

Vivado Design Suite User Guide

Programming and Debugging

UG908 (v2014.4) November 19, 2014

Revision History

The following table shows the revision history for this document.

Date	Version	Revision
11/19/2014	2014.4	<ul style="list-style-type: none"> • Added new sections for Using Advanced Encryption Standard (AES-256) Keys with the Battery-Backed sRAM (BBR) Register.
10/01/2014	2014.3	<ul style="list-style-type: none"> • Added new sections for the following: <ul style="list-style-type: none"> ◦ Readback and Verify ◦ eFUSE Operations ◦ System Monitor ◦ Renaming Debug Probes ◦ Partial Buffer Capture ◦ Trigger at Startup • Added new table for Setting the Number of Capture Windows • Added new table for Description of hw_sysmon Tcl Commands • Added new properties for UltraScale bitstream settings • Updated section on Auto Re-Trigger
05/30/2014	2014.1	Fixed linking targets. No content changes.
04/23/2014	2014.1	<p>New sections have been added for the following.</p> <ul style="list-style-type: none"> • Changing Device Configuration Bitstream Settings • Programming the FPGA Device • Viewing ILA Probe Data in the Waveform Viewer • Configuration Memory Support <p>The following sections have been updated.</p> <ul style="list-style-type: none"> • Enabling Trigger In and Out Ports • Programming Configuration Memory Devices • Setting the Number of Capture Windows • Using Auto Re-Trigger • Connecting to a Remote hw_server Running on a Lab Machine

Table of Contents

Revision History	2
Chapter 1: Introduction	
Getting Started	6
Debug Terminology	7
Chapter 2: Programming the Device	
Introduction	9
Generating the Bitstream	9
Changing the Bitstream File Format Settings	10
Changing Device Configuration Bitstream Settings	11
Programming Configuration Memory Devices	13
Programming the FPGA Device	19
Using a Vivado Hardware Manager to Program an FPGA Device	20
Readback and Verify	27
eFUSE Operations	30
Using Advanced Encryption Standard (AES-256) Keys with the Battery-Backed sRAM (BBR) Register 30	
FUSE_DNA: Unique Device DNA	33
System Monitor	39
Chapter 3: Debugging the Design	
Introduction	41
RTL-level Design Simulation	41
Post-Implemented Design Simulation	42
In-System Logic Design Debugging	42
In-System Serial I/O Design Debugging	42
Chapter 4: In-System Logic Design Debugging Flows	
Introduction	43
Probing the Design for In-System Debugging	43
Using the Netlist Insertion Debug Probing Flow	44
HDL Instantiation Debug Probing Flow Overview	56

Using the HDL Instantiation Debug Probing Flow	57
Implementing the Design Containing the Debug Cores	61

Chapter 5: Debugging Logic Designs in Hardware

Introduction	62
Using Vivado® Logic Analyzer to Debug the Design.	62
Connecting to the Hardware Target and Programming the FPGA Device	62
Setting up the ILA Core to Take a Measurement.	63
Writing ILA Probes Information	83
Reading ILA Probes Information	84
Viewing Captured Data from the ILA Core in the Waveform Viewer.	84
Saving and Restoring Captured Data from the ILA Core	84
Setting Up the VIO Core to Take a Measurement	85
Viewing the VIO Core Status	87
Interacting with VIO Core Output Probes	92
Hardware System Communication Using the JTAG-to-AXI Master Debug Core	94
Using Vivado Logic Analyzer in a Lab Environment.	96
Description of Hardware Manager Tcl Objects and Commands.	98
Using Tcl Commands to Interact with a JTAG-to-AXI Master Core.	102
Using Tcl Commands to Take an ILA Measurement	103
Trigger At Startup.	104

Chapter 6: Viewing ILA Probe Data in the Waveform Viewer

Introduction	106
Customizing the Configuration	110
Renaming Objects	115
Bus Radixes.	117
Viewing Analog Waveforms	118
Zoom Gestures	121

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

Introduction	122
Generating an IBERT Core using the Vivado IP Catalog.	122
Generating and Implementing the IBERT Example Design	123

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

Introduction	125
Using Vivado® Serial I/O Analyzer to Debug the Design.	125

Appendix A: Device Configuration Bitstream Settings

Appendix B: Trigger State Machine Language Description

States	148
Goto Action	148
Conditional Branching	149
Counters	149
Flags	150
Conditional Statements	150

Appendix C: Configuration Memory Support

Artix®-7 Configuration Memory Devices	156
Kintex®-7 Configuration Memory Devices	158
Virtex®-7 Configuration Memory Devices	160
Kintex® UltraScale™ Configuration Memory Devices	162
Virtex® UltraScale™ Configuration Memory Devices	164
Zynq®-7000 Configuration Memory Devices	166

Appendix D: Additional Resources and Legal Notices

Xilinx Resources	168
Solution Centers	168
References	168
Training Courses	169
Please Read: Important Legal Notices	169

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](#))

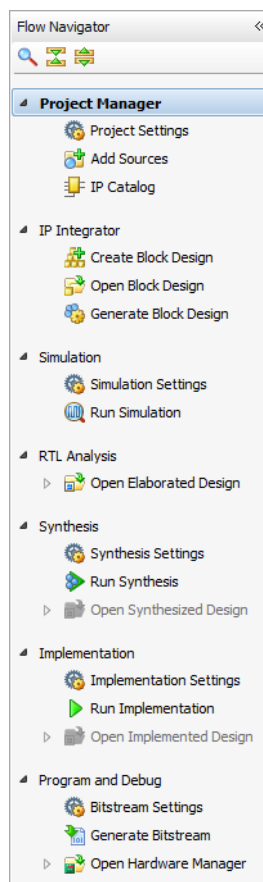


Figure 1-1: Program and Debug section of the Flow Navigator panel

Debug Terminology

ILA: The Integrated Logic Analyzer (ILA) feature allows you to perform in-system debugging of post-implemented designs on an FPGA device. This feature should be used when there is a need to monitor signals in the design. You can also use this feature to trigger on hardware events and capture data at system speeds.

The ILA core can be instantiated in your RTL code or inserted post synthesis in the Vivado design flow. [Chapter 4](#) and [Chapter 5](#) of this guide have more details on the ILA core and its usage methodology in the Vivado Design Suite. Detailed documentation on the ILA core IP can be found in the *LogiCORE IP Integrated Logic Analyzer Product Guide* (PG172) [[Ref 16](#)].

VIO: The Virtual Input/Output (VIO) debug feature can both monitor and drive internal FPGA signals in real time. In the absence of physical access to the target hardware, you can use this debug feature to drive and monitor signals that are present on the real hardware.

This debug core needs to be instantiated in the RTL code, hence you need to know up-front, what nets to drive. The IP Catalog lists this core under the Debug category. [Chapter 5](#) of this guide has more details on the VIO core and its usage methodology in the Vivado Design Suite. Detailed documentation on the ILA core IP can be found in the *LogiCORE IP Virtual Input/Output Product Guide* (PG 159)[[Ref 12](#)].

IBERT: The IBERT (Integrated Bit Error Ratio Tester) Serial Analyzer design enables in-system serial I/O validation and debug. This allows you to measure and optimize your high-speed serial I/O links in your FPGA-based system. Xilinx recommends using the IBERT Serial Analyzer design when you are interested in addressing a range of in-system debug and validation problems from simple clocking and connectivity issues to complex margin analysis and channel optimization issues.

Xilinx recommends using the IBERT Serial Analyzer design when you are interested in measuring the quality of a signal after a 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. Users can access this design by selecting, configuring, and generating the IBERT core from the IP Catalog and selecting the Open Example Design feature of this core. [Chapter 7](#) and [Chapter 8](#) of this guide have more details on the IBERT core and its usage methodology in the Vivado Design Suite. Detailed documentation on the IBERT design can be found in the *LogiCORE IP IBERT for 7 Series GTX Transceivers* (PG132) [[Ref 13](#)], *LogiCORE IP IBERT for 7 Series GTP Transceivers* (PG133) [[Ref 14](#)], and *LogiCORE IP IBERT for 7 Series GTH Transceivers* (PG152) [[Ref 15](#)].

JTAG-to-AXI Master: JTAG-to-AXI Master debug feature is used to generate AXI transactions that interact with various AXI full and AXI lite slave cores in a system that is running in hardware. Xilinx recommends that you use this core to generate AXI transactions and debug/drive AXI signals internal to an FPGA at run time. This core can be used in designs without processors as well.

The IP Catalog lists this core under the Debug category. [Chapter 5](#) of this guide has more details on the JTAG-to-AXI Master core and its usage methodology in the Vivado Design Suite. Detailed documentation on the JTAG-to-AXI IP core can be found in the *LogiCORE IP JTAG to AXI Master v1.0 Product Guide* (PG174) [[Ref 17](#)].

Programming the Device

Introduction

The basic hardware programming phase has two steps for FPGA devices:

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.

For more information on how to perform advanced hardware programming, refer to [Programming Configuration Memory Devices, page 13](#).

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 Vivado 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 Vivado flow navigator.

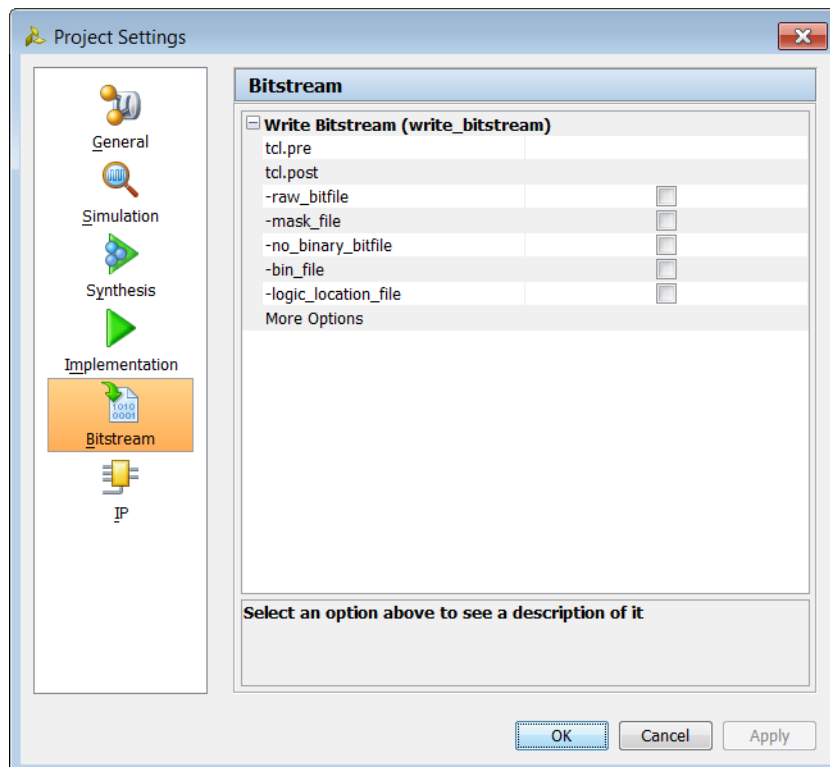


Figure 2-1: Bitstream settings panel

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) This switch 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 **Edit Device Properties** dialog for the selected synthesized or implemented design netlist. The following steps describe how to set various bitstream properties using this method:

1. Select **Tools > Edit Device Properties**.
2. In the **Edit Device Properties** dialog, select one of the categories in the left-hand column (see [Figure 2-2](#)).



TIP: You can type a property in the Search field. For example, type `jtag` into the Search text field to find and select properties related to JTAG programming.

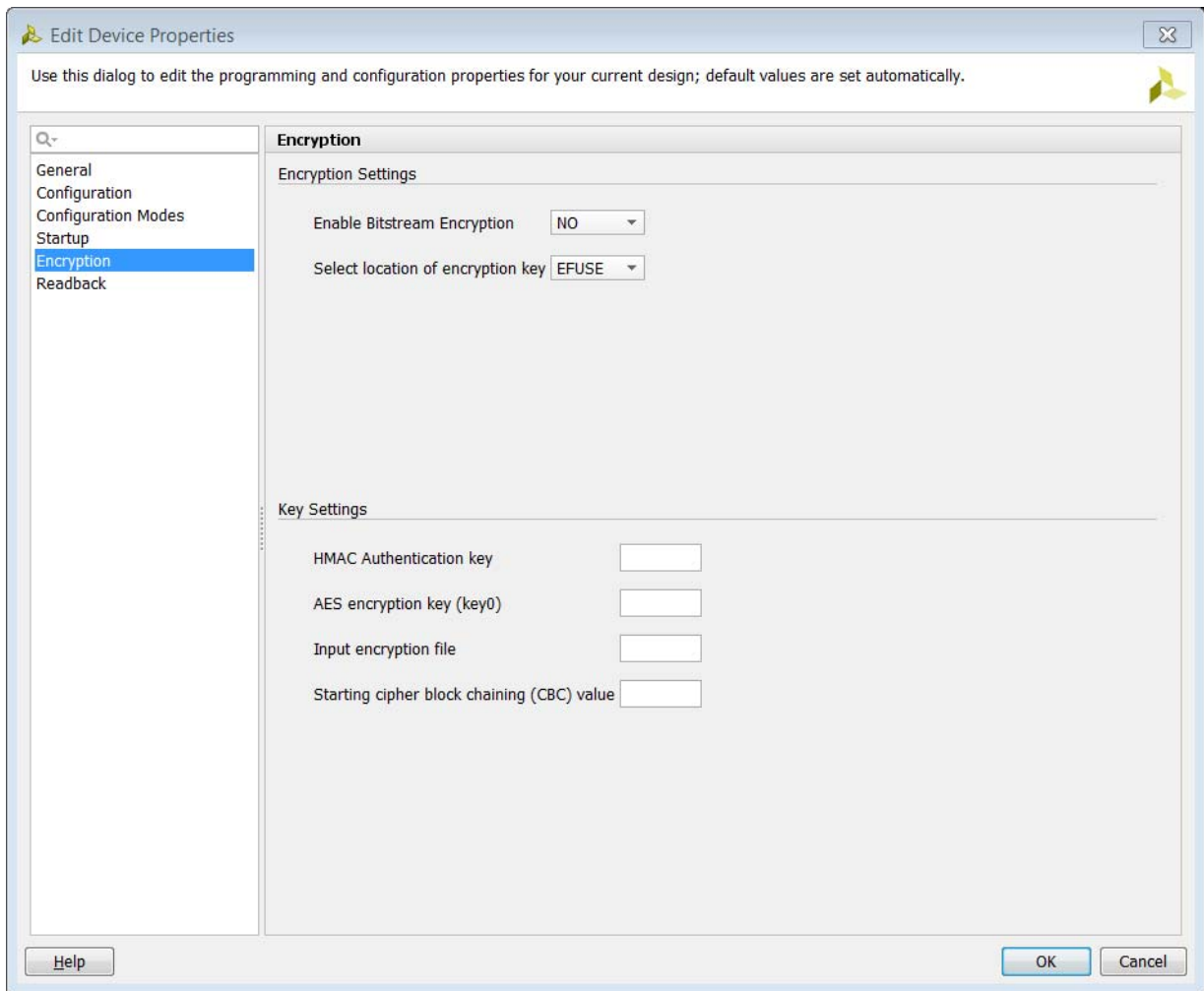


Figure 2-2: Edit Device Properties

3. Set the properties to the desired values, and click **OK**.
4. Select **File > Save Constraints** to save the updated properties to the target XDC file.

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 Configuration Memory Devices

The Vivado device programmer feature enables you to directly program Xilinx® FPGA devices via JTAG. Vivado can also indirectly program Flash-based configuration memory devices via JTAG. Do this by first programming the Xilinx FPGA device with a special configuration that provides a data path between JTAG and the Flash device interface followed by programming the configuration memory device contents using this data path.

The Vivado device configuration feature enables you to directly configure Xilinx® FPGAs or Memory Devices using either Xilinx or Digilent cables. See [Connecting to a Hardware Target Using hw_server, page 21](#) for a list of appropriate cables. Operating in Boundary-Scan mode, Vivado can configure or program Xilinx FPGAs, and Configuration Memory Devices.

Refer to [Appendix C, Configuration Memory Support](#) for a complete list of configuration memory devices supported by Vivado.

To program and boot from a Configuration Memory Device in Vivado follow the steps below.

1. Generate bitstreams for use with configuration memory devices.
2. Create a Configuration Memory File (.mcs).
3. Connect to the Hardware target in Vivado.
4. Add the configuration memory device.
5. Program the configuration memory device using the Vivado IDE.
6. Boot the FPGA device (optional).

Generate Bitstreams for use with Configuration Memory Devices

On the synthesized or implemented design select **Tools->Edit Device Properties** to open the **Edit Device Properties** dialog as shown below.

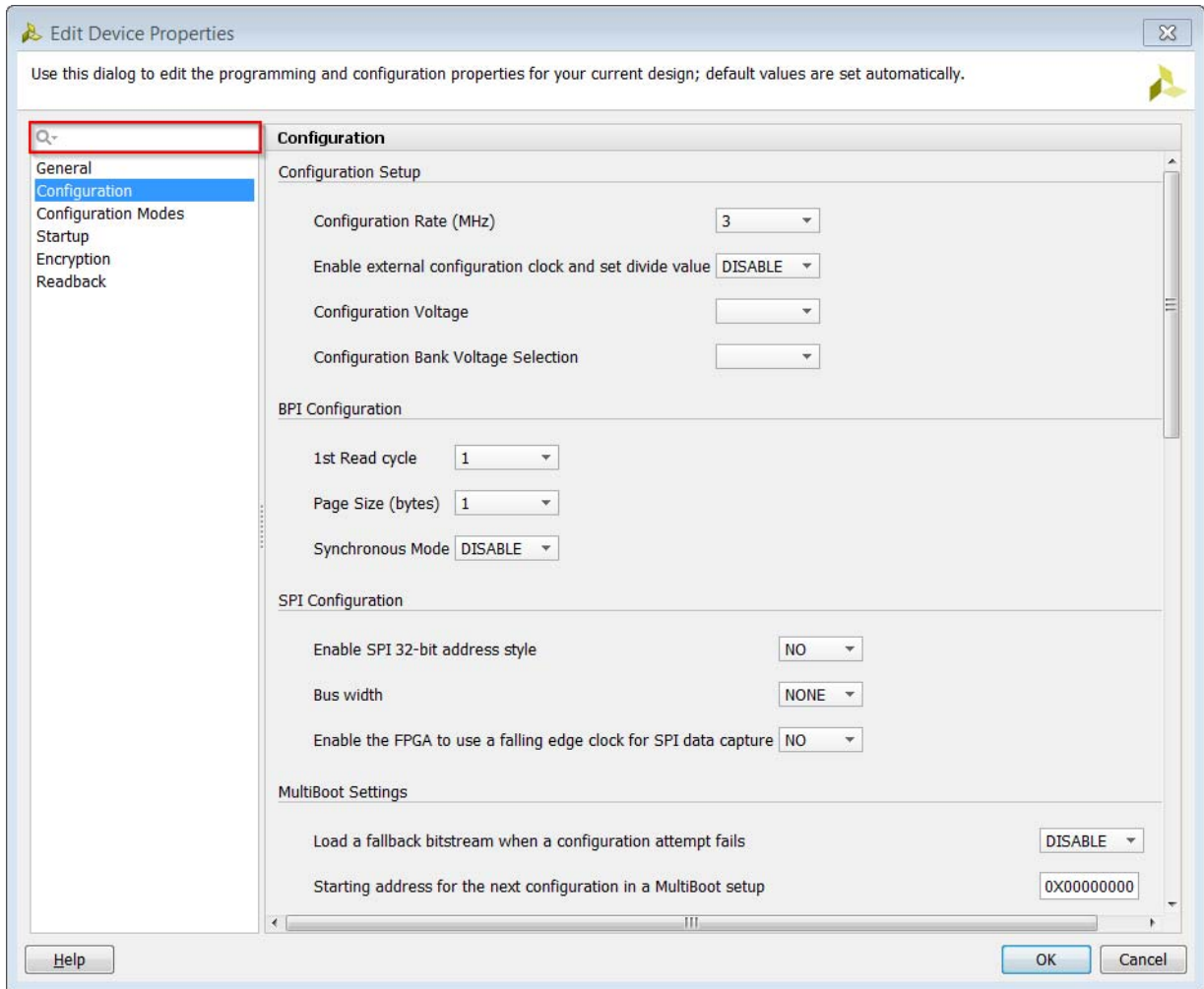


Figure 2-3: Edit Device Properties: Search Field

Use the search field in the upper left of the dialog box to search for all SPI or BPI related fields and select the appropriate option settings. See [Appendix A, Device Configuration Bitstream Settings](#) for the device configuration settings.

Creating a Configuration Memory File

Use the `write_cfgmem` Tcl command to create the `.mcs` programming file. This file will be used in programming the configuration memory device.

For example, to generate an .mcs file for a single 1Gbit BPI configuration memory device:

```
write_cfgmem -format mcs -interface bpix16 -size 128 /
             -loadbit "up 0x0 design.bit"-file design.mcs
```

Note: The `-size` argument to `write_cfgmem` is in Mbytes, different from flash device capacity which is based on Mbits. Hence, a 1Gbit sized flash device is provided as 128 Mbytes to `write_cfgmem` in the example above. Note that `write_cfgmem` automatically sizes the configuration memory file to the size of the bitstream.

Vivado IDE supports the ability to chain multiple .bit files together using the `write_cfgmem` command. To generate an .mcs file for a single 1Gbit BPI configuration memory device containing multiple bitstreams:

```
write_cfgmem -format mcs -interface bpix16 -size 128 /
             -loadbit "up 0 design1.bit up 0xFFFFF design2.bit" /
             -file design1_design2.mcs
```

For more information on `write_cfgmem` command refer to the *Vivado Design Suite Tcl Command Reference* (UG835) [Ref 7].

Connect to the Hardware Target in Vivado

To connect to a hardware target in Vivado, do the following:

1. Ensure the appropriate configuration mode (Master SPI or Master BPI) is selected on the FPGA mode pins of the hardware target to configure the FPGA from a configuration memory device.

For more information, see the appropriate Configuration User Guide for the device you are targeting.

2. Follow the steps in section [Programming the FPGA Device, page 19](#) to connect to the hardware target.

Adding a Configuration Memory Device

To add the configuration memory device to a hardware target in Vivado device programmer, do the following:

1. After connecting to the hardware target as outlined above, add the configuration memory device by right-clicking the hardware target as shown below and selecting **Add Configuration Memory Device**.

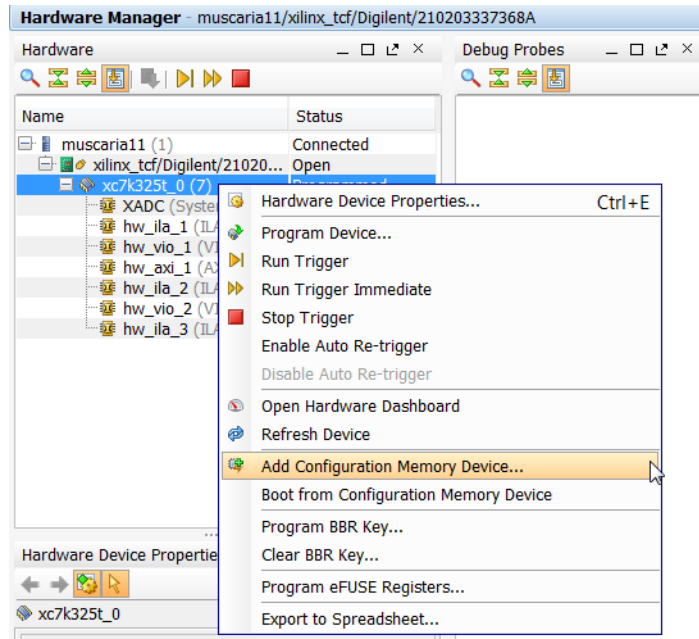


Figure 2-4: Add Configuration Memory Device Menu Item

On clicking on this menu item the **Add Configuration Memory Device** dialog box opens.

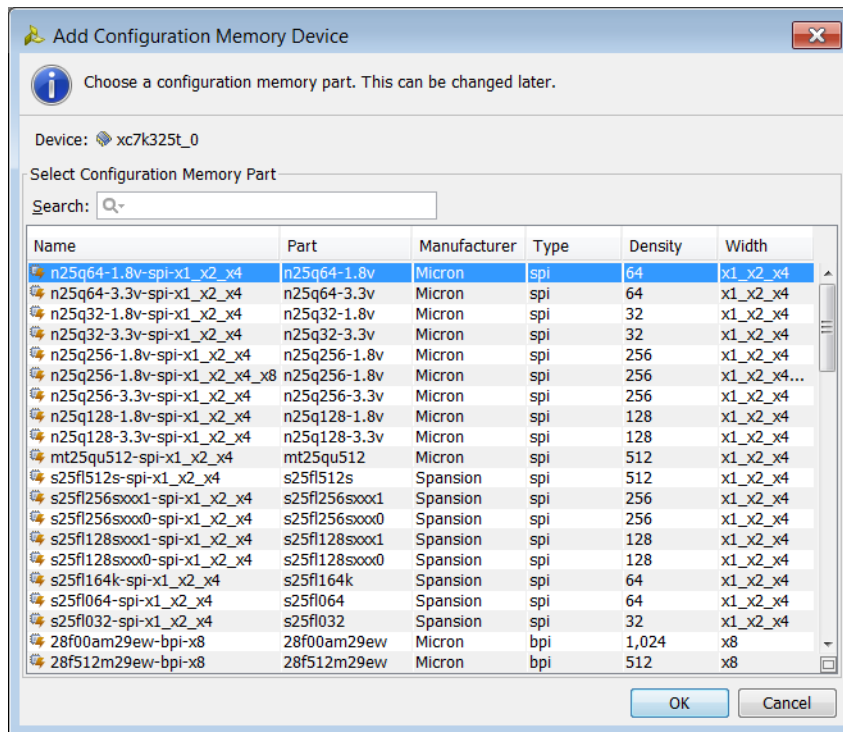


Figure 2-5: Add Configuration Memory Device part selector

2. Select the appropriate configuration memory part and click **OK**.



TIP: Use the **Search** field to pare down the list using Vendor, Density, or Type information.

The configuration memory device is now added to the hardware target device.

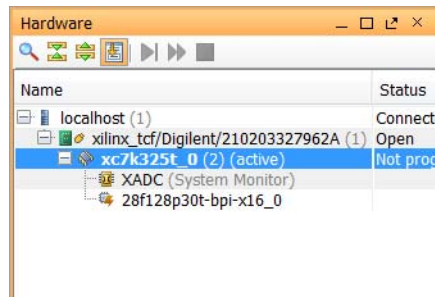


Figure 2-6: Configuration Memory Device Added to Hardware Target

Programming a Configuration Memory Device

1. After creating the configuration memory device, Vivado device programmer prompts "Do you want to program the configuration memory device now?" as shown below.

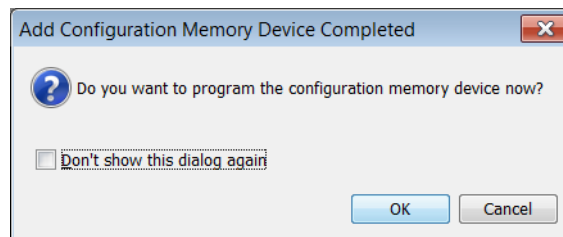


Figure 2-7: Prompt to Program Configuration Memory Device

Click **OK** to open the **Program Configuration Memory Device** dialog box.

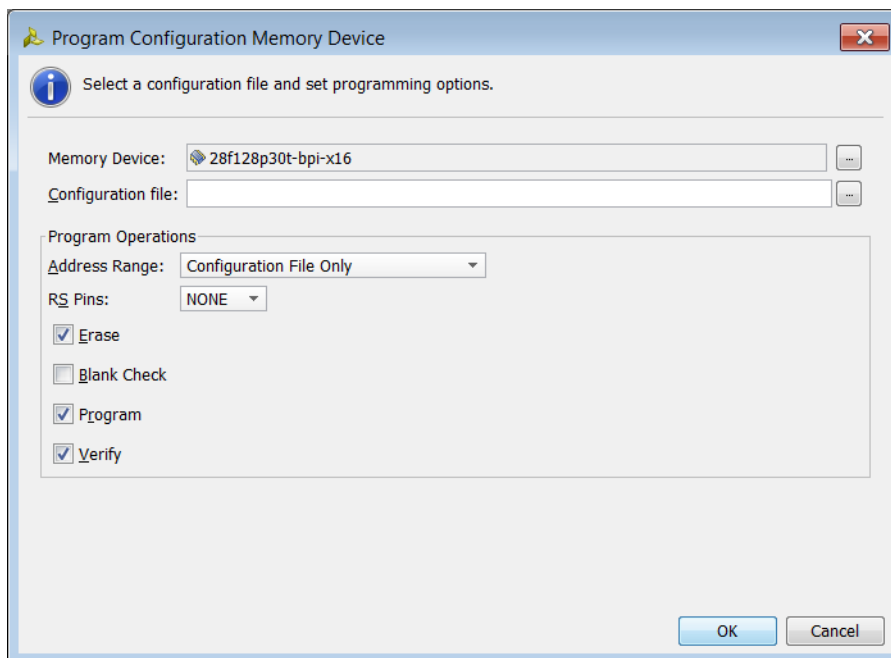


Figure 2-8: Program Configuration Memory Device Dialog

2. Set all the fields in this dialog box appropriately.
 - **Configuration file** (.mcs) - Specifies the file to use for programming the configuration memory device. The memory configuration file is created with the `write_cfgmem` Tcl command. See [Creating a Configuration Memory File, page 14](#) for more information.
 - **Program Operations** (performed on the configuration memory device):
 - **Address Range** - Specifies the address range of the configuration memory device to program. The address range values can be:
 - **Configuration File Only** - Use only the address space required by the memory configuration file to erase, blank check, program, and verify.
 - **Entire Configuration Memory Device** - Erase, blank check, program, and verify will be performed on the entire device.
 - **RS Pins** - Optional. Revision Select Pin Mapping that is used with BPI configuration memory devices only (where the upper two FPGA address pins on the flash are tied to the FPGA RS[1:0]). When the option is enabled, Vivado drives the FPGA RS[1:0] for programming. Refer to the appropriate FPGA Configuration User Guide on application usage.
 - **Erase** - Erases the contents of the configuration memory device.
 - **Blank Check** - Checks the configuration memory device to make sure the device is void of data prior to programming.

- **Program** - Program the configuration memory device with the specified Configuration File (.mcs).
 - **Verify** - Verify that the configuration memory device contents match the Configuration File (.mcs) after programming.
3. Click **OK** to start the Erase, Blank Check, Program, and Verify operations on the configuration memory device per the selections in this dialog box. Vivado notifies you as each operation finishes..

Booting the Device

After programming the configuration memory device, you can issue a soft boot operation (ie. JPROGRAM) to initiate the FPGA configuration from the attached configuration memory device. If you want to perform a Boot operation on the target FPGA device select the target device and right-click and select **Boot Device**.

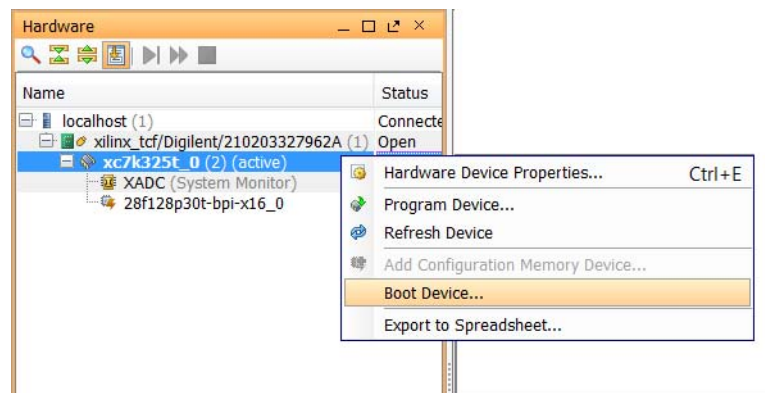


Figure 2-9: Boot FPGA Device

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 Vivado flow navigator.
- Select **Flow > Open Hardware Manager**.
- 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 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 Vivado flow navigator to open new or recent hardware targets (see Figure 2-10).

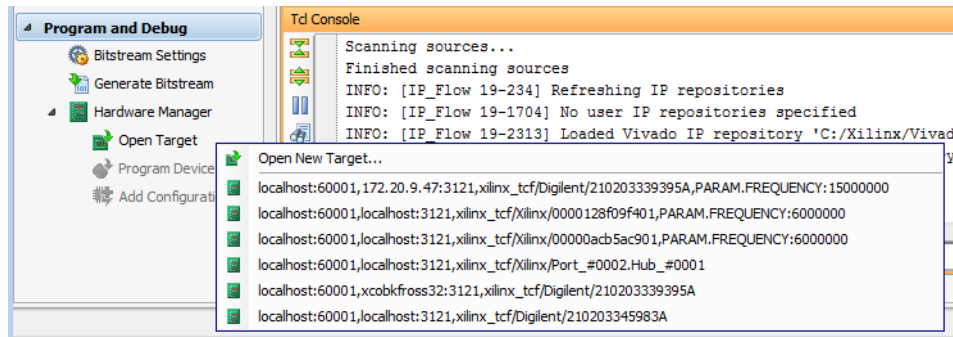


Figure 2-10: Using the Flow Navigator to Open a Hardware Target

- 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 2-11).

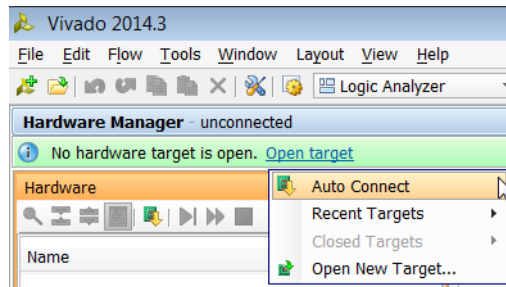


Figure 2-11: Using the User Assistance Bar to Open a Hardware Target

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



TIP: Use the **Auto Connect** selection to automatically connect to a local hardware target.

Connecting to a Hardware Target Using 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` is automatically started by Vivado when connecting to targets on the local machine. However, you can also start the `hw_server` manually on either local or remote machines. For instance, in a full Vivado installation on a Windows platform, at a `cmd` prompt run the following command:

```
C:\Xilinx\Vivado\<Vivado_version>\bin\hw_server.bat
```

If you are using a Hardware Server (Standalone) installation on a Windows platform, at a `cmd` prompt run the following command:

```
c:\Xilinx\HWSRVR\<Vivado_version>\bin\hw_server.bat
```

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 process has the following steps:

1. Select a local or remote server, depending on what machine your hardware target is connected to:
 - Local server: Use this setting if your hardware target is connected to the same machine on which you are running the Vivado IDE (See [Figure 2-12](#)). The Vivado software automatically starts the Vivado hardware server (`hw_server`) application on the local machine.
 - Remote server: Use this setting if your hardware target is connected to a different machine on which you are running the Vivado IDE. Specify the host name or IP address of the remote machine and the port number for the hardware server (`hw_server`) application that is running on that machine (see [Figure 2-13](#)). Refer to [Connecting to a Remote hw_server Running on a Lab Machine in Chapter 5](#) for more details on remote debugging.



IMPORTANT: *When using remote server, you need to manually start the Vivado hardware server (`hw_server`) application of the same or later version of Vivado software that you will use to connect to the hardware server.*



TIP: *If you only want to connect to your lab machine remotely, you do not need to install the full Vivado design suite on that remote machine. Instead, you can install the light-weight Vivado Hardware Server (Standalone) tool on the remote machine.*

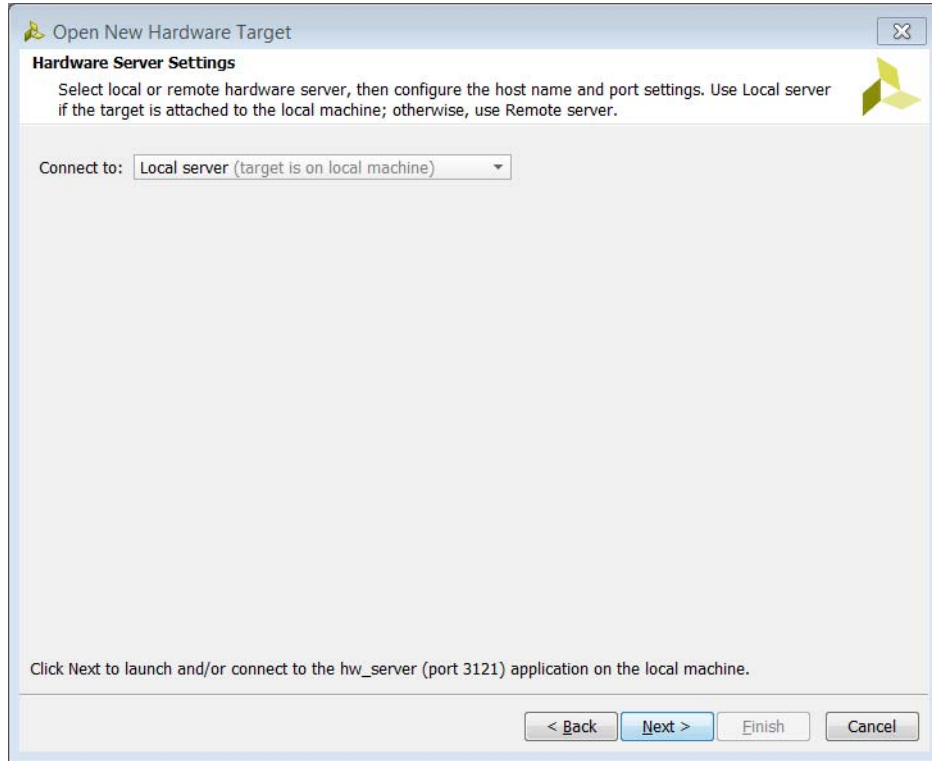


Figure 2-12: Using a Local Hardware Server

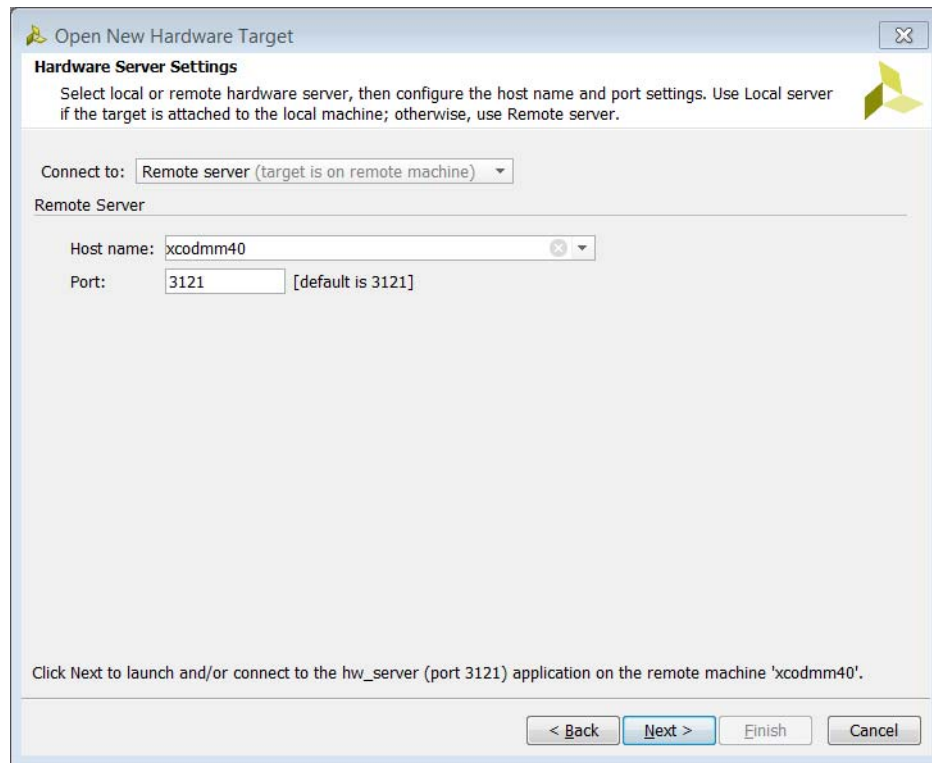


Figure 2-13: Using a Remote Hardware Server

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 2-14](#)).

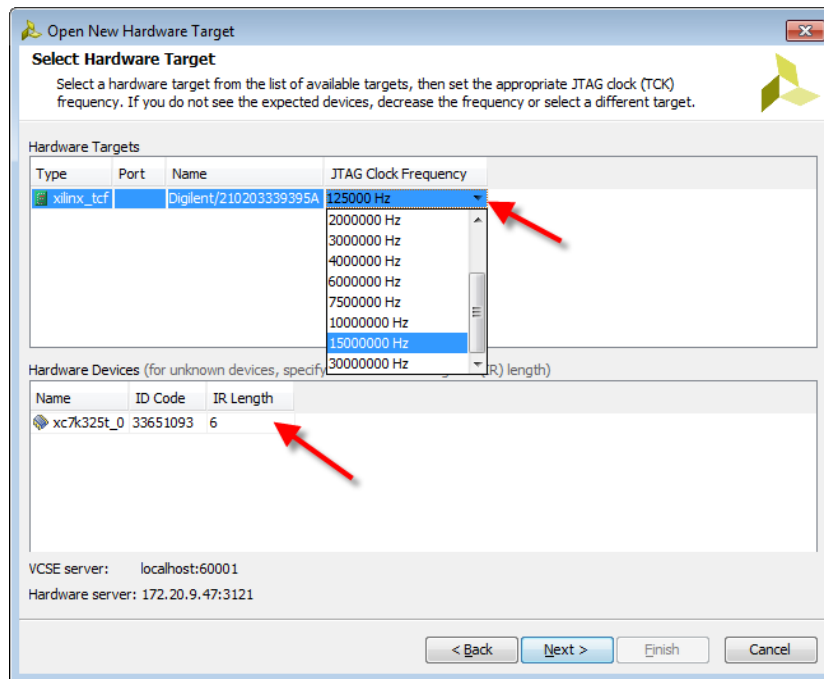


Figure 2-14: Selecting a Hardware Target

Troubleshooting a Hardware Target

You might run into issues when trying to connect to a hardware target. Here are some common issues and recommendations on how to resolve them:

- If you are not able to correctly identify the hardware devices on your target, it might mean that your hardware is not capable of running at the default target frequency. You can adjust the frequency of the TCK pin of the hardware target or cable (see [Figure 2-14](#)). 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.
- While the Vivado hardware server will attempt to automatically determine the instruction register (IR) length of all devices in the JTAG chain, in some rare circumstances it might not be able to correctly do so. You should check the IR length for each unknown device to make sure it is correct. If you need to specify the IR length, you can do so directly in the **Hardware Devices** table of the **Open New Hardware Target** wizard (see [Figure 2-14](#)).

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 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 210203339395A) that is managed by the `hw_server` running on localhost 3121, use the following Tcl commands:

```
connect_hw_server -url localhost:3121
current_hw_target [get_hw_targets */xilinx_tcf/Digilent/210203339395A]
set_property PARAM.FREQUENCY 1500000 [get_hw_targets
*/xilinx_tcf/Digilent/210203339395A]
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 2-15](#)).

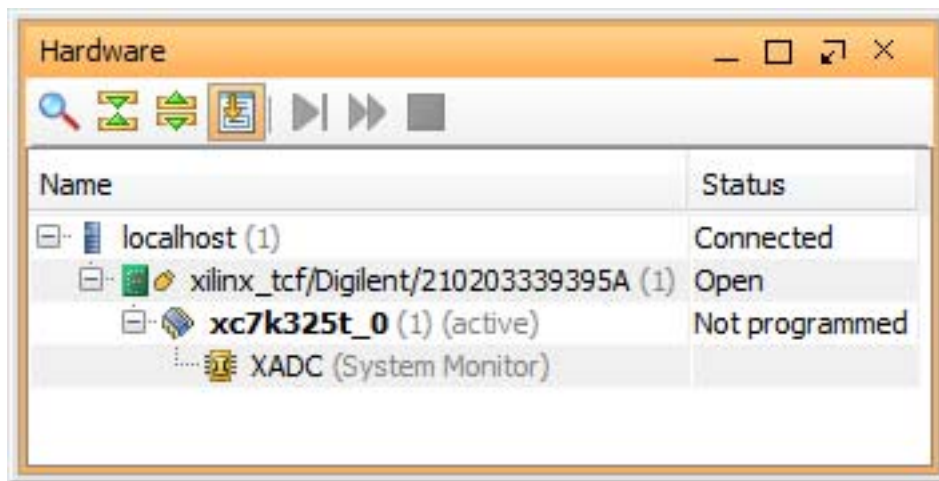


Figure 2-15: Hardware View after Opening a Connection to the Hardware Target

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 2-16](#)).

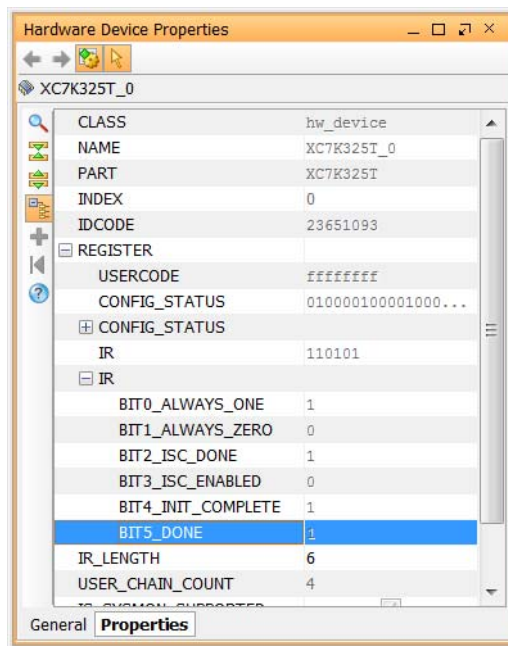


Figure 2-16: Checking the DONE status of an FPGA device

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/210203339395A}
```

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
```

Readback and Verify

Bitstream Verify and Readback

Vivado IDE can verify and/or readback the configuration data (i.e., `.bit` file) downloaded into an FPGA. When using `write_bitstream` to generate the `.bit` file, use the `-mask_file` option to create a corresponding mask (`.msk`) file. Run `write_bitstream -help` in the Vivado IDE Tcl Console for details on bitstream generation options.

When performing a verify operation, the `verify_hw_devices` Tcl command reads data back from the FPGA and uses the `.msk` file to determine which readback data bits to skip and which ones to compare against the corresponding bits in the `.bit` file.

Following is an example of a bitstream verify Tcl command sequence (the `.bit` and `.msk` files were generated by a previous call to `write_bitstream`):

```
create_hw_bitstream -hw_device [current_hw_device] \  
    -mask kcu105_cnt_ila_uncmpr.msk    kcu105_cnt_ila_uncmpr.bit  
verify_hw_devices [current_hw_device]
```

Use the `readback_hw_device` Tcl command with at least one of the following options to read back the FPGA configuration data:

- To save readback data in ASCII format:

```
-readback_file <filename.rbd>
```

- To save readback data in binary format:

```
-bin_file <filename.bin>
```

Example: Readback FPGA configuration data in both ASCII and binary formats

```
readback_hw_device [current_hw_device] \  
    -readback_file    kcu105_cnt_ila_uncmpr_rb.rbd \  
    -bin_file         kcu105_cnt_ila_uncmpr_rb.bin
```

Notes:

1. For the 2014.3 release, bitstream, verify, and readback operations are done through the Tcl Console.
2. Verify and readback operations do not work for FPGAs programmed with encrypted bitstreams. Encrypted bitstreams contain commands that disable readback. Readback is re-enabled by pulsing the FPGA PROG pin, or if the FPGA/board is powered down and powered back up again.
3. The data readback using `readback_hw_device` contains configuration data only (no configuration commands are included).

For more information on these features, see the *Ultrascale Architecture Configuration: Advance Specification User Guide* (UG570) [Ref 10] or the *7 Series FPGAs Configuration User Guide* (UG470) [Ref 8]

Configuration Memory Verify and Readback

You can convert a bitstream file (`.bit`) to an `.mcs` file and then program it into a configuration memory device, such as serial/SPI or parallel/BPI flash, via the `write_cfgmem` command. See the *Vivado Design Suite Tcl Command Reference* (UG835) [Ref 7] for details.

Verify the configuration memory device through the Vivado IDE **Hardware Manager** as shown in [Figure 2-17](#).

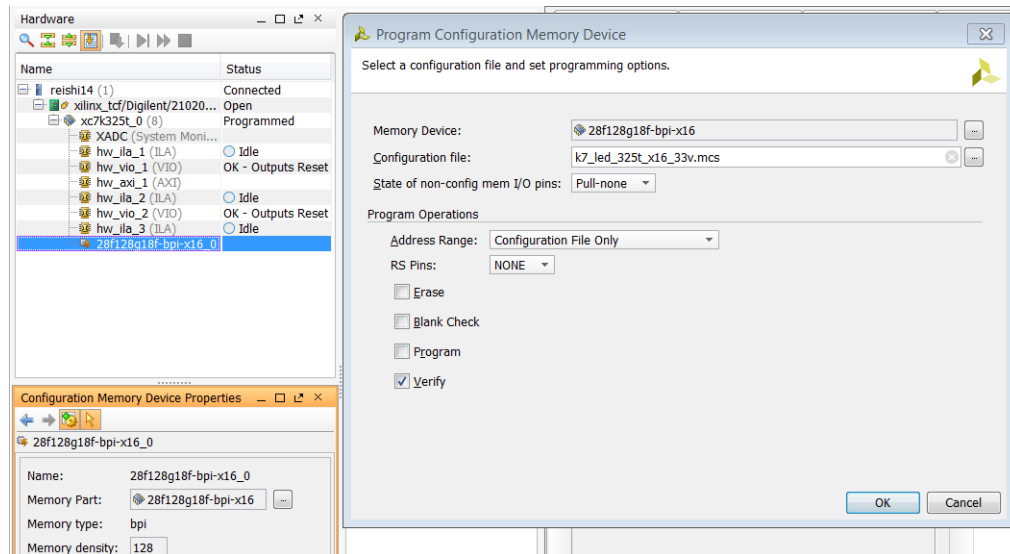


Figure 2-17: Configuration Memory Verification

You can also verify the configuration memory device by setting the appropriate `HW_CFGMEM` properties and calling `program_hw_cfgmem` as shown in the following code:

```
set_property PROGRAM.ADDRESS_RANGE {use_file} [ get_property PROGRAM.HW_CFGMEM [lindex [get_hw_devices] 0 ]]
set_property PROGRAM.FILES [list "H:/projects/k7_led/k7_led_325t_afx_x16_33v.mcs" ] [ get_property PROGRAM.HW_CFGMEM [lindex [get_hw_devices] 0]]
set_property PROGRAM.BPI_RS_PINS {none} [ get_property PROGRAM.HW_CFGMEM [lindex [get_hw_devices] 0 ]]
set_property PROGRAM.UNUSED_PIN_TERMINATION {pull-none} [ get_property PROGRAM.HW_CFGMEM [lindex [get_hw_devices] 0 ]]
set_property PROGRAM.BLANK_CHECK 0 [ get_property PROGRAM.HW_CFGMEM [lindex [get_hw_devices] 0 ]]
set_property PROGRAM.ERASE 0 [ get_property PROGRAM.HW_CFGMEM [lindex [get_hw_devices] 0 ]]
set_property PROGRAM.CFG_PROGRAM 0 [ get_property PROGRAM.HW_CFGMEM [lindex [get_hw_devices] 0 ]]
set_property PROGRAM.VERIFY 1 [ get_property PROGRAM.HW_CFGMEM [lindex [get_hw_devices] 0 ]]
startgroup
if {![string equal [get_property PROGRAM.HW_CFGMEM_TYPE [lindex [get_hw_devices] 0]] [get_property MEM_TYPE [get_property CFGMEM_PART [get_property PROGRAM.HW_CFGMEM [lindex [get_hw_devices] 0 ]]]]} { create_hw_bitstream -hw_device [lindex [get_hw_devices] 0] [get_property PROGRAM.HW_CFGMEM_BITFILE [lindex [get_hw_devices] 0]];
program_hw_devices [lindex [get_hw_devices] 0]; };
program_hw_cfgmem -hw_cfgmem [get_property PROGRAM.HW_CFGMEM [lindex [get_hw_devices] 0 ]]
endgroup
```

The contents of the configuration memory can be readback through the Vivado IDE Tcl Console using the following command sequence:

```
readback_hw_cfgmem -file test.bin -hw_cfgmem \
[get_property PROGRAM.HW_CFGMEM [lindex [get_hw_devices] 0]]
```

Notes:

1. For the 2014.3 release, the only readback format supported is binary (.bin).
2. For the 2014.3 release you can only perform configuration memory readback through the Tcl Console.

For more information on these features, see the *Ultrascale Architecture Configuration: Advance Specification User Guide* (UG570) [Ref 10] or the *7 Series FPGAs Configuration User Guide* (UG470) [Ref 8]

eFUSE Operations

7 Series® and UltraScale® devices have one-time programmable bits called eFUSE bits that perform specific functions. The different eFUSE bit types are as follows:

- FUSE_DNA - Unique device identifier bits.
- FUSE_USER - Stores a 32-bit user-defined code.
- FUSE_KEY - Stores a key for use by AES bitstream decryptor.
- FUSE_CNTL - Controls key use and read/write access to eFUSE registers.
- FUSE_SEC - Controls special device security settings in Ultrascale devices.



IMPORTANT: *Programming eFUSE register bits is a one-time only operation. Once eFUSE register bits are programmed, they cannot be reset and/or programmed again. You should take great care to double-check your settings before programming any eFUSE registers.*

Using Advanced Encryption Standard (AES-256) Keys with the Battery-Backed sRAM (BBR) Register

You can protect IP in bitstreams by encrypting the bitstreams with a 256-bit Advanced Encryption Standard (AES) key. Do this in authorized FPGAs, by loading this 256-bit key into the FPGA Battery-Backed sRAM (BBR) before downloading the encrypted bitstream.

Generating an Encrypted Bitstream

To generate an encrypted bitstream that works with the BBR, open an implemented design in Vivado IDE. Select **Bitstream Settings > Bitstream**, then click **Configure Additional Bitstream Settings**.

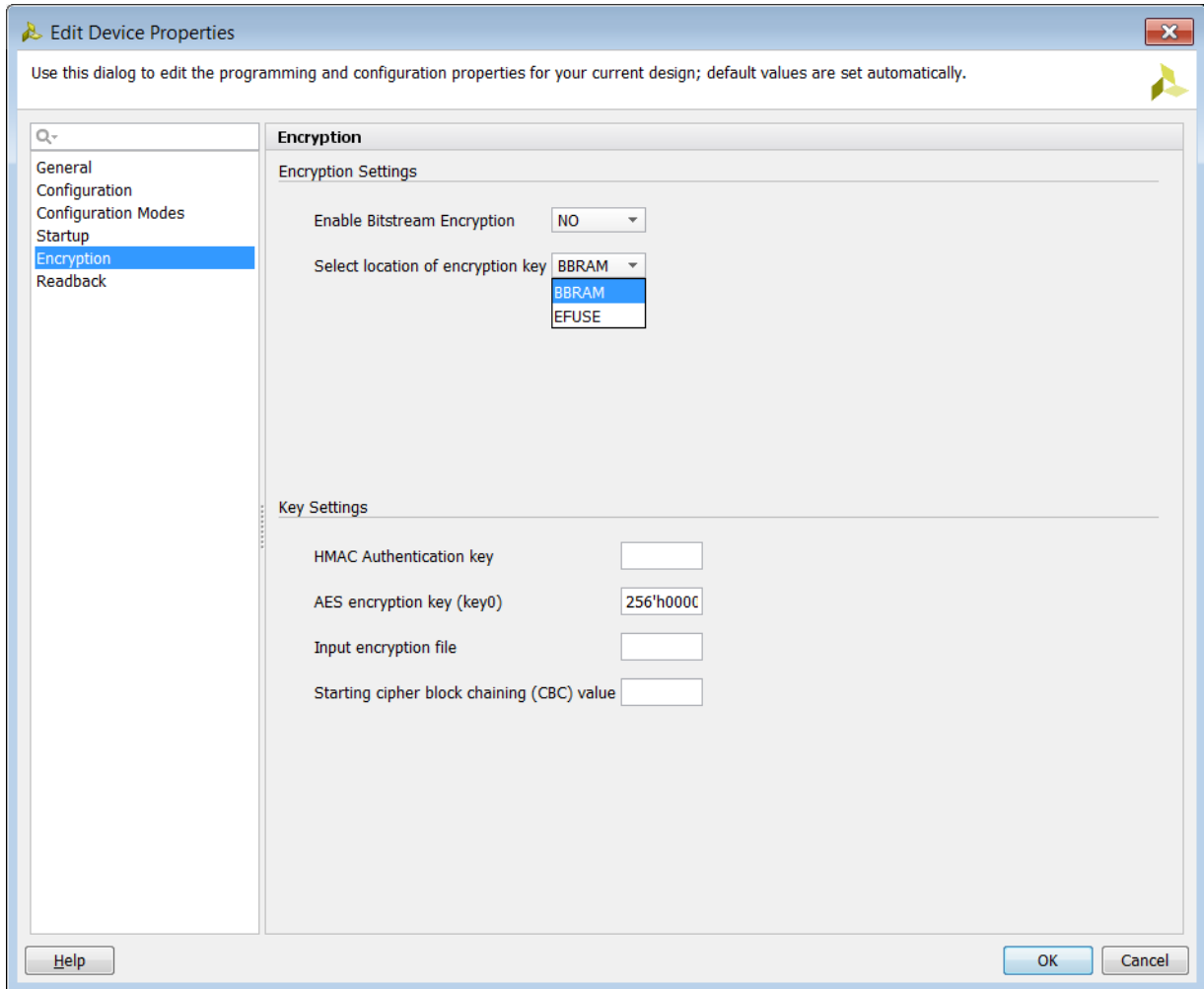


Figure 2-18: Configure Additional Bitstream Settings

In the **Edit Device Properties** dialog box, select **Encryption** in the left-hand pane, and specify the following encryption and key settings:

- **Encryption Settings**
 - Set **Enable Bitstream Encryption** to **YES**.
 - Set **Select location of encryption key** to either **BBRAM** or **EFUSE**.
 - The key location will be embedded in the encrypted bitstream.

- When the encrypted bitstream is downloaded to the device, it instructs the FPGA to use the key loaded into the BBR or the eFUSE key register to decrypt the encrypted bitstream.
- **Key Settings**
 - Specify the 256-bit **AES key to use when encrypting the bitstream**.
 - The key will be written to a file with the `.nky` file extension. Use this file when loading the key into the BBR or when programming the key into the eFUSE key register.
 - Specify **HMAC authentication key** and **Starting cipher block chaining (CBC) value**.
 - If these values are unspecified, Vivado generates a random value for you.
 - These values will be embedded in the encrypted bitstream and do not have to be programmed or loaded into the FPGA.
 - Specify **Input encryption file**.
 - Specify an existing `.nky` file to obtain the encryption key settings.

Programming the AES Key

To program the AES key into the BBR, right-click the FPGA device in the **Hardware** window, select **Program BBR Key**, and specify the AES key (`.nky`) file. When you click **OK**, the Hardware Manager programs/loads the key into the BBR. You can now program the FPGA with an encrypted bitstream that:

- was encrypted using the same AES key as was loaded into BBR.
- had **BBRAM** selected as the specified encryption key location.



IMPORTANT: For UltraScale devices, if you attempt to download an encrypted bitstream (which uses the BBR as the key source) before the key is programmed into the BBR register, the FPGA device will lock up and you will not be able to load the BBR key. You can still download **unencrypted** bitstreams, but you will not be able to download **encrypted** bitstreams because the FPGA device will prevent you from downloading a key into BBR. You must power-cycle the board to unlock the UltraScale device and then reload the BBR key.

eFUSE Bits

UltraScale FPGAs have a security feature that allows you to program two eFUSE bits that force the FPGA device to only allow encrypted bitstreams. These eFUSE bits can be set by clicking **Program eFUSE Registers** and following the prompts.

Note: The 7 Series devices only restrict the use of encrypted bitstreams with the AES key in the FUSE_KEY register.

- Set FUSE_SEC[1:0] to "11" to allow only encrypted bitstreams that use the AES key in the FUSE_KEY register.
- Set FUSE_SEC[1:0] to "10" to allow only encrypted bitstreams, but the AES key can be in the BBR or FUSE_KEY.

Clearing Keys from the BBR Register

To clear the key from the UltraScale or 7 Series FPGA BBR register:

- Right-click the FPGA device name and select **Clear BBR Key**.
- Do not connect the Vbatt pins and power-cycle the board.

Note: Pressing or pulsing the PROG pin when the board/FPGA is powered up will not clear the BBR register.

FUSE_DNA: Unique Device DNA

Each 7 Series and UltraScale device has a unique device ID called a DNA that has already been programmed into it by Xilinx. 7 Series devices have a 64-bit DNA, while UltraScale devices have a 96-bit DNA. You can read these by running the following Tcl command in the Vivado IDE Tcl Console:

```
report_property [lindex [get_hw_device] 0] REGISTER.EFUSE.FUSE_DNA
```

You can also access the device DNA by viewing the eFUSE registers in the **Hardware Device Properties** window in Vivado I DE as shown in [Figure 2-19](#):

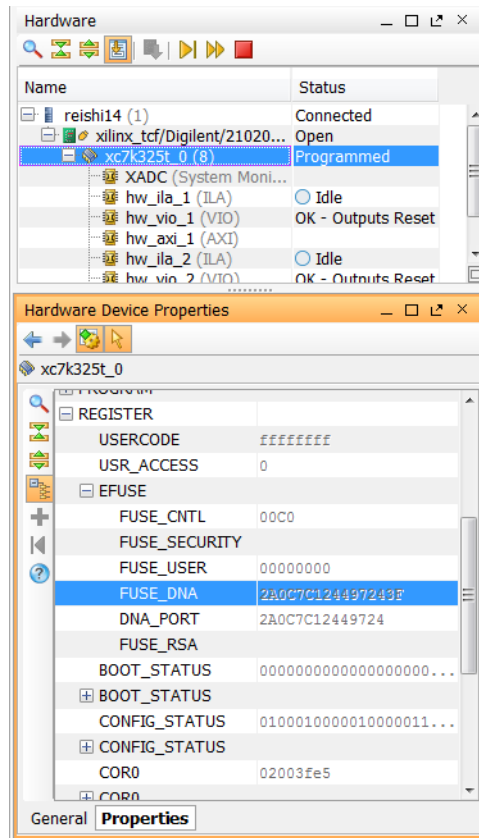


Figure 2-19: eFUSE DNA

For more information on these features, see the *Ultrascale Architecture Configuration: Advance Specification User Guide* (UG570) [Ref 10] or the *7 Series FPGAs Configuration User Guide* (UG470) [Ref 8]

FUSE_USER

The FUSE_USER eFUSES are provided to allow users to program their own special 32-bit pattern. The FUSE_USER bits are programmed using the Vivado eFUSE programming wizard. For 7 Series devices, the lower eight FUSE_USER bit are programmed at the same time as the 256-bit Advanced Encryption Engine (AES) key. The upper 24 FUSE_USER bits can be programmed then or at a later time using the same wizard.

Access the Vivado eFUSE programming wizard by right-clicking the target device in the **Hardware Manager** window, and selecting the **Program eFUSE Registers...** option. [Figure 2-20](#) shows the **Program eFUSE Registers** page of the Vivado eFUSE programming wizard.

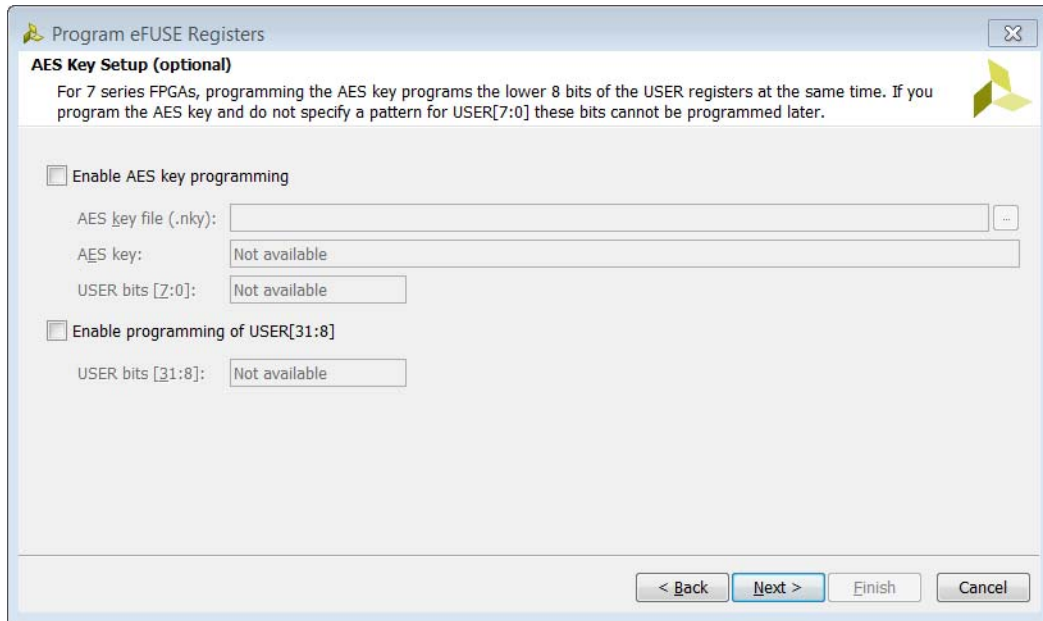


Figure 2-20: Program eFUSE Registers-AES

For UltraScale devices, all 32 bits of the 32-bit FUSE_USER are programmed at the same time, and independently of the AES key in the **Program eFUSE Registers** page as shown in Figure 2-21.

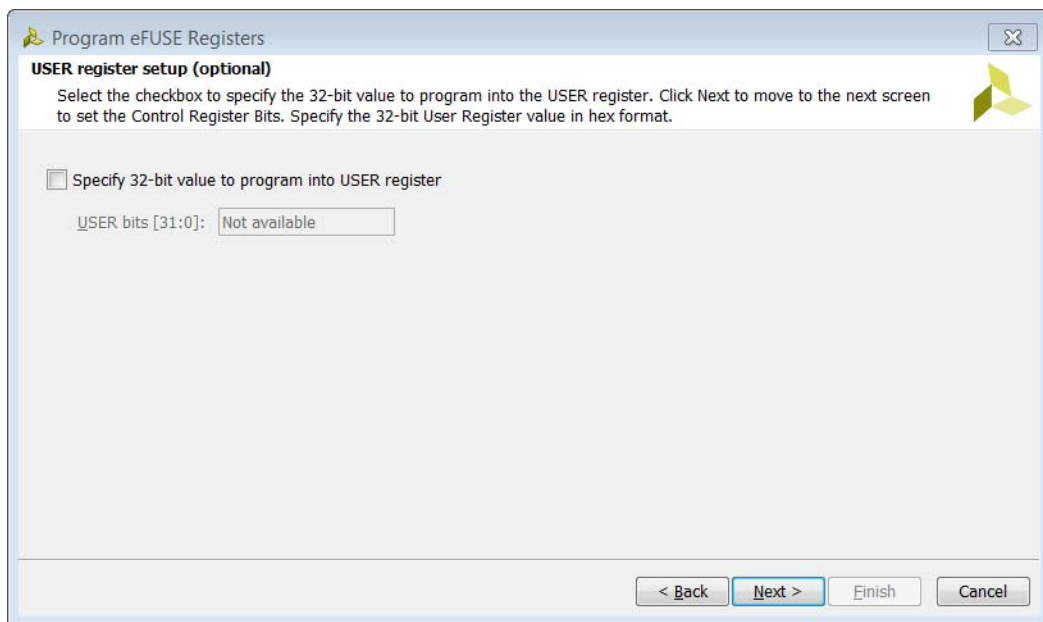


Figure 2-21: Program eFUSE Registers-ctrl-reg-ultrascale

After programming the FUSE_USER eFUSEs, the pattern can be read in several ways:

- Using the Tcl command.

```
report_property [lindex [get_hw_device] 0] REGISTER.EFUSE.FUSE_USER
```
- Through the Vivado **Hardware Device Properties** window after running a `refresh_hw_device` operation.
- Running the FUSE_USER command through the JTAG interface.

FUSE_KEY

The FUSE_KEY eFUSE bits are used to store a 256-bit AES key. The FPGA configuration logic uses this key to decrypt bitstreams that were encrypted by `write_bitstream` Tcl command using the same AES key. The encrypted bitstreams are sent to the FPGA through the JTAG interface, or through the BPI/SPI configuration interface via configuration memories.

When creating the encrypted bitstream, the correct key location must be specified (i.e., select **EFUSE**) to instruct the FPGA configuration logic to decrypt the incoming encrypted bitstream using the AES key from the FUSE_KEY eFUSEs and not from the volatile BBRAM key register. In the Vivado IDE, click **Implemented Design**, then click **Bitstream Settings** to see the following window:

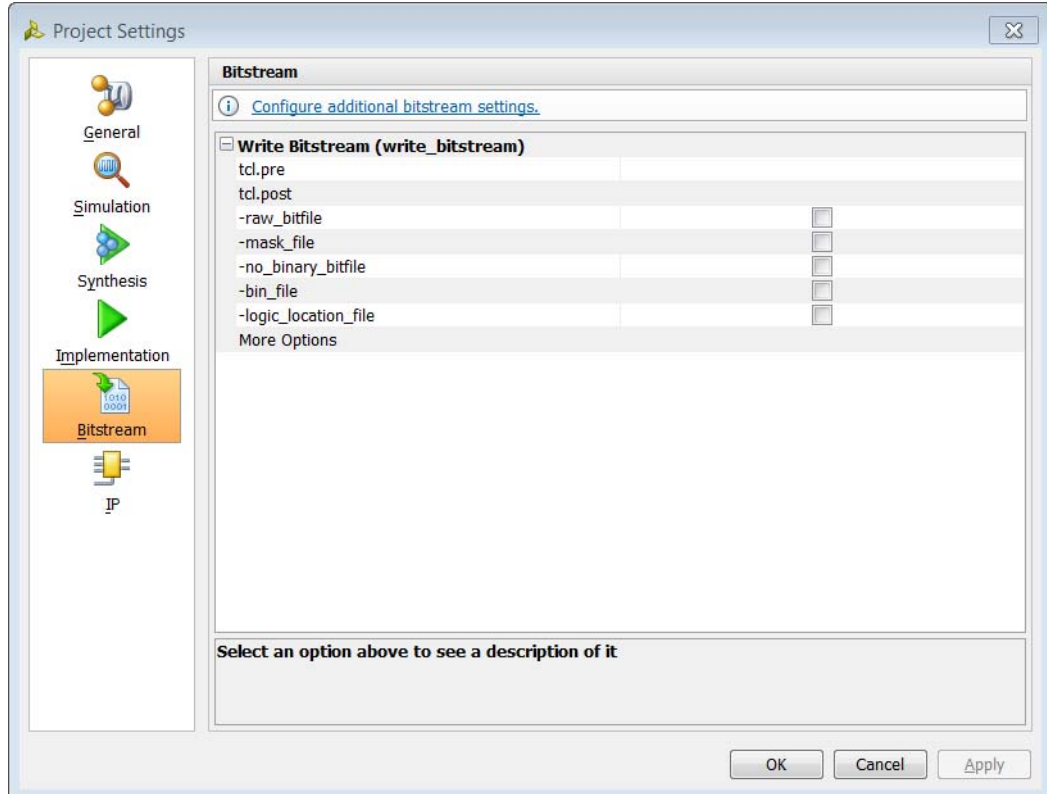


Figure 2-22: Bitstream Settings

Click **Configure additional bitstream settings** to open the **Edit Device Properties** dialog box. There you can set the encryption options as shown in [Figure 2-23](#).

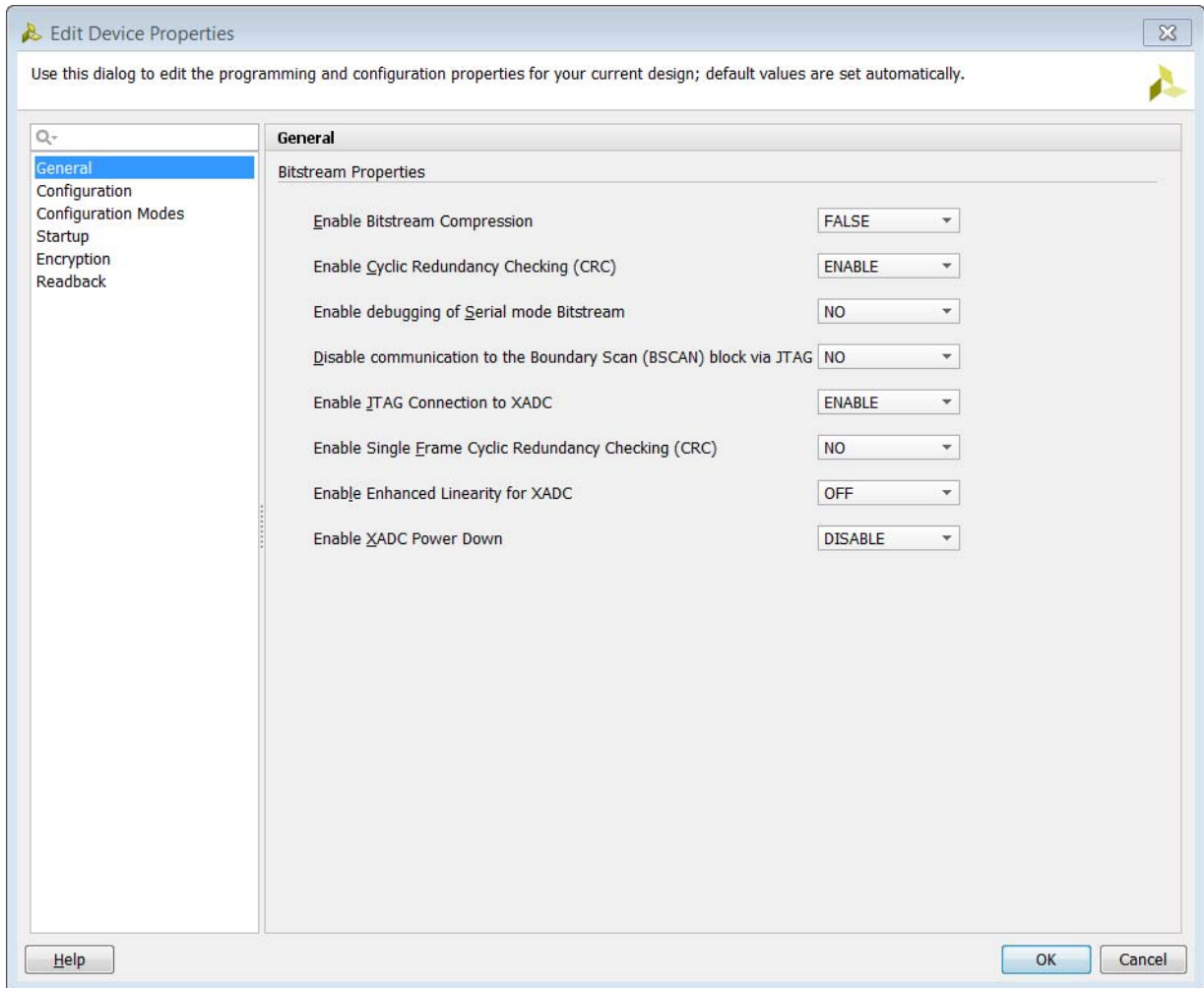


Figure 2-23: Edit Device Properties

FUSE_CNTL

The FUSE_CNTL eFUSEs control access to special device features.

For 7 Series devices use the dialog shown in [Figure 2-24](#) to specify the FUSE_CNTL eFUSEs.

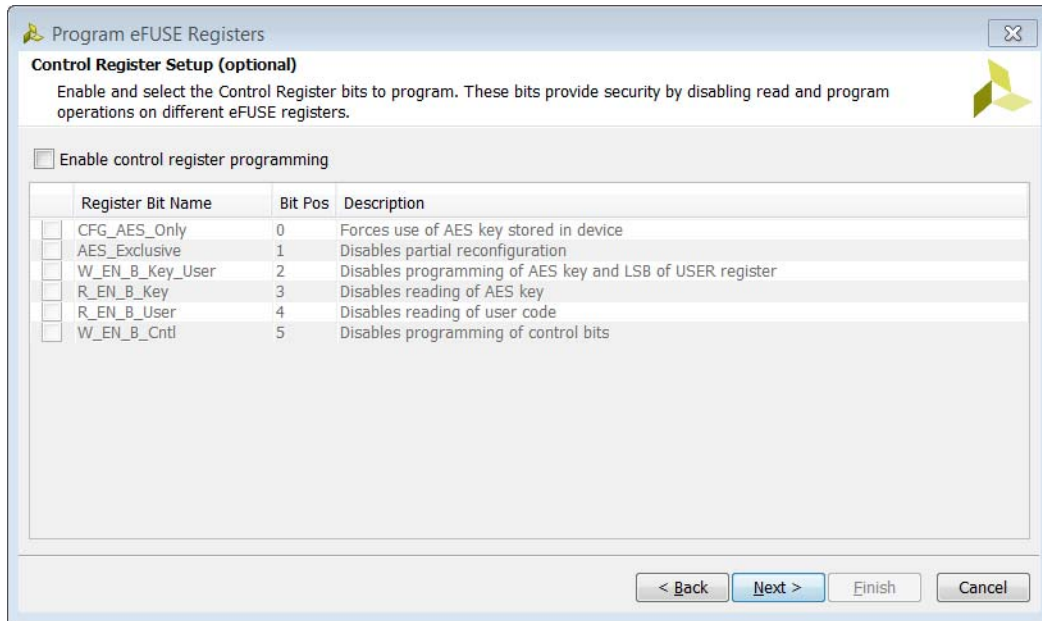


Figure 2-24: Program eFUSE Registers-ctrl-reg-7series

For UltraScale devices use the dialog box shown in Figure 2-25 to specify the FUSE_CNTL eFUSES.

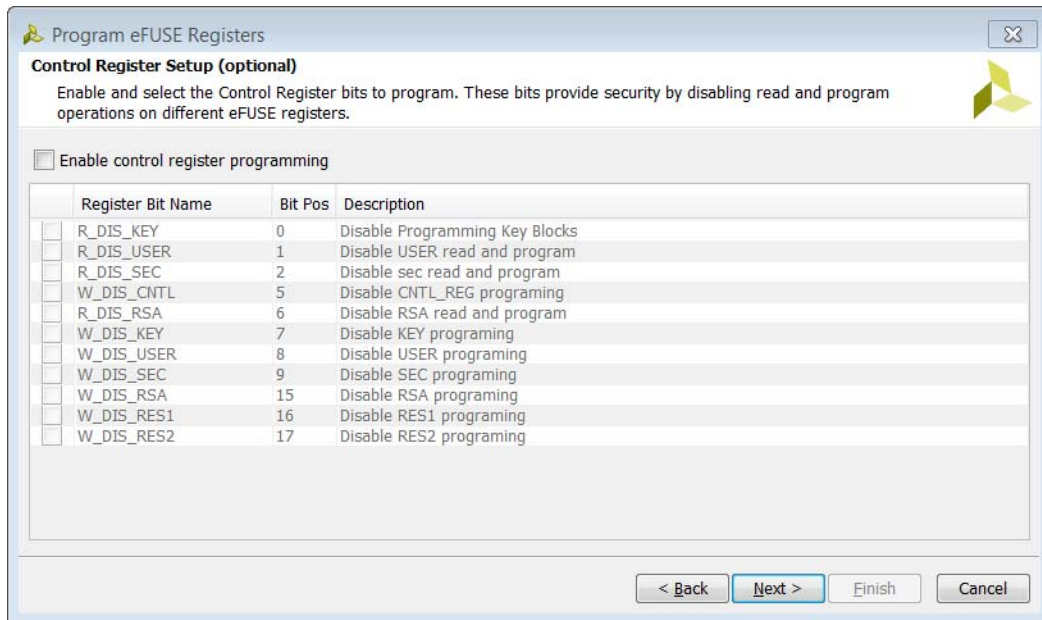


Figure 2-25: Program eFUSE Registers-ctrl-reg-ultrascale_optional

FUSE_SEC (UltraScale FPGAs only)

The FUSE_SEC eFUSEs control special device security settings in Ultrascale devices. There are six eFUSE bits controlling the security features shown in [Figure 2-26](#). For details refer to the *Ultrascale Architecture Configuration: Advance Specification User Guide* (UG570) [\[Ref 10\]](#)

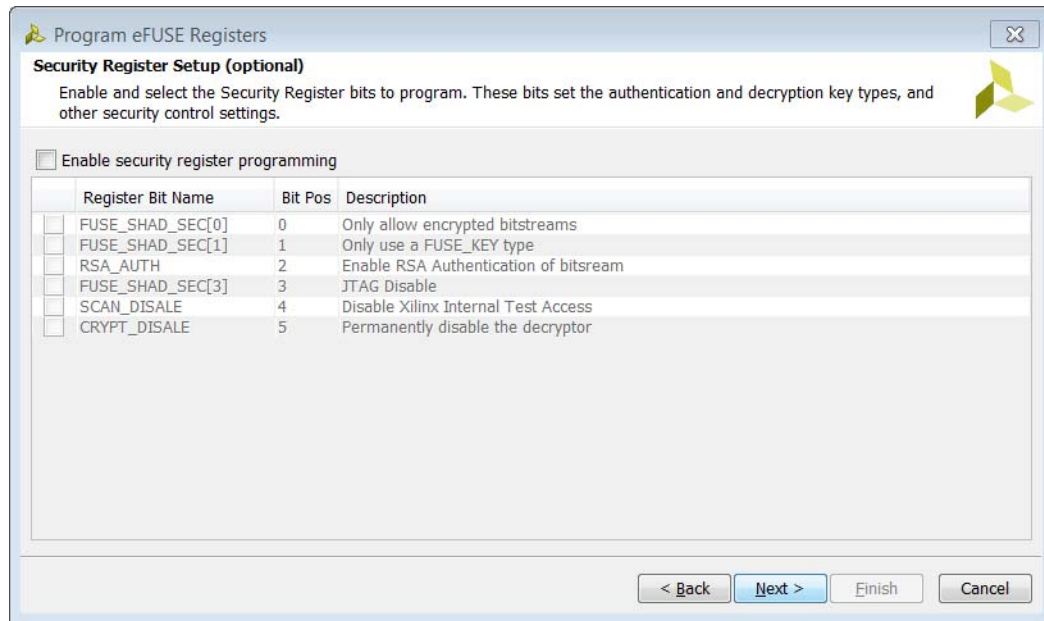


Figure 2-26: Program eFUSE Registers-sec-reg-ultrascale_optional



IMPORTANT: If the JTAG Disable bit is programmed, the JTAG interface will be disabled, preventing future test and configuration access to the device. This bit should only be programmed if JTAG access to the device is no longer required.

For information on these configuration features, see the *Ultrascale Architecture Configuration: Advance Specification User Guide* (UG570) [\[Ref 10\]](#) or the *7 Series FPGAs Configuration User Guide* (UG470) [\[Ref 8\]](#)

System Monitor

The System Monitor (SYSMON) Analog-to-Digital Converter (ADC) measures die temperature and voltage on the hardware device. The SYSMON monitors the physical environment via on-chip temperature and supply sensors. The ADC provides a high-precision analog interface for a range of applications.

The ADC can access up to 17 external analog input channels. Refer to *UltraScale Architecture System Monitor Advance Specification User Guide* (UG580) [\[Ref 11\]](#), or *7 Series*

FPGAs and Zynq-7000 All Programmable SoC XADC Dual 12-Bit 1 MSPS Analog-to-Digital Converter User Guide (UG480) [Ref 9] for more information on a specific device architecture.

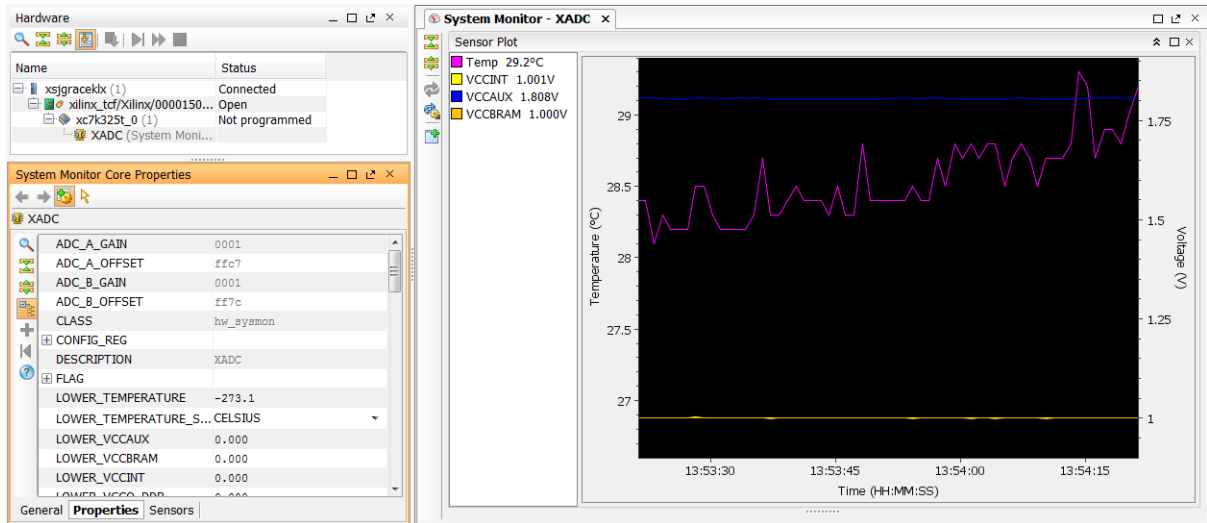


Figure 2-27: System Monitor

The hw_sysmon data is stored in dedicated registers called status registers accessible through the hw_sysmon_reg object. You can get the contents of the System Monitor registers by using the get_hw_sysmon_reg command.

Every device that supports the System Monitor automatically has one or more hw_sysmon objects created when refresh_hw_device is called. When the hw_sysmon object is created, it is assigned a property for all the temperature and voltage registers, as well as the control registers. On the hw_sysmon object, the values assigned to the temperature and voltage registers are already translated to Celsius/Fahrenheit and Volts.

Although you can use the get_hw_sysmon_reg command to access the hex values stored in registers of a System Monitor, you can also retrieve values of certain registers as formatted properties of the hw_sysmon object. For example, the following code retrieves the TEMPERATURE property of the specified hw_sysmon object rather than directly accessing the hex value of the register:

```
set opTemp [get_property TEMPERATURE [lindex [get_hw_sysmons] 0]]
```

Complete list of all the System Monitor commands can be found in [Table 5-16, page 102](#).

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"> Use mark_debug property to tag signals for debugging in HDL. Use the Set up Debug wizard to guide you through the Netlist Insertion probing flow.
Identify debug nets in synthesized design netlist without having to modify the HDL source code.	<ul style="list-style-type: none"> Use the Mark Debug right-click menu option to select nets for debugging in the synthesized design netlist. Use the Set up Debug wizard to guide you through the Netlist Insertion probing flow.
Automated debug probing flow using Tcl commands.	<ul style="list-style-type: none"> Use set_property Tcl command to set the mark_debug property on debug nets. Use Netlist Insertion probing flow Tcl commands to create debug cores and connect to them to debug nets.
Explicitly attach signals in the HDL source to an ILA debug core instance.	<ul style="list-style-type: none"> Identify HDL signals for debugging. 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.

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 XDC 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"}
define_attribute {n:char_fifo_din[*]} {syn_keep} {"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).

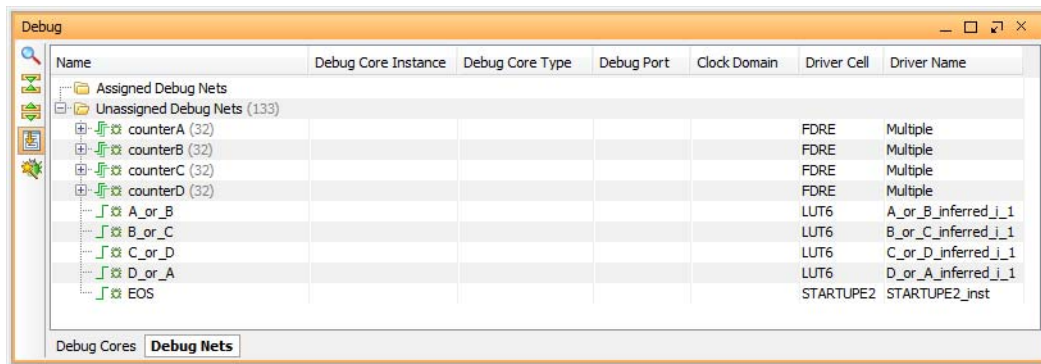


Figure 4-1: Unassigned Debug Nets

- 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, or click **Set up Debug** in the **Flow Navigator** under the **Synthesized Design** section.

3. Click **Next** to get to the **Specify Nets to Debug** panel (see [Figure 4-2](#)).
4. Optionally, click **Find Nets to Add...** 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.



IMPORTANT: You can also select nets in the Netlist or other windows, then drag them to the list of Nets to Debug.

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.

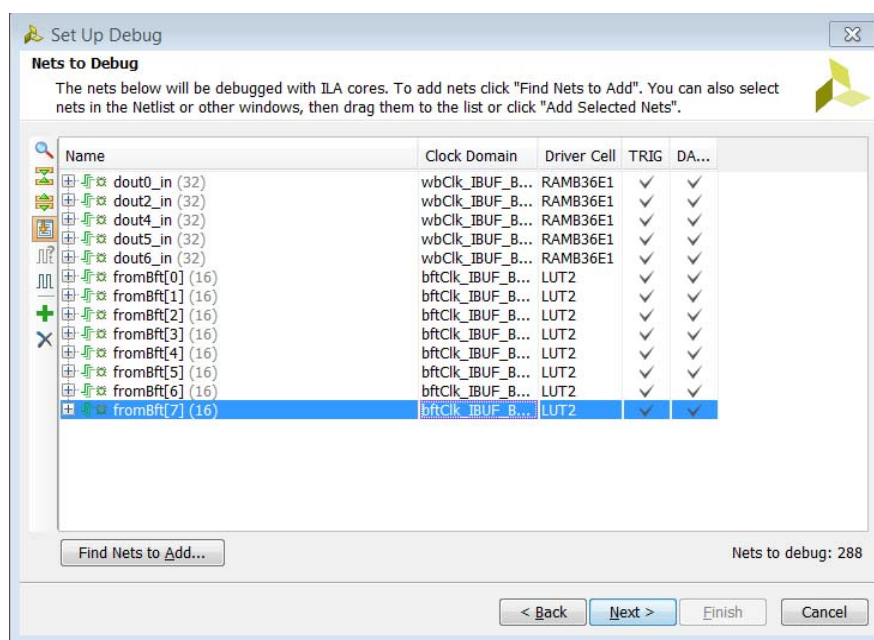


Figure 4-2: Set Up Debug Wizard

- Configure the ILA core general options such as ILA data depth (C_DATA_DEPTH), number of input pipe stages (C_INPUT_PIPE_STAGES), enabling the capture control feature (C_EN_STRG_QUAL), and enabling the advanced trigger feature (C_ADV_TRIGGER). Refer to [Table 4-2, page 52](#) for descriptions of these options.

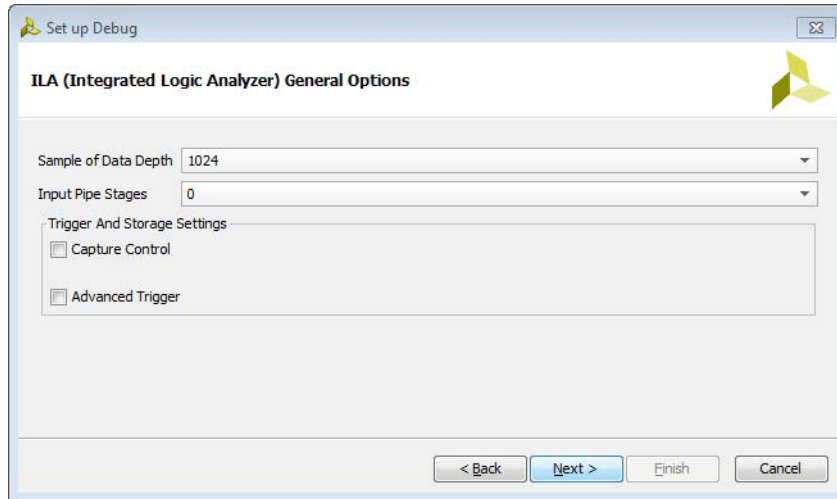


Figure 4-3: Configuring ILA General Options

- The debug nets are now assigned to the ILA debug core, as shown in [Figure 4-4](#).

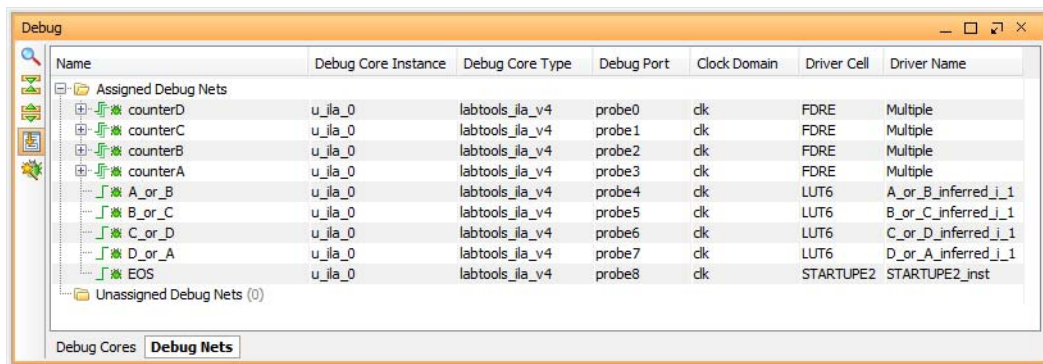


Figure 4-4: Assigned Debug Nets

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 4-5](#)), you can change the parent instance, debug core name, and set parameters for the core. To remove an existing debug core, right-click the core in the **Debug** window and select **Delete**. Refer to [Table 4-2, page 52](#) for a description of the ILA core options found in the **Create Debug Core** dialog.

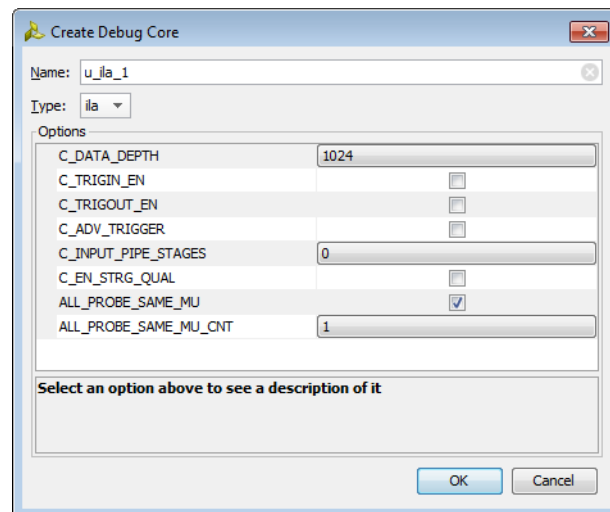


Figure 4-5: Creating a New Debug Core

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 4-6](#).
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**.

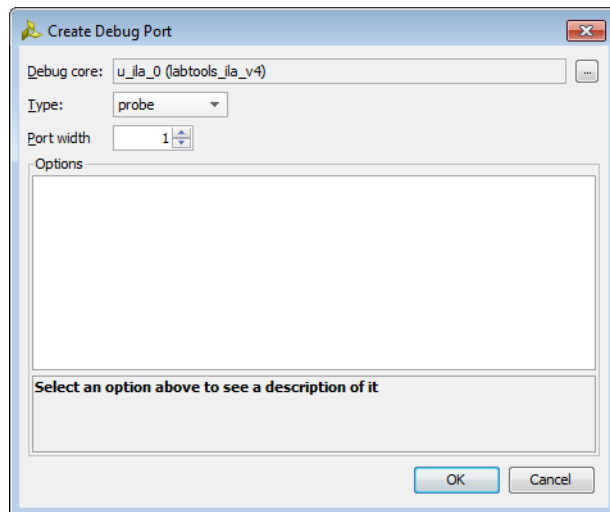


Figure 4-6: Creating a New Debug Port

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 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 60.

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 4-7](#)), 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.

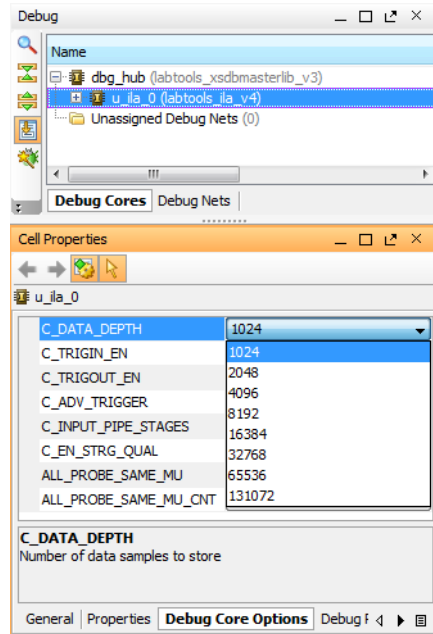


Figure 4-7: Changing the Data Depth of the ILA Core

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 Chapter 5 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 Chapter 5 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"> If C_ADV_TRIGGER is false and C_EN_STRG_QUAL is true, then set to 2. 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). 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). <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 ila
```

3. Set the various properties of the ILA core.

```
set_property C_DATA_DEPTH 1024 [get_debug_cores u_ila_0]
set_property C_TRIGIN_EN false [get_debug_cores u_ila_0]
set_property C_TRIGOUT_EN false [get_debug_cores u_ila_0]
set_property C_ADV_TRIGGER false [get_debug_cores u_ila_0]
set_property C_INPUT_PIPE_STAGES 0 [get_debug_cores u_ila_0]
set_property C_EN_STRG_QUAL false [get_debug_cores u_ila_0]
set_property ALL_PROBE_SAME_MU true [get_debug_cores u_ila_0]
set_property ALL_PROBE_SAME_MU_CNT 1 [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 probe0 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]}]]
```

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 may cause 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](#)).

Debug Core Insertion in Non-Project Mode

Debug cores can be inserted in either Project Mode or Non-Project Mode. The following sample Tcl script shows how to create the debug core, set debug core attributes, and connect the debug core probes to the signals in the design being probed. In Non-Project Mode, the insertion of the debug core needs to happen after synthesizing the design, and prior to the `opt_design` step as shown below.



IMPORTANT: *Debug core insertion is only supported for ILA cores.*

The following Tcl script is an example of using the debug core insertion commands in a Non-Project flow.

```
#read relevant design source files
read_vhdl ./*.vhd
read_verilog [ glob ./Sources/*.v ]
read_xdc ./target.xdc

#Synthesize Design

synth_design -top top -part xc7k325tffg900-2
```

```
#Create the debug core
create_debug_core u_ila_0 ila

#set debug core properties
set_property C_DATA_DEPTH 1024 [get_debug_cores u_ila_0]
set_property C_TRIGIN_EN false [get_debug_cores u_ila_0]
set_property C_TRIGOUT_EN false [get_debug_cores u_ila_0]
set_property C_ADV_TRIGGER false [get_debug_cores u_ila_0]
set_property C_INPUT_PIPE_STAGES 0 [get_debug_cores u_ila_0]
set_property C_EN_STRG_QUAL false [get_debug_cores u_ila_0]
set_property ALL_PROBE_SAME_MU true [get_debug_cores u_ila_0]
set_property ALL_PROBE_SAME_MU_CNT 1 [get_debug_cores u_ila_0]

#connect the probe ports in the debug core to the signals being probed in the design
set_property port_width 1 [get_debug_ports u_ila_0/clock]
connect_debug_port u_ila_0/clock [get_nets [list clk ]]
set_property port_width 1 [get_debug_ports u_ila_0/probe0]
connect_debug_port u_ila_0/probe0 [get_nets [list A_or_B]]
create_debug_port u_ila_0 probe
#Optionally, create more probe ports, set their width,
# and connect them to the nets you want to debug

#Implement design

opt_design
place_design
route_design
write_bitstream
```

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](#).

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)	v5.0	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

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 62](#)).
- 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 4-8](#)). 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) v4.0 Datasheet* (PG172) [[Ref 16](#)].
- For more information on customizing the VIO core, refer to *LogiCORE IP Virtual Input/Output (VIO) v3.0 Product Guide* (PG159) [[Ref 12](#)].
- 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 17](#)].

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.

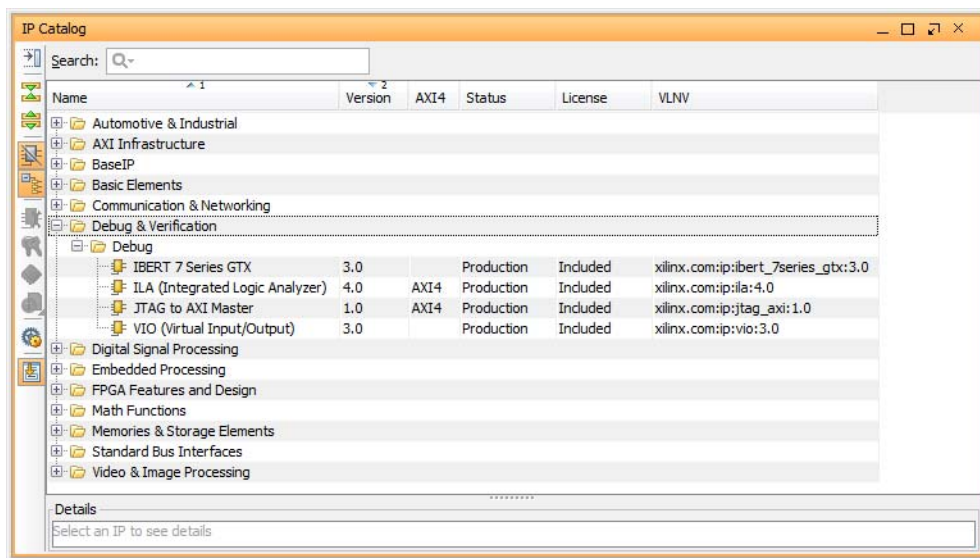


Figure 4-8: Debug Cores in the IP Catalog

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:

```
u_ila_0
(
  .clk(clk),
  .probe0(counterA),
  .probe1(counterB),
  .probe2(counterC),
  .probe3(counterD),
  .probe4(A_or_B),
  .probe5(B_or_C),
  .probe6(C_or_D),
  .probe7(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 (`dbg_hub`) 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 4-9](#)).

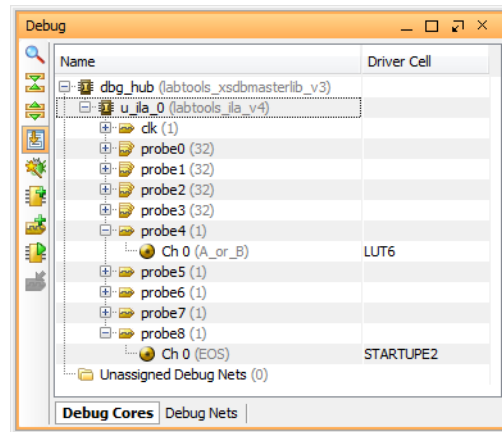


Figure 4-9: Debug Window Showing ILA Core and Debug Core Hub

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 4-10).



IMPORTANT: The default values for **C_USER_SCAN_CHAIN** are 1 or 3 for the debug hub core. If using a scan chain value other than 1 or 3 for the debug hub core, you must manually launch `hw_server`. To detect the debug hub at User Scan Chain of 2 or 4, use the following command:

```
-e "set xsdb-user-bscan <C_USER_SCAN_CHAIN scan_chain_number>"
```



IMPORTANT: If you plan to use the Microprocessor Debug Module (MDM) or other IP that uses the BSCAN primitive with the Vivado logic debug cores, you need to set the **C_USER_SCAN_CHAIN** property of the `dbg_hub` to a user scan chain that does not conflict with the other IP's Boundary Scan Chain setting. Failure to do so results in errors later in the implementation flow.

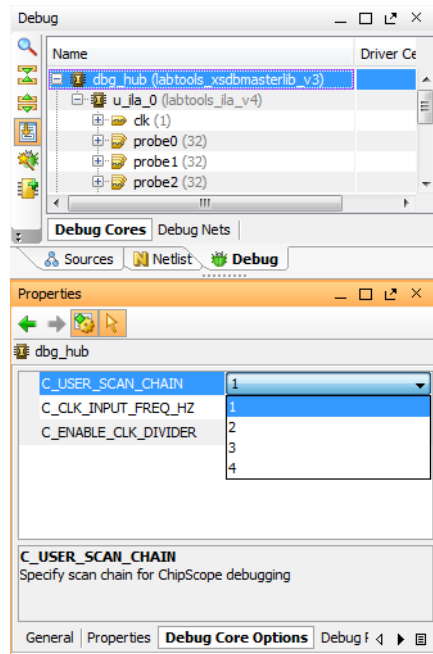


Figure 4-10: Changing the User Scan Chain Property of the Debug Core Hub

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 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.

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

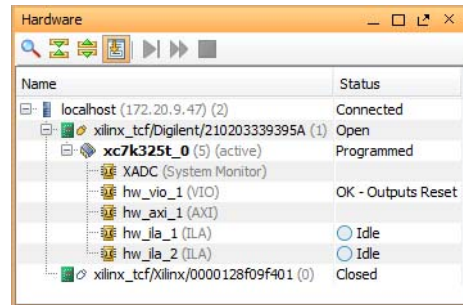


Figure 5-1: Hardware Window Showing Debug Cores

For more information on using the ILA core, refer to [Setting up the ILA Core to Take a Measurement, page 63](#). For more information on using the VIO core, refer to [Setting Up the VIO Core to Take a Measurement, page 85](#).

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 the device and select **Refresh Device**. 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 [Figure 5-2, page 64](#)).

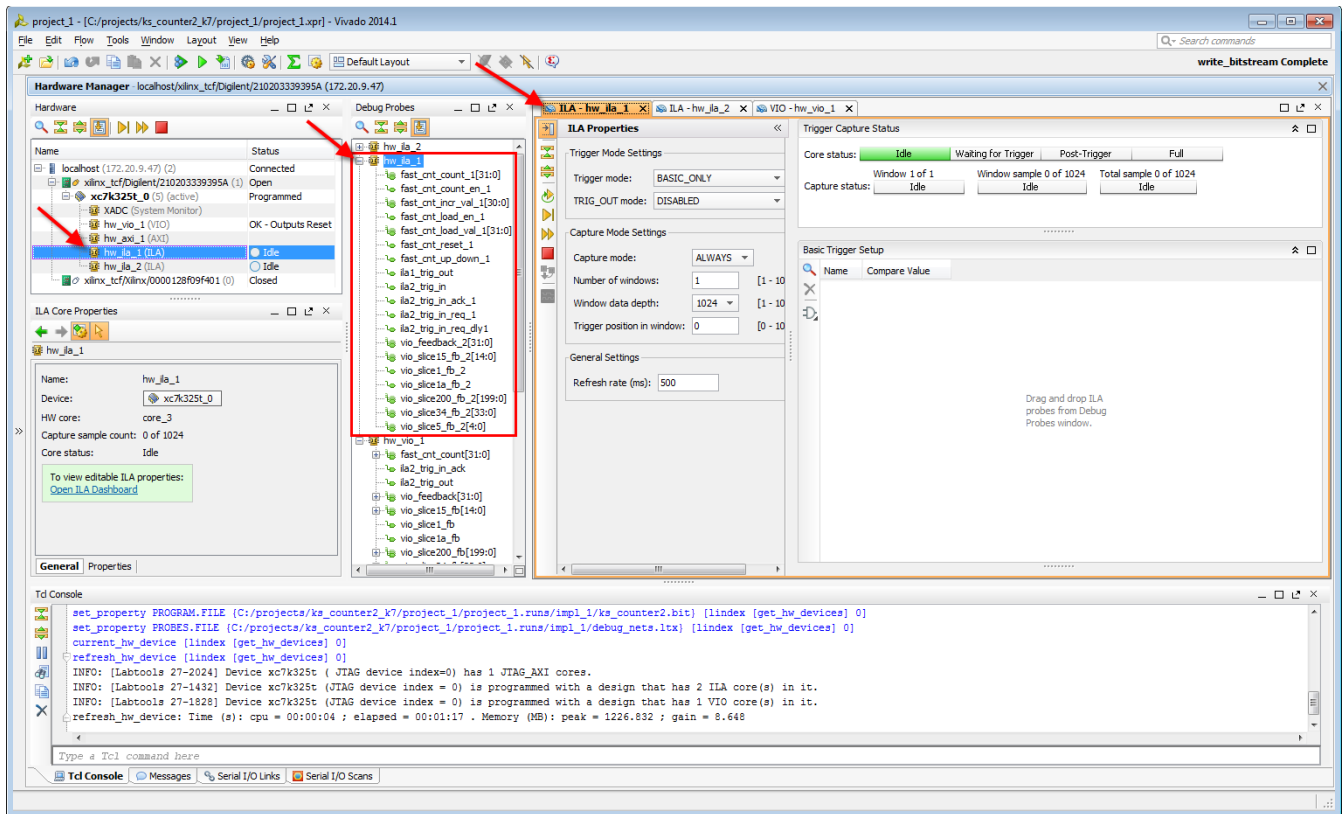


Figure 5-2: Selection of the ILA Core in Various Views

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 65). 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 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_ONLY" or "BASIC_OR_TRIG_IN" 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 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.

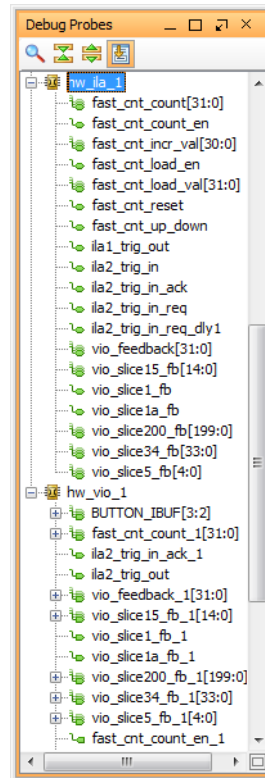


Figure 5-3: ILA Debug Probes

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.



IMPORTANT: *If you are using non-project mode, you must manually call the `write_debug_probes` command immediately following the `opt_design` command (or `place_design` command, if you do not use the `opt_design` command in your script).*

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 is 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]
```

Renaming Debug Probes

You can use the **Debug Probes** window to rename debug probes that belong to an ILA or VIO core (see [Figure 5-3, page 65](#)). You can rename the debug probes and add them to an existing Waveform Viewer for the core, or you can add them to the various trigger and/or capture windows of the ILA Dashboard. These names could be the *custom*, *long* or *short* name associated with the debug probe.

To perform these operations, right-click an ILA/VIO core's debug probe(s) and select one of the following:

- **Rename:** Prompts you to rename the probe to a custom name.
- **Name:** Allows you to select the long, short, or custom name of the debug probe. Subsequent references to the debug probe in the Vivado IDE window will use the name that you selected.
 - **Long** - Displays the full hierarchical name of the signal or bus being probed.
 - **Short** - Displays the name of the signal or bus being probed.
 - **Custom** - Displays the custom name given to the signal or bus when renamed.

Using the ILA Dashboard

The **ILA Dashboard** (see [Figure 5-4, page 67](#)) 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**.

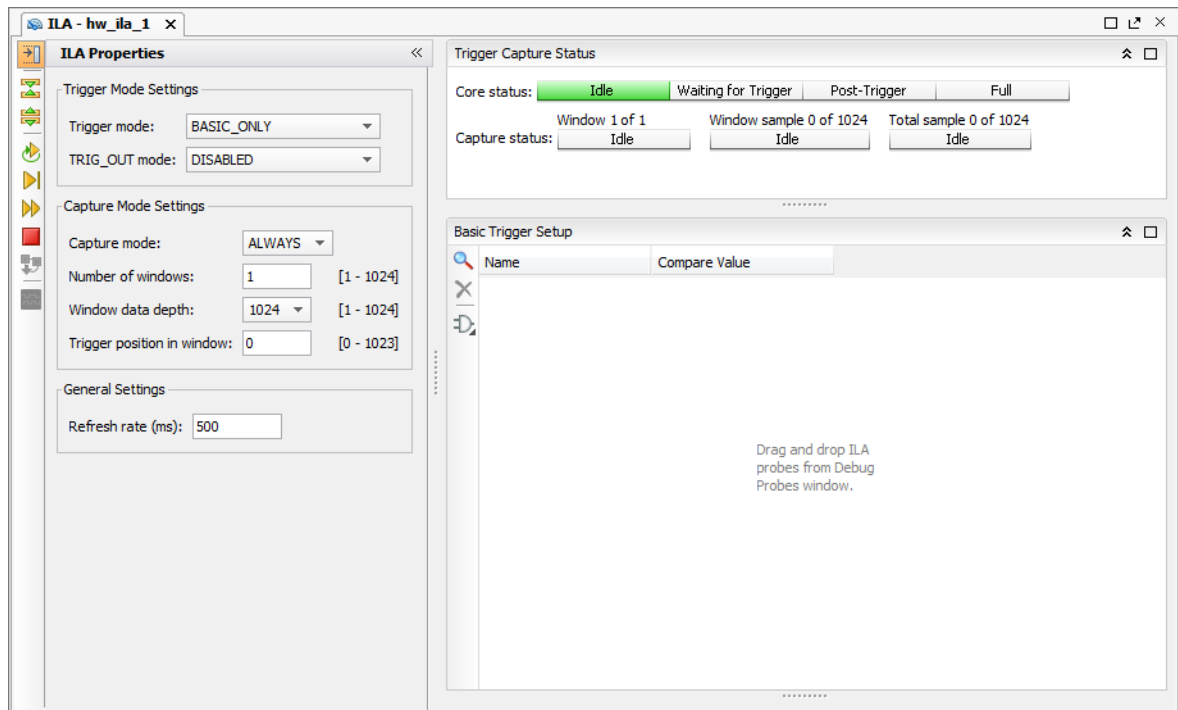


Figure 5-4: ILA Dashboard

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

- Set the trigger mode to trigger on various events in hardware:
 - **BASIC_ONLY:** The ILA Basic Trigger Mode can be used to trigger the ILA core when a basic AND/OR functionality of debug probe comparison result is satisfied.
 - **ADVANCED_ONLY:** The ILA Advanced Trigger Mode can be used to trigger the ILA core as specified by a user defined state machine.
 - **TRIG_IN_ONLY:** The ILA TRIG_IN Trigger Mode can be used to trigger the ILA core when the TRIG_IN pin of the ILA core transitions from a low to high.
 - **BASIC_OR_TRIG_IN:** The ILA BASIC_OR_TRIG_IN Trigger Mode can be used to trigger the ILA core when a logical OR-ing of the TRIG_IN pin of the ILA core and BASIC_ONLY trigger mode is desired.

- **ADVANCED_OR_TRIG_IN:** The ILA **ADVANCED_OR_TRIG_IN** Trigger Mode can be used to trigger the ILA core when a logical OR-ing of the **TRIG_IN** pin of the ILA core and **ADVANCED_ONLY** trigger mode is desired.
- Set the trigger output mode.
- Use **ALWAYS** and **BASIC** capture modes to control filtering of data to be captured.
- Set the number of ILA capture windows.
- 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

Use the basic trigger mode to describe a trigger condition that is a global Boolean equation of participating debug probe comparators. Basic trigger mode is enabled when the Trigger Mode is set to either **BASIC_ONLY** or **BASIC_OR_TRIG_IN**. Use the **Basic Trigger Setup** window (see [Figure 5-5](#)) to create this trigger condition and debug probe compare values.

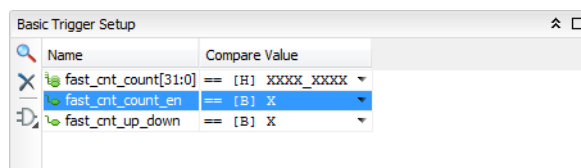


Figure 5-5: ILA Basic Trigger Setup Window

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_ONLY**, use the following command:

```
set_property CONTROL.TRIGGER_MODE BASIC_ONLY [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

Use the ILA debug probe trigger comparators 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).

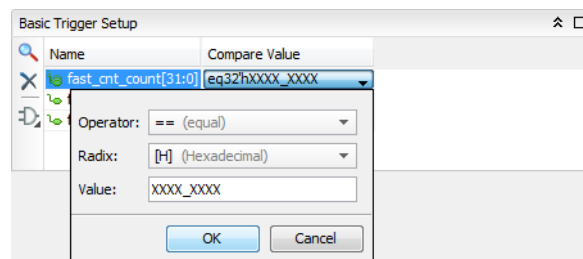


Figure 5-6: ILA Probe Compare Value Dialog Box

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](#).

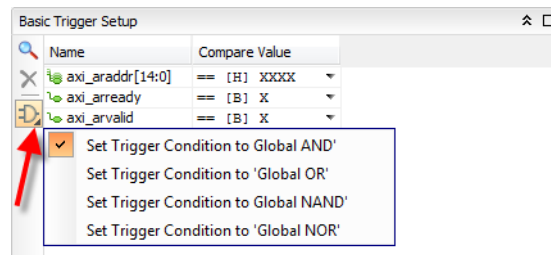


Figure 5-7: Setting the Basic Trigger Condition

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_ONLY** or **ADVANCED_OR_TRIG_IN**.

Specifying the Trigger State Machine Program File

When you set the **Trigger mode** to **ADVANCED_ONLY** or **ADVANCED_OR_TRIG_IN**, 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](#).

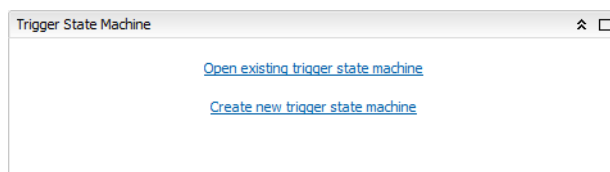


Figure 5-8: Creating or Opening a Trigger State Machine Program File

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](#)).

```

Trigger State Machine - new.tsm
C:/projects/ks_counter2_full/new.tsm
1 #####
2 #
3 # For info on creating trigger state machines:
4 # 1) In the main Vivado menu bar, select
5 # Window > Language Templates
6 # 2) In the Templates window, select
7 # Debug > Trigger State Machine
8 # 3) Refer to the entry 'Info' for an overview
9 # of the trigger state machine language.
10 #
11 # More information can be found in this document:
12 #
13 # Vivado Design Suite User Guide: Programming
14 # and Debugging (UG908)
15 #
16 #####
17 state my_state0:
18 trigger;
19

```

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 74](#)). 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](#).

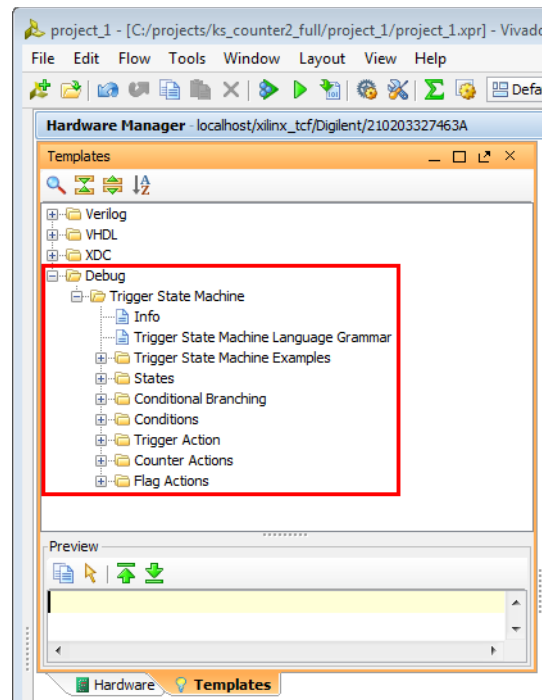


Figure 5-10: Trigger State Machine Language Templates

Compiling the Trigger State Machine

The trigger state machine is compiled every time you run the ILA trigger. If you would like to compile the trigger state machine without running or arming the ILA trigger, click the Compile trigger state machine button in the ILA Dashboard (see [Figure 5-11, page 75](#)).

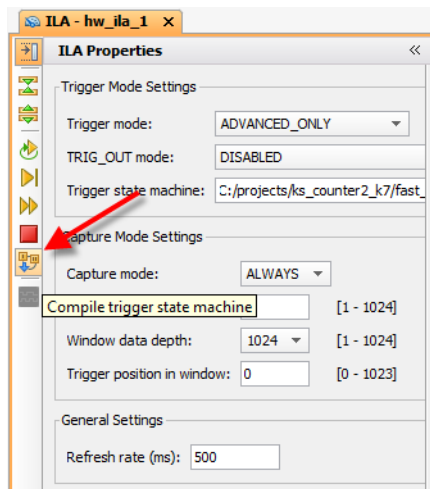


Figure 5-11: Compiling the Trigger State Machine without Arming the Trigger

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 has trigger input ports enabled, then you can use the following Trigger Mode settings to trigger on events on the TRIG_IN port:

- **BASIC_OR_TRIG_IN:** used to trigger the ILA core when a logical OR-ing of the TRIG_IN pin of the ILA core and BASIC_ONLY trigger mode is desired.
- **ADVANCED_OR_TRIG_IN:** used to trigger the ILA core when a logical OR-ing of the TRIG_IN pin of the ILA core and ADVANCED_ONLY trigger mode is desired.
- **TRIG_IN_ONLY:** used to trigger the ILA core when the TRIG_IN pin of the ILA core transitions from a low to high.

If the ILA core has trigger output ports enabled, then you can use the following TRIG_OUT Mode to control the propagation of trigger events to the TRIG_OUT port:

- **DISABLED:** disables the TRIG_OUT port.
- **TRIGGER_ONLY:** enables the result of the basic/advanced trigger condition to propagate to the TRIG_OUT port.
- **TRIG_IN_ONLY:** propagates the TRIG_IN port to the TRIG_OUT port.
- **TRIGGER_OR_TRIG_IN:** enables the result of a logical OR-ing of the basic/advanced trigger condition and TRIG_IN port to propagate to the TRIG_OUT port.

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 81](#) 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

Use the basic capture mode to describe a capture condition that is a global Boolean equation of participating debug probe comparators. Use the **Basic Capture Setup** window (see [Figure 5-12](#)) to create this capture condition and debug probe compare values.

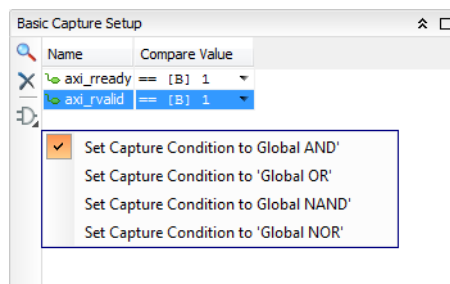


Figure 5-12: Setting the Basic Capture Condition

Configuring the Basic Capture Setup Window

The process for configuring debug probes and basic capture condition in the **Basic Capture** 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 68](#).
- 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 69](#)

- For information on setting the basic capture condition in the **Basic Capture Setup** window, refer to the section called [Setting Basic Trigger Condition, page 71](#). One key difference is the ILA core property used to control the capture condition is called CONTROL.CAPTURE_CONDITION.

Setting the Number of Capture Windows

The ILA capture data buffer can be subdivided into one or more capture windows, the depth each of which is a power of 2 number of samples from 1 to (((buffer size) / (number of windows)) - 1). For example, if the ILA data buffer is 1024 samples deep and is segmented into four capture windows, then each window can be up to 256 samples deep. Each capture window has its own trigger mark corresponding to the trigger event that caused the capture window to fill.



TIP: Clicking **Stop Trigger** before the entire ILA data capture buffer is full will upload and display all capture windows that have been filled. For example, if the ILA data buffer is segmented into four windows and three of them have filled with data, clicking **Stop Trigger** will halt the ILA core and upload and display the three filled capture windows.

The table below illustrates the interaction between the Vivado runtime software and hardware when a you set the **Number of Capture Windows** to more than 1 and the **Trigger Position** to 0.

Table 5-2: **Number of Capture Windows > 1 and Trigger Position = 0**

Software	Hardware
User Runs Trigger on the ILA core	<ul style="list-style-type: none"> • Window 0: ILA is armed • Window 0: ILA triggers • Window 0: ILA fills capture window 0 • Window 1: ILA is rearmed • Window 1: ILA triggers • Window 1: ILA fills capture window 1 • ... • Window n-1: ILA is rearmed • Window n-1: ILA triggers • Window n-1: ILA fills capture window n Entire ILA Capture Buffer is Full
Data is uploaded and displayed	The ILA core is rearmed such that it is ready to trigger on the clock cycle immediately following the last sample captured of the previous window.

The table below illustrates the interaction between the Vivado runtime software and hardware when a you set the **Number of Capture Windows** to more than 1 and the **Trigger Position** to greater than 0.

Table 5-3: **Number of Capture Windows > 1 and Trigger Position > 0**

Software	Hardware
User Runs Trigger on the ILA core	<ul style="list-style-type: none"> • Window 0: ILA is armed • Window 0: ILA fills capture buffer up to trigger position • Window 0: ILA triggers • Window 0: ILA fills the rest of capture window 0 • Window 1: ILA is rearmed • Window 1: ILA fills capture buffer up to trigger position • Window 1: ILA triggers • Window 1: ILA fills capture buffer • Window 1: ILA fills window 1 • ... • Window n-1: ILA is rearmed • Window n-1: ILA fills capture buffer up to trigger position • Window n-1: ILA triggers • Window n-1: ILA fills capture buffer • Window n-1: ILA fills window n <p>Entire ILA Capture Buffer is Full</p>
Data is uploaded and displayed	Triggers could be missed between two windows since the ILA now has to fill the capture data up to the trigger position.

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 marker in the captured data window. You can set the trigger position to any sample number in the captured data window. For instance, in the case of a captured data window that is 1024 samples deep:

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

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 51](#) 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 5-13](#)).

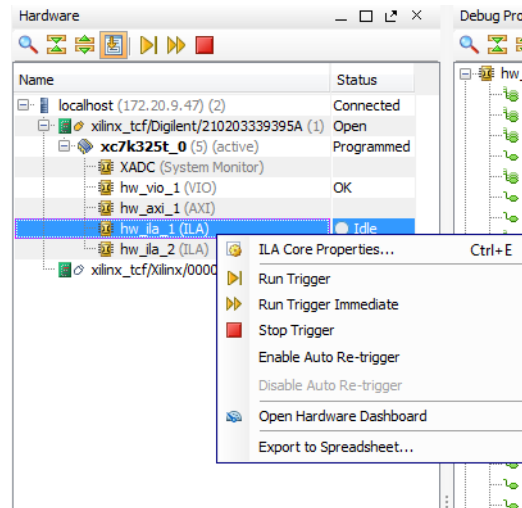


Figure 5-13: ILA Core Trigger Commands



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 5-13).

Using Auto Re-Trigger

Select the **Enable Auto Re-Trigger** right-click menu option (or corresponding button on the ILA Dashboard toolbar) on the ILA core to enable Vivado IDE to automatically re-arm the ILA core trigger after a successful trigger+upload+display operation has completed. The captured data displayed in the waveform viewer corresponding to the ILA core will be overwritten upon each successful trigger event. The **Auto Re-Trigger** option can be used with the **Run Trigger** and **Run Trigger Immediate** operations. Click **Stop Trigger** to stop the trigger currently in progress.

The table below illustrates the interaction between the Vivado IDE runtime software and hardware when you invoke the **Auto Re-Trigger** option.

Table 5-4: **Auto Re-Trigger**

Software	Hardware
Click the Auto Re-trigger option on the ILA core	<ul style="list-style-type: none"> • ILA is armed • ILA triggers • ILA fills capture buffer • ILA is full
Data is uploaded and displayed	<ul style="list-style-type: none"> • ILA is rearmed • ILA triggers • ILA fills capture buffer • ILA is full <p>Lots of cycles are missed between the ILA "full" event and display of the ILA data</p>



IMPORTANT: *As there is a delay between the time the ILA data is full and the data is uploaded and displayed in the GUI, there is a very high probability of missing cycles between these events where the ILA could have triggered.*

Viewing Trigger and Capture Status

The ILA debug core trigger and capture status is displayed in two places in 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 5-5, page 82](#)).

Table 5-5: 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 window. 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 window. 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.

Partial Buffer Capture

Clicking **Stop Trigger** before the entire ILA data capture buffer is full uploads and displays all capture windows that have been filled. For example, if the ILA data buffer is segmented into four windows and three of them have filled with data, clicking **Stop Trigger** halts the ILA core and uploads and displays the three filled capture windows. In addition, clicking **Stop Trigger** will halt the ILA core and display a partially filled capture window so long as the trigger event occurred in that capture window.

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 5-14](#)):

- **Core status:** indicates the status of the ILA core trigger/capture engine (see [Table 5-5](#) for a description of the status indicators)
- **Capture status:** indicates the current capture window, the current number of samples captured in the current capture window, and the total number of samples captured by the ILA core. These values are all reset to 0 once the ILA core status is Idle.

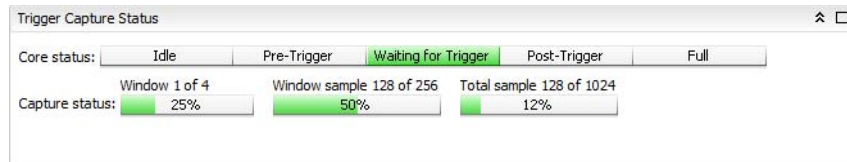


Figure 5-14: Basic Trigger Mode Trigger Capture Status Window

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 5-15):

- **Core status:** indicates the status of the ILA core trigger/capture engine (see Table 5-5, page 82 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 status:** indicates the current capture window, the current number of samples captured in the current capture window, and the total number of samples captured by the ILA core. These values are all reset to 0 once the ILA core status is **Idle**.

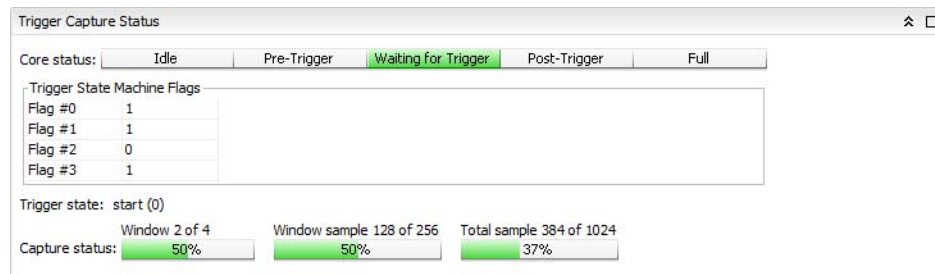


Figure 5-15: Advanced Trigger Mode Trigger Capture Status Window

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 `.1tx` 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. If you are in project mode, open the Synthesized or Netlist Design. If you are in non-project mode, open the synthesized design checklist.
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 in the 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.ila [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.ila`. 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.ila]
```

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.



IMPORTANT: Do NOT use the `open_wave_config` command to open a waveform created from ILA captured data. This is a simulator-only command and will not function correctly with ILA captured data waveforms.

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 5-16](#)) 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 5-17](#)).

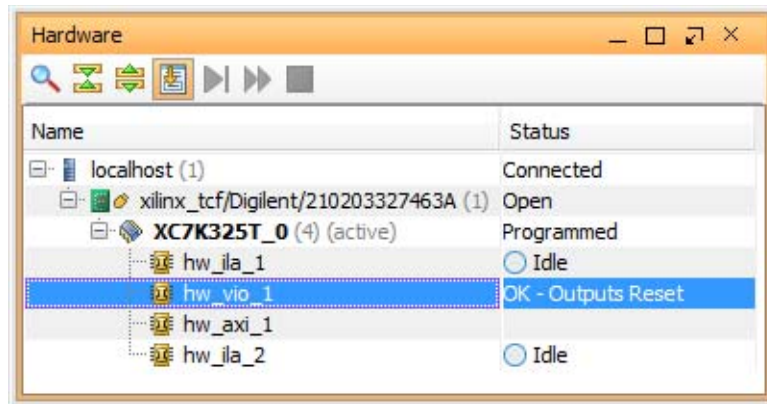


Figure 5-16: VIO Core in the Hardware Window

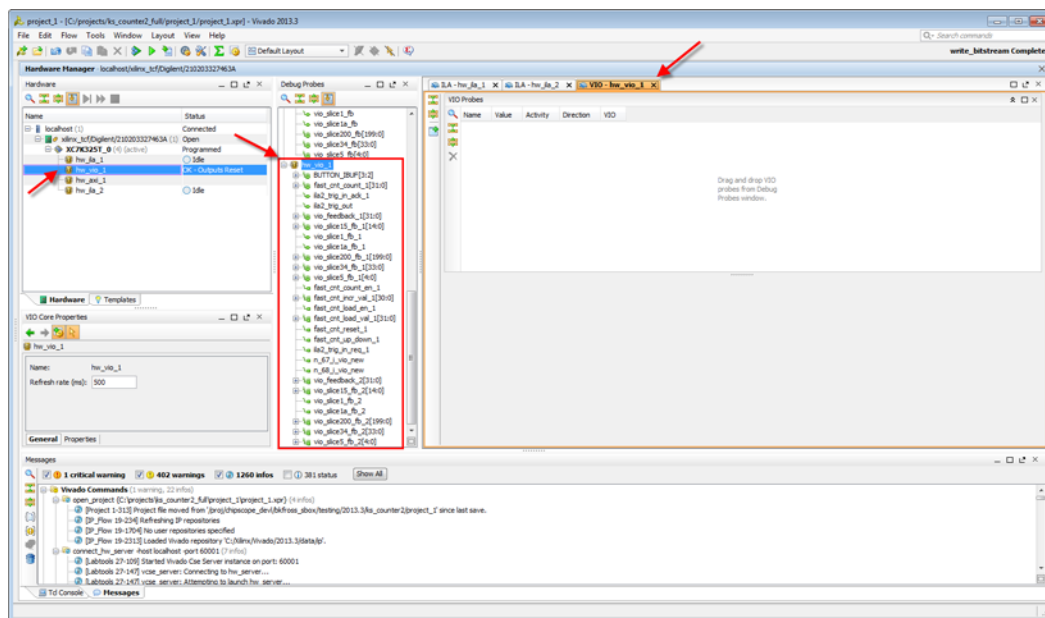


Figure 5-17: Selection of the VIO Core in Various Views

The VIO core can become out-of-sync with the Vivado IDE. Refer to [Viewing the VIO Core Status, page 87](#) 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 89](#) 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 92](#) 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 5-6](#).

Table 5-6: 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"> • 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. • 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.

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 5-18](#)). 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**. You can also use drag and drop mouse gestures to add VIO Probes to the VIO Dashboard window and re-arrange VIO Probes within the VIO Dashboard 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.

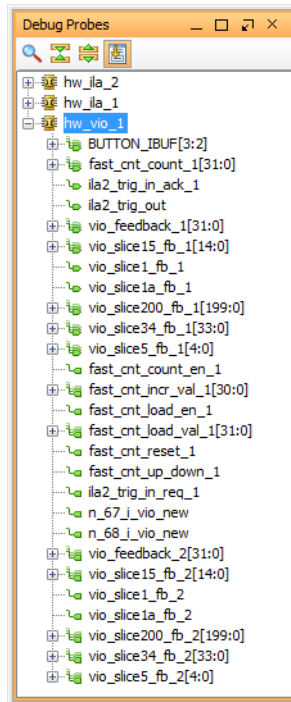


Figure 5-18: VIO Debug Probes

Using the VIO Dashboard

The **VIO Dashboard** (see Figure 5-19) 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**.

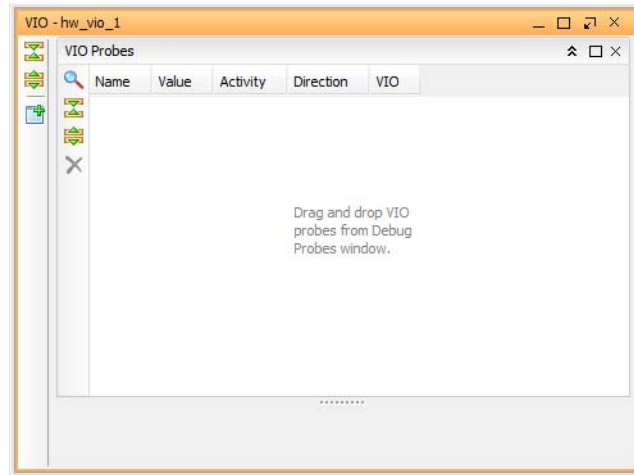


Figure 5-19: VIO Dashboard

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 87](#) 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 5-20](#).

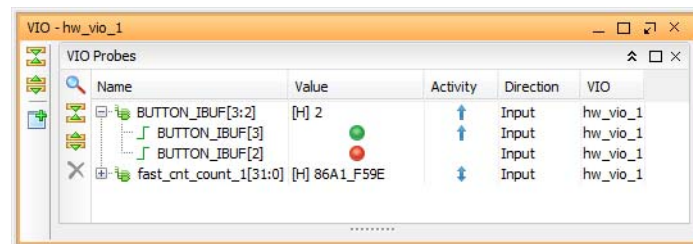


Figure 5-20: Core Input Probes

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 5-20](#)). 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 87](#) 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 5-21](#).

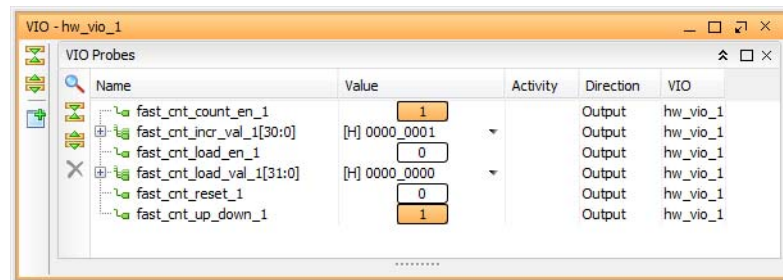


Figure 5-21: VIO Outputs in the VIO Probes window of the VIO Dashboard

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 5-21](#)). 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 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 5-22) to see its properties in the **AXI Core Properties** window.

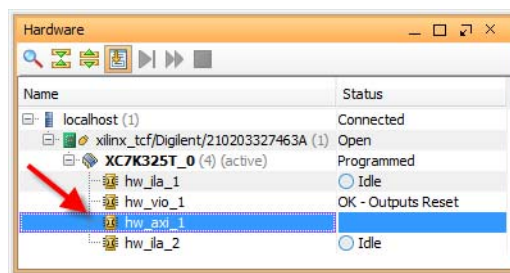


Figure 5-22: JTAG-to-AXI Master Core in the Hardware Window

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 WRITE -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 or Vivado Hardware Server (Standalone) on your remote lab machine, and use the Vivado logic analyzer feature on your local machine to connect to a remote instance of the Vivado Hardware Server (`hw_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* (UG973) [\[Ref 4\]](#).



IMPORTANT: *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 hw_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 Hardware 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 Hardware Server (hw_server.bat on Windows platforms or hw_server on Linux platforms) that is running on the lab machine:

1. Install the latest version of the Vivado Design Suite or Vivado Hardware Server (Standalone) on the lab machine.



IMPORTANT: *You do NOT need to install the full Vivado Design Suite on the lab machine to only use the remote hardware server feature. However, if you do want to use the Vivado Hardware Manager features (such as the Vivado logic analyzer or Vivado serial I/O analyzer) on the lab machine, then you will need to install the full Vivado Design Suite on the lab machine. Also, You do NOT need any software licenses to run the Hardware Server or any of the Hardware Manager features of the Vivado tools.*

2. Start up the hw_server application on the remote lab machine. Assuming you installed the Vivado Hardware Server (Standalone) to the default location and your lab machine is a 64-bit Windows machine, here is the command line:

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

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.

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 5-7](#)).

Table 5-7: Hardware Manager Tcl Objects

Tcl Object	Description
hw_server	Object referring to hardware 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.
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 5-8 contains descriptions of all Tcl commands used to interact with hardware servers.

Table 5-8: 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 hardware servers.
refresh_hw_server	Refresh a connection to a hardware server.

Description of hw_target Tcl Commands

Table 5-9 contains descriptions of all Tcl commands used to interact with hardware targets.

Table 5-9: 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 5-10 Descriptions of hw_device Tcl Commands contains descriptions of all Tcl commands used to interact with hardware devices.

Table 5-10: Descriptions of hw_device Tcl Commands

Tcl Command	Description
current_hw_device	Get or set the current hardware device.
get_hw_devices	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 5-11 Descriptions of hw_ila Tcl Commands contains descriptions of all Tcl commands used to interact with ILA debug cores.

Table 5-11: 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 5-12 Descriptions of hw_ila_data Tcl Commands contains descriptions of all Tcl commands used to interact with captured ILA data.

Table 5-12: 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 5-13 contains descriptions of all Tcl commands used to interact with captured ILA data.

Table 5-13: Descriptions of hw_probe Tcl Commands

Tcl Command	Description
get_hw_probes	Get list of hardware probes.

Description of hw_vio Tcl Commands

Table 5-14 contains descriptions of all Tcl commands used to interact with VIO cores.

Table 5-14: 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_vio	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.

Description of hw_axi and hw_axi_txn Tcl Commands

Table 5-15 contains descriptions of all Tcl commands used to interact with JTAG-to-AXI Master cores.

Table 5-15: Description of hw_axi and hw_axi_txn Tcl Commands

Tcl Command	Description
create_hw_axi_txn	Creates hardware AXI transaction object.
delete_hw_axi_txn	Deletes hardware AXI transaction objects.
get_hw_axi_txns	Gets a list of hardware AXI transaction objects.
get_hw_axis	Gets a list of hardware AXI objects.
refresh_hw_axi	Refreshes hardware AXI object status.
report_hw_axi_txn	Reports formatted hardware AXI transaction data.
reset_hw_axi	Resets hardware AXI core state.
run_hw_axi	Runs hardware AXI read/write transactions and update transaction status in the corresponding hw_axi object.

Description of hw_sysmon Tcl Commands

Table 5-16 contains descriptions of all Tcl commands used to interact with System Monitor core.

Table 5-16: Descriptions of hw_sysmon Tcl commands

Tcl Command	Description
commit_hw_sysmon	Commits the current property values defined on a hw_sysmon object to the System Monitor registers on the hardware device.
get_hw_sysmon_reg	Returns the hex value of the System Monitor register defined at the specified address of the specified hw_sysmon object.
get_hw_sysmons	Returns the list of Sysmon debug core objects defined on the current hardware device.
refresh_hw_sysmon	Refreshes the properties of the hw_sysmon object with the values on the System Monitor from the current hw_device.
set_hw_sysmon_reg	Sets the System Monitor register at the specified address to the hex value specified.

Note: Detailed help for each of these commands can be obtained by typing `<command name> -help` on the Vivado TCL Console.

Using Tcl Commands to Interact with a JTAG-to-AXI Master Core

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 hw_server running on localhost:3121.
- Single JTAG-to-AXI Master core in a design running in the XC7K325T device on the KC705 board.
- JTAG-to-AXI Master core is in an AXI-based system that has an AXI BRAM Controller Slave core in it.

Example Tcl Command Script

```
# Connect to the Digilent Cable on localhost:3121
connect_hw_server -url localhost:3121
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]

# Reset the JTAG-to-AXI Master core
reset_hw_axi [get_hw_axis hw_axi_1]

# Create a read transaction bursts 128 words starting from address 0
create_hw_axi_txn read_txn [get_hw_axis hw_axi_1] -type read \
-address 00000000 -len 128

# Create a write transaction bursts 128 words starting at address 0
# using a repeating fill value of 11111111_22222222_33333333_44444444
# (where LSB is to the left)
create_hw_axi_txn write_txn [get_hw_axis hw_axi_1] -type write \
-address 00000000 -len 128 -data {11111111_22222222_33333333_44444444}

# Run the write transaction
run_hw_axi [get_hw_axi_txns wrte_txn]

# Run the read transaction
run_hw_axi [get_hw_axi_txns read_txn]
```

Using Tcl Commands to Take an ILA Measurement

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:3121.
- 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:3121
connect_hw_server -url localhost:3121
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]
```

Trigger At Startup

The Trigger at Startup feature is used to configure the trigger settings of an ILA core in a design .bit file so that it is pre-armed to trigger immediately after device startup. You do this by taking the various trigger settings that ordinarily get applied to an ILA core running in a design in hardware, and applying them to the ILA core in the implemented design.



IMPORTANT: *The following process for using Trigger at Startup assumes that you have a valid ILA 4.0 design working in hardware, and that the ILA 4.0 core has NOT been flattened during the synthesis flow.*

To use the Trigger at Startup feature perform the following steps:

1. Run through the first pass of the ILA flow as usual to set up the trigger condition.
 - a. Open the target, configure the device, and bring up the ILA Dashboard.
 - b. Enter the trigger equations for the ILA core in the ILA Dashboard.
2. From the Vivado Tcl command line, export the trigger register map file for the ILA core. This file contains all of the register settings to "stamp" back on to the implemented netlist. The output from this is a single file.


```
% run_hw_ila -file ila_trig.tas [get_hw_ilas hw_ila_1]
```

3. Go back and open the previously implemented routed design in Vivado IDE. There are two ways to do this depending on your project flow.
 - a. Project Mode: Use the **Flow Navigator** to open the implemented design.
 - b. Non-Project Mode: Open your routed checkpoint: `%open_checkpoint <file>.dcp`
4. At the Implemented Design Tcl console, apply the trigger settings to the current design in memory, which is your routed netlist.

```
%apply_hw_ila_trigger ila_trig.tas
```

Note: If you see an ERROR indicating that the ILA core has been flattened during synthesis, you will need to regenerate your design and force synthesis to preserve hierarchy for the ILA core. Ensure that you have a valid ILA 4.0 design working in hardware, and that the ILA 4.0 core has NOT been flattened during the synthesis flow

5. At the Implemented Design Tcl console, write the bitstream with Trigger at Startup settings.



IMPORTANT: To pick up the routed design changes do this at the tcl command console only:

```
write_bitstream trig_at_startup.bit
```

6. Go back to the **Hardware Manger** and reconfigure with the new `.bit` file that you generated in the previous step. You will have to set the property for the updated `.bit` file location either through the GUI or through a Tcl command. Make sure you set the new `.bit` file as the one to use for configuration in the hardware tool.
 - a. Select the device in the hardware tree
 - b. Assign the `.bit` file generated in [step 5](#)
7. Program the device using the new `.bit` file.

Once programmed, the new ILA core should immediately arm at startup. You should see an indication in the Trigger Capture Status for the ILA core. If trigger or capture events have occurred, the ILA core is now populated with captured data samples.

Viewing ILA Probe Data in the Waveform Viewer

Introduction

The ILA waveform viewer in the Vivado® Integrated Design Environment (IDE) provides a powerful way to analyze data captured from the ILA Debug Core. After successfully triggering an ILA core and capturing data, Vivado automatically populates a corresponding waveform viewer with data collected from the ILA core. When using Vivado in project mode, configurable waveform settings such as coloring, radix selection, and signal ordering persist and are conveniently remembered between Vivado sessions.

ILA Data and Waveform Relationship

It is useful to understand the relationship between the `hw_ila_data` captured ILA data object and the waveform, as shown in [Figure 6-1](#).

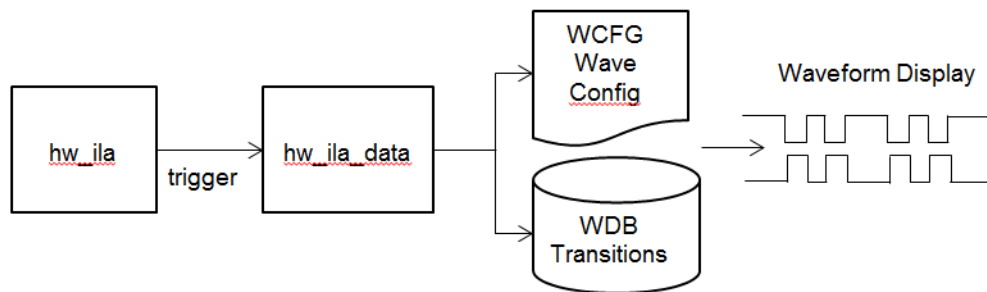


Figure 6-1: ILA Data and Waveform Relationship

The `hw_ila` Tcl object represents the ILA core in hardware. Every time an ILA core uploads captured data, it is stored in memory in a corresponding Tcl `hw_ila_data` object. These objects are named predictably so the first ILA core in hardware `'hw_ila_1'` produces data in a corresponding Tcl data object named `'hw_ila_data_1'` after trigger and upload. When working online with hardware, every waveform is backed by the in-memory `hw_ila_data` object and has a 1:1 correspondence with this object illustrated by the

diagram in [Figure 6-1](#). For each Tcl `hw_ila_data` object, a wave database (WDB) file and wave configuration (WCFG) file are created and automatically tracked in a directory of the Vivado project. [Figure 6-1](#) illustrates the flow of data from the hardware `hw_ila` on the left through to the waveform display on the right.

The combination of the wave configuration, WCFG, file and wave transition database, WDB, file contain the waveform database and customizations displayed in the Vivado waveform user interface. These waveform files are automatically managed in the Vivado ILA flow and users are not expected to modify the WDB or WCFG files directly. The wave configuration can be modified by changing objects in the waveform viewer (such as signal color, bus radix, signal order, markers, etc) then clicking the **Save** button. This automatically saves the wave configuration changes to the appropriate WCFG file in the Vivado project.

It is possible to archive waveform configurations and data for later viewing by using the Tcl command `write_hw_ila_data`. This stores the `hw_ila_data`, wave database and wave configuration in an archive for later viewing offline. See the section, [Saving and Restoring Captured Data from the ILA Core in Chapter 5](#) for details on using `read_hw_ila_data` and `write_hw_ila_data` for offline storage and retrieval of waveforms.

Waveform Viewer Layout

The ILA waveform viewer (sometimes referred to as waveform configuration) is composed of several dynamic objects working together to provide a complete visualization tool for the captured ILA data, as shown in [Figure 6-2](#).

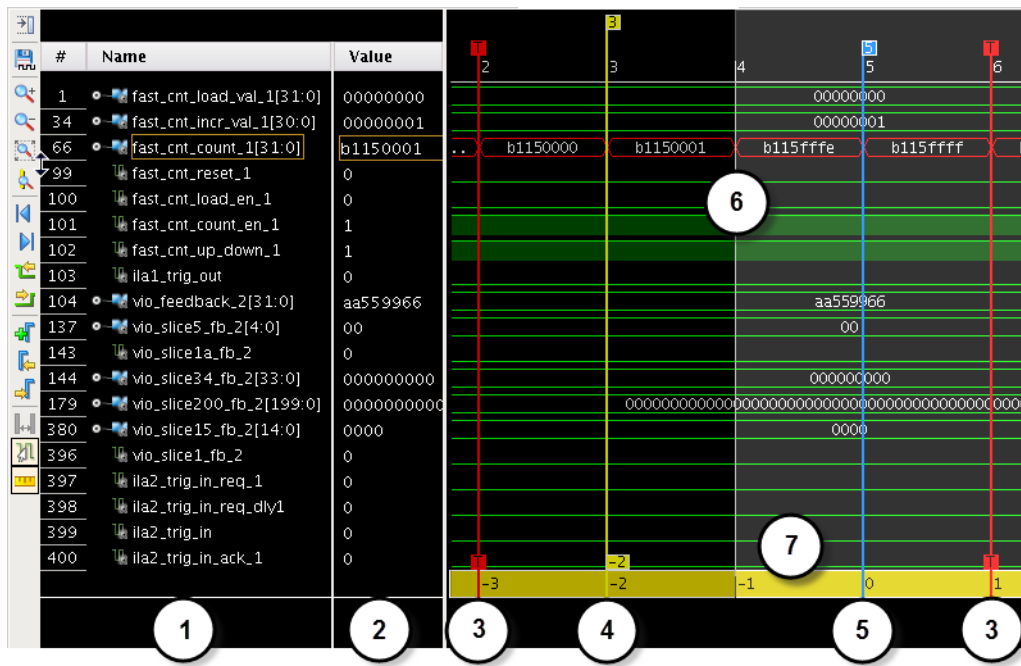


Figure 6-2: Waveform Viewer Showing Captured ILA Data

The description for the labelled objects in [Figure 6-2](#) is as follows:

1. Net or Bus Name from the ILA probes file (.1tx)
2. Net or Bus Value at the cursor
3. Trigger Markers (red lines)
4. Cursor (yellow line)
5. Markers (blue line)
6. ILA capture window transitions (alternating clear/grey regions)
7. Floating measurement ruler (yellow bar)

Waveform Viewer Operation

The scalars and buses shown in the **Name** column of the wave viewer represent the names of the probe design objects in the waveform (see [Figure 6-3](#)). These correspond to the hardware probes of the ILA core (see the `get_hw_probes` Tcl command).

Name	Value
fast_cnt_load_val_1[31:0]	00000000
fast_cnt_incr_val_1[30:0]	00000001
fast_cnt_count_1[31:0]	0d7b0004
fast_cnt_reset_1	0
fast_cnt_load_en_1	0
fast_cnt_count_en_1	1
fast_cnt_up_down_1	1
ila1_trig_out	0
vio_feedback_2[31:0]	aa559966
vio_slice5_fb_2[4:0]	00
vio_slice1a_fb_2	0
vio_slice34_fb_2[33:0]	00000000
vio_slice200_fb_2[199:0]	00000000
vio_slice15_fb_2[14:0]	0000
vio_slice1_fb_2	0
ila2_trig_in_req_1	0

Figure 6-3: ILA Probe Names and Values Shown in Waveform Viewer

Immediately after triggering and uploading ILA data for the first time, the waveform viewer populates with all probes connected to the ILA core. It is possible to customize probes in the viewer in addition to removing existing probes or adding new probes to the viewer. This section covers the basic operation of the waveform viewer.

Removing Probes from the Waveform

All probes by default are added to the waveform during the first trigger and upload operation. If you do not want the waveform to contain all probes, it is simple to remove probes from the viewer.

To remove a probe from the waveform viewer, right-click the scalar or bus to delete in the **Name** column and select **Delete** from the pop up menu. Alternatively, select the signal or bus to delete and press the **Delete** key. Probe transition data is not actually deleted from memory it is just hidden from view when probes are removed.

Adding Probes to the Waveform

To add probes to the waveform, select the Probes to add for the associated ILA core in the **Debug Probes** window, right-click, and select **Add Probes to Waveform** from the pop-up menu.

To add another copy of a signal or bus to the waveform window, select the signal or bus in the waveform window. Then select **Edit > Copy** or type **Ctrl+C**. This copies the object to the clipboard. Select **Edit > Paste** or type **Ctrl+V** to paste a copy of the object in the waveform.

Using the Zoom Features

Toolbar buttons provide quick access to waveform zooming features (see [Figure 6-4](#)). Alternatively, use the mouse wheel combined with the **CTRL** key to zoom in and out of the currently selected waveform. It is important to note the zoom level is not persistent and will be reset between Vivado sessions.



Figure 6-4: Waveform Zoom Buttons

Waveform Options Dialog Box

The waveform viewer allows you do customize the way objects are displayed.

When you select the Waveforms Options button the **Waveform Options** dialog box in [Figure 6-5](#) opens:

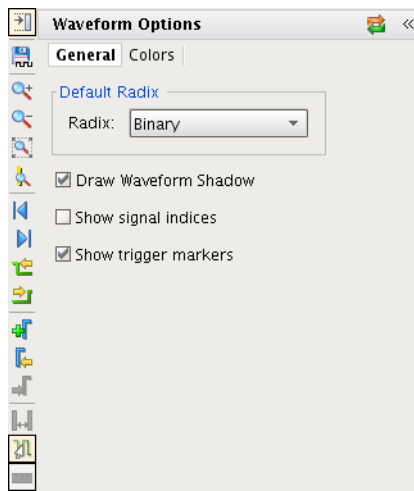


Figure 6-5: Waveform Options Dialog Box

The options are as follows:

- **Colors Tab:** Lets you choose custom colors for waveform objects
- **Default Radix:** Sets the default radix for bus probes
- **Draw Waveform Shadow:** Displays a light green shadow under scalar '1' to help differentiate between '1' and '0'
- **Show signal indices:** Display index position number to the left side of scalar and bus names
- **Show trigger markers:** Show (or hide) the red trigger markers in the wave viewer

Customizing the 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
Cursors	The main cursor and secondary cursor in the Waveform window let you display and measure time, and they form the focal point for various navigation activities.
Markers	You can add markers to navigate through the waveform, and to display the waveform value at a particular time.
Dividers	You can add a divider to create a visual separator of signals.
Using Groups	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.
Using Virtual Buses	You can add a virtual bus to your wave configuration, to which you can add logic scalars and arrays.
Renaming Objects	You can rename objects, signals, buses, and groups.
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 112](#) 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 Markers

The red trigger marker (whose label is a red letter 'T') 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 67](#)).

Note: The trigger markers are not movable using the same technique as regular markers. Set their position 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.

You can also rename Dividers; see [Renaming Objects, page 115](#).

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 115](#).



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.

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 [Bus Radixes, page 117](#) 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 sample 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.

Bus Radixes

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 6-6](#).

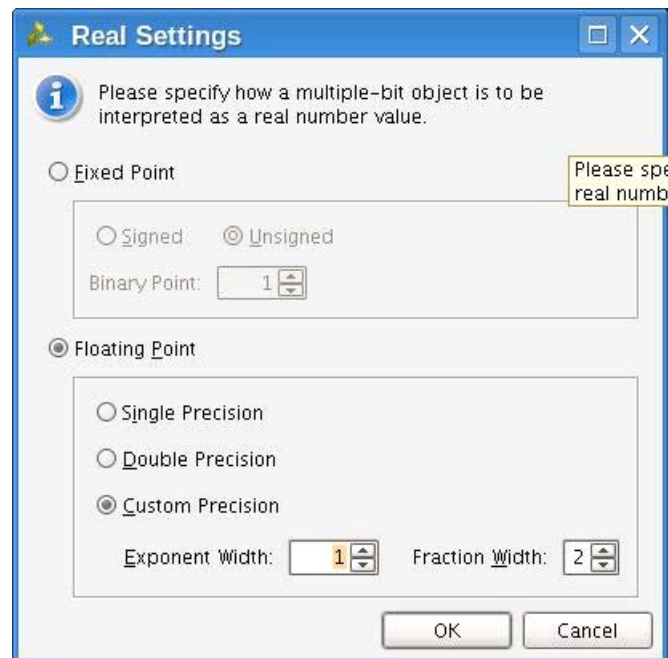


Figure 6-6: Real Settings Dialog Box

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 the bus.
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 6-7](#) shows the **Analog Settings** dialog box with the settings for analog waveform drawing.



Figure 6-7: 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 6-8.

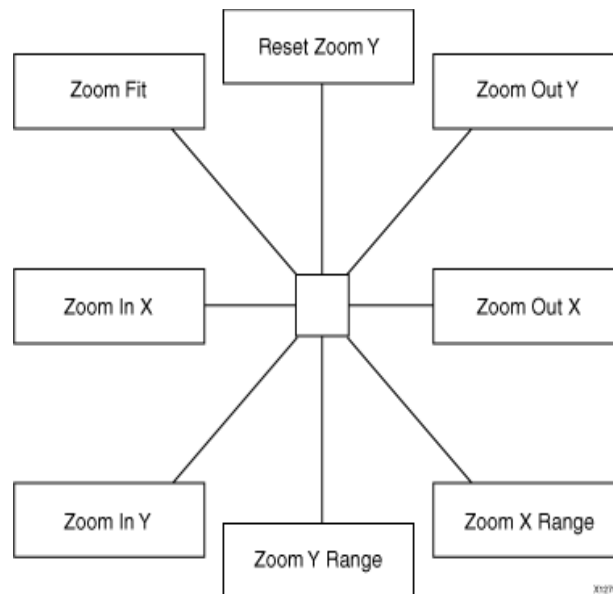


Figure 6-8: Analog Zoom Options

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 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 opens.
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.
5. Double-click 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 13], *LogiCORE IP IBERT for 7 Series GTP Transceivers*, (PG133) [Ref 14], *LogiCORE IP IBERT for 7 Series GTH Transceivers*, (PG152) [Ref 15].

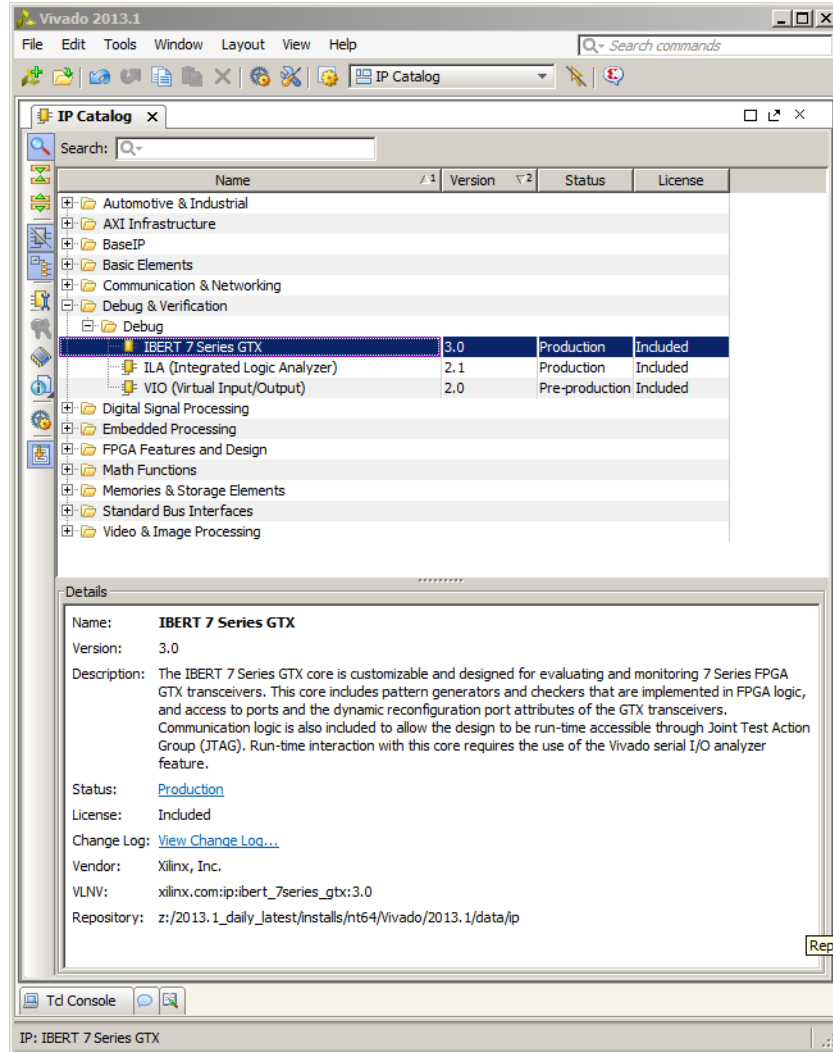


Figure 7-1: IP Catalog Showing the IBERT 7 Series GTX Core

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 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.

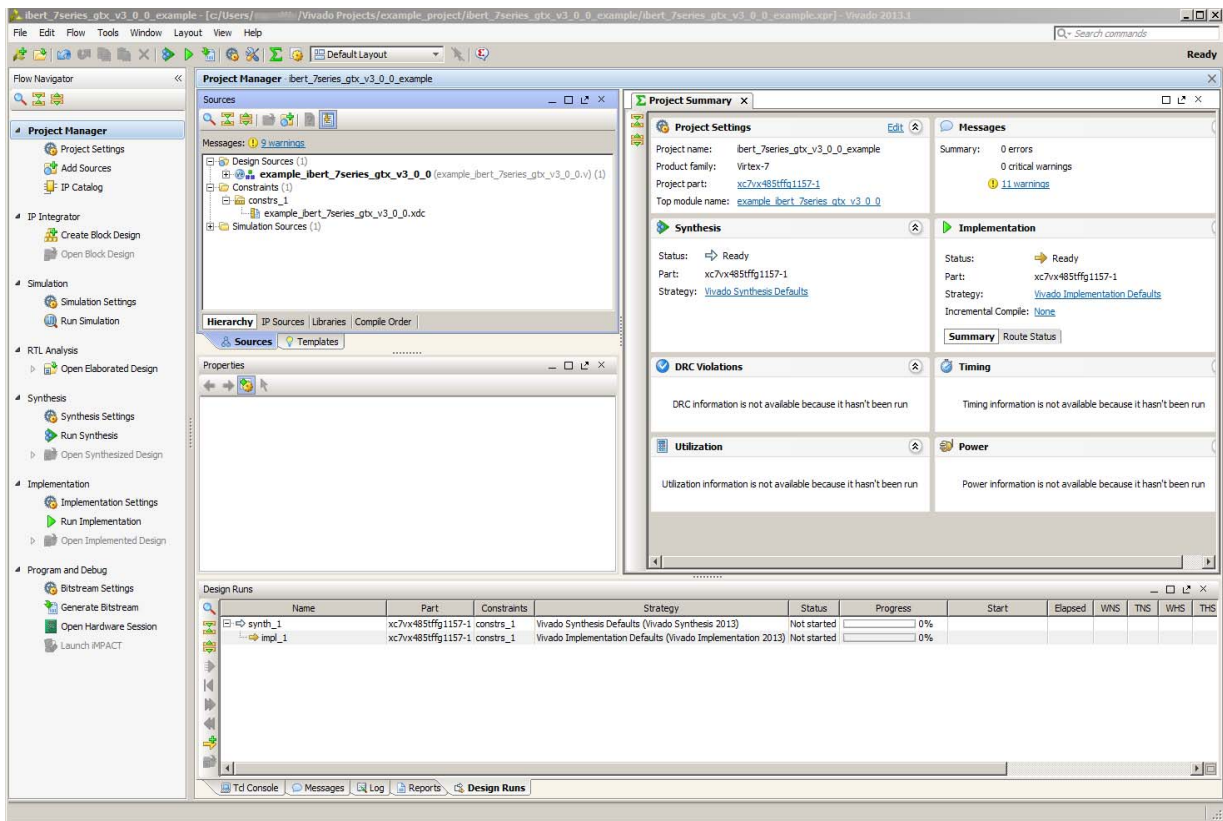


Figure 7-2: IBERT Example Design

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 v3.0 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 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 core, the **Hardware** window now shows the components of the IBERT core that were detected when scanning the device (see [Figure 8-1](#)).

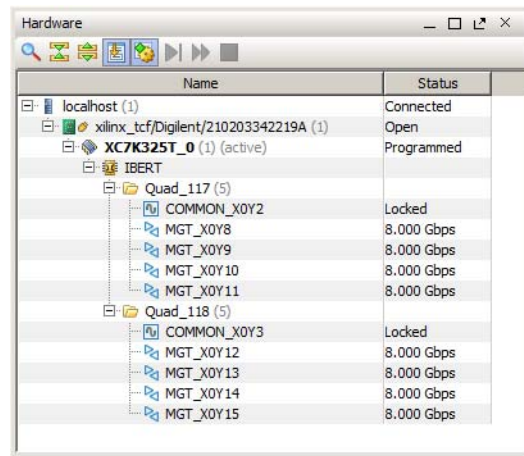


Figure 8-1: Hardware Window Showing the IBERT Core

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 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](#).

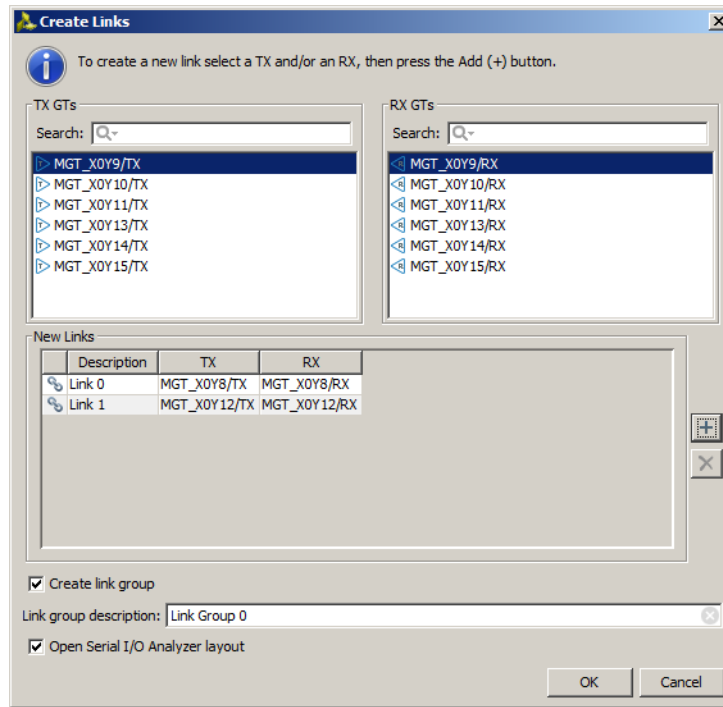


Figure 8-2: Create Links Dialog

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.

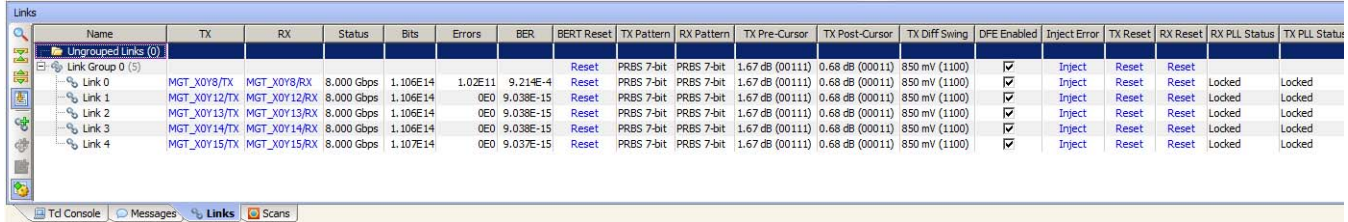


IMPORTANT: A given TX or RX endpoint can only belong to one link.

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	9.214E-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](#).

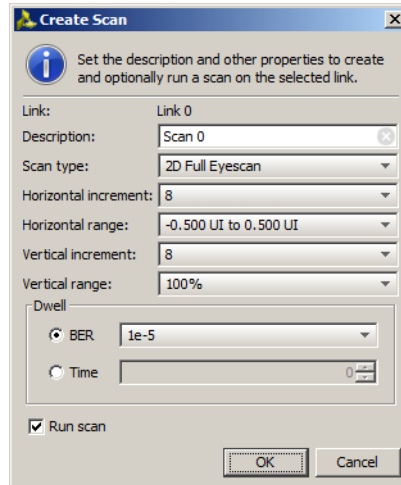


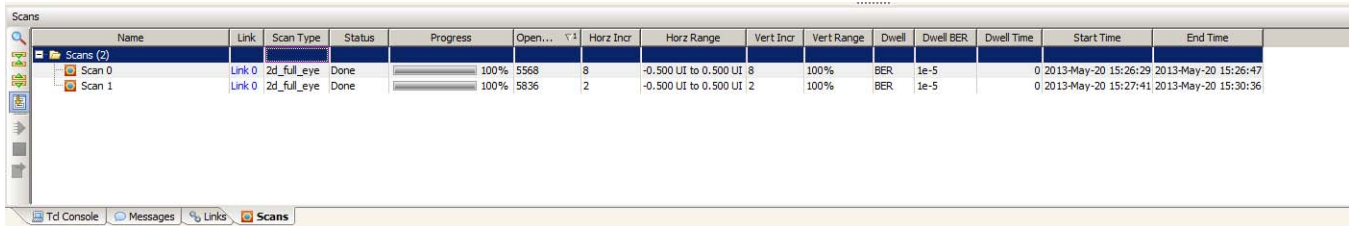
Figure 8-4: Create Scan Dialog

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** check box.

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

Creating and Running Link Sweeps

To analyze the margin of a given link, it can be helpful to run multiple scans of the link with different MGT settings. This helps determine which settings are the best. The Vivado serial I/O analyzer feature enables you to define, run, save, and recall link sweeps, which are a collection of link scans.

A sweep runs on a link. To create a sweep, select a link in the **Link** window, and either right-click and choose **Create Sweep**, or click the **Create Sweep** button in the **Link** window toolbar. This will bring up the **Create Sweep** dialog box, which looks similar to the **Create Scan** dialog box, except that it has additional options for defining which properties to sweep, and how.

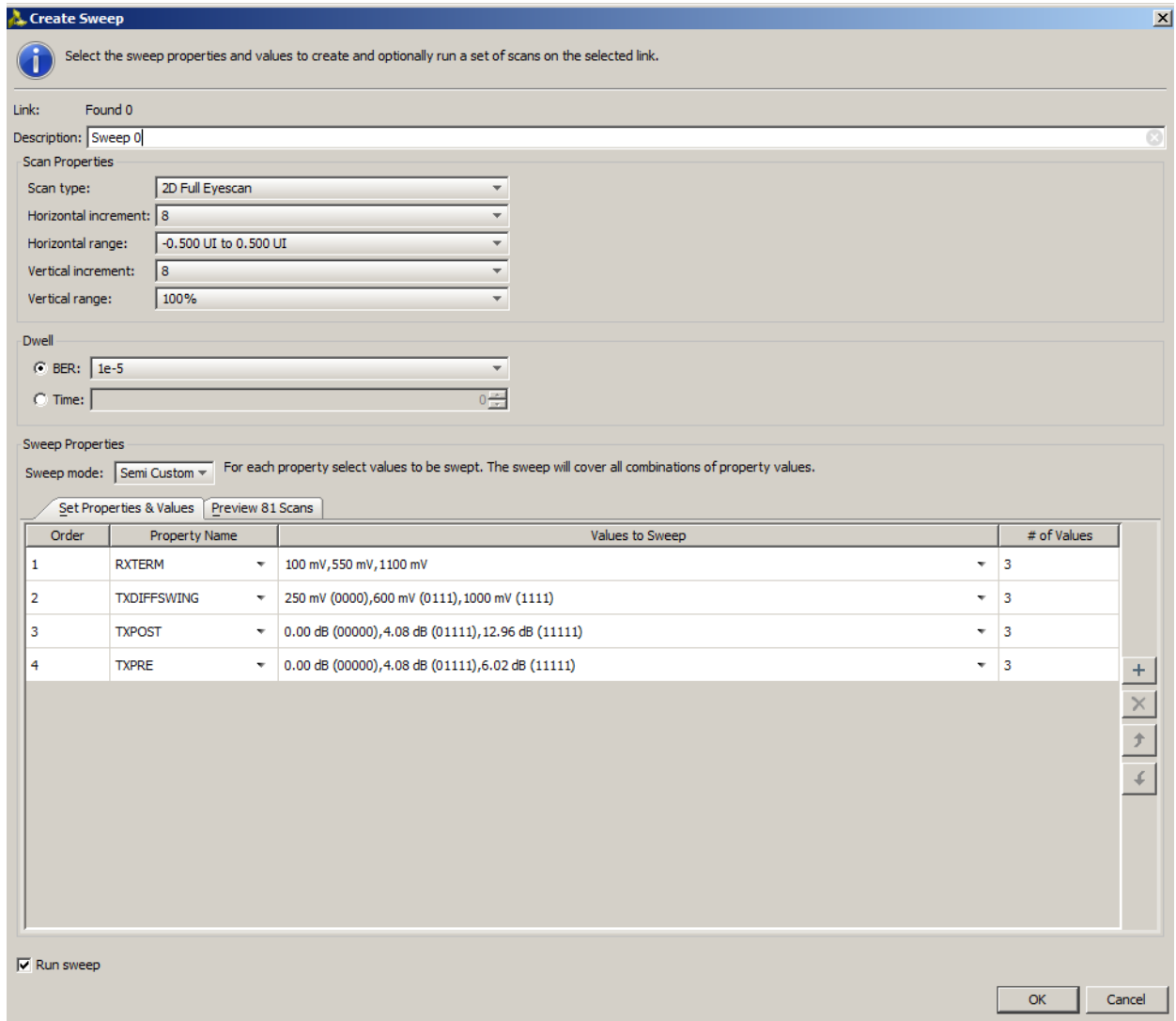


Figure 8-6: Create Sweep Dialog Box

Table 8-3: Sweep Settings

Sweep Setting	Description
Description	A user-defined name for the sweep.
Scan Type	The type of scan to run.
Horizontal Increment	Allows you 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).

Table 8-3: Sweep Settings

Sweep Setting	Description
Vertical Increment	Allows the user 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 (BER).
Dwell Time	Dwell Time allows you to choose the scan depth by inputting the desired time in seconds.
Sweep Mode	The type of sweep to run. The choices are Semi Custom, Full Custom, and Exhaustive.

After these settings are chosen, the next step is to choose the Sweep Properties. Any writable properties of the link can be swept. To add a property, click the + button on the right side to add another row to the table. Click the **Property Name** to choose a property to sweep.

To change the values, click the **Values to Sweep Cell**, and use the chooser to select which values to sweep. If the property does not have enumerated values, type one hex value on each line of the text area provided.

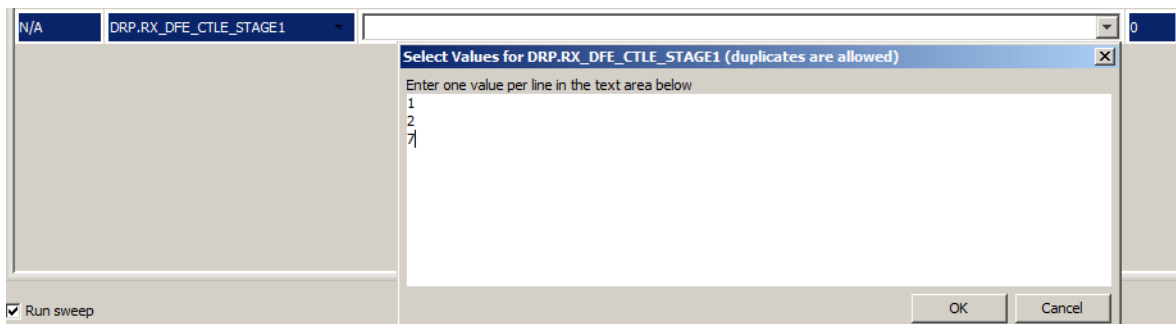


Figure 8-7: Values to Sweep Cell

- In the Semi Custom case shown in Figure 8-8, every combination of the properties chosen is defined for a single scan, and that scan is performed according to the sweep properties. The number of sweeps that are performed, and in what order can be previewed by selecting the **Preview & Scans** tab.
- In the Full Custom case, the first choice for each of the properties listed is used for the first scan, the second choice for each of the properties is used for the second scan, etc. If one of the properties has fewer choices than other properties, the last choice will be used for all subsequent scans. With the same properties choices but Full Custom as the sweep Mode, only three scans would be performed.

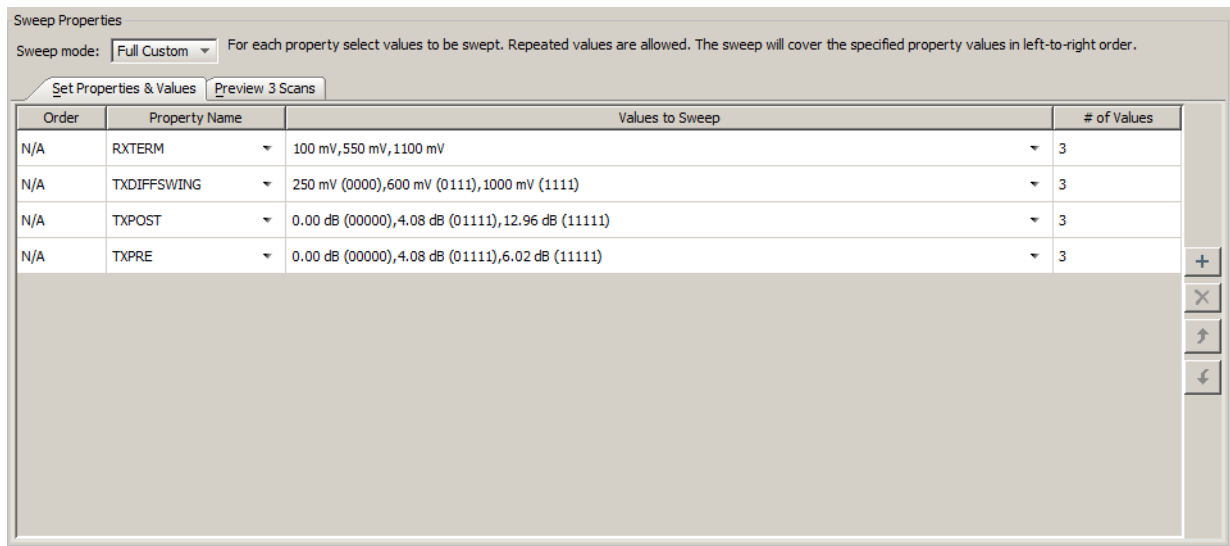


Figure 8-8: Sweep Properties Dialog Box

- In the Exhaustive case, the **Values to Sweep** is no longer editable, as all values are chosen for a given property.

When all the properties are set, to run each of the scans sequentially, keep **Run Sweep** checked. The list of scans is elaborated in the Scan window once you click the **OK** button.

During the sweep, the progress is tracked in the **Scan** window, and the latest Scan result is displayed.

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.

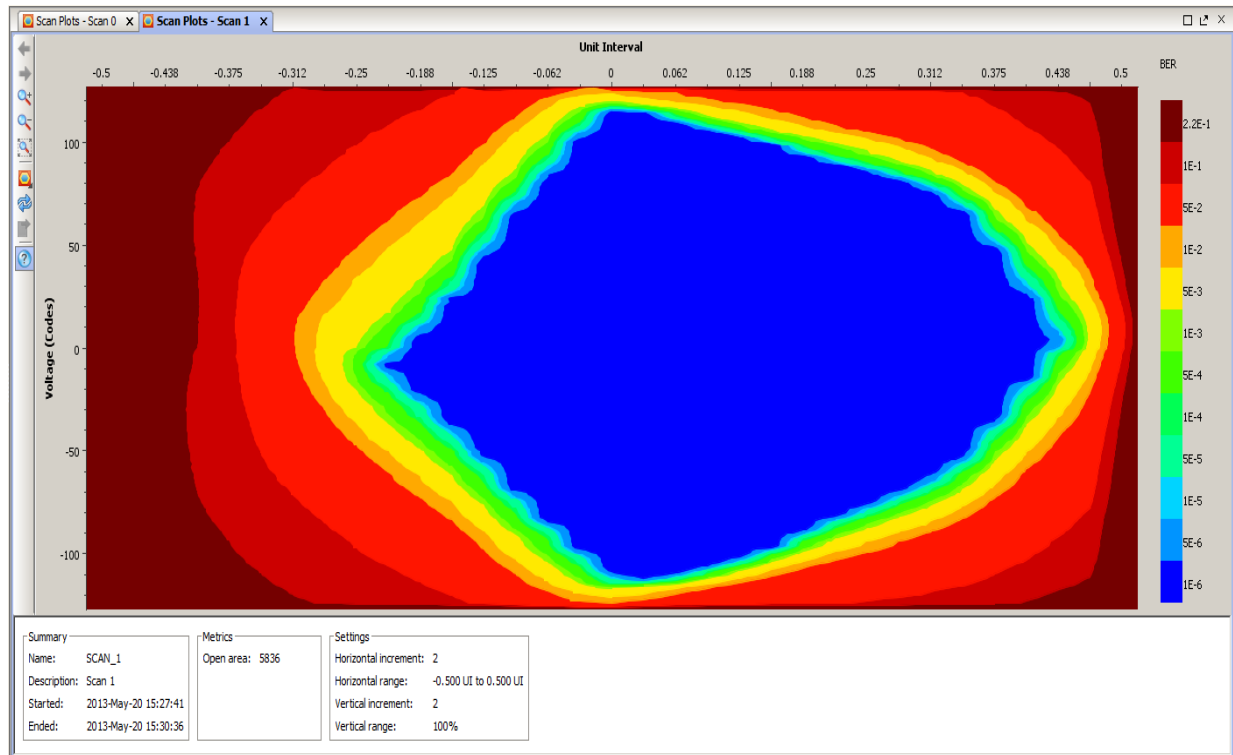


Figure 8-9: Scan Plot Window

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 8-10](#)), or by writing Tcl commands to change and commit the properties. Some properties are read-only and cannot be changed.

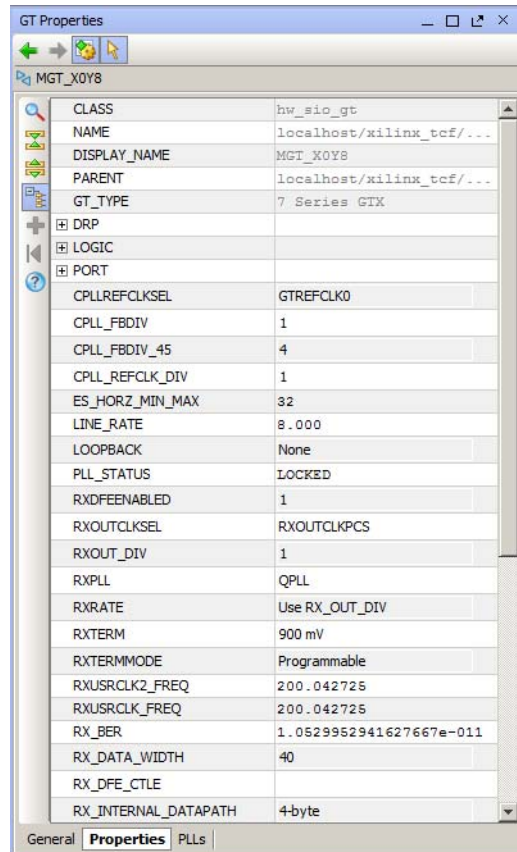


Figure 8-10: Properties Window

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 8-4](#)).

Table 8-4: 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.
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. Note: 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 8-5](#) 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 8-5: 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 8-6 contains descriptions of all Tcl commands used to interact with links.

Table 8-6: 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 8-7 contains descriptions of all Tcl commands used to interact with linkgroups.

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

Table 8-8: 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 8-9 contains descriptions of all Tcl commands used to get serial I/O objects.

Table 8-9: 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.
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 Tcl Commands to Take an IBERT Measurement

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 hw_server running on localhost:3121
- 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:3121
connect_hw_server -url localhost:3121
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"> • Disable (the default) disables the synchronous configuration mode. • Type1 enables the synchronous configuration mode and settings to support the Micron G18(F) family. • Type2 enables the synchronous configuration mode and settings to support the Micron (Numonyx) P30 family.
BITSTREAM.CONFIG.CCLKPIN ^a	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.D00_MOSI ^a	Pullup	Pullup, Pulldown, Pullnone	Adds an internal pull-up, pull-down, or neither to the D00_MOSI pin. Select Pullnone to disable both the pull-up resistor and the pull-down resistor on the D00_MOSI pin. (UltraScale)

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

Settings	Default Value	Possible Values	Description
BITSTREAM.CONFIG.D01_DIN ^a	Pullup	Pullup, Pulldown, Pullnone	Adds an internal pull-up, pull-down, or neither to the D01_DIN pin. Select Pullnone to disable both the pull-up resistor and the pull-down resistor on the D01_DIN pin. (UltraScale)
BITSTREAM.CONFIG.D02 ^a	Pullup	Pullup, Pulldown, Pullnone	Adds an internal pull-up, pull-down, or neither to the D02 pin. Select Pullnone to disable both the pull-up resistor and the pull-down resistor on the D02 pin. (UltraScale)
BITSTREAM.CONFIG.D03 ^a	Pullup	Pullup, Pulldown, Pullnone	Adds an internal pull-up, pull-down, or neither to the D03 pin. Select Pullnone to disable both the pull-up resistor and the pull-down resistor on the D03 pin. (UltraScale)
BITSTREAM.CONFIG.DCIUPDATEMODE	AsRequired	AsRequired, Continuous, Quiet	Controls how often the Digitally Controlled Impedance circuit attempts to update the impedance match for DCI IOSTANDARDS.
BITSTREAM.CONFIG.DONEPIN ^a	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.EXTMASTERCCLK_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 USERCCLK pin.
BITSTREAM.CONFIG.INITPIN ^a	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 ^a	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 ^a	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 ^a	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 .bit file.

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

Settings	Default Value	Possible Values	Description
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. (7 Series, Zynq 7000)
BITSTREAM.CONFIG.OVERTEMPSHUTDOWN	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. (UltraScale)
BITSTREAM.CONFIG.PERSIST	No	No, Yes	Prohibits use of the SelectMAP mode pins for use as user I/O. Refer to the user guide 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 ^a	Pullup	Pullup, Pullnone	Adds an internal pull-up to the PROGRAM_B pin. The Pullnone setting disables the pullup. The pullup affects the pin after configuration. (UltraScale)
BITSTREAM.CONFIG.PUDC_B	Pullup	Pullup, Pulldown, Pullnone	Pull-up During Configuration. (UltraScale)
BITSTREAM.CONFIG.RDWR_B_FCS_B	Pullup	Pullup, Pulldown, Pullnone	Support SPI x1/x2/x4 in bank-0. (UltraScale)
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	Enable	Disable, Enable	Specifies whether the RS[1:0] 3-state is enabled by setting the option in the Warm Boot Start Address (WBSTAR). <ul style="list-style-type: none"> • RS[1:0] pins 3-state enable. • 0: Enable RS 3-state (default) • 1: Disable RS 3-state
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	NONE	NONE, 1, 2, 4	Sets the SPI bus to Dual (x2) or Quad (x4) mode for Master SPI configuration from third party SPI flash devices.

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

Settings	Default Value	Possible Values	Description
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 ^a	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 ^a	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 ^a	Pullup	Pullup, Pulldown, Pullnone	Adds a pull-up, a pull-down, or neither to the TDO 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 ^a	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.
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 dddd MM yy 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.

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

Settings	Default Value	Possible Values	Description
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. Note: 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.
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.

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

Settings	Default Value	Possible Values	Description
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"> • Writes 32 0s to the LOUT register after the synchronization word. • Loads each frame individually. • Performs a Cyclic Redundancy Check (CRC) after each frame. • Writes the frame address to the LOUT register after each frame.
BITSTREAM.GENERAL.DISABLE_JTAG	No	No, Yes	Disables communication to the Boundary Scan (BSCAN) block via JTAG after configuration.
BITSTREAM.GENERAL.JTAG_SYSMON	Enable	Enable, Disable, StatusOnly	Enables or disables the JTAG connection to SYSMON. (UltraScale)
BITSTREAM.GENERAL.JTAG_XADC	Enable	Enable, Disable, StatusOnly	Enables or disables the JTAG connection to the XADC.
BITSTREAM.GENERAL.SYSMONPOWERDOWN	Disable	Disable, Enable	Enables the device to power down SYSMON to save power. Only recommended for permanently powering down SYSMON.(UltraScale)
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. Note: Specifying Security Level1 disables Readback. Specifying Security Level2 disables Readback and Reconfiguration.
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.

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

Settings	Default Value	Possible Values	Description
BITSTREAM.STARTUP.DONEPIPE	Yes	Yes, No	Tells the FPGA device to wait on the CFG_DONE (DONE) pin to go High and then wait for 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 block RAMs 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.
BITSTREAM.STARTUP.MATCH_CYCLE	Auto	Auto, NoWait, 0, 1, 2, 3, 4, 5, 6	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. 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. Note: 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.
BITSTREAM.STARTUP.STARTUPCLK	Cclk	Cclk, UserClk, JtagClk	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. <ul style="list-style-type: none"> • Cclk lets you synchronize to an internal clock provided in the FPGA device. • UserClk lets you synchronize to a user-defined signal connected to the CLK pin of the STARTUP symbol. • JtagClk lets you synchronize to the clock provided by JTAG. This clock sequences the TAP controller which provides the control logic for JTAG.

a. For the dedicated configuration pins Xilinx recommends that you use the bitstream setting default.

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_2;
  else
    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 150

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 150.

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]
```

Configuration Memory Support

This section covers the various Flash device memories that are supported by Vivado® software. Use this section as a guide to select the appropriate configuration memory device for your application by Xilinx device, interface, manufacturer, Flash device, density, and data width.

Artix®-7 Configuration Memory Devices

The Flash devices supported for configuration of Artix-7® devices that can be erased, blank checked, programmed, and verified by Vivado software are shown in [Table C-1](#).

Table C-1: Supported Flash memory devices for Artix-7 device configuration

Interface	Manufacturer	Device (Alias)	Density (Mbits)	Data Widths (bits)
BPI	Spansion	S29GL128S	128	x16
BPI	Spansion	S29GL256S	256	x16
BPI	Spansion	S29GL512S	512	x16
BPI	Spansion	S29GL01GS	1024	x16
BPI	Micron	28F640P30T	64	x16
BPI	Micron	28F640P30B	64	x16
BPI	Micron	28F128P30T	128	x16
BPI	Micron	28F128P30B	128	x16
BPI	Micron	28F256P30T	256	x16
BPI	Micron	28F256P30B	256	x16
BPI	Micron	28F512P30T	512	x16
BPI	Micron	28F512P30E	512	x16
BPI	Micron	28F512P30B	512	x16
BPI	Micron	28F00AP30T	1024	x16
BPI	Micron	28F00AP30E	1024	x16
BPI	Micron	28F00AP30B	1024	x16
BPI	Micron	28F00BP30E ^a	2048	x16
BPI	Micron	28F128G18F	128	x16

Table C-1: Supported Flash memory devices for Artix-7 device configuration

Interface	Manufacturer	Device (Alias)	Density (Mbits)	Data Widths (bits)
BPI	Micron	MT28GU256AAX1E (28F256G18F)	256	x16
BPI	Micron	MT28GU512AAX1E (28F512G18F)	512	x16
BPI	Micron	MT28GU01GAAX1E (28F00AG18F)	1024	x16
BPI	Micron	28F064M29EWH	64	x8, x16
BPI	Micron	28F064M29EWL	64	x8, x16
BPI	Micron	28F064M29EWT	64	x8, x16
BPI	Micron	28F064M29EWB	64	x8, x16
BPI	Micron	28F128M29EW	128	x8, x16
BPI	Micron	28F256M29EW	256	x8, x16
BPI	Micron	28F512M29EW	512	x8, x16
BPI	Micron	28F00AM29EW	1024	x8, x16
BPI	Micron	28F00AM29EW	1048	x8, x16
SPI	Spansion	S25FL032P	32	x8, x16
SPI	Spansion	S25FL132K	32	x1, x2, x4
SPI	Spansion	S25FL064P	64	x1, x2, x4
SPI	Spansion	S25FL164K	64	x1, x2, x4
SPI	Spansion	S25FL128SXXX0	128	x1, x2, x4
SPI	Spansion	S25FL128SXXX1	128	x1, x2, x4
SPI	Spansion	S25FL256SXXX0	256	x1, x2, x4
SPI	Spansion	S25FL256SXXX1	256	x1, x2, x4
SPI	Spansion	S25FL512S	512	x1, x2, x4
SPI	Micron	MT25QU512	512	x1, x2, x4
SPI	Micron	MT25QL512	512	x1, x2, x4
SPI	Micron	N25Q128-3.3V	128	x1, x2, x4
SPI	Micron	N25Q128-1.8V	128	x1, x2, x4
SPI	Micron	N25Q256-3.3V	256	x1, x2, x4
SPI	Micron	N25Q256-1.8V	256	x1, x2, x4
SPI	Micron	N25Q32-3.3V	32	x1, x2, x4
SPI	Micron	N25Q32-1.8V	32	x1, x2, x4

Table C-1: Supported Flash memory devices for Artix-7 device configuration

Interface	Manufacturer	Device (Alias)	Density (Mbits)	Data Widths (bits)
SPI	Micron	N25Q64-3.3V	64	x1, x2, x4
SPI	Micron	N25Q64-1.8V	64	x1, x2, x4

a. For the 28F00BP30E, only lower 1Gb of device is programmable. Select 28F00AP30E to program.

Kintex[®]-7 Configuration Memory Devices

The Flash devices supported for configuration of Kintex-7[®] devices that can be erased, blank checked, programmed, and verified by Vivado[®] software are shown in Table C-2.

Table C-2: Supported Flash memory devices for Kintex-7 device configuration

Interface	Manufacturer	Device (Alias)	Density (Mbits)	Data Widths (bits)
BPI	Spansion	S29GL128P	128	x8, x16
BPI	Spansion	S29GL256P	256	x8, x16
BPI	Spansion	S29GL512P	512	x8, x16
BPI	Spansion	S29GL01GP	1024	x8, x16
BPI	Spansion	S29GL128S	128	x16
BPI	Spansion	S29GL256S	256	x16
BPI	Spansion	S29GL512S	512	x16
BPI	Spansion	S29GL01GS	1024	x16
BPI	Micron	28F640P30T	64	x16
BPI	Micron	28F640P30B	64	x16
BPI	Micron	28F128P30T	128	x16
BPI	Micron	28F128P30B	128	x16
BPI	Micron	28F256P30T	256	x16
BPI	Micron	28F256P30B	256	x16
BPI	Micron	28F512P30T	512	x16
BPI	Micron	28F512P30E	512	x16
BPI	Micron	28F512P30B	512	x16
BPI	Micron	28F00AP30T	1024	x16
BPI	Micron	28F00AP30E	1024	x16
BPI	Micron	28F00AP30B	1024	x16
BPI	Micron	28F00BP30E ^a	2048	x16
BPI	Micron	28F640P33T	64	x16

Table C-2: Supported Flash memory devices for Kintex-7 device configuration

Interface	Manufacturer	Device (Alias)	Density (Mbits)	Data Widths (bits)
BPI	Micron	28F640P33B	64	x16
BPI	Micron	28F128P33T	128	x16
BPI	Micron	28F128P33B	128	x16
BPI	Micron	28F256P33T	256	x16
BPI	Micron	28F256P33B	256	x16
BPI	Micron	28F512P33T	512	x16
BPI	Micron	28F512P33E	512	x16
BPI	Micron	28F512P33B	512	x16
BPI	Micron	28F00AP33T	1024	x16
BPI	Micron	28F00AP33E	1024	x16
BPI	Micron	28F00AP33B	1024	x16
BPI	Micron	28F128G18F	128	x16
BPI	Micron	MT28GU256AAX1E (28F256G18F)	256	x16
BPI	Micron	MT28GU512AAX1E (28F512G18F)	512	x16
BPI	Micron	MT28GU01GAAX1E (28F00AG18F)	1024	x16
BPI	Micron	28F064M29EWH	64	x8, x16
BPI	Micron	28F064M29EWL	64	x8, x16
BPI	Micron	28F064M29EWT	64	x8, x16
BPI	Micron	28F064M29EWB	64	x8, x16
BPI	Micron	28F128M29EW	128	x8, x16
BPI	Micron	28F256M29EW	256	x8, x16
BPI	Micron	28F512M29EW	512	x8, x16
BPI	Micron	28F00AM29EW	1024	x8, x16
BPI	Micron	28F00BM29EW	1048	x8, x16
SPI	Spansion	S25FL032P	32	x1, x2, x4
SPI	Spansion	S25FL064P	64	x1, x2, x4
SPI	Spansion	S25FL164K	64	x1, x2, x4
SPI	Spansion	S25FL128SXXX0	128	x1, x2, x4
SPI	Spansion	S25FL128SXXX1	128	x1, x2, x4
SPI	Spansion	S25FL256SXXX0	256	x1, x2, x4
SPI	Spansion	S25FL256SXXX1	256	x1, x2, x4
SPI	Spansion	S25FL512S	512	x1, x2, x4

Table C-2: Supported Flash memory devices for Kintex-7 device configuration

Interface	Manufacturer	Device (Alias)	Density (Mbits)	Data Widths (bits)
SPI	Micron	MT25QL512	512	x1, x2, x4
SPI	Micron	MT25QU512	512	x1, x2, x4
SPI	Micron	N25Q128-3.3V	128	x1, x2, x4
SPI	Micron	N25Q128-1.8V	128	x1, x2, x4
SPI	Micron	N25Q256-3.3V	256	x1, x2, x4
SPI	Micron	N25Q256-1.8V	256	x1, x2, x4
SPI	Micron	N25Q32-3.3V	32	x1, x2, x4
SPI	Micron	N25Q32-1.8V	32	x1, x2, x4
SPI	Micron	N25Q64-3.3V	64	x1, x2, x4
SPI	Micron	N25Q64-1.8V	64	x1, x2, x4

a. For the 28F00BP30E, only lower 1Gb of device is programmable. Select 28F00AP30E to program.

Virtex[®]-7 Configuration Memory Devices

The Flash devices supported for configuration of Virtex[®]-7 devices that can be erased, blank checked, programmed, and verified by Vivado software are shown in [Table C-3](#).

Table C-3: Supported Flash memory devices for Virtex-7 device configuration

Interface	Manufacturer	Device (Alias)	Density (Mbits)	Data Widths (bits)
BPI	Spansion	S29GL256P	256	x8, x16
BPI	Spansion	S29GL512P	512	x8, x16
BPI	Spansion	S29GL01GP	1024	x8, x16
BPI	Spansion	S29GL256S	256	x16
BPI	Spansion	S29GL512S	512	x16
BPI	Spansion	S29GL01GS	1024	x16
BPI	Micron	28F256P30T	256	x16
BPI	Micron	28F256P30B	256	x16
BPI	Micron	28F512P30T	512	x16
BPI	Micron	28F512P30E	512	x16
BPI	Micron	28F512P30B	512	x16
BPI	Micron	28F00AP30T	1024	x16
BPI	Micron	28F00AP30E	1024	x16
BPI	Micron	28F00AP30B	1024	x16

Table C-3: Supported Flash memory devices for Virtex-7 device configuration

Interface	Manufacturer	Device (Alias)	Density (Mbits)	Data Widths (bits)
BPI	Micron	28F00BP30E ^a	2048	x16
BPI	Micron	28F128G18F	128	x16
BPI	Micron	MT28GU256AAX1E (28F256G18F)	256	x16
BPI	Micron	MT28GU512AAX1E (28F512G18F)	512	x16
BPI	Micron	MT28GU01GAAX1E (28F00AG18F)	1024	x16
BPI	Micron	28F256M29EW	256	x8, x16
BPI	Micron	28F512M29EW	512	x8, x16
BPI	Micron	28F00AM29EW	1024	x8, x16
BPI	Micron	28F00BM29EW	2048	x8, x16
SPI	Spansion	S25FL256SXXX0	256	x1, x2, x4
SPI	Spansion	S25FL256SXXX1	256	x1, x2, x4
SPI	Spansion	S25FL512S	512	x1, x2, x4
SPI	Micron	MT25QU512	512	x1, x2, x4
SPI	Micron	N25Q256-1.8V	256	x1, x2, x4
SPI	Micron	N25Q32-1.8V	32	x1, x2, x4
SPI	Micron	N25Q64-1.8V	64	x1, x2, x4

a. For the 28F00BP30E, only lower 1Gb of device is programmable. Select 28F00AP30E to program.

Kintex® UltraScale™ Configuration Memory Devices

The Flash devices supported for configuration of Kintex® UltraScale™ devices that can be erased, blank checked, programmed, and verified by Vivado software are shown in [Table C-4](#).

Table C-4: Supported Flash memory devices for Kintex UltraScale device configuration

Interface	Manufacturer	Device (Alias)	Density (Mbits)	Data Widths (bits)
BPI	Spansion	S29GL128P	128	x8, x16
BPI	Spansion	S29GL256P	256	x8, x16
BPI	Spansion	S29GL512P	512	x8, x16
BPI	Spansion	S29GL01GP	1024	x8, x16
BPI	Spansion	S29GL128S	128	x16
BPI	Spansion	S29GL256S	256	x16
BPI	Spansion	S29GL512S	512	x16
BPI	Spansion	S29GL01GS	1024	x16
BPI	Micron	28F128P30T	128	x16
BPI	Micron	28F128P30B	128	x16
BPI	Micron	28F256P30T	256	x16
BPI	Micron	28F256P30B	256	x16
BPI	Micron	28F512P30T	512	x16
BPI	Micron	28F512P30E	512	x16
BPI	Micron	28F512P30B	512	x16
BPI	Micron	28F00AP30T	1024	x16
BPI	Micron	28F00AP30E	1024	x16
BPI	Micron	28F00AP30B	1024	x16
BPI	Micron	28F00BP30E ^a	2048	x16
BPI	Micron	28F128P33T	128	x16
BPI	Micron	28F128P33B	128	x16
BPI	Micron	28F256P33T	256	x16
BPI	Micron	28F256P33B	256	x16
BPI	Micron	28F512P33T	512	x16
BPI	Micron	28F512P33E	512	x16
BPI	Micron	28F512P33B	512	x16
BPI	Micron	28F00AP33T	1024	x16
BPI	Micron	28F00AP33E	1024	x16

Table C-4: Supported Flash memory devices for Kintex UltraScale device configuration

Interface	Manufacturer	Device (Alias)	Density (Mbits)	Data Widths (bits)
BPI	Micron	28F00AP33B	1024	x16
BPI	Micron	MT28GU256AAX1E (28F256G18F)	256	x16
BPI	Micron	MT28GU512AAX1E (28F512G18F)	512	x16
BPI	Micron	MT28GU01GAAX1E (28F00AG18F)	1024	x16
BPI	Micron	28F128M29EW	128	x8, x16
BPI	Micron	28F256M29EW	256	x8, x16
BPI	Micron	28F512M29EW	512	x8, x16
BPI	Micron	28F00AM29EW	1024	x8, x16
BPI	Micron	28F00BM29EW	2048	x8, x16
SPI	Spansion	S25FL128SXXX0	128	x1, x2, x4
SPI	Spansion	S25FL128SXXX1	128	x1, x2, x4
SPI	Spansion	S25FL256SXXX0	256	x1, x2, x4
SPI	Spansion	S25FL256SXXX1	256	x1, x2, x4
SPI	Spansion	S25FL512S	512	x1, x2, x4
SPI	Spansion	MT25QL512	512	x1, x2, x4, x8
SPI	Micron	MT25QU512	512	x1, x2, x4
SPI	Micron	N25Q128-1.8V	128	x1, x2, x4, x8
SPI	Micron	N25Q256-1.8V	256	x1, x2, x4, x8
SPI	Micron	N25Q256-1.8V	256	x1, x2, x4

a. For the 28F00BP30E, only lower 1Gb of device is programmable. Select 28F00AP30E to program.

Virtex® UltraScale™ Configuration Memory Devices

The Flash devices supported for configuration of Virtex® UltraScale™ devices that can be erased, blank checked, programmed, and verified by Vivado software are shown in [Table C-5](#).

Table C-5: Supported Flash memory devices for Virtex UltraScale device configuration

Interface	Manufacturer	Device (Alias)	Density (Mbits)	Data Widths (bits)
BPI	Spansion	S29GL256P	256	x8, x16
BPI	Spansion	S29GL512P	512	x8, x16
BPI	Spansion	S29GL01GP	1024	x8, x16
BPI	Spansion	S29GL256S	256	x16
BPI	Spansion	S29GL512S	512	x16
BPI	Spansion	S29GL01GS	1024	x16
BPI	Micron	28F256P30T	256	x16
BPI	Micron	28F256P30B	256	x16
BPI	Micron	28F512P30T	512	x16
BPI	Micron	28F512P30E	512	x16
BPI	Micron	28F512P30B	512	x16
BPI	Micron	28F00AP30T	1024	x16
BPI	Micron	28F00AP30E	1024	x16
BPI	Micron	28F00AP30B	1024	x16
BPI	Micron	28F00BP30E ^a	2048	x16
BPI	Micron	28F256P33T	256	x16
BPI	Micron	28F256P33B	256	x16
BPI	Micron	28F512P33T	512	x16
BPI	Micron	28F512P33E	512	x16
BPI	Micron	28F512P33B	512	x16
BPI	Micron	28F00AP33T	1024	x16
BPI	Micron	28F00AP33E	1024	x16
BPI	Micron	28F00AP33B	1024	x16
BPI	Micron	MT28GU256AAX1E (28F256G18F)	256	x16
BPI	Micron	MT28GU512AAX1E (28F512G18F)	512	x16
BPI	Micron	MT28GU01GAAX1E (28F00AG18F)	1024	x16

Table C-5: Supported Flash memory devices for Virtex UltraScale device configuration

Interface	Manufacturer	Device (Alias)	Density (Mbits)	Data Widths (bits)
BPI	Micron	28F128M29EW	128	x8, x16
BPI	Micron	28F256M29EW	256	x8, x16
BPI	Micron	28F512M29EW	512	x8, x16
BPI	Micron	28F00AM29EW	1024	x8, x16
BPI	Micron	28F00BM29EW	2048	x8, x16
SPI	Spansion	S25FL128SXXX0	128	x1, x2, x4
SPI	Spansion	S25FL128SXXX1	128	x1, x2, x4
SPI	Spansion	S25FL256SXXX0	256	x1, x2, x4
SPI	Spansion	S25FL256SXXX1	256	x1, x2, x4
SPI	Spansion	S25FL512S	512	x1, x2, x4
SPI	Spansion	MT25QL512	512	x1, x2, x4, x8
SPI	Micron	MT25QU512	512	x1, x2, x4
SPI	Micron	N25Q128-1.8V	128	x1, x2, x4, x8
SPI	Micron	N25Q256-1.8V	256	x1, x2, x4, x8
SPI	Micron	N25Q256-1.8V	256	x1, x2, x4

a. For the 28F00BP30E, only lower 1Gb of device is programmable. Select 28F00AP30E to program.

Zynq®-7000 Configuration Memory Devices

The Flash devices supported for configuration of Zynq®-7000 devices that can be erased, blank checked, programmed, and verified by Vivado software are shown in [Table C-6](#).

Table C-6: Supported Flash memory devices for Zynq-7000 device configuration

Interface	Manufacturer	Device	Density (Mbits)	Data Widths (bits)
NOR	Micron	28F032M29EWT	32	x8
NOR	Micron	28F064M29EWT	64	x8
NOR	Micron	28F128M29EWH	128	x8
NOR	Micron	28F256M29EWH	256	x8
NOR	Micron	28F512M29EWH	512	x8
NAND	Micron	MT29F2G08AB	2048	x8
NAND	Micron	MT29F2G16AB	2048	x16
QSPI	Micron	N25Q64	64	x4 (single), x8 (dual parallel)
QSPI	Micron	N25Q128A	128	x4 (single), x4 (dual stacked), x8 (dual parallel)
QSPI	Micron	N25Q128	128	x4 (single), x4 (dual stacked), x8 (dual parallel)
QSPI	Micron	N25Q256	256	x4 (single), x4 (dual stacked), x8 (dual parallel)
QSPI	Micron	N25Q512	512	x4 (single), x4 (dual stacked), x8 (dual parallel)
QSPI	Micron	N25Q00A	1024	x4 (single), x4 (dual stacked), x8 (dual parallel)
QSPI	Spansion	S25FL129P	128	x4 (single), x4 (dual stacked), x8 (dual parallel)
QSPI	Spansion	S25FL128S-3.3V	128	x4 (single), x4 (dual stacked), x8 (dual parallel)
QSPI	Spansion	S25FL128S-1.8V	128	x4 (single), x4 (dual stacked), x8 (dual parallel)

Table C-6: Supported Flash memory devices for Zynq-7000 device configuration

Interface	Manufacturer	Device	Density (Mbits)	Data Widths (bits)
QSPI	Spansion	S25FL256S-3.3V	256	x4 (single), x4 (dual stacked), x8 (dual parallel)
QSPI	Spansion	S25FL256S-1.8V	256	x4 (single), x4 (dual stacked), x8 (dual parallel)
QSPI	Spansion	S25FL512S	512	x4 (single), x4 (dual stacked), x8 (dual parallel)
QSPI	Spansion	S70FL01GS_00	1024	x4 (dual stacked)
QSPI	Winbond	W25Q128	128	x4 (single), x4 (dual stacked), x8 (dual parallel)

Additional Resources and Legal Notices

Xilinx Resources

For support resources such as Answers, Documentation, Downloads, and Forums, see [Xilinx Support](#).

Solution Centers

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

References

These documents provide supplemental material useful with this guide:

[Vivado® Design Suite Documentation](#)

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](#))
6. *Vivado Design Suite User Guide: I/O and Clock Planning* ([UG899](#))
7. *Vivado Design Suite Tcl Command Reference* ([UG835](#))
8. *7 Series FPGAs Configuration User Guide* ([UG470](#))
9. *7 Series FPGAs and Zynq-7000 All Programmable SoC XADC Dual 12-Bit 1 MSPS Analog-to-Digital Converter User Guide* ([UG480](#))

10. *Ultrascale Architecture Configuration: Advance Specification User Guide* ([UG570](#))
11. *UltraScale Architecture System Monitor: Advance Specification User Guide* ([UG580](#)),

Xilinx IP Documentation

12. *LogiCORE IP Virtual Input/Output (VIO) v3.0 Product Guide* ([PG159](#))
13. *LogiCORE IP IBERT for 7 Series GTX Transceivers* ([PG132](#))
14. *LogiCORE IP IBERT for 7 Series GTP Transceivers* ([PG133](#))
15. *LogiCORE IP IBERT for 7 Series GTH Transceivers* ([PG152](#))
16. *LogiCORE IP Integrated Logic Analyzer Product Guide* ([PG172](#))
17. *LogiCORE IP JTAG to AXI Master v1.0 Product Guide* ([PG174](#))

Training Courses

Xilinx provides a variety of training courses and QuickTake videos to help you learn more about the concepts presented in this document. Use these links to explore related training resources:

1. [Vivado Design Suite Hands-on Introductory Workshop Training Course](#)
2. [Vivado Design Suite Tool Flow Training Course](#)
3. [Essentials of FPGA Design Training Course](#)
4. [Vivado Design Suite QuickTake Video: Designing with Vivado IP Integrator](#)
5. [Vivado Design Suite QuickTake Video: Targeting Zynq Devices Using Vivado IP Integrator](#)
6. [Vivado Design Suite QuickTake Video: Partial Reconfiguration in Vivado Design Suite](#)
7. [Vivado Design Suite QuickTake Video: Version Control Overview](#)
8. [Debugging Techniques Using the Vivado Logic Analyzer](#)
9. [Vivado Design Suite QuickTake Video Tutorials](#)

Please Read: Important Legal Notices

The information disclosed to you hereunder (the "Materials") is provided solely for the selection and use of Xilinx products. To the maximum extent permitted by applicable law: (1) Materials are made available "AS IS" and with all faults, Xilinx hereby DISCLAIMS ALL WARRANTIES AND CONDITIONS, EXPRESS, IMPLIED, OR STATUTORY, INCLUDING BUT NOT LIMITED TO WARRANTIES OF MERCHANTABILITY, NON-INFRINGEMENT, OR FITNESS FOR ANY PARTICULAR PURPOSE; and (2) Xilinx shall not be liable (whether in contract or tort, including negligence, or under any other theory of liability) for any loss or damage of any kind or nature related to, arising under, or in connection with, the Materials (including your use of the Materials), including for any direct, indirect, special, incidental, or consequential loss or damage (including loss of data, profits, goodwill, or any type of loss or damage suffered as a result of any action brought by a third party) even if such damage or loss was reasonably foreseeable or Xilinx had been advised

of the possibility of the same. Xilinx assumes no obligation to correct any errors contained in the Materials or to notify you of updates to the Materials or to product specifications. You may not reproduce, modify, distribute, or publicly display the Materials without prior written consent. Certain products are subject to the terms and conditions of Xilinx's limited warranty, please refer to Xilinx's Terms of Sale which can be viewed at <http://www.xilinx.com/legal.htm#tos>; IP cores may be subject to warranty and support terms contained in a license issued to you by Xilinx. Xilinx products are not designed or intended to be fail-safe or for use in any application requiring fail-safe performance; you assume sole risk and liability for use of Xilinx products in such critical applications, please refer to Xilinx's Terms of Sale which can be viewed at <http://www.xilinx.com/legal.htm#tos>.

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