clocks found and ILA cores to be created and/or removed.

7. If you want to enable either advanced trigger mode or basic capture mode, use the corresponding check boxes to do so. Click **Next** to move to the last panel.

**Note:** The advanced trigger mode and basic capture mode features are described in more detail in [Chapter 5, Debugging Logic Designs in Hardware](#).

8. If you are satisfied with the results, click **Finish** to insert and connect the ILA cores in your synthesized design netlist.

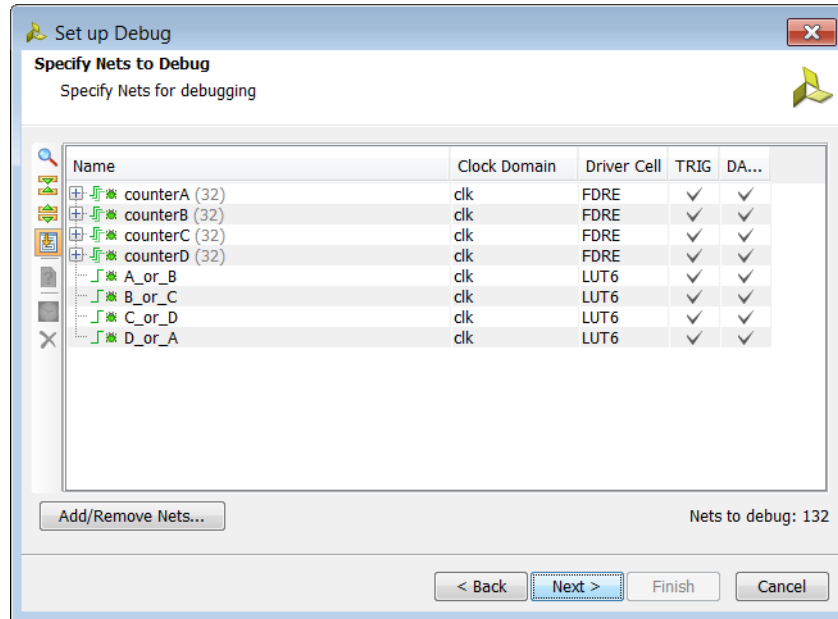


Figure 4-2: Set Up Debug Wizard

9. The debug nets are now assigned to the ILA debug core, as shown in Figure 4-3.

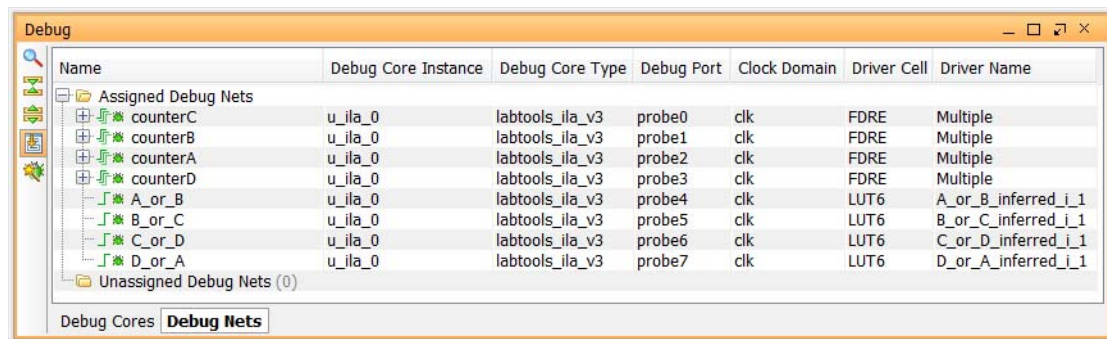


Figure 4-3: Assigned Debug Nets

## Using the Debug Window to Add and Customize Debug Cores

The **Debug Cores** tab in the **Debug** window provides more fine-grained control over ILA core and debug core hub insertion than what is available in the **Set up Debug** wizard. The controls available in this window allow core creation, core deletion, debug net connection, and core parameter changes.

The **Debug Cores** tab of the **Debug** window:

- Shows the list of debug cores that are connected to the Debug Hub (dbg\_hub) core.
- Maintains the list of unassigned debug nets at the bottom of the window.

You can manipulate debug cores and ports from the popup menu or the toolbar buttons on the top of the window.

## Creating and Removing Debug Cores

To create debug cores in the **Debug** window, click **Create Debug Core**. Using this interface (see [Figure 4-4](#)), you can change the parent instance, debug core name, and set parameters for the core. To remove an existing debug core, right-click on the core in the **Debug** window and select **Delete**. Refer to [Table 4-2, page 33](#) for a description of the ILA core options found in the **Create Debug Core** dialog.

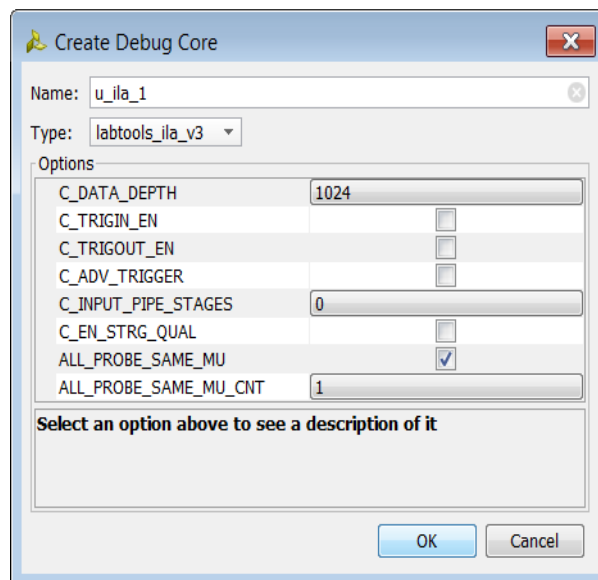


Figure 4-4: Creating a New Debug Core

## Adding, Removing, and Customizing Debug Core Ports

In addition to adding and removing debug cores, you can also add, remove, and customize ports of each debug core to suit your debugging needs. To add a new debug port:

1. Select the debug core in the **Debug** window.
2. Click **Create Debug Port** to open the dialog shown in [Figure 4-5](#).
3. Select or type in the port width
4. Click **OK**.
5. To remove a debug port, first select the port on the core in the **Debug** window, then select **Delete**.

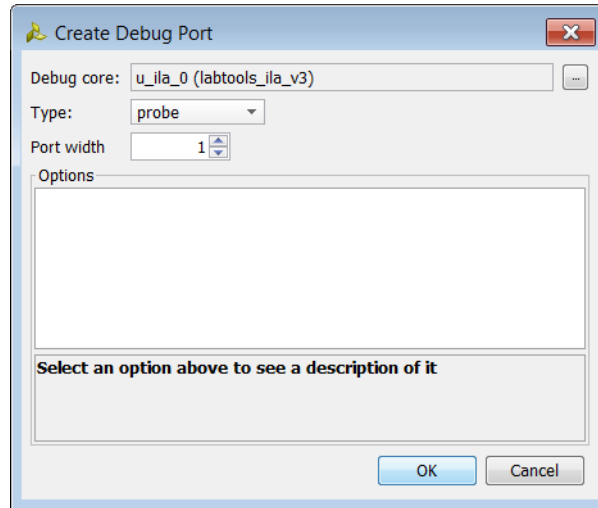


Figure 4-5: Creating a New Debug Port

## Connecting and Disconnecting Nets to Debug Cores

You can select, drag, and drop nets and buses (also called bus nets) from the **Schematic** or **Netlist** windows onto the debug core ports. This expands the debug port as needed to accommodate the net selection. You can also right-click on any net or bus, and select **Assign to Debug Port**.

To disconnect nets from the debug core port, select the nets that are connected to the debug core port, and click **Disconnect Net**.

## Modifying Properties on the Debug Cores

Each debug core has properties you can change to customize the behavior of the core. To learn how to change properties on the `debug_core_hub` debug core, refer to [Changing the BSCAN User Scan Chain of the Debug Core Hub, page 41](#).

You can also change properties on the ILA debug core. For instance, to change the number of samples captured by the ILA debug core (see [Figure 4-6](#)), do the following:

1. In the **Debug** window, select the desired ILA core (such as **u\_ila\_0**).
2. In the **Cell Properties** window, select the **Debug Core Options** view.
3. Using the `C_DATA_DEPTH` pull-down list, select the desired number of samples to be captured.



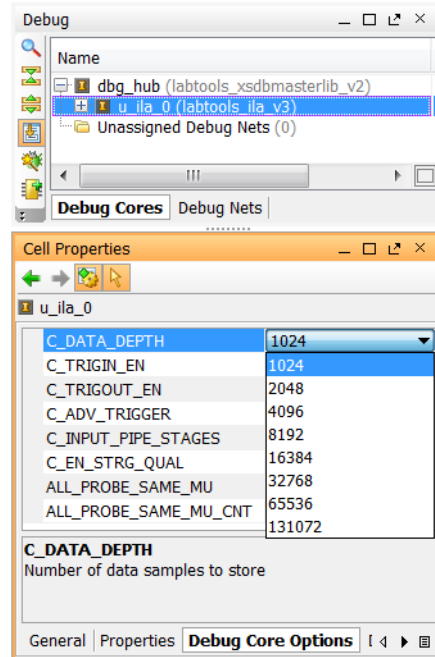


Figure 4-6: Changing the Data Depth of the ILA Core

A full description of all ILA core properties can be found in [Table 4-2](#).

Table 4-2: ILA Debug Core Properties

Debug Core Property	Description	Possible Values
C_DATA_DEPTH	Maximum number of data samples that can be stored by the ILA core. Increasing this value causes more block RAM to be consumed by the ILA core and can adversely affect design performance.	1024 (Default) 2048 4096 8192 16384 32768 65536 131072
C_TRIGIN_EN	Enables the TRIG_IN and TRIG_IN_ACK ports of the ILA core. Note that you need to use the advanced netlist change commands to connect these ports to nets in your design. If you wish to use the ILA trigger input or output signals, you should consider using the HDL instantiation method of adding ILA cores to your design.	false (Default) true
C_TRIGOUT_EN	Enables the TRIG_OUT and TRIG_OUT_ACK ports of the ILA core. Note that you need to use the advanced netlist change commands to connect these ports to nets in your design. If you wish to use the ILA trigger input or output signals, you should consider using the HDL instantiation method of adding ILA cores to your design.	false (Default) true

Table 4-2: ILA Debug Core Properties

Debug Core Property	Description	Possible Values
C_ADV_TRIGGER	Enables the advanced trigger mode of the ILA core. Refer to <a href="#">Chapter 5</a> for more details on this feature.	false (Default) true
C_INPUT_PIPE_STAGES	Enables extra levels of pipe stages (for example, flip-flop registers) on the PROBE inputs of the ILA core. This feature can be used to improve timing performance of your design by allowing the Vivado tools to place the ILA core away from critical sections of the design.	0 (Default) 1 2 3 4 5 6
C_EN_STRG_QUAL	Enables the basic capture control mode of the ILA core. Refer to <a href="#">Chapter 5</a> for more details on this feature.	false (Default) true
C_ALL_PROBE_SAME_MU	Enables all PROBE inputs of the ILA core to have the same number of comparators (also called "match units"). This property should always be set to true.	true (Default) false (not recommended)
C_ALL_PROBE_SAME_MU_CNT	<p>The number of comparators (or match units) per PROBE input of the ILA core. The number of comparators that are required depends on the settings of the C_ADV_TRIGGER and C_EN_STRG_QUAL properties:</p> <p>If C_ADV_TRIGGER is false and C_EN_STRG_QUAL is false, then set to 1</p> <ul style="list-style-type: none"> <li>If C_ADV_TRIGGER is false and C_EN_STRG_QUAL is true, then set to 2.</li> <li>If C_ADV_TRIGGER is true and C_EN_STRG_QUAL is false, then set to 1 through 4 (4 is recommended in this case).</li> <li>If C_ADV_TRIGGER is true and C_EN_STRG_QUAL is true, then set to 2 through 4 (4 is recommended in this case).</li> </ul> <p>IMPORTANT: if you do not follow the rules above, you will encounter an error during implementation when the ILA core is generated.</p>	1 2 3 4

## Using XDC Commands to Insert Debug Cores

In addition to using the **Set up Debug** wizard, you can also use XDC commands to create, connect, and insert debug cores into your synthesized design netlist. Follow these steps by typing the XDC commands in the Tcl Console:

1. Open the synthesized design netlist from the synthesis run called `synth_1`.

```
open_run synth_1.
```




---

**IMPORTANT:** *The XDC commands in the following steps are only valid when a synthesized design netlist is open.*

---

2. Create the ILA core black box.

```
create_debug_core u_ila_0 labtools_ila_v3
```

3. Set the data depth property of the ILA core.

```
set_property C_DATA_DEPTH 1024 [get_debug_cores u_ila_0]
```

4. Set the width of the clk port of the ILA core to 1 and connect it to the desired clock net.

```
set_property port_width 1 [get_debug_ports u_ila_0/clk]
connect_debug_port u_ila_0/clk [get_nets [list clk ]]
```

**Note:** You do not have to create the clk port of the ILA core because it is automatically created by the `create_debug_core` command.




---

**IMPORTANT:** *All debug port names of the debug cores are lower case. Using upper-case or mixed-case debug port names will result in an error.*

---

5. Set the width of the prob0 port to the number of nets you plan to connect to the port.

```
set_property port_width 1 [get_debug_ports u_ila_0/probe0]
```

**Note:** You do not have to create the first probe port (probe0) of the ILA core because it is automatically created by the `create_debug_core` command.

6. Connect the probe0 port to the nets you want to attach to that port.

```
connect_debug_port u_ila_0/probe0 [get_nets [list A_or_B]]
```

7. Optionally, create more probe ports, set their width, and connect them to the nets you want to debug.

```
create_debug_port u_ila_0 probe
set_property port_width 2 [get_debug_ports u_ila_0/probe1]
connect_debug_port u_ila_0/probe1 [get_nets [list {A[0]} {A[1]}]]
```

8. Optionally, generate and synthesize the debug cores so you can floorplan them with the rest of your synthesized design.

```
implement_debug_core [get_debug_cores]
```

For more information on these and other related Tcl commands, type `help -category ChipScope` in the Tcl Console of the Vivado IDE.

## Saving Constraints After Running Debug XDC Commands

You need to save constraints after using the **Set up Debug** wizard, using Vivado IDE to create debug cores or ports, and/or running the following XDC commands:

- `create_debug_core`
- `create_debug_port`

- `connect_debug_port`
- `set_property` (on any `debug_core` or `debug_port` object)

The corresponding XDC commands are saved to the target constraints file and are used during implementation to insert and connect the debug cores.



**IMPORTANT:** *Saving constraints to the target constraints file while in project mode causes the synthesis and implementation steps to go out-of-date. However, you do not need to re-synthesize the design since the debug XDC constraints are only used during implementation. You can force the synthesis step up-to-date by selecting the **Design Runs** window, right-clicking the synthesis run (e.g., `synth_1`), and selecting **Force up-to-date**.*

## Implementing the Design

After inserting, connecting and customizing your debug cores, you are now ready for implementing your design (refer to [Implementing the Design Containing the Debug Cores](#)).

# HDL Instantiation Debug Probing Flow Overview

The HDL instantiation probing flow involves the manual customization, instantiation, and connection of various debug core components directly in the HDL design source. The new debug cores that are supported in this flow in the Vivado tool are shown in table [Table 4-3](#). The legacy debug cores that are supported in this flow in the Vivado tool are shown in [Table 4-4](#).

**Table 4-3: Debug Cores in Vivado IP Catalog available for use in the HDL Instantiation Probing Flow**

Debug Core	Version	Description	Run Time Analyzer Tool
ILA (Integrated Logic Analyzer)	v3.0	Debug core that is used to trigger on hardware events and capture data at system speeds.	Vivado logic analyzer
VIO (Virtual Input/Output)	v3.0	Debug core that is used to monitor or control signals in a design at JTAG chain scan rates.	Vivado logic analyzer
JTAG-to-AXI Master	v1.0	Debug core that is used to generate AXI transactions to interact with various AXI full and AXI lite slave cores in a system that is running in hardware.	Vivado logic analyzer

**Table 4-4: Legacy ChipScope Pro Debug Cores available for use in the HDL Instantiation Probing Flow**

Debug Core	Version	Description	Run Time Analyzer Tool
ICON (Integrated Controller)	v1.06a	Debug core hub used to connect the ILA 1.05a and VIO 1.05a cores to the JTAG chain.	ChipScope Pro Analyzer
VIO (Virtual Input/Output)	v1.05a	Debug core that is used to monitor or control signals in a design at JTAG chain scan rates. Requires connection to an ICON core.	ChipScope Pro Analyzer
ILA (Integrated Logic Analyzer)	v1.05a	Debug core that is used to trigger on hardware events and capture data at system speeds. Requires connection to an ICON core.	ChipScope Pro Analyzer

## Using Legacy Debug Cores in Vivado Designs

The ICON, VIO, and ILA v1.x cores are supported in the Vivado tool flow to provide compatibility with legacy designs that contain these cores.

However, these cores are not available in the Vivado IP Catalog. You must supply the legacy debug cores netlist (.ngc) file, synthesis (.v or .vhdl) template file, and constraints (.xdc) file that were previously generated using the Xilinx ISE CORE Generator as source files in your Vivado project. You must also set properties on these files to make sure they work properly in the Vivado tool flow. Here is an example of how to add legacy ICON, ILA, and VIO files to your Vivado project:

- Add the .ngc and .v files.

```
add_files -norecurse {./icon_v1_06a.ngc}
add_files -norecurse {./ila_v1_05a.ngc}
add_files -norecurse {./vio_v1_05a.ngc}
add_files -norecurse {./icon_v1_06a.v}
add_files -norecurse {./ila_v1_05a.v}
add_files -norecurse {./vio_v1_05a.v}
```

- Import the .xdc files and specify that they are not used during synthesis.

```
import_files -fileset constrs_1 -force -norecurse {./icon_v1_06a.xdc}
import_files -fileset constrs_1 -force -norecurse {./ila_v1_05a.xdc}
import_files -fileset constrs_1 -force -norecurse {./vio_v1_05a.xdc}
set_property used_in_synthesis false [get_files icon_v1_06a.xdc]
set_property used_in_synthesis false [get_files vio_v1_05a.xdc]
set_property used_in_synthesis false [get_files ila_v1_05a.xdc]
```

- Scope the legacy ILA core's .xdc file to the cell reference for the ILA core module.

```
set_property SCOPED_TO_REF {ila_v1_05a} [get_files ila_v1_05a.xdc]
```

The new ILA core has two distinct advantages over the legacy ILA v1.x core:

- Works with the integrated Vivado logic analyzer feature (refer to [Debugging Logic Designs in Hardware](#), page 43).
- No ICON core instance or connection is required.

---

## Using the HDL Instantiation Debug Probing Flow

The steps required to perform the HDL instantiation flow are:

1. Customize and generate the ILA and/or VIO debug cores that have the right number of probe ports for the signals you want to probe.
2. (Optional) Customize and generate the JTAG-to-AXI Master debug core and connect it to an AXI slave interface of an AXI peripheral or interconnect core in your design.
3. Synthesize the design containing the debug cores.
4. (Optional) Modify debug hub core properties.
5. Implement the design containing the debug cores.

## Customizing and Generating the Debug Cores

Use the **IP Catalog** button in the **Project Manager** to locate, select, and customize the desired debug core. The debug cores are located in the **Debug & Verification > Debug** category of the IP Catalog (see [Figure 4-7](#)). You can customize the debug core by double-clicking on the IP core or by right-click selecting the **Customize IP** menu selection.

- For more information on customizing the ILA core, refer to *LogiCORE IP Integrated Logic Analyzer (ILA) v3.0 Datasheet* (DS875) [\[Ref 7\]](#).
- For more information on customizing the VIO core, refer to *LogiCORE IP Virtual Input/Output (VIO) v3.0 Product Guide* (PG159) [\[Ref 8\]](#).
- For more information on customizing the JTAG-to-AXI Master core, refer to *LogiCORE IP JTAG to AXI Master v1.0 Product Guide* (PG174) [\[Ref 12\]](#).

After customizing the core, click the **Generate** button in the IP customization wizard. This generates the customized debug core and add it to the **Sources** view of your project.

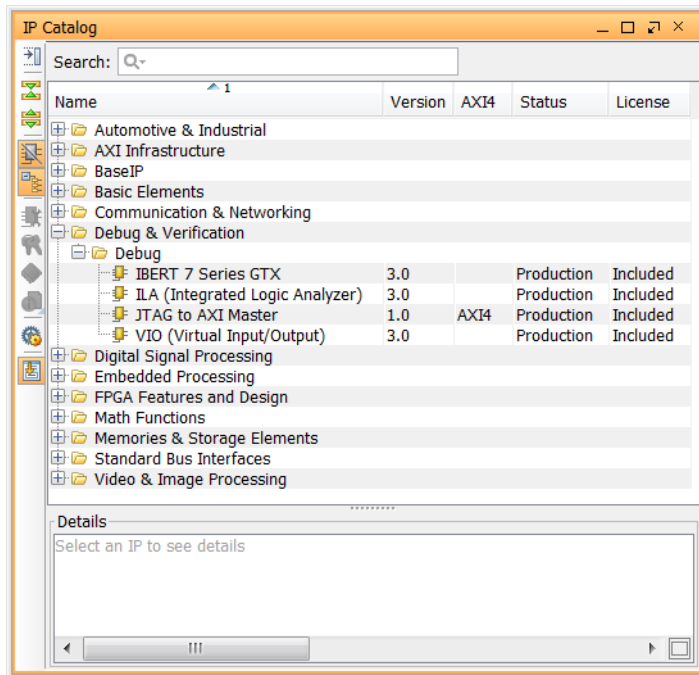


Figure 4-7: Debug Cores in the IP Catalog

## Instantiating the Debug Cores

After generating the debug core, instantiate it in your HDL source code and connect it to the signals that you wish to probe for debugging purposes. Following is an example of the ILA instance in a Verilog HDL source file:

```
ila_v3_0 i_ila
(
    .clk(clk),
    .probe0(counterA),
    .probe1(counterB),
    .probe2(counterC),
    .probe3(counterD),
    .probe4(A_or_B),
    .probe5(B_or_C),
    .probe6(C_or_D),
    .probe(D_or_A)
);
```

**Note:** Unlike the legacy VIO and ILA v1.x cores, the new ILA core instance does not require a connection to an ICON core instance. Instead, a debug\_core\_hub debug core is automatically inserted into the synthesized design netlist to provide connectivity between the new ILA core and the JTAG scan chain.

## Synthesizing the Design Containing the Debug Cores

In the next step, synthesize the design containing the debug cores by clicking **Run Synthesis** in the Vivado IDE or by running the following Tcl commands:

```
launch_runs synth_1
wait_on_run synth_1
```

You can also use the `synth_design` Tcl command to synthesize the design. Refer to *Vivado Design Suite User Guide: Synthesis (UG901)* [Ref 2] for more details on the various ways you can synthesize your design.

## Viewing the Debug Cores in the Synthesized Design

After synthesizing your design, you can open the synthesized design to view the debug cores and modify their properties. Open the synthesized design by clicking **Open Synthesized Design** in the **Flow Navigator** and select the **Debug** window layout to see the **Debug** window that shows your ILA debug cores connected to the debug hub core (`dbg_hub`) (see Figure 4-8).

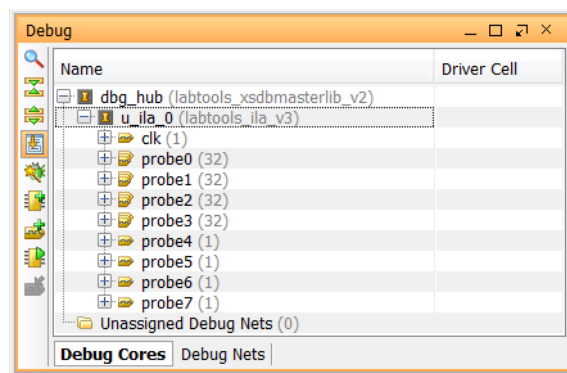


Figure 4-8: Debug Window Showing ILA Core and Debug Core Hub



## Changing the BSCAN User Scan Chain of the Debug Core Hub

You can view and change the BSCAN user scan chain index of the debug core hub by selecting the **dbg\_hub** in the **Debug** window, selecting the **Debug Core Options** view in the **Properties** window, then changing the value of the **C\_USER\_SCAN\_CHAIN** property (see Figure 4-9).



**IMPORTANT:** If you plan to mix legacy ICON, ILA, and/or VIO v1.x cores with new ILA cores, you need to set the **C\_USER\_SCAN\_CHAIN** property of the **debug\_core\_hub** to a user scan chain that does not conflict with the ICON v1.x core's Boundary Scan Chain setting. Failure to do so results in errors later in the implementation flow.

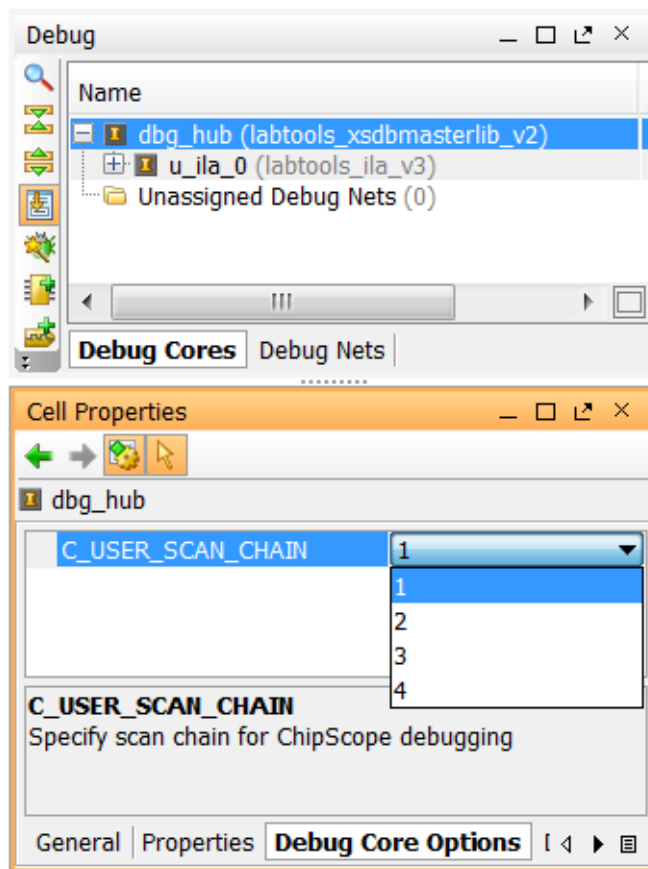


Figure 4-9: Changing the User Scan Chain Property of the Debug Core Hub

---

# Implementing the Design Containing the Debug Cores

The Vivado software creates the debug core hub initially a black box. This core must be implemented prior to running the placer and router.

## Implementing the Debug Cores

Debug core implementation is done automatically when implementing the design; however, you can also force debug core implementation manually for floorplanning or timing analysis. To manually implement the cores, do one of the following:

- Click the **Implement Debug Cores** icon on the toolbar menu of the **Debug** window.
- Right-click on any debug core in the **Debug** window and select the **Implement Debug Cores** option from the popup menu.

The Vivado IDE generates and synthesizes each black box debug core. This operation can take some time. A progress indicator shows that the operation is running. When the debug core implementation is complete, the debug core black boxes are resolved and you can access the generated instances.

## Implementing the Design

Implement the design containing the debug core by clicking **Run Implementation** in the Vivado IDE or by running the following Tcl commands:

```
launch_runs impl_1  
wait_on_run impl_1
```

You can also implement the design using the implementation commands `opt_design`, `place_design`, and `route_design`. Refer to the *Vivado Design Suite User Guide: Implementation (UG904)* [Ref 3] for more details on the various ways you can implement your design.

# Debugging Logic Designs in Hardware

---

## Introduction

Once you have the debug cores in your design, you can use the run time logic analyzer features to debug the design in hardware. Two different tools can be used depending on the type of debug cores in your design:

- ChipScope™ Pro Analyzer: used with ICON v1.x, ILA v1.x, VIO v1.x, and IBERT v2.x debug cores.
- Vivado® logic analyzer feature: used with new ILA v3.x, VIO v3.x, JTAG-to-AXI Master, and IBERT 7 Series GTH/GTP/GTX/GTZ v3.x debug cores.

If you have a mixture of ICON/ILA/VIO v1.x and new ILA/VIO/JTAG-to-AXI Master debug cores in your design, you can simultaneously use both the ChipScope Pro Analyzer tool and Vivado logic analyzer feature to debug the same design running on the same hardware target board (see section called [Using Vivado Logic Analyzer to Debug the Design, page 44](#) for more details).

---

## Launching ChipScope Pro Analyzer to Debug the Design

The ChipScope Pro Analyzer tool is used to interact with ICON v1.x, ILA v1.x, and VIO v1.x debug cores that are in your design. When the ChipScope Pro Analyzer software is installed, you can launch it directly from the Vivado IDE on any implemented design on which Generate Bitstream has been run.



**IMPORTANT:** *The ChipScope Pro and iMPACT tools are not included in the Vivado Design Suite installation. However, these tools are available in an installation package called the ISE Design Suite: Standalone Lab Tools that is freely downloadable from the Xilinx Download site at <http://www.xilinx.com/downloads>.*

---

To launch ChipScope Pro Analyzer, do one of the following:

- Use the **Flow > Launch ChipScope Analyzer** command from the main menu
- Run the `launch_chipscope_analyzer` Tcl command in the Tcl Console

The Vivado IDE passes the BIT bitstream and CDC net connection name files automatically to the ChipScope Pro Analyzer tool. For more information about ChipScope Pro Analyzer see the Xilinx website, [http://www.xilinx.com/support/index.html/content/xilinx/en/supportNav/design\\_tools/chipscope\\_pro.html](http://www.xilinx.com/support/index.html/content/xilinx/en/supportNav/design_tools/chipscope_pro.html)

[http://www.xilinx.com/support/index.html/content/xilinx/en/supportNav/design\\_tools/chipscope\\_pro.html](http://www.xilinx.com/support/index.html/content/xilinx/en/supportNav/design_tools/chipscope_pro.html)

---

## Using Vivado Logic Analyzer to Debug the Design

The Vivado logic analyzer feature is used to interact with new ILA, VIO, and JTAG-to-AXI Master debug cores that are in your design. To access the Vivado logic analyzer feature, click the **Open Hardware Manager** button in the **Program and Debug** section of the **Flow Navigator**.

The steps to debug your design in hardware using an ILA debug core are:

1. Connect to the hardware target and program the FPGA device with the `.bit` file
2. Set up the ILA debug core trigger and capture controls.
3. Arm the ILA debug core trigger.
4. View the captured data from the ILA debug core in the **Waveform** window.
5. Use the VIO debug core to drive control signals and/or view design status signals.
6. Use the JTAG-to-AXI Master debug core to run transactions to interact with various AXI slave cores in your design.

---

## Connecting to the Hardware Target and Programming the FPGA Device

Programming an FPGA device prior to debugging are exactly the same steps described in [Using a Vivado Hardware Manager to Program an FPGA Device in Chapter 2](#). After programming the device with the `.bit` file that contains the new ILA, VIO, and JTAG-to-AXI Master debug cores, the **Hardware** window now shows the ILA and VIO cores

that were detected when scanning the device (see [Figure 5-1](#)).

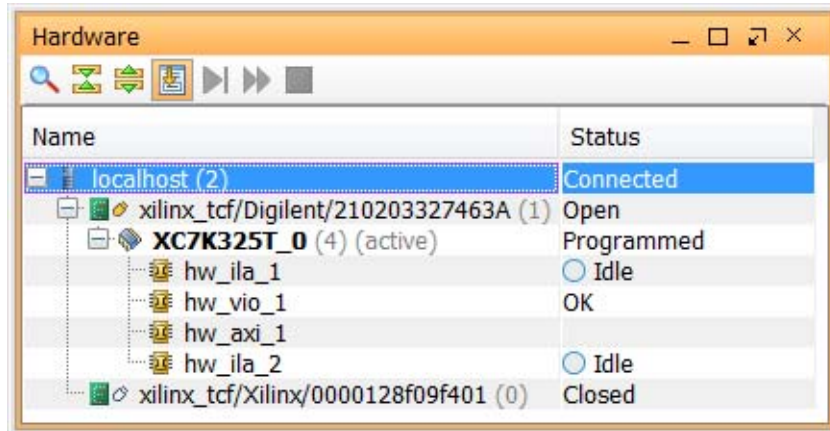


Figure 5-1: Hardware Window Showing the ILA and VIO Debug Cores

For more information on using the ILA core, refer to [Setting up the ILA Core to Take a Measurement](#), page 45. For more information on using the VIO core, refer to [Setting up the ILA Core to Take a Measurement](#).

## Setting up the ILA Core to Take a Measurement

The ILA cores that you add to your design appear in the **Hardware** window under the target device. If you do not see the ILA cores appear, right-click on the device and select **Refresh Hardware**. This re-scans the FPGA device and refreshes the **Hardware** window.

**Note:** If you still do not see the ILA core after programming and/or refreshing the FPGA device, check to make sure the device was programmed with the appropriate `.bit` file and check to make sure the implemented design contains an ILA core. Also, check to make sure the appropriate `.ltx` probes file that matches the `.bit` file is associated with the device.

Click the ILA core (called `hw_ila_1` in [Figure 5-1](#)) to see its properties in the **ILA Core Properties** window. By selecting the ILA core, you should also see the probes corresponding to the ILA core in the **Debug Probes** window as well as the corresponding **ILA Dashboard** in the Vivado IDE workspace (see, [Selection of the ILA Core in Various Views](#) [Figure 5-2](#), page 46).

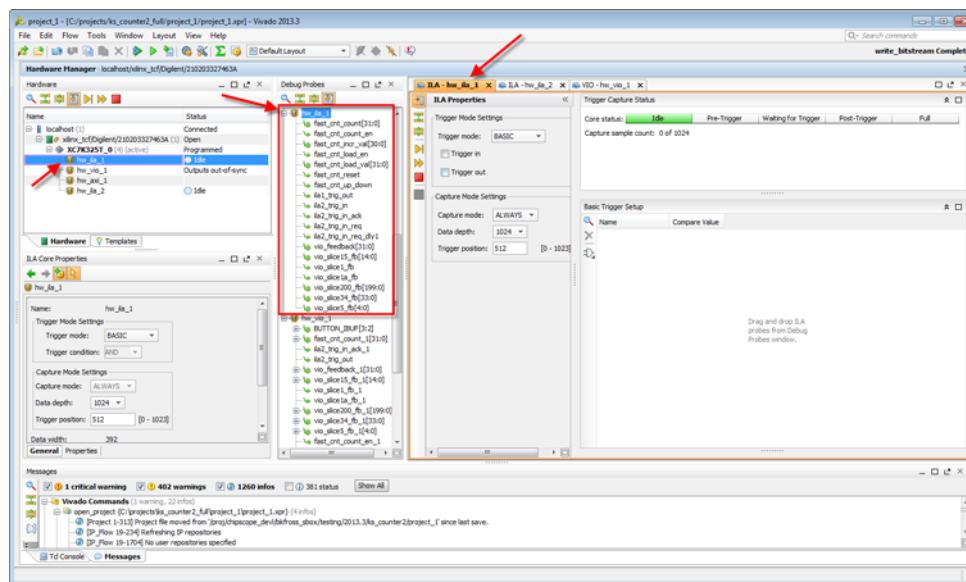


Figure 5-2: Selection of the ILA Core in Various Views

## Viewing ILA Cores in the Debug Probes Window

The **Debug Probes** window is used to view all of the debug probes that belong to an ILA or VIO core (see [Figure 5-3, page 47](#)). The ILA debug probes can be added to an existing waveform viewer for the ILA core or can be added to the various trigger and/or capture windows of the **ILA Dashboard**. To perform these operations, right-click on an ILA core's debug probe(s) and select one of the following:

- **Add Probes to Waveform:** adds selected probes to the waveform window corresponding to the ILA core to which the probe belongs.
- **Add Probes to Basic Trigger Setup:** adds selected probes to the **Basic Trigger Setup** window of the dashboard corresponding to the ILA core to which the probe belongs. Note that the ILA core's **Trigger mode** should be set to "BASIC" for this selection to be enabled.
- **Add Probes to Basic Capture Setup:** adds selected probes to the **Basic Capture Setup** window of the dashboard corresponding to the ILA core to which the probe belongs. Note that the ILA core's **Capture mode** should be set to "BASIC" for this selection to be enabled.



**TIP:** *Tip: if you right-click on the ILA core object in the **Debug Probes** or **Hardware** window and select one of the Add Probes... options, the selection option will apply to all probes that belong to that ILA core.*

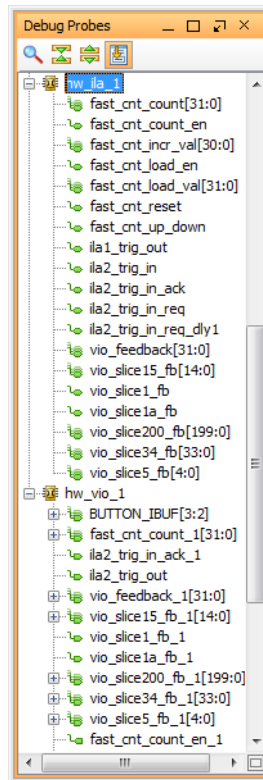


Figure 5-3: ILA Debug Probes

## Writing Debug Probes Information

The **Debug Probes** window contains information about the nets that you probed in your design using the ILA and/or VIO cores. This debug probe information is extracted from your design and is stored in a data file that typically has an `.ltx` file extension.

Normally, the debug probes file is automatically created during the implementation process. However, you can also use the `write_debug_probes` Tcl command to write out the debug probes information to a file:

1. Open the Synthesized or Netlist Design.
2. Run the `write_debug_probes filename.ltx` Tcl command.

## Reading Debug Probes Information

The debug probes file is automatically associated with the hardware device if the Vivado IDE is in project mode and a probes file called `debug_nets.ltx` is found in the same directory as the bitstream programming (`.bit`) file that is associated with the device.

You can also specify the location of the probes file:

1. Select the hardware device in the **Hardware** window.
2. Set the Probes file location in the **Hardware Device Properties** window.
3. Right-click the hardware device in the **Hardware** window and select **Refresh Device** to read the contents of the debug probes file and associate and validate the information with the debug cores found in the design running in the hardware device.

You can also set the location using the following Tcl commands to associate a debug probes file called `C:\myprobes.ltx` with the first device on the target board:

```
% set_property PROBES.FILE {C:/myprobes.ltx} [lindex [get_hw_devices] 0]  
% refresh_hw_device [lindex [get_hw_devices] 0]
```

## Using the ILA Dashboard

The **ILA Dashboard** (see [Figure 5-4, page 48](#)) is a central location for all status and control information pertaining to a given ILA core. When an ILA core is first detected upon refreshing a hardware device, the **ILA Dashboard** for the core is automatically opened. If you need to manually open or re-open the dashboard, just right-click the ILA core object in either the **Hardware** or **Debug Probes** windows and select **Open Dashboard**.

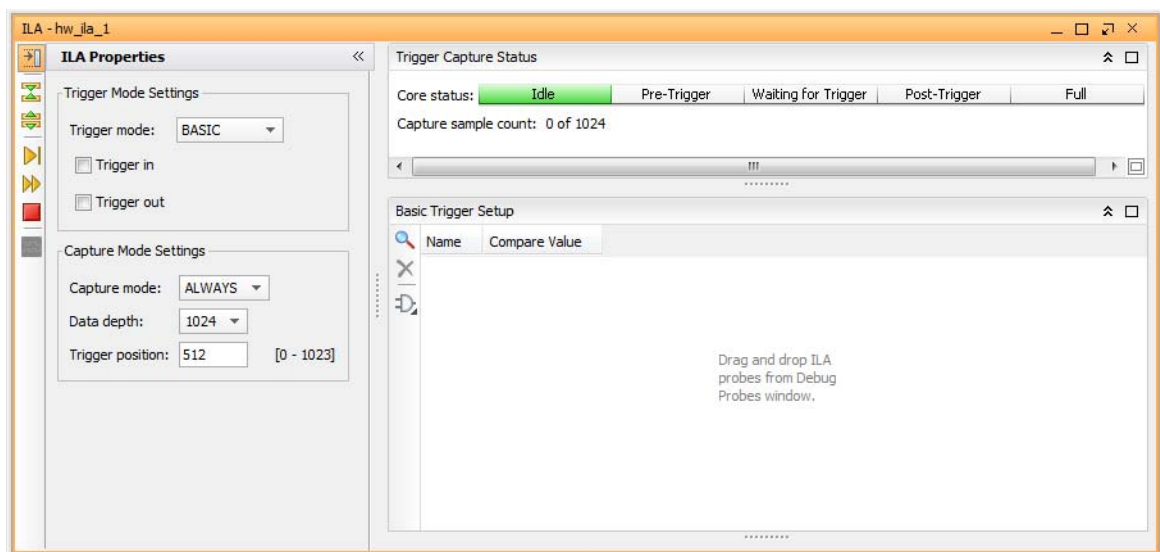


Figure 5-4: ILA Dashboard

You can use the **ILA Dashboard** to interact with the ILA debug core in several ways:

- Use BASIC and ADVANCED trigger modes to trigger on various events in hardware.
- Enable or disable trigger input and output behavior.
- Use ALWAYS and BASIC capture modes to control filtering of data to be captured.



- Set the data depth of the ILA capture window.
- Set the trigger position to any sample within the capture window.
- Monitor the trigger and capture status of the ILA debug core.

## Using Basic Trigger Mode

The basic trigger mode is used to describe a trigger condition that is a global Boolean equation of participating debug probe comparators. The **Basic Trigger Setup** window (see [Figure 5-5](#)) is used to create this trigger condition and debug probe compare values.

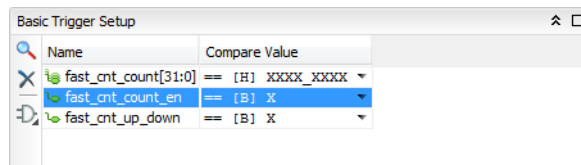


Figure 5-5: ILA Basic Trigger Setup Window

You can also use the `set_property` Tcl command to change the trigger mode of the ILA core. For instance, to change the trigger mode of ILA core `hw_ila_1` to BASIC, use the following command:

```
set_property CONTROL.TRIGGER_MODE BASIC [get_hw_ilas hw_ila_1]
```

## Adding Probes to Basic Trigger Setup Window

The first step in using the BASIC trigger mode is to decide what ILA debug probes you want to participate in the trigger condition. Do this by selecting the desired ILA debug probes from the **Debug Probes** window and either right-click selecting **Add Probes to Basic Trigger Setup** or by dragging and dropping the probes into the **Basic Trigger Setup** window.

**Note:** You can drag-and-drop the first probe anywhere in the **Basic Trigger Setup** window, but you must drop the second and subsequent probes on top of the first probe. The new probe is always added above the previously added probe in the table. You can also use drag-and-drop operations in this manner to re-arrange probes in the table.



**IMPORTANT:** Only probes that are in the **Basic Trigger Setup** window participate in the trigger condition. Any probes that are not in the window are set to "don't care" values and are not used as part of the trigger condition.

You can remove probes from the **Basic Trigger Setup** window by selecting the probe and hitting the **Delete** key or by right-click selecting the **Remove** option.

## Setting Basic Trigger Compare Values

The ILA debug probe trigger comparators are used to detect specific equality or inequality conditions on the probe inputs to the ILA core. The trigger condition is the result of a Boolean "AND", "OR", "NAND", or "NOR" calculation of each of the ILA probe trigger comparator results. To specify the compare values for a given ILA probe, select the Compare Value cell in for a given ILA debug probe in the **Basic Trigger Setup** window to open the **Compare Value** dialog box (see [Figure 5-6](#)).

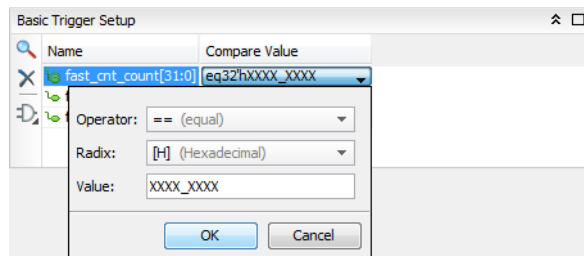


Figure 5-6: ILA Probe Compare Value Dialog Box

## ILA Probe Compare Value Settings

The **Compare Value** dialog box contains three fields that you can configure:

1. **Operator:** This is the comparison operator that you can set to the following values:
  - == (equal)
  - != (not equal)
  - < (less than)
  - <= (less than or equal)
  - > (greater than)
  - >= (greater than or equal)

2. **Radix:** This is the radix or base of the Value that you can set to the following values:
  - "[B] Binary
  - [H] Hexadecimal
  - [O] Octal
  - [A] ASCII
  - [U] Unsigned Decimal
  - [S] Signed Decimal
3. **Value:** This is the comparison value that will be compared (using the **Operator**) with the real-time value on the net(s) in the design that are connected to the probe input of the ILA debug core. Depending on the Radix settings, the Value string is as follows:
  - Binary
    - 0: logical zero
    - 1: logical one
    - X: don't care
    - R: rising or low-to-high transition
    - F: falling or high-to-low transition
    - B: either low-to-high or high-to-low transitions
    - N: no transition (current sample value is the same as the previous value)
  - Hexadecimal
    - X: All bits corresponding to the value string character are "don't care" values
    - 0-9: Values 0 through 9
    - A-F: Values 10 through 15
  - Octal
    - X: All bits corresponding to the value string character are "don't care" values
    - 0-7: Values 0 through 7
  - ASCII
    - Any string made up of ASCII characters
  - Unsigned Decimal
    - Any non-negative integer value
  - Signed Decimal
    - Any integer value

## Setting Basic Trigger Condition

You can set up the trigger condition using the toolbar button on the left side of the **Basic Trigger Setup** window that has an icon the shape of a logic gate on it (see [Figure 5-7](#)). You can also use the `set_property Tcl` command to change the trigger condition of the ILA core:

```
set_property CONTROL.TRIGGER_CONDITION AND [get_hw_ilas hw_ila_1]
```

The meaning of the four possible values is shown in [Table 5-1](#).

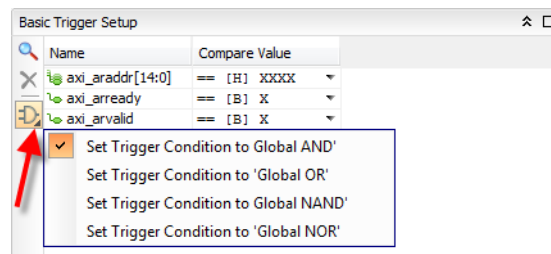


Figure 5-7: Setting the Basic Trigger Condition

Table 5-1: Basic Trigger Condition Setting Descriptions

Trigger Condition Setting in GUI	CONTROL.TRIGGER_CONDITION property value	Trigger Condition Output
Global AND	AND	Trigger condition is "true" if all participating probe comparators evaluate "true", otherwise trigger condition is "false."
Global OR	OR	Trigger condition is "true" if at least one participating probe comparator evaluates "true", otherwise trigger condition is "false."
Global NAND	NAND	Trigger condition is "true" if at least one participating probe comparator evaluates "false", otherwise trigger condition is "false."
Global NOR	NOR	Trigger condition is "true" if all participating probe comparators evaluate "false", otherwise trigger condition is "false."



**IMPORTANT:** If the ILA core has two or more debug probes that concatenated together to share a single physical probe port of the ILA core, then only the "Global AND" (AND) and "Global NAND" (NAND) trigger condition settings are supported. The "Global OR" (OR) and "Global NOR" (NOR) functions are not supported due to limitations of the probe port comparator logic. If you want to use the "Global OR"

(OR) or "Global NOR" (NOR) trigger condition settings, then make sure you assign each unique net or bus net to separate probe ports of the ILA core.

---

## Using Advanced Trigger Mode

The ILA core can be configured at core generation or insertion time to have advanced trigger capabilities that include the following features:

- Trigger state machine consisting of up to 16 states
- Each state can consist of one-, two-, or three-way conditional branching
- Up to four counters can be used in a trigger state machine program to keep track of multiple events
- Up to four flags can be used in a trigger state machine program to indicate when certain branches are taken
- The state machine can execute "goto", "trigger", and various counter- and flag-related actions

If the ILA core in the design that is running in the hardware device has advanced trigger capabilities, the advanced trigger mode features can be enabled by setting the **Trigger mode** control in the **ILA Properties** window of the **ILA Dashboard** to **ADVANCED**.

## Specifying the Trigger State Machine Program File

When you set the **Trigger mode** to **ADVANCED**, two changes happen in the **ILA Dashboard**:

1. A new control called **Trigger State Machine** appears in the **ILA Properties** window
2. The **Basic Trigger Setup** window is replaced by a **Trigger State Machine** code editor window.

If you are specifying an ILA trigger state machine program for the first time, the **Trigger State Machine** code editor window will appear as the one shown in [Figure 5-8](#).

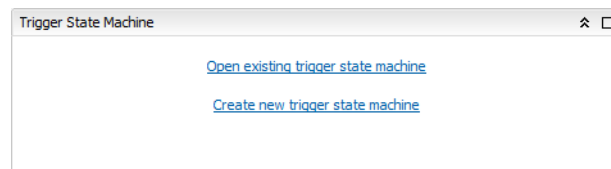


Figure 5-8: Creating or Opening a Trigger State Machine Program File

To create a new trigger state machine, click the **Create new trigger state machine** link, otherwise click the **Open existing trigger state machine** link to open a trigger state machine program file (.tsm extension). You can also open an existing trigger state machine program file using the **Trigger state machine** text field and/or browse button in the **ILA Properties** window of the **ILA Dashboard**.

## Editing the Trigger State Machine Program

If you created a new trigger state machine program file, the **Trigger State Machine** code editor window will be populated with a simple trigger state machine by default (see [Figure 5-9](#)).

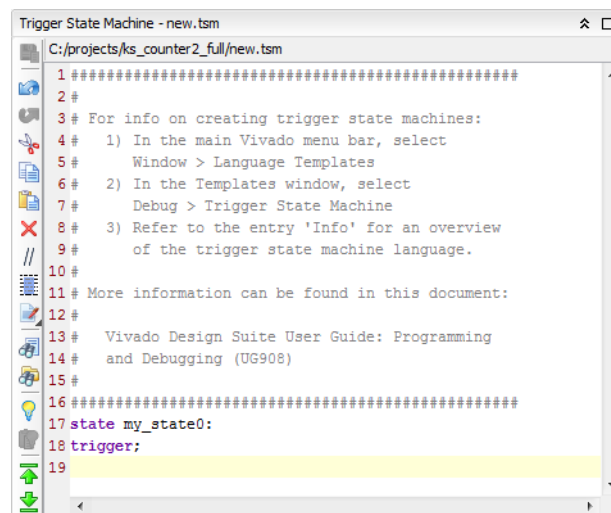


Figure 5-9: Simple Default Trigger State Machine Program

The simple default trigger state machine program is designed to be valid for any ILA core configuration regardless of debug probe or trigger settings. This means that you can click the **Run Trigger** for the ILA core without modifying the trigger state machine program.

However, it is likely that you will want to modify the trigger state machine program to implement some advanced trigger condition. The comment block at the top of the simple state machine shown in [Figure 5-9](#) gives some instructions on how to use the built-in language templates in the Vivado IDE to construct a trigger state machine program (see [Figure 5-10](#), page 55). A full description of the ILA trigger state machine language, including examples, is found in the section of this document called [Appendix B, Trigger State Machine Language Description](#).

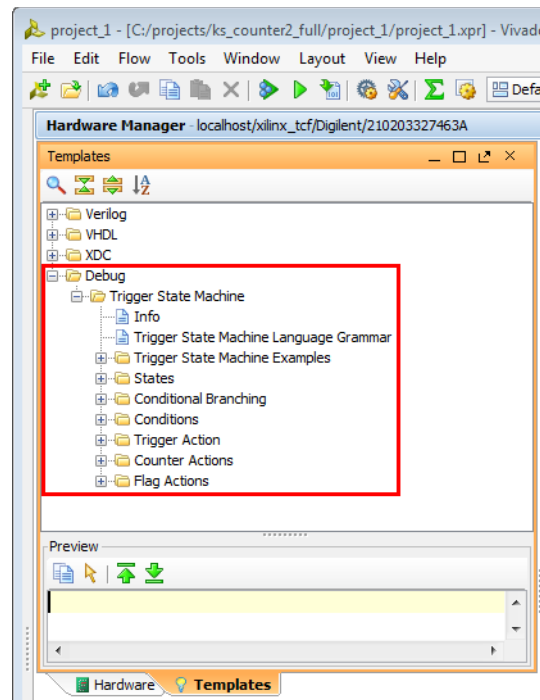


Figure 5-10: Trigger State Machine Language Templates

## Enabling Trigger In and Out Ports

The ILA core can be configured at core generation-time to have dedicated trigger input ports (TRIG\_IN and TRIG\_IN\_ACK) and dedicated trigger output ports (TRIG\_OUT and TRIG\_OUT\_ACK). If the ILA core detected in the design that is running in the hardware device has trigger input and/or output ports, then you can use the **Trigger In** and/or **Trigger Out** controls in the **ILA Properties** window of the **ILA Dashboard** to enable or disable these ports from participating in the overall trigger operation.

## Configuring Capture Mode Settings

The ILA core can capture data samples when the core status is Pre-Trigger, Waiting for Trigger, or Post-Trigger (refer to the section called [Viewing Trigger and Capture Status](#), [page 58](#) for more details). The **Capture mode** control is used to select what condition is evaluated before each sample is captured:

- **ALWAYS:** store a data sample during a given clock cycle regardless of any capture conditions
- **BASIC:** store a data sample during a given clock cycle only if the capture condition evaluates "true"

You can also use the `set_property` Tcl command to change the capture mode of the ILA core. For instance, to change the capture mode of ILA core `hw_ila_1` to BASIC, use the following command:

```
set_property CONTROL.CAPTURE_MODE BASIC [get_hw_ilas hw_ila_1]
```

## Using BASIC Capture Mode

The basic capture mode is used to describe a capture condition that is a global Boolean equation of participating debug probe comparators. The **Basic Capture Setup** window (see [Figure 5-11](#)) is used to create this capture condition and debug probe compare values.

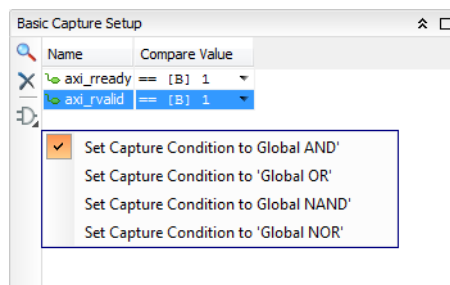


Figure 5-11: Setting the Basic Capture Condition

## Configuring the Basic Capture Setup Window

The process for configuring debug probes and basic capture condition in the **Basic Capture Setup** window is very similar to working with debug probes in the **Basic Trigger Setup** window:

- For information on adding probes to the **Basic Capture Setup** window, refer to the section called [Adding Probes to Basic Trigger Setup Window](#), page 49.
- For information on setting the compare values on each probe in the **Basic Capture Setup** window, refer to the section called [ILA Probe Compare Value Settings](#), page 50
- For information on setting the basic capture condition in the **Basic Capture Setup** window, refer to the section called [Setting Basic Trigger Condition](#), page 52. One key difference is the ILA core property used to control the capture condition is called `CONTROL.CAPTURE_CONDITION`.

## Setting the Trigger Position in the Capture Window

Use the **Trigger position** control in the **Capture Mode Settings** window (or the **Trigger Position** property in the **ILA Core Properties** window) to set the position of the trigger mark in the captured data buffer. You can set the trigger position to any sample number in the captured data buffer. For instance, in the case of a captured data buffer that is 1024 samples deep:



- Sample number 0 corresponds to the first (left-most) sample in the captured data buffer.
- Sample number 1023 corresponds to the last (right-most) sample in the captured data buffer.
- Samples numbers 511 and 512 correspond to the two "center" samples in the captured data buffer.

You can also use the `set_property` Tcl command to change the ILA core trigger position:

```
set_property CONTROL.TRIGGER_POSITION 512 [get_hw_ilas hw_ila_1]
```

## Setting the Data Depth of the Capture Window

Use the Data Depth control in the Capture Mode Settings window (or the Capture data depth property in the ILA Core Properties window) to set the data depth of the ILA core's captured data window. You can set the data depth to any power of two from 1 to the maximum data depth of the ILA core (set during core generation or insertion time).

**Note:** Refer to the section called [Modifying Properties on the Debug Cores, page 32](#) for more details on how to set the maximum capture buffer data depth on ILA cores that are added to the design using the Netlist Insertion probing flow.

You can also use the `set_property` Tcl command to change the ILA core data depth:

```
set_property CONTROL.DATA_DEPTH 512 [get_hw_ilas hw_ila_1]
```

## Running the Trigger

You can run or arm the ILA core trigger in two different modes:

- **Run Trigger:** Selecting the ILA core(s) to be armed followed by clicking the **Run Trigger** button on the **ILA Dashboard** or **Hardware** window toolbar arms the ILA core to detect the trigger event that is defined by the ILA core basic or advanced trigger settings.
- **Run Trigger Immediate:** Selecting the ILA core(s) to be armed followed by clicking the **Run Trigger Immediate** button on the **ILA Dashboard** or **Hardware** window toolbar arms the ILA core to trigger immediately regardless of the ILA core trigger settings. This command is useful for detecting the "aliveness" of the design by capturing any activity at the probe inputs of the ILA core.

You can also arm the trigger by selecting and right-clicking on the ILA core and selecting **Run Trigger** or **Run Trigger Immediate** from the popup menu (see [Figure 5-12](#)).

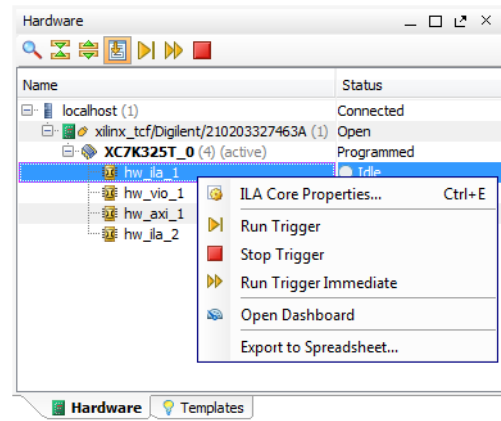


Figure 5-12: ILA Core Trigger Commands



**TIP:** You can run or stop the triggers of multiple ILA cores by selecting the desired ILA cores, then using the **Run Trigger**, **Run Trigger Immediate**, or **Stop Trigger** buttons in the **Hardware** window toolbar. You can also run or stop the triggers of all ILA cores in a given device by selecting the device in the **Hardware** window and click the appropriate button in the **Hardware** window toolbar.

## Stopping the Trigger

You can stop the ILA core trigger by selecting the appropriate ILA core followed by clicking on the Stop Trigger button on the **ILA Dashboard** or **Hardware** window toolbar. You can also stop the trigger by selecting and right-clicking on the appropriate ILA core(s) and selecting **Stop Trigger** from the popup menu (see Figure 5-12).

## Viewing Trigger and Capture Status

The ILA debug core trigger and capture status is displayed in two places in the Vivado IDE:

- In the **Hardware** window **Status** column of the row(s) corresponding to the ILA debug core(s).
- In the **Trigger Capture Status** window of the **ILA Dashboard**.

The Status column in the **Hardware** window indicates the current state or status of each ILA core (see Table 5-2, page 59).

Table 5-2: ILA Core Status Description

ILA Core Status	Description
Idle	The ILA core is idle and waiting for its trigger to be run. If the trigger position is 0, then the ILA core will transition to the Waiting for Trigger status once the trigger is run, otherwise the ILA core will transition to the Pre-Trigger status.
Pre-Trigger	The ILA core is capturing pre-trigger data into its data capture buffer. Once the pre-trigger data has been captured, the ILA core will transition to the Waiting for Trigger status.
Waiting for Trigger	The ILA core trigger is armed and is waiting for the trigger event to occur as described by the basic or advanced trigger settings. Once the trigger occurs, the ILA core will transition to the Full status if the trigger position is set to the last data sample in the capture window, otherwise it will transition to the Post-Trigger status.
Post-Trigger	The ILA core is capturing post-trigger data into its data capture buffer. Once the post-trigger data has been captured, the ILA core will transition to the Full status.
Full	The ILA core capture buffer is full and is being uploaded to the host for display. The ILA core will transition to the Idle status once the data has been uploaded and displayed.

The contents of the **Trigger Capture Status** window in the **ILA Dashboard** depend on the **Trigger Mode** setting of the ILA core.

## Basic Trigger Mode Trigger and Capture Status

The **Trigger Capture Status** window contains two status indicators when the **Trigger Mode** is set to BASIC (see [Figure 5-13](#)):

- **Core status:** indicates the status of the ILA core trigger/capture engine (see [Table 5-2](#) for a description of the status indicators)
- **Capture sample count:** indicates the current number of samples captured by the ILA core. This number is reset to 0 once the ILA core status is Idle.

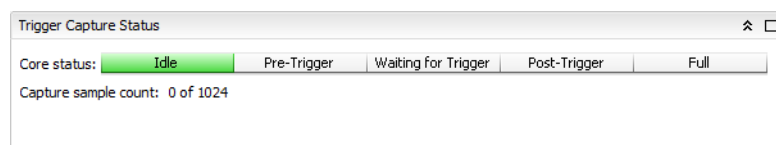


Figure 5-13: Basic Trigger Mode Trigger Capture Status Window

## Advanced Trigger Mode Trigger and Capture Status

The **Trigger Capture Status** window contains four status indicators when the **Trigger Mode** is set to ADVANCED (see [Figure 5-14](#)):

- **Core status:** indicates the status of the ILA core trigger/capture engine (see [Table 5-2](#), [page 59](#) for a description of the status indicators)
- **Trigger State Machine Flags:** indicates the current state of the four trigger state machine flags.
- **Trigger State:** when the core status is Waiting for Trigger, this field indicates the current state of the trigger state machine.
- **Capture sample count:** indicates the current number of samples captured by the ILA core. This number is reset to 0 once the ILA core status is Idle.

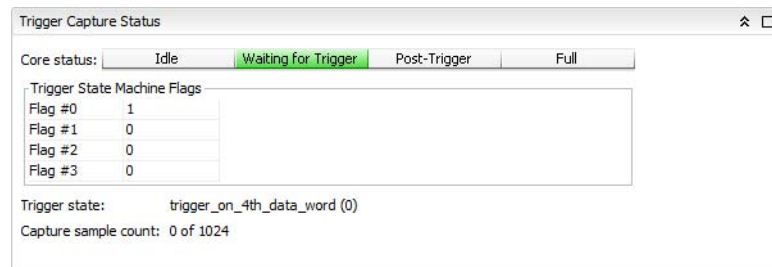


Figure 5-14: Advanced Trigger Mode Trigger Capture Status Window

---

## Writing ILA Probes Information

The **ILA Cores** tab view in the **Debug Probes** window contains information about the nets that you probed in your design using the ILA core. This ILA probe information is extracted from your design and is stored in a data file that typically has an `.ltx` file extension.

Normally, the ILA probe file is automatically created during the implementation process. However, you can also use the `write_debug_probes` Tcl command to write out the debug probes information to a file:

1. Open the Synthesized or Netlist Design.
2. Run the `write_debug_probes filename.ltx` Tcl command.

---

## Reading ILA Probes Information

The ILA probe file is automatically associated with the FPGA hardware device if the probes file is called `debug_nets.ltx` and is found in the same directory as the bitstream programming (`.bit`) file that is associated with the device.

You can also specify the location of the probes file:

1. Select the FPGA device in the **Hardware** window.
2. Set the **Probes file** location in the **Hardware Device Properties** window.
3. Click **Apply** to apply the change.

You can also set the location using the `set_property` Tcl command:

```
set_property PROBES.FILE {C:/myprobes.ltx} [lindex [get_hw_devices] 0]
```

---

## Viewing Captured Data from the ILA Core in the Waveform Viewer

Once the ILA core captured data has been uploaded to the Vivado IDE, it is displayed in the **Waveform Viewer**. See [Chapter 6, Viewing ILA Probe Data Using Waveform Viewer](#) for details on using the Waveform Viewer to view captured data from the ILA core.

---

## Saving and Restoring Captured Data from the ILA Core

In addition to displaying the captured data that is directly uploaded from the ILA core, you can also write the captured data to a file then read the data from a file and display it in the waveform viewer.

### Saving Captured ILA Data to a File

Currently, the only way to upload captured data from an ILA core and save it to a file is to use the following Tcl command:

```
write_hw_ila_data my_hw_ila_data_file.zip [upload_hw_ila_data hw_ila_1]
```

This Tcl command sequence uploads the captured data from the ILA core and writes it to an archive file called `my_hw_ila_data_file.zip`. The archive file contains the waveform

database file, the waveform configuration file, a waveform comma separated value file, and a debug probes file.

## Restoring Captured ILA Data from a File

Currently, the only way to restore captured data from a file and display it in the waveform viewer is to use the following Tcl command:

```
display_hw_ila_data [read_hw_ila_data my_hw_ila_data_file.zip]
```

This Tcl command sequence reads the previously saved captured data from the ILA core and displays it in the waveform window.

**Note:** The waveform configuration settings (dividers, markers, colors, probe radices, etc.) for the ILA data waveform window is also saved in the ILA captured data archive file. Restoring and displaying any previously saved ILA data uses these stored waveform configuration settings.

---

## Setting Up the VIO Core to Take a Measurement

The VIO cores that you add to your design appear in the **Hardware** window under the target device. If you do not see the VIO cores appear, right-click the device and select **Refresh Hardware**. This re-scans the FPGA device and refreshes the **Hardware** window.

**Note:** If you still do not see the VIO core after programming and/or refreshing the FPGA device, check to make sure the device was programmed with the appropriate .bit file and check to make sure the implemented design contains an VIO core. Also, check to make sure the appropriate .ltx probes file that matches the .bit file is associated with the device.

Click the VIO core (called hw\_vio\_1 in Figure 5-15) to see its properties in the **VIO Core Properties** window. By selecting the VIO core, you should also see the probes corresponding to the VIO core in the **Debug Probes** window as well as the corresponding **VIO Dashboard** in the **Vivado IDE** workspace (see Figure 5-16).

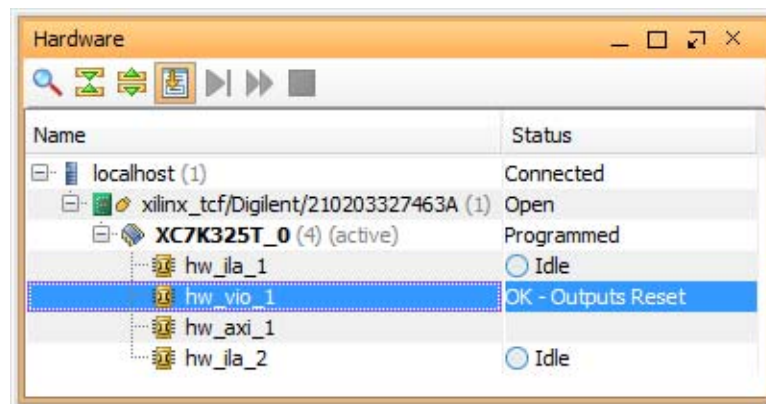


Figure 5-15: VIO Core in the Hardware Window

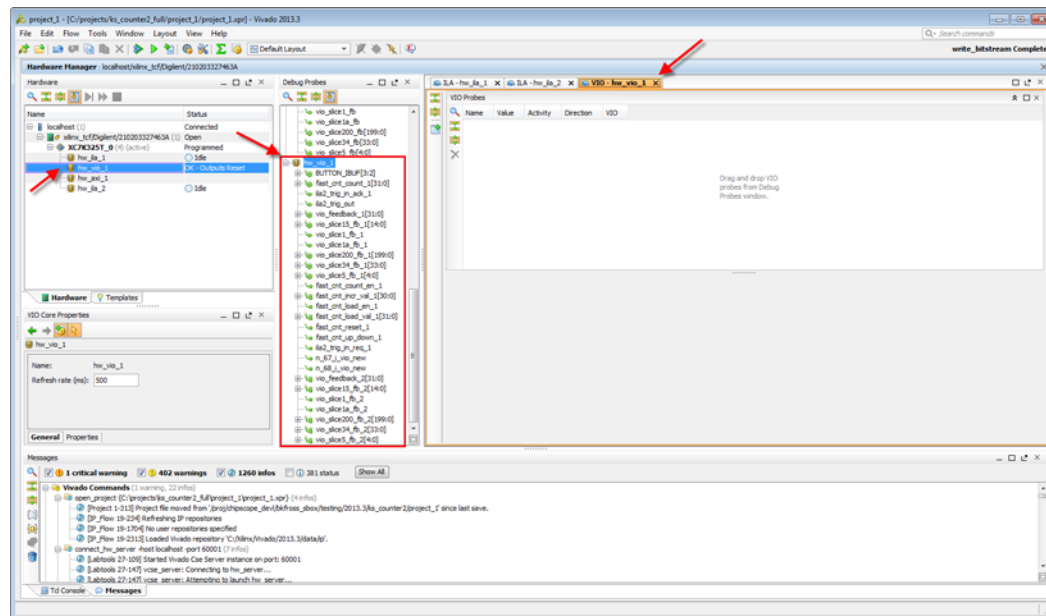


Figure 5-16: Selection of the VIO Core in Various Views

The VIO core can become out-of-sync with the Vivado IDE. Refer to [Viewing the VIO Core Status](#), page 63 for more information on how to interpret the VIO status indicators.

The VIO core operates on an object property-based set/commit and refresh/get model:

- To read VIO input probe values, first refresh the hw\_vio object with the VIO core values. Observe the input probe values by getting the property values of the corresponding hw\_probe object. Refer to the section called [Interacting with VIO Core Input Probes](#), page 66 for more information.
- To write VIO output probe values, first set the desired value as a property on the hw\_probe object. These property values are then committed to the VIO core in hardware in order to write these values to the output probe ports of the core. Refer to the section called [Interacting with VIO Core Output Probes](#), page 69 for more information.

## Viewing the VIO Core Status

The VIO core can have zero or more input probes and zero or more output probes (note that the VIO core must have at least one input or output probe). The VIO core status shown

in the **Hardware** window is used to indicate the current state of the VIO core output probes. The possible status values and any action that you need to take are described in [Table 5-3](#).

Table 5-3: VIO Core Status and Required User Action

VIO Status	Description	Required User Action
OK – Outputs Reset	The VIO core outputs are in sync with the Vivado IDE and the outputs are in their initial or "reset" state.	None
OK	The VIO core outputs are in sync with the Vivado IDE, however, the outputs are not in their initial or "reset" state.	None
Outputs out-of-sync	The VIO core outputs are not in sync with the Vivado IDE.	<p>You must choose one of two user actions:</p> <ul style="list-style-type: none"> <li>• Write the values from the Vivado IDE to the VIO core by right-clicking the VIO core in the <b>Hardware</b> window and selecting the <b>Commit VIO Core Outputs</b> option.</li> <li>• Update the Vivado IDE with the current values of the VIO core output probe ports by right-clicking the VIO core in the <b>Hardware</b> window and selecting the <b>Refresh Input and Output Values from VIO Core</b> option.</li> </ul>

## Viewing VIO Cores in the Debug Probes Window

The **Debug Probes** window is used to view all of the debug probes that belong to an ILA or VIO core (see [Figure 5-17](#)). The VIO debug probes can be added to **VIO Probes** windows of the **VIO Dashboard**. To perform these operations, right-click a VIO core's debug probes and select **Add Probes to VIO Window**.



**TIP:** If you right-click the VIO core object in the **Debug Probes** or **Hardware** window and select the **Add Probes to VIO Window** option, the selection option will apply to all probes that belong to that VIO core.



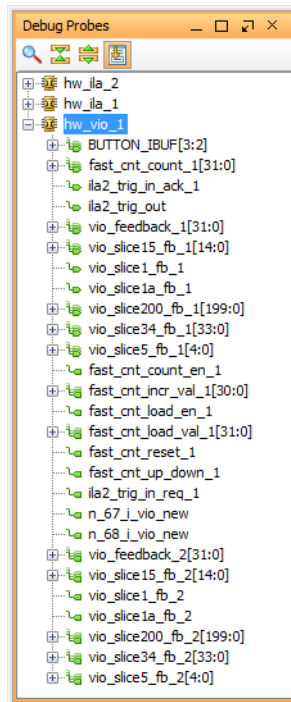


Figure 5-17: VIO Debug Probes

## Using the VIO Dashboard

The **VIO Dashboard** (see Figure 5-18) is a central location for all status and control information pertaining to a given VIO core. When a VIO core is first detected upon refreshing a hardware device, the **VIO Dashboard** for the core is automatically opened. If you need to manually open or re-open the dashboard, right-click the VIO core object in either the **Hardware** or **Debug Probes** windows and select **Open Dashboard**.

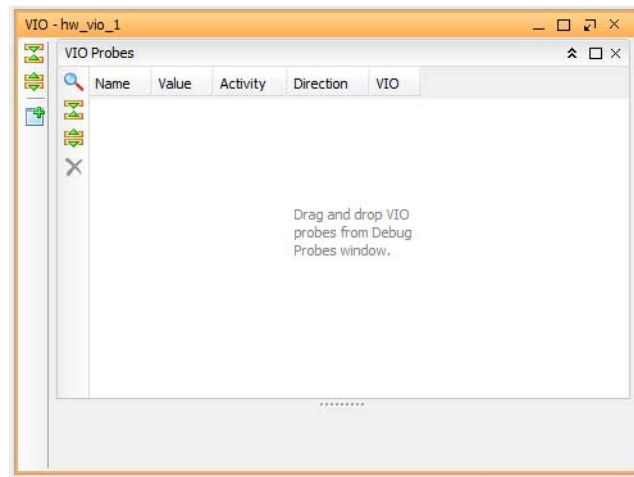


Figure 5-18: VIO Dashboard

## Interacting with VIO Core Input Probes

The VIO core input probes are used to read values from a design that is running in an FPGA in actual hardware. The VIO input probes are typically used as status indicators for a design-under-test. VIO debug probes need to be added manually to the **VIO Probes** window in the **VIO Dashboard**. Refer to the section called [Viewing VIO Cores in the Debug Probes Window](#), page 64 on how to do this. An example of what VIO input probes look like in the VIO Probes window of the VIO Dashboard is shown in [Figure 5-19](#).

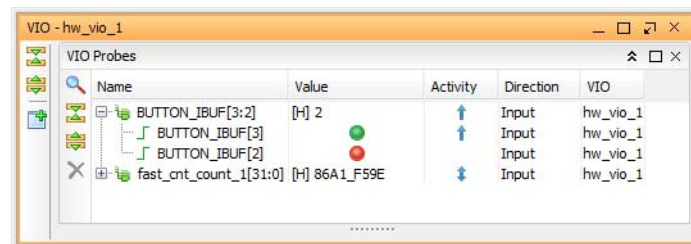


Figure 5-19: Core Input Probes

## Reading VIO Inputs Using the VIO Cores View

The VIO input probes can be viewed using the **VIO Probes** window of the **VIO Dashboard** window. Each input probe is viewed as a separate row in the table. The value of the VIO input probes are shown in the **Value** column of the table (see [Figure 5-19](#)). The VIO core input values are periodically updated based on the value of the refresh rate of the VIO core. You can set the refresh rate by changing the **Refresh Rate (ms)** in the **VIO Properties** window or by running the following Tcl command:

```
set_property CORE_REFRESH_RATE_MS 1000 [get_hw_vios hw_vio_1]
```

**Note:** Setting the refresh rate to 0 causes all automatic refreshes from the VIO core to stop. Also note that very small refresh values may cause your Vivado IDE to become sluggish. Xilinx recommends a refresh rate of 500 ms or longer.

If you want to manually read a VIO input probe value, you can use Tcl commands to do so. For instance, if you wanted to refresh and get the value of the input probe called `BUTTON_IBUF` of the VIO core `hw_vio_1`, run the following Tcl commands:

```
refresh_hw_vio [get_hw_vios {hw_vio_1}]
get_property INPUT_VALUE [get_hw_probes BUTTON_IBUF]
```

## Setting the VIO Input Display Type and Radix

The display type of VIO input probes can be set by right-clicking a VIO input probe in the **VIO Probes** window of the **VIO Dashboard** window and selecting:

- **Text** to display the input as a text field. This is the only display type for VIO input probe vectors (more than one bit wide).
- **LED** to display the input as a graphical representation of a light-emitting diode (LED). This display type is only applicable to VIO input probe scalars and individual elements of VIO input probe vectors. You can set the high and low values to one of four colors:
  - Gray (off)
  - Red
  - Green
  - Blue

When the display type of the VIO input probe is set to **Text**, you can change the radix by right-clicking a VIO input probe in the **VIO Probes** window of the **VIO Dashboard** window and selecting:

- **Radix > Binary** to change the radix to binary.
- **Radix > Octal** to change the radix to octal.
- **Radix > Hex** to change the radix to hexadecimal.
- **Radix > Unsigned** to change the radix to unsigned decimal.
- **Radix > Signed** to change the radix to signed decimal.

You can also set the radix of the VIO input probe using a Tcl command. For instance, to change the radix of a VIO input probe called `"BUTTON_IBUF"`, run the following Tcl command:

```
set_property INPUT_VALUE_RADIX HEX [get_hw_probes BUTTON_IBUF]
```

## Observing and Controlling VIO Input Activity

In addition to reading values from the VIO input probes, you can also monitor the activity of the VIO input probes. The activity detectors are used to indicate when the values on the VIO inputs have changed in between periodic updates to the Vivado IDE.

The VIO input probe activity values are shown as arrows in the activity column of the **VIO Probes** window of the **VIO Dashboard** window:

- An up arrow indicates that the input probe value has transitioned from a 0 to a 1 during the activity persistence interval.
- A down arrow indicates that the input probe value has transitioned from a 1 to a 0 during the activity persistence interval.
- A double-sided arrow indicates that the input probe value has transitioned from a 1 to a 0 and from a 0 to a 1 at least once during the activity persistence interval.

The persistence of how long the input activity status is displayed can be controlled by right-clicking a VIO input probe in the **VIO Probes** window of the **VIO Dashboard** window and selecting:

- **Activity Persistence > Infinite** to accumulate and retain the activity value until you reset it.
- **Activity Persistence > Long (80 samples)** to accumulate and retain the activity for a longer period of time.
- **Activity Persistence > Short (8 samples)** to accumulate and retain the activity for a shorter period of time.

You can also set the activity persistence using a Tcl command. For instance, to change the activity persistence on the VIO input probe called `BUTTON_IBUF` to a long interval, run the following Tcl command:

```
set_property ACTIVITY_PERSISTENCE LONG [get_hw_probes BUTTON_IBUF]
```

The activity for all input probes for a given core can be reset by right-clicking the VIO core in the **Hardware** window and selecting **Reset All Input Activity**. You can also do this by running the following Tcl command:

```
reset_hw_vio_activity [get_hw_vios {hw_vio_1}]
```



---

**TIP:** You can change the type, radix, and/or activity persistence of multiple scalar members of a VIO input probe vector by right-clicking the whole probe or multiple members of the probe, then making a menu choice. The menu choice applies to all selected probe scalars.

---

## Interacting with VIO Core Output Probes

The VIO core output probes are used to write values to a design that is running in an FPGA device in actual hardware. The VIO output probes are typically used as low-bandwidth control signals for a design-under-test. VIO debug probes need to be added manually to the **VIO Probes** window in the **VIO Dashboard**. Refer to the section called [Viewing VIO Cores in the Debug Probes Window, page 64](#) on how to do this. An example of what VIO output probes look like in the **VIO Probes** window of the **VIO Dashboard** is shown in Figure 5-20..

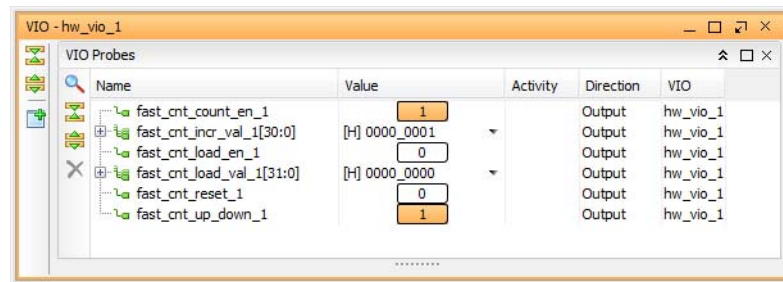


Figure 5-20: VIO Outputs in the VIO Probes window of the VIO Dashboard

## Writing VIO Outputs Using the VIO Cores View

The VIO output probes can be set using the **VIO Probes** window of the **VIO Dashboard** window. Each output probe is viewed as a separate row in the table. The value of the VIO output probes are shown in the Value column of the table (see [Figure 5-20](#)). The VIO core output values are updated whenever a new value is entered into the **Value** column. Clicking on the **Value** column causes a pull-down dialog to appear. Type the desired value into the **Value** text field and click **OK**.

You can also write out a new value to the VIO core using Tcl commands. For instance, if you wanted to write a binary value of "11111" to the VIO output probe called `vio_slice5_fb_2` whose radix is already set to BINARY, run the following Tcl commands:

```
set_property OUTPUT_VALUE 11111 [get_hw_probes vio_slice5_fb_2]
commit_hw_vio [get_hw_probes {vio_slice5_fb_2}]
```

## Setting the VIO Output Display Type and Radix

The display type of VIO output probes can be set by right-clicking a VIO output probe in the **VIO Probes** window of the **VIO Dashboard** window and selecting:

- **Text** to display the output as a text field. This is the only display type for VIO input probe vectors (more than one bit wide).

- **Toggle Button** to display the output as a graphical representation of a toggle button. This display type is only applicable to VIO output probe scalars and individual elements of VIO input probe vectors.

When the display type of the VIO output probe is set to "Text", you can change the radix by right-clicking a VIO output probe in the VIO Cores tabbed view of the Debug Probes window and selecting:

- **Radix > Binary** to change the radix to binary.
- **Radix > Octal** to change the radix to octal.
- **Radix > Hex** to change the radix to hexadecimal.
- **Radix > Unsigned** to change the radix to unsigned decimal.
- **Radix > Signed** to change the radix to signed decimal.

You can also set the radix of the VIO output probe using a Tcl command. For instance, to change the radix of a VIO output probe called "vio\_slice5\_fb\_2" to hexadecimal, run the following Tcl command:

```
set_property OUTPUT_VALUE_RADIX HEX [get_hw_probes vio_slice5_fb_2]
```

## Resetting the VIO Core Output Values

The VIO v2.0 core has a feature that allows you to specify an initial value for each output probe port. You can reset the VIO core output probe ports to these initial values by right-clicking the VIO core in the Hardware window and selecting the **Reset VIO Core Outputs** option. You can also reset the VIO core outputs using a Tcl command:

```
reset_hw_vio_outputs [get_hw_vios {hw_vio_1}]
```

**Note:** Resetting the VIO output probes to their initial values may cause the output probe values to become out-of-sync with the Vivado IDE. Refer to the section called Synchronizing the VIO Core Output Values to the Vivado IDE on how to handle this situation.

## Synchronizing the VIO Core Output Values to the Vivado IDE

The output probes of a VIO core can become out-of-sync with the Vivado IDE after resetting the VIO outputs, re-programming the FPGA, or by another Vivado tool instance setting output values before the current instance has started. In any case, if the VIO status indicates "Outputs out-of-sync", you need to take one of two actions:

- Write the values from the Vivado IDE to the VIO core by right-clicking the VIO core in the **Hardware** window and selecting the **Commit VIO Core Outputs** option. You can also do this running a Tcl command:

```
commit_hw_vio [get_hw_vios {hw_vio_1}]
```

- Update the Vivado IDE with the current values of the VIO core output probe ports by right-clicking the VIO core in the **Hardware** window and selecting the **Refresh Input and Output Values from VIO Core** option. You can also do this running a Tcl command:

```
refresh_hw_vio -update_output_values 1 [get_hw_vios {hw_vio_1}]
```

## Hardware System Communication Using the JTAG-to-AXI Master Debug Core

The JTAG-to-AXI Master debug core is a customizable core that can generate the AXI transactions and drive the AXI signals internal to an FPGA at run time. The core supports all memory mapped AXI and AXI-Lite interfaces and can support 32- or 64-bit wide data interfaces.

The JTAG-to-AXI Master (JTAG-AXI) cores that you add to your design appear in the **Hardware** window under the target device. If you do not see the JTAG-AXI cores appear, right-click on the device and select **Refresh Hardware**. This re-scans the FPGA device and refreshes the **Hardware** window.

**Note:** If you still do not see the ILA core after programming and/or refreshing the FPGA device, check to make sure the device was programmed with the appropriate .bit file and check to make sure the implemented design contains an ILA core.

Click to select the JTAG-AXI core (called hw\_axi\_1 in Figure 1) to see its properties in the **AXI Core Properties** window.

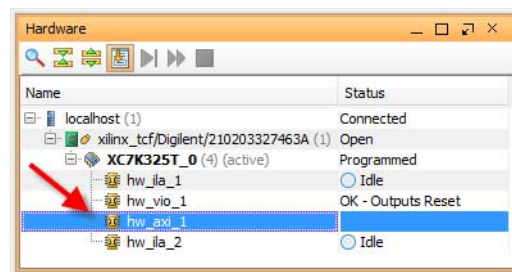


Figure 5-21: JTAG-to-AXI Master Core in the Hardware Window

## Interacting with the JTAG-to-AXI Master Debug Core in Hardware

The JTAG-to-AXI Master debug core can only be communicated with using Tcl commands. You can create and run AXI read and write transactions using the `create_hw_axi_txn` and `run_hw_axi` commands, respectively.

## Resetting the JTAG-to-AXI Master Debug Core

Before creating and issuing transactions, it is important to reset the JTAG-to-AXI Master core using the following Tcl command:

```
reset_hw_axi [get_hw_axis hw_axi_1]
```

## Creating and Running a Read Transaction

The Tcl command used to create an AXI transaction is called `create_hw_axi_txn`. For more information on how to use this command, type `"help create_hw_axi_txn"` at the Tcl Console in the Vivado IDE. Here is an example on how to create a 4-word AXI read burst transaction from address 0:

```
create_hw_axi_txn read_txn [get_hw_axis hw_axi_1] -type READ -address 00000000 -len 4
```

where:

- `read_txn` is the user-defined 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 4 words

The next step is to run the transaction that was just created using the `run_hw_axi` command. Here is an example on how to do this:

```
run_hw_axi [get_hw_axi_txns read_txn]
```

The last step is to get the data that was read as a result of running the transaction. You can use either the `report_hw_axi_txn` or `report_property` commands to print the data to the screen or you can use the `get_property` to return the value for use elsewhere.

```
report_hw_axi_txn [get_hw_axi_txns read_txn]
```

```
0 00000000 00000000
8 00000000 00000000
```

```
report_property [get_hw_axi_txns read_txn]
```

Property	Type	Read-only	Visible	Value
CLASS	string	true	true	hw_axi_txn
CMD.ADDR	string	false	true	00000000
CMD.BURST	enum	false	true	INCR
CMD.CACHE	int	false	true	3
CMD.ID	int	false	true	0
CMD.LEN	int	false	true	4
CMD.SIZE	enum	false	true	32
DATA	string	false	true	00000000000000000000000000000000
HW_AXI	string	true	true	hw_axi_1
NAME	string	true	true	read_txn
TYPE	enum	false	true	READ



### Creating and Running a Write Transaction

Here is an example on how to create a 4-word AXI write burst transaction from address 0:

```
create_hw_axi_txn write_txn [get_hw_axis hw_axi_1] -type READ -address 00000000 -len 4 -data {11111111_22222222_33333333_44444444}
```

where:

- `write_txn` is the user-defined 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 4 words
- `-data {11111111_22222222_33333333_44444444}` - The `-data` direction is LSB to the left (i.e., address 0) and MSB to the right (i.e., address 3).

The next step is to run the transaction that was just created using the `run_hw_axi` command. Here is an example on how to do this:

```
run_hw_axi [get_hw_axi_txns write_txn]
```

---

## Using Vivado Logic Analyzer in a Lab Environment

The Vivado logic analyzer feature is integrated into the Vivado IDE. To use Vivado logic analyzer feature to debug a design that is running on a target board that is in a lab environment, you need to do one of two things:

- Install and run the full Vivado IDE on your lab machine.
- Install latest version of the Vivado Design Suite on your lab machine, and use the Vivado logic analyzer feature on your local machine to connect to a remote instance of the Vivado CSE Server (`vcse_server`).

## Installing and Running the Full Vivado IDE on a Lab Machine

The requirements for installing the Vivado IDE on your lab machine are found in *Vivado Design Suite: Release Notes, Installation and Licensing (UG73)* [\[Ref 4\]](#).



**IMPORTANT:** *The Vivado logic analyzer is only compatible with the Vivado CSE server (vcse\_server.exe on Windows platforms or vcse\_server on Linux platforms) and is not compatible with the legacy CSE server (cse\_server.exe on Windows platforms or cse\_server on Linux platforms) that is installed as part of the ISE Lab Tools. The legacy CSE server application is only compatible with the legacy ChipScope Pro Analyzer tool. The Vivado logic analyzer only needs two files from the original project: the bitstream programming (.bit) file and the probes (.ltx) file.*

Here are the steps to use the Vivado logic analyzer feature on a lab machine.

1. Install the Vivado IDE on your lab machine.
2. Copy the bitstream programming (.bit) file and the probes (.ltx) file to the lab machine.
3. Start Vivado IDE in GUI mode.
4. Open the Hardware Manager by selecting the **Flow > Open Hardware Manager** menu option or typing "open\_hw" in the **Tcl Console** window.
5. Follow the steps in the [Connecting to the Hardware Target and Programming the FPGA Devices](#) section to open a connection to the target board that is connected to your lab machine. Use the bitstream programming (.bit) file that you copied to the lab machine to program target FPGA device.
6. Follow the steps in the [Setting up the ILA Core to Take a Measurement](#) section and beyond to debug your design in hardware. Use the probes (.ltx) file that you copied to the lab machine when you get to the [Reading ILA Probes Information](#) section.

## Connecting to a Remote CSE Server Running on a Lab Machine

If you have a network connection to your lab machine, you can also connect to the target board by connecting to a CSE server that is running on that remote lab machine. Here are the steps to using the Vivado logic analyzer feature to connect to a Vivado CSE server (vcse\_server.exe on Windows platforms or vcse\_server on Linux platforms) that is running on the lab machine:

1. Install the latest version of the Vivado Design Suite on the lab machine.



**IMPORTANT:** *You do NOT need any software licenses to run any of the Hardware Manager features of the Vivado tools (such as the Vivado logic analyzer or Vivado serial I/O analyzer features).*

2. Start up the vcse\_server application on the remote lab machine. Assuming you installed the Vivado Design Suite to the default location and your lab machine is a 64-bit Windows machine, here is the command line:

```
C:\Xilinx\Vivado\vivado_release.version\bin\vcse_server.bat -port 60001
```

3. Start Vivado IDE in GUI mode on a different machine than your lab machine.
4. Follow the steps in the [Connecting to the Hardware Target and Programming the FPGA Device](#) section to open a connection to the target board that is connected to your lab machine. However, instead of connecting to a Vivado CSE server running on localhost, use the host name of your lab machine.
5. Follow the steps in the [Setting up the ILA Core to Take a Measurement](#) section and beyond to debug your design in hardware.

---

## Using Vivado Logic Analyzer and ChipScope™ Pro Analyzer Simultaneously

You can use the Vivado logic analyzer feature and ChipScope™ Pro Analyzer tools to simultaneously debug the same design running on the same target board. Some examples of situations where this capability becomes important are:

- You have a design that contains a new ILA debug core and a legacy VIO v1.x debug core that you need to interact with using the Vivado logic analyzer feature and ChipScope Pro Analyzer feature, respectively.
- You want to interact with a new ILA debug core in your design using the Vivado logic analyzer feature and you want to monitor the XADC temperature or voltage sensors using the ChipScope Pro Analyzer tool's System Monitor feature.
- You have a 7 series device and a 6 series device that you need to debug at the same time using the Vivado logic analyzer feature and the ChipScope Pro Analyzer feature, respectively.

This section covers the first case of having both a new ILA debug core and a legacy VIO v1.x debug core in the same design. The steps that you need to follow to take advantage of this capability are:

1. Use two separate BSCAN user scan chains, one for each JTAG controller core.
2. Start two separate CSE servers, one for each run time analyzer applications.
3. Use two different run time analyzer applications, the Vivado logic analyzer feature and ChipScope Pro Analyzer tool.

## Using Separate BSCAN User Scan Chains

Make sure that the debug\_core\_hub core (which is used to connect the new ILA core to a BSCANE2 primitive) and the legacy ICON v1.x core (which is used to connect the VIO v1.x core to a BSCANE2 primitive) are configured to use different user scan chains (see section [Changing the BSCAN User Scan Chain of the Debug Core Hub in Chapter 4](#) for more details).

## Setting Up Separate CSE Servers

Make sure two separate CSE server instances are running on the machine that is connected to the target board. For instance, on a Windows 64-bit platform, do the following in two separate `cmd` windows:

1. Make sure the latest version of the Vivado Design Suite and ISE Lab Tools are installed on the lab machine. This example assumes that ISE Lab Tools and Vivado Design Suite are both installed in their default locations.
2. Open the first `cmd` window and run the following to start the Vivado CSE server.

```
C:\Xilinx\Vivado\vivado_release.version\bin\vcse_server.bat -port 60001
```

3. Open the second `cmd` window and run to start the legacy CSE server.

```
C:\Xilinx\ise_release.version\LabTools\LabTools\bin\nt64\cse_server -port 50001
```

The Vivado CSE server running on port 60001 and the legacy CSE server running on port 50001 is used by the Vivado logic analyzer feature and ChipScope Pro Analyzer tool, respectively.

## Running Vivado Logic Analyzer Feature and ChipScope Pro Analyzer Tool

Now that you have set up the design and the CSE servers, you can use the Vivado logic analyzer feature and ChipScope Pro Analyze tool to debug the design by following these steps:

1. Start the Vivado IDE in GUI mode.
2. Click **Open Hardware Manager** in the **Flow Navigator** or select the **Flow > Open Hardware Manager** menu option.
3. Connect to the Vivado CSE server that is running on localhost:60001, and program the target device (see [Using a Vivado Hardware Manager to Program an FPGA Device in Chapter 2](#) for more details)
4. Launch the ChipScope Pro Analyzer using the **Flow > Launch ChipScope Analyzer** menu option.

5. In the ChipScope Pro Analyzer tool, select **JTAG Chain > Server Host Setting**. Change the server to **localhost:50001** (see Figure 5-22)

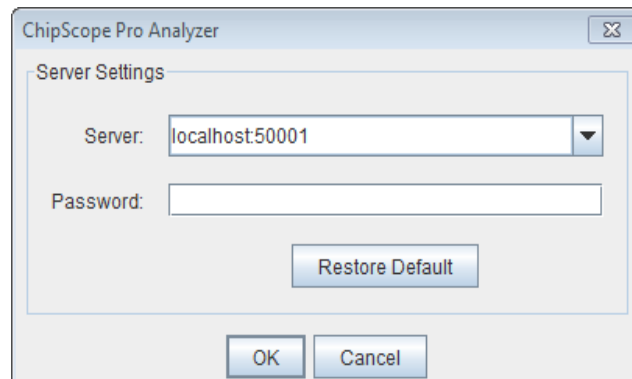


Figure 5-22: Changing the ChipScope Pro Analyzer Server Host Settings

6. In the ChipScope Pro Analyzer, connect to the target board by using the **JTAG Chain > Open Plug-in** menu option set to **xilinx\_tcf** (see Figure 5-23).

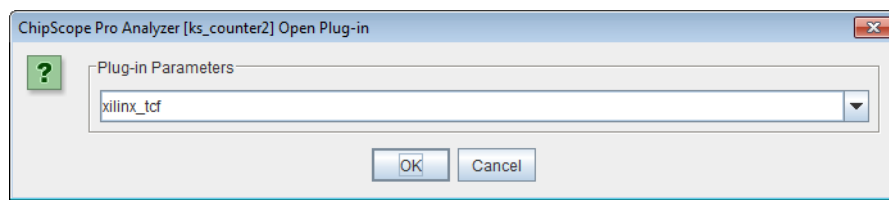


Figure 5-23: Connecting to the Target Board



**IMPORTANT:** The ChipScope Pro Analyzer can only handle a single JTAG target plugged into the machine when using the *xilinx\_tcf* plug-in.

- Use each of the run time analyzer tools to interact with their respective debug cores (see Figure 5-24).

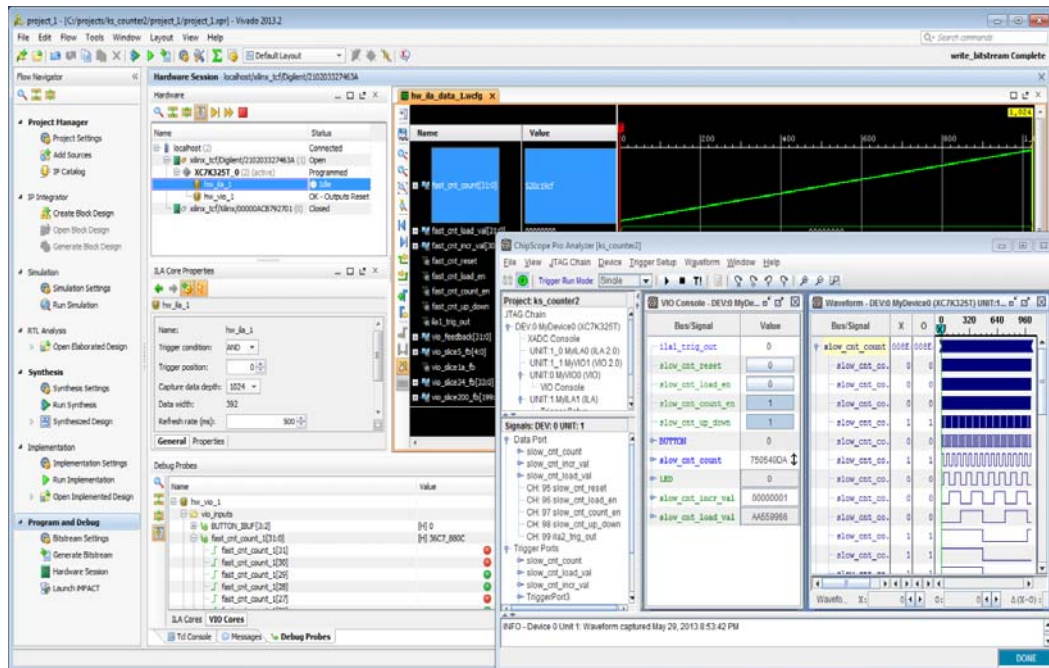


Figure 5-24: Using Vivado Logic Analyzer Feature and ChipScope Pro Analyzer Tool to Debug the Design

## Description of Hardware Manager Tcl Objects and Commands

You can use Tcl commands to interact with your hardware under test. The hardware is organized in a set of hierarchical first class Tcl objects (see Table 5-4).

Table 5-4: Hardware Manager Tcl Objects

Tcl Object	Description
hw_server	Object referring to CSE server. Each hw_server can have one or more hw_target objects associated with it.
hw_target	Object referring to JTAG cable or board. Each hw_target can have one or more hw_device objects associated with it.
hw_device	Object referring to a device in the JTAG chain, including Xilinx FPGA devices. Each hw_device can have one or more hw_ila objects associated with it.
hw_ila	Object referring to an ILA core in the Xilinx FPGA device. Each hw_ila object can have only one hw_ila_data object associated with it. Each hw_ila object can have one or more hw_probe objects associated with it.

**Table 5-4: Hardware Manager Tcl Objects (Cont'd)**

Tcl Object	Description
hw_ila_data	Object referring to data uploaded from an ILA debug core.
hw_probe	Object referring to the probe input of an ILA debug core.
hw_vio	Object referring to a VIO core in the Xilinx FPGA device.

For more information about the hardware manager commands, run the `help -category hardware` Tcl command in the Tcl Console.

## Description of hw\_server Tcl Commands

Table 5-5 contains descriptions of all Tcl commands used to interact with hardware servers.

**Table 5-5: Descriptions of hw\_server Tcl Commands**

Tcl Command	Description
connect_hw_server	Open a connection to a hardware server.
current_hw_server	Get or set the current hardware server.
disconnect_hw_server	Close a connection to a hardware server.
get_hw_servers	Get list of hardware server names for the CSE servers.
refresh_hw_server	Refresh a connection to a hardware server.

## Description of hw\_target Tcl Commands

Table 5-6 contains descriptions of all Tcl commands used to interact with hardware targets.

Table 5-6: Descriptions of hw\_target Tcl Commands

Tcl Command	Description
close_hw_target	Close a hardware target.
current_hw_target	Get or set the current hardware target.
get_hw_targets	Get list of hardware targets for the hardware servers.
open_hw_target	Open a connection to a hardware target on the hardware server.
refresh_hw_target	Refresh a connection to a hardware target.

## Description of hw\_device Tcl Commands

Table 5-7 Descriptions of hw\_device Tcl Commands contains descriptions of all Tcl commands used to interact with hardware devices.

Table 5-7: Descriptions of hw\_device Tcl Commands

Tcl Command	Description
current_hw_device	Get or set the current hardware device.
get_hw_device	Get list of hardware devices for the target.
program_hw_device	Program Xilinx FPGA devices.
refresh_hw_device	Refresh a hardware device.

## Description of hw\_ila Tcl Commands

Table 5-8 Descriptions of hw\_ila Tcl Commands contains descriptions of all Tcl commands used to interact with ILA debug cores.

Table 5-8: Descriptions of hw\_ila Tcl Commands

Tcl Command	Description
current_hw_ila	Get or set the current hardware ILA.
get_hw_ilas	Get list of hardware ILAs for the target.
reset_hw_ila	Reset hw_ila control properties to default values.
run_hw_ila	Arm hw_ila triggers.
wait_on_hw_ila	Wait until all data has been captured.



## Description of hw\_ila\_data Tcl Commands

[Table 5-9](#) Descriptions of hw\_ila\_data Tcl Commands contains descriptions of all Tcl commands used to interact with captured ILA data.

**Table 5-9: Descriptions of hw\_ila\_data Tcl Commands**

Tcl Command	Description
current_hw_ila_data	Get or set the current hardware ILA data
display_hw_ila_data	Display hw_ila_data in waveform viewer
get_hw_ila_data	Get list of hw_ila_data objects
read_hw_ila_data	Read hw_ila_data from a file
upload_hw_ila_data	Stop the ILA core from capturing data and upload any captured data.
write_hw_ila_data	Write hw_ila_data to a file.

## Description of hw\_probe Tcl Commands

[Table 5-10](#) contains descriptions of all Tcl commands used to interact with captured ILA data.

**Table 5-10: Descriptions of hw\_probe Tcl Commands**

Tcl Command	Description
get_hw_probes	Get list of hardware probes.

## Description of hw\_vio Tcl Commands

[Table 5-11](#) contains descriptions of all Tcl commands used to interact with VIO cores.

**Table 5-11: Descriptions of hw\_vio Tcl Commands**

Tcl Command	Description
commit_hw_vio	Write hw_probe OUTPUT_VALUE properties values to VIO cores.
get_hw_vios	Get a list of hw_vios
refresh_hw_vios	Update hw_probe INPUT_VALUE and ACTIVITY_VALUE properties with values read from VIO cores.
reset_hw_vio_activity	Reset VIO ACTIVITY_VALUE properties, for hw_probes associated with specified hw_vio objects.
reset_hw_vio_outputs	Reset VIO core outputs to initial values.

## Using Hardware Manager Tcl Commands

Below is an example Tcl command script that interacts with the following example system:

- One KC705 board's Digilent JTAG-SMT1 cable (serial number 12345) accessible via a Vivado CSE server running on localhost:60001.
- Single ILA core in a design running in the XC7K325T device on the KC705 board.
- ILA core has a probe called counter[3:0].

### Example Tcl Command Script

```
# Connect to the Digilent Cable on localhost:60001
connect_hw_server -host localhost -port 60001
current_hw_target [get_hw_targets */xilinx_tcf/Digilent/12345]
open_hw_target

# Program and Refresh the XC7K325T Device
current_hw_device [lindex [get_hw_devices] 0]
refresh_hw_device -update_hw_probes false [lindex [get_hw_devices] 0]
set_property PROGRAM.FILE {C:/design.bit} [lindex [get_hw_devices] 0]
set_property PROBES.FILE {C:/design.ltx} [lindex [get_hw_devices] 0]
program_hw_devices [lindex [get_hw_devices] 0]
refresh_hw_device [lindex [get_hw_devices] 0]

# Set Up ILA Core Trigger Position and Probe Compare Values
set_property CONTROL.TRIGGER_POSITION 512 [get_hw_ilas hw_ila_1]
set_property COMPARE_VALUE.0 eq4'b0000 [get_hw_probes counter]

# Arm the ILA trigger and wait for it to finish capturing data
run_hw_ila hw_ila_1
wait_on_hw_ila hw_ila_1

# Upload the captured ILA data, display it, and write it to a file
current_hw_ila_data [upload_hw_ila_data hw_ila_1]
display_hw_ila_data [current_hw_ila_data]
write_hw_ila_data my_hw_ila_data [current_hw_ila_data]
```

# Viewing ILA Probe Data Using Waveform Viewer

---

## Introduction

With the Vivado™ Integrated Design Environment (IDE) simulator open, you can begin working with the waveform to analyze your design and debug your code. The Vivado® logic analyzer populates design data in other areas, such as the **Hardware** and the **ILA Debug Probes** windows.

---

## About Wave Configurations and Windows

Although both a wave configuration and a WCFG file refer to the customization of lists of waveforms, there is a conceptual difference between them:

- The wave configuration is an object that is loaded into memory with which you can work.
- The WCFG file is the saved form of a wave configuration on disk.

A wave configuration can have a name or be "**Untitled#**". The name shows on the view of the **Wave Configuration** window.

## Opening a WCFG File

Open a WCFG file to use with the ILA waveform window by following the steps described below.



**IMPORTANT:** Before opening a WCFG file, you must have a waveform window open that contains ILA probes in it, otherwise you get an error.

---

1. Select **File > Open Waveform Configuration**.

The **Open Waveform Configuration** dialog box opens.

2. Locate and select a WCFG file.
3. Click **OK**

A new waveform window appears showing the captured ILA probe data using the waveform configuration contained in the WCFG file that was just opened.

**Note:** You now have two waveform windows showing the same data: one waveform window that uses the original waveform configuration, and a second waveform window that uses the newly opened waveform configuration. The underlying waveform data is the same in both windows. You can safely close the original waveform window if you want to continue to use the newly opened waveform configuration.

**Note:** When you open a WCFG file that contains references to hw\_probe objects that are not present in an ILA core, the Vivado logic analyzer ignores those hw\_probe objects and omits them from the loaded waveform configuration.

## Saving a Wave Configuration

To save a wave configuration to a WCFG file, select **File > Save Waveform Configuration As**, and type a name for the waveform configuration.



---

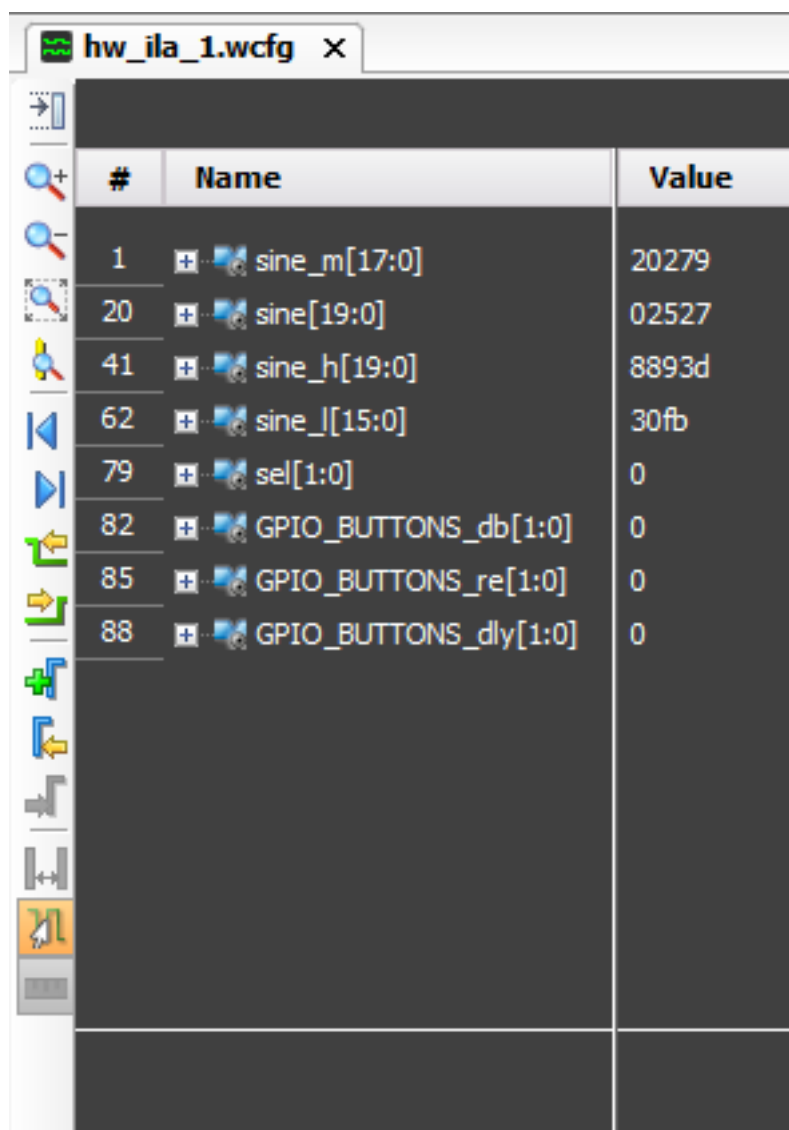
**IMPORTANT:** When saving the waveform configuration to a WCFG file, make sure you select a directory location other than the `<project>/Xil` directory. This is a temporary directory whose contents are deleted when Vivado exits. You will lose your WCFG file if you store it in this location

---

## Waveform Viewer Configuration Signals and Buses

The scalar and vector ILA probes in the **Waveform** window are the design objects in the waveform.

The ILA probes display with a corresponding identifying button. You can hover the mouse over the button for a description. [Figure 6-1](#) is an example of ILA probes in the **Waveform Configuration** window.



#	Name	Value
1	sine_m[17:0]	20279
20	sine[19:0]	02527
41	sine_h[19:0]	8893d
62	sine_l[15:0]	30fb
79	sel[1:0]	0
82	GPIO_BUTTONS_db[1:0]	0
85	GPIO_BUTTONS_re[1:0]	0
88	GPIO_BUTTONS_dly[1:0]	0

Figure 6-1: Waveform ILA Probes

The ILA probes display with an **ID Number**, **Name**, and **Value**. The toolbar buttons on the left give you access to navigation features that are described in the following sections.


## Using the Zoom Features

You have zoom functions as toolbar buttons to zoom in and out of a wave configuration as needed.

You can also use the mouse wheel with the **CTRL** key in combination after clicking within the waveform to zoom in and out to emulate the operation of the dials on an oscilloscope.



## Waveform Options Dialog Box

When you select the **Waveforms Options** button  the **Waveform Options** dialog box, shown in [Figure 6-2](#), opens.

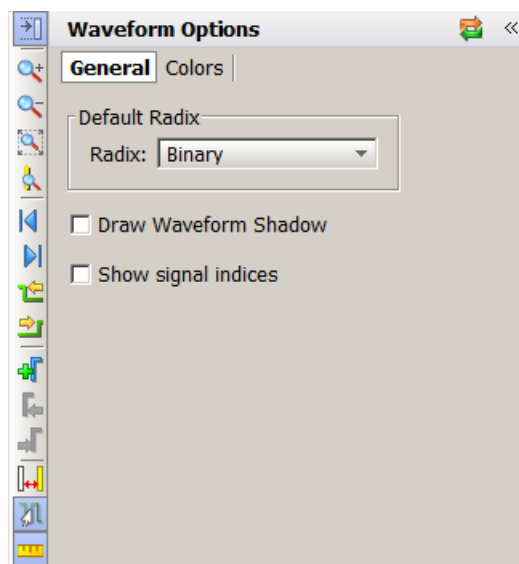


Figure 6-2: Waveform Options Dialog Box

The options are as follows:

- **General:** Set the default radix.
- **Show signal indices:** Checkbox inserts a small definition line between signal numbers.
- **Colors:** Lets you set colors for the objects within the waveform.

---

## ILA Probes in Waveform Configuration

You can add ILA probe scalars and vectors, also called signals and buses, to the waveform configuration file, then save that configuration to a WDB file. You can populate the **Wave** window with the probes from your ILA core using menu commands, or using Tcl commands in the **Tcl** Console.

To add ILA probes to the waveform configuration:

1. In the **ILA Cores** view of the **Debug Probes** window, expand the desired ILA core, and select a probe.
2. Right-click, and select **Add Probes to Waveform** window from the popup menu.

You can add copies of the same signal or bus in a wave configuration for comparing waveforms. You can place copies of the same signal or bus anywhere in the wave configuration, such as in groups or virtual buses.

To add a copy of a signal or bus, do the following:

1. Select a signal or bus in the wave configuration in the **Waveform** window.
2. Select **Edit > Copy** or type **Ctrl+C**.

The signal name is copied to the clipboard.

3. Select **Paste** or type **Ctrl+V**.

The signal or bus is now copied to the wave configuration. You can move the signal or bus using drag and drop as needed.

## Customizing the Waveform Configuration

You can customize the Waveform configuration using the features that are listed and briefly described in [Table 6-1](#); the feature name links to the subsection that fully describes the feature.

**Table 6-1: Customization Features in the Waveform Configuration**

Feature	Description
<a href="#">Cursors</a>	The main cursor and secondary cursor in the <b>Waveform</b> window let you display and measure time, and they form the focal point for various navigation activities.
<a href="#">Markers</a>	You can add markers to navigate through the waveform, and to display the waveform value at a particular time.
<a href="#">Dividers</a>	You can add a divider to create a visual separator of signals.
<a href="#">Using Groups</a>	You can add a group, that is a collection to which signals and buses can be added in the wave configuration as a means of organizing a set of related signals.
<a href="#">Using Virtual Buses</a>	You can add a virtual bus to your wave configuration, to which you can add logic scalars and arrays.
<a href="#">Renaming Objects</a>	You can rename objects, signals, buses, and groups.
<a href="#">Displaying Names</a>	You can display the full hierarchical name (long name), the simple signal or bus name (short name), or a custom name for each signal.

Table 6-1: Customization Features in the Waveform Configuration (Cont'd)

Feature	Description
Radixes	The default radix controls the bus radix that displays in the wave configuration, Objects panel, and the Console panel.
Bus Bit Order	You can change the Bus bit order from Most Significant Bit (MSB) to Least Significant Bit (LSB) and vice versa.

## Cursors

Cursors are used primarily for temporary indicators of sample position and are expected to be moved frequently, as in the case when you are measuring the distance (in samples) between two waveform edges.



**TIP:** For more permanent indicators, used in situations such as establishing a time-base for multiple measurements, add markers to the Wave window instead. See [Markers, page 88](#) for more information.

You can place the main cursor with a single click in the **Waveform** window.

To place a secondary cursor, **Ctrl+Click** and hold the waveform, and drag either left or right. You can see a flag that labels the location at the top of the cursor.

Alternatively, you can hold the **SHIFT** key and click a point in the waveform. The main cursor remains the original position, and the other cursor is at the point in the waveform that you clicked.

**Note:** To preserve the location of the secondary cursor while positioning the main cursor, hold the **Shift** key while clicking. When placing the secondary cursor by dragging, you must drag a minimum distance before the secondary cursor appears.

To move a cursor, hover over the cursor until you see the grab symbol, and click and drag the cursor to the new location.

As you drag the cursor in the **Waveform** window, you see a hollow or filled-in circle if the **Snap to Transition** button is selected, which is the default behavior.


- A hollow circle ○ indicates that you are between transitions in the waveform of the selected signal.
- A filled-in circle ● indicates that you are hovering over the waveform transition of the selected signal. A secondary cursor can be hidden by clicking anywhere in the **Waveform** window where there is no cursor, marker, or floating ruler.

## Markers




Use a marker when you want to mark a significant event within your waveform in a permanent fashion. Markers allow you to measure distance (in samples) relevant to that marked event.



You can add, move, and delete markers as follows:

- You add markers to the wave configuration at the location of the main cursor.
  - a. Place the main cursor at the sample number where you want to add the marker by clicking in the **Waveform** window at the sample number or on the transition.
  - b. Select **Edit > Markers > Add Marker**, or click the **Add Marker** button. 

A marker is placed at the cursor, or slightly offset if a marker already exists at the location of the cursor. The sample number of the marker displays at the top of the line.

- You can move the marker to another location in the waveform using the drag and drop method. Click the marker label (at the top of the marker) and drag it to the location.
  - The drag symbol  indicates that the marker can be moved. As you drag the marker in the **Waveform** window, you see a hollow or filled-in circle if the **Snap to Transition** button is selected, which is the default behavior.
  - A filled-in circle  indicates that you are hovering over a transition of the waveform for the selected signal or over another marker.
  - For markers, the filled-in circle is white.
  - A hollow circle  indicates that you are between transitions in the waveform of the selected signal.
  - Release the mouse key to drop the marker to the new location.
- You can delete one or all markers with one command. Right-click over a marker, and do one of the following:
  - Select **Delete Marker** from the popup menu to delete a single marker.
  - Select **Delete All Markers** from the popup menu to delete all markers.

**Note:** You can also use the **Delete** key to delete a selected marker.

  - Use **Edit > Undo** to reverse a marker deletion.

## Trigger Marker

The red trigger marker (whose label is a red letter 'T') is a special marker that indicates the occurrence of the trigger event in the capture buffer. The position of the trigger marker in the buffer directly corresponds to the Trigger Position setting (see [Using the ILA Dashboard, page 48](#)).

**Note:** The trigger marker is not movable using the same technique as regular markers. Its position is set using the ILA core's Trigger Position property setting.

## Dividers

Dividers create a visual separator between signals. You can add a divider to your wave configuration to create a visual separator of signals, as follows:

1. In a **Name** column of the **Waveform** window, click a signal to add a divider below that signal.
2. From the popup menu, select **Edit > New Divider**, or right-click and select **New Divider**.

The change is visual and nothing is added to the HDL code. The new divider is saved with the wave configuration file when you save the file.

You can move or delete Dividers as follows:

- Move a Divider to another location in the waveform by dragging and dropping the divider name.
- To delete a Divider, highlight the divider, and click the **Delete** key, or right-click and select **Delete** from the popup menu.

Dividers can be renamed also; see [Renaming Objects, page 92](#).

## Using Groups

A Group is a collection of expandable and collapsible categories, to which you can add signals and buses in the wave configuration to organize related sets of signals. The group itself displays no waveform data but can be expanded to show its contents or collapsed to hide them. You can add, change, and remove groups.

To add a Group:

1. In a wave configuration, select one or more signals or buses to add to a group.  
**Note:** A group can include dividers, virtual buses, and other groups.
2. Select **Edit > New Group**, or right-click and select **New Group** from the popup menu.

A Group that contains the selected signal or bus is added to the wave configuration.

A Group is represented with the **Group** button. 

The change is visual and nothing is added to the ILA core.

You can move other signals or buses to the group by dragging and dropping the signal or bus name.

You can move or remove Groups as follows:

- Move Groups to another location in the **Name** column by dragging and dropping the group name.
- Remove a group, by highlighting it and selecting **Edit > Wave Objects > Ungroup**, or right-click and select **Ungroup** from the popup menu. Signals or buses formerly in the group are placed at the top-level hierarchy in the wave configuration.

Groups can be renamed also; see [Renaming Objects](#), page 92.



---

**CAUTION!** The **Delete** key removes the group and its nested signals and buses from the wave configuration.

---

## Using Virtual Buses

You can add a virtual bus to your wave configuration, which is a grouping to which you can add logic scalars and arrays. The virtual bus displays a bus waveform, which shows the signal waveforms in the vertical order that they appear under the virtual bus, flattened to a one-dimensional array. You can then change or remove virtual buses after adding them.

To add a virtual bus:

1. In a wave configuration, select one or more signals or buses you want to add to a virtual bus.
2. Select **Edit > New Virtual Bus**, or right-click and select **New Virtual Bus** from the popup menu.

The virtual bus is represented with the **Virtual Bus** button .

The change is visual and nothing is added to the HDL code.

You can move other signals or buses to the virtual bus by dragging and dropping the signal or bus name. The new virtual bus and its nested signals or buses are saved when you save the wave configuration file. You can also move it to another location in the waveform by dragging and dropping the virtual bus name.

You can rename a virtual bus; see [Renaming Objects](#).

To remove a virtual bus, and ungroup its contents, highlight the virtual bus, and select **Edit > Wave Objects > Ungroup**, or right-click and select **Ungroup** from the popup menu.



---

**CAUTION!** The **Delete** key removes the virtual bus and its nested signals and buses from the wave configuration.

---

## Renaming Objects

You can rename any object in the **Waveform** window, such as signals, dividers, groups, and virtual buses.

1. Select the object name in the **Name** column.
2. Select **Rename** from the popup menu.
3. Replace the name with a new one.
4. Press **Enter** or click outside the name to make the name change take effect.

You can also double-click the object name and then type a new name. The change is effective immediately. Object name changes in the wave configuration do not affect the names of the nets attached to the ILA core probe inputs.

## Displaying Names

You can display the full hierarchical name (long name), the simple signal or bus name (short name), or a custom name for each signal. The signal or bus name displays in the **Name** column of the wave configuration. If the name is hidden:

- Expand the **Name** column until you see the entire signal name.
- Use the scroll bar in the **Name** column to view the name.

To change the display name:

1. Select one or more signal or bus names. Use **Shift+** click or **Ctrl+** click to select many signal names.
2. Right-click, and select **Name >**:
  - **Long** to display the full hierarchical name.
  - **Short** to display the name of the signal or bus only.
  - **Custom** to display the custom name given to the signal when renamed.

The name changes immediately according to your selection.

## Radixes

Understanding the type of data on your bus is important. You need to recognize the relationship between the radix setting and the data type to use the waveform options of Digital and Analog effectively. See [About Radixes and Analog Waveforms, page 94](#) for more information about the radix setting and its effect on Analog waveform analysis.

You can change the radix of an individual signal (ILA probe) in the **Waveform** window as follows:

1. Right-click a bus in the **Waveform** window.
2. Select **Radix** and the format you want from the drop-down menu:
  - **Binary**
  - **Hexadecimal**
  - **Unsigned Decimal**
  - **Signed Decimal**
  - **Octal**
  - **ASCII**



**IMPORTANT:** Changes to the radix of an item in the Objects window do not apply to values in the **Waveform** window or the **Tcl** Console. To change the radix of an individual signal (ILA probe) in the **Waveform** window, use the **Waveform** window popup menu.

- Maximum bus width of 64 bits on real. Incorrect values are possible for buses wider than 64 bits.
- Floating point supports only 32- and 64-bit arrays.

## Using the Floating Ruler

The floating ruler assists with time measurements using a sample number base other than the absolute sample numbers shown on the standard ruler at the top of the **Waveform** window.

You can display (or hide) a floating ruler and move it to a location in the **Waveform** window. The sample base (sample 0) of the floating ruler is the secondary cursor, or, if there is no secondary cursor, the selected marker.

The floating ruler button and the floating ruler itself are visible only when the secondary cursor (or selected marker) is present.

1. Do either of the following to display or hide a floating ruler:
  - Place the secondary cursor.
  - Select a marker.
2. Select **View > Floating Ruler**, or click the **Floating Ruler** button.



You only need to follow this procedure the first time. The floating ruler displays each time the secondary cursor is placed or a marker is selected.

Select the command again to hide the floating ruler.

## Bus Bit Order

You can reverse the bus bit order in the wave configuration to switch between MSB-first and LSB-first signal representation.

To reverse the bit order:

1. Select a bus.
2. Right-click and select **Reverse Bit Order**.

The bus bit order is reversed. The **Reverse Bit Order** command is marked to show that this is the current behavior.

---

## About Radixes and Analog Waveforms

Bus values are interpreted as numeric values, which are determined by the radix setting on the bus wave object, as follows:

- Binary, octal, hexadecimal, ASCII, and unsigned decimal radixes cause the bus values to be interpreted as unsigned integers. The format of data on the bus must match the radix setting.
- Any non-0 or -1 bits cause the entire value to be interpreted as 0.
- The signed decimal radix causes the bus values to be interpreted as signed integers.
- Real radixes cause bus values to be interpreted as fixed point or floating point real numbers, as determined by the settings of the **Real Settings** dialog box, shown in [Figure 6-3, page 95](#).

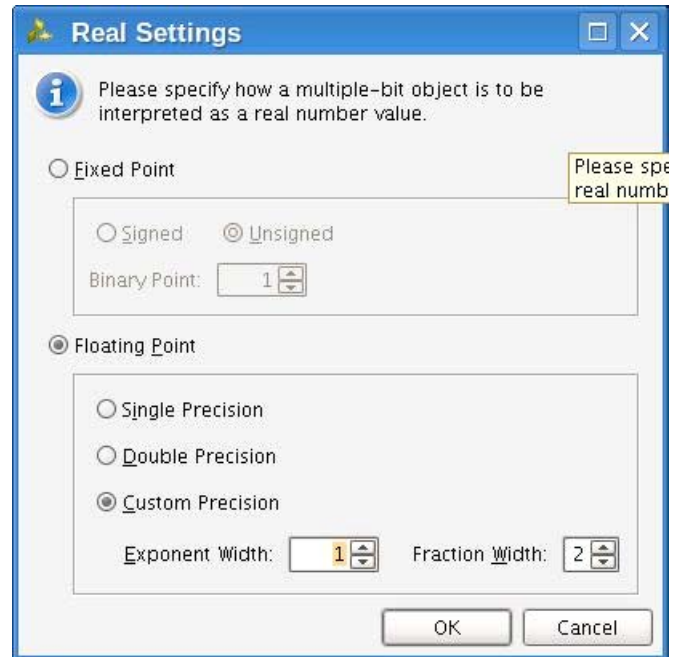


Figure 6-3: Real Settings Dialog Box

The options are as follows:

- **Fixed Point:** Specifies that the bits of the selected bus wave objects is interpreted as a fixed point, signed, or unsigned real number.
- **Binary Point:** Specifies how many bits to interpret as being to the right of the binary point. If **Binary Point** is larger than the bit width of the wave object, wave object values cannot be interpreted as fixed point, and when the wave object is shown in Digital waveform style, all values show as <Bad Radix>. When shown as analog, all values are interpreted as 0.
- **Floating Point:** Specifies that the bits of the selected bus wave objects should be interpreted as an IEEE floating point real number.

**Note:** Only single precision and double precision (and custom precision with values set to those of single and double precision) are supported.

Other values result in <Bad Radix> values as in **Fixed Point**.

**Exponent Width** and **Fraction Width** must add up to the bit width of the wave object, or else <Bad Radix> values result.

## Viewing Analog Waveforms

To convert a digital waveform to analog, do the following:

1. In the **Name** area of a **Waveform** window, right-click on the bus for the popup menu.

2. Select **Waveform Style** and then **Analog Settings** to choose an appropriate drawing setting.

The digital drawing of the bus converts to an analog format.

You can adjust the height of either an analog waveform or a digital waveform by selecting and then dragging the rows.

Figure 6-4 shows the **Analog Settings** dialog box with the settings for analog waveform drawing.

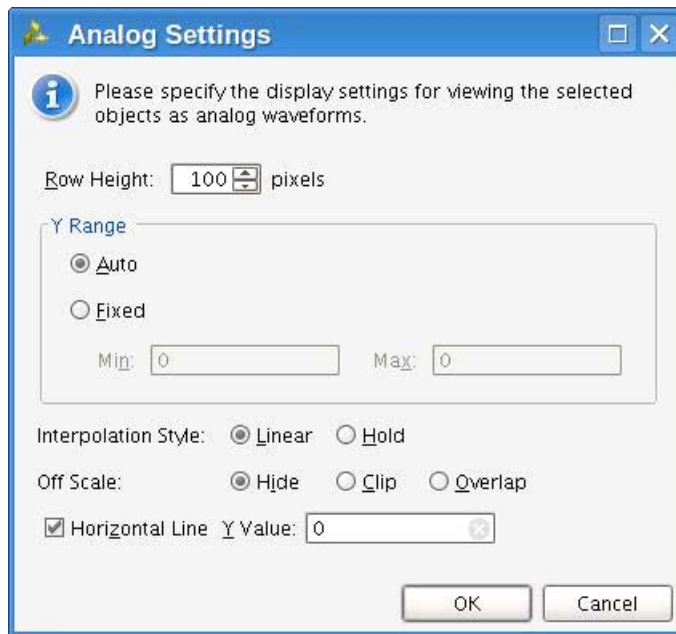


Figure 6-4: Analog Settings Dialog Box

The **Analog Settings** dialog box options are as follows:

- **Row Height:** Specifies how tall to make the select wave objects, in pixels. Changing the row height does not change how much of a waveform is exposed or hidden vertically, but rather stretches or contracts the height of the waveform.

When switching between Analog and Digital waveform styles, the row height is set to an appropriate default for the style (20 for digital, 100 for analog).

- **Y Range:** Specifies the range of numeric values to be shown in the waveform area.
  - **Auto:** Specifies that the range should continually expand whenever values in the visible time range of the window are discovered to lie outside the current range.
  - **Fixed:** Specifies that the time range is to remain at a constant interval.
  - **Min:** Specifies the value displays at the bottom of the waveform area.



- **Max:** Specifies the value displays at the top.

Both values can be specified as floating point; however, if radix of the wave object radix is integral, the values are truncated to integers.

- **Interpolation Style:** Specifies how the line connecting data points is to be drawn.
  - **Linear:** Specifies a straight line between two data points.
  - **Hold:** Specifies that of two data points, a horizontal line is drawn from the left point to the X-coordinate of the right point, then another line is drawn connecting that line to the right data point, in an L shape.
  - **Off Scale:** Specifies how to draw waveform values that lie outside the Y range of the waveform area.
  - **Hide:** Specifies that outlying values are not shown, such that a waveform that reaches the upper or lower bound of the waveform area disappears until values are again within the range.
  - **Clip:** Specifies that outlying values be altered so that they are at the top or bottom of the waveform area, such that a waveform that reaches the upper- or lower-bound of the waveform area follows the bound as a horizontal line until values are again within the range.
  - **Overlap:** Specifies that the waveform be drawn wherever its values are, even if they lie outside the bounds of the waveform area and overlap other waveforms, up to the limits of the wave window itself.
- **Horizontal Line:** Specifies whether to draw a horizontal rule at the given value. If the check-box is on, a horizontal grid line is drawn at the vertical position of the specified Y value, if that value is within the Y range of the waveform.

As with Min and Max, the Y value accepts a floating point number but truncates it to an integer if the radix of the selected wave objects is integral.



---

**IMPORTANT:** Analog settings are saved in a wave configuration; however, because control of zooming in the Y dimension is highly interactive, unlike other wave object properties such as radix, they do not affect the modification state of the wave configuration. Consequently, zoom settings are not saved with the wave configuration.

---

## Zoom Gestures

In addition to the zoom gestures supported for zooming in the X dimension, when over an analog waveform, additional zoom gestures are available, as shown in Figure 6-5.

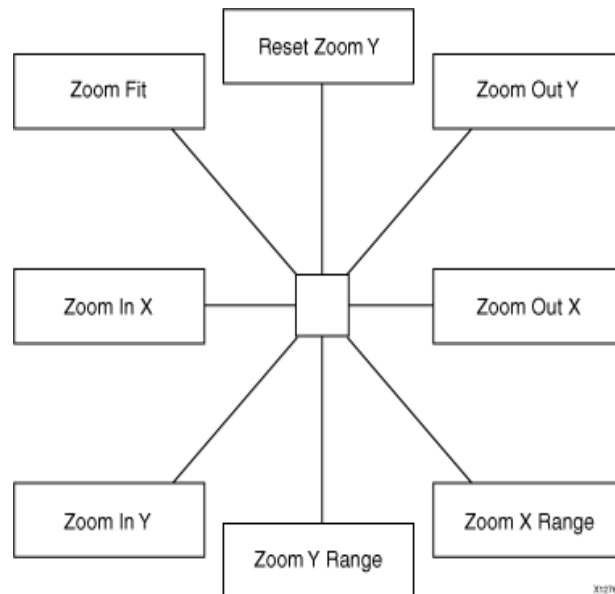


Figure 6-5: Analog Zoom Options

To invoke a zoom gesture, hold down the left mouse button and drag in the direction indicated in the diagram, where the starting mouse position is the center of the diagram.

The additional Zoom gestures are as follows:

- **Zoom Out Y:** Zooms out in the Y dimension by a power of 2 determined by how far away the mouse button is released from the starting point. The zoom is performed such that the Y value of the starting mouse position remains stationary.
- **Zoom Y Range:** Draws a vertical curtain which specifies the Y range to display when the mouse is released.
- **Zoom In Y:** Zooms in toward the Y dimension by a power of 2 determined by how far away the mouse button is released from the starting point.

The zoom is performed such that the Y value of the starting mouse position remains stationary.

- **Reset Zoom Y:** Resets the Y range to that of the values currently displayed in the wave window and sets the Y Range mode to **Auto**.

All zoom gestures in the Y dimension set the Y Range analog settings. **Reset Zoom Y** sets the Y Range to **Auto**, whereas the other gestures set Y Range to **Fixed**.

# In-System Serial I/O Debugging Flows

---

## Introduction

The Vivado® IDE provides a quick and easy way to generate a design that helps you debug and verify your system that uses Xilinx 7 Series high-speed gigabit transceiver (GT) technology. The in-system serial I/O debugging flow has three distinct phases:

1. IBERT Core generation phase: Customizing and generating the IBERT core that best meets your hardware high-speed serial I/O requirements.
2. IBERT Example Design Generation and Implementation phase: Generating the example design for the IBERT core generated in the previous step.
3. Serial I/O Analysis phase: Interacting with the IBERT IP contained in the design to debug and verify issues in your high-speed serial I/O links.

The rest of this chapter shows how to complete the first two phases. The third phase is covered in the chapter called Debugging the Serial I/O Design in Hardware.

---

## Generating an IBERT Core using the Vivado IP Catalog

The first phase of getting a suitable hardware design to help debug and validate your system's high-speed serial I/O interfaces is to generate the IBERT core. The following steps outline how to do this:

1. Open the Vivado IDE
2. On the first panel, choose **Manage IP > New IP Location**, then click **Next** when the **Open IP Catalog** wizard appears.
3. Select the desired part, target language, target simulator, and IP location. Click **Finish**.
4. In the **IP Catalog** under **Debug and Verification > Debug**, you will find one or more available IBERT cores as shown in [Figure 7-1](#), depending on the device selected in the previous step.

- Double-click on the IBERT architecture desire to open the Customize IP Wizard for that core

Customize the IBERT core for your given hardware system requirements. For details on the various IBERT cores available, see the following IP Documents: LogiCORE IP IBERT for 7 Series GTX Transceivers (PG132) [Ref 9], LogiCORE IP IBERT for 7 Series GTP Transceivers (PG133) [Ref 10], LogiCORE IP IBERT for 7 Series GTH Transceivers (PG152) [Ref 11].

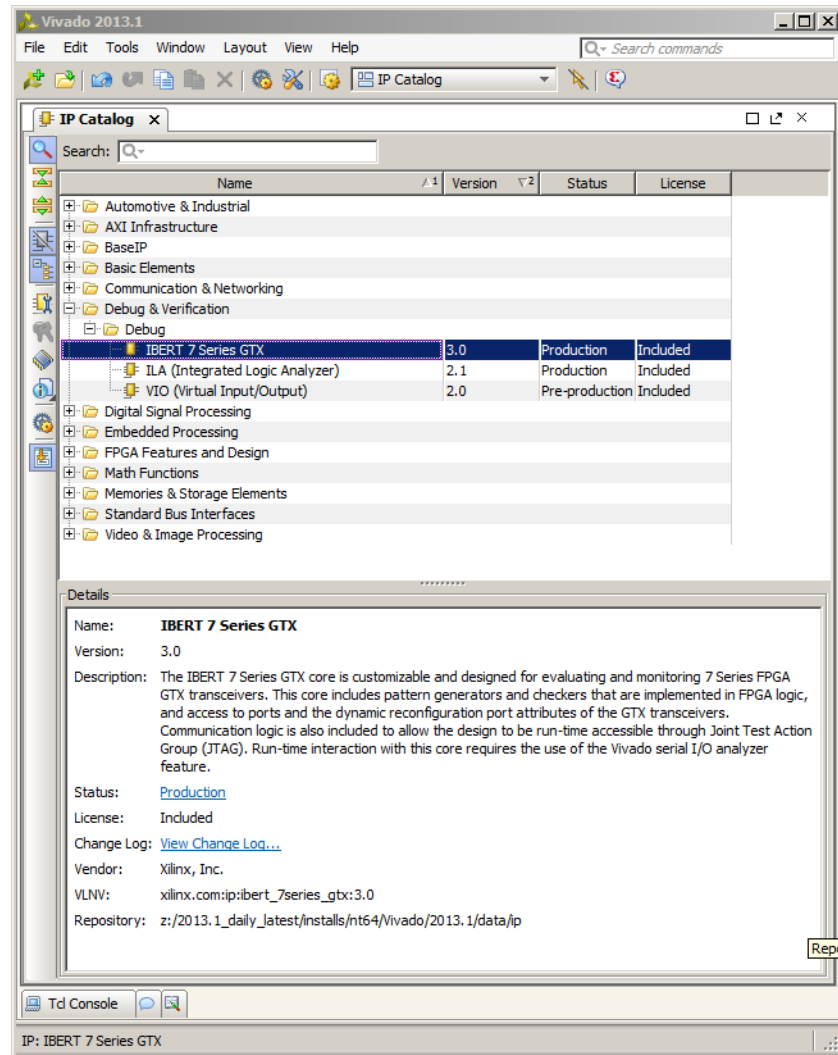


Figure 7-1: IP Catalog Showing the IBERT 7 Series GTX Core

## Generating and Implementing the IBERT Example Design

After generating the IBERT IP core, it appears in the Sources window as "ibert\_7series\_gtx" or something similar. To generate the example design, right-click

on the IBERT IP in the **Sources** window and select **Open IP Example Design**, then specify the desired location of the example design project in the resulting dialog window. This command opens a new Vivado project window for the example design and adds the proper top-level wrapper and constraints file to the project, as shown in [Figure 7-2](#).

Once the example design is generated, you can implement the IBERT example design through bitstream creation core by clicking **Generate Bitstream** in the **Program and Debug** section of the Vivado IDE flow navigator or by running the following Tcl commands:

```
launch_runs impl_1 -to_step write_bitstream
wait_on_run impl_1
```

6. Refer to the *Vivado Design Suite User Guide: Design Flows Overview (UG892)* [\[Ref 5\]](#) for more details on the various ways you can implement your design.

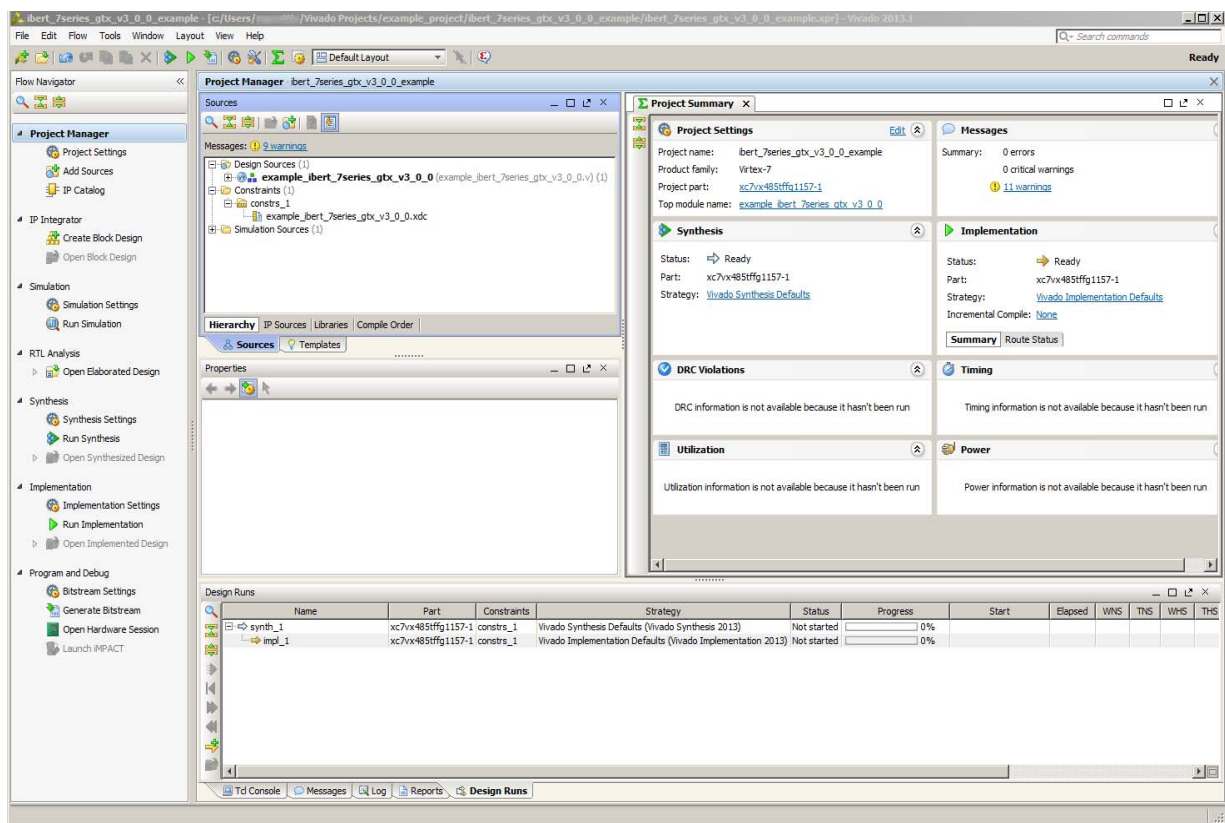


Figure 7-2: IBERT Example Design

# Debugging the Serial I/O Design in Hardware

---

## Introduction

Once you have IBERT core implemented, you can use the run time serial I/O analyzer features to debug the design in hardware. Only IBERT cores version v3.x and later can be accessed using the serial I/O analyzer feature.

---

## Using Vivado® Serial I/O Analyzer to Debug the Design

The Vivado® serial I/O analyzer feature is used to interact with IBERT v3.0 debug IP cores that are in your design. To access the Vivado serial I/O analyzer feature, click the **Open Hardware Manager** button in the Program and Debug section of the Flow Navigator.

The steps to debug your design in hardware are:

1. Connect to the hardware target and programming the FPGA device with the bit file.
2. Create Links.
3. Modify link settings and examine status.
4. Run scans as needed.

## Connecting to the Hardware Target and Programming the FPGA Device

Programming an FPGA device prior to debugging involves exactly the same steps described in [Using a Vivado Hardware Manager to Program an FPGA Device in Chapter 2](#). After programming the device with the `.bit` file that contains the IBERT v3.0 core, the **Hardware** window now shows the components of the IBERT core that were detected when scanning the device (see [Figure 8-1](#)).

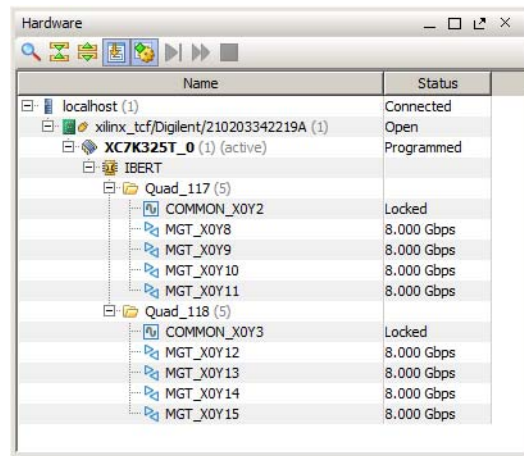


Figure 8-1: Hardware Window Showing the IBERT Core

## Creating Links and Link Groups

The IBERT core present in the design appears in the Hardware window under the target device. If you do not see the core appear, right-click on the device and select the **Refresh Hardware** command. This re-scans the FPGA device and refreshes the **Hardware** window.

**Note:** If you still do not see the IBERT core after programming and/or refreshing the FPGA device, check to make sure the device was programmed with the appropriate `.bit` file. Also check to make sure the implemented design contains an IBERT v3.0 core.

The Vivado serial I/O analyzer feature is built around the concept of links. A link is analogous to a channel on a board, with a transmitter and a receiver. The transmitter and receiver may or may not be the same GT, on the same device, or be the same architecture. To create one or more links, go to the **Links** tab in Vivado, and click either the **Create Links** button, or right-click and choose **Create Links**. This causes the **Create Links** dialog window to appear, as shown in Figure 8-2.

When an IBERT core is detected, the Hardware Manager notes that there are no links present, and show a green banner at the top. Click **\*Create Links\*** to open the **Create Links** dialog window, as shown in Figure 8-2.

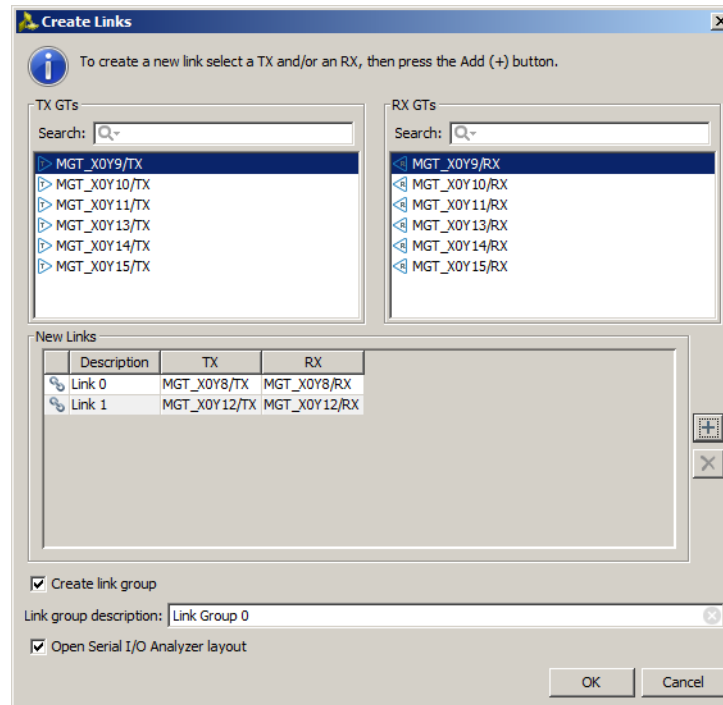


Figure 8-2: Create Links Dialog

Choose a TX and/or an RX from the list available. Or type in a string into the search field to narrow down the list. Then click the Add (+) button to add the link to the list. Repeat for all links desired.



**IMPORTANT:** A given TX or RX endpoint can only belong to one link.

Links can also be a part of a link group. By default, all new links are grouped together. You can choose not to add the links to a group by unchecking **Create link group** check box. The name of the link group is specified in the Link group description field.



## Viewing and Changing Links Settings Using the Links Window

Once links are created, they are added to the **Link** view (see [Figure 8-3](#)) which is the primary and best way to change link settings and view status.

Name	TX	RX	Status	Bits	Errors	BER	BERT Reset	TX Pattern	RX Pattern	TX Pre-Cursor	TX Post-Cursor	TX Diff Swing	DFE Enabled	Inject Error	TX Reset	RX Reset	RX PLL Status	TX PLL Status
Link 0	MGT_X0Y8/TX	MGT_X0Y8/RX	8.000 Gbps	1.106E14	1.02E11	9.21E-4	Reset	PRBS 7-bit	PRBS 7-bit	1.67 dB (00111)	0.68 dB (00011)	850 mV (1100)	<input checked="" type="checkbox"/>	Inject	Reset	Reset	Locked	Locked
Link 1	MGT_X0Y12/TX	MGT_X0Y12/RX	8.000 Gbps	1.106E14	0E0	9.038E-15	Reset	PRBS 7-bit	PRBS 7-bit	1.67 dB (00111)	0.68 dB (00011)	850 mV (1100)	<input checked="" type="checkbox"/>	Inject	Reset	Reset	Locked	Locked
Link 2	MGT_X0Y13/TX	MGT_X0Y13/RX	8.000 Gbps	1.106E14	0E0	9.038E-15	Reset	PRBS 7-bit	PRBS 7-bit	1.67 dB (00111)	0.68 dB (00011)	850 mV (1100)	<input checked="" type="checkbox"/>	Inject	Reset	Reset	Locked	Locked
Link 3	MGT_X0Y14/TX	MGT_X0Y14/RX	8.000 Gbps	1.106E14	0E0	9.038E-15	Reset	PRBS 7-bit	PRBS 7-bit	1.67 dB (00111)	0.68 dB (00011)	850 mV (1100)	<input checked="" type="checkbox"/>	Inject	Reset	Reset	Locked	Locked
Link 4	MGT_X0Y15/TX	MGT_X0Y15/RX	8.000 Gbps	1.107E14	0E0	9.037E-15	Reset	PRBS 7-bit	PRBS 7-bit	1.67 dB (00111)	0.68 dB (00011)	850 mV (1100)	<input checked="" type="checkbox"/>	Inject	Reset	Reset	Locked	Locked

Figure 8-3: Links Window

Each row in the **Links** window represents a link. Common and useful status and controls are enabled by default, so the health of the links can be quickly seen. The various settings that can be viewed in the Links window's table columns are shown in [Table 8-1](#).

Table 8-1: Links Window Settings

Link View Column Name	Description
Name	The name of the link
TX	The GT location of the transmitter
RX	The GT location of the receiver
Status	If linked (meaning the incoming RX data as expected). Status displays the measured line rate. Otherwise, it displays "No Link".
Bits	The measured number of bits received.
Errors	The measured number of bit errors by the receiver.
BER	Bit Error Ratio = (1 + Errors) / (Bits).
BERT Reset	Resets the received bits and error counters.
RX Pattern	Selects which pattern the receiver is expecting.
TX Pattern	Selects which pattern the transmitter is sending.
TX Pre-Cursor	Selects the pre-cursor emphasis on the transmitter.
TX Post-Cursor	Selects the post-cursor emphasis on the transmitter.
TX Diff Swing	Selects the differential swing values for the transmitter.
DFE Enabled	Selects whether the Decision Feedback Equalizer is enabled on the receiver (not available for all architectures).
Inject Error	Injects a single bit error into the transmit path.
TX Reset	Resets the transmitter.
RX Reset	Resets the receiver and BERT counters (see BERT Reset).

Table 8-1: Links Window Settings

Link View Column Name	Description
Loopback Mode	Selects the loopback mode on the receiver GT. Warning: Changing this value might effect the link status depending on the system topology.
Termination Voltage	Selects the termination voltage of the receiver.
RX Common Mode	Selects the RX Commn Mode setting of the receiver.
TXUSERCLK Freq	Shows the measured TXUSERCLK frequency in MHz.
TXUSERCLK2 Freq	Shows the measured TXUSERCLK2 frequency in MHz.
RXUSERCLK Freq	Shows the measured RXUSERCLK frequency in MHz.
RXUSERCLK2 Freq	Shows the measured RXUSERCLK2 frequency in MHz.
TX Polarity Invert	Inverts the polarity of the transmitted data.
RX Polarity Invert	Inverts the polarity of the received data.

It is possible to change the values of a given property for all links in a link group by changing the setting in the link group row. For instance, changing the TX Pattern to "PRBS 7-bit" in the "Link Group 0" row changes the TX Pattern of all the links to "PRBS 7-bit". If not all the links in the group have the same setting, "Multiple" appears for that column in the link group row.

## Creating and Running Link Scans

To analyze the margin of a given link, it is often helpful to run a scan of the link using the specialized Eye Scan hardware of the Xilinx 7 Series FPGA transceivers. The Vivado serial I/O analyzer feature enables you to define, run, save, and recall link scans.

A scan runs on a link. To create a scan, select a link in the **Link** window, and either right-click and choose **Create Scan**, or click the **Create Scan** button in the Link window toolbar. This brings up the **Create Scan** dialog (see [Figure 8-4](#)). The **Create Scan** dialog shows the settings for performing a scan, as shown in [Table 8-2](#).

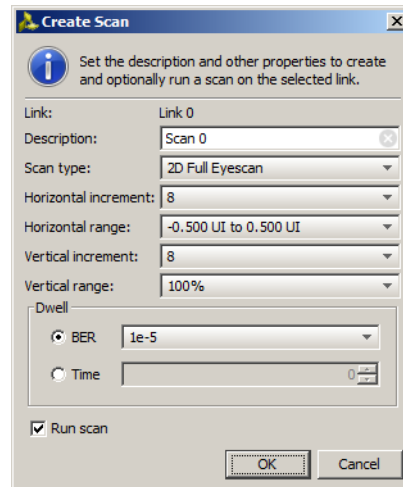


Figure 8-4: Create Scan Dialog

Table 8-2: Scan Settings

Scan Setting	Description
Description	A user-defined name for the scan.
Scan Type	The type of scan to run.
Horizontal Increment	Allows you to choose to scan the eye at a reduced resolution, but at increased speed by skipping horizontal codes.
Horizontal Range	Reducing the horizontal range increases the scan speed. By default, the entire eye is scanned (-1/2 of a unit interval to +1/2 in reference to the center of the eye).
Vertical Increment	Allows you to choose to scan the eye at a reduced resolution, but increased speed by skipping vertical codes.
Vertical Range	Reducing the vertical range increases the scan speed. By default, the entire eye is scanned.
Dwell BER	Each point in the chart is scanned for a certain amount of time. Dwell BER allows you to choose the scan depth by selecting the desired Bit Error Ratio.
Dwell Time	Dwell Time allows you to choose the scan depth by inputting the desired time in seconds.

By default, the scan is run after it is created. If you do not want to run the scan, and only define it, uncheck the **Run Scan** checkbox.

If a scan is created, but not run, it can be subsequently run or run by right-clicking on a scan in the **Scans** window and choosing **Run Scan** (see Figure 8-5). While a scan is running, it can be prematurely stopped by right-clicking on a scan and choosing **Stop Scan**, or clicking the **Stop Scan** button in the **Scans** window toolbar.

Name	Link	Scan Type	Status	Progress	Open...	Horz Incr	Horz Range	Vert Incr	Vert Range	Dwell	Dwell BER	Dwell Time	Start Time	End Time
Scans (2)														
Scan 0	Link 0	2d_full_eye	Done	100%	5568	8	-0.500 UI to 0.500 UI	8	100%	BER	1e-5		0 2013-May-20 15:26:29	2013-May-20 15:26:47
Scan 1	Link 0	2d_full_eye	Done	100%	5836	2	-0.500 UI to 0.500 UI	2	100%	BER	1e-5		0 2013-May-20 15:27:41	2013-May-20 15:30:36

Figure 8-5: Scans Window

## Displaying and Navigating the Scan Plots

After a scan is created, it automatically launches the **Scan Plots** window for that scan. For 2D Eyescan, the plot is a heat map of the BER value.

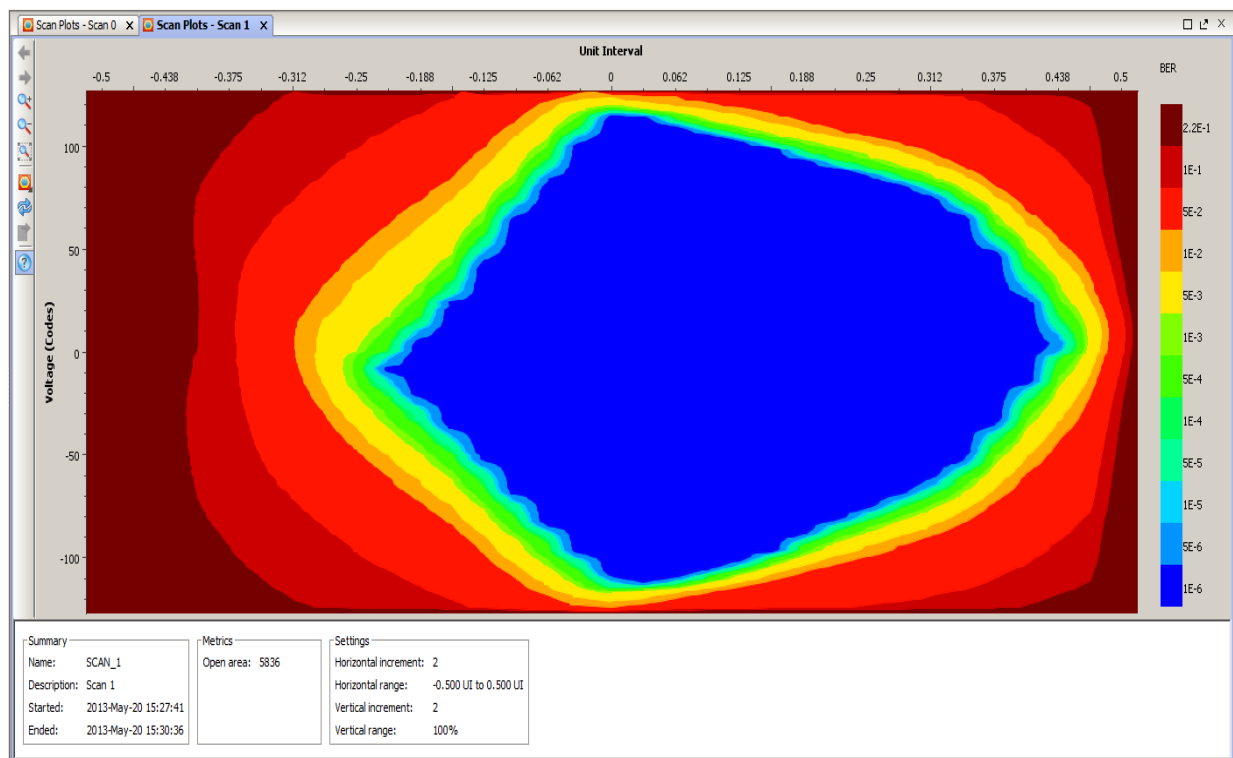


Figure 8-6: Scan Plot Window

As in other charts and displays within the Vivado IDE, the mouse gestures for zooming in the eye scan plot window are as follows:

- Zoom Area: left-click drag from top-left to bottom-right
- Zoom Fit: left-click drag from bottom-right to top-left

- Zoom In: left-click drag from top-right to bottom-left
- Zoom Out: left-click drag from bottom-left to top-right

Also, when the mouse cursor is over the Plot, the current horizontal and vertical codes, along with the scanned BER value is displayed in the tooltip. You can also change the plot type by clicking the **\*Plot Type\*** button in the plot window, and choosing **Show Contour (filled)**, **Show Contour (lines)**, and **Heat Map**.

A summary view is present at the bottom of the scan plot, stating the scan settings, along with basic information like when the scan was performed. During the 2D Eyescan, the number of pixels in the scan with zero errors is calculated (taking into account the horizontal and vertical increments), and this result is displayed as Open Area. The **\*Scan\*** window contents are sorted by **Open Area** by default, so the scans with the largest open area appear at the top.

## Writing the Scan Results to a File

When scan data exists due to a partial or full 2D Eyescan, these results can be written to a CSV file by clicking the **Write Scan** button in the **Scans Window**. This saves the scan results to comma-delimited file, with the BER values in a block that replicated the scan plot.

## Properties Window

Whenever a GT or a COMMON block in the hardware window, a Link in the **Links** window, or a scan in the **Scans** window is selected, the properties of that object shows in the **Properties** window. For GTs and COMMONs, these include all the attribute, port, and other settings of those objects. These settings can be changed in the **Properties** window (see [Figure 8-7](#)), or by writing Tcl commands to change and commit the properties. Some properties are read-only and cannot be changed.

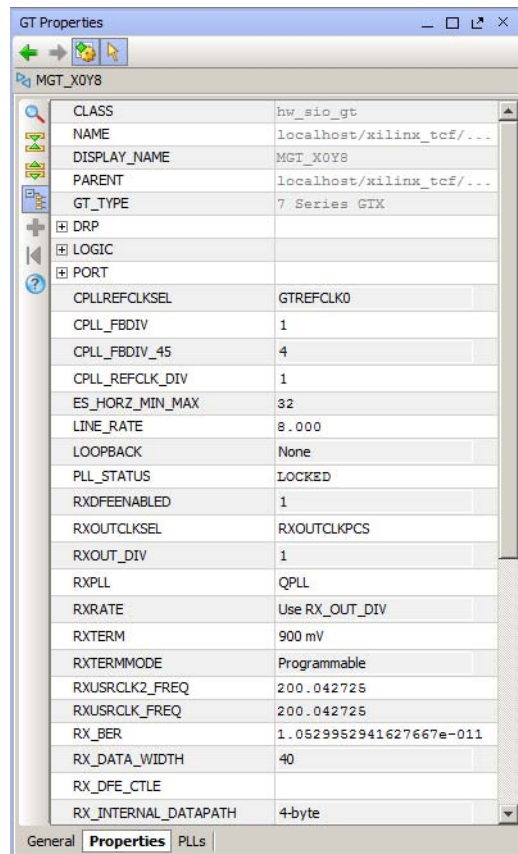


Figure 8-7: Properties Window

## Description of Serial I/O Analyzer Tcl Objects and Commands

You can use Tcl commands to interact with your hardware under test. The hardware is organized in a set of hierarchical first class Tcl objects (see [Table 8-3](#)).

Table 8-3: Serial I/O Analyzer Tcl Objects

Tcl Object	Description
hw_sio_ibert	Object referring to an IBERT core. Each IBERT object can have one or more hw_sio_gt, or hw_sio_common objects associated with it.
hw_sio_gt	Object referring to a single Xilinx Gigabit Transceiver (GT).
hw_sio_gtgroups	Object referring to a logical grouping of GTs, could be a Quad or an Octal.
hw_sio_common	Object referring to a COMMON block.
hw_sio_tx	Object referring to the transmitter side of a hw_sio_gt. Only the TX related ports, attributes, and logic properties flows to the hw_sio_tx.
hw_sio_rx	Object referring to the receiver side of a hw_sio_gt. Only the RX related ports, attributes, and logic properties flows to the hw_sio_rx.

Table 8-3: Serial I/O Analyzer Tcl Objects

Tcl Object	Description
hw_sio_pll	Object referring to a PLL in either an hw_sio_gt or an hw_sio_common object. Only the related ports, attributes, and logic properties flow to the hw_sio_pll.
hw_sio_link	Object referring to a link, a TX-RX pair. <b>Note:</b> A link can also consist of a TX only or an RX only.
hw_sio_linkgroup	Object referring to a group of links.
hw_sio_scan	Object referring to a margin analysis scan.

For more information about the hardware manager commands, run the **help -category hardware** Tcl command in the Tcl Console.

## Description of Tcl Commands to Access Hardware

Table 8-4 contains descriptions of all Tcl commands used to interact with the IBERT core.



**IMPORTANT:** Using the *get\_property* or *set\_property* command does not read or write information to/from the IBERT core. You must use the *refresh\_hw\_sio* and *commit\_hw\_sio* commands to read and write information from/to the hardware, respectively.

Table 8-4: Descriptions of hw\_server Tcl Commands

Tcl Command	Description
refresh_hw_sio	Read the property values out of the provided object. Works for any hw_sio object that refers to hardware.
commit_hw_sio	Writes property changes to the hardware. Works for any hw_sio object that refers to hardware.

## Description of hw\_sio\_link Tcl Commands

Table 8-5 contains descriptions of all Tcl commands used to interact with links.

Table 8-5: Descriptions of hw\_sio\_link Tcl Commands

Tcl Command	Description
create_hw_sio_link	Create an hw_sio_link object with the given hw_sio_rx and/or hw_sio_tx objects.
remove_hw_sio_link	Deletes the given link.
get_hw_sio_links	Get list of hw_sio_links for the given object.

## Description of hw\_sio\_linkgroup Tcl Commands

Table 8-6 contains descriptions of all Tcl commands used to interact with linkgroups.

Table 8-6: Descriptions of hw\_sio\_linkgroup Tcl Commands

Tcl Command	Description
create_hw_sio_linkgroup	Create an hw_sio_linkgroup object with the hw_sio_link objects.
remove_hw_sio_linkgroup	Deletes the given linkgroup.
get_hw_sio_linkgroups	Get list of hw_sio_linkgroups for the given object.

## Description of hw\_sio\_scan Tcl Commands

Table 8-7 contains descriptions of all Tcl commands used to interact with scans.

Table 8-7: Descriptions of hw\_sio\_scan Tcl Commands

Tcl Command	Description
create_hw_sio_scan	Creates a scan object.
remove_hw_sio_scan	Deletes a scan object.
run_hw_sio_scan	Runs a scan.
stop_hw_sio_scan	Stops a scan.
wait_on_hw_sio_scan	Blocks the Tcl console prompt until a given run_hw_sio_scan operation is complete.
display_hw_sio_scan	Displays a partial or complete scan in the Scan Plot.
write_hw_sio_scan	Writes the scan data to a file.
read_hw_sio_scan	Reads scan data from a file into a scan object.
get_hw_sio_scans	Get a list of hw_sio_scan objects.

## Description of Tcl Commands to Get Objects

Table 8-8 contains descriptions of all Tcl commands used to get serial I/O objects.

Table 8-8: Descriptions of Tcl Commands to Get Objects

Tcl Command	Description
get_hw_sio_iberts	Get list of IBERT objects.
get_hw_sio_gts	Get list of GTs.
get_hw_sio_commons	Get list of COMMON blocks.



**Table 8-8: Descriptions of Tcl Commands to Get Objects**

Tcl Command	Description
get_hw_sio_txs	Get list of transmitters.
get_hw_sio_rxs	Get list of receivers.
get_hw_sio_plls	Get list of PLLs.
get_hw_sio_links	Get list of links.
get_hw_sio_linkgroups	Get list of linkgroups.
get_hw_sio_scans	Get list of scans.

## Using Hardware Manager Tcl Commands

Below is an example Tcl command script that interacts with the following example system

- One KC705 board's Digilent JTAG-SMT1 cable (serial number 12345) accessible via a CSE server running on localhost:50001
- Single IBERT core in a design running in the XC7K325T device on the KC705 board
- IBERT core has Quad 117 and Quad 118 enabled

### Example Tcl Command Script

```
# Connect to the Digilent Cable on localhost:50001
connect_hw_server -host localhost -port 50001
current_hw_target [get_hw_targets */digilent_plugin/SN:12345]
open_hw_target

# Program and Refresh the XC7K325T Device
current_hw_device [lindex [get_hw_devices] 0]
refresh_hw_device -update_hw_probes false [lindex [get_hw_devices] 0]
set_property PROGRAM.FILE {C:/design.bit} [lindex [get_hw_devices] 0]
program_hw_devices [lindex [get_hw_devices] 0]
refresh_hw_device [lindex [get_hw_devices] 0]

# Set Up Link on first GT
set tx0 [lindex [get_hw_sio_txs] 0]
set rx0 [lindex [get_hw_sio_rxs] 0]
set link0 [create_hw_sio_link $tx0 $rx0]
set_property DESCRIPTION {Link 0} [get_hw_sio_links $link0]

# Set link to use PCS Loopback, and write to hardware
set_property LOOPBACK "Near-End PCS" $link0
commit_hw_sio $link0

# Create, run, display and save scan
set scan0 [create_hw_sio_scan 2d_full_eye [get_hw_sio_rxs -of $link0]]
run_hw_sio_scan $scan0
display_hw_sio_scan $scan0
write_hw_sio_scan "scan0.csv" $scan0
```

## Device Configuration Bitstream Settings

This table describes all of the device configuration settings available for use with the `set_property <Bitstream Setting> <Value> [current_design]` Vivado® tool Tcl command.

**Table A-1: Bitstream Settings**

Settings	Default Value	Possible Values	Description
BITSTREAM.CONFIG.BPI_1ST_READ_CYCLE	1	1, 2, 3, 4	Helps synchronize BPI configuration with the timing of page mode operations in flash devices. It allows you to set the cycle number for a valid read of the first page. The BPI_page_size must be set to 4 or 8 for this option to be available.
BITSTREAM.CONFIG.BPI_PAGE_SIZE	1	1, 4, 8	For BPI configuration, this sub-option lets you specify the page size which corresponds to the number of reads required per page of flash memory.
BITSTREAM.CONFIG.BPI_SYNC_MODE	Disable	Disable, Type1, Type2	Sets the BPI synchronous configuration mode for different types of BPI flash devices. <ul style="list-style-type: none"> <li>• Disable (the default) disables the synchronous configuration mode.</li> <li>• Type1 enables the synchronous configuration mode and settings to support the Micron G18(F) family.</li> <li>• Type2 enables the synchronous configuration mode and settings to support the Micron (Numonyx) P30 family.</li> </ul>
BITSTREAM.CONFIG.CCLKPIN	Pullup	Pullup, Pullnone	Adds an internal pull-up to the Cclk pin. The Pullnone setting disables the pullup.
BITSTREAM.CONFIG.CONFIGFALLBACK	Disable	Disable, Enable	Enables or disables the loading of a default bitstream when a configuration attempt fails.
BITSTREAM.CONFIG.CONFIGRATE	3	3, 6, 9, 12, 16, 22, 26, 33, 40, 50, 66	Bitstream generation uses an internal oscillator to generate the configuration clock, Cclk, when configuring is in a master mode. Use this sub-option to select the rate for Cclk.
BITSTREAM.CONFIG.DCIUPDATEMODE	AsRequired	AsRequired, Continuous, Quiet	Controls how often the Digitally Controlled Impedance circuit attempts to update the impedance match for DCI IOSTANDARDS.

**Table A-1: Bitstream Settings (Cont'd)**

Settings	Default Value	Possible Values	Description
BITSTREAM.CONFIG.DONEPIN	Pullup	Pullup, Pullnone	Adds an internal pull-up to the DONE pin. The Pullnone setting disables the pullup. Use DonePin only if you intend to connect an external pull-up resistor to this pin. The internal pull-up resistor is automatically connected if you do not use DonePin.
BITSTREAM.CONFIG.EXTMASTERCLK_EN	Disable	Disable, div-8, div-4, div-2, div-1	Allows an external clock to be used as the configuration clock for all master modes. The external clock must be connected to the dual-purpose USERCLK pin.
BITSTREAM.CONFIG.INITPIN	Pullup	Pullup, Pullnone	Specifies whether you want to add a Pullup resistor to the INIT pin, or leave the INIT pin floating.
BITSTREAM.CONFIG.INITSIGNALSERROR	Enable	Enable, Disable	When Enabled, the INIT_B pin asserts to '0' when a configuration error is detected.
BITSTREAM.CONFIG.M0PIN	Pullup	Pullup, Pulldown, Pullnone	Adds an internal pull-up, pull-down, or neither to the M0 pin. Select Pullnone to disable both the pull-up resistor and the pull-down resistor on the M0 pin.
BITSTREAM.CONFIG.M1PIN	Pullup	Pullup, Pulldown, Pullnone	Adds an internal pull-up, pull-down, or neither to the M1 pin. Select Pullnone to disable both the pull-up resistor and the pull-down resistor on the M1 pin.
BITSTREAM.CONFIG.M2PIN	Pullup	Pullup, Pulldown, Pullnone	Adds an internal pull-up, pull-down, or neither to the M2 pin. Select Pullnone to disable both the pull-up resistor and the pull-down resistor on the M2 pin.
BITSTREAM.CONFIG.NEXT_CONFIG_ADDR	none	<string>	Sets the starting address for the next configuration in a MultiBoot set up, which is stored in the General1 and General2 registers.
BITSTREAM.CONFIG.NEXT_CONFIG_REBOOT	Enable	Enable, Disable	When set to Disable the IPROG command is removed from the bitfile.
BITSTREAM.CONFIG.OVERTEMPPOWERDOWN	Disable	Disable, Enable	Enables the device to shut down when the system monitor detects a temperature beyond the acceptable operational maximum. An external circuitry set up for the System Monitor is required to use this option.
BITSTREAM.CONFIG.PERSIST	No	No, Yes	Prohibits use of the SelectMAP mode pins for use as user I/O. Refer to the data sheet for a description of SelectMAP mode and the associated pins. Persist is needed for Readback and Partial Reconfiguration using the SelectMAP configuration pins, and should be used when either SelectMAP or Serial modes are used. Only the SelectMAP pins are affected, but this option should be used for access to config pins (other than JTAG) after configuration.
BITSTREAM.CONFIG.PROGPIN	Pullup	Pullup, Pullnone	Adds an internal pull-up to the ProgPin pin. The Pullnone setting disables the pullup. The pullup affects the pin after configuration.

**Table A-1: Bitstream Settings (Cont'd)**

Settings	Default Value	Possible Values	Description
BITSTREAM.CONFIG. REVISIONSELECT	00	00, 01, 10, 11	Specifies the internal value of the RS[1:0] settings in the Warm Boot Start Address (WBSTAR) register for the next warm boot.
BITSTREAM.CONFIG. REVISIONSELECT_ TRISTATE	Disable	Disable, Enable	Specifies the whether the RS[1:0] 3-state is enabled by setting the option in the Warm Boot Start Address (WBSTAR).
BITSTREAM.CONFIG. SELECTMAPABORT	Enable	Enable, Disable	Enables or disables the SelectMAP mode Abort sequence. If disabled, an Abort sequence on the device pins is ignored.
BITSTREAM.CONFIG. SPI_32BIT_ADDR	No	No, Yes	Enables SPI 32-bit address style, which is required for SPI devices with storage of 256 Mb and larger.
BITSTREAM.CONFIG. SPI_BUSWIDTH	1	1, 2, 4	Sets the SPI bus to Dual (x2) or Quad (x4) mode for Master SPI configuration from third party SPI flash devices.
BITSTREAM.CONFIG. SPI_FALL_EDGE	No	No, Yes	Sets the FPGA to use a falling edge clock for SPI data capture. This improves timing margins and may allow faster clock rates for configuration.
BITSTREAM.CONFIG. TCKPIN	Pullup	Pullup, Pulldown, Pullnone	Adds a pull-up, a pull-down, or neither to the TCK pin, the JTAG test clock. The Pullnone setting shows that there is no connection to either the pull-up or the pull-down.
BITSTREAM.CONFIG. TDIPIN	Pullup	Pullup, Pulldown, Pullnone	Adds a pull-up, a pull-down, or neither to the TDI pin, the serial data input to all JTAG instructions and JTAG registers. The Pullnone setting shows that there is no connection to either the pull-up or the pull-down.
BITSTREAM.CONFIG. TDOPIN	Pullup	Pullup, Pulldown, Pullnone	Adds a pull-up, a pull-down, or neither to the TdoPin pin, the serial data output for all JTAG instruction and data registers. The Pullnone setting shows that there is no connection to either the pull-up or the pull-down.
BITSTREAM.CONFIG. TIMER_CFG	none	<8-digit hex string>	Sets the value of the Watchdog Timer in Configuration mode. This option cannot be used at the same time as TIMER_USR.
BITSTREAM.CONFIG. TIMER_USR	0x00000000	<8-digit hex string>	Sets the value of the Watchdog Timer in User mode. This option cannot be used at the same time as TIMER_CFG.
BITSTREAM.CONFIG. TMSPIN	Pullup	Pullup, Pulldown, Pullnone	Adds a pull-up, pull-down, or neither to the TMS pin, the mode input signal to the TAP controller. The TAP controller provides the control logic for JTAG. The Pullnone setting shows that there is no connection to either the pull-up or the pull-down.

Table A-1: Bitstream Settings (Cont'd)

Settings	Default Value	Possible Values	Description
BITSTREAM.CONFIG.UNUSEDPIN	Pulldown	Pulldown, Pullup, Pullnone	Adds a pull-up, a pull-down, or neither to unused SelectIO pins (IOBs). It has no effect on dedicated configuration pins. The list of dedicated configuration pins varies depending upon the architecture. The Pullnone setting shows that there is no connection to either the pull-up or the pull-down.
BITSTREAM.CONFIG.USERID	0xFFFFFFFF	<8-digit hex string>	Used to identify implementation revisions. You can enter up to an 8-digit hexadecimal string in the User ID register.
BITSTREAM.CONFIG.USR_ACCESS	None	none, <8-digit hex string>, TIMESTAMP	Writes an 8-digit hexadecimal string, or a timestamp into the AXSS configuration register. The format of the timestamp value is dddddd MMMM yyyy hh mm ss : day, month, year (year 2000 = 00000), hour, minute, seconds. The contents of this register may be directly accessed by the FPGA fabric via the USR_ACCESS primitive.
BITSTREAM.ENCRYPTION.ENCRYPT	No	No Yes	Encrypts the bitstream.
BITSTREAM.ENCRYPTION.ENCRYPTKEYSELECT	bbram	bbram, efuse	Determines the location of the AES encryption key to be used, either from the battery-backed RAM (BBRAM) or the eFUSE register. <b>Note:</b> This property is only available when the Encrypt option is set to True.
BITSTREAM.ENCRYPTION.HKEY	Pick	Pick, <hex string>	HKey sets the HMAC authentication key for bitstream encryption. 7 series devices have an on-chip bitstream-keyed Hash Message Authentication Code (HMAC) algorithm implemented in hardware to provide additional security beyond AES decryption alone. These devices require both AES and HMAC keys to load, modify, intercept, or clone the bitstream. The pick setting tells the bitstream generator to select a random number for the value. To use this option, you must first set Encrypt to Yes
BITSTREAM.ENCRYPTION.KEY0	Pick	Pick, <hex string>	Key0 sets the AES encryption key for bitstream encryption. The pick setting tells the bitstream generator to select a random number for the value. To use this option, you must first set Encrypt to Yes.
BITSTREAM.ENCRYPTION.KEYFILE	none	<string>	Specifies the name of the input encryption file (with a .nky file extension). To use this option, you must first set Encrypt to Yes.
BITSTREAM.ENCRYPTION.STARTCBC	Pick	Pick, <32-bit hex string>	Sets the starting cipher block chaining (CBC) value. The pick setting enables selection of a random number for the value.

Table A-1: Bitstream Settings (Cont'd)

Settings	Default Value	Possible Values	Description
BITSTREAM.GENERAL.COMPRESS	False	True, False	Uses the multiple frame write feature in the bitstream to reduce the size of the bitstream, not just the Bitstream (.bit) file. Using Compress does not guarantee that the size of the bitstream shrinks.
BITSTREAM.GENERAL.CRC	Enable	Enable, Disable	Controls the generation of a Cyclic Redundancy Check (CRC) value in the bitstream. When enabled, a unique CRC value is calculated based on bitstream contents. If the calculated CRC value does not match the CRC value in the bitstream, the device will fail to configure. When CRC is disabled a constant value is inserted in the bitstream in place of the CRC, and the device does not calculate a CRC.
BITSTREAM.GENERAL.DEBUGBITSTREAM	No	No, Yes	Lets you create a debug bitstream. A debug bitstream is significantly larger than a standard bitstream. DebugBitstream can be used only for master and slave serial configurations. DebugBitstream is not valid for Boundary Scan or Slave Parallel/SelectMAP. In addition to a standard bitstream, a debug bitstream offers the following features: <ul style="list-style-type: none"> <li>Writes 32 0s to the LOUT register after the synchronization word.</li> <li>Loads each frame individually.</li> <li>Performs a Cyclic Redundancy Check (CRC) after each frame.</li> <li>Writes the frame address to the LOUT register after each frame.</li> </ul>
BITSTREAM.GENERAL.DISABLE_JTAG	No	No, Yes	Disables communication to the Boundary Scan (BSCAN) block via JTAG after configuration.
BITSTREAM.GENERAL.JTAG_XADC	Enable	Enable, Disable, StatusOnly	Enables or disables the JTAG connection to the XADC.
BITSTREAM.GENERAL.XADCENHANCEDLINEARITY	Off	Off, On	Disables some built-in digital calibration features that make INL look worse than the actual analog performance.
BITSTREAM.READBACK.ACTIVERECONFIG	No	No, Yes	Prevents the assertions of GHIGH and GSR during configuration. This is required for the active partial reconfiguration enhancement features.
BITSTREAM.READBACK.ICAP_SELECT	Auto	Auto, Top, Bottom	Selects between the top and bottom ICAP ports.
BITSTREAM.READBACK.READBACK	False	True, False	Lets you perform the Readback function by creating the necessary readback files.
BITSTREAM.READBACK.SECURITY	None	None, Level1, Level2	Specifies whether to disable Readback and Reconfiguration. <b>Note:</b> Specifying Security Level1 disables Readback. Specifying Security Level2 disables Readback and Reconfiguration.

**Table A-1: Bitstream Settings (Cont'd)**

Settings	Default Value	Possible Values	Description
BITSTREAM.READBACK.XADCPARTIALRECONFIG	Disable	Disable, Enable	When Disabled XADC can work continuously during Partial Reconfiguration. When Enabled XADC works in Safe mode during partial reconfiguration.
BITSTREAM.STARTUP.DONEPIPE	Yes	Yes, No	Tells the FPGA device to wait on the CFG_DONE (DONE) pin to go High and then wait or the first clock edge before moving to the Done state.
BITSTREAM.STARTUP.DONE_CYCLE	4	4, 1, 2, 3, 5, 6, Keep	Selects the Startup phase that activates the FPGA Done signal. Done is delayed when DonePipe=Yes.
BITSTREAM.STARTUP.GTS_CYCLE	5	5, 1, 2, 3, 4, 6, Done, Keep	Selects the Startup phase that releases the internal 3-state control to the I/O buffers.
BITSTREAM.STARTUP.GWE_CYCLE	6	6, 1, 2, 3, 4, 5, Done, Keep	Selects the Startup phase that asserts the internal write enable to flip-flops, LUT RAMs, and shift registers. GWE_cycle also enables the BRAMS. Before the Startup phase, both BRAM writing and reading are disabled.
BITSTREAM.STARTUP.LCK_CYCLE	NoWait	NoWait, 0, 1, 2, 3, 4, 5, 6	Selects the Startup phase to wait until DLLs/DCMs/PLLs lock. If you select NoWait, the Startup sequence does not wait for DLLs/DCMs/PLLs to lock.

Table A-1: Bitstream Settings (Cont'd)

Settings	Default Value	Possible Values	Description
BITSTREAM.STARTUP.MATCH_CYCLE	Auto	Auto, NoWait, 0, 1, 2, 3, 4, 5, 6	<p>Specifies a stall in the Startup cycle until digitally controlled impedance (DCI) match signals are asserted. DCI matching does not begin on the Match_cycle that was set in BitGen. The Startup sequence waits in this cycle until DCI has matched. Given that there are a number of variables in determining how long it takes DCI to match, the number of CCLK cycles required to complete the Startup sequence may vary in any given system. Ideally, the configuration solution should continue driving CCLK until DONE goes high.</p> <p><b>Note:</b> When the Auto setting is specified, write_bitstream searches the design for any DCI I/O standards. If DCI standards exist, write_bitstream uses BITSTREAM.STARTUP.MATCH_CYCLE=2. Otherwise, write_bitstream uses BITSTREAM.STARTUP.MATCH_CYCLE=NoWait.</p>
BITSTREAM.STARTUP.STARTUPCLK	Cclk	Cclk, UserClk, JtagClk	<p>The StartupClk sequence following the configuration of a device can be synchronized to either Cclk, a User Clock, or the JTAG Clock. The default is Cclk.</p> <ul style="list-style-type: none"> <li>• Cclk lets you synchronize to an internal clock provided in the FPGA device.</li> <li>• UserClk lets you synchronize to a user-defined signal connected to the CLK pin of the STARTUP symbol.</li> <li>• JtagClk lets you synchronize to the clock provided by JTAG. This clock sequences the TAP controller which provides the control logic for JTAG.</li> </ul>



# Trigger State Machine Language Description

The trigger state machine language is used to describe complex trigger conditions that map to the advanced trigger logic of the ILA debug core. The trigger state machine has the following features:

- Up to 16 states.
- One-, two-, and three-way conditional branching used for complex state transitions.
- Four built-in 16-bit counters used to count events, implement timers, etc.
- Four built-in flags used for monitoring trigger state machine execution status.
- Trigger action.

---

## States

Each state machine program can have up to 16 states declared. Each state is composed of a state declaration and a body:

```
state <state_name>:  
    <state_body>
```

---

## Goto Action

The `goto` action is used to transition between states. Here is an example of using the `goto` action to transition from one state to another before triggering:

```
state my_state_0:  
    goto my_state_1;  
  
state my_state_1:  
    trigger;
```

---

## Conditional Branching

The trigger state machine language supports one-, two-, and three-way conditional branching per state.

- One-way branching involves using `goto` actions without any `if/elseif/else/endif` constructs:

```
state my_state_0:
    goto my_state_1;
```

- Two-way conditional branching uses `goto` actions with `if/else/endif` constructs:

```
state my_state_0:
    if (<condition1>) then
        goto my_state_1;
    else
        goto my_state_0;
    endif
```

- Three-way conditional branching uses `goto` actions with `if/else/elseif/endif` constructs:

```
state my_state_0:
    if (<condition1>) then
        goto my_state_1;
    elseif (<condition2>) then
        goto my_state_0;
    endif
```

For more information on how to construct conditional statements represented above with `<condition1>` and `<condition2>`, refer to the section called [Conditional Statements](#), page 123

---

## Counters

The four built-in 16-bit counters have fixed names and are called `$counter0`, `$counter1`, `$counter2`, `$counter3`. The counters can be reset, incremented, and used in conditional statements.

- To reset a counter, use the `reset_counter` action:

```
state my_state_0:
    reset_counter $counter0;
    goto my_state_1;
```

- To increment a counter, use the `increment_counter` action:

```
state my_state_0:
    increment_counter $counter3;
    goto my_state_1;
```

For more information on how to use counters in conditional statements, refer to [Conditional Statements, page 123](#).

---

## Flags

Flags can be used to monitor progress of the trigger state machine program as it executes. The four built-in flags have fixed names and are called `$flag0`, `$flag1`, `$flag2`, and `$flag3`. The flags can be set and cleared.

- To set a flag, use the `set_flag` action:

```
state my_state_0:
  set_flag $flag0;
  goto my_state_1;
```

- To clear a flag, use the `clear_flag` action:

```
state my_state_0:
  clear_flag $flag2;
  goto my_state_1;
```

---

## Conditional Statements

### Debug Probe Conditions

Debug probe conditions can be used in two-way and three-way branching conditional statements. Each debug probe condition consumes one trigger comparator on the PROBE port of the ILA to which the debug probe is attached.



**IMPORTANT:** Each PROBE port can have from 1 to 4 trigger comparators as configured at compile time. This means that you can only use a particular debug probe in a debug probe condition up from 1 to 4 times in the entire trigger state machine program, depending on the number of comparators on the PROBE port. Also, if the debug probe shares a PROBE port of the ILA core with other debug probes, each debug probe condition will count towards the use of one PROBE comparator.

The debug probe conditions consist of a comparison operator and a value. The valid debug probe condition comparison operators are:

- `==` (equals)
- `!=` (not equals)
- `>` (greater than)
- `<` (less than)

- `>=` (greater than or equal to)
- `<=` (less than or equal to)

Valid values are of the form:

```
<bit_width>'<radix><value>
```

Where:

- `<bit width>` is the width of the probe (in bits)
- `<radix>` is one of
  - `b` (binary)
  - `h` (hexadecimal)
  - `u` (unsigned decimal)

Examples of valid debug probe condition values are:

- 1-bit binary value of 0  
`1'b0`
- 12-bit hex value of 7A  
`12'h07A`
- 9-bit integer value of 123  
`9'u123`

Examples of debug probe condition statements are:

- A single-bit debug probe called `abc` equals 0  
`if (abc == 1'b0) then`
- A 23-bit debug probe `xyz` equals 456  
`if (xyz >= 23'u456) then`
- A 23-bit debug probe `klm` does not equal hex A5  
`if (klm != 23'h0000A5) then`

Examples of multiple debug probe condition statements are:

- Two debug probe comparisons combined with an "OR" function:  
`if ((xyz >= 23'u456) || (abc == 1'b0)) then`
- Two debug probe comparisons combined with an "AND" function:  
`if ((xyz >= 23'u456) && (abc == 1'b0)) then`

- Three debug probe comparisons combined with an "OR" function:

```
if ((xyz >= 23'u456) || (abc == 1'b0) || (klm != 23'h0000A5)) then
```

- Three debug probe comparisons combined with an "AND" function:

```
if ((xyz >= 23'u456) && (abc == 1'b0) && (klm != 23'h0000A5)) then
```

## Counter Conditions

Counter conditions can be used in two-way and three-way branching conditional statements. Each counter condition consumes one counter comparator.




---

**IMPORTANT:** *Each counter has only one counter comparator. This means that you can only use a particular counter in a counter condition once in the entire trigger state machine program.*

---

The probe port conditions consist of a comparison operator and a value. The valid probe condition comparison operators are:

- `==` (equals)
- `!=` (not equals)




---

**IMPORTANT:** *Each counter is always 16 bits wide.*

---

Examples of valid counter condition values are:

- 16-bit binary value of 0

```
16'b0000_0000_0000_0000
16'b0000000000000000
```

- 16-bit hex value of 7A

```
16'h007A
```

- 16-bit integer value of 123

```
16'u123
```

Examples of counter condition statements:

- Counter `$counter0` equals binary 0

```
($counter0 == 16'b0000000000000000)
```

- Counter `$counter2` does not equal decimal 23

```
($counter2 != 16'u23)
```

## Combined Debug Probe and Counter Conditions

Debug probe conditions and counter conditions can be combined together to form a single condition using the following rules:

- All debug probe comparisons must be combined together using the same "||" (OR) or "&&" (AND) operators.
- The combined debug probe condition can be combined with the counter condition using either the "||" (OR) or "&&" (AND) operators, regardless of the operator used to combine the debug probe comparisons together.

Examples of multiple debug probe and counter condition statements are:

- Two debug probe comparisons combined with an "OR" function, then combined with counter conditional using "AND" function:

```
if ((xyz >= 23'u456) || (abc == 1'b0)) && ($counter0 == 16'u0023)) then
```

- Two debug probe comparisons combined with an "AND" function, then combined with counter conditional using "OR" function:

```
if ((xyz >= 23'u456) && (abc == 1'b0)) || ($counter0 == 16'u0023)) then
```

- Three debug probe comparisons combined with an "OR" function, then combined with counter conditional using "AND" function:

```
if ((xyz >= 23'u456) || (abc == 1'b0) || (klm != 23'h0000A5)) && ($counter0 == 16'u0023)) then
```

- Three debug probe comparisons combined with an "AND" function, then combined with counter conditional using "OR" function:

```
if ((xyz >= 23'u456) && (abc == 1'b0) && (klm != 23'h0000A5)) || ($counter0 == 16'u0023)) then
```

## Trigger State Machine Language Grammar

NOTES:

- The language is case insensitive
- Comment character is hash '#' character. Anything including and after a # character is ignored.
- 'THING' = THING is a terminal
- {<thing>} = 0 or more thing
- [<thing>] = 0 or 1 thing

```

<program> ::= <state_list>

<state_list> ::= <state_list> <state> | <state>

<state> ::= 'STATE' <state_label> ':' <if_condition> | <action_block>

<action_block> ::= <action_list> 'GOTO' <state_label> ';'
| <action_list> 'TRIGGER' ';'
| 'GOTO' <state_label> ';'
| 'TRIGGER' ';'

<action_list> ::= <action_statement> | <action_list> <action_statement>

<action_statement> ::= 'SET_FLAG' <flag_name> ';'
| 'CLEAR_FLAG' <flag_name> ';'
| 'INCREMENT_COUNTER' <counter_name> ';'
| 'RESET_COUNTER' <counter_name> ';'

<if_condition> ::= 'IF' '(' <condition> ')' 'THEN' <actionblock>
                ['ELSEIF' '(' <condition> ')' 'THEN' <actionblock>]
                'ELSE' <actionblock>
                'ENDIF'

<condition> ::= <probe_match_list>
| <counter_match>
| <probe_counter_match>

<probe_counter_match> ::= '(' <probe_counter_match> ')'
| <probe_match_list> <boolean_logic_op> <counter_match>
| <counter_match> <boolean_logic_op> <probe_match_list>

<probe_match_list> ::= '(' <probe_match> ')'
| <probe_match>

<probe_match> ::= <probe_match_list> <boolean_logic_op> <probe_match_list>
| <probe_name> <compare_op> <constant>
| <constant> <compare_op> <probe_name>

<counter_match> ::= '(' <counter_match> ')'
| <counter_name> <compare_op> <constant>
| <constant> <compare_op> <counter_name>

<constant> ::= <integer_constant>
| <hex_constant>
| <binary_constant>

<compare_op> ::= '==' | '!=' | '>' | '>=' | '<' | '<='

<boolean_logic_op> ::= '&&' | '||'

```

```

--- The following are in regular expression format to simplify expressions:
--- [A-Z0-9] means match any single character in AB...Z,0..9
--- [AB]+ means match [AB] one or more times like A, AB, ABAB, AAA, etc
--- [AB]* means match [AB] zero or more times
<probe_name>      ::= [A-Z_\[\]<>/] [A-Z_0-9\[\]<>/]+
<state_label>     ::= [A-Z_] [A-Z_0-9]+
<flag_name>       ::= \$FLAG[0-3]
<counter_name>    ::= \$COUNTER[0-3]
<hex_constant>    ::= <integer>*'h<hex_digit>+
<binary_constant> ::= <integer>*'b<binary_digit>+
<integer_constant> ::= <integer>*'u<integer_digit>+
<integer>         ::= <digit>+
<hex_digit>       ::= [0-9ABCDEFBN_]
<binary_digit>    ::= [01XRFBN_]
<digit>           ::= [0-9]

```



# Additional Resources

---

## Xilinx Resources

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

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

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

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

---

## Solution Centers

See the [Xilinx Solution Centers](#) for support on devices, software tools, and intellectual property at all stages of the design cycle. Topics include design assistance, advisories, and troubleshooting tips.

---

## References

These documents provide supplemental material useful with this guide:

Vivado® Design Suite 2013.3 Documentation

(<http://www.xilinx.com/cgi-bin/docs/rdoc?v=2013.3;t=vivado+docs>)

1. *Vivado Design Suite User Guide: Logic Simulation* ([UG937](#))
2. *Vivado Design Suite User Guide: Synthesis* ([UG901](#))
3. *Vivado Design Suite User Guide: Implementation* ([UG904](#))
4. *Vivado Design Suite: Release Notes, Installation and Licensing* ([UG973](#))
5. *Vivado Design Suite User Guide: Design Flows Overview* ([UG892](#))

## ISE Design Suite 14.7 Documentation

6. iMPACT Help  
([http://www.xilinx.com/cgi-bin/docs/rdoc?l=en;v=14.7;d=isehelp\\_start.htm;a=pim\\_c\\_overview.htm](http://www.xilinx.com/cgi-bin/docs/rdoc?l=en;v=14.7;d=isehelp_start.htm;a=pim_c_overview.htm))

## Xilinx IP Documentation

7. *LogiCORE IP ChipScope Pro Integrated Logic Analyzer (ILA) (v2) Datasheet* ([DS875](#))
8. *LogiCORE IP Virtual Input/Output (VIO) v2.0 Product Guide* ([PG159](#))
9. *LogiCORE IP IBERT for 7 Series GTX Transceivers* ([PG132](#))
10. *LogiCORE IP IBERT for 7 Series GTP Transceivers* ([PG133](#))
11. *LogiCORE IP IBERT for 7 Series GTH Transceivers* ([PG152](#))
12. *LogiCORE IP JTAG to AXI Master v1.0 Product Guide* ([PG174](#))