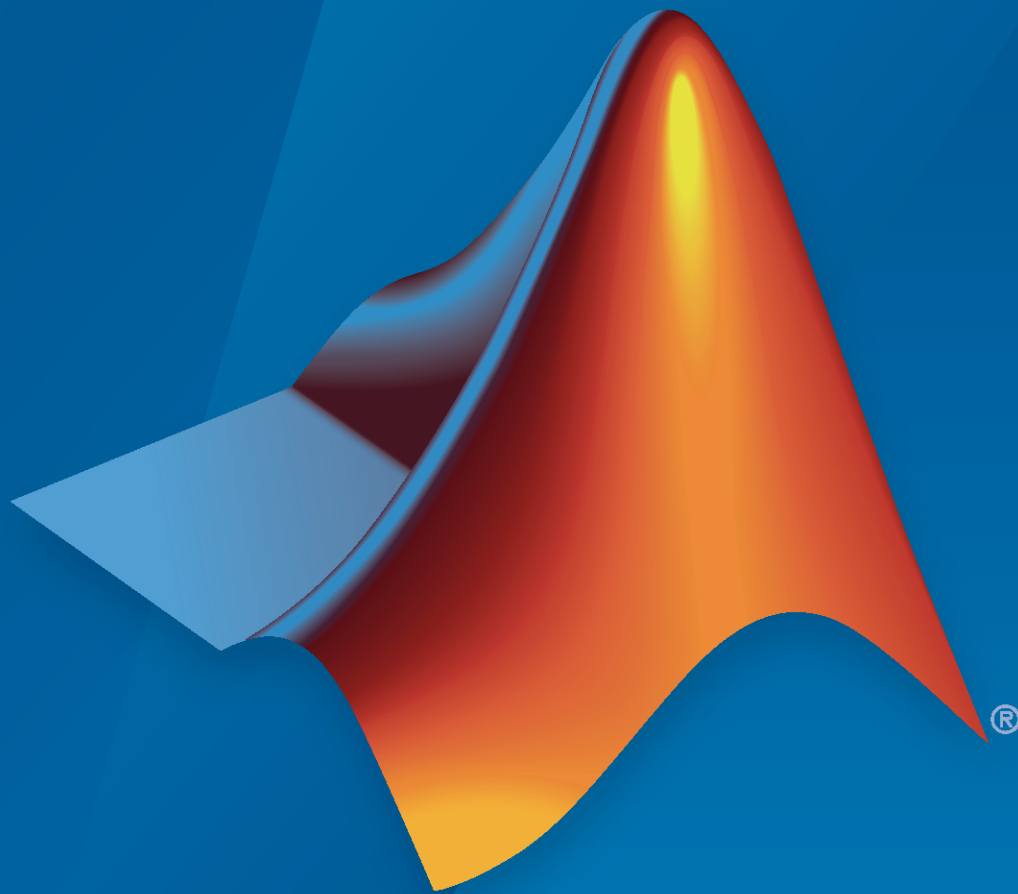


**HDL Verifier™ Support Package for Xilinx®
FPGA Boards**
User's Guide



MATLAB® & SIMULINK®

R2023a



How to Contact MathWorks



Latest news: www.mathworks.com
Sales and services: www.mathworks.com/sales_and_services
User community: www.mathworks.com/matlabcentral
Technical support: www.mathworks.com/support/contact_us



Phone: 508-647-7000



The MathWorks, Inc.
1 Apple Hill Drive
Natick, MA 01760-2098

HDL Verifier™ Support Package for Xilinx® FPGA Boards User's Guide

© COPYRIGHT 2014–2023 by The MathWorks, Inc.

The software described in this document is furnished under a license agreement. The software may be used or copied only under the terms of the license agreement. No part of this manual may be photocopied or reproduced in any form without prior written consent from The MathWorks, Inc.

FEDERAL ACQUISITION: This provision applies to all acquisitions of the Program and Documentation by, for, or through the federal government of the United States. By accepting delivery of the Program or Documentation, the government hereby agrees that this software or documentation qualifies as commercial computer software or commercial computer software documentation as such terms are used or defined in FAR 12.212, DFARS Part 227.72, and DFARS 252.227-7014. Accordingly, the terms and conditions of this Agreement and only those rights specified in this Agreement, shall pertain to and govern the use, modification, reproduction, release, performance, display, and disclosure of the Program and Documentation by the federal government (or other entity acquiring for or through the federal government) and shall supersede any conflicting contractual terms or conditions. If this License fails to meet the government's needs or is inconsistent in any respect with federal procurement law, the government agrees to return the Program and Documentation, unused, to The MathWorks, Inc.

Trademarks

MATLAB and Simulink are registered trademarks of The MathWorks, Inc. See www.mathworks.com/trademarks for a list of additional trademarks. Other product or brand names may be trademarks or registered trademarks of their respective holders.

Patents

MathWorks products are protected by one or more U.S. patents. Please see www.mathworks.com/patents for more information.

Revision History

October 2014	Online only	Revised for Version 14.2.0 (R2014b)
March 2015	Online only	Revised for Version 15.1.0 (R2015a)
September 2015	Online only	Revised for Version 15.2.0 (R2015b)
March 2016	Online only	Revised for Version 16.1.0 (R2016a)
September 2016	Online only	Revised for Version 16.2.0 (R2016b)
March 2017	Online only	Revised for Version 17.1.0 (R2017a)
September 2017	Online only	Revised for Version 17.2.0 (R2017b)
March 2018	Online only	Revised for Version 18.1.0 (R2018a)
September 2018	Online only	Revised for Version 18.2.0 (R2018b)
March 2019	Online only	Revised for Version 19.1.0 (R2019a)
September 2019	Online only	Revised for Version 19.2.0 (R2019b)
March 2020	Online only	Revised for Version 20.1.0 (R2020a)
September 2020	Online only	Revised for Version 20.2.0 (R2020b)
March 2021	Online only	Revised for Version 21.1.0 (R2021a)
September 2021	Online only	Revised for Version 21.2.0 (R2021b)
March 2022	Online only	Revised for Version 22.1.0 (R2022a)
September 2022	Online only	Revised for Version 22.2.0 (R2022b)
March 2023	Online only	Revised for Version 23.1.0 (R2023a)

1	HDL Verifier Support for Xilinx FPGA Boards	
	Xilinx FPGA Board Support from HDL Verifier	1-2
	Supported Xilinx FPGA Boards	1-2
	Supported EDA Tools and Hardware	1-6
	Software	1-6
	Board Connections	1-6
	Download HDL Verifier FPGA Board Support Packages	1-9
	HDL Verifier Support Package for Xilinx FPGA Boards	1-9
	Install with Connection to Internet	1-9
	Install Support Package Offline	1-10
	Customize Xilinx FPGA Board	1-12
	Install Digilent Adept 2 Runtime	1-14
	Download Digilent Adept 2 Runtime Installer	1-14
	Install Digilent Adept 2 Runtime	1-14

	Setup and Configuration	
2	Guided Hardware Setup	2-2
	Select Board and Interface	2-2
	Setup Checklist	2-2
	Setup Steps	2-3
	Configure NIC on Host Computer	2-6
	Select a Drive and Load Firmware	2-7
	Install PCI Express Driver	2-8
	Set Jumper Switches	2-8
	Connect Hardware	2-13
	Verify Setup	2-15
	Open Examples	2-16
	Configure Network Interface Card (NIC) on Development Computer ..	2-17
	Windows	2-17
	Linux	2-17

3

Set Up AXI Manager	3-2
Integrate AXI Manager IP in FPGA Design	3-2
JTAG Considerations	3-3
Use Simulink to Access FPGA Locations	3-4
PCI Express AXI Manager	3-6
PCIe AXI Manager IP	3-6
PCI Express Core	3-7
Ethernet AXI Manager	3-10
Ethernet MAC Hub IP	3-10
UDP AXI Manager IP	3-14
Ethernet AXI Manager for Xilinx Zynq SoC Devices	3-16
Step 1. Complete Hardware Checklist	3-16
Step 2. Configure Host Computer	3-16
Step 3. Copy Image to SD Card in Host System	3-18
Step 4. Update SD Card Image in SoC Device (Optional)	3-19
Step 5: Load Bitstream File to SoC Device (Optional)	3-20
JTAG AXI Manager	3-22
AXI Manager IP	3-22

AXI Manager Simulation

4

AXI Manager Simulation	4-2
HDL Wrapper Creation	4-2
SystemVerilog Test Bench	4-2
writememory(addr,wdata,burst_type) SystemVerilog Task	4-2
readmemory(addr,length,burst_type) SystemVerilog Task	4-3
Memory Mapping Guidelines	4-3

AXI Manager Reference

5

FPGA Data Capture

6

Data Capture Workflow	6-2
Generate and Integrate Data Capture IP Using HDL Workflow Advisor ...	6-3

Configure and Generate IP Core for an Existing HDL Design	6-3
Integrate IP into FPGA	6-4
Capture Data	6-5
Triggers	6-7
What Is a Trigger Condition?	6-7
Sequential Trigger	6-7
Configure a Trigger Condition	6-8
Trigger Position	6-9
Design Considerations for Data Capture	6-11
Signals to Capture	6-11
Capture Timing	6-11
JTAG Considerations	6-11
Ethernet Considerations	6-12
Capture Conditions	6-13
What Is Capture Condition?	6-13
Configure Capture Condition	6-13
Differences Between Triggers and Capture Conditions	6-14

Data Capture Reference

7

HDL Verifier Support Package for Xilinx FPGA Boards Examples

8

Capture Temperature Sensor Data from Xilinx FPGA Board Using FPGA Data Capture	8-2
Access FPGA Memory Using JTAG-Based AXI Manager	8-15
Perform Large Matrix Multiplication on FPGA External DDR Memory Using Ethernet-Based AXI Manager	8-18
Access FPGA Memory Using Ethernet-Based AXI Manager	8-24
Access FPGA External Memory Using AXI Manager over PCI Express ..	8-28
Leverage Built-In Ethernet on Zynq to Perform Memory Access Using AXI Manager	8-31
Access DUT Registers on Xilinx Pure FPGA Board Using IP Core Generation Workflow	8-37
Verify OFDM Transmit and Receive using FPGA Data Capture	8-48

HDL Verifier Support for Xilinx FPGA Boards

- “Xilinx FPGA Board Support from HDL Verifier” on page 1-2
- “Supported EDA Tools and Hardware” on page 1-6
- “Download HDL Verifier FPGA Board Support Packages” on page 1-9
- “Customize Xilinx FPGA Board” on page 1-12
- “Install Digilent Adept 2 Runtime” on page 1-14

Xilinx FPGA Board Support from HDL Verifier

HDL Verifier automates the verification of HDL code on FPGA boards by providing connections between your FPGA board and your simulations in Simulink® or MATLAB®.

- FPGA-in-the-loop (FIL) enables you to run a Simulink or MATLAB simulation that is synchronized with an HDL design running on an FPGA board.
- FPGA data capture is a way to observe signals from your design while the design is running on the FPGA. It captures a window of signal data from the FPGA, based on your configuration and trigger settings, and returns the data to MATLAB or Simulink.
- AXI manager provides access to live on-board memory locations from Simulink or MATLAB. You must include the AXI manager IP in your FPGA design.

To use each of these features, you must have a supported FPGA board connected to your MATLAB host computer using a supported connection type, and a supported synthesis tool.

Supported Xilinx FPGA Boards

This support package enables FIL simulation, FPGA data capture, and AXI manager for the boards in the table.

FPGA data capture and AXI manager are supported for Xilinx devices using Vivado® projects. Xilinx ISE projects are not supported.

Note

- AXI manager and FPGA data capture in HDL Workflow Advisor support programmable logic (PL) Ethernet only. Processing system (PS) Ethernet is not supported.
- FPGA data capture in HDL Workflow Advisor supports GMII and MII interfaces. SGMII interface is not supported.

Device Family	Board	Ethernet			JTAG			PCI Express			Comments
		FIL	FPGA Data Capture	AXI Manager	FIL	FPGA Data Capture	AXI Manager	FIL ^a	FPGA Data Capture	AXI Manager	
Xilinx Artix®-7	Digilent® Nexys™ 4 Artix-7	x			x	x	x				
	Digilent Arty Board	x	x	x	x	x	x				
Xilinx Kintex®-7	Kintex-7 KC705	x	x	x	x	x	x	x		x	

Device Family	Board	Ethernet			JTAG			PCI Express			Comments
		FIL	FPGA Data Capture	AXI Manager	FIL	FPGA Data Capture	AXI Manager	FIL ^a	FPGA Data Capture	AXI Manager	
Xilinx Kintex UltraScale™	Kintex UltraScale FPGA KCU105 Evaluation Kit	x	x	x	x	x	x			x	
Xilinx Kintex UltraScale+™	Kintex UltraScale+ FPGA KCU116 Evaluation Kit		x	x		x	x			x	For more information, see “PCI Express AXI Manager” on page 3-6.
Xilinx Spartan®-6	Spartan-6 SP605	x									
	Spartan-6 SP601	x									
	XUP Atlys Spartan-6	x									
Xilinx Spartan-7	Digilent Arty S7-25				x	x	x				
Xilinx Virtex® UltraScale	Virtex UltraScale FPGA VCU108 Evaluation Kit	x	x	x	x	x	x			x	
Xilinx Virtex UltraScale+	Virtex UltraScale+ FPGA VCU118 Evaluation Kit		x	x	x	x	x	x		x	
Xilinx Virtex-7	Virtex-7 VC707	x	x	x	x	x	x	x		x	
	Virtex-7 VC709				x	x	x	x		x	
Xilinx Virtex-6	Virtex-6 ML605	x									
Xilinx Virtex-5	Virtex ML505	x									
	Virtex ML506	x									
	Virtex ML507	x									
	Virtex XUPV5-LX110T	x									
Xilinx Virtex-4	Virtex ML401	x									Note Support for Virtex-4 device family will be
	Virtex ML402	x									

Device Family	Board	Ethernet			JTAG			PCI Express			Comments
		FIL	FPG A Data Capture	AXI Manager	FIL	FPG A Data Capture	AXI Manager	FIL ^a	FPG A Data Capture	AXI Manager	
	Virtex ML403	x									removed in a future release.
Xilinx Zynq®	Zynq-7000 ZC702	x		x	x	x	x				This board supports PS Ethernet.
	Zynq-7000 ZC706	x		x	x	x	x				This board supports PS Ethernet.
	ZedBoard™	x		x	x	x	x				Use the USB port marked "PROG" for programming. This board supports PS Ethernet.
	ZYBO™ Zynq-7000 Development Board				x	x	x				
	PicoZed™ SDR Development Kit				x	x	x				
	MiniZed™					x	x				Supported only for FPGA data capture and AXI manager via FTDI JTAG.
Xilinx Zynq UltraScale+	Zynq UltraScale+ MPSoC ZCU102 Evaluation Kit	x		x	x	x	x				This board supports PS Ethernet.
	Zynq UltraScale+ MPSoC ZCU104 Evaluation Kit				x	x	x				
	Zynq UltraScale+ MPSoC ZCU106 Evaluation Kit				x	x	x				
	Zynq UltraScale+ RFSoc ZCU111 Evaluation Kit	x		x	x	x	x				This board supports PS Ethernet.

Device Family	Board	Ethernet			JTAG			PCI Express			Comments
		FIL	FPG A Data Capture	AXI Manager	FIL	FPG A Data Capture	AXI Manager	FIL ^a	FPG A Data Capture	AXI Manager	
	Zynq UltraScale + RFSoc ZCU216 Evaluation Kit	x		x	x	x	x				This board supports PS Ethernet.
Xilinx Versal [®]	Versal AI Core Series VCK190 Evaluation Kit	x			x	x	x				

^a FIL over PCI Express[®] connection is supported only for 64-bit Windows[®] operating systems.

See Also

More About

- “FPGA-in-the-Loop Simulation”
- “FPGA-in-the-Loop Simulation Workflows”
- “Data Capture Workflow” on page 6-2
- “Set Up AXI Manager” on page 3-2

Supported EDA Tools and Hardware

Software

Xilinx Vivado and ISE

Use this support package with these recommended versions:

- Xilinx Vivado 2022.1
- Xilinx ISE 14.7
 - Xilinx ISE is not supported for FPGA data capture or AXI manager.
 - Xilinx ISE is required for FPGA boards in the Spartan-6, Virtex-4, Virtex-5, and Virtex-6 families.

For tool setup instructions, see “Set Up FPGA Design Software Tools”.

Board Connections

JTAG Connection

You can run FPGA-in-the-loop, FPGA data capture, or AXI manager over a JTAG cable to your board. However, each feature requires exclusive use of the JTAG cable, so you cannot run more than one feature at the same time. To allow other tools access to the JTAG cable, such as programming the FPGA, and Xilinx ChipScope, you must discontinue the JTAG connection in MATLAB. To release the JTAG cable:

- FPGA-in-the-loop — Close the Simulink model, or call the `release` method of the System object™.
- FPGA data capture — Close the FPGA Data Capture tool, release the System object, or close the Simulink model.
- AXI manager — Call the `release` method of the object.

However, the nonblocking capture mode enables you to simultaneously use FPGA data capture and AXI manager, which share a common JTAG interface. For more information, see the "Simultaneous Use of FPGA Data Capture and AXI Manager" section of “JTAG Considerations” on page 6-11.

For Xilinx boards, the JTAG clock frequency is 33 or 66 MHz. The JTAG frequency depends on the type of cable and the maximum clock frequency supported by the FPGA board.

Required Hardware	Required Software
Digilent download cable. <ul style="list-style-type: none"> • If your board has an onboard Digilent USB-JTAG module, use a USB cable. • If your board has a standard Xilinx 14 pin JTAG connector, you can obtain an HS2 or HS3 cable from Digilent. 	<ul style="list-style-type: none"> • For Windows operating systems: Xilinx Vivado executable directory must be on system path. • For Linux® operating systems: Digilent Adept 2. For the installation steps, see “Install Digilent Adept 2 Runtime” on page 1-14.

Required Hardware	Required Software
FTDI USB-JTAG cable <ul style="list-style-type: none"> Supported for boards with onboard FT4232H, FT232H, or FT2232H devices implementing USB-to JTAG 	Install these D2XX drivers. <ul style="list-style-type: none"> For Windows operating systems: 2.12.28 (64 bit) For Linux operating systems: 1.4.22 (64 bit) For the installation guide, see D2XX Drivers from the FTDI Chip website.

Note When simulating your FPGA design through Digilent JTAG cable with Simulink or MATLAB, you cannot use any debugging software that requires access to the JTAG; for example, Vivado Logic Analyzer.

Ethernet Connection

You can run FPGA-in-the-loop, FPGA data capture, or AXI manager over an Ethernet connection. To use FPGA data capture and AXI manager over an Ethernet connection in a single HDL project, connect the FPGA data capture and AXI manager IPs to the same Ethernet MAC Hub IP using different port addresses.

Required Hardware	Supported Interfaces	Required Software
<ul style="list-style-type: none"> Gigabit Ethernet card Cross-over Ethernet cable FPGA board with supported Ethernet connection 	<ul style="list-style-type: none"> Gigabit Ethernet — GMII Gigabit Ethernet — RGMII Gigabit Ethernet — SGMII Ethernet — MII Ethernet — RMII 	There are no software requirements for an Ethernet connection, but ensure that the firewall on the host computer does not prevent UDP communication.

Note

- FPGA data capture and AXI manager support GMII, MII, and SGMII interfaces only.
- RMII is supported with Vivado versions older than 2019.2.
- Ethernet connection to Virtex-7 VC707 not supported for Vivado versions older than 2013.4.

PCI Express

FPGA-in-the-loop over a PCI Express connection is supported only for 64-bit Windows operating systems.

AXI manager is supported over PCI Express for Xilinx Kintex UltraScale+ FPGA KCU116 Evaluation Kit boards.

Board	Required Software
<ul style="list-style-type: none">• Kintex-7 KC705 Evaluation Kit• Virtex -7 VC707 Evaluation Kit• Xilinx Virtex-7 VC709 Evaluation Board• Virtex UltraScale+ FPGA VCU118 Evaluation Kit	Vivado 2017.4 or newer.

See Also

More About

- “FPGA-in-the-Loop Simulation”
- “FPGA-in-the-Loop Simulation Workflows”
- “Data Capture Workflow” on page 6-2
- “Set Up AXI Manager” on page 3-2

Download HDL Verifier FPGA Board Support Packages

In this section...

“HDL Verifier Support Package for Xilinx FPGA Boards” on page 1-9

“Install with Connection to Internet” on page 1-9

“Install Support Package Offline” on page 1-10

HDL Verifier Support Package for Xilinx FPGA Boards

The support package for Xilinx FPGA boards contains the board definition files for FPGA-in-the-Loop (FIL) simulation with HDL Verifier and supported Xilinx hardware. To perform FIL simulation with Xilinx FPGA boards, first download the Xilinx FPGA board support package.

To install support packages:

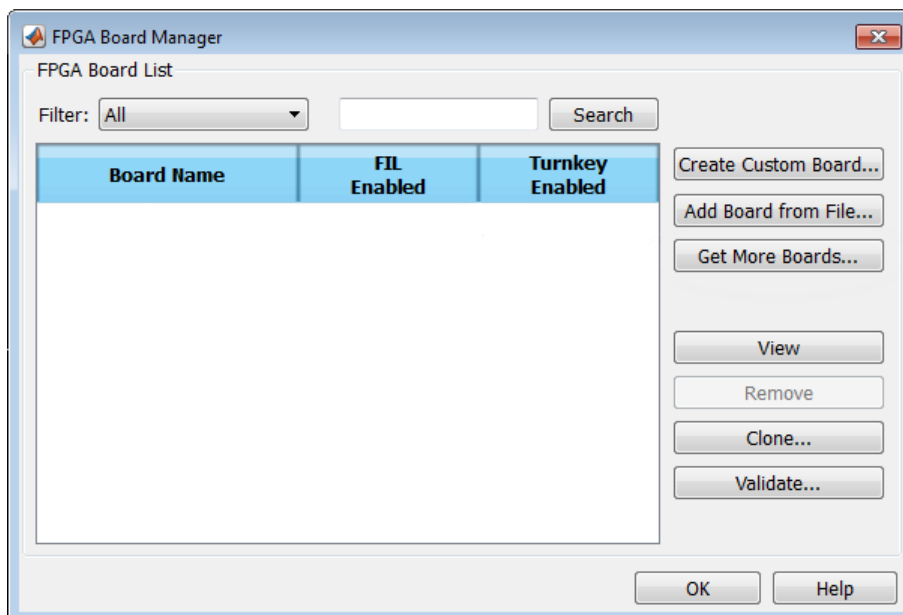
- On the MATLAB **Home** tab, in the **Environment** section, click **Add-Ons > Get Hardware Support Packages**.

You can also download FPGA board support packages from within the FPGA-in-the-Loop Wizard or the FPGA Board Manager.

Install with Connection to Internet

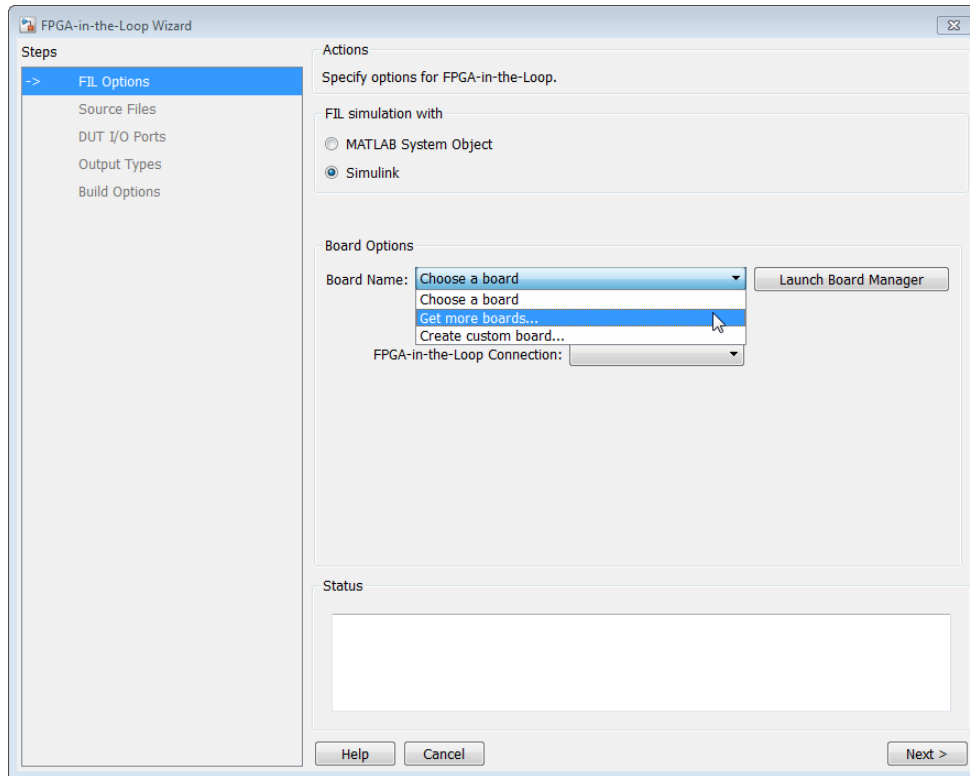
From the FPGA Board Manager

- 1 In the MATLAB command window, enter the following command:
`fpgaBoardManager`
- 2 In the FPGA Board Manager dialog box, click **Get More Boards**.



From the FIL Wizard

- 1 In the MATLAB command window, enter the following command:
`filWizard`
- 2 In the **FIL Options** pane, at **Board Name**, select **Get more boards** from the drop-down menu.



Install Support Package Offline

To install the support packages without an internet connection, first download the packages on a computer that *does* have an internet connection.

- 1 On the computer with the internet connection, start MATLAB.
- 2 On the MATLAB **Home** tab, in the **Environment** section, click **Add-Ons > Get Hardware Support Packages**.
- 3 Select your desired support package, and use the **Install** button pull-down menu to select **Download Only...**
- 4 Accept the license and select a folder for the download.
- 5 Copy the entire downloaded folder, for example, the R2016b folder, to a shared network drive or removable media, such as a USB drive.

Then, on the computer where you want to install the support packages:

- 1 Copy the downloaded folder to the host computer.
- 2 To start the installer, run the `install_supportsoftware.exe` executable file.

- 3 Follow the installer prompts to install the support package. If you do actually have an internet connection, you are prompted to log in to your MathWorks® account.

See Also

Related Examples

- “Block Generation with the FIL Wizard”
- “System Object Generation with the FIL Wizard”
- “Customize Xilinx FPGA Board” on page 1-12

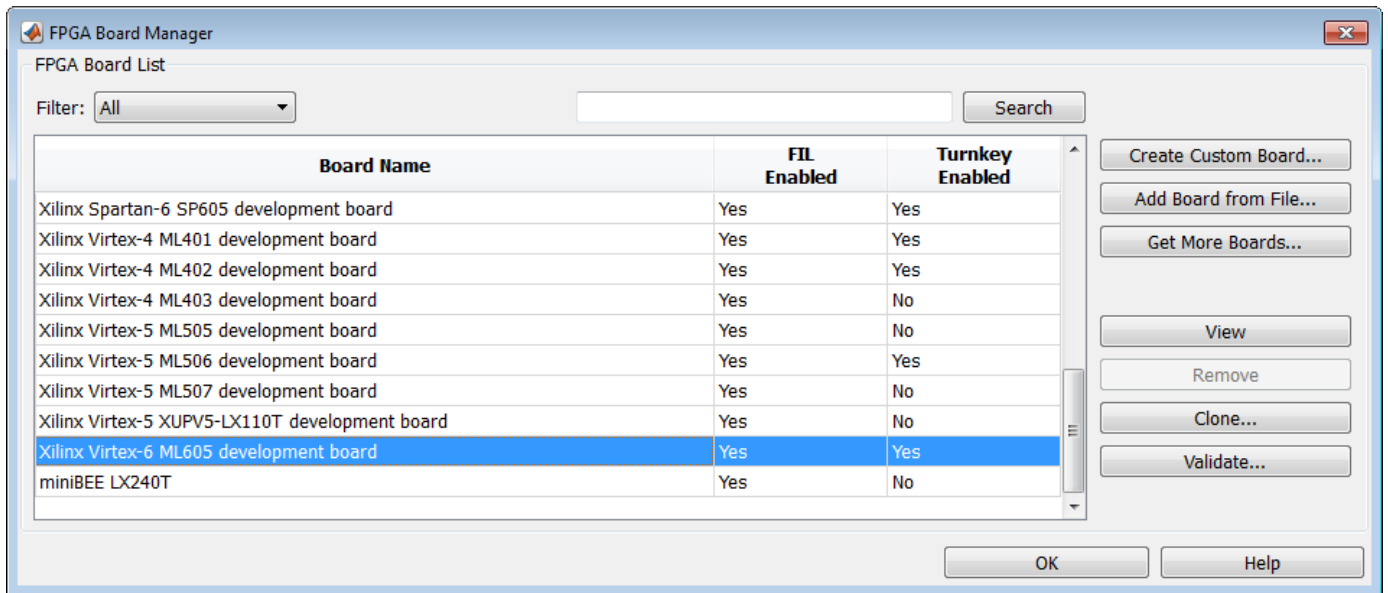
Customize Xilinx FPGA Board

You can change part of the board definition file for a Xilinx FPGA board to customize it. For example, you want to change the default interface for the ML605 to MII. However, you cannot change the board definition file directly. Instead, make a copy of the file and then change the copied file.

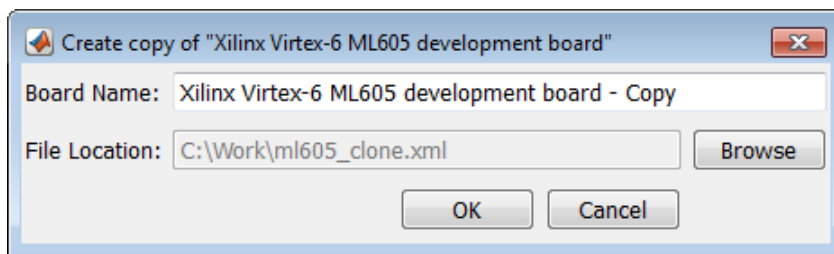
Create a copy of the board definition file using the FPGA Board Manager. Follow these steps:

- 1 Start the FPGA Board Manager by typing the following at the MATLAB command prompt:

```
fpgaBoardManager
```
- 2 In the FPGA Board List, select the board you want to copy and click **Clone**.

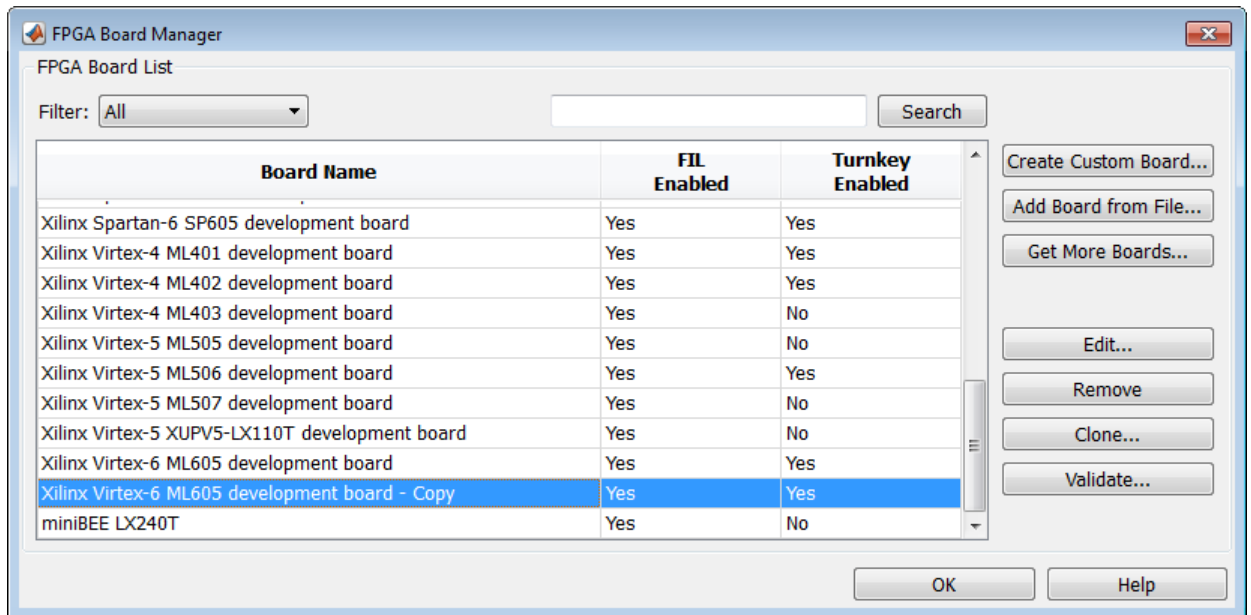


- 3 In the Create copy dialog box, specify the file name and location and click **OK**. Choose a new file name (and, optionally, a location) so that the original definition file is preserved.



- 4 Specify FPGA board information for the copy to customize it. By default, the copy inherits its information from the original, so you have to change only those fields that differ.
- 5 Click **OK**.

The copy of the original board now appears in the FPGA Board List. It also appears on the Board Name list in the FPGA-in-the-Loop Wizard and you can use the board with FIL simulation.



See Also

Related Examples

- “FPGA Board Customization”
- “FPGA Board Manager”

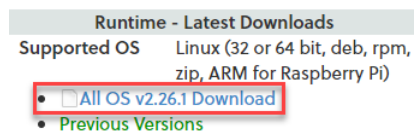
Install Digilent Adept 2 Runtime

Supported Platforms: Linux

Digilent Adept 2 Runtime contains the libraries needed to use the Digilent download cable on Linux operating systems.

Download Digilent Adept 2 Runtime Installer

Download the latest version of the Digilent Adept 2 Runtime installer from the Digilent website at Adept 2 - Digilent Reference. Click on the installer link, follow the instructions, and set **Operating System** to Linux 64-bit Zip. The installer is in the TAR.GZ format.



Install Digilent Adept 2 Runtime

Install Digilent Adept 2 Runtime by entering the following commands in the terminal.

- 1 Switch to the root user.

```
sudo su
```

Enter the password for the root user.

- 2 Create the /tmp/digilent_install installation directory.

```
root@user:/home/user# mkdir -p /tmp/digilent_install
```

- 3 Copy the downloaded Digilent Adept 2 Runtime installer to the directory you created for the purpose of installation. For example, if you downloaded the Runtime version 2.26.1, copy the `digilent.adept.runtime_2.26.1-x86_64.tar.gz` file to the `/tmp/digilent_install` directory.

Note The version part of the installer file name (2.26.1) changes from one version to the next.

- 4 Change the directory to /tmp/digilent_install.

```
root@user:/home/user# cd /tmp/digilent_install
```

- 5 Make the tar file executable. For example, if you are making the `digilent.adept.runtime_2.26.1-x86_64.tar.gz` file executable, in the terminal, type:

```
root@user:/tmp/digilent_install# chmod +x \
> digilent.adept.runtime_2.26.1-x86_64.tar.gz
```

- 6 Decompress the installer. For example, if you are decompressing the `digilent.adept.runtime_2.26.1-x86_64.tar.gz` file, in the terminal, type:

```
root@user:/tmp/digilent_install# tar -xvzf \
> digilent.adept.runtime_2.26.1-x86_64.tar.gz
```

- 7 Change the directory to the one whose name starts with `digilent.adept.runtime`. For example, if you are installing Digilent Adept 2 Runtime version 2.26.1, in the terminal, type:

```
root@user:/tmp/digilent_install# cd \  
> digilent.adept.runtime_2.26.1-x86_64
```

- 8** Make the install script executable. For example, if you are installing Digilent Adept 2 Runtime version 2.26.1, in the terminal, type:

```
root@user:/tmp/digilent_install/digilent.adept.runtime_2.26.1-x86_64# \  
> chmod +x install.sh
```

- 9** Run the install script. For example, if you are installing Digilent Adept 2 Runtime version 2.26.1, in the terminal, type:

```
root@user:/tmp/digilent_install/digilent.adept.runtime_2.26.1-x86_64# \  
> ./install.sh
```

- 10** The Digilent Adept 2 Runtime installer displays the log information in the terminal. The installer asks you in which directory to install the libraries, system binaries, data files, and Adept Runtime Configuration file. Select the default directories by pressing the **Enter** key. To confirm the successful installation of the libraries on the `/usr/lib64` path, in the terminal, type:

```
# ls -l /usr/lib64/digilent/adept
```

See Also

External Websites

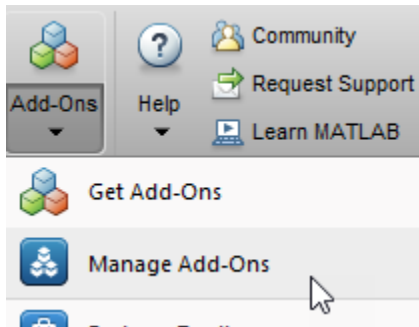
- Adept 2 - Digilent Reference


Setup and Configuration

Guided Hardware Setup

Before you can use the features in the HDL Verifier Support Package for Xilinx FPGA Boards, you must establish communication between the host and the hardware board. After the installer completes the support package installation, it guides you through the process of establishing communication with the hardware board.

If the support package is already installed, you can start the hardware setup by opening the Add-On Manager.



In the Add-On Manager, start the hardware setup process by clicking the **Gear** icon .

The setup process includes these steps:

- Specify a hardware board and interface.
- Configure the network interface card in the host computer (for the Ethernet interface only).
- Copy or transfer the compatible SD card image files for the hardware board to an SD card drive path (for the Ethernet interface on Zynq SoC boards only).
- Configure your hardware board to start up from the SD card (for the Ethernet interface on Zynq SoC boards only).
- Install the PCIe driver on the host computer (for the PCI Express interface only).
- Connect your hardware board to the host computer (for the Ethernet interface on Zynq SoC boards only).
- Verify the connection between the host computer and the hardware board.

Select Board and Interface

Choose a hardware board and an interface to use with this board from the list. For the full list of supported boards and interfaces, see “Supported Xilinx FPGA Boards” on page 1-2. HDL Verifier supports a PCI Express connection for FPGA-in-the-loop (FIL) with Windows operating systems only.

Setup Checklist

The guided setup wizard displays a checklist of the hardware requirements. Confirm that you have the hardware required to complete the setup process.

Note Do not connect to the board or turn it on until you are prompted at a later step.

Ethernet Requirements

- FPGA development board
- USB-JTAG cable
- Installed Vivado software
- Dedicated Gigabit network interface card (NIC) or USB 3.0 Gigabit Ethernet adapter dongle
- Ethernet cable
- Power supply adapter (if the board requires one)

Ethernet on Zynq SoC or Versal Board Requirements

- Zynq SoC or Versal board
- Dedicated Gigabit NIC or USB 3.0 Gigabit Ethernet adapter dongle
- Ethernet cable
- Power supply adapter (if the board requires one)
- Memory secure digital (SD) card and SD card reader

JTAG Requirements

- FPGA or SoC development board
- USB-JTAG cable
- Installed Vivado software
- Installed Digilent Adept 2 Runtime (for Linux operating systems only)
- Power supply adapter (if the board requires one)

PCI Express Requirements

- FPGA development board
- USB-JTAG cable
- Installed Vivado software
- PCI Express slot and available space on the motherboard
- Power supply adapter (if the board requires one)

Setup Steps

The guided setup wizard displays the setup steps for the selected interface. Follow these steps to set up your hardware board with the selected interface.

Ethernet

- 1** Make sure that the board power switch is off during these setup steps. You are prompted to turn the power on at a later step.
- 2** Connect the AC power cord to the power plug and plug the power supply adapter cable into the hardware board.
- 3** Use the crossover Ethernet cable to connect the Ethernet connector on the hardware board directly to the Ethernet adapter on your host computer.
- 4** Use the USB-JTAG download cable to connect the hardware board to the host computer.

- 5 Make sure that all the jumpers on the hardware board are in the factory default position.
- 6 Turn the power switch of the hardware board on.

Ethernet on Zynq SoC Board

- 1 Make sure that the board power switch is off during these setup steps. You are prompted to turn the power on at a later step.
- 2 Follow the guided setup to:
 - a Configure the network interface card in the host computer. See “Configure NIC on Host Computer” on page 2-6.
 - b Copy the compatible SD card image files for the hardware board to an SD card drive path. See “Select a Drive and Load Firmware” on page 2-7.
 - c Configure the jumpers on the hardware board. See “Set Jumper Switches” on page 2-8.
 - d Connect the hardware board. See “Connect Hardware” on page 2-13.

Ethernet on Versal Board

The Versal board supports Ethernet connection for FIL, but not through the guided setup.

- 1 Make sure that the board power switch is off during these setup steps.
- 2 Follow these manual steps setup to:
 - a Configure the network interface card in the host computer. See “Configure Network Interface Card (NIC) on Development Computer” on page 2-17.
 - b To copy the compatible image files to an SD card drive path, use the `copyImageToHostSDCardPath` function.
 - c Configure the jumpers on the hardware board. See “Set Jumper Switches” on page 2-8.
 - d Connect the hardware board. See “Connect Hardware” on page 2-13.

JTAG

- 1 Make sure that the board power switch is off during these setup steps. You are prompted to turn the power on at a later step.
- 2 Connect the AC power cord to the power plug and plug the power supply adapter cable into the hardware board.
- 3 Use the USB-JTAG download cable to connect the hardware board to the host computer.
- 4 Make sure that all the jumpers on the hardware board are in the factory default position.
- 5 Turn the power switch of the hardware board on.

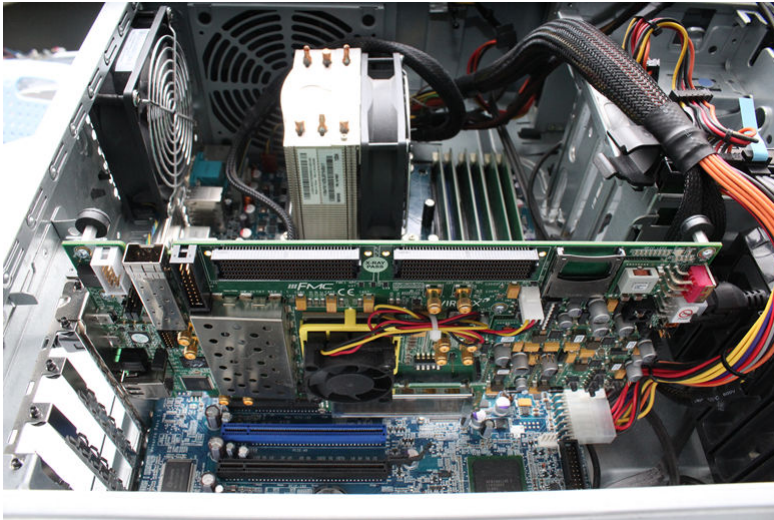
PCI Express

- 1 Make sure that the board power switch is off during these setup steps. You are prompted to turn the power on at a later step.
- 2 Select the maximum number of PCI Express lanes that the board supports. For details, refer to the user manual for the board.

Supported Board	PCI Express Setup	Documentation
Kintex-7 KC705	Set jumper J32 so that it connects pins 5 and 6. This setting selects 8-lane PCIe (default board setting).	https://www.xilinx.com/products/boards-and-kits/ek-k7-kc705-g.html
Virtex-7 VC707	Set jumper J49 so that it connects pins 5 and 6. This setting selects 8-lane PCIe (not the default board setting).	https://www.xilinx.com/products/boards-and-kits/ek-v7-vc707-g.html

- 3 Turn the host computer off.
- 4 Install the hardware board in a PCI Express slot inside the host computer.

This figure shows the VC707 board installed in a host computer. The power cable is on the right. This installation applies to all supported Xilinx boards.



- 5 Plug the external power supply into the wall outlet. Then, plug the power supply adapter cable into the hardware board.
- 6 Connect the JTAG cable to the hardware board and the host computer. The JTAG cable is required to program the FPGA.



- 7 Turn the power switch of the hardware board on.
- 8 Start up the host computer.

Configure NIC on Host Computer

This step is required only when you select the Ethernet interface.

In this step, you configure the host computer so that it can communicate with the hardware board. You must have a dedicated Gigabit Ethernet NIC for the hardware board, with an Ethernet cable connecting the card to your hardware board. If you also want simultaneous internet access and you do not have a wireless connection, your host computer requires a second Ethernet NIC.

In the guided setup, select the NIC that you want to connect with the hardware board. If you have already configured the NIC, select **Skip this step if your network card is already configured for communicating with the FPGA or SoC board.**

The list displays the connected NICs detected on your host computer. The menu options show each NIC as (In Use) or (Available). The installer marks an NIC as (In Use) when the NIC is connected to a device and has an assigned IP address.

If you do not see your NIC listed, click **Refresh** to trigger the NIC detection and refresh the list. Refreshing the list is useful when, for example, you plug in a USB Ethernet adapter dongle while viewing this pane.

- If all the NICs listed are in use, free up a NIC for use with the hardware and then click **Refresh**.
- If the NIC list is empty, VMWare software, if present, can interfere with NIC detection. To get an accurate list of NICs on your host computer, remove the VMWare software.
- Check whether the missing NIC is disabled in the control panel. If the NIC is disabled, enable it.

Leave the IP address for the NIC as the default. Alternatively, specify the IP address in dotted quad format, for example, 192.168.0.1.

When you click **Next**, the software configures the NIC.

Note Guided setup does not support Ethernet on Versal boards. For manual configuration, see “Configure Network Interface Card (NIC) on Development Computer” on page 2-17.

Select a Drive and Load Firmware

This step is required only when you select the Ethernet interface on a Zynq SoC board.

Next, the installer must write an FPGA image to an SD card. This FPGA image is included with the support package. The image includes the embedded software and the FPGA programming file necessary for using the hardware board as an I/O peripheral.

- 1 Insert an SD card into the card reader on the host computer.
 - 8 GB or larger for Versal boards
 - 4 GB or larger for other boards

The card must be in FAT32 format. Select the appropriate drive from the list. If you have already downloaded the FPGA image, skip this step.



Note Unlock the SD card before downloading the firmware image to the card. Keep the card unlocked while the card is in the Zynq board card reader.

- 2 Write the FPGA image to the SD card. In the guided setup, select the location of the SD drive containing the card, then click **Next**. On the next screen, to copy the programming file from the host computer to the SD card, click **Write**. This process erases any existing data on the card.

Note Guided setup does not support Ethernet on Versal boards. Use the `copyImageToHostSDCardPath` function.

Install PCI Express Driver

This step is required only when you select the PCI Express interface.

If you have already installed the PCI Express drivers, skip this step.

Install the PCI Express drivers before you use FIL, FPGA data capture, or AXI manager with a PCI Express connection. This step performs the driver installation for you. The process can take 10 or more minutes to install, and might require system administrator privileges.

You can install the drivers now, or you can choose to perform the setup again later. To run the support package setup, on the MATLAB **Home** tab, in the **Environment** section, select **Help > Check for Updates**.

Set Jumper Switches

This step is required only when you select the Ethernet interface on a Versal or Zynq SoC board or the PCI Express interface.

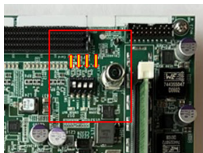
Configure the jumpers on the Versal or Zynq SoC board so that you can use it as a peripheral device. These jumper settings make it so that the board starts up from the SD card. Make sure that the board is turned off.

The jumper settings are different for each board. To learn more about the settings, see the board documentation.

Set Jumpers on Versal VCK190

SW1 Switch Positions

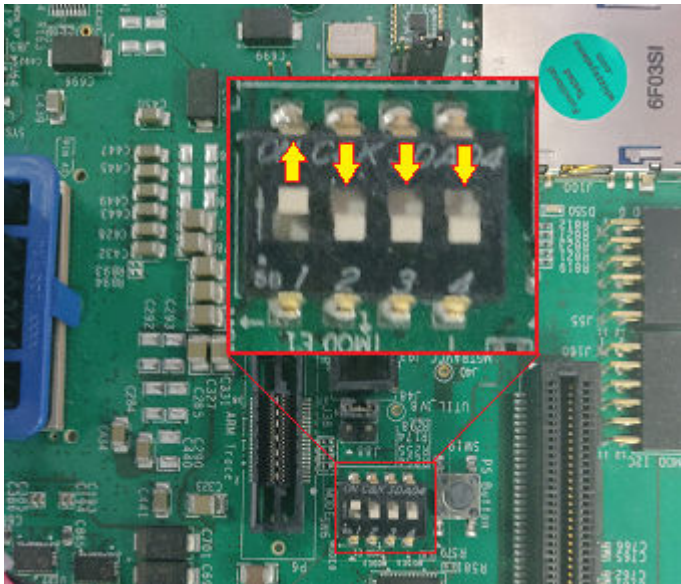
Switch	Switch Position
1	Up
2	Down
3	Down
4	Down



Set Jumpers on ZCU102

SW6 Switch Positions

Switch	Switch Position
1	Up
2	Down
3	Down
4	Down



Set Jumpers on ZCU111

SW6 Switch Positions

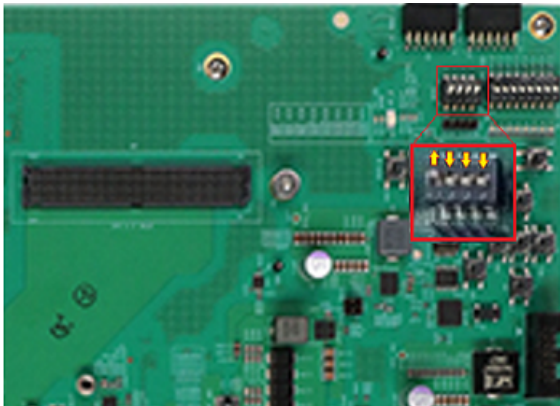
Switch	Switch Position
1	Up
2	Down
3	Down
4	Down



Set Jumpers on ZCU216

SW2 Switch Positions

Switch	Switch Position
1	Up
2	Down
3	Down
4	Down



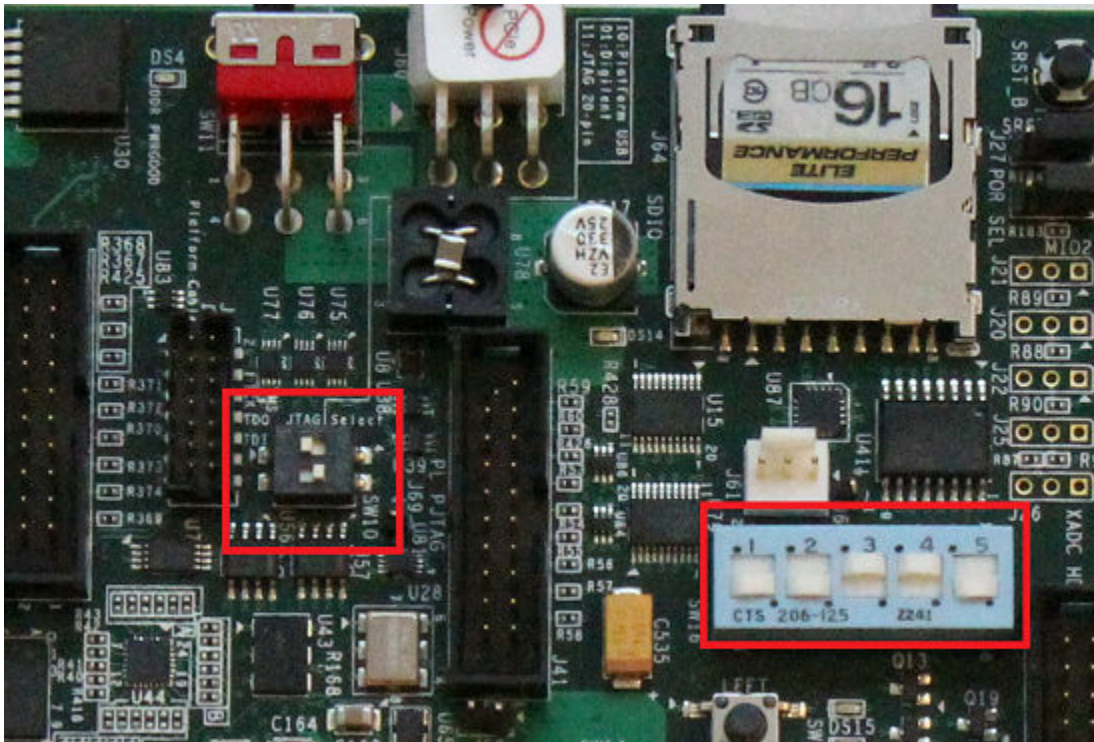
Set Jumpers on ZC702

JTAG Select Jumper Positions

Switch	Switch Position
Top	Left
Bottom	Right

SW10 Jumper Positions

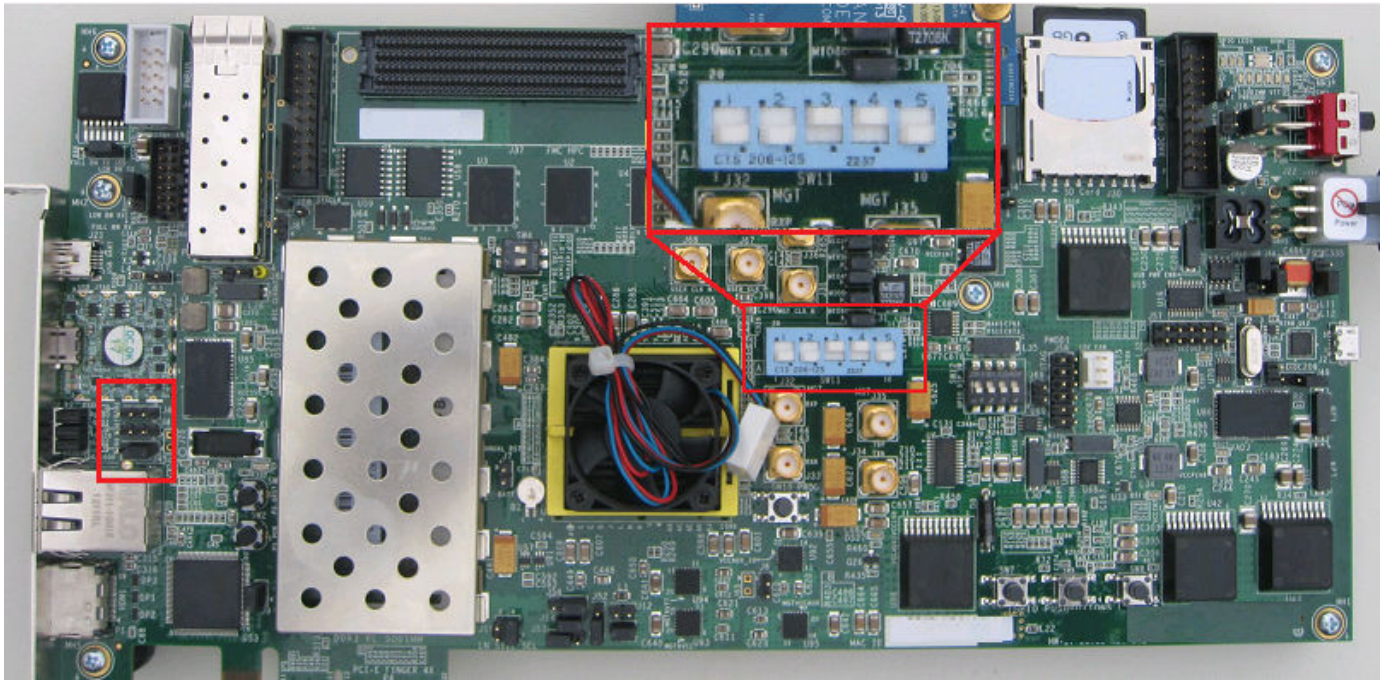
Switch	Switch Position
1	Down
2	Down
3	Up
4	Up
5	Down



Set Jumpers on ZC706

SW11 Jumper Positions

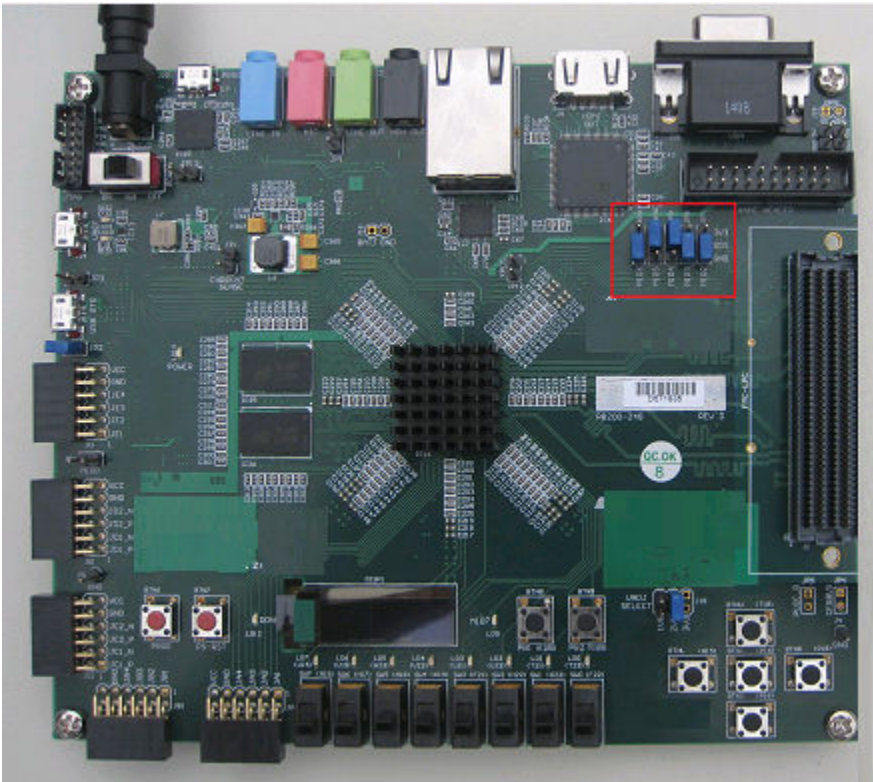
Switch	Switch Position
1	Down
2	Down
3	Up
4	Up
5	Down



Set Jumpers on ZedBoard

Jumper Positions

Switch	Switch Position
1	Down
2	Up
3	Up
4	Down
5	Down

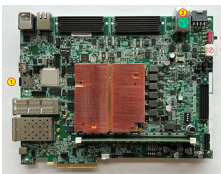


Connect Hardware

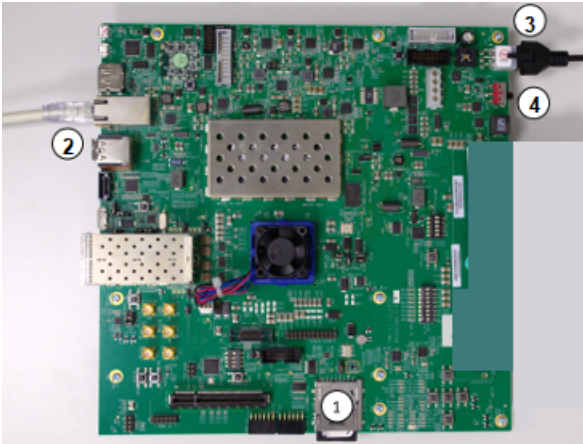
This step is required only when you select the Ethernet interface on a Versal or Zynq SoC board.

Follow these instructions for connecting the hardware. The guided setup wizard provides labeled pictures of the steps for each board.

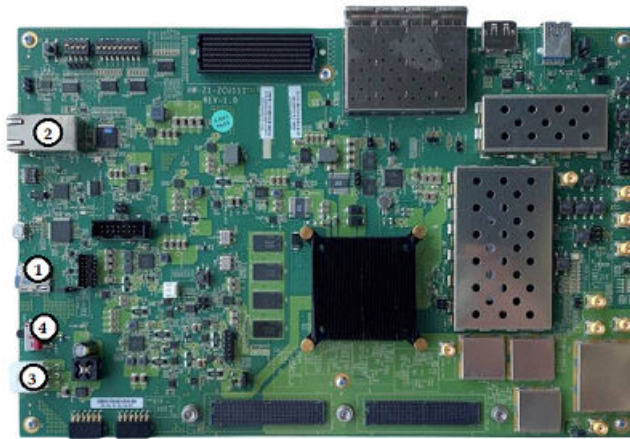
- 1 Remove the SD card from the host computer and insert it into the hardware board.
- 2 Connect an Ethernet cable to the board. Connect the other end of the Ethernet cable to the selected NIC.
- 3 Connect the power cable.
- 4 Turn the power on.



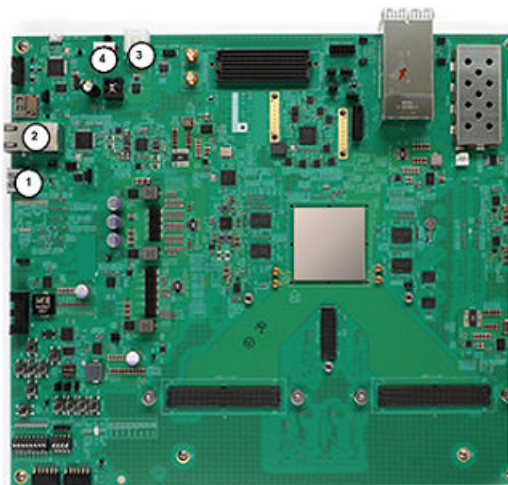
Connect VCK190 Board



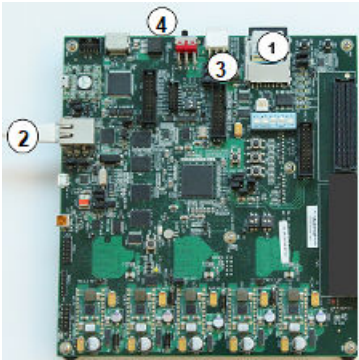
Connect ZCU102 Board



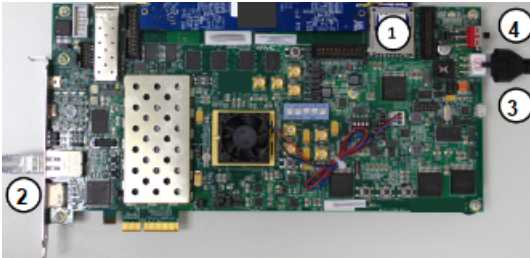
Connect ZCU111 Board



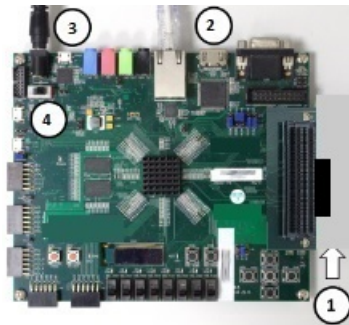
Connect ZCU216 Board



Connect ZC702 Board



Connect ZC706 Board



Connect ZedBoard

Verify Setup

You can verify the hardware setup for Ethernet and JTAG interfaces. This step runs the tests to verify the connection between the host computer and the hardware board based on the selected interface. Before you run the test, make sure that:

- 1 You have installed the appropriate vendor tool and the tool is on the MATLAB path. See “Set Up FPGA Design Software Tools”.
- 2 The board is turned on.

This step runs these tests to verify the connection for the selected interface.

Ethernet

- 1 Generate an FPGA programming file for your hardware board.
- 2 Program the FPGA.
- 3 Detect an Ethernet connection.

Ethernet on Versal or Zynq SoC Board

- 1 Verify the IP address configuration on the host computer.
- 2 Verify the Ethernet connection between the host computer and the hardware board.
- 3 Zynq SoC — Read and write the memory locations on the hardware board using AXI manager.

JTAG

- 1 Generate an FPGA programming file for your board.
- 2 Program the FPGA.
- 3 Perform the data transaction between the FPGA and the host computer.

If the connection is not successful, the most common reasons are that the board is not connected properly or it is not turned on. Check the cable connections and power switch and try again.

Open Examples

When the installer completes your hardware setup, you can exit the installer or open the examples to get started.

See Also

Hardware Setup | “FPGA-in-the-Loop Simulation” | “Data Capture Workflow” on page 6-2 | “Set Up AXI Manager” on page 3-2

See Also

Related Examples

- Verify HDL Implementation of PID Controller Using FPGA-in-the-Loop
- “Capture Temperature Sensor Data from Xilinx FPGA Board Using FPGA Data Capture” on page 8-2
- “Access FPGA Memory Using JTAG-Based AXI Manager” on page 8-15
- “Access FPGA Memory Using Ethernet-Based AXI Manager” on page 8-24
- “Access FPGA External Memory Using AXI Manager over PCI Express” on page 8-28

Configure Network Interface Card (NIC) on Development Computer

To connect the Xilinx devices to the development computer, you must configure an available network connection on the development computer. Follow the steps outlined for your specific operating system.

Windows

- 1 Open the **Control Panel**.
- 2 Set **View by** to **Category**.
- 3 Click **Network and Internet**.
- 4 Click **Network and Sharing Center**.
- 5 On the left pane, click **Change adapter settings**.
- 6 Right-click the local area network connection that is connected to the radio hardware and select **Properties**.
 - If an unused network connection is available, the local area connection appears as **Unidentified network**.
 - If you plan to reuse your network connection, select the local area connection that you plan to use for the radio hardware.
 - If you have only one network connection, check if you can connect wirelessly to the existing local area network. If you can, you can use the network connection for the radio hardware.
 - You can use a pluggable USB to Gigabit Ethernet LAN adapter instead of a NIC. The instructions are the same.
- 7 On the **Networking** tab of the **Properties** dialog box, clear all options except **Internet Protocol Version 4 (TCP/IPv4)**. Other services, particularly antiviral software, can cause intermittent connection problems with the radio hardware.
- 8 Double-click **Internet Protocol Version 4 (TCP/IPv4)**.
- 9 On the **General** tab, select **Use the following IP Address**.
- 10 Leave the subnet mask set to the default value of `255.255.255.0` and click **OK**.

Linux

- 1 Set the host network IP address to `192.168.1.x`, where `x` is any number in the range 1 through 255, apart from 101. Set this value using the `ifconfig` command.

```
% sudo ifconfig ethZ 192.168.3.3 netmask 255.255.255.0
```

In this syntax, `ethZ` is the name of the host Ethernet port (usually `eth0`, `eth1`, and so on). To use the `sudo` command, you might have to enter a password.

- 2 Confirm the changes by entering the following command in the shell:

```
% ifconfig ethZ
```

Hardware Setup

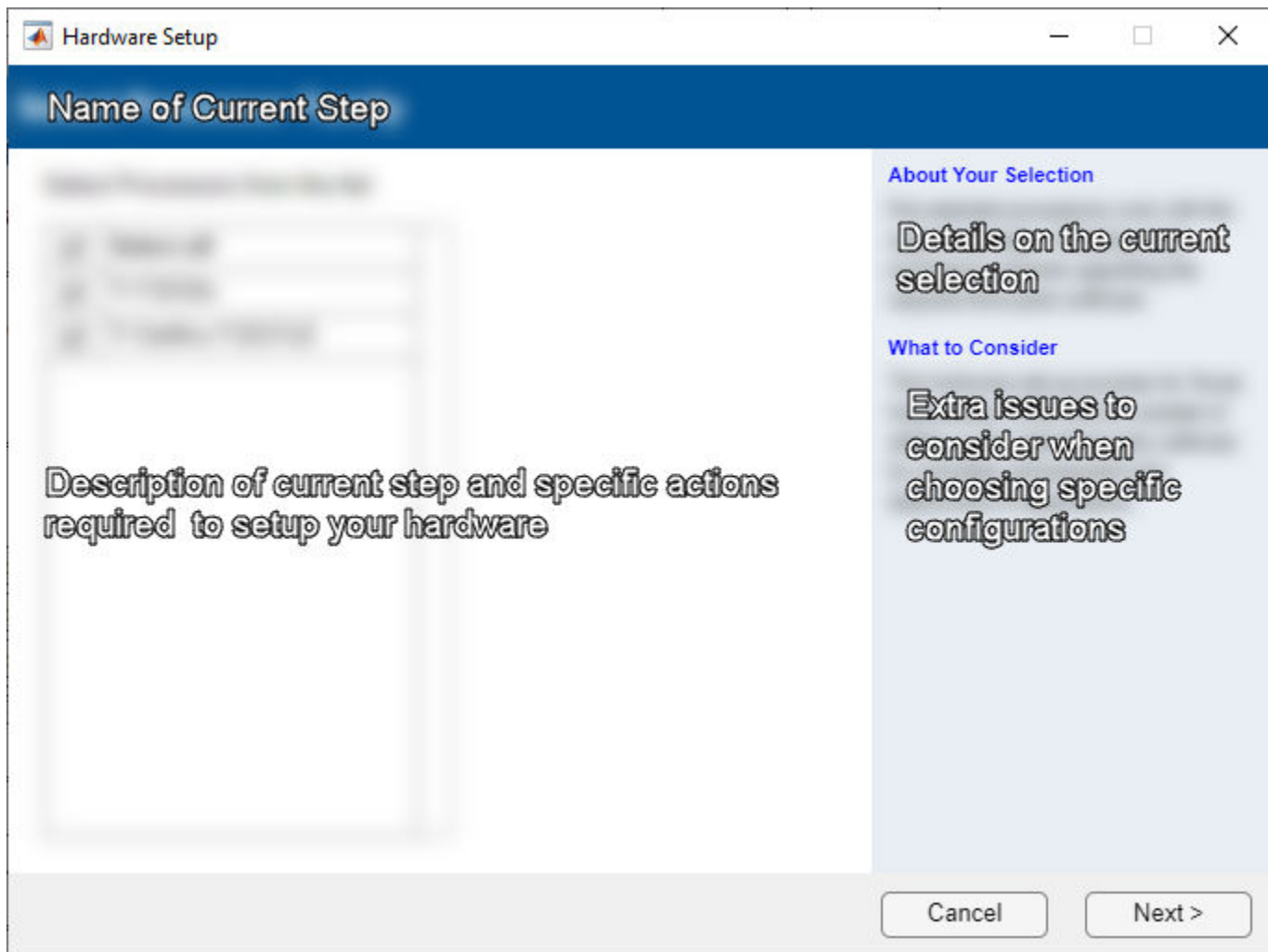
Set up and connect your hardware boards

Description


Hardware boards supported by MathWorks require additional setup steps to connect to MATLAB and Simulink software. The **Hardware Setup** tool guides you through the hardware setup process. Use this tool to configure a target hardware board for use with FPGA-in-the-loop (FIL), FPGA data capture, and AXI manager over the JTAG, Ethernet, and PCI Express interfaces. For the full list of supported boards and interfaces, see “Supported Xilinx FPGA Boards” on page 1-2.

The hardware setup process for the HDL Verifier Support Package for Xilinx FPGA Boards comprises these steps:

- Specify a hardware board and interface.
- Configure the network interface card in the host computer (for the Ethernet interface only).
- Copy or transfer the compatible SD card image files for the hardware board to an SD card drive path (for the Ethernet interface on Zynq SoC boards only).
- Configure your hardware board to start up from the SD card (for the Ethernet interface on Zynq SoC boards only).
- Install the PCIe driver on the host computer (for the PCI Express interface only).
- Connect your hardware board to the host computer (for the Ethernet interface on Zynq SoC boards only).
- Verify the connection between the host computer and the hardware board.



Open the Hardware Setup

- In the install window, at the end of the installation process, click the **Setup Now** button.
- After installing the HDL Verifier Support Package for Xilinx FPGA Boards, use **Get and Manage Add-Ons**. When the installation is complete, in the Add-On Manager, click the **Gear** icon .

Version History

Introduced in R2016a

See Also

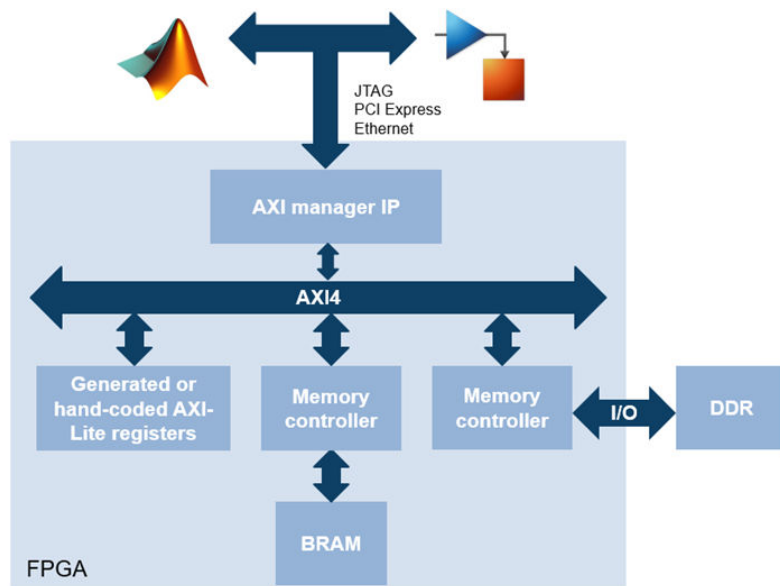
“Guided Hardware Setup” on page 2-2

AXI Manager

Set Up AXI Manager

Note MATLAB AXI master has been renamed to AXI manager. In the software and documentation, the terms "manager" and "subordinate" replace "master" and "slave," respectively.

To access on-board memory locations from MATLAB or Simulink, you must include the AXI manager IP in your FPGA design. This IP connects to subordinate memory locations on the board. The IP also responds to read and write commands from MATLAB or Simulink, over JTAG, PCI Express, or Ethernet cable.



Integrate AXI Manager IP in FPGA Design

To set up the AXI manager IP for access from MATLAB or Simulink, follow these setup steps:

- 1 Add the path for the AXI manager IP files to your project using the `setupAXIManagerForVivado` function.
- 2 Open Vivado, and from the IP Catalog select the AXI manager IP in your FPGA design.
 - When using JTAG as a physical connection, select AXI Manager.
 - When using Ethernet as a physical connection, select UDP AXI Manager and Ethernet MAC Hub and add them to your project.
 - When using PCIe as a physical connection, select PCIe AXI Manager and add it to your project.
- 3 In your FPGA project, specify which addresses the AXI manager IP is allowed to access.
- 4 Compile your FPGA project, including the AXI manager IP.
- 5 Connect your FPGA board to your host computer using a physical cable (JTAG, PCI Express, or Ethernet cable).

6 Program the FPGA with your compiled design.

Note Alternatively, you can perform these steps in the HDL Coder™ guided workflow by using a sample reference design, such as the one included in this example: “Access DUT Registers on Xilinx Pure FPGA Board Using IP Core Generation Workflow” on page 8-37.

After loading the design on your FPGA, you can access memory-mapped locations on the board.

To access the board from MATLAB, create an `aximanager` object and use the `readmemory` and `writememory` methods to read and write memory-mapped locations on the board.

To access the board from Simulink, create a Simulink model and include AXI Manager Write and AXI Manager Read in it. Configure the blocks to read and write memory-mapped locations on the board. For more information, see “Use Simulink to Access FPGA Locations” on page 3-4.

JTAG Considerations

When using JTAG as a physical connection to your board, you might have additional IPs that use the same JTAG connection. Such IPs include Intel® SignalTap II or Xilinx Vivado Logic Analyzer cores. However, only one of these applications can use the JTAG cable at a time. You must release the `aximanager` object to return the JTAG resource for use by other applications.

However, the nonblocking capture mode enables you to simultaneously use FPGA data capture and AXI manager, which share a common JTAG interface. In this capture mode, you do not need to close or release the JTAG resource to switch between FPGA data capture and AXI manager. For more information, see “Simultaneous Use of FPGA Data Capture and AXI Manager” on page 6-12.

The most common conflicting use of the JTAG cable is to reprogram the FPGA. You must stop any FPGA data capture or AXI manager JTAG connection before you can use the cable to program the FPGA.

The maximum data rate between host computer and FPGA is limited by the JTAG clock frequency. For Intel boards, the JTAG clock frequency is 12 MHz or 24 MHz. For Xilinx boards, the JTAG clock frequency is 33 MHz or 66 MHz. The JTAG frequency depends on the type of cable and the maximum clock frequency supported by the FPGA board.

See Also

`aximanager` | AXI Manager Read | AXI Manager Write

Related Examples

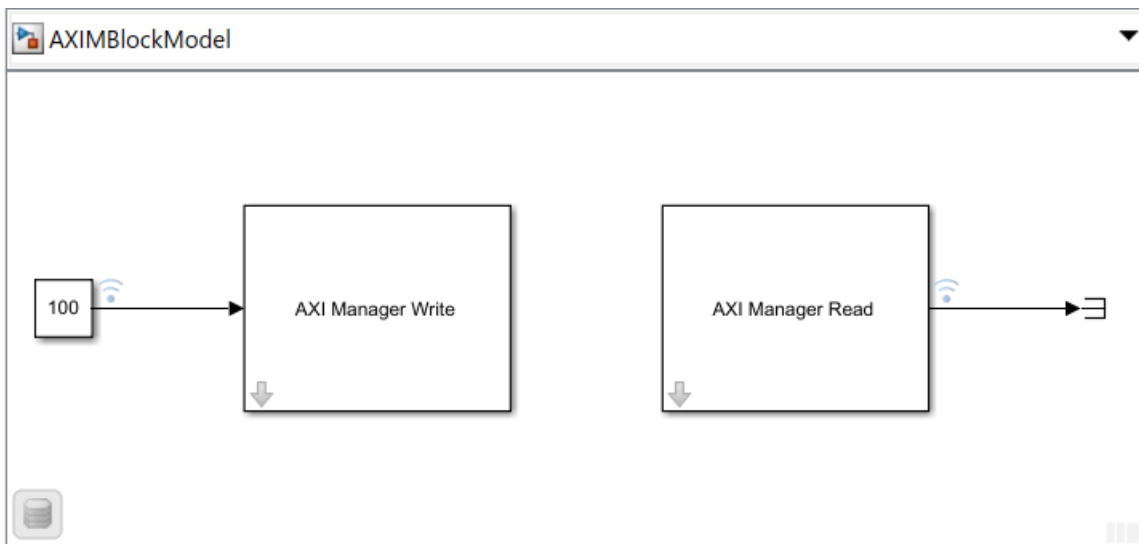
- “Access FPGA Memory Using JTAG-Based AXI Manager” on page 8-15
- “Ethernet AXI Manager” on page 3-10
- “PCI Express AXI Manager” on page 3-6
- “Access DUT Registers on Xilinx Pure FPGA Board Using IP Core Generation Workflow” on page 8-37

Use Simulink to Access FPGA Locations

Note MATLAB AXI master has been renamed to AXI manager. In the software and documentation, the terms "manager" and "subordinate" replace "master" and "slave," respectively.

To read and write memory-mapped locations on your FPGA board using Simulink, you must first integrate an AXI manager IP into your FPGA design. For more information, see "Integrate AXI Manager IP in FPGA Design" on page 3-2.

After integrating an AXI manager IP into your FPGA design, load the design on the FPGA. Then, create a Simulink model that includes source, sink, AXI Manager Write, and AXI Manager Read blocks, as in this figure.

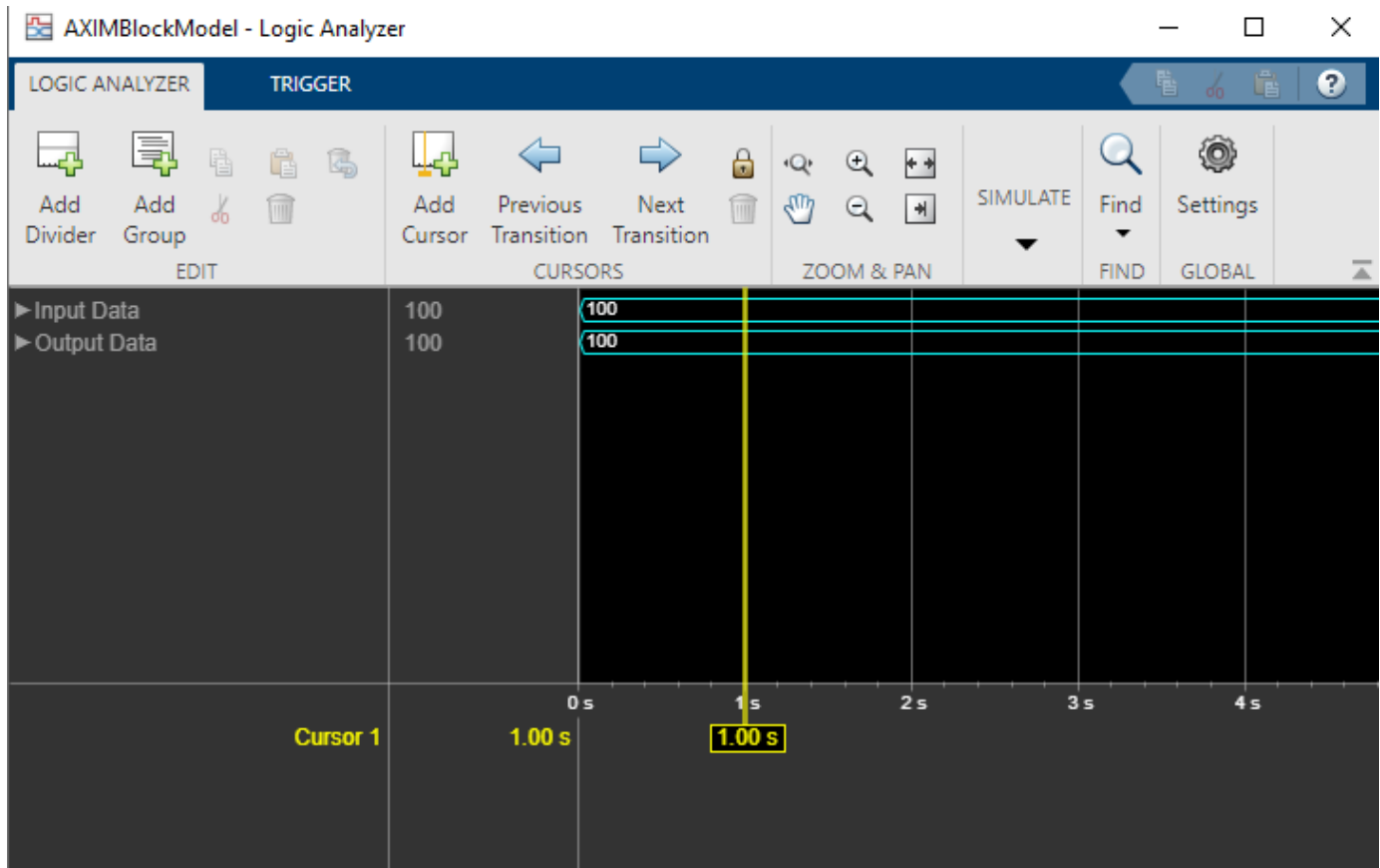


Configure the AXI Manager Write block. Set the write **Address** and **Burst type** parameters. On the **Interface** tab, select the type of interface used for communication with the FPGA board by using the **Type** parameter. Then, click **Configure global parameters** to configure the global interface parameters for that AXI manager interface.

Next, configure the AXI Manager Read block. Set the read **Address**, **Burst type**, **Output data type**, and **Output vector size** parameters. On the **Interface** tab, select the type of interface used for communication with the FPGA board by using the **Type** parameter.

Run the simulation. For each Simulink step, the write block writes to the FPGA, and the read block reads from the FPGA. View the results by using the **Logic Analyzer** app, or directing the data to a file.

This figure shows the input and output data displayed in the **Logic Analyzer** app. In this example, the AXI Manager Write block writes 100 to address 0, and the AXI Manager Read block reads from the same address.



See Also

AXI Manager Read | AXI Manager Write

More About

- "Set Up AXI Manager" on page 3-2

PCI Express AXI Manager

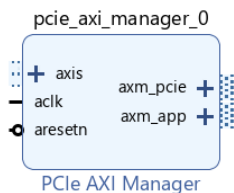
Note PCI Express AXI master has been renamed to PCI Express AXI manager and the PCIe MATLAB as AXI Master IP has been renamed to the PCIe AXI Manager IP. In the software and documentation, the terms "manager" and "subordinate" replace "master" and "slave," respectively.

When using PCI Express AXI manager, you must first include the following two intellectual property (IP) blocks in your Xilinx Vivado project.

- PCIe AXI Manager IP
- PCI Express Core

PCIe AXI Manager IP

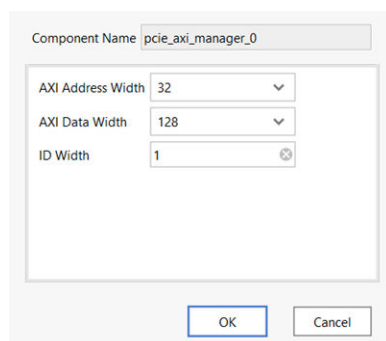
PCIe AXI Manager is an HDL IP provided by MathWorks. This IP connects the PCI Express (PCIe) core to your application code. The IP has a configuration port for accessing configuration registers. This block diagram shows the interface to the HDL IP. To know how to include the *PCIe AXI Manager* IP in your FPGA design, see "Set Up AXI Manager" on page 3-2.



The interface includes the following parts:

- `clock` and `resetn` are the clock and reset inputs. Connect them to the AXI clock and reset.
- `axs_s0` is a 32-bit subordinate interface and is used for accessing the PCIe configuration registers. Connect this interface to the Kintex UltraScale+ FPGA KCU116 memory mapped manager interface.
- `axm_pcie` is a 128-bit AXI manager interface. Connect this interface to the `S_AXI_B` subordinate port on the PCIe core.
- `axm_app` is a 128-bit AXI manager interface. Connect this interface to your application logic.

After instantiating this IP in your design, open the block parameters for configuration.



Configure these parameters:

- **AXI Address Width** - This parameter is the address bus width. The IP supports 32-bit address.
- **AXI Data Width** - This parameter is the data bus width. The IP supports 128-bit or 256-bit data. Note that this parameter is not identical to the data width of the `axi_manager` object or the AXI Manager Read or AXI Manager Write blocks. If the data width is set to 32 bits, and the **AXI Data Width** of your IP is set to 128 bits, HDL Verifier packs four 32-bit words to transfer on the 128-bit bus.
- **ID Width** - This parameter is the ID width in bits. Its value must match the ID width of the AXI subordinate.

PCI Express Core

The *DMA/Bridge Subsystem for PCI Express Core* is a board-specific IP provided by Xilinx. Use this IP for configuring and integrating the PCI Express port. For more information on how to include this IP in your FPGA design, see “Set Up AXI Manager” on page 3-2.

After instantiating the PCIe core HDL IP in your Xilinx Vivado project, configure the PCIe core using these steps. This example is for a Kintex UltraScale+ FPGA KCU116 board.

- 1 On the **Basic** tab, set the parameters as shown in this figure.

DMA/Bridge Subsystem for PCI Express (PCIe) (4.1)

Documentation IP Location

Component Name: `xdma_0`

Basic | PCIe ID | PCIe : BARs | PCIe : MISC | AXI : BARs | AXI : MISC

Functional Mode: AXI Bridge

Mode: Basic

Device / Port Type: PCI Express Endpoint device

PCIe Block Location: X0Y0

PCIe Interface

Lane Width: X8

Maximum Link Speed: 2.5 GT/s (selected), 5.0 GT/s, 8.0 GT/s

Reference Clock Frequency (MHz): 100 MHz

AXI Interface

AXI Address Width: 32 [32 - 64]

AXI Data Width: 64 bit, 128 bit (selected)

AXI Clock Frequency: 125

Enable AXI Slave Interface

Enable AXI Master Interface

Enable PIPE Simulation

Enable GT Channel DRP Ports

Enable PCIe DRP Ports

Additional Transceiver Control and Status Ports

OK Cancel

- On the **PCIe ID** tab, set the parameters as shown in this figure.

DMA/Bridge Subsystem for PCI Express (PCIe) (4.1)

Documentation IP Location

Component Name:

Basic **PCIe ID** PCIe : BARs PCIe : MISC AXI : BARs AXI : MISC

ID Initial Values

Vendor ID	<input type="text" value="10EE"/>	
Device ID	<input type="text" value="101A"/>	
Revision ID	<input type="text" value="01"/>	
Subsystem Vendor ID	<input type="text" value="015B"/>	
Subsystem ID	<input type="text" value="0001"/>	

Class Code Lookup Assistant

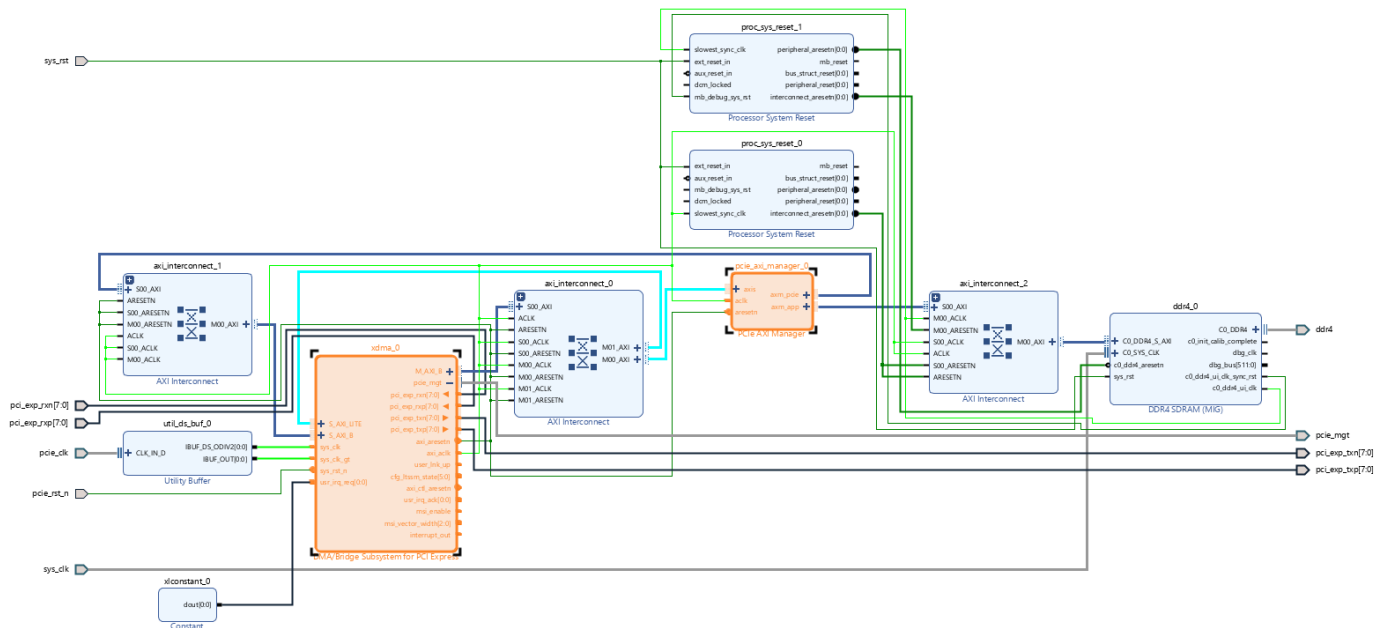
Use Class Code Lookup Assistant

Base Class Menu	<input type="text" value="Memory controller"/>	
Base Class Value	<input type="text" value="05"/>	Range: 00..FF
Sub Class Interface Menu	<input type="text" value="Other memory controller"/>	
Sub Class Value	<input type="text" value="80"/>	Range: 00..FF
Interface Value	<input type="text" value="00"/>	Range: 00..FF
Class Code	<input type="text" value="058000"/>	Range: 000000..FFFFFF

OK Cancel

The **ID Initial Values** listed in the **PCIe** tab screen are the required PCIe ID settings to ensure compatibility with MathWorks PCIe device driver for Xilinx FPGA boards.

- Connect the PCIe AXI Manager IP to the PCIe core. This example shows the Kintex UltraScale+ FPGA KCU116 DMA/Bridge Subsystem IP for PCI Express.



- 4 Compile and build your FPGA project.
- 5 Insert the FPGA board into the PCI Express slot on the motherboard of the host machine.
- 6 Program FPGA with the bitstream generated for your design.
- 7 Restart the host machine.

Once the program is running on your FPGA board, you can create an AXI manager object in your MATLAB command window. For more information, see `axi_manager`. To access the subordinate memory locations on the board, use the `readmemory` and `writememory` functions of this object.

See Also

`axi_manager`

Related Examples

- “Access FPGA External Memory Using AXI Manager over PCI Express” on page 8-28

More About

- “Set Up AXI Manager” on page 3-2

Ethernet AXI Manager

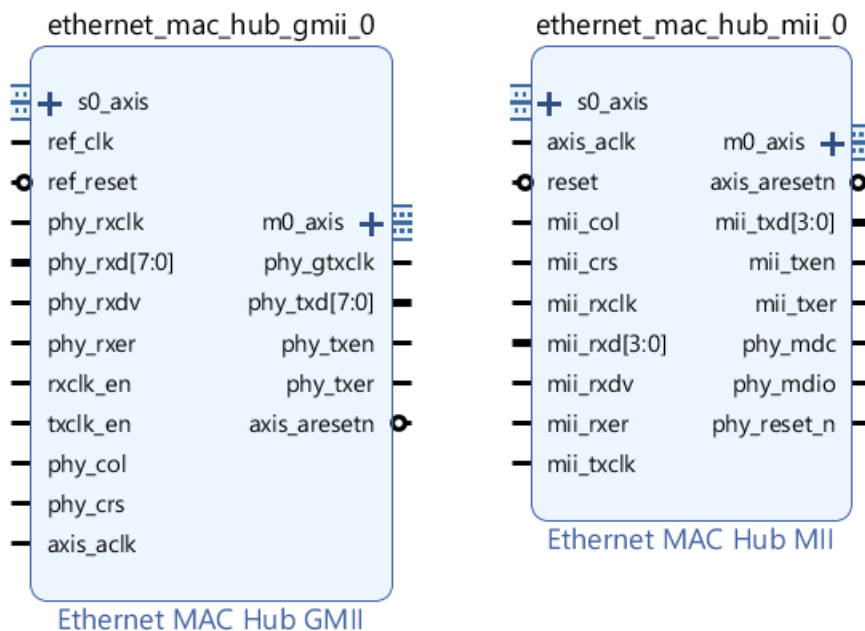
Note Ethernet AXI master has been renamed to Ethernet AXI manager and the UDP MATLAB as AXI Master IP has been renamed to the UDP AXI Manager IP. In the software and documentation, the terms "manager" and "subordinate" replace "master" and "slave," respectively.

Integrate and configure AXI manager over Ethernet using user datagram protocol (UDP). To use Ethernet AXI manager, you must first include these two intellectual property (IP) blocks in your project: Ethernet media access controller (MAC) Hub and UDP AXI Manager.

Ethernet MAC Hub IP

The Ethernet MAC Hub IP connects the Ethernet physical layer (PHY) to the UDP AXI Manager IP. Use the following Ethernet MAC Hub IPs to connect the target FPGA board through various types of interfaces.

- Ethernet MAC Hub GMII IP — This IP supports the gigabit media independent interface (GMII).
- Ethernet MAC Hub MII IP — This IP supports the media independent interface (MII).
- Ethernet MAC Hub GMII IP and 1G/2.5G Ethernet PCS/PMA or SGMII Xilinx IP — Connect these two IPs to support the serial gigabit media independent interface (SGMII).



Interface of Ethernet MAC Hub IP

The following tables provide the port description of Ethernet MAC Hub GMII and Ethernet MAC Hub MII.

Port	Description
s0_axis	AXI-stream subordinate interface. Connect this port to the m_axis port on the UDP AXI Manager IP.
m0_axis	AXI-stream manager interface. Connect this port to the s_axis port on the UDP AXI Manager IP.

Ethernet MAC Hub GMII IP Ports

Port	Direction	Description
ref_clk	Input	Reference clock signal that drives phy_gtxclk . The frequency of ref_clk must be the same as the phy_rxclk clock frequency.
ref_reset	Input	IP reset signal.
phy_rxclk	Input	Receive clock from PHY.
phy_rxd[7:0]	Input	Receive data signal from PHY.
phy_rxdv	Input	Receive data valid control signal from PHY.
phy_rxer	Input	Receive error signal from PHY.
rxclk_en	Input	Receiver clock enable.
txclk_en	Input	Transmitter clock enable.
phy_col	Input	Collision detect signal from PHY.
phy_crs	Input	Carrier sense detect signal from PHY.
axis_aclk	Input	Clock signal for AXI-stream interface.
phy_gtxclk	Output	Clock to PHY.
phy_txd[7:0]	Output	Transmit data signal to PHY.
phy_txen	Output	Transmit enable control signal to PHY.
phy_txer	Output	Transmit error signal to PHY.
axis_aresetn	Output	Active-low reset. Reset signal for AXI-stream interface. You can use this port to reset the downstream AXI peripherals.

Ethernet MAC Hub MII IP Ports

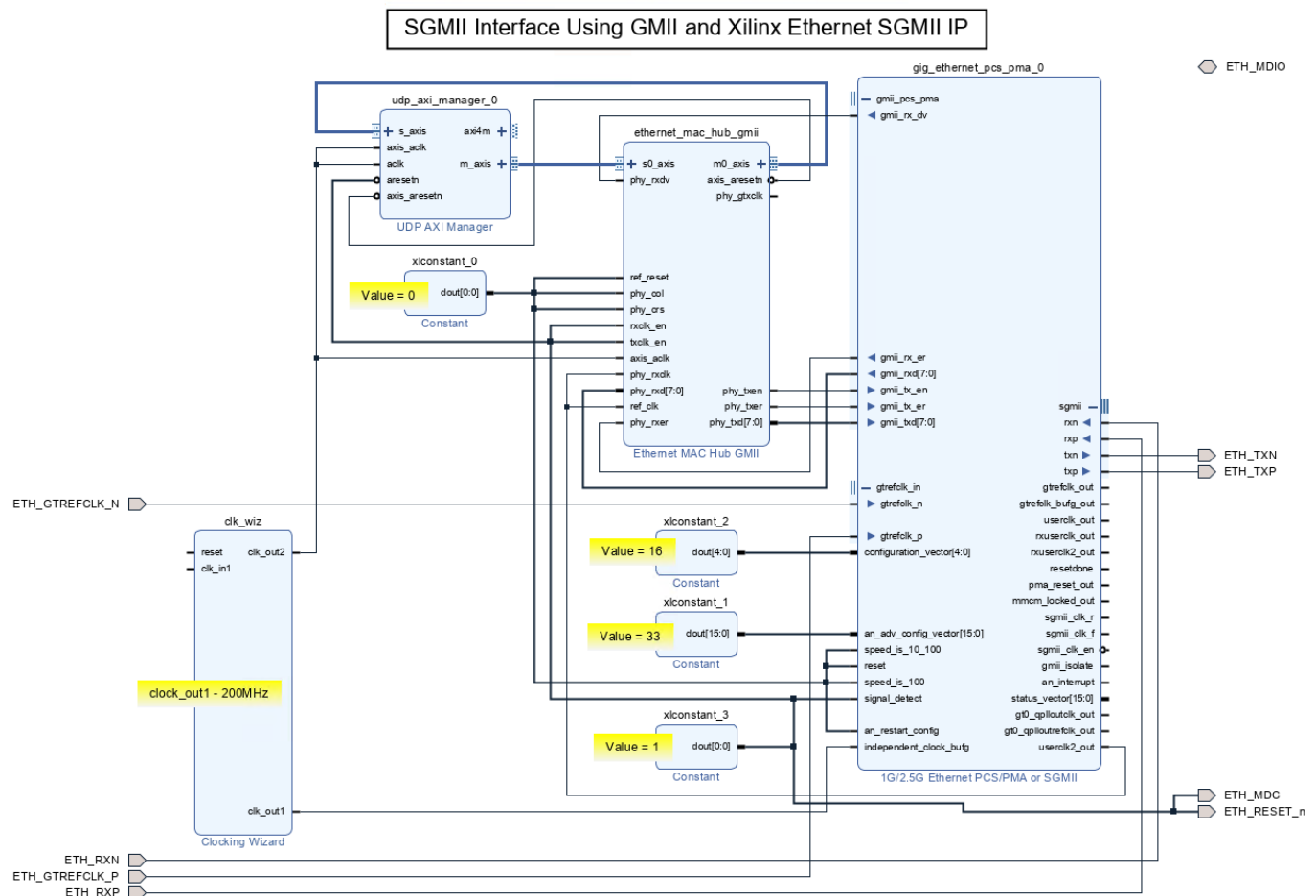
Port	Direction	Description
axis_aclk	Input	Clock signal for AXI-stream interface.
reset	Input	IP reset signal.
mii_col	Input	Collision detect signal from PHY.
mii_crs	Input	Carrier sense detect signal from PHY.
mii_rxclk	Input	Receive clock signal from PHY.
mii_rxd[3:0]	Input	Receive data signal from PHY.
mii_rxdv	Input	Receive data valid control signal from PHY.
mii_rxer	Input	Receive error signal from PHY.
mii_txclk	Input	Transmit clock signal from PHY.

Port	Direction	Description
axis_aresetn	Output	Active-low reset. Reset signal for AXI-stream interface. You can use this port to reset the downstream AXI peripherals.
mii_txd[3:0]	Output	Transmit data signal to PHY.
mii_txen	Output	Transmit enable control signal to PHY.
mii_txer	Output	Transmit error signal to PHY.
phy_mdc	Output	Management data clock (MDC) signal to PHY.
phy_mdio	Inout	Data signal for communication with management data input/output (MDIO) controller.
phy_reset_n	Output	Active-low reset signal to PHY.

For more information about port connections, see “Access FPGA Memory Using Ethernet-Based AXI Manager” on page 8-24.

Ethernet MAC Hub IP Connections for SGMII

For an SGMII, connect the Ethernet MAC Hub GII IP to the 1G/2.5G Ethernet PCS/PMA or SGMII Xilinx IP as this figure shows.



Based on the type of Ethernet interface of your target FPGA board, instantiate the Ethernet MAC Hub GMII or Ethernet MAC Hub MII HDL IP in your design. After instantiating the Ethernet MAC Hub IP in your design, open the block parameters for configuration. This figure shows the block parameters for the Ethernet MAC Hub GMII IP.

Component Name: ethernet_mac_hub_gmii_0

Number of AXI Stream Channels	1
MAC Address	0x000A3502218A
IP Address Byte1	192
IP Address Byte2	168
IP Address Byte3	0
IP Address Byte4	2
UDP Port For Channel 1	50101
UDP Port For Channel 2	50102
UDP Port For Channel 3	50103
UDP Port For Channel 4	50104
UDP Port For Channel 5	50105
UDP Port For Channel 6	50106
UDP Port For Channel 7	50107
UDP Port For Channel 8	50108

Buttons: OK, Cancel

Ethernet MAC Hub IP Parameters

Configure these parameters:

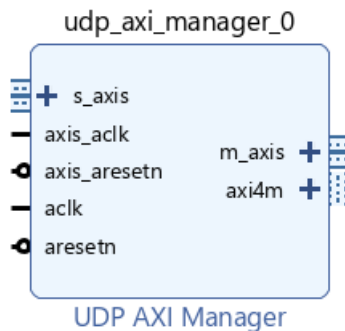
- **Number of AXI Stream Channels** — This parameter decides the number of AXI-stream channels in the Ethernet MAC Hub IP. Select this value as an integer from 1 to 8. The default value is 1.
- **IP Address Byte1, IP Address Byte2, IP Address Byte3, IP Address Byte4** — These parameters set the four bytes in the range from 0 to 255 composing the UDP internet protocol (IP) address of the device. This address must match the DeviceAddress property value of the aximanager object.
- **UDP Port For Channel 1, UDP Port For Channel 2, UDP Port For Channel 3, UDP Port For Channel 4, UDP Port For Channel 5, UDP Port For Channel 6, UDP Port For Channel 7, UDP Port For Channel 8** — These parameters set the UDP port numbers. Specify each parameter value as an integer from 255 to 65,535. These port numbers must match the Port property value of the aximanager object.

Ethernet MAC Hub IP Limitations

- For SGMII, the clock signal for the AXI-Stream interface (**axis_aclk**) is limited to 50 MHz.
- Ethernet management interfaces MDC and MDIO do not enable you to configure the Ethernet PHY.

UDP AXI Manager IP

The UDP AXI Manager HDL IP connects the Ethernet MAC Hub IP to your application IP. The UDP AXI Manager IP acts as a bridge that translates data between an AXI peripheral and MATLAB.



Interface of UDP AXI Manager IP

The interface of the UDP AXI Manager IP includes the ports described in these tables.

Port	Description
s_axis	AXI-stream subordinate interface.
m_axis	AXI-stream manager interface.
axi4m	AXI4-full manager interface.

UDP AXI Manager IP Ports

Port	Direction	Description
axis_aclk	Input	Clock signal for AXI-stream interface.
axis_aresetn	Input	Active-low reset signal for AXI-stream interface.
aclk	Input	Clock signal for AXI4-full interface.
aresetn	Input	Active-low reset. Reset signal for AXI4-full interface.

After instantiating the UDP AXI Manager IP in your design, open the block parameters for configuration.

Component Name:

AXI Address Width:

AXI Data Width:

ID Width:

UDP AXI Manager IP Parameters

Configure these parameters:

- **AXI Address Width** — This parameter is the address bus width in bits. The IP supports 32 bits or 64 bits.
- **AXI Data Width** — This parameter is the data bus width in bits. The IP supports 32 bits or 64 bits.
- **ID Width** — This parameter is the ID width in bits. Its value must match the ID width of the AXI4 subordinate.

When the program is running on your FPGA board, you can create an AXI manager object using the `aximanager` object. To access the subordinate memory locations on the board, use the `readmemory` and `writememory` object functions.

See Also

`aximanager`

Related Examples

- “Access FPGA Memory Using Ethernet-Based AXI Manager” on page 8-24
- “Leverage Built-In Ethernet on Zynq to Perform Memory Access Using AXI Manager” on page 8-31

More About

- “Set Up AXI Manager” on page 3-2

Ethernet AXI Manager for Xilinx Zynq SoC Devices

Note Ethernet AXI master has been renamed to Ethernet AXI manager. In the software and documentation, the terms "manager" and "subordinate" replace "master" and "slave," respectively.

To implement HDL Verifier Support Package for Xilinx Zynq-based hardware features, you must configure the host computer and the hardware for proper communication. After you install the support package, follow these steps to manually set up the hardware.

Step 1. Complete Hardware Checklist

Confirm that you have all required hardware and accessories to complete the hardware setup.

- Gigabit Ethernet connection - This connection is often referred to as a network connection. You can use either an integrated network interface card (NIC) with a Gigabit Ethernet cable or a universal serial bus (USB) 3.0 Gigabit Ethernet adapter dongle. This connection is necessary for transmitting data, such as a programming file, from the host computer to the hardware. It is also necessary for sending and receiving signals to and from the hardware.
- SD card reader and writable SD card - If the host machine does not have an integrated card reader, use an external USB SD card reader.
- Supported hardware - This feature provides support to Xilinx Zynq-7000 ZC706 and Xilinx Zynq ZedBoard boards. Do not connect or turn on the device until you are prompted at a later step.
- Ethernet cable - This cable connects the hardware to the host.

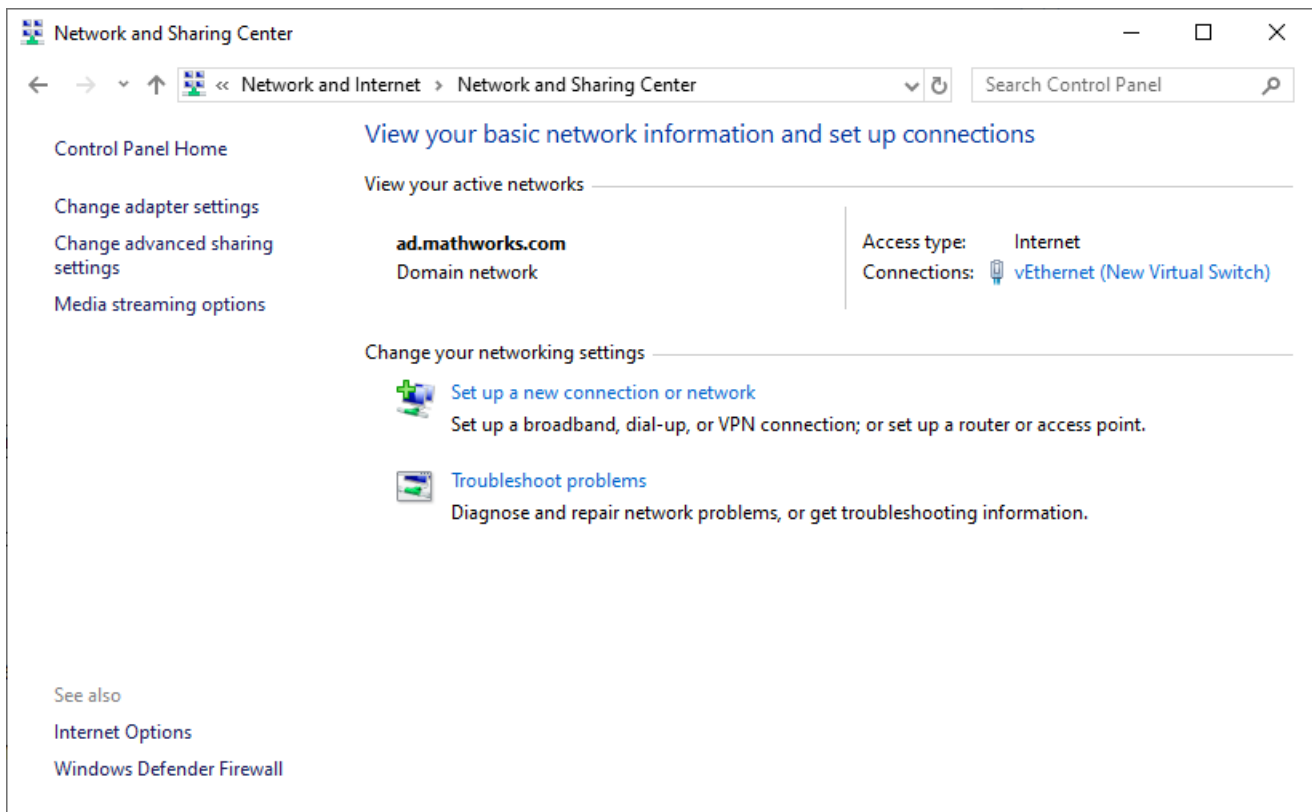
Step 2. Configure Host Computer

To connect the hardware to the host, you must configure an available network connection for the hardware on the host. Follow the steps for your specific operating system.

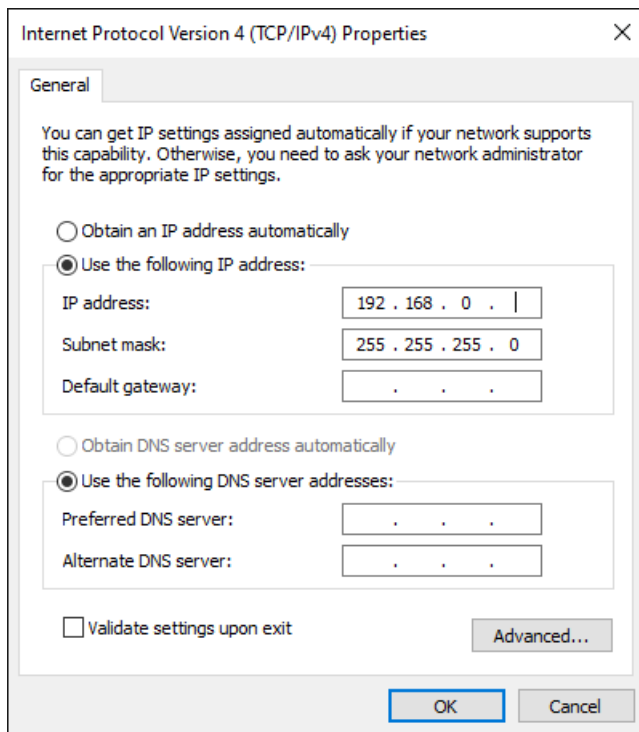
Configure Windows

Follow these instructions for Windows 7 or later.

- 1** From the **Start** menu, click **Control Panel**.
- 2** Set **View by** to **Category**.
- 3** Click **Network and Internet**.
- 4** Click **Network and Sharing Center**.
- 5** On the left pane, click **Change adapter settings**.



- 6 Right-click the local area network connection that is connected to the hardware and select **Properties**.
 - If an unused network connection is available, the local area connection appears as Unidentified network.
 - If you plan to repurpose your network connection, select the local area connection that you plan to use for the hardware.
 - If you have only one network connection, check if you can connect wireless to the existing local area network. You can use the network connection for the hardware.
 - You can use a pluggable USB to Gigabit Ethernet LAN adapter instead of a NIC.
- 7 On the **Networking** tab of the Properties dialog box, clear all options except **Internet Protocol Version 4 (TCP/IPv4)**. Other services, particularly antivirus software, can cause intermittent connection problems with the hardware.
- 8 Double-click **Internet Protocol Version 4 (TCP/IPv4)**.
- 9 On the **General** tab, select **Use the following IP address**.
- 10 The default IP address of the hardware is 192.168.0.2. The host network connection must be on the same subnet as the hardware. To meet this requirement, a compatible IP address must be assigned to the host network connection. Set the host network IP address to 192.168.0.x, where x is either 1 or an integer in the range [3, 255].



If the first three octets of the IP address field are not 192.168.0, then your hardware is on another subnet. Enter the same subnet number in the IP address.

- 11 Leave the subnet mask set to the default value of 255.255.255.0.
- 12 Click **OK**.

Configure Linux

Set the host Ethernet interface to have a static IP address. This configuration enables communication with the hardware. The default IP address of the hardware is 192.168.0.2. The host network connection must be on the same subnet as the hardware. To meet this requirement, you must assign a compatible IP address to the host network connection.

- 1 Set the host network IP address to 192.168.0.x, where x is either 1 or an integer in the range [3, 255]. Set this value by using the `ifconfig` command. For example, enter this command in the shell.

```
% sudo ifconfig ethZ 192.168.0.4 netmask 255.255.255.0
```

In this syntax, `ethZ` is the name of the host Ethernet port (usually `eth0`, `eth1`, and so on). To use the `sudo` command, you might have to enter a password.

- 2 Confirm the changes by entering this command in the shell.

```
% ifconfig ethZ
```

`ethZ` is the name of the host Ethernet port you set in the previous step.

Step 3. Copy Image to SD Card in Host System

You need an SD memory card that is configured with the firmware of this support package. The firmware includes the embedded software and the FPGA programming file necessary for using the

hardware as an I/O peripheral. If you have already copied the SD card with the required image, skip this step.

- 1 Insert a 4 GB or larger SD memory card into the memory card reader on the host computer.

Note If the SD card is lockable, you must unlock it first. If you use a lockable SD card adapter for a microSD card, you must unlock the card before inserting into the memory card reader.

- 2 Use the `copyImageToHostSDCardPath` function to copy the board-specific SD card image files to the specified SD card drive location in the host system. The SD card image files contain a bootloader and supported operating system information. This function also copies the server daemon for handling the AXI manager host commands on the target SoC device.

The default SD card image is the SD card files that come with the HDL Verifier Support Package. A custom SD card image is a user-created SD card files.

Examples

To copy a default SD card image to a specified location on the host computer on a Windows platform for a Xilinx Zynq-7000 ZC706 board with a default IP address, enter this code at the MATLAB command prompt.

```
copyImageToHostSDCardPath('ZC706','G:');
```

To copy a default SD card image to a specified location on the host computer for a Xilinx Zynq-7000 ZC706 board with a custom IP address and specify the gateway on a Linux platform, enter this code at the MATLAB command prompt. Set the host NIC address to `192.168.10.x`, where `x` must be an integer in the range [1, 3] or [5, 255].

```
copyImageToHostSDCardPath('ZC706','/media/username/261D-2F2B', ...
    'DeviceAddress','192.168.10.4','Gateway','192.168.10.1');
```

To copy a custom SD card image to a specified location on the host computer for a Xilinx Zynq-7000 ZC706 board with a default IP address on a Windows platform, enter this code at the MATLAB command prompt.

Before running this command, if your custom SD image is in a zipped format, such as `.zip`, `.tgz`, or `.gz`, extract zipped format files to a specific folder or directory in your host system.

```
copyImageToHostSDCardPath('ZC706','G:','SDCardImage', ...
    'C:\mywork\hdlv_prj\sdcard_image\zc706_sdcard_zynq7000');
```

Step 4. Update SD Card Image in SoC Device (Optional)

If you have already copied the SD card image files by using the process in “Step 3. Copy Image to SD Card in Host System” on page 3-18, skip this step.

Use the `loadImageToTargetSDCardPath` function to update the SD card image in the SoC device. Before proceeding with this step, ensure that the SD card is inserted in the target SoC device SD card location and that an Ethernet connection is established between the host system and the target SoC device.

This function updates the existing board-specific SD card image files in the SoC device. You might need to wait for at least 20 seconds for the SD card to update.

Examples

To copy a default SD card image to the target SoC device SD card location, enter this code at the MATLAB command prompt.

```
loadImageToTargetSDCardPath('ZC706');
```

To copy a custom SD card image to the target SoC device SD card location for a Xilinx Zynq-7000 ZC706 board with the default IP address on a Windows platform, enter this code at the MATLAB command prompt.

Before running this command, if your custom SD image is in a zipped format, such as .zip, .tgz, or .gz, extract zipped format files to a specific folder or directory in your host system.

```
loadImageToTargetSDCardPath('ZC706','SDCardImage', ...  
    'C:\mywork\hdlv_prj\sdcard_image\zc706_sdcard_zynq7000');
```

To copy a default SD card image to a specified location on the host computer for a Xilinx Zynq-7000 ZC706 board with an IP address that is different from the default value, enter this code at the MATLAB command prompt.

```
loadImageToTargetSDCardPath('ZC706', ...  
    'DeviceAddress','192.168.10.2','Gateway','192.168.10.1');
```

Step 5: Load Bitstream File to SoC Device (Optional)

Use the `loadBitstream` function only if you have any new FPGA design to load on the target SoC device. Otherwise, skip this step.

This function loads the custom FPGA bitstream file and its corresponding device tree blob (DTB) file to the target SoC device. You might need to wait for at least 20 seconds to get the changes updated to the target SoC device.

Examples

To load a custom FPGA bitstream and its corresponding DTB file to the target SoC device for a Xilinx Zynq-7000 ZC706 board with a default IP address, enter this code at the MATLAB command prompt.

```
loadBitstream('ZC706','C:\mywork\hdlv_bitstreams\system.bit', ...  
    'C:\mywork\hdlv_bitstreams\devicetree.dtb');
```

To load a custom FPGA bitstream and its corresponding DTB file to the target SoC device for a Xilinx Zynq-7000 ZC706 board with an IP address that is different from the default value, enter this code at the MATLAB command prompt.

```
loadBitstream('ZC706','C:\mywork\hdlv_bitstreams\system.bit', ...  
    'C:\mywork\hdlv_bitstreams\devicetree.dtb', ...  
    'DeviceAddress','192.168.10.2');
```

Once the program is running on your FPGA board, you can create an AXI manager object by using the `aximanager` object. To access the subordinate memory locations on the board, use the `readmemory` and `writememory` object functions of this object.

See Also

`copyImageToHostSDCardPath` | `loadImageToTargetSDCardPath` | `loadBitstream` | `aximanager`

Related Examples

- “Leverage Built-In Ethernet on Zynq to Perform Memory Access Using AXI Manager” on page 8-31

More About

- “Set Up AXI Manager” on page 3-2

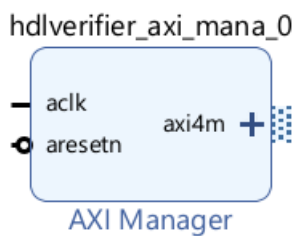
JTAG AXI Manager

Integrate and configure AXI manager over a JTAG connection. To use JTAG AXI manager, you must first include the AXI Manager intellectual property (IP) in your Xilinx Vivado project.

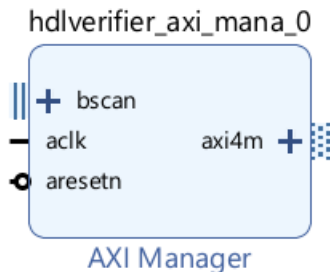
AXI Manager IP

The AXI Manager IP acts as a bridge that translates data between an AXI peripheral and MATLAB or Simulink software. This IP connects to your application IP over a JTAG connection.

This block diagram shows the interface of the AXI Manager IP.



This block diagram shows the interface of the AXI Manager IP for the Xilinx Versal devices.



AXI Manager IP Ports

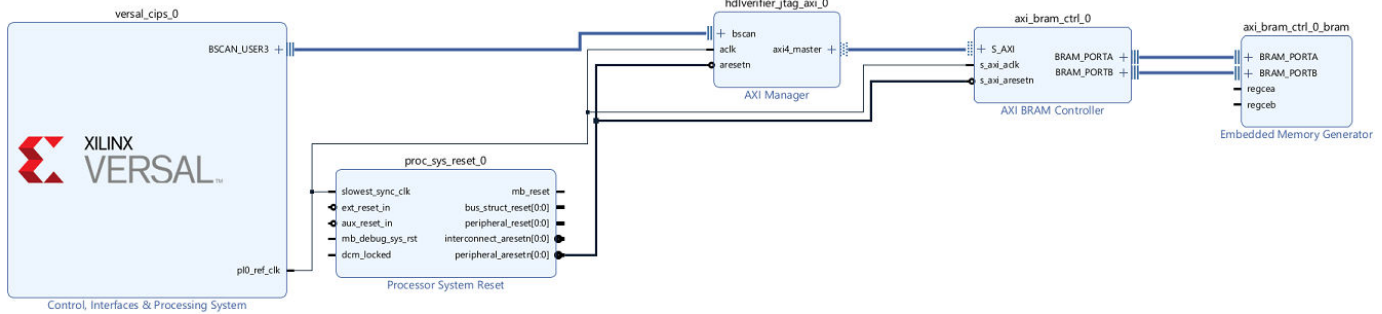
The interface of the AXI Manager IP includes the ports described in these tables.

Port	Description
bscan (Versal only)	BSCAN interface to connect to CIPS IP.
axi4m	AXI4-full manager interface.

Port	Direction	Description
aclk	Input	Clock signal for AXI4-full interface.
aresetn	Input	Active-low reset. Reset signal for AXI4-full interface.

AXI Manager IP Connections for Versal Devices

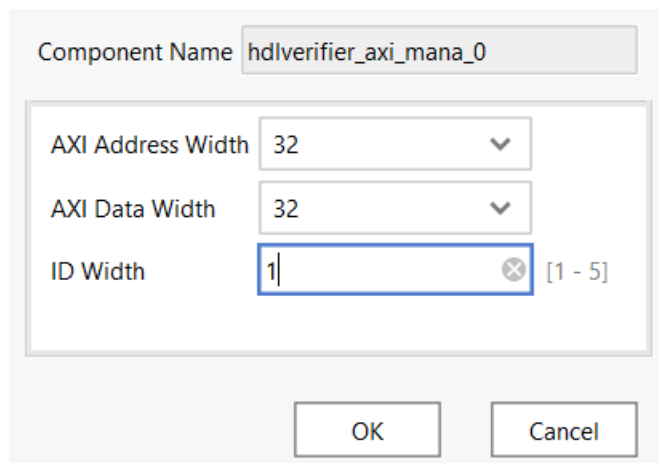
For a Xilinx Versal device, connect the AXI Manager IP to the BSCAN_USER3 interface of the Xilinx Versal platform CIPS IP, as this figure shows.



To enable the `BSCAN_USER3` interface, enable the PL `BSCAN2` interface in the CIPS IP. For more information about the CIPS IP, see *Control Interfaces and Processing System LogiCORE IP Product Guide* on the Xilinx website.

AXI Manager IP Parameters

After you include the AXI Manager IP in your design, open the block.



Configure these parameters:

- **AXI Address Width** — Address bus width in bits. The IP supports 32, 40, or 64 bits.
- **AXI Data Width** — Data bus width in bits. The IP supports 32 or 64 bits.
- **ID Width** — ID width in bits. The value of this parameter must match the ID width of the AXI4 subordinate.

When the program is running on your FPGA board, you can communicate with the AXI Manager IP by creating an `aximanager` object. To access the subordinate memory locations on the board, use the `readmemory` and `writememory` object functions.

See Also

`aximanager` | `readmemory` | `writememory`

Related Examples

- “Access FPGA Memory Using JTAG-Based AXI Manager” on page 8-15

More About

- “Set Up AXI Manager” on page 3-2

AXI Manager Simulation

AXI Manager Simulation

Note MATLAB AXI master has been renamed to AXI manager. In the software and documentation, the terms "manager" and "subordinate" replace "master" and "slave," respectively.

Prior to deploying your algorithm on an FPGA, you can simulate the algorithm and perform read and write operations to simulated memory or memory-mapped registers on your board. Use the provided SystemVerilog `readmemory` and `writememory` tasks to read and write memory-mapped locations on the board. Simulation is available for only the Vivado Simulator, on which you can perform only behavior simulations.

To integrate the AXI manager IP in your design, see “Integrate AXI Manager IP in FPGA Design” on page 3-2. After you integrate the AXI manager IP in your design, create a SystemVerilog wrapper that includes your design under test (DUT), memory interface generator (MIG), AXI manager IP, clock, and reset logic. You can then instantiate the wrapper in a SystemVerilog test bench and simulate.

HDL Wrapper Creation

Create a wrapper for the FPGA design. The wrapper includes all of the user logic and FPGA logic. The interface to the wrapper includes clock and reset ports and an optional DDR3 interface to the memory or other on-board peripherals. After creating and configuring the design in Vivado, navigate to the Sources window in Vivado. In the **Sources** tab, under **Design Sources**, right-click the design name and select **Create HDL Wrapper**. This action creates an HDL file named `design_name_wrapper.v` in the `design_name/hdl` folder, where `design_name` is the name of the Vivado project.

SystemVerilog Test Bench

HDL Verifier provides SystemVerilog tasks to interact with the design in a SystemVerilog simulation. After you create a SystemVerilog wrapper, instantiate the wrapper in a SystemVerilog test bench. The test bench also includes on-board IP models, such as memory models. Drive a clock and a reset signal to your DUT. To include definitions specific to HDL Verifier, import the `hdlverifier` package to the test bench by entering this code in the test bench file.

```
import hdlverifier::*;
```

To write data to memory locations or to read data from memory locations, use the `writememory` and `readmemory` SystemVerilog tasks, respectively, as detailed in the next two sections. For an example that uses AXI manager simulation, see “Access FPGA Memory Using JTAG-Based AXI Manager” on page 8-15.

writememory(addr,wdata,burst_type) SystemVerilog Task

Write data to AXI4 memory-mapped subordinates in simulation.

- `addr` - Start address for write operation

The address is zero-extended to 32 or 64 bits, depending on the AXI manager IP address width. The address must refer to an AXI subordinate memory location controlled by the AXI manager IP on your FPGA board. For more information, see “Memory Mapping Guidelines” on page 4-3.

- `wdata` - Data words to write

By default, the test bench writes data to a contiguous address block, incrementing the address for each operation. To turn off address increment mode and write each data value to the same location, set the `burst_type` parameter to `HDLV_AXI_BURST_TYPE_FIXED`.

- `burst_type` - AXI4 burst type

Specify `HDLV_AXI_BURST_TYPE_INCR` for increment mode, or `HDLV_AXI_BURST_TYPE_FIXED` for fixed-burst mode.

In `HDLV_AXI_BURST_TYPE_INCR` mode, the AXI manager reads a vector of data from contiguous memory locations, starting with the specified address. In `HDLV_AXI_BURST_TYPE_FIXED` mode, the AXI manager reads all of the data from the same address.

readmemory(addr,length,burst_type) SystemVerilog Task

Read data out of AXI4 memory-mapped subordinates in simulation.

- `addr` - Start address for read operation

The address is zero-extended to 32 or 64 bits, depending on the AXI manager IP address width. By default, the address width is set to 32 bits. To set the width to 64 bits, specify parameter `AXI_ADDR_WIDTH=64` in the HDL test bench. The address must refer to an AXI subordinate memory location controlled by the AXI manager IP on your FPGA board. For more information, see “Memory Mapping Guidelines” on page 4-3.

- `length` - Number of locations to read

Specify the number of memory locations to read. By default, the test bench reads from a contiguous address block, incrementing the address for each operation. To turn off address increment mode and read repeatedly from the same location, set the `burst_type` parameter to `HDLV_AXI_BURST_TYPE_FIXED`.

- `burst_type` - AXI4 burst type

Specify `HDLV_AXI_BURST_TYPE_INCR` for increment mode, or `HDLV_AXI_BURST_TYPE_FIXED` for fixed-burst mode.

In `HDLV_AXI_BURST_TYPE_INCR` mode, the AXI manager reads a vector of data from contiguous memory locations, starting with the specified address. In `HDLV_AXI_BURST_TYPE_FIXED` mode, the AXI manager reads all of the data from the same address.

Memory Mapping Guidelines

- If the AXI manager IP data width is 32 bits, the memory is 4 bytes aligned, and addresses have 4-byte increments (`0x0`, `0x4`, `0x8`). In this case, `0x1` is an invalid address and you get an error.
- If the AXI manager IP data width is 64 bits, the memory is 8 bytes aligned, and addresses have 8-byte increments (`0x0`, `0x8`, `0x10`). In this case, `0x1` or `0x4` are invalid addresses and will emit an error.
- If the AXI manager IP data width is 32 bits and the **Burst type** parameter is set to **Increment**, the address is incremented by 4 bytes.
- If the AXI manager IP data width is 64 bits and the **Burst type** parameter is set to **Increment**, the address is incremented by 8 bytes.

- Do not use a 64-bit AXI manager for accessing 32-bit registers.

See Also

More About

- “Integrate AXI Manager IP in FPGA Design” on page 3-2
- “Access FPGA Memory Using JTAG-Based AXI Manager” on page 8-15

AXI Manager Reference

aximanager

Read and write memory locations on FPGA board from MATLAB

Description

The `aximanager` object communicates with the AXI manager IP when it is running on an FPGA board. The object forwards read and write commands to the IP to access subordinate memory locations on the FPGA board. Before using this object, follow the steps in “Set Up AXI Manager” on page 3-2.

Note The `aximaster` object has been renamed to the `aximanager` object. For more information, see “Compatibility Considerations” on page 5-7.

Creation

Syntax

```
mem = aximanager(vendor)
mem = aximanager(vendor,Name,Value)
```

Description

`mem = aximanager(vendor)` returns an object, that controls an AXI4 manager IP for the FPGA that is running on your board. `vendor` specifies the FPGA brand name. This connection enables you to access memory locations in an SoC design from MATLAB.

`mem = aximanager(vendor,Name,Value)` sets properties using one or more name-value pair arguments. Enclose each property name and value in quotes. For example, `'DeviceAddress','192.168.0.10'` specifies the internet protocol (IP) address of the FPGA board as 192.168.0.10.

Input Arguments

vendor — FPGA brand name

`'Intel' | 'Xilinx'`

FPGA brand name, specified as `'Intel'` or `'Xilinx'`. This value specifies the manufacturer of the FPGA board. The AXI manager IP varies depending on the type of FPGA that you specify.

Properties

Interface — Type of interface used for communication with FPGA board

`'JTAG' (default) | 'PCIe' | 'UDP'`

Type of interface used for communication with the FPGA board, specified as `'JTAG'` (default), `'PCIe'`, or `'UDP'`. This value specifies the interface type for communicating between the host and the FPGA.

JTAGCableType — Type of JTAG cable used for communication with FPGA board

'auto' (default) | 'FTDI'

Type of JTAG cable used for communication with the FPGA board (Xilinx boards only), specified as 'auto' (default) or 'FTDI'. This value specifies the type of JTAG cable used for communication with the FPGA board. This property is most useful when more than one cable is connected to the host computer.

When this property is set to 'auto' (default), the object autodetects the JTAG cable type. The object prioritizes searching for Digilent cables and uses this process to autodetect the cable type.

- 1 The `aximanager` object searches for a Digilent cable. If the object finds:
 - Exactly one Digilent cable, it uses that cable for communication with the FPGA board.
 - More than one Digilent cable, it returns an error. To resolve this error, specify the desired cable using `JTAGCableName`.
 - No Digilent cables, it searches for an FTDI cable.
- 2 If no Digilent cable is found, the `aximanager` object then searches for an FTDI cable. If the object finds:
 - Exactly one FTDI cable, it uses that cable for communication with the FPGA board.
 - More than one FTDI cable, it returns an error. To resolve this error, specify the desired cable using `JTAGCableName`.
 - No FTDI cables, it returns an error. To resolve this error, connect a Digilent or FTDI cable.
- 3 If the object finds two cables of different types, it prioritizes the Digilent cable. To use an FTDI cable, set this property to 'FTDI'.

When this property is set to 'FTDI', the object searches for FTDI cables. If the object finds:

- Exactly one FTDI cable, it uses that cable for communication with the FPGA board.
- More than one FTDI cable, it returns an error. To resolve this error, specify the desired cable using `JTAGCableName`.
- No FTDI cables, it returns an error. To resolve this error, connect a Digilent or FTDI cable.

For more details, see “Select from Multiple JTAG Cables” on page 5-6.

Dependencies

To enable this property, set the `vendor` input to 'Xilinx'.

JTAGCableName — Name of JTAG cable used for communication with FPGA board

'auto' (default) | character vector | string scalar

Name of the JTAG cable used for communication with the FPGA board, specified as a character vector or string scalar representing a JTAG cable name. Specify this property if more than one JTAG cable of the same type are connected to the host computer. If the host computer has more than one JTAG cable and you do not specify this property, the object returns an error. The error message contains the names of the available JTAG cables. For more details, see “Select from Multiple JTAG Cables” on page 5-6.

Data Types: `char` | `string`

DeviceAddress — IP address of Ethernet port on FPGA board

'192.168.0.2' (default) | character vector | string scalar

Internet protocol (IP) address of the Ethernet port on the FPGA board, specified as a character vector or string scalar representing an IP address.

Example: '192.168.0.10'

Dependencies

To enable this property, set the Interface to 'UDP'.

Data Types: char | string

DeviceType — Type of target device

'FPGA' (default) | 'SoC'

Type of target device, specified as 'FPGA' (default) or 'SoC'. When you are using a Xilinx Zynq or an Intel SoC as a target device, specify this property as 'SoC'.

Example: 'SoC'

Port — UDP port number of target FPGA board

'50101' (default) | integer

User datagram protocol (UDP) port number of the target FPGA board, specified as an integer.

Example: '12345'

Dependencies

To enable this property, set the Interface property to 'UDP' and the DeviceType property to 'FPGA'.

Data Types: uint16

TckFrequency — JTAG clock frequency

15 (default) | positive integer

Specify the JTAG clock frequency (Xilinx boards only), specified as a positive integer. Units are in MHz. The JTAG frequency depends on the type of cable and the maximum clock frequency supported by the FPGA board. Check the board documentation for maximum frequency.

Dependencies

To enable this property, set the vendor input to 'Xilinx'.

JTAGChainPosition — Position of FPGA in JTAG chain

0 (default) | nonnegative integer

Position of the FPGA in the JTAG chain (Xilinx boards only), specified as a nonnegative integer. Specify this property value if more than one FPGA or Zynq device is on the JTAG chain.

Dependencies

To enable this property, set the vendor input to 'Xilinx'.

IRLengthBefore — Sum of instruction register length for all devices before target FPGA

0 (default) | nonnegative integer

Sum of instruction register length for all devices before target FPGA (Xilinx boards only), specified as a nonnegative integer. Specify this property value if more than one FPGA or Zynq device is on the JTAG chain.

Dependencies

To enable this property, set the vendor input to 'Xilinx'.

IRLengthAfter — Sum of instruction register length for all devices after target FPGA

0 (default) | nonnegative integer

Sum of instruction register length for all devices after target FPGA (Xilinx boards only), specified as a nonnegative integer. Specify this property value if more than one FPGA or Zynq device is on the JTAG chain.

Dependencies

To enable this property, set the vendor input to 'Xilinx'.

Object Functions

readmemory	Read data out of AXI4 memory-mapped subordinates
release	Release JTAG or Ethernet cable resource
writememory	Write data to AXI4 memory-mapped subordinates

Examples

Access Memory on FPGA Board from MATLAB

This example shows how to read and write the memory locations on a Xilinx® FPGA board from MATLAB®.

Before you can use this example, you must have a design running on an FPGA board connected to the MATLAB host machine. The FPGA design must include an AXI manager IP that is customized for your FPGA vendor. The support package installation includes this IP. To include the IP in your project, see the “Access FPGA Memory Using JTAG-Based AXI Manager” on page 8-15 example.

Create an AXI manager object. The object connects MATLAB with the FPGA board and confirms that the IP is present.

```
mem = aximanager('Xilinx')
mem =
    aximanager with properties:
        Vendor: 'Xilinx'
        JTAGCableName: 'auto'
```

Write 10 addresses and then read data from a single location. By default, these functions auto-increment the address for each word of data.

```
writememory(mem,140,[10:19]);
rd_d = readmemory(mem,140,1)
rd_d =
```

```
uint32
```

```
10
```

Read data from 10 locations.

```
rd_d = readmemory(mem,140,10)
```

```
rd_d =
```

```
1x10 uint32 row vector
```

```
10 11 12 13 14 15 16 17 18 19
```

Read data 10 times from the same address by specifying that the AXI manager read all data from the same address (disabling auto-incrementation).

```
rd_d = readmemory(mem,140,10,'BurstType','Fixed')
```

```
rd_d =
```

```
1x10 uint32 row vector
```

```
10 10 10 10 10 10 10 10 10 10
```

Write data 10 times to the same address. In this case, the final value stored in address 140 is 29.

```
writememory(mem,140,[20:29],'BurstType','Fixed');
```

```
rd_d = readmemory(mem,140,10)
```

```
rd_d =
```

```
1x10 uint32 row vector
```

```
29 11 12 13 14 15 16 17 18 19
```

Specify the address as a hexadecimal value. Specify for the function to cast the read data to a data type other than uint32.

```
writememory(mem,0x1c,[0:4:64]);
```

```
rd_d = readmemory(mem,0x1c,16,'OutputDataType',numerictype(0,6,4))
```

```
rd_d =
```

```
Columns 1 through 10
```

```
0 0.2500 0.5000 0.7500 1.0000 1.2500 ...
1.5000 1.7500 2.0000 2.2500
```

```
Columns 11 through 16
```

```
2.5000 2.7500 3.0000 3.2500 3.5000 3.7500
```

```
DataTypeMode: Fixed-point: binary point scaling
Signedness: Unsigned
WordLength: 6
FractionLength: 4
```

When you no longer need to access the board, release the JTAG connection.

```
release(mem);
```

Select from Multiple JTAG Cables

This example shows how to select the required JTAG cable from the multiple JTAG cables that are connected to your host computer.

When multiple JTAG cables are connected to your host computer, the object prioritizes searching for Digilent® cables over FTDI cables. To use an FTDI cable, specify the JTAG cable type.

```
h = aximanager('Xilinx', 'JTAGCableType', 'FTDI');
```

If two cables of the same type are connected to your host computer, specify the “JTAGCableName” on page 5-0 property identifier for the board where the AXI manager IP is running. To see the JTAG cable identifiers, attempt to create an `aximanager` object. The object returns a list of the current JTAG cable names.

```
h = aximanager('Xilinx')
```

```
Error using fpgadebug_mex
Found more than one JTAG cable:
0 (JtagSmt1): #tpt_0001#ptc_0002#210203991642
1 (Arty): #tpt_0001#ptc_0002#210319789795
Please disconnect the extra cable, or specify the cable name as an
input argument. See documentation of FPGA Data Capture or AXI Manager
to learn how to set the cable name.
```

To communicate with the Arty board, specify the matching JTAG cable name.

```
h = aximanager('Xilinx', 'JTAGCableName', '#tpt_0001#ptc_0002#210319789795');
```

Version History

Introduced in R2017a

R2022a: aximaster renamed to aximanager

Warns starting in R2022a

The `aximaster` object has been renamed to the `aximanager` object. In the software and documentation, the terms “manager” and “subordinate” replace “master” and “slave,” respectively.

To create an AXI manager object, use the `aximanager` object. Using the `aximaster` object is not recommended and will be removed in a future release. If you use the `aximaster` object, the object now gives this warning message.

`aximaster` has been renamed to `aximanager` in R2022a. `aximaster` will be removed in a future release. Use `aximanager` instead.

See Also

Topics

“Access FPGA Memory Using JTAG-Based AXI Manager” on page 8-15

“Access DUT Registers on Xilinx Pure FPGA Board Using IP Core Generation Workflow” (HDL Coder)

“Set Up AXI Manager” on page 3-2

readmemory

Package: hdlverifier

Read data out of AXI4 memory-mapped subordinates

Syntax

```
data = readmemory(mem,addr,size)
data = readmemory(mem,addr,size,Name,Value)
```

Description

`data = readmemory(mem,addr,size)` reads `size` locations of data, starting from the address specified in `addr` and then incrementing the address for each word. The function typecasts the data to the `uint32` or `uint64` data type (depending on the data size of the AXI manager IP). The address, `addr`, must refer to an AXI subordinate memory location controlled by the AXI manager IP on your FPGA board. The `aximanager` object, `mem`, manages the connection between MATLAB and the AXI manager IP.

`data = readmemory(mem,addr,size,Name,Value)` specifies options using one or more name-value arguments.

Examples

Access Memory on FPGA Board from MATLAB

This example shows how to read and write the memory locations on a Xilinx® FPGA board from MATLAB®.

Before you can use this example, you must have a design running on an FPGA board connected to the MATLAB host machine. The FPGA design must include an AXI manager IP that is customized for your FPGA vendor. The support package installation includes this IP. To include the IP in your project, see the “Access FPGA Memory Using JTAG-Based AXI Manager” on page 8-15 example.

Create an AXI manager object. The object connects MATLAB with the FPGA board and confirms that the IP is present.

```
mem = aximanager('Xilinx')
mem =
    aximanager with properties:
        Vendor: 'Xilinx'
        JTAGCableName: 'auto'
```

Write 10 addresses and then read data from a single location. By default, these functions auto-increment the address for each word of data.

```
writememory(mem,140,[10:19]);
rd_d = readmemory(mem,140,1)
```

```
rd_d =
    uint32
    10
```

Read data from 10 locations.

```
rd_d = readmemory(mem,140,10)
```

```
rd_d =
    1x10 uint32 row vector
    10  11  12  13  14  15  16  17  18  19
```

Read data 10 times from the same address by specifying that the AXI manager read all data from the same address (disabling auto-incrementation).

```
rd_d = readmemory(mem,140,10,'BurstType','Fixed')
```

```
rd_d =
    1x10 uint32 row vector
    10  10  10  10  10  10  10  10  10  10
```

Write data 10 times to the same address. In this case, the final value stored in address 140 is 29.

```
writememory(mem,140,[20:29],'BurstType','Fixed');
```

```
rd_d = readmemory(mem,140,10)
```

```
rd_d =
    1x10 uint32 row vector
    29  11  12  13  14  15  16  17  18  19
```

Specify the address as a hexadecimal value. Specify for the function to cast the read data to a data type other than uint32.

```
writememory(mem,0x1c,[0:4:64]);
```

```
rd_d = readmemory(mem,0x1c,16,'OutputDataType',numeric(0,6,4))
```

```
rd_d =
    Columns 1 through 10
           0    0.2500    0.5000    0.7500    1.0000    1.2500 ...
           1.5000    1.7500    2.0000    2.2500
    Columns 11 through 16
           2.5000    2.7500    3.0000    3.2500    3.5000    3.7500

    DataTypeMode: Fixed-point: binary point scaling
    Signedness: Unsigned
    WordLength: 6
    FractionLength: 4
```

When you no longer need to access the board, release the JTAG connection.

```
release(mem);
```

Input Arguments

mem — Connection to AXI manager IP on FPGA board

`aximanager` object

Connection to the AXI manager IP on your FPGA board, specified as an `aximanager` object.

addr — Starting address for read operation

nonnegative integer multiple of 4 | nonnegative hexadecimal value multiple of 4

Starting address for the read operation, specified as a nonnegative integer multiple of 4 or hexadecimal value multiple of 4. The function supports the address width of 32, 40, and 64 bits. The function casts the address to the `uint32` or `uint64` data type, depending on the AXI manager IP address width. The address must refer to an AXI subordinate memory location controlled by the AXI manager IP on your FPGA board.

Memory-Mapping Guidelines

- If the AXI manager IP data width is 32 bits, the memory is 4 bytes aligned, and addresses have 4-byte increments (`0x0`, `0x4`, `0x8`). In this case, `0x1` is an illegal address and emits an error.
- If the AXI manager IP data width is 64 bits, the memory is 8 bytes aligned, and addresses have 8-byte increments (`0x0`, `0x8`, `0x10`). In this case, `0x1` and `0x4` are illegal and emit errors.
- If the AXI manager IP data width is 32 bits and you set the `'BurstType'` argument to `'Increment'`, the address has 4-byte increments.
- If the AXI manager IP data width is 64 bits and you set the `'BurstType'` argument to `'Increment'`, the address has 8-byte increments.
- If the AXI manager IP data width is 32 bits and you set the `OutputDataType` argument to `'half'`, the function reads the lower 2 bytes and ignores the higher 2 bytes.
- If the AXI manager IP data width is 64 bits and you set the `OutputDataType` argument to `'half'`, the function reads the lower 2 bytes and ignores the higher 6 bytes.
- Do not use a 64-bit AXI manager for accessing 32-bit registers.

Example: `0xa4`, specifies a starting address of `0xa4`.

Data Types: `uint32` | `uint64`

size — Number of memory locations to read

nonnegative integer

Number of memory locations to read, specified as a nonnegative integer. By default, the function reads data from a contiguous address block, incrementing the address for each operation. To disable address incrementation and read repeatedly from the same location, set the `'BurstType'` argument to `'Fixed'`.

When you specify a large operation size, such as reading a block of DDR memory, the function automatically breaks the operation into multiple bursts, using the maximum supported burst size of 256 words.

Example: `5` specifies five contiguous memory locations.

Name-Value Pair Arguments

Specify optional pairs of arguments as `Name1=Value1, . . . , NameN=ValueN`, where `Name` is the argument name and `Value` is the corresponding value. Name-value arguments must appear after other arguments, but the order of the pairs does not matter.

Before R2021a, use commas to separate each name and value, and enclose `Name` in quotes.

Example: `'BurstType', 'Fixed'` directs the AXI manager to read all data from the same address.

BurstType — AXI4 burst type

`'Increment'` (default) | `'Fixed'`

AXI4 burst type, specified as one of these options:

- `'Increment'` — The AXI manager reads a vector of data from contiguous memory locations, starting with the specified address.
- `'Fixed'` — The AXI manager reads all data from the same address.

Note The `'Fixed'` burst type is not supported for the PCI Express interface. Use the `'Increment'` burst type instead.

OutputDataType — Data type assigned to read data

`'uint32'` (default) | `'uint8'` | `'int8'` | `'uint16'` | `'int16'` | `'half'` | `'int32'` | `'single'` | `'uint64'` | `'int64'` | `'double'` | numeric type object

Data type assigned to the read data, specified as one of these options:

- `'int8'`
- `'uint8'`
- `'uint16'`
- `'int16'`
- `'half'`
- `'uint32'`
- `'int32'`
- `'single'`
- `'uint64'`
- `'int64'`
- `'double'`
- numeric type object

The function typecasts the data read out of the FPGA to the specified data type. `double` is supported for 64-bit UDP connections only.

Output Arguments

data — Read data

scalar | vector

Read data, returned as a scalar or vector depending on the value you specify for the `size`. The function typecasts the data to the data type specified by the `'OutputDataType'` input.

Version History

Introduced in R2017a

R2023a: Support for half data type

The function reads `half` data from the memory locations on the FPGA board. The address for the read operation must refer to an AXI subordinate memory location controlled by the AXI manager IP on your FPGA board.

- If the AXI manager IP data width is 32 bits, the memory is 4 bytes aligned, and addresses have 4-byte increments (0x0, 0x4, 0x8). In this case, the function reads the lower 2 bytes and ignores the higher 2 bytes.
- If the AXI manager IP data width is 64 bits, the memory is 8 bytes aligned, and addresses have 8-byte increments (0x0, 0x8, 0x10). In this case, the function reads the lower 2 bytes and ignores the higher 6 bytes.

See Also

`writememory` | `aximanager`

release

Package: hdlverifier

Release JTAG or Ethernet cable resource

Syntax

```
release(mem)
```

Description

`release(mem)` releases the JTAG cable or Ethernet cable resource, depending on the interface that you use.

- When you use a JTAG interface, the function releases the JTAG cable resource, freeing the cable for use to reprogram the FPGA. After initialization, the AXI manager object, `mem`, holds the JTAG cable resource, and other programs cannot access the JTAG cable. While you have an active AXI manager object, FPGA programming over JTAG fails. Call the `release` function before reprogramming the FPGA.
- When you use an Ethernet interface, the function closes the Ethernet communications channel and clears the associated resources. During the creation of AXI manager object `mem`, the object initializes a communication stream to enable the exchange of data between the host computer and the target processor. Call the `release` function when you no longer need to access the board.

Examples

Access Memory on FPGA Board from MATLAB

This example shows how to read and write the memory locations on a Xilinx® FPGA board from MATLAB®.

Before you can use this example, you must have a design running on an FPGA board connected to the MATLAB host machine. The FPGA design must include an AXI manager IP that is customized for your FPGA vendor. The support package installation includes this IP. To include the IP in your project, see the “Access FPGA Memory Using JTAG-Based AXI Manager” on page 8-15 example.

Create an AXI manager object. The object connects MATLAB with the FPGA board and confirms that the IP is present.

```
mem = aximanager('Xilinx')  
mem =  
    aximanager with properties:  
        Vendor: 'Xilinx'  
        JTAGCableName: 'auto'
```

Write 10 addresses and then read data from a single location. By default, these functions auto-increment the address for each word of data.

```
writememory(mem,140,[10:19]);
rd_d = readmemory(mem,140,1)
```

```
rd_d =
    uint32
    10
```

Read data from 10 locations.

```
rd_d = readmemory(mem,140,10)
rd_d =
```

```
1x10 uint32 row vector
    10    11    12    13    14    15    16    17    18    19
```

Read data 10 times from the same address by specifying that the AXI manager read all data from the same address (disabling auto-incrementation).

```
rd_d = readmemory(mem,140,10,'BurstType','Fixed')
rd_d =
```

```
1x10 uint32 row vector
    10    10    10    10    10    10    10    10    10    10
```

Write data 10 times to the same address. In this case, the final value stored in address 140 is 29.

```
writememory(mem,140,[20:29],'BurstType','Fixed');
rd_d = readmemory(mem,140,10)
```

```
rd_d =
    1x10 uint32 row vector
    29    11    12    13    14    15    16    17    18    19
```

Specify the address as a hexadecimal value. Specify for the function to cast the read data to a data type other than uint32.

```
writememory(mem,0x1c,[0:4:64]);
rd_d = readmemory(mem,0x1c,16,'OutputDataType',numeric(0,6,4))
```

```
rd_d =
    Columns 1 through 10
         0    0.2500    0.5000    0.7500    1.0000    1.2500 ...
         1.5000    1.7500    2.0000    2.2500
    Columns 11 through 16
         2.5000    2.7500    3.0000    3.2500    3.5000    3.7500

    DataTypeMode: Fixed-point: binary point scaling
    Signedness: Unsigned
    WordLength: 6
    FractionLength: 4
```

When you no longer need to access the board, release the JTAG connection.

```
release(mem);
```

Input Arguments

mem — **Connection to JTAG-based AXI manager IP or to Ethernet-based AXI manager IP**
aximanager object

Connection to a JTAG-based AXI manager IP or to an Ethernet-based AXI manager IP, specified as an aximanager object.

Version History

Introduced in R2017a

See Also

readmemory | writememory

writememory

Package: hdlverifier

Write data to AXI4 memory-mapped subordinates

Syntax

```
writememory(mem, addr, data)
writememory(mem, addr, data, Name, Value)
```

Description

`writememory(mem, addr, data)` writes all words specified in `data`, starting from the address specified in `addr` and then incrementing the address for each word. The address, `addr`, must refer to an AXI subordinate memory location controlled by the AXI manager IP on your FPGA board. The AXI manager object, `mem`, manages the connection between MATLAB and the AXI manager IP.

`writememory(mem, addr, data, Name, Value)` specifies options using one or more name-value arguments.

Examples

Access Memory on FPGA Board from MATLAB

This example shows how to read and write the memory locations on a Xilinx® FPGA board from MATLAB®.

Before you can use this example, you must have a design running on an FPGA board connected to the MATLAB host machine. The FPGA design must include an AXI manager IP that is customized for your FPGA vendor. The support package installation includes this IP. To include the IP in your project, see the “Access FPGA Memory Using JTAG-Based AXI Manager” on page 8-15 example.

Create an AXI manager object. The object connects MATLAB with the FPGA board and confirms that the IP is present.

```
mem = aximanager('Xilinx')
mem =
    aximanager with properties:
        Vendor: 'Xilinx'
        JTAGCableName: 'auto'
```

Write 10 addresses and then read data from a single location. By default, these functions auto-increment the address for each word of data.

```
writememory(mem, 140, [10:19]);
rd_d = readmemory(mem, 140, 1)
rd_d =
```

```
uint32
```

```
10
```

Read data from 10 locations.

```
rd_d = readmemory(mem,140,10)
```

```
rd_d =
```

```
1x10 uint32 row vector
```

```
10 11 12 13 14 15 16 17 18 19
```

Read data 10 times from the same address by specifying that the AXI manager read all data from the same address (disabling auto-incrementation).

```
rd_d = readmemory(mem,140,10,'BurstType','Fixed')
```

```
rd_d =
```

```
1x10 uint32 row vector
```

```
10 10 10 10 10 10 10 10 10 10
```

Write data 10 times to the same address. In this case, the final value stored in address 140 is 29.

```
writememory(mem,140,[20:29],'BurstType','Fixed');
```

```
rd_d = readmemory(mem,140,10)
```

```
rd_d =
```

```
1x10 uint32 row vector
```

```
29 11 12 13 14 15 16 17 18 19
```

Specify the address as a hexadecimal value. Specify for the function to cast the read data to a data type other than uint32.

```
writememory(mem,0x1c,[0:4:64]);
```

```
rd_d = readmemory(mem,0x1c,16,'OutputDataType',numeric(0,6,4))
```

```
rd_d =
```

```
Columns 1 through 10
```

```
0 0.2500 0.5000 0.7500 1.0000 1.2500 ...
1.5000 1.7500 2.0000 2.2500
```

```
Columns 11 through 16
```

```
2.5000 2.7500 3.0000 3.2500 3.5000 3.7500
```

```
DataTypeMode: Fixed-point: binary point scaling
Signedness: Unsigned
WordLength: 6
FractionLength: 4
```

When you no longer need to access the board, release the JTAG connection.

```
release(mem);
```

Input Arguments

mem — Connection to AXI manager IP on FPGA board

`aximanager` object

Connection to the AXI manager IP on your FPGA board, specified as an `aximanager` object.

addr — Starting address for write operation

nonnegative integer multiple of 4 | nonnegative hexadecimal value multiple of 4

Starting address for the write operation, specified as a nonnegative integer multiple of 4 or hexadecimal value multiple of 4. The function supports the address width of 32, 40, and 64 bits. The function casts the address to the `uint32` or `uint64` data type, according to the AXI manager IP address width. The address must refer to an AXI subordinate memory location controlled by the AXI manager IP on your FPGA board.

Memory-Mapping Guidelines

- If the AXI manager IP data width is 32 bits, the memory is 4 bytes aligned, and addresses have 4-byte increments (0x0, 0x4, 0x8). In this case, 0x1 is an illegal address and emits an error.
- If the AXI manager IP data width is 64 bits, the memory is 8 bytes aligned, and addresses have 8-byte increments (0x0, 0x8, 0x10). In this case, 0x1 and 0x4 are illegal and emit errors.
- If the AXI manager IP data width is 32 bits and you set the 'BurstType' argument to 'Increment', the address has 4-byte increments.
- If the AXI manager IP data width is 64 bits and you set the 'BurstType' argument to 'Increment', the address has 8-byte increments.
- If the AXI manager IP data width is 32 bits and the input data is `half`, the function writes data to the lower 2 bytes and pads the higher 2 bytes with zeros.
- If the AXI manager IP data width is 64 bits and the input data is `half`, the function writes data to the lower 2 bytes and pads the higher 6 bytes with zeros.
- Do not use a 64-bit AXI manager for accessing 32-bit registers.

Example: 64, specifies a starting address of 64.

Data Types: `uint32` | `uint64`

data — Data words to write

`scalar` | `vector`

Data words to write, specified as a scalar or vector. By default, the function writes the data to a contiguous address block, incrementing the address for each operation. To disable address incrementation and write each data value to the same location, set the 'BurstType' argument to 'Fixed'.

Before sending the write request to the FPGA, the function typecasts the input data to the `uint32`, `int32`, `uint64`, or `int64` data type. The type conversion follows these rules:

- If the input data is `double`, then the data is typecast to `int32` or `int64`, depending on the AXI manager IP data width.

- If the input data is `single`, then the data is typecast to `uint32` or `uint64`, depending on the AXI manager IP data width.
- If the input data is `half`, then the data is typecast to `uint16` and packed to `uint32` or `uint64`, depending on the AXI manager IP data width.
- If the bit width of the input data type is less than the AXI manager IP data width, then the data is sign-extended to the width of the AXI manager IP data width.
- If the bit width of the input data type is greater than the AXI manager IP data width, then the data is typecast to `int32`, `uint32`, `int64`, `uint64`. The data is typecast to match the AXI manager IP data width and the signedness of the original data type.
- If the input data is a fixed-point data type, then the function writes the stored integer value of the data.

When you specify a large operation size, such as writing a block of DDR memory, the function automatically breaks the operation into multiple bursts, using the maximum supported burst size of 256 words.

Example: `[1:100]` specifies 100 contiguous memory locations.

Data Types: `uint8` | `int8` | `uint16` | `int16` | `half` | `uint32` | `int32` | `single` | `uint64` | `int64` | `double` | `fi`

Name-Value Pair Arguments

Specify optional pairs of arguments as `Name1=Value1, ..., NameN=ValueN`, where `Name` is the argument name and `Value` is the corresponding value. Name-value arguments must appear after other arguments, but the order of the pairs does not matter.

Before R2021a, use commas to separate each name and value, and enclose Name in quotes.

Example: `'BurstType', 'Fixed'` directs the AXI manager to write all data to the same address.

BurstType — AXI4 burst type

`'Increment'` (default) | `'Fixed'`

AXI4 burst type, specified as one of these options:

- `'Increment'` — The AXI manager writes a vector of data to contiguous memory spaces, starting with the specified address.
- `'Fixed'` — The AXI manager writes all data to the same address.

Note The `'Fixed'` burst type is not supported for the PCI Express interface. Use the `'Increment'` burst type instead.

Version History

Introduced in R2017a

R2023a: Support for half data type

The function writes `half` data to the memory locations on the FPGA board. Before sending the write request to the FPGA, the function typecasts the `half` input data to the `uint16` and then packs the data to `uint32` or `uint64`, depending on the AXI manager IP data width.

The address for the write operation must refer to an AXI subordinate memory location controlled by the AXI manager IP on your FPGA board.

- If the AXI manager IP data width is 32 bits, the memory is 4 bytes aligned, and addresses have 4-byte increments (0x0, 0x4, 0x8). In this case, the function writes data to the lower 2 bytes and pads the higher 2 bytes with zeros.
- If the AXI manager IP data width is 64 bits, the memory is 8 bytes aligned, and addresses have 8-byte increments (0x0, 0x8, 0x10). In this case, the function writes data to the lower 2 bytes and pads the higher 6 bytes with zeros.

See Also

`readmemory` | `aximanager`

setupAXIManagerForVivado

Add AXI manager IP path to Vivado project

Syntax

```
setupAXIManagerForVivado(projectName)  
setupAXIManagerForVivado
```

Description

setupAXIManagerForVivado(projectName) adds the AXI manager IP folder to the path of the Vivado project, projectName.

Note The setupAXIMasterForVivado function has been renamed to the setupAXIManagerForVivado function. For more information, see “Compatibility Considerations” on page 5-22.

setupAXIManagerForVivado displays the location of the AXI manager IP that is included with this support package.

Examples

Add AXI Manager IP to FPGA Project

Call the setup function with a project file name. If the project is not in the current working folder, include the path.

```
setupAXIManagerForVivado('arty.xpr')
```

The location of the AXI manager IP is added to your project IP search path.

Find Location of AXI Manager IP

To find the folder in your MATLAB® installation that contains the AXI manager IP, call the setup function without a project file name.

```
setupAXIManagerForVivado
```

```
The Vivado project was not specified. You can manually add ...  
C:\Program Files\MATLAB\R2022a\toolbox\hdlverifier\supportpackages ...  
\fpgadebug_xilinx\hdlverifier\+fpga\+vivado ...  
to the IP search path setting of your Vivado project.
```

To use the IP, add this path to your FPGA project.

Input Arguments

projectName — File name of Vivado project

character vector

File name of an existing Vivado project. This function modifies the project to add the location of the AXI manager IP to the IP search path. If you do not specify this argument, the function displays the path to the AXI manager IP.

Version History

Introduced in R2017a

R2022a: setupAXIMasterForVivado renamed to setupAXIManagerForVivado

Warns starting in R2022a

The `setupAXIMasterForVivado` function has been renamed to the `setupAXIManagerForVivado` function. In the software and documentation, the terms "manager" and "subordinate" replace "master" and "slave," respectively.

To display the path to the AXI manager IP, use the `setupAXIManagerForVivado` function. Using the `setupAXIMasterForVivado` function is not recommended and will be removed in a future release. If you use the `setupAXIMasterForVivado` function, the function now gives this warning message.

`setupAXIMasterForVivado` has been renamed to `setupAXIManagerForVivado` in R2022a. `setupAXIMasterForVivado` will be removed in a future release. Use `setupAXIManagerForVivado` instead.

See Also

Classes

`aximanager`

Topics

“Access FPGA Memory Using JTAG-Based AXI Manager” on page 8-15

“Set Up AXI Manager” on page 3-2

loadImageToTargetSDCardPath

Load board-specific SD card image files to target SoC device SD card location

Syntax

```
loadImageToTargetSDCardPath(BoardName)
loadImageToTargetSDCardPath(BoardName,Name,Value)
```

Description

`loadImageToTargetSDCardPath(BoardName)` loads the SD card image files of the specified board to the specified SD card drive location on the target SoC device with default IP address by using Ethernet. The SD card image files contain a bootloader and the suitable operating system information.

`loadImageToTargetSDCardPath(BoardName,Name,Value)` specifies options using one or more name-value arguments. For example, 'Gateway', '192.168.0.4' sets the gateway for the network interface.

Input Arguments

BoardName — Targeted SoC board name

'ZC702' | 'ZC706' | 'ZedBoard' | 'ZCU102' | 'ZCU111' | 'ZCU216'

Targeted SoC board name, specified as one of these values.

- 'ZC702' — Xilinx Zynq-7000 ZC702
- 'ZC706' — Xilinx Zynq-7000 ZC706
- 'ZedBoard' — Xilinx Zynq ZedBoard
- 'ZCU102' — Xilinx Zynq UltraScale+ MPSoC ZCU102 Evaluation Kit
- 'ZCU111' — Xilinx Zynq UltraScale+ RFSoc ZCU111 Evaluation Kit
- 'ZCU216' — Xilinx Zynq UltraScale+ RFSoc ZCU216 Evaluation Kit

Data Types: char | string

Name-Value Pair Arguments

Specify optional pairs of arguments as `Name1=Value1, ..., NameN=ValueN`, where `Name` is the argument name and `Value` is the corresponding value. Name-value arguments must appear after other arguments, but the order of the pairs does not matter.

Before R2021a, use commas to separate each name and value, and enclose Name in quotes.

Example:

```
loadImageToTargetSDCardPath('ZC706','DeviceAddress','192.168.10.2','Gateway',
'192.168.10.1');
```

DeviceAddress — IP address of target SoC device SD card location

'192.168.0.2' (default) | character vector | string scalar

Internet protocol (IP) address of the target SoC device, specified as a character vector or string scalar. The target IP address must be a set of four numbers consisting of integers in the range [0, 255] that are separated by dots.

Use this name-value argument to specify a nondefault address when you use the `copyImageToHostSDCardPath` function.

Example: '192.168.0.8'

Data Types: char | string

Gateway — Gateway for network interface

'192.168.0.1' (default) | character vector | string scalar

Gateway for the network interface, specified as a character vector or string scalar. The gateway must be a set of four numbers consisting of integers in the range [0, 255] that are separated by dots.

Example: '192.168.0.4'

Data Types: char | string

SDCardImage — SD card image to copy

character vector | string scalar

SD card image to copy, specified as a character vector or string scalar. Use this name-value argument to copy a custom SD card image.

The default SD card image is the SD card files that come with the HDL Verifier support package.

Example: 'C:\mywork\hdlv_prj\sdcard_image\zc706_sdcard_zynq7000'

Data Types: char | string

Username — Username to log into target Linux operating system

'root' (default) | character vector | string scalar

Username to log into the target Linux operating system, specified as a character vector or string scalar. Use this name-value argument to specify a nondefault user account name.

Example: 'root'

Data Types: char | string

Password — Secret password associated with specified username

'root' (default) | character vector | string scalar

Secret password associated with the specified username, specified as a character vector or string scalar. Use this name-value argument to configure a nondefault user account password.

Example: 'root'

Data Types: char | string

Version History

Introduced in R2020a

See Also

copyImageToHostSDCardPath | loadBitstream

Topics

“Ethernet AXI Manager for Xilinx Zynq SoC Devices” on page 3-16

copyImageToHostSDCardPath

Copy board-specific SD card image files to host SD card location

Syntax

```
copyImageToHostSDCardPath(BoardName, SDCardDrive)
copyImageToHostSDCardPath(BoardName, SDCardDrive, Name, Value)
```

Description

`copyImageToHostSDCardPath(BoardName, SDCardDrive)` copies the SD card image files of the specified board to the specified SD card drive location on the host system. The SD card image files contain a bootloader and the suitable operating system information. Also, this function copies the server daemon for handling the AXI manager host commands on the target SoC device.

`copyImageToHostSDCardPath(BoardName, SDCardDrive, Name, Value)` specifies options using one or more name-value arguments. For example, 'DeviceAddress', '192.168.0.8' sets the internet protocol (IP) address of the target SoC device.

Input Arguments

BoardName — Targeted SoC board name

'VCK190' | 'ZC702' | 'ZC706' | 'ZedBoard' | 'ZCU102' | 'ZCU111' | 'ZCU216'

Targeted SoC board name, specified as one of these values.

- 'VCK190' — Xilinx Versal AI Core Series VCK190 Evaluation Kit
- 'ZC702' — Xilinx Zynq-7000 ZC702
- 'ZC706' — Xilinx Zynq-7000 ZC706
- 'ZedBoard' — Xilinx Zynq ZedBoard
- 'ZCU102' — Xilinx Zynq UltraScale+ MPSoC ZCU102 Evaluation Kit
- 'ZCU111' — Xilinx Zynq UltraScale+ RFSoc ZCU111 Evaluation Kit
- 'ZCU216' — Xilinx Zynq UltraScale+ RFSoc ZCU216 Evaluation Kit

Data Types: char | string

SDCardDrive — Name of SD card drive location

character vector | string scalar

Name of SD card drive location on the host computer, specified as a character vector or string scalar.

Example: 'G:' for Windows operating system

Example: '/media/username/261D-2F2B' for Linux operating system

Data Types: char | string

Name-Value Pair Arguments

Specify optional pairs of arguments as `Name1=Value1, . . . , NameN=ValueN`, where `Name` is the argument name and `Value` is the corresponding value. Name-value arguments must appear after other arguments, but the order of the pairs does not matter.

Before R2021a, use commas to separate each name and value, and enclose Name in quotes.

Example: `copyImageToHostSDCardPath('ZC706', 'G:', 'SDCardImage', 'C:\mywork\hdlv_prj\sdcard_image\zc706_sdcard_zynq7000');`

DeviceAddress — IP address of target SoC device

'192.168.0.2' (default) | character vector | string scalar

IP address of the target SoC device, specified as a character vector or string scalar. The target IP address must be a set of four numbers consisting of integers in the range [0, 255] that are separated by dots.

Example: '192.168.0.8'.

The host network interface card (NIC) address must be on the same subnet as the target SoC device.

Example: For example, if you specify this value as '192.168.0.8' the host NIC address can be 192.168.0.x. The variable x is any integer in the range [1, 7] and [9, 255].

Data Types: char | string

Gateway — Gateway for network interface

'192.168.0.1' (default) | character vector | string scalar

Gateway for the network interface, as a character vector or string scalar. The gateway must be a set of four numbers consisting of integers in the range from [0, 255] that are separated by dots.

Example: '192.168.0.4'

Data Types: char | string

SDCardImage — SD card image to copy

character vector | string scalar

SD card image to copy, specified as a character vector or string scalar. Use this name-value pair argument to copy a custom SD card image.

The default SD card image is the SD card files that come with the HDL Verifier support package.

Example: 'C:\mywork\hdlv_prj\sdcard_image\zc706_sdcard_zynq7000'

Data Types: char | string

Version History

Introduced in R2020a

See Also

`loadImageToTargetSDCardPath` | `loadBitstream`

Topics

“Ethernet AXI Manager for Xilinx Zynq SoC Devices” on page 3-16
“Guided Hardware Setup”

loadBitstream

Load custom FPGA bitstream and corresponding DTB file to target SoC device

Syntax

```
loadBitstream(BoardName, FPGAImage, DeviceTree)
loadBitstream(BoardName, FPGAImage, DeviceTree, Name, Value)
```

Description

`loadBitstream(BoardName, FPGAImage, DeviceTree)` loads the custom FPGA bitstream and corresponding device tree blob (DTB) file of the specified board to the targeted SoC device by using an Ethernet connection.

`loadBitstream(BoardName, FPGAImage, DeviceTree, Name, Value)` specifies options using one or more name-value arguments. For example, `'DeviceAddress', '192.168.0.8'` sets the internet protocol (IP) address of the target SoC device.

Input Arguments

BoardName — Targeted SoC board name

'ZC702' | 'ZC706' | 'ZedBoard' | 'ZCU102' | 'ZCU111' | 'ZCU216'

Targeted SoC board name, specified as one of these values.

- 'ZC702' — Xilinx Zynq-7000 ZC702
- 'ZC706' — Xilinx Zynq-7000 ZC706
- 'ZedBoard' — Xilinx Zynq ZedBoard
- 'ZCU102' — Xilinx Zynq UltraScale+ MPSoC ZCU102 Evaluation Kit
- 'ZCU111' — Xilinx Zynq UltraScale+ RFSoc ZCU111 Evaluation Kit
- 'ZCU216' — Xilinx Zynq UltraScale+ RFSoc ZCU216 Evaluation Kit

Data Types: char | string

FPGAImage — Custom FPGA bitstream file

character vector | string scalar

Custom FPGA bitstream file to be loaded onto the target board, specified as a character vector or string scalar.

Example: 'C:\mywork\hdlv_bitstreams\system.bit'

Data Types: char | string

DeviceTree — Custom DTB file

character vector | string scalar

Custom DTB file to load onto the target board, specified as a character vector or string scalar.

Example: 'C:\mywork\hdlv_bitstreams\devicetree.dtb'

You can also provide only the DTB file name if the file is located on the target SD card path.

Example: 'devicetree_axilite.dtb'

Data Types: char | string

Name-Value Pair Arguments

Specify optional pairs of arguments as `Name1=Value1, . . . ,NameN=ValueN`, where `Name` is the argument name and `Value` is the corresponding value. Name-value arguments must appear after other arguments, but the order of the pairs does not matter.

Before R2021a, use commas to separate each name and value, and enclose Name in quotes.

```
Example: loadBitstream('ZC706','C:\mywork\hdlv_bitstreams
\system.bit','C:\mywork\hdlv_bitstreams
\devicetree.dtb','DeviceAddress','192.168.10.2');
```

DeviceAddress — IP address of target SoC device

'192.168.0.2' (default) | character vector | string scalar

IP address of the target SoC device, specified as a character vector or string scalar. The target IP address must be a set of four numbers consisting of integers in the range [0, 255] that are separated by dots.

Use this name-value argument to specify a nondefault address when you use the `copyImageToHostSDCardPath` function.

Example: '192.168.0.8'

Data Types: char | string

Username — Username to log into target Linux operating system

'root' (default) | character vector | string scalar

Username to log into the target Linux operating system, specified as a character vector or string scalar. Use this name-value argument to specify a nondefault user account name.

Example: 'root'

Data Types: char | string

Password — Secret password associated with specified username

'root' (default) | character vector | string scalar

Secret password associated with the specified username, specified as a character vector or string scalar. Use this name-value argument to specify a nondefault user account password.

Example: 'root'

Data Types: char | string

Version History

Introduced in R2020a

See Also

`loadImageToTargetSDCardPath` | `copyImageToHostSDCardPath`

Topics

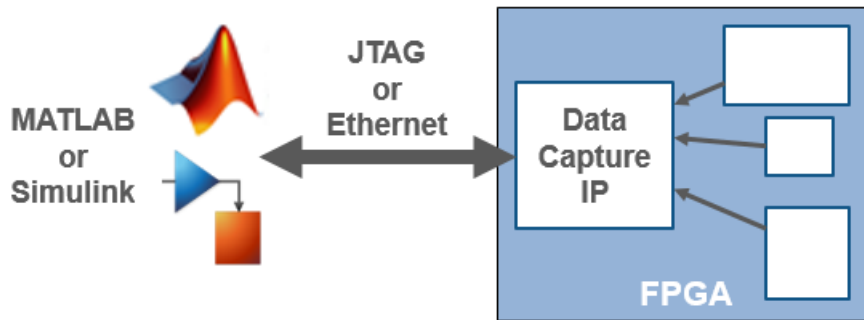
“Ethernet AXI Manager for Xilinx Zynq SoC Devices” on page 3-16

FPGA Data Capture

- “Data Capture Workflow” on page 6-2
- “Triggers” on page 6-7
- “Design Considerations for Data Capture” on page 6-11
- “Capture Conditions” on page 6-13

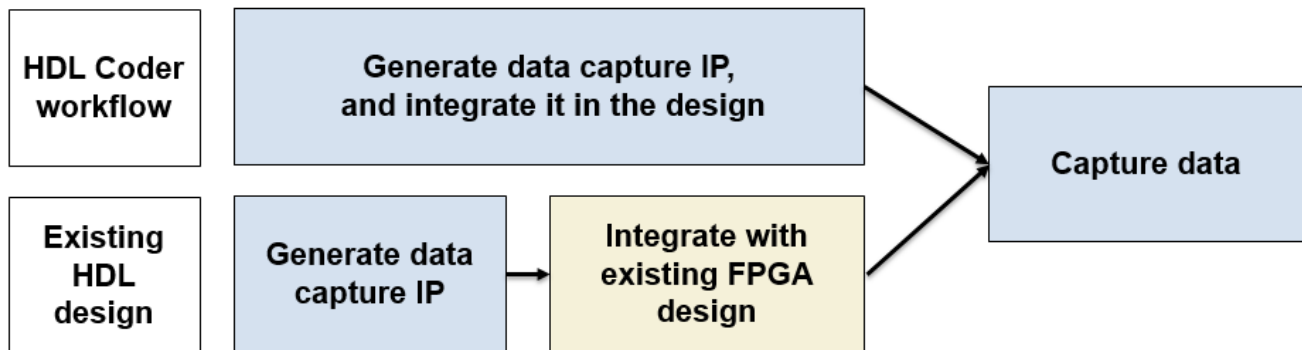
Data Capture Workflow

Use FPGA data capture to observe signals from your design while the design is running on the FPGA. This feature captures a window of signal data from the FPGA and returns the data to MATLAB or Simulink.



There are two workflows to capture data from your FPGA board into MATLAB or Simulink:

- First workflow — When you generate the HDL IP with HDL Coder, use the **HDL Workflow Advisor** tool to generate the data capture IP and integrate it in the design.
- Second workflow — If you have an existing HDL design, HDL Verifier provides tools to generate the data capture IP. Then, manually integrate the generated IP into your FPGA design.



To capture signals from your design, HDL Verifier generates an IP core that communicates with MATLAB. Use the HDL Coder workflow for automatic integration of the data capture IP core in your design. Otherwise, manually integrate this IP core into your HDL project and deploy it to the FPGA along with the rest of your design. Then, use one of the following methods to capture data.

- For capturing data to MATLAB - HDL Verifier generates a customized tool that returns the captured signal data. Alternatively, you can use the generated System object to capture data programmatically.
- For capturing data to Simulink - HDL Verifier generates a block that has output ports corresponding to the signals you captured.

In both cases, you can specify data types for the captured data, number of windows to capture, trigger condition that controls when to capture the data, and capture condition that controls which data to capture.

When the design is running on the FPGA, first the generated IP core waits for the trigger condition that you specify. Define a trigger condition by specific values matched on one or more signals. When the trigger is detected, the logic captures the designated signals to a buffer and returns the data over the JTAG or Ethernet interface to the host machine. You can then analyze and display these signals in your MATLAB workspace or Simulink model.

To make the best use of the buffer size and capture only the valid data, you can also define a capture condition. Define a capture condition in the same way as you define the trigger condition. When both the trigger is detected and the capture condition is satisfied, the logic captures only the valid values of the designated signals.

Generate and Integrate Data Capture IP Using HDL Workflow Advisor

When you use the **HDL Workflow Advisor** tool to generate your HDL design, first mark interesting signals as test points in Simulink.

Note FPGA data capture does not support Xilinx Versal devices in the **HDL Workflow Advisor** tool. To capture data from a Versal device, use the existing HDL design workflow.

Configure your design using the **HDL Workflow Advisor** tool to:

- Select the type of connection channel by setting the **FPGA Data Capture (HDL Verifier required)** parameter in the **Set Target Reference Design** task. For more information, see “Set Target Reference Design” (HDL Coder).
- Enable test point generation by selecting the **Enable HDL DUT port generation for test points** parameter in the **Set Target Interface** task. For more information, see “Set Target Interface” (HDL Coder).
- Connect test point signals to the FPGA Data Capture interface in the **Set Target Interface** task.
- Set up buffer size and maximum sequence depth for data collection in the **Generate RTL Code and IP Core** task. To include capture condition logic in the IP core, select **Include capture condition logic in FPGA Data Capture**. For more information, see “Generate RTL Code and IP Core” (HDL Coder).

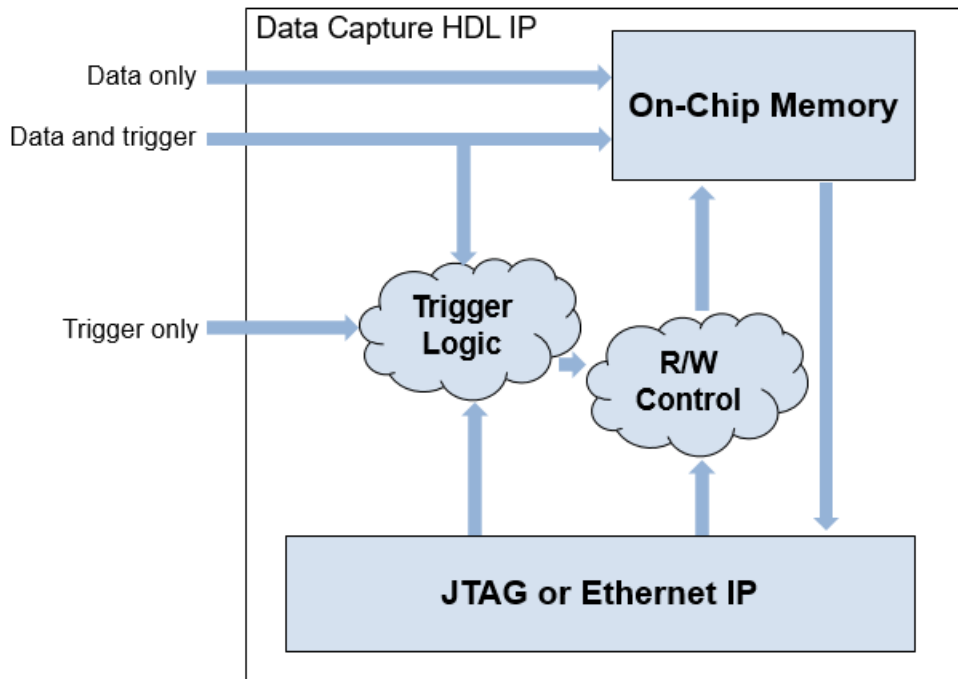
Then, run through the remaining steps to generate HDL for your design and program the FPGA. The data capture IP core is integrated in the generated design. You are now ready to “Capture Data” on page 6-5.

Configure and Generate IP Core for an Existing HDL Design

Before you can capture FPGA data, first specify which signals to capture, and how many samples to return. Use the **FPGA Data Capture Component Generator** to configure these and other settings, and to generate the HDL IP core. The IP core contains:

- A port for each signal you want to capture or use as part of a trigger condition

- Memory to capture the number of samples you requested for each signal
- JTAG or Ethernet interface logic to communicate with MATLAB
- Trigger and capture condition logic that can be configured at run time
- A ready-to-capture signal to control data flow from the FPGA



The tool also generates a customized **FPGA Data Capture** tool, System object, and model that communicate with the FPGA.

Integrate IP into FPGA

For MATLAB to communicate with the FPGA, you must integrate the generated HDL IP core into your FPGA design. If you used the **HDL Workflow Advisor** tool to generate your data capture IP, this step is automated. In this case, data capture IP operates on a single-clock rate, which is the primary clock of your device under test (DUT). If you did not use the **HDL Workflow Advisor** tool, follow the instructions in the generation report based on your device family.

Non-Versal Devices

Follow these instructions to integrate the `datacapture` HDL IP core into your FPGA design that targets a Xilinx non-Versal device.

- 1 Create a Vivado project.
- 2 Navigate to the `hdlsrc` folder.
- 3 Follow one of these steps based on your connection type.

- JTAG — Add the generated HDL files in the `hdlsrc` folder to your Vivado project. Then, instantiate the HDL IP core, `datacapture`, in your HDL code. Connect `datacapture` to the signals you requested for capture and triggers.
- Ethernet — Run the `insertEthernet` script by executing this command in the Vivado Tcl console.

```
source ./insertEthernet.tcl
```

Versal Devices

Follow these instructions to integrate the `datacapture` HDL IP core into your FPGA design targeted on a Xilinx Versal device.

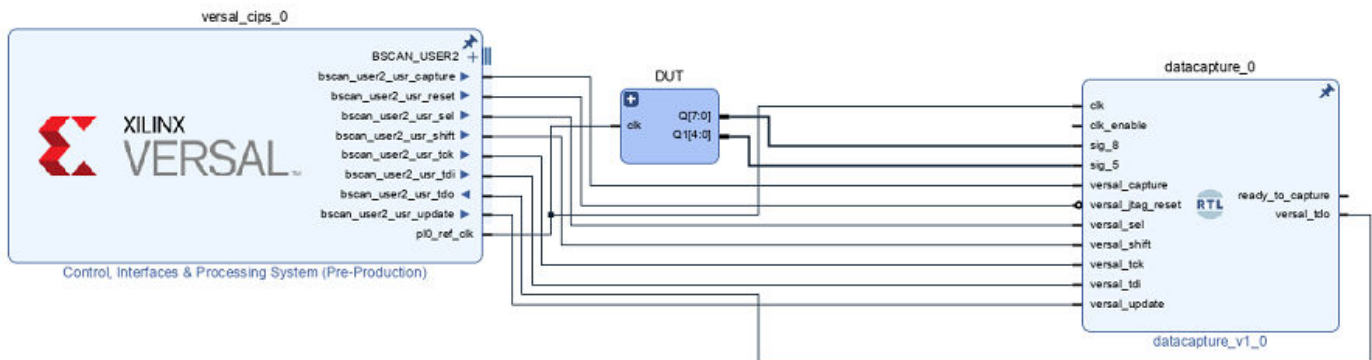
Note FPGA data capture support for Versal devices is available for JTAG connections only.

- 1 Open your block design in Vivado.
- 2 Navigate to the `hdlsrc` folder.
- 3 Insert the `datacapture` IP into your block design and connect the IP to the BSCAN_USER2 interface of the Xilinx Versal platform CIPS IP by executing this command in the Vivado Tcl console.

```
source ./insertVersalFPGADataCaptureIP.tcl
```

To enable the BSCAN_USER2 interface, enable the PL BSCAN1 interface in the CIPS IP.

- 4 Complete the block design by connecting the `clk`, `clk_enable`, and input data ports of the data capture IP.



Compile the project and program the FPGA with the new image through a JTAG cable.

Capture Data

The FPGA data capture IP core communicates over the JTAG or Ethernet cable between your FPGA board and the host computer. Make sure that the required cable is connected. Before capturing data, you can set data types for the captured data, set trigger condition that specifies when to capture the data, and set a capture condition that specifies the data to be captured. To configure these options and capture data, you can:

- Open the **FPGA Data Capture** tool. Set the trigger, capture condition, and data type parameters, and then capture data into the MATLAB workspace.

- Use the generated System object derived from `hdlverifier.FPGADataReader`. Set the data types, trigger condition, and capture condition using the methods and properties of the System object, and then call the object to capture data.
- In Simulink, open the generated model and configure the parameters of the FPGA Data Reader block. Then run the model to capture data.

After you capture the data and import it into the MATLAB workspace or Simulink model, you can analyze, verify, and display the data.

See Also

FPGA Data Capture Component Generator | FPGA Data Capture |
`hdlverifier.FPGADataReader` | FPGA Data Reader

Related Examples

- “Capture Temperature Sensor Data from Xilinx FPGA Board Using FPGA Data Capture” on page 8-2
- “Debug IP Core Using FPGA Data Capture” (HDL Coder)

Triggers

What Is a Trigger Condition?

A trigger condition is a logical statement that defines when to capture data from the FPGA. Use a trigger condition to capture data around an event of interest on the FPGA. Capture multiple occurrences of an event by setting Number of capture windows to the desired value. A trigger condition is composed of value comparison tests on one or more FPGA signals. For example:

```
counter == 100
```

All trigger comparisons are synchronous. When you specify an edge condition for a Boolean signal, the IP core compares the current sampled value with the sampled value from the previous clock cycle.

```
fifo_full == 'Rising edge'
```

The trigger condition is met when all terms of the condition are true on the same clock cycle. You can use only a single value comparison per signal.

```
receiver_state == 3 OR message_detected == 'High'
```

```
fifo_cnt == 0 AND fifo_pop == 'High'
```

You can use only a single type of logical operator in the trigger condition. You cannot mix AND and OR conditions.

```
fifo_empty == 'Rising edge' OR fifo_full == 'Rising edge' OR memctrl_state == 2
```

```
receiver_state == 3 AND message_addr == 148 AND pkt_type == 5
```

You can use multiple comparison operators in the trigger condition.

```
fifo_empty == 'Rising edge' OR fifo_full != 'LOW' OR memctrl_state == 2
```

```
receiver_state == 3 AND message_addr > 148 AND pkt_type >= 5
```

You can use X or x (don't-care value) in the trigger condition. While comparing, the trigger condition ignores the place values with X. When the trigger condition is 0b1X1, the possible trigger condition values are 0b101 or 0b111.

```
receiver_state == 3 AND message_addr == 148 AND pkt_type == 0b1X1
```

Sequential Trigger

A sequential trigger enables you to give a set of trigger conditions in multiple stages to capture specified data from an FPGA. With a sequential trigger, you can read data to MATLAB or Simulink only after all of the specified trigger conditions happen in sequence. For multiple trigger stages, set the **Max trigger stages** parameter of the **FPGA Data Capture Component Generator** tool to a value greater than 1. **Max trigger stages** sets the maximum number of trigger stages for providing trigger conditions. For example, if **Max trigger stages** is 3, the **Trigger** tab in the **FPGA Data Capture** tool or in the FPGA Data Reader block can have maximum of 3 trigger stages.

Define a trigger condition by specific values matched on one or more signals in each stage. For example, if the number of trigger stages is 3 and 10 signals exist, you can set these trigger conditions.

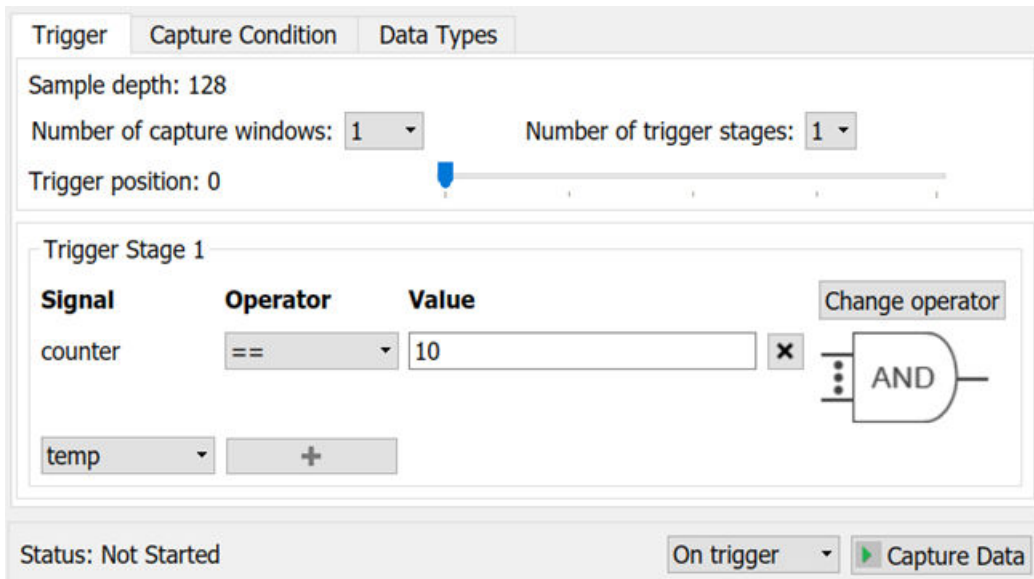
- Trigger condition for stage 1:
`((signal1 > 10) and (signal3 == true) and (signal7 < 5));`
- Trigger condition for stage 2:
`((signal1 == 0b0110) or (signal4 == 0XXX) or (signal8 < 5));`
- Trigger condition for stage 3:
`((signal2 != 5) and (signal6 == true) and (signal8 == 8));`

Configure a Trigger Condition

At generation time, specify which signals you want to be available for use in trigger conditions. A signal can be a trigger without capturing data, or it can be both a trigger and a captured signal. You can modify the trigger condition at capture time, using any signals you specified as triggers. The data capture IP core on the FPGA receives the trigger definition from MATLAB and configures on-chip muxes to detect the event.

When you use the **FPGA Data Capture** tool, or the FPGA Data Reader block, set the trigger condition on the **Trigger** tab. Each line in the table is the value comparison for one signal. To

combine the signal values, use the trigger combination operator. To show a signal on this tab, you must specify the signal as a trigger at generation time.



For an `hdlverifier.FPGADataReader` System object, configure the trigger condition using the `setTriggerCondition`, `setTriggerComparisonOperator`, and `setTriggerCombinationOperator` object functions. To check your configuration, call the `displayTriggerCondition` object function.

If you do not enable a trigger condition on any signal, the data capture IP core captures data immediately.

Trigger Position

You can change the relative position of the trigger detection cycle within the capture buffer. Use this feature to capture the relevant data, whether it is before or after the trigger event.

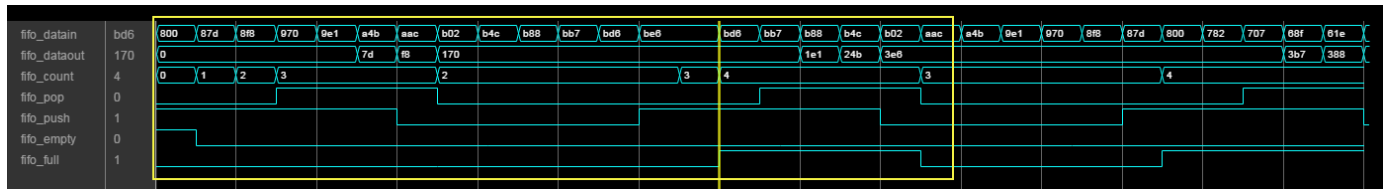
Suppose you want to debug the rates of pushes and pops to a FIFO design. You can set a trigger on a High value of the signals `fifo_empty` or `fifo_full`.



By default, the clock cycle when the trigger is detected is the first sample of the capture buffer. The IP core captures a buffer starting from the cycle when `fifo_full` changes to high.



To debug the `fifo_full` condition, observe the signals before the trigger condition occurs. In the capture settings, change the **Trigger position** to 3/4 of the window depth using the tic mark on the slider. For example, if **Sample depth** is 128, **Number of capture windows** is 1, and **Number of trigger stages** is 1, then *window depth* is 128. The trigger event is at sample 96 of that window. The IP core captures a buffer that contains 96 samples before the trigger event, and 36 samples after the trigger event. This setting captures data that shows the lead-up to the trigger event, and the aftermath. The location of the trigger event is shown with the vertical cursor at `fifo_full`.



You can set the **Trigger position** to a number of samples between 0 and the *window depth*-1, inclusive. When you set the trigger position equal to *window depth*-1, the last sample corresponds to the cycle when the trigger occurs.

To observe more than one occurrence of the trigger event, change the **Number of capture windows** to the desired number.

In this example, **Number of capture windows** is 4, **Number of trigger stages** is 1, **Sample depth** is 128, and **Trigger position** is 0. HDL IP captures four windows, where each *window depth* is 32 samples, starting when `fifo_full` changes to high.



See Also

Tools

FPGA Data Capture Component Generator | FPGA Data Capture

Objects

hdlverifier.FPGADatReader

Blocks

FPGA Data Reader

Related Examples

- “Capture Temperature Sensor Data from Xilinx FPGA Board Using FPGA Data Capture” on page 8-2

More About

- “Data Capture Workflow” on page 6-2

Design Considerations for Data Capture

Signals to Capture

To get started with FPGA data capture, you must specify port names and sizes for the generated IP. You then connect these ports to the signals in your design that you want to capture. You can specify bit widths between 1 and 128 bits. The default data type of the captured data depends on this bit width.

The FPGA data capture tools do not limit the total number of signals or bits you can capture. You are limited only by the hardware resource usage on your FPGA. When you select signals and the depth of the capture buffer, consider the memory and signal routing resources required on the FPGA.

In the **FPGA Data Capture Component Generator**, you can specify a signal for use as data or trigger. When you specify a signal as data, the signal is captured to the sample buffer and returned to MATLAB, but it cannot contribute to a trigger condition and capture condition. A data signal uses memory resources on the FPGA. When you specify a signal as a trigger, it is available for defining a trigger condition and capture condition at capture time, but is not captured and returned to MATLAB. A trigger signal uses logic resources on the FPGA. You can also specify that the signal is used as both trigger and data.

At capture time, you can configure the data type of the variable returned to MATLAB or Simulink. You can select built-in types, or, with Fixed-Point Designer™, you can specify fixed-point data types. If you do not have Fixed-Point Designer, data capture can only return built-in data types, such as `uint8`. In this case, you must specify ports for the generated IP that match the sizes of the built-in data types, that is, 1, 8, 16, 32, or 64 bits.

Capture Timing

The data capture feature captures a fixed-size buffer of data each time you request a capture. The feature does not stream continuous data from your FPGA into MATLAB or Simulink. You can capture a buffer immediately, or you can configure a logical trigger condition to control when the buffer is captured. You can configure the timing of the capture relative to the cycle the trigger is detected in, and configure the capturing of multiple windows of trigger events. You can also configure a logical capture condition to filter the data to be captured. While the data capture IP waits for a trigger, captures data, and returns the captured data to MATLAB, you cannot initiate a new capture request. Therefore, you cannot capture back-to-back buffers from the FPGA.

Use this feature to investigate design behavior around a specific event or to sample data occasionally, rather than for continuous observation. For more information about how to use trigger condition and capture condition, see “Triggers” on page 6-7 and “Capture Conditions” on page 6-13, respectively.

JTAG Considerations

The generated data capture IP can coexist in your design with other IPs that use the JTAG connection, such as Altera® SignalTap II or Xilinx Vivado Logic Analyzer cores. However, only one of these applications can use the JTAG cable at a time. You must close the **FPGA Data Capture** tool or model, or release the object, to return the JTAG resource for use by other applications.

The most common conflicting use of the JTAG cable is to reprogram the FPGA. You must stop any FPGA data capture or AXI manager JTAG connection before you can use the cable to program the FPGA.

The maximum data rate between host computer and FPGA is limited by the JTAG clock frequency. For Intel boards, the JTAG clock frequency is 12 or 24 MHz. For Xilinx boards, the JTAG clock frequency is 33 or 66 MHz. The JTAG frequency depends on the type of cable and the maximum clock frequency supported by the FPGA board.

Simultaneous Use of FPGA Data Capture and AXI Manager

The nonblocking capture mode enables you to simultaneously use FPGA data capture and AXI manager, which share a common JTAG interface. You do not need to close or release the JTAG resource to switch between FPGA data capture and AXI manager.

FPGA data capture supports these two capture modes.

- Blocking mode — FPGA data capture blocks MATLAB while retrieving captured data. In this capture mode, the JTAG resource is allocated to either FPGA data capture or AXI manager at a time.
- Nonblocking mode — FPGA data capture does not block MATLAB while retrieving captured data. In this capture mode, you can use FPGA data capture and AXI manager simultaneously.

By default, FPGA data capture is configured in blocking mode. Change the capture mode to nonblocking mode by using the `CaptureMode` property for an `hdlverifier.FPGADataReader` System object. After changing the capture mode to nonblocking, you can use the command line interface or graphical user interface for performing the remaining steps in FPGA data capture and AXI manager. For an example, see “Debug IP Core Using FPGA Data Capture” (HDL Coder).

Ethernet Considerations

The generated data capture IP can coexist in your design with other IPs that use the Ethernet connection, such as UDP AXI Manager IP. However, you must connect these IPs to the same Ethernet MAC Hub IP using different port addresses. For more information on Ethernet MAC Hub IP, see “Ethernet AXI Manager” on page 3-10.

See Also

More About

- “Xilinx FPGA Board Support from HDL Verifier” on page 1-2
- “Supported EDA Tools and Hardware” on page 1-6
- “Data Capture Workflow” on page 6-2

Capture Conditions

What Is Capture Condition?

A capture condition is a logical statement that controls which data to capture from the FPGA. Use a capture condition when you want to:

- Capture only the valid data to debug custom designs with FPGA data capture.
- Filter the data to capture based on trigger conditions.
- Optimize the use of an FPGA data capture buffer.
- Efficiently analyze the captured data when you have only a few captured samples of interest.

A capture condition is composed of value comparison tests on one or more FPGA signals. For example:

```
counter == 100
```

All capture comparisons are synchronous. When you specify an edge condition for a Boolean signal, the IP core compares the current sampled value with the sampled value from the previous clock cycle.

```
fifo_full == 'Rising edge'
```

The capture condition is met when all terms of the condition are true on the same clock cycle. You can use only a single value comparison per signal.

```
receiver_state == 3 OR message_detected == 'High'
```

```
fifo_cnt == 0 AND fifo_pop == 'High'
```

You can use only a single type of logical operator in the capture condition. You cannot mix AND and OR conditions.

```
fifo_empty == 'Rising edge' OR fifo_full == 'Rising edge' OR memctrl_state == 2
```

```
receiver_state == 3 AND message_addr == 148 AND pkt_type == 5
```

You can use multiple comparison operators in the capture condition.

```
fifo_empty == 'Rising edge' OR fifo_full != 'LOW' OR memctrl_state == 2
```

```
receiver_state == 3 AND message_addr > 148 AND pkt_type >= 5
```

You can use X or x (don't-care value) in the capture condition. While comparing, the capture condition ignores the place values with X. When the capture condition is 0b1X1, the possible trigger condition values are 0b101 or 0b111.

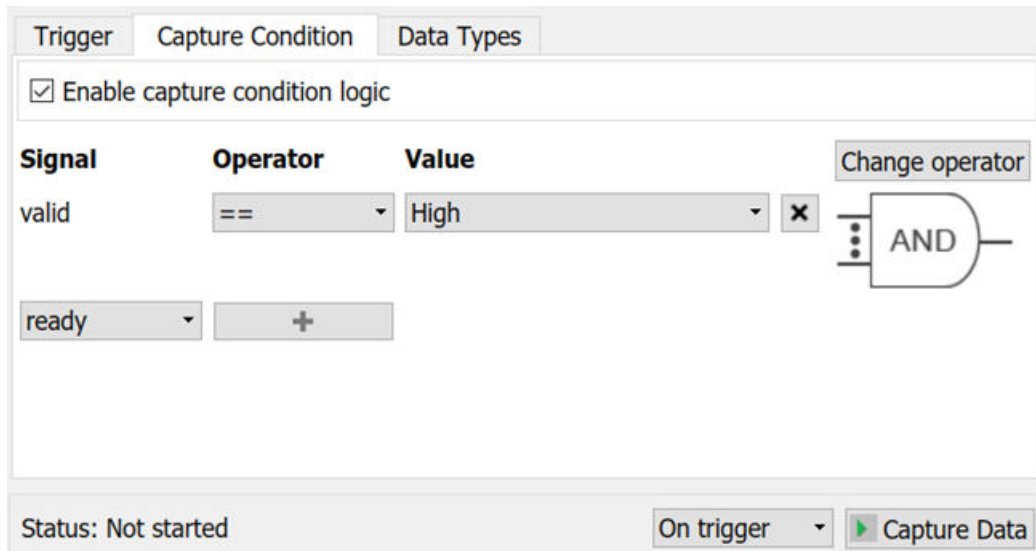
```
receiver_state == 3 AND message_addr == 148 AND pkt_type == 0b1X1
```

Configure Capture Condition

At generation time, specify which signals you want to be available for use in the capture condition. You can use a signal containing only a trigger or both a trigger and captured data. You can modify the capture condition at capture time using any signals you specify as triggers. Also, at generation time, you can include capture condition logic to use the capture condition. The data capture IP core on the

FPGA receives the capture definition from MATLAB and configures on-chip muxes to detect the capture event.

When you use the **FPGA Data Capture** tool or the FPGA Data Reader block, on the **Capture Condition** tab, select the **Enable capture condition logic** parameter and then set the capture condition. Each line in the table is the value comparison for one signal. To combine the signal values, use the capture condition combination operator. To show a signal on this tab, you must specify the signal as a trigger at generation time.



For an `hdlverifier.FPGADataReader` System object, enable capture condition logic using the `EnableCaptureCtrl` property. Then configure the capture condition using the `setCaptureCondition`, `setCaptureConditionComparisonOperator`, and `setCaptureConditionCombinationOperator` object functions. To check your configuration, call the `displayCaptureCondition` object function.

Differences Between Triggers and Capture Conditions

The trigger condition controls when to capture data from the FPGA. Once the trigger condition is satisfied, the data capture IP core captures data from that trigger event. The capture condition controls which data to capture. The data capture IP core evaluates the capture condition at each clock cycle and captures only the data that satisfies the capture condition.

FPGA data capture operates in two modes: immediate mode and trigger mode. The data capture IP core captures data from the FPGA without checking for the trigger condition in immediate mode and based on a trigger condition in trigger mode. You can provide a capture condition in both modes.

Filter the data to capture using a capture condition. In immediate mode, use a capture condition to capture data only when a certain condition is met. In trigger mode, use a capture condition to capture data only when a certain condition is met after satisfying the trigger condition.

Data Capture Mode	Trigger Condition Only	Capture Condition Only	Both Trigger and Capture Conditions
Immediate	<p>Ignores all trigger conditions</p> <p>Captures data immediately at each clock cycle</p>	Captures data only when the capture condition is true	<p>Ignores all trigger conditions</p> <p>Captures data only when the capture condition is true</p>
Trigger	Waits until the trigger condition is true, then captures data	Not supported, use immediate mode instead	Waits until the trigger condition is true, then captures data only when the capture condition is true

See Also

Tools

FPGA Data Capture Component Generator | FPGA Data Capture

Objects

hdlverifier.FPGADataReader

Blocks

FPGA Data Reader

More About

- “Data Capture Workflow” on page 6-2

Data Capture Reference

FPGA Data Capture Component Generator

Configure and generate FPGA data capture components

Description

The **FPGA Data Capture Component Generator** tool configures and generates components for capturing data from a design running on an FPGA. The generated components capture a window of signal data from the FPGA and return the data to MATLAB or Simulink.

Description
Generate an HDL IP for integration into your FPGA design.

Specify port names and bit-widths for the interface of the generated IP, and specify how many samples the IP captures at a time.

```

graph LR
    A[Generate data capture IP] --> B[Integrate with existing FPGA design]
    B --> C[Capture data]
  
```

[Read more about the data capture workflow](#)

Ports

Port Name	Bit Width	Use As
	1	Both trigger and data

Target

Generated IP name:

FPGA vendor:

Generated IP language:

Connection type:

Destination folder: ...

Capture

Sample depth:

Max trigger stages:

Include capture condition logic

To use this tool, you must have an existing HDL design and FPGA project. To capture the signals, HDL Verifier generates an IP core that you must integrate into your HDL project, and deploy to the FPGA along with the rest of your design.

The **Generate** button in this tool generates these components:

- HDL IP core, for integration into your FPGA design. Connect the signals you want to capture and use as triggers, and connect a clock and clock enable.
- Generation report, with list of generated files and instructions for next steps.
- Tool to set capture parameters and capture data to the MATLAB workspace. See **FPGA Data Capture**.
- Customized version of the `hdlverifier.FPGADataReader` System object that provides an alternative, programmatic, way to configure and capture data.
- Simulink model that contains a customized FPGA Data Reader block. If you have a DSP System Toolbox™ license, this model streams the captured signals into the **Logic Analyzer** waveform viewer. Otherwise, the Scope block displays the signals.
- MAT file in the `datacapture_gensettings.mat` format, where `datacapture` is the name of the generated HDL IP core. This MAT file holds the data capture build parameters. To reload the same design in your next iteration, provide this MAT file as an input argument to the `generateFPGADataCaptureIP` function.

For a workflow overview, see “Data Capture Workflow” on page 6-2.

Open the FPGA Data Capture Component Generator

At the MATLAB command prompt, enter this command.

```
generateFPGADataCaptureIP
```

To reload the parameters of the most recent design, use the `restore` argument.

```
generateFPGADataCaptureIP('restore',true);
```

To reload the parameters of a design you already generated and saved in a MAT file, use the `matFile` argument.

```
generateFPGADataCaptureIP('datacapture_gensettings.mat');
```

Where `datacapture` is the name of the generated HDL IP core that you specify in the **Generated IP name** parameter.

Examples

- “Capture Temperature Sensor Data from Xilinx FPGA Board Using FPGA Data Capture” on page 8-2

Parameters

Ports

Port Name — Name of input port on generated IP
character vector | string scalar

The name does not have to match the signal name in your HDL files. This name is used for:

- Input port on the generated HDL IP core. Internal to the IP, this signal is routed to the capture buffer, or to use as part of trigger condition and capture condition, depending on your selection for **Use As**.
- Structure field in the captured data returned to the MATLAB workspace
- Port on the generated Simulink block
- Table of signals in the trigger, capture condition, and data types parameters editor at capture time

Data Types: `char` | `string`

Bit Width — Number of bits in signal
positive integer

This number is used to generate the HDL IP port definition, and contributes to the total width of the capture buffer. You can specify the data type for the captured data at capture time.

Note If you do not have Fixed-Point Designer, data capture can only return built-in data types, such as `uint8`. You must specify ports for the generated IP that match the sizes of the built-in data types, that is 1, 8, 16, 32, or 64 bits. We recommend Fixed-Point Designer to enable fixed-point data types and captured signals of any size.

Use As — How signal is routed inside IP logic
`Both trigger and data (default)` | `Data` | `Trigger`

When you specify a signal as `Data`, the signal is captured to the sample buffer and returned to MATLAB, but it cannot contribute to a trigger condition and capture condition. When you specify a signal as `Trigger`, it is available for defining a trigger condition and capture condition at capture time, but is not captured and returned to MATLAB. You can also specify that the signal is used as `Both trigger and data`.

Target

Generated IP name — Name of generated components
`datacapture (default)` | character vector

This name is used for the generated HDL IP core, the System object, and the Simulink model.

FPGA vendor — FPGA and software vendor
`Altera (default)` | `Xilinx`

The available vendors depend on which HDL Verifier support package you have installed. There are separate support packages for Intel (Altera) and Xilinx boards.

Generated IP language — Language used for generated HDL IP core
`VHDL (default)` | `Verilog`

Select the language used for the generated HDL IP core as `Verilog` or `VHDL`.

Connection type — Type of connection channel
`JTAG (default)` | `Ethernet`

Select the type of connection channel as `JTAG` or `Ethernet`.

Note Ethernet connection is available for Xilinx FPGA boards only.

Destination folder — Location to save generated files

hdlsrc (default) | character vector | string scalar

Location to save the generated files, specified as the name of a folder on the host computer.

Data Types: char | string

Capture

Sample depth — Number of samples captured for each signal

128 (default) | 256 | 512 | 1024 | 2048 | 4096 | 8192 | 16384 | 32768 | 65536 | 131072 | 262144 | 524288 | 1048576

Use this parameter to specify the size of the memory in the generated HDL IP core. The width of the memory is the total bit width of the data signals.

When you specify the sample depth, consider the number of windows you plan to configure when reading the data, because together they impact the window depth of each capture window. The window depth is the sample depth divided by the number of capture windows. Specify the number of capture windows by using the **Number of capture windows** parameter in the **FPGA Data Capture** tool or by using the NumCaptureWindows property for an hdlverifier.FPGADatReader System object.

For example, if the sample depth is 4096 and the number of capture windows is 4, then each capture window has a window depth of 1024.

Max trigger stages — Maximum number of trigger stages for providing trigger conditions

1 (default) | integer from 1 to 10

Use this parameter to enable a sequential trigger. To capture specified data from an FPGA, give a set of trigger conditions in multiple stages. For more information on sequential trigger, see “Sequential Trigger” on page 6-7.

When you specify the **Max trigger stages**, consider the maximum number of trigger stages in which you plan to configure the trigger conditions. Specify the number of trigger stages by using the **Number of trigger stages** parameter in the **FPGA Data Capture** tool or by using the NumTriggerStages property for an hdlverifier.FPGADatReader System object.

For example, if the maximum number of trigger stages is 4, then the number of trigger stages can be 1, 2, 3, or 4.

Include capture condition logic — Option to include capture condition logic in HDL IP core

off (default) | on

Select this parameter to include capture condition logic in the HDL IP core. Include capture condition logic to use a capture condition to control which data to capture from the FPGA. The HDL IP core evaluates the capture condition at each clock cycle and captures only the data that satisfies the capture condition. For more information on capture conditions, see “Capture Conditions” on page 6-13.

Set up a capture condition in the **FPGA Data Capture** tool or the hdlverifier.FPGADatReader System object.

Ethernet settings

IP address — IP address of Ethernet port on target FPGA board
192.168.0.2 (default) | dotted-quad value

Specify the internet protocol (IP) address of the Ethernet port on the target FPGA board as a dotted-quad value. The target IP address must be a set of four numbers consisting of integers in the range from 0 to 255 that are separated by three dots.

Dependencies

To enable this parameter, in the **Target** section, set the **Connection type** parameter to Ethernet.

Port address — UDP port number of target FPGA board
50101 (default) | integer from 255 to 65,535

Specify the user datagram protocol (UDP) port number of the target FPGA board as an integer from 255 to 65,535.

Dependencies

To enable this parameter, in the **Target** section, set the **Connection type** parameter to Ethernet.

Data Types: single | double | int8 | int16 | int32 | int64 | uint8 | uint16 | uint32 | uint64

Interface type — Type of Ethernet interface for target FPGA board
GMII (default) | MII | SGMII

Select the Ethernet interface type as GMII, MII, or SGMII based on the interface for your target FPGA board.

Dependencies

To enable this parameter, in the **Target** section, set the **Connection type** parameter to Ethernet.

Version History

Introduced in R2017a

See Also

FPGA Data Capture | generateFPGADataCaptureIP

Topics

“Capture Temperature Sensor Data from Xilinx FPGA Board Using FPGA Data Capture” on page 8-2

“Data Capture Workflow” on page 6-2

generateFPGADataCaptureIP

Open FPGA Data Capture Component Generator

Syntax

```
generateFPGADataCaptureIP  
generateFPGADataCaptureIP('restore',true)  
generateFPGADataCaptureIP(matFile)
```

Description

generateFPGADataCaptureIP opens the **FPGA Data Capture Component Generator**.

generateFPGADataCaptureIP('restore',true) opens the **FPGA Data Capture Component Generator** and reloads the parameters of the most recent design.

generateFPGADataCaptureIP(matFile) opens the **FPGA Data Capture Component Generator** and reloads the parameters of a design you already generated and saved in matFile.

Examples

Launch FPGA Data Capture Component Generator

This example shows how to launch the **FPGA Data Capture Component Generator** tool.

Open Tool With Default Settings

Run the following command to launch the **FPGA Data Capture Component Generator** tool:

```
generateFPGADataCaptureIP
```

FPGA Data Capture Component Generation
✕

Description

Generate an HDL IP for integration into your FPGA design.

Specify port names and bit-widths for the interface of the generated IP, and specify how many samples the IP captures at a time.

Generate data capture IP

→

Integrate with existing FPGA design

→

Capture data

[Read more about the data capture workflow](#)

Ports

Port Name	Bit Width	Use As	
sig1	1	Both trigger and data	<div style="border: 1px solid gray; padding: 2px; width: 100px; margin: 0 auto;">Add</div> <div style="border: 1px solid gray; padding: 2px; width: 100px; margin: 0 auto;">Remove</div>

Target

Generated IP name:

FPGA vendor:

Generated IP language:

Connection type:

Destination folder:

Capture

Sample depth:

Max trigger stages:

Include capture condition logic

Input Arguments

restore – Option to reload most recent design parameters

false (default) | true

Option to reload the most recent design parameters, specified as `true` or `false`. When you specify this input as `true`, the **FPGA Data Capture Component Generator** opens and reloads the parameters of the most recent design. Otherwise, the tool opens with default settings.

Example: `'restore', true`

Data Types: `logical`

matFile – MAT file containing design parameters

character vector | string scalar

MAT file containing design parameters, specified as a character vector or string scalar. Specify this input in the form *datacapture_gensettings.mat*, replacing *datacapture* with the generated HDL IP core name.

This MAT file is a generated file in the same folder as your other generated data capture components. When you specify this input, the **FPGA Data Capture Component Generator** opens and reloads the parameters of the design specified by this value.

You must provide only the MAT file name if the file is in the current working folder.

Example: 'datacapture_gensettings.mat'

You must provide the full path to the saved MAT file if the file is not in the current working folder.

Example: 'C:/home/user/datacapture_gensettings.mat'

Data Types: char | string

Version History

Introduced in R2017a

R2023a: Reload previous design parameters

The function reloads the parameters of a design you already generated and saved in a MAT file.

To reload the same design in your next iteration, provide the MAT file containing design parameters as an input argument to the function.

```
generateFPGADataCaptureIP('datacapture_gensettings.mat');
```

Where *datacapture* is the **Generated IP name** that you specify in the **FPGA Data Capture Component Generator** tool.

See Also

FPGA Data Capture Component Generator

Topics

“Capture Temperature Sensor Data from Xilinx FPGA Board Using FPGA Data Capture” on page 8-2

FPGA Data Capture

Capture data from live FPGA into MATLAB workspace interactively

Description

The **FPGA Data Capture** tool captures data from a design running on an FPGA and returns it to the MATLAB workspace. You can configure the data types of the returned values, specify the number of capture windows and number of trigger stages, set up a trigger condition to control when the data is captured, and set up a capture condition to control which data to capture.

The screenshot shows the 'FPGA Data Capture' tool window. It includes a description, a workflow diagram, output settings, and trigger configuration tabs. The 'Trigger' tab is active, showing two trigger stages with signal 'in1' and an AND operator. The status is 'Not started' and the 'Capture Data' button is visible.

Description
Capture data from a design running on your FPGA board.
Specify data types for the returned data structure, and specify a logical trigger condition that defines when the data is captured.

Workflow: Generate data capture IP → Integrate with existing FPGA design → Capture data

[Read more about the data capture workflow](#)

Output
Output variable name: dataCaptureOut Display data with Logic Analyzer

Trigger | Capture Condition | Data Types

Sample depth: 4096
Number of capture windows: 1 | Number of trigger stages: 2
Trigger position: 0

Trigger Stage 1

Signal	Operator	Value
in1	+	

Change operator: AND

Trigger Stage 2

Signal	Operator	Value
in1	+	

Trigger time out 1

Change operator: AND

Status: Not started | Immediately | Capture Data

Before using this tool, you must have generated the customized data capture components using the **FPGA Data Capture Component Generator** tool. You must also integrate the generated IP code

into your project and deploy it to the FPGA. The tool communicates with the FPGA over a JTAG or Ethernet cable. Make sure that the required cable is connected between the board and the host computer.

The tool is a wrapper over your generated `hdlverifier.FPGADataReader` System object. The **FPGA Data Capture** tool defines the variable, `fpgadc_obj` in the workspace. If this variable already exists, the tool opens using the existing object, and saves modifications to that object.

For a workflow overview, see “Data Capture Workflow” on page 6-2.

Open the FPGA Data Capture

- MATLAB command prompt: Enter `launchDataCaptureApp`. This function is a generated script in the same folder as your other generated data capture components.

Examples

- “Capture Temperature Sensor Data from Xilinx FPGA Board Using FPGA Data Capture” on page 8-2

Parameters

Capture Data — When to capture data

`Immediately` (default) | `On trigger`

The default setting, capture `Immediately`, ignores any trigger condition and captures the buffer of data when you click **Capture Data**. To capture data that includes a particular event in the FPGA logic, configure a trigger condition and select `On trigger`. In this case, the data capture logic waits until the trigger condition is true, then captures the buffer of data.

When you click **Capture Data**, a window with a **Stop** button opens. If you want to cancel the capture attempt (for example, if the trigger condition does not occur), click **Stop** to return control to the tool. When you abort a capture attempt, no data is returned to MATLAB.

Output

Output variable name — Name of structure in which to return captured data

`dataCaptureOut` (default) | character vector

The captured data is returned to a structure variable in the base MATLAB workspace. The data returned from each signal is a vector of `Sample depth` values. Each signal becomes a field in the structure. The field name in the structure is the same as the **Signal Name**.

Display data with Logic Analyzer — Automatically display data in Logic Analyzer

`on` (default) | `off`

This option appears if you have a DSP System Toolbox license. When you select this option, after data capture is complete, the tool opens the Logic Analyzer window to display the captured data. The time axes is measured in samples. The cursor location indicates the time the trigger was detected.

Trigger

Sample depth — Number of samples captured for each signal
integer power of two

This parameter is read-only. It reflects the value you specified at generation time.

Number of capture windows — Number of data capture recurrences
1 (default) | integer power of two

Specify the number of recurrences to capture. This value must be a power of two, and cannot be greater than **Sample depth**. When specifying the sample depth, consider the number of windows you plan to configure when reading the data, because together they impact the window depth of each capture window. The window depth is the **Sample depth** divided by the **Number of capture windows**. Specify **Sample depth** in the **FPGA Data Capture Component Generator** tool.

For example: If **Sample depth** is 4096 and **Number of capture windows** is 4, then each capture window has a window depth of 1024.

Number of trigger stages — Number of trigger stages for providing trigger conditions
 M (default) | integer from 1 to M

Specify the number of trigger stages. This value must be an integer from 1 to M , where M is set by the **Max trigger stages** parameter of the **FPGA Data Capture Component Generator** tool. When you specify the **Max trigger stages** parameter, consider the maximum number of trigger stages in which you plan to configure the trigger conditions to capture data.

For example, if **Max trigger stages** is 4, then **Number of trigger stages** can be 1, 2, 3, or 4.

Trigger position — Position of the trigger detection cycle within the capture buffer
0 (default) | integer up to $window\ depth - 1$

By default, the clock cycle when the trigger is detected is the first sample of the capture buffer. You can change the relative position of the trigger detection cycle within the capture buffer. A nondefault trigger position means that some samples are captured before the trigger occurs. You can set this parameter to any number from 0 to $window\ depth - 1$, inclusive. When the trigger position is equal to the $window\ depth - 1$, the last sample corresponds to the cycle when the trigger occurs. If **Number of capture windows** is greater than one, the same trigger position applies to all windows. For more information, see “Triggers” on page 6-7.

Signal — Trigger component signal name
character vector

This parameter is read-only. The signal names you specified at generation time are listed in the drop-down menu at the bottom. Click the + button to add a signal to the trigger condition.

Operator — Operator to compare signals within trigger condition
== | != | < | > | <= | >=

To compare signals, select one of these operators: ==, !=, <, >, <=, or >=. To compare signals containing X or x (don't-care value), specify either == or != operator.

Value — Value to compare signal to as part of overall trigger condition
decimal | binary | hexadecimal | Low | High | Falling edge | Rising edge | Both edges

The trigger condition can be composed of value comparisons of one or more signals. This parameter specifies the value to match for each signal.

For a multibit signal, specify a decimal, binary, or a hexadecimal value within the range of the data type associated with the signal. While providing hexadecimal or binary values, you can provide values with a combination of X or x (don't care value) to enable bit masking. While comparing the values, the trigger condition discards place values with X or x and provides the output.

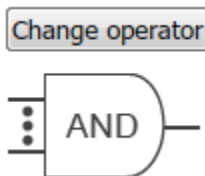
To separate a group of bits for better readability, you can use `_` between bits. For example, you can represent a 32-bit binary value as `0b1010_XXXX_1011_XXXX_1110_XXXX_1111XXXX` and a 32-bit hexadecimal value as `0xAB_CDEFX`.

For **boolean** signals, select a level or edge condition. For more information, see “Triggers” on page 6-7.

Trigger combination operator — Logical operator for creating trigger condition

AND (default) | OR

This parameter is indicated by the logic gate icon. Click the **Change operator** button to toggle between AND and OR.



The trigger condition can be composed of value comparisons of one or more signals. Combine these value comparisons with only one type of logical operator. Suppose three signals, A, B, and C, make up the trigger condition. The options are:

`A == 10 AND B == 'Falling edge' AND C == 0`

or

`A == 10 OR B == 'Falling edge' OR C == 0`

You cannot mix and match the combination operators. For more information, see “Triggers” on page 6-7.

Trigger time out — Maximum number of data capture IP core clock cycles within which trigger condition must occur in a trigger stage

1 (default) | integer from 1 to 65,536

Within this many data capture IP core clock cycles, the trigger condition must occur in a trigger stage in which you are enabling this parameter. You can specify any integer value from 1 to 65,536 according to your requirements. Select this parameter to enable trigger time out in a trigger stage. A trigger time out is not allowed in **Trigger Stage 1**.

Capture Condition

Enable capture condition logic — Option to enable capture condition logic

off (default) | on

Select this parameter to enable capture condition logic in the data capture IP core. Enable capture condition logic to use a capture condition to control which data to capture from the FPGA. The data capture IP core evaluates the capture condition at each clock cycle and captures only the data that satisfies the capture condition. For more information on capture conditions, see “Capture Conditions” on page 6-13.

Dependencies

To enable this parameter, in the **FPGA Data Capture Component Generator** tool, select **Include capture condition logic**.

Signal — Capture component signal name

character vector

This parameter is read-only. The signal names you specified as triggers at generation time are listed in the drop-down menu at the bottom. Click the + button to add a signal to the capture condition.

Dependencies

To enable this parameter, select **Enable capture condition logic**.

Operator — Operator to compare signals within capture condition

== | != | < | > | <= | >=

To compare signals, select one of these operators: ==, !=, <, >, <=, or >=. To compare signals containing X or x (don't-care value), specify either == or != operator.

Dependencies

To enable this parameter, select **Enable capture condition logic**.

Value — Value to compare signal to as part of overall capture condition

decimal | binary | hexadecimal | Low | High | Falling edge | Rising edge | Both edges

The capture condition can be composed of value comparisons of one or more signals. This parameter specifies the value to match for each signal.

For a multibit signal, specify a decimal, binary, or a hexadecimal value within the range of the data type associated with the signal. While providing hexadecimal or binary values, you can provide values with a combination of X or x (don't care value) to enable bit masking. While comparing the values, the capture condition discards place values with X or x and provides the output.

To separate a group of bits for better readability, you can use _ between bits. For example, you can represent a 32-bit binary value as 0b1010_XXXX_1011_XXXX_1110_XXXX_1111XXXX and a 32-bit hexadecimal value as 0xAB_CDEFX.

For boolean signals, select a level or edge condition. For more information, see “Capture Conditions” on page 6-13.

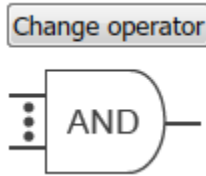
Dependencies

To enable this parameter, select **Enable capture condition logic**.

Capture condition combination operator — Logical operator for creating capture condition

AND (default) | OR

This parameter is indicated by the logic gate icon. Click the **Change operator** button to toggle between AND and OR.



The capture condition can be composed of value comparisons of one or more signals. Combine these value comparisons with only one type of logical operator. You cannot mix and match the combination operators. For more information, see “Capture Conditions” on page 6-13.

Dependencies

To enable this parameter, select **Enable capture condition logic**.

Data Types

Signal Name — Captured signal name
character vector

This parameter is read-only. It reflects the value you specified at generation time. This name is the name of the field in the structure variable.

Bit Width — Number of bits in the signal
positive integer

This parameter is read-only. It reflects the value you specified at generation time.

Data Type — Data type for captured data
built-in type | numeric type

The **Data Type** menu provides data type suggestions that match the bit width of the captured signal. This size is the width you specified for the port on the generated IP. You can type in this field to specify a custom data type. If the signal is 8, 16, or 32 bits, the default is `uint`. If the signal has one bit, the default is `boolean`. If the signal is a different width, the default is `numericType(0, bitWidth, 0)`.

The tool supports these data types, depending on the signal bit width: `boolean`, `uint8`, `int8`, `uint16`, `int16`, `half`, `uint32`, `int32`, `single`, `uint64`, `int64`, `double`, and `numericType`.

Version History

Introduced in R2017a

See Also

Tools

FPGA Data Capture Component Generator

Objects

`hdlverifier.FPGADataReader`

Blocks

FPGA Data Reader

Topics

“Capture Temperature Sensor Data from Xilinx FPGA Board Using FPGA Data Capture” on page 8-2

“Data Capture Workflow” on page 6-2

“Triggers” on page 6-7

“Capture Conditions” on page 6-13

hdlverifier.FPGADatReader

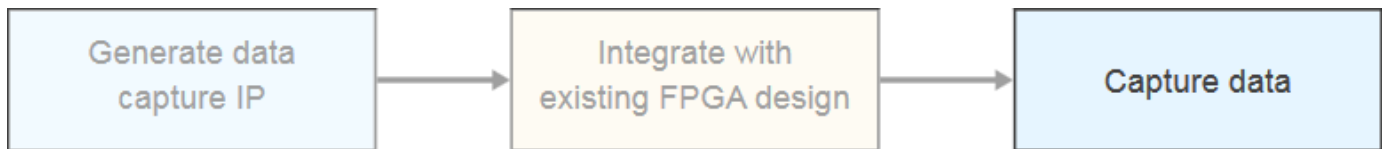
Package: hdlverifier

Capture data from live FPGA into MATLAB workspace

Description

The `hdlverifier.FPGADatReader` System object communicates with a generated HDL IP core running on an FPGA board to capture signals from the FPGA into MATLAB.

The `hdlverifier.FPGADatReader` System object cannot be created directly. To use it, run **FPGA Data Capture Component Generator** and generate your own customized `FPGADatReader` System object. You can use the generated object directly or use the wrapper tool, **FPGA Data Capture**, to set trigger condition, capture condition, and data types, and capture data.



Before you create the System object, you must have previously generated the customized data capture components. You must also have integrated the generated IP code into your project and deployed it to the FPGA. The object communicates with the FPGA over a JTAG or Ethernet cable. Make sure that the required cable is connected between the board and the host computer.

For a workflow overview, see “Data Capture Workflow” on page 6-2.

Note Alternatively, instead of using the `step` method to perform the operation defined by the System object, you can call the object with arguments, as if it were a function. For example, `y = step(obj, x)` and `y = obj(x)` perform equivalent operations.

Creation

`DC = mydc` creates a customized object, `DC`, that captures data from a design running on an FPGA. `mydc` is the component name you specified in the **FPGA Data Capture Component Generator** tool.

Properties

TimeOut — Number of seconds until aborting data capture

10 (default) | positive integer

If a trigger condition is enabled, but the HDL IP core does not detect the condition, the data capture request times out after the specified number of seconds. If the data capture is aborted, no data is returned to MATLAB.

When you use the tool for data capture, this property is ignored. Use the **Stop** button on the pop-up window to abort a capture using the tool.

NumCaptureWindows — Number of data capture recurrences

1 (default) | integer power of two

Specify the number of recurrences to capture. This value must be a power of two, and cannot be greater than *Sample depth*. When specifying the sample depth, consider the number of windows you plan to configure when reading the data, because together they impact the window depth of each capture window. The window depth is the *Sample depth* divided by the *Number of capture windows*. Specify *Sample depth* in the **FPGA Data Capture Component Generator** tool.

For example: If *Sample depth* is 4096 and *Number of capture windows* is 4, then each capture window has a window depth of 1024.

NumTriggerStages — Number of trigger stages for providing trigger conditions*M* (default) | integer from 1 to *M*

Specify the number of trigger stages. This value must be an integer from 1 to *M*, where *M* is set by the **Max trigger stages** parameter of the **FPGA Data Capture Component Generator** tool. When you specify the **Max trigger stages** parameter, consider the maximum number of trigger stages in which you plan to configure the trigger conditions to capture data.

For example, if **Max trigger stages** is 4, then **NumTriggerStages** can be 1, 2, 3, or 4.

TriggerPosition — Position of the trigger detection cycle within the capture buffer0 (default) | integer up to *window depth-1*

By default, the clock cycle when the trigger is detected is the first sample of the capture buffer. You can change the relative position of the trigger detection cycle within the capture buffer. A nondefault trigger position means that some samples are captured before the trigger occurs. You can set this parameter to an integer from 0 to *window depth-1*, inclusive. When the trigger position is equal to *window depth-1*, the last sample corresponds to the cycle when the trigger occurs. For more information, see “Triggers” on page 6-7.

EnableCaptureCtrl — Argument to enable capture condition logic

false (default) | true

Set this property to `true` to enable capture condition logic in the HDL IP core. Enable capture condition logic to use a capture condition to control which data to capture from the FPGA. The HDL IP core evaluates the capture condition at each clock cycle and captures only the data that satisfies the capture condition. For more information on capture conditions, see “Capture Conditions” on page 6-13.

Dependencies

To enable this parameter, in the **FPGA Data Capture Component Generator** tool, select **Include capture condition logic**.

CaptureMode — Capture mode

'blocking' (default) | 'nonblocking'

Specify the capture mode as one of these options:

- 'blocking' — The data capture object blocks MATLAB while retrieving captured data. In this capture mode, the JTAG resource is allocated to either FPGA data capture or AXI manager at a time.

- 'nonblocking' — The data capture object does not block MATLAB while retrieving captured data. In this capture mode, you can use FPGA data capture and AXI manager simultaneously.

If your development board has multiple FPGAs or multiple JTAG connections, the data capture software cannot detect the location of your FPGA in the JTAG chain. Specify these advanced parameters to locate the FPGA that contains the data capture IP core.

Advanced Board Setup

JTAGCableType — Type of JTAG cable used for communication with FPGA board

'auto' (default) | 'FTDI'

Specify this property if more than one JTAG cable is connected to the host computer. When not specified, the object will auto-detect the JTAG cable type, in this order:

- The FPGADatReader object first searches for a Digilent cable.
- If it does not find a Digilent JTAG cable, it searches for an FTDI cable.
- If it finds two cables of the same type, the object returns an error. Specify JTAGCableName to resolve it.
- If it finds two cables of different types, it will prioritize the Digilent cable. To use an FTDI cable, set this property to 'FTDI'.

Dependencies

To enable this property, in the **FPGA Data Capture Component Generator** tool, set the **Connection type** parameter to JTAG.

JTAGCableName — Name of JTAG cable used for data capture

'auto' (default) | character vector

Name of the JTAG cable used for data capture, specified as a character vector. Use this argument when the board is connected to two JTAG cables of the same type

Dependencies

To enable this property, in the **FPGA Data Capture Component Generator** tool, set the **Connection type** parameter to JTAG.

JTAGChainPosition — Position of FPGA in JTAG scan chain

0 (default) | positive integer

Position of the FPGA in the JTAG scan chain, specified as a positive integer.

Dependencies

To enable this property, in the **FPGA Data Capture Component Generator** tool, set the **Connection type** parameter to JTAG.

IRLengthBefore — Instruction register lengths before the FPGA

0 (default) | nonnegative integer

Number of instruction register lengths before the FPGA, specified as a nonnegative integer.

Dependencies

To enable this property, in the **FPGA Data Capture Component Generator** tool, set the **Connection type** parameter to JTAG.

IRLengthAfter — Instruction register lengths after the FPGA

0 (default) | nonnegative integer

Number of instruction register lengths after the FPGA, specified as a nonnegative integer.

Dependencies

To enable this property, in the **FPGA Data Capture Component Generator** tool, set the **Connection type** parameter to JTAG.

TckFrequency — JTAG clock frequency

15 (default) | integer

Specify the JTAG clock frequency, in MHz. For Xilinx FPGAs, the JTAG clock frequency is 33 or 66 MHz. The JTAG frequency depends on the type of cable and the maximum clock frequency supported by the FPGA board.

Dependencies

To enable this property, in the **FPGA Data Capture Component Generator** tool, set the **Connection type** parameter to JTAG.

DeviceAddress — IP address of Ethernet port on FPGA board

192.168.0.2 (default) | dotted-quad value

Specify the internet protocol (IP) address of the Ethernet port on the FPGA board as a dotted-quad value. The device IP address must be a set of four numbers consisting of integers in the range from 0 to 255 that are separated by three dots.

Dependencies

To enable this property, in the **FPGA Data Capture Component Generator** tool, set the **Connection type** parameter to Ethernet.

Port — UDP port number of FPGA board

50101 (default) | integer from 255 to 65,535

Specify the user datagram protocol (UDP) port number of the FPGA board as an integer from 255 to 65,535.

Dependencies

To enable this property, in the **FPGA Data Capture Component Generator** tool, set the **Connection type** parameter to Ethernet.

Object Functions

checkStatus

Check current status of FPGA data capture in nonblocking mode

clone

Create hdlverifier.FPGADataReader System object with same property values

collectData	Collect captured data from FPGA to host in nonblocking mode
displayCaptureCondition	Display overall capture condition
displayDataTypes	Display data types for all captured signals
displayTriggerCondition	Display overall trigger condition
isLocked	Locked status
launchApp	Open FPGA Data Capture app
release	Release control of JTAG interface
setCaptureCondition	Configure comparison for each signal value
setCaptureConditionCombinationOperator	Configure operator that combines individual signal value comparisons into overall capture condition
setCaptureConditionComparisonOperator	Configure operator that compares individual signal values within capture condition
setDataType	Configure data type for the data captured from a signal
setNumberOfTriggerStages	Configure number of trigger stages for capturing data
setRunImmediateFlag	Configure data capture to run immediately without any trigger condition
setTriggerCombinationOperator	Configure operator that combines individual signal value comparisons into overall trigger condition
setTriggerComparisonOperator	Configure operator that compares individual signal values within trigger condition
setTriggerCondition	Configure each signal value comparison
setTriggerTimeOut	Configure maximum number of FDC IP core clock cycles within which trigger condition must occur in a trigger stage
step	Capture one buffer of data from HDL IP core running on FPGA
stop	Stop FPGA data capture execution based on current status in nonblocking mode

Examples

Capture Data from FPGA over JTAG Connection

This example shows how to use the `hdlverifier.FPGADatReader` System object™ to capture data from a design running on an FPGA over a JTAG connection. The `hdlverifier.FPGADatReader` System object provides a programmatic way to configure and capture data. Generate an FPGA data reader System object by using the **FPGA Data Capture Component Generator** tool. Then use the generated System object directly to set data types and trigger conditions and capture data.

Generate `hdlverifier.FPGADatReader` System Object

To generate a customized `hdlverifier.FPGADatReader` System object, open the **FPGA Data Capture Component Generator** tool by entering following command at the MATLAB® command prompt. To use this tool, you must have an existing HDL design and FPGA project.

```
generateFPGADatCaptureIP;
```

This example uses a generated object, `mydc`, that defines two signals for data capture. Signal A is 1 bit and signal B is 8 bits. Both signals are also available for use in trigger conditions. The sample

depth is 4096 samples. To configure the `hdlverifier.FPGADataReader` System object to operate on these two signals, follow these steps.

- 1 Add one row to the **Ports** table by clicking the **Add** button once.
- 2 Name the first signal A and the second signal B.
- 3 Set **Bit Width** of the two signals to 1 and 8, respectively.
- 4 Specify **Generated IP name** as `mydc`.
- 5 Set **FPGA vendor** to `Xilinx`.
- 6 Set **Sample depth** to 4096.
- 7 Set **Max trigger stages** to 2.

This figure shows these tool settings.

FPGA Data Capture Component Generation

Description
Generate an HDL IP for integration into your FPGA design.

Specify port names and bit-widths for the interface of the generated IP, and specify how many samples the IP captures at a time.

Generate data capture IP → Integrate with existing FPGA design → Capture data

[Read more about the data capture workflow](#)

Ports

Port Name	Bit Width	Use As
A	1	Both trigger and data
B	8	Both trigger and data

Target

Generated IP name: `mydc`

FPGA vendor: `Xilinx`

Generated IP language: `VHDL`

Connection type: `JTAG`

Destination folder: `hdlsrc`

Capture

Sample depth: `4096`

Max trigger stages: `2`

Include capture condition logic

Generate **Cancel** **Help**

To generate the `hdlverifier.FPGADataReader` System object, click **Generate**. A report shows the results of the generation. Integrate the generated IP code into your existing FPGA project and deploy

it to the FPGA. The System object communicates with the FPGA over a JTAG cable. Make sure that the JTAG cable connects the board and the host computer.

Go to the directory where the `hdlverifier.FPGADatReader` System object is generated.

```
cd hdlsrc;
```

Create a data capture object using your generated System object.

```
captureData = mydc
captureData =
    mydc with properties:
        Connection: 'JTAG'
        IsConditionalCapture: 0
        TriggerPosition: 0
        NumCaptureWindows: 1
        NumTriggerStages: 2
        Timeout: 10
        EnableCaptureCtrl: 0
        CaptureMode: 'blocking'
        JTAGCableName: 'auto'
        JTAGCableType: 'auto'
        JTAGChainPosition: 0
        IRLengthBefore: 0
        IRLengthAfter: 0
        TckFrequency: 15
        MaxNumTriggerStages: 2
```

Capture Data Immediately

Create a data capture object. The default trigger condition is to trigger immediately. The default configuration of the generated object does not enable any signals as part of the overall trigger condition.

```
captureData = mydc;
```

Display the data types of the captured signals. The default data type for an 8-bit signal is `uint8`.

```
displayDataTypes(captureData);
```

```
Signal Name : Data Type
Capture_Window : uint32
Trigger_Position : boolean
A : boolean
B : uint8
```

Call the object. The data is captured immediately from the FPGA.

```
[Capture_Window,Trigger_Position,dataOut] = captureData();
```

The captured data is returned as a structure containing a field for the `Capture_Window` signal, a field for the `Trigger_Position` signal, and a field for each signal captured by the data capture object. The `dataOut` structure contains field `A`, which is a vector of 4096 logical values, and field `B`, which is a vector of 4096 `uint8` values.

Capture Data on Trigger Event

To debug signal values near a specific event, set up a trigger condition. The trigger condition can be composed of value comparisons of one or more signals. You can combine these value comparisons with only one type of logical operator, either an AND or OR operator.

Define a trigger condition to capture data when the FPGA detects a high value on A at the same time as signal B is greater than 7.

```
captureData = mydc;  
setTriggerCondition(captureData, 'A', true, 'High');  
setTriggerCondition(captureData, 'B', true, 7);  
setTriggerComparisonOperator(captureData, 'B', '>');
```

Display the overall trigger condition.

```
displayTriggerCondition(captureData);
```

The **trigger condition is:**
A==High and B>7

Call the object to capture data on the specified trigger event.

```
[~,~,dataOut] = captureData();
```

Define a trigger condition to capture data when the FPGA detects a high value on A at the same time as the value of signal B is 0xAX. In signal B, the trigger condition checks the leftmost 4 bits provided as A and ignores the rightmost 4 bits provided as X (X indicates bits for the function to ignore).

```
captureData = mydc;  
setTriggerCondition(captureData, 'A', true, 'High');  
setTriggerCondition(captureData, 'B', true, '0xAX');
```

Display the overall trigger condition.

```
displayTriggerCondition(captureData);
```

The **trigger condition is:**
A==High and B==0xAX

Call the object to capture data on the specified trigger event.

```
[~,~,dataOut] = captureData();
```

dataOut is returned after the HDL IP core detects the trigger condition from the signals on the FPGA. dataOut contains samples starting from the cycle when the trigger condition is detected.

Capture Data on Multiple Trigger Events

Define trigger conditions to capture data when the FPGA detects two trigger conditions in sequence.

- Trigger condition 1 - High value on A at the same time as signal B is equal to 7
- Trigger condition 2 - High value on A at the same time as signal B is greater than 15

```
captureData = mydc;  
setNumberOfTriggerStages(captureData, 2);  
setTriggerCondition(captureData, 'A', true, 'High');  
setTriggerCondition(captureData, 'B', true, 7);
```

```
setTriggerCondition(captureData, 'A', true, 'High', 2);
setTriggerCondition(captureData, 'B', true, 15, 2);
setTriggerComparisonOperator(captureData, 'B', '>', 2);
```

Display the trigger condition. By default, the function displays the trigger condition in trigger stage 1.

```
displayTriggerCondition(captureData);
```

```
The trigger condition is:
A==High and B==7
```

Display the trigger condition in trigger stage 2.

```
displayTriggerCondition(captureData, 2);
```

```
The trigger condition is:
A==High and B>15
```

Call the object to capture data on the specified trigger events.

```
[~,~,dataOut] = captureData();
```

`dataOut` is returned when the HDL IP core detects the trigger condition set in trigger stage 2 after detecting the trigger condition set in trigger stage 1, satisfying the set sequence.

Capture Fixed-Point Data

The default data type for an 8-bit signal is `uint8`, but in your HDL design, you can represent the signal using a fixed-point number. Set the data type of the captured data to cast it to the fixed-point representation.

```
captureData = mydc;
setDataTypes(captureData, 'B', numerictype(1,8,6));
```

Display the data types of the captured signals.

```
displayDataTypes(captureData);
```

```
Signal Name : Data Type
Capture_Window : uint32
Trigger_Position : boolean
A : boolean
B : numerictype(1,8,6)
```

Call the object to capture data on the specified trigger event.

```
[~,~,dataOut] = captureData();
```

In the `dataOut` structure, field `A` is a vector of 4096 logical values and field `B` is a vector of 4096 signed 8-bit fixed-point values, with 6 fractional bits.

Version History

Introduced in R2017a

See Also

Tools

FPGA Data Capture Component Generator | FPGA Data Capture

Blocks

FPGA Data Reader

Topics

“Capture Temperature Sensor Data from Xilinx FPGA Board Using FPGA Data Capture” on page 8-2

“Debug IP Core Using FPGA Data Capture” (HDL Coder)

“Data Capture Workflow” on page 6-2

“Triggers” on page 6-7

“Capture Conditions” on page 6-13

clone

Create `hdlverifier.FPGADataReader` System object with same property values

Syntax

```
DC2 = clone(DC)
```

Description

`DC2 = clone(DC)` creates a copy of the specified `hdlverifier.FPGADataReader` System object, with the same property values. If a System object is locked, then `clone` creates a copy that is also locked and has states initialized to the same values as the original. If a System object is not locked, then `clone` creates a new unlocked System object with uninitialized states.

Input Arguments

DC — Data capture System object

`hdlverifier.FPGADataReader` System object

Customized data capture object, specified as an `hdlverifier.FPGADataReader` System object.

Output Arguments

DC2 — Data capture object

`hdlverifier.FPGADataReader` System object

Data capture object, with the same property values as input DC, returned as an `hdlverifier.FPGADataReader` System object.

Version History

Introduced in R2017a

See Also

`hdlverifier.FPGADataReader` | **FPGA Data Capture Component Generator** | **FPGA Data Capture**

displayDataTypes

Display data types for all captured signals

Syntax

```
displayDataTypes(DC)
```

Description

`displayDataTypes(DC)` displays the data type configured for each data capture signal. The default data type depends on the bit width of the captured signal in the specified data capture System object. This size is the width you specified for the port on the generated IP. If the signal is 8, 16, or 32 bits, the default data type is `uint`. If the signal has one bit, the default data type is `boolean`. If the signal is a different width, the default data type is `numerictype(0,bitWidth,0)`.

To modify the data type of a signal, use the `setDataType` object function. The function supports these data types, depending on the bit width of the captured signal: `boolean`, `uint8`, `int8`, `uint16`, `int16`, `half`, `uint32`, `int32`, `single`, `uint64`, `int64`, `double`, and `numerictype`.

Input Arguments

DC — Data capture System object

`hdlverifier.FPGADataReader` System object

Customized data capture object, specified as an `hdlverifier.FPGADataReader` System object.

Version History

Introduced in R2017a

See Also

`hdlverifier.FPGADataReader` | **FPGA Data Capture Component Generator** | **FPGA Data Capture**

displayTriggerCondition

Display overall trigger condition

Syntax

```
displayTriggerCondition(DC)  
displayTriggerCondition(DC,N)
```

Description

`displayTriggerCondition(DC)` displays the signal value comparisons and logical operator that define the overall trigger condition in trigger stage 1. DC is a customized data capture object.

`displayTriggerCondition(DC,N)` displays the signal value comparisons and logical operator that define the overall trigger condition in a trigger stage specified by N. DC is a customized data capture object.

Input Arguments

DC — Customized data capture object

`hdlverifier.FPGADataReader` System object

Customized data capture object, specified as an `hdlverifier.FPGADataReader` System object.

N — Trigger stage

integer from 1 to *M*

Trigger stage, specified as an integer from 1 to *M*, where *M* is set by the **Max trigger stages** parameter of the **FPGA Data Capture Component Generator** tool. Use N to display the trigger condition in Nth trigger stage. If you do not specify N, by default, the function displays the trigger condition in trigger stage 1.

Version History

Introduced in R2017a

See Also

Objects

`hdlverifier.FPGADataReader`

Tools

FPGA Data Capture Component Generator | **FPGA Data Capture**

isLocked

Locked status

Syntax

```
tf = isLocked(DC)
```

Description

`tf = isLocked(DC)` returns the locked status, of the specified `hdlverifier.FPGADataReader System` object.

`isLocked` returns a logical value that indicates whether input attributes and nontunable properties for the object are locked. The object performs an internal initialization the first time that you call the object. This initialization locks nontunable properties and input specifications, such as the dimensions, complexity, and data type of the input data. After locking, `isLocked` returns a `true` value.

Input Arguments

DC — Customized data capture object

`hdlverifier.FPGADataReader System` object

Customized data capture object, specified as an `hdlverifier.FPGADataReader System` object.

Output Arguments

tf — True or false result

1 | 0

True or false result indicating the locked status of the input System object DC, returned as a 1 or 0 of data type `logical`. A 1 indicates the System object is locked.

Version History

Introduced in R2017a

See Also

`hdlverifier.FPGADataReader` | **FPGA Data Capture Component Generator** | **FPGA Data Capture**

launchApp

Open FPGA Data Capture app

Syntax

```
launchApp(DC)
```

Description

launchApp(DC) opens the **FPGA Data Capture** app, which captures data from a design running on an FPGA and returns the captured data to the MATLAB workspace. The app is a wrapper on the specified `hdlverifier.FPGADataReader` System object. Changes that you make in the app are saved in the properties of the System object.

You can configure the data types of the returned values and set up a trigger condition to control when the data is captured. You must have previously generated the customized data capture components by using **FPGA Data Capture Component Generator**. You must also have integrated the generated IP code into your project and deployed it to the FPGA. The tool communicates with the FPGA over a JTAG cable. You must connect the JTAG cable between the board and the host computer.

Input Arguments

DC — Customized data capture object

`hdlverifier.FPGADataReader` System object

Customized data capture object, specified as an `hdlverifier.FPGADataReader` System object.

Version History

Introduced in R2017a

See Also

`hdlverifier.FPGADataReader` | **FPGA Data Capture Component Generator** | **FPGA Data Capture**

release

Release control of JTAG interface

Syntax

```
release(DC)
```

Description

release(DC) releases system resources, including control of the JTAG interface, of the specified `hdlverifier.FPGADataReader System` object. Releasing the System object enables you to change its properties and input characteristics. While the System object exists and is locked, no other processes can use the JTAG cable.

Input Arguments

DC — Customized data capture object

`hdlverifier.FPGADataReader System` object

Customized data capture object, specified as an `hdlverifier.FPGADataReader System` object.

Version History

Introduced in R2017a

See Also

`hdlverifier.FPGADataReader` | **FPGA Data Capture Component Generator** | **FPGA Data Capture**

setDataType

Configure data type for the data captured from a signal

Syntax

```
setDataType(DC, name, type)
```

Description

`setDataType(DC, name, type)` specifies the data type, `type`, for the data captured from a signal, `name`.

Input Arguments

DC — Customized data capture object

`hdlverifier.FPGADataReader` System object

Customized data capture object, specified as an `hdlverifier.FPGADataReader` System object.

name — Trigger component signal

character vector

Specify a signal name matching one that you configured when you generated the object. The signal must be configured as a data signal.

type — Data type for the captured data

built-in data type | `numerictype`

The bit width of the data type must match the bit width of the captured signal. This size is the width you specified for the port on the generated IP.

The function supports these data types, depending on the bit width of the captured signal: `boolean`, `uint8`, `int8`, `uint16`, `int16`, `half`, `uint32`, `int32`, `single`, `uint64`, `int64`, `double`, and `numerictype`.

Version History

Introduced in R2017a

See Also

`hdlverifier.FPGADataReader` | **FPGA Data Capture Component Generator** | **FPGA Data Capture**

setTriggerComparisonOperator

Configure operator that compares individual signal values within trigger condition

Syntax

```
setTriggerComparisonOperator(DC,name,operator)
setTriggerComparisonOperator(DC,name,operator,N)
```

Description

`setTriggerComparisonOperator(DC,name,operator)` configures a comparison operator that compares individual signal values within the trigger condition in trigger stage 1. `DC` is a customized data capture object, `name` is the name of a trigger component signal.

`setTriggerComparisonOperator(DC,name,operator,N)` configures a comparison operator that compares individual signal values within the trigger condition in a trigger stage specified by `N`. `DC` is a customized data capture object, `name` is the name of a trigger component signal.

Input Arguments

DC — Customized data capture object

`hdlverifier.FPGADataReader` System object

Customized data capture object, specified as an `hdlverifier.FPGADataReader` System object.

name — Name of trigger component signal

character vector

Name of a trigger component signal, specified as a character vector. This name must match one of the signal names configured on creation of the input System object `DC`. The signal must be configured as a possible trigger signal.

operator — Operator to compare signals within trigger condition

`==` (default) | `!=` | `<` | `>` | `<=` | `>=`

Operator to compare signals within the trigger condition, specified as one of these operators: `==` (default), `!=`, `<`, `>`, `<=`, or `>=`.

The trigger condition comprises value comparisons of one or more signals. For a multibit signal, specify one of these operators: `==` (default), `!=`, `<`, `>`, `<=`, or `>=`. For a trigger condition containing `X` or `x` (don't-care value), specify either `==` or `!=` operators. For a logical signal, specify one of these operators: `==` or `!=`. For details on trigger conditions, see "Triggers" on page 6-7.

N — Trigger stage

integer from 1 to M

Trigger stage, specified as an integer from 1 to M , where M is set by the Max trigger stages on page 7-0 `parameter` of the **FPGA Data Capture Component Generator** tool. Use `N` to set the trigger comparison operator in N th trigger stage. If you do not specify `N`, by default, the function sets the trigger comparison operator in trigger stage 1.

Version History

Introduced in R2019b

See Also

hdlverifier.FPGADataReader | **FPGA Data Capture Component Generator** | **FPGA Data Capture**

setTriggerCombinationOperator

Configure operator that combines individual signal value comparisons into overall trigger condition

Syntax

```
setTriggerCombinationOperator(DC,operator)  
setTriggerCombinationOperator(DC,operator,N)
```

Description

`setTriggerCombinationOperator(DC,operator)` configures the logical operator that combines comparisons of individual signals into an overall trigger condition in trigger stage 1. DC is a customized data capture object.

`setTriggerCombinationOperator(DC,operator,N)` configures the logical operator that combines comparisons of individual signals into an overall trigger condition in a trigger stage specified by N. DC is a customized data capture object.

Input Arguments

DC — Customized data capture object

`hdlverifier.FPGADataReader` System object

Customized data capture object, specified as an `hdlverifier.FPGADataReader` System object.

operator — Logical operator to combine comparisons of individual signals into trigger condition

AND | OR

Logical operator to combine comparisons of individual signals into a trigger condition, specified as AND or OR. The trigger condition comprises value comparisons of one or more signals. To combine value comparisons, you can use only one type of logical operator. For example, suppose three signals, A, B, and C, make up the trigger condition. The options are:

- `A == 10 AND B == 'Falling edge' AND C == 0`
- `A == 10 OR B == 'Falling edge' OR C == 0`

You cannot mix and match the combination operators. For details on trigger conditions, see “Triggers” on page 6-7.

N — Trigger stage

integer from 1 to *M*

Trigger stage, specified as an integer from 1 to *M*, where *M* is set by the Max trigger stages on page 7-0 parameter of the **FPGA Data Capture Component Generator** tool. Use N to set the combination operator in Nth trigger stage. If you do not specify N, by default, the function sets the combination operator in trigger stage 1.

Version History

Introduced in R2017a

See Also

hdlverifier.FPGADataReader | **FPGA Data Capture Component Generator** | **FPGA Data Capture**

setTriggerCondition

Configure each signal value comparison

Syntax

```
setTriggerCondition(DC,name,enable,value)
setTriggerCondition(DC,name,enable,value,N)
```

Description

`setTriggerCondition(DC,name,enable,value)` configures a trigger value comparison for signal name in trigger stage 1. DC is a customized data capture object. The `enable` argument indicates whether this signal is part of the overall trigger condition.

`setTriggerCondition(DC,name,enable,value,N)` configures a trigger value comparison for signal name in a trigger stage specified by N. DC is a customized data capture object. The `enable` argument indicates whether this signal is part of the overall trigger condition.

Input Arguments

DC — Customized data capture object

`hdlverifier.FPGADataReader` System object

Customized data capture object, specified as an `hdlverifier.FPGADataReader` System object.

name — Name of trigger component signal

character vector

Name of trigger component signal, specified as a character vector.

This name must match one of the signal names configured on creation of the input System object DC. The signal must be configured as a possible trigger signal.

Data Types: char

enable — Indication that signal is part of trigger condition

true or 1 | false or 0

Indication that the signal is part of the trigger condition, specified as a numeric or logical 1 (true) or 0 (false). To use this signal in the overall trigger condition, set this value to 1 (true). When you set this value to 0 (false), the signal is not used for the overall trigger condition.

value — Value to compare this signal to as part of the trigger condition

decimal | binary | hexadecimal | 'Low' | 'High' | 'Rising edge' | 'Falling edge' | 'Both edges'

The trigger condition comprises value comparisons of one or more signals. This input specifies the value to match for each signal.

For a multibit signal, specify a decimal, binary, or a hexadecimal value within the range of the data type associated with the signal. While providing hexadecimal or binary values, you can provide values

with a combination of X or x (don't care value) to enable bit masking. That means, while comparing the values, the trigger condition ignores the place values with X or x and provides the output.

To separate a group of bits for better readability, you can use `_` between bits. For example, you can represent a 32-bit binary value as `'0b1010_XXXX_1011_XXXX_1110_XXXX_1111XXXX'` and a 32-bit hexadecimal value as `'0xAB_CDEFX'`.

For logical signals, specify a string that indicates the level or edge to match. For more information, see “Triggers” on page 6-7.

N – Trigger stage

integer from 1 to *M*

Trigger stage, specified as an integer from 1 to *M*, where *M* is set by the Max trigger stages on page 7-0 `parameter` of the **FPGA Data Capture Component Generator** tool. Use *N* to set the trigger condition in *N*th trigger stage. If you do not specify *N*, by default, the function sets the trigger condition in trigger stage 1.

Version History

Introduced in R2017a

See Also

`hdlverifier.FPGADataReader` | **FPGA Data Capture Component Generator** | **FPGA Data Capture**

step

Capture one buffer of data from HDL IP core running on FPGA

Syntax

```
dataOut = step(DC)
```

Description

Note Alternatively, instead of using the `step` object function to perform the operation defined by the System object, you can call the System object with arguments, as if it were a function. For example, `y = step(obj, x)` and `y = obj(x)` perform equivalent operations.

`dataOut = step(DC)` captures live signal data from a design running on an FPGA. The FPGA must contain an HDL IP core generated from the **FPGA Data Capture Component Generator** tool. `dataOut` is a structure that contains a field for each signal captured. Call the `setDataTypes` object function to specify the data type of each captured signal.

If at least one signal is enabled as part of the trigger condition, the HDL IP core waits for a match of the trigger condition and captures the data. If no signals are enabled as part of the trigger condition, the HDL IP core captures and returns the buffered data immediately. When you create the object, no trigger condition is set by default. Call the `setTriggerCondition` and `setTriggerCombinationOperator` object functions to configure a trigger condition.

Input Arguments

DC — Customized data capture object

FPGADataReader System object

Customized data capture object, specified as an `hdlverifier.FPGADataReader` System object.

Output Arguments

dataOut — Captured data

structure

Captured data, returned as a structure containing a field for the `Capture_Window` signal, a field for the `Trigger_Position` signal, and a field for each signal captured by FPGA data capture. The captured signal field is a vector of *Sample depth* values for each signal requested for data capture at generation time. The fields of the structure have these signal names.

- `Capture_Window` — This signal indicates the current capture window.
- `Trigger_Position` — This signal indicates the position of the trigger detection clock cycle within a capture buffer.
- All remaining fields — The signal names you specified at generation time.

Version History

Introduced in R2017a

See Also

hdlverifier.FPGADatReader | **FPGA Data Capture Component Generator** | **FPGA Data Capture**

Topics

"Triggers" on page 6-7

setNumberOfTriggerStages

Configure number of trigger stages for capturing data

Syntax

```
setNumberOfTriggerStages(DC,N)
```

Description

`setNumberOfTriggerStages(DC,N)` specifies an integer value configures the number of trigger stages, *N*, for capturing data. *DC* is customized data capture object.

Input Arguments

DC — Customized data capture object

`hdlverifier.FPGADataReader` System object

Customized data capture object, specified as an `hdlverifier.FPGADataReader` System object.

N — Total number of trigger stages

integer from 1 to *M*

Total number of trigger stages in which you want to capture the data, specified as an integer from 1 to *M*, where *M* is set by the Max trigger stages on page 7-0 parameter of the **FPGA Data Capture Component Generator** tool.

Version History

Introduced in R2020b

See Also

`hdlverifier.FPGADataReader` | **FPGA Data Capture Component Generator** | **FPGA Data Capture**

setTriggerTimeOut

Configure maximum number of FDC IP core clock cycles within which trigger condition must occur in a trigger stage

Syntax

```
setTriggerTimeOut(DC,enable,value,N)
```

Description

`setTriggerTimeOut(DC,enable,value,N)` configures the maximum number of FPGA Data Capture (FDC) IP core clock cycles, within which the trigger condition must occur in a trigger stage specified by `N`. `DC` is a customized data capture object. Use `enable` argument to enable the trigger time out in trigger stage `N`, specify the number of FDC IP core clock cycles using `value` argument.

Input Arguments

DC — Customized data capture object

`hdlverifier.FPGADatReader` System object

Customized data capture object, specified as an `hdlverifier.FPGADatReader` System object.

enable — Indication that trigger time out is part of specified trigger stage

`true` or `1` | `false` or `0`

Indication that the trigger time out is part of the trigger stage, specified as a numeric or logical `1` (`true`) or `0` (`false`). To use the trigger time out in a particular trigger stage, set this value to `1` (`true`). When you set this value to `0` (`false`), the trigger time out is not used for the specified trigger stage.

value — Number of FDC IP core clock cycles

integer from 1 to 65,536

Specify an integer from 1 to 65,536. Within this many FDC IP core clock cycles, the trigger condition must occur in a trigger stage specified by `N`.

N — Trigger stage

integer from 2 to M

Trigger stage, specified as an integer from 2 to M , where M is set by the Max trigger stages on page 7-0 parameter of the **FPGA Data Capture Component Generator** tool. Use `N` to set the trigger time out in N th trigger stage. Trigger time out is not allowed for trigger stage 1.

Version History

Introduced in R2020b

See Also

hdlverifier.FPGADataReader | **FPGA Data Capture Component Generator** | **FPGA Data Capture**

setRunImmediateFlag

Configure data capture to run immediately without any trigger condition

Syntax

```
setRunImmediateFlag(DC,value)
```

Description

`setRunImmediateFlag(DC,value)` specifies whether the data capture object `DC` runs in immediate mode. If `value` is `true`, the data capture object runs in immediate mode and captures data immediately without checking for the trigger condition. In this mode, the data capture object captures data either at each clock cycle or based on a capture condition if capture condition logic is enabled. If `value` is `false`, the data capture object does not run in immediate mode.

Input Arguments

DC — Customized data capture object

`hdlverifier.FPGADataReader` System object

Customized data capture object, specified as an `hdlverifier.FPGADataReader` System object.

value — Flag to capture data immediately

`true` or `1` | `false` or `0`

Flag to capture data immediately, specified as a numeric or logical `1` (`true`) or `0` (`false`). To ignore the trigger condition and capture data immediately, set this value to `1` (`true`). In this case, the generated object does not enable any signals as part of the overall trigger condition. To capture data that includes a particular event in the FPGA logic, configure a trigger condition and set this value to `0` (`false`). In this case, the data capture object waits until the trigger condition is true, then captures the data.

Version History

Introduced in R2022a

See Also

Objects

`hdlverifier.FPGADataReader`

Tools

FPGA Data Capture Component Generator | FPGA Data Capture

displayCaptureCondition

Display overall capture condition

Syntax

```
displayCaptureCondition(DC)
```

Description

`displayCaptureCondition(DC)` displays the signal value comparisons and logical operator that define the overall capture condition. DC is a customized data capture object.

Examples

Display Capture Condition

This example uses a customized data capture object, DC, that defines two signals for both trigger and data capture. Signal A is 1 bit and signal B is 8 bits.

Enable capture condition logic.

```
DC.EnableCaptureCtrl = true;
```

To enable capture condition logic, you must select the **Include capture condition logic** parameter while generating the data capture IP core using the **FPGA Data Capture Component Generator** tool.

Set up a capture condition to capture data when the FPGA detects a high value on signal A at the same time as signal B is greater than 7.

```
setCaptureCondition(DC, 'A', true, 'High');  
setCaptureCondition(DC, 'B', true, 7);  
setCaptureConditionComparisonOperator(DC, 'B', '>');
```

Combine comparisons of signals A and B into an overall capture condition using an AND operator.

```
setCaptureConditionCombinationOperator(DC, 'AND');
```

Display the overall capture condition.

```
displayCaptureCondition(DC);
```

```
The capture condition is:  
A==High and B>7
```

Input Arguments

DC — Customized data capture object

`hdlverifier.FPGADataReader` System object

Customized data capture object, specified as an `hdlverifier.FPGADataReader` System object.

Version History

Introduced in R2022a

See Also

Objects

`hdlverifier.FPGADataReader`

Tools

FPGA Data Capture Component Generator | FPGA Data Capture

setCaptureCondition

Configure comparison for each signal value

Syntax

```
setCaptureCondition(DC, name, enable, value)
```

Description

`setCaptureCondition(DC, name, enable, value)` configures a capture value comparison for the signal name. DC is a customized data capture object. The `enable` argument indicates whether this signal is part of the overall capture condition.

Examples

Set Up Capture Condition

This example uses a customized data capture object, DC, that defines two signals for both trigger and data capture. Signal A is 1 bit and signal B is 8 bits.

Enable capture condition logic.

```
DC.EnableCaptureCtrl = true;
```

To enable capture condition logic, you must select the **Include capture condition logic** parameter while generating the data capture IP core using the **FPGA Data Capture Component Generator** tool.

Set up a capture condition to capture data when the FPGA detects a high value on signal A and a value 17 on signal B.

```
setCaptureCondition(DC, 'A', true, 'High');  
setCaptureCondition(DC, 'B', true, uint8(17));
```

Input Arguments

DC — Customized data capture object

`hdlverifier.FPGADataReader` System object

Customized data capture object, specified as an `hdlverifier.FPGADataReader` System object.

name — Name of capture component signal

character vector

Name of the capture component signal, specified as a character vector.

This name must match one of the signal names configured on creation of the input System object DC. The signal must be configured as a possible trigger signal.

Data Types: `char`

enable — Indication that signal is part of capture condition

true | false

Indication that the signal is part of the capture condition, specified as `true` or `false`. To use this signal in the overall capture condition, set this value to `true`. When you set this value to `false`, the signal is not used for the overall capture condition.

value — Value to compare signal to as part of capture condition

decimal | binary | hexadecimal | 'Low' | 'High' | 'Rising edge' | 'Falling edge' | 'Both edges'

Value to compare the signal to as part of the capture condition, specified as one of the following.

- Decimal, binary, or hexadecimal value — For a multibit signal, specify a value within the range of the data type associated with the signal. If you specify a binary or hexadecimal value, you can use an `X` or `x` to indicate signals for the function to ignore during the value comparison.

To separate a group of bits for better readability, you can use `_` between bits. For example, you can represent a 32-bit binary value as `'0b1010_XXXX_1011_XXXX_1110_XXXX_1111XXXX'` and a 32-bit hexadecimal value as `'0xAB_CDEFX'`.

- 'Low', 'High', 'Rising edge', 'Falling edge', or 'Both edges' — For a logical signal, specify a string that indicates the level or edge to match. For more information, see “Capture Conditions” on page 6-13.

Version History

Introduced in R2022a

See Also

Objects

hdlverifier.FPGADataReader

Tools

FPGA Data Capture Component Generator | FPGA Data Capture

setCaptureConditionCombinationOperator

Configure operator that combines individual signal value comparisons into overall capture condition

Syntax

```
setCaptureConditionCombinationOperator(DC, operator)
```

Description

`setCaptureConditionCombinationOperator(DC, operator)` configures the logical operator `operator` that combines comparisons of individual signals into an overall capture condition. `DC` is a customized data capture object.

Examples

Combine Comparisons of Individual Signals into Overall Capture Condition

This example uses a customized data capture object, `DC`, that defines two signals for both trigger and data capture. Signal A is 1 bit and signal B is 8 bits.

Enable capture condition logic.

```
DC.EnableCaptureCtrl = true;
```

To enable capture condition logic, you must select the **Include capture condition logic** parameter while generating the data capture IP core using the **FPGA Data Capture Component Generator** tool.

Set up a capture condition to capture data when the FPGA detects a high value on signal A at the same time as signal B is equal to 17.

```
setCaptureCondition(DC, 'A', true, 'High');  
setCaptureCondition(DC, 'B', true, uint8(17));
```

Combine comparisons of signals A and B into an overall capture condition using an AND operator.

```
setCaptureConditionCombinationOperator(DC, 'AND');
```

Input Arguments

DC — Customized data capture object

`hdlverifier.FPGADataReader` System object

Customized data capture object, specified as an `hdlverifier.FPGADataReader` System object.

operator — Logical operator to combine comparisons of individual signals into capture condition

'AND' (default) | 'OR'

Logical operator to combine comparisons of individual signals into a capture condition, specified as 'AND' or 'OR'. The capture condition comprises value comparisons of one or more signals. To combine value comparisons, you can use only one type of logical operator. For example, suppose three signals, A, B, and C, make up the capture condition. The options are:

- `A == 10 AND B == 'Falling edge' AND C == 0`
- `A == 10 OR B == 'Falling edge' OR C == 0`

You cannot mix and match the combination operators. For details on capture conditions, see “Capture Conditions” on page 6-13.

Version History

Introduced in R2022a

See Also

Objects

`hdlverifier.FPGADatReader`

Tools

FPGA Data Capture Component Generator | FPGA Data Capture

setCaptureConditionComparisonOperator

Configure operator that compares individual signal values within capture condition

Syntax

```
setCaptureConditionComparisonOperator(DC,name,operator)
```

Description

`setCaptureConditionComparisonOperator(DC,name,operator)` configures a comparison operator `operator` that compares individual signal values within the capture condition. `DC` is a customized data capture object. `name` is the name of a capture component signal.

Examples

Set Up Capture Condition Using Comparison Operator

This example uses a customized data capture object, `DC`, that defines two signals for both trigger and data capture. Signal A is 1 bit and signal B is 8 bits.

Enable capture condition logic.

```
DC.EnableCaptureCtrl = true;
```

To enable capture condition logic, you must select the **Include capture condition logic** parameter while generating the data capture IP core using the **FPGA Data Capture Component Generator** tool.

Set up a capture condition to capture data when the FPGA detects a high value on signal A at the same time as signal B is greater than 7.

```
setCaptureCondition(DC,'A',true,'High');  
setCaptureCondition(DC,'B',true,7);  
setCaptureConditionComparisonOperator(DC,'B','>');
```

Input Arguments

DC — Customized data capture object

`hdlverifier.FPGADataReader` System object

Customized data capture object, specified as an `hdlverifier.FPGADataReader` System object.

name — Name of capture component signal

character vector

Name of a capture component signal, specified as a character vector. This name must match one of the signal names configured on creation of the input System object `DC`. The signal must be configured as a possible trigger signal.

operator — Operator to compare signals within capture condition`== (default) | != | < | > | <= | >=`

Operator to compare signals within the capture condition, specified as one of these operators: ==, !=, <, >, <=, or >=.

The capture condition comprises value comparisons of one or more signals. For a multibit signal, specify one of these operators: == (default), !=, <, >, <=, or >=. For a capture condition containing X or x (which indicate bits for the function to ignore), specify either the == or != operators. For a logical signal, specify either the == or != operators. For details on capture conditions, see “Capture Conditions” on page 6-13.

Version History

Introduced in R2022a

See Also

Objects

hdlverifier.FPGADatReader

Tools

FPGA Data Capture Component Generator | FPGA Data Capture

checkStatus

Check current status of FPGA data capture in nonblocking mode

Syntax

```
status = checkStatus(DC)
```

Description

`status = checkStatus(DC)` returns the current status of the data capture object DC in nonblocking capture mode.

Note The `checkStatus` function is not supported in blocking capture mode.

Examples

Check Current Status of Data Capture Object

Before you use this example, you must have previously generated the customized data capture object using the **FPGA Data Capture Component Generator** tool. You must also have integrated the generated IP code into your project and deployed it to the FPGA. The data capture object communicates with the FPGA over a JTAG cable. Make sure that the required cable is connected between the board and the host computer.

Create a data capture object, DC, that captures data from a design running on an FPGA. `datacapture` is the generated IP name you specified in the **FPGA Data Capture Component Generator** tool.

```
DC = datacapture
```

```
DC =
```

```
    datacapture with properties:
```

```
        Connection: 'JTAG'
    IsConditionalCapture: 0
        TriggerPosition: 0
    NumCaptureWindows: 1
        NumTriggerStages: 1
            Timeout: 10
    EnableCaptureCtrl: 0
        CaptureMode: 'blocking'
        JTAGCableName: 'auto'
        JTAGCableType: 'auto'
    JTAGChainPosition: 0
        IRLengthBefore: 0
        IRLengthAfter: 0
        TckFrequency: 15
    MaxNumTriggerStages: 1
```

Change the capture mode to nonblocking mode.

```
DC.CaptureMode = 'nonblocking';
```

Check the current status of the data capture object.

```
status = checkStatus(DC)

status =

    struct with fields:

        CapturedWindows: 0
        RunStatus: 'Not started'
        TriggerStage: 0
```

Use the `step` function to capture data. The data is captured immediately from the FPGA.

```
dataOut = step(DC);
```

Check the current status of the data capture object.

```
status = checkStatus(DC)

status =

    struct with fields:

        CapturedWindows: 1
        RunStatus: 'Successfully captured data from FPGA'
        TriggerStage: 0
```

Input Arguments

DC — Customized data capture object

`hdlverifier.FPGADataReader` System object

Customized data capture object, specified as an `hdlverifier.FPGADataReader` System object.

Output Arguments

status — Current status of data capture object

structure

Current status of the data capture object, returned as a structure containing fields for these signals.

- `CapturedWindows` — This signal indicates the total number of windows captured so far.
- `RunStatus` — This signal indicates the current running status of the data capture object using one of these options.
 - `'Not started'` — Data capture not started.
 - `'Waiting for trigger'` — Data capture object is waiting for a trigger event to start data capture.
 - `'Evaluating capture condition'` — Data capture object is evaluating the capture condition.

- 'Successfully captured data from FPGA' — Data captured successfully from the FPGA.
- 'Stopped' — Data capture has stopped.
- TriggerStage — This signal indicates the trigger stage under evaluation for the run status 'Waiting for trigger'. For the run statuses 'Evaluating capture condition', 'Successfully captured data from FPGA', and 'Stopped', this signal indicates the updated value of the trigger stage.

Version History

Introduced in R2022a

See Also

Objects

hdlverifier.FPGADataReader

Tools

FPGA Data Capture Component Generator | FPGA Data Capture

collectData

Collect captured data from FPGA to host in nonblocking mode

Syntax

```
capturedData = collectData(DC)
```

Description

`capturedData = collectData(DC)` returns the captured data from an FPGA to the host machine in nonblocking capture mode. DC is a customized data capture object.

Note The `collectData` function is not supported in blocking capture mode.

Examples

Collect Captured Data from FPGA

Before you use this example, you must have previously generated the customized data capture object, using the **FPGA Data Capture Component Generator** tool. You must also have integrated the generated IP code into your project and deployed it to the FPGA. The data capture object communicates with the FPGA over a JTAG cable. Make sure that the required cable is connected between the board and the host computer.

This example uses a generated object, `datacapture`, that defines two signals for data capture. Signal A is 16 bits and signal B is 8 bits. Both signals are also available for use in trigger conditions. The sample depth is 1024 samples.

Create a data capture object, DC, that captures data from a design running on an FPGA. `datacapture` is the generated IP name you specified in the **FPGA Data Capture Component Generator** tool.

```
DC = datacapture
```

```
DC =
```

```
    datacapture with properties:
```

```
        Connection: 'JTAG'
    IsConditionalCapture: 0
        TriggerPosition: 0
    NumCaptureWindows: 1
    NumTriggerStages: 1
        Timeout: 10
    EnableCaptureCtrl: 0
        CaptureMode: 'blocking'
        JTAGCableName: 'auto'
        JTAGCableType: 'auto'
    JTAGChainPosition: 0
```

```
        IRLengthBefore: 0
        IRLengthAfter: 0
        TckFrequency: 15
    MaxNumTriggerStages: 1
```

Change the capture mode to nonblocking mode.

```
DC.CaptureMode = 'nonblocking';
```

Check the current status of the data capture object.

```
status = checkStatus(DC)

status =

    struct with fields:

        CapturedWindows: 0
        RunStatus: 'Not started'
        TriggerStage: 0
```

Define a trigger condition to capture data when the signal B is equal to 10.

```
setTriggerCondition(DC, 'B', true, 10);
```

Use the `step` function to capture data on the specified trigger event.

```
dataOut = step(DC);
```

Check the current status of the data capture object.

```
status = checkStatus(DC)

status =

    struct with fields:

        CapturedWindows: 1
        RunStatus: 'Successfully captured data from FPGA'
        TriggerStage: 0
```

Collect captured data.

```
capturedData = collectData(DC)

Captured 1 windows of data from FPGA.

capturedData =

    struct with fields:

        Capture_Window: [1024x1 uint32]
        Trigger_Position: [1024x1 logical]
```



```
A: [1024x1 uint16]
B: [1024x1 uint8]
```

Input Arguments

DC — Customized data capture object

hdlverifier.FPGADataReader System object

Customized data capture object, specified as an `hdlverifier.FPGADataReader` System object.

Output Arguments

capturedData — Captured data

structure

Captured data, returned as a structure containing a field for the `Capture_Window` signal, a field for the `Trigger_Position` signal, and a field for each signal obtained by FPGA data capture. The captured signal field is a vector of *Sample depth* values for each signal requested for data capture at generation time. The fields of the structure have these signal names.

- `Capture_Window` — This signal indicates the current capture window.
- `Trigger_Position` — This signal indicates the position of the trigger detection clock cycle within a capture buffer.
- All remaining fields — The signal names you specified at generation time.

Version History

Introduced in R2022a

See Also

Objects

hdlverifier.FPGADataReader

Tools

FPGA Data Capture Component Generator | FPGA Data Capture

stop

Stop FPGA data capture execution based on current status in nonblocking mode

Syntax

```
stop(DC)
```

Description

`stop(DC)` stops the execution of the data capture object `DC` based on the current status in nonblocking capture mode. Stop execution of the specified data capture object when the current status of the object is 'Waiting for trigger'. If you want to cancel the capture attempt (for example, if the trigger condition does not occur), use this function to return control to the object. When you cancel a capture attempt, no data is returned to MATLAB.

Note The `stop` function is not supported in blocking capture mode.

Examples

Stop FPGA Data Capture Execution

Before you use this example, you must have previously generated the customized data capture object using the **FPGA Data Capture Component Generator** tool. You must also have integrated the generated IP code into your project and deployed it to the FPGA. The data capture object communicates with the FPGA over a JTAG cable. Make sure that the required cable is connected between the board and the host computer.

This example uses a generated object, `datacapture`, that defines two signals for data capture. Signal A is 16 bits and signal B is 8 bits. Both signals are also available for use in trigger conditions. The sample depth is 1024 samples.

Create a data capture object, `DC`, that captures data from a design running on an FPGA. `datacapture` is the generated IP name you specified in the **FPGA Data Capture Component Generator** tool.

```
DC = datacapture
```

```
DC =
```

```
datacapture with properties:
```

```
    Connection: 'JTAG'  
IsConditionalCapture: 0  
    TriggerPosition: 0  
    NumCaptureWindows: 1  
    NumTriggerStages: 1  
           Timeout: 10  
    EnableCaptureCtrl: 0
```

```

        CaptureMode: 'blocking'
        JTAGCableName: 'auto'
        JTAGCableType: 'auto'
    JTAGChainPosition: 0
        IRLengthBefore: 0
        IRLengthAfter: 0
        TckFrequency: 15
    MaxNumTriggerStages: 1

```

Change the capture mode to nonblocking mode.

```
DC.CaptureMode = 'nonblocking';
```

Check the current status of the data capture object.

```
status = checkStatus(DC)

status =

    struct with fields:
        CapturedWindows: 0
        RunStatus: 'Not started'
        TriggerStage: 0

```

Define a trigger condition to capture data when the signal B is equal to 255.

```
setTriggerCondition(DC, 'B', true, 255);
```

Use the `step` function to capture data on the specified trigger event.

```
dataOut = step(DC);
```

Check the current status of the data capture object.

```
status = checkStatus(DC)

status =

    struct with fields:
        CapturedWindows: 0
        RunStatus: 'Waiting for trigger'
        TriggerStage: 1

```

Stop data capture.

```
stop(DC);
```

Check the current status of the data capture object.

```
status = checkStatus(DC)

status =

    struct with fields:
        CapturedWindows: 0

```

```
RunStatus: 'Stopped'  
TriggerStage: 1
```

Input Arguments

DC — Customized data capture object

hdlverifier.FPGADataReader System object

Customized data capture object, specified as an hdlverifier.FPGADataReader System object.

Version History

Introduced in R2022a

See Also

Objects

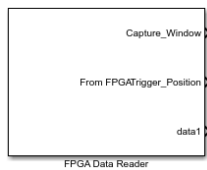
hdlverifier.FPGADataReader

Tools

FPGA Data Capture Component Generator | FPGA Data Capture

FPGA Data Reader

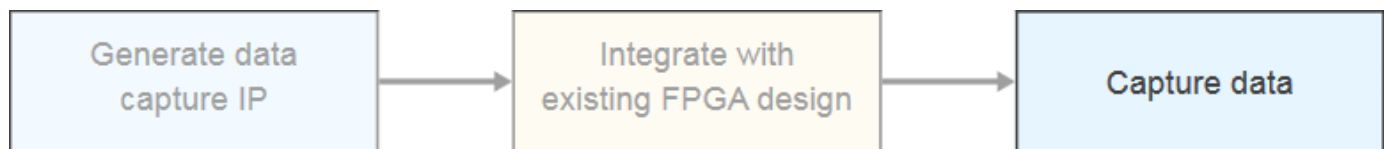
Capture data from live FPGA into Simulink model



Libraries:
Generated

Description

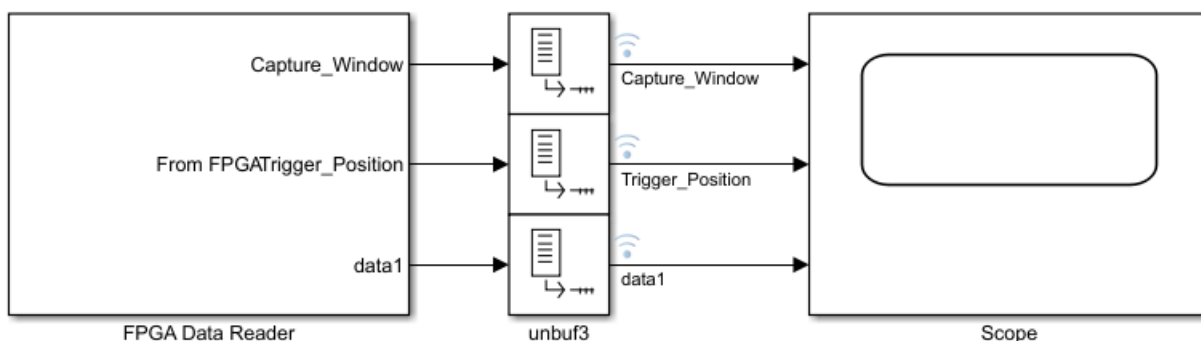
The FPGA Data Reader block communicates with a generated IP core on an FPGA to return captured data into Simulink.



Before you run this block, you must generate the customized data capture components. Integrate the generated HDL IP core into your project and deploy it to the FPGA. The block communicates with the FPGA over a JTAG or Ethernet cable. Make sure that the required cable is connected between the board and the host computer.

For a workflow overview, see “Data Capture Workflow” on page 6-2.

By default, the **FPGA Data Capture Component Generator** tool generates a data capture model that contains this block and a scope. If you have a DSP System Toolbox license, the captured data is streamed to the **Logic Analyzer** tool. Otherwise, the Scope block shows the captured data. You can add other blocks to the model for analysis, verification, and display.



Ports

The output ports of the FPGA Data Reader block correspond to the signals you requested to capture in **FPGA Data Capture Component Generator**. Set the data types for these ports in the **Signal and Trigger Editor**, opened from the block parameters.

Output

Capture_Window — Current capture window
scalar

This output port indicates the current capture window. The value of this output port is an integer from 1 to the value of the **Sample depth** parameter.

Trigger_Position — Position of trigger detection clock cycle within capture buffer
Boolean scalar

This output port indicates the position of the trigger detection clock cycle within a capture buffer.

Parameters

Sample time — Rate of output signals
double

The block returns one frame of data per time step, where the frame is the entire capture buffer for each signal. Each frame contains **Sample depth** values, as specified at generation time. The default sample time provides for unbuffering each frame into single samples, which results in a sample time of 1.

Trigger

Sample depth — Number of samples captured for each signal
integer

This parameter is read-only. It reflects the value you specified at generation time.

Number of capture windows — Number of data capture recurrences
1 (default) | integer power of two

Specify the number of recurrences to capture. This value must be a power of two, up to **Sample depth**. A *window depth* is defined as **Sample depth** / **Number of capture windows**. Consider the **Number of capture windows** when setting the **Sample depth**, to allow for sufficient buffering.

Number of trigger stages — Number of trigger stages for providing trigger conditions
 M (default) | integer from 1 to M

Specify the number of trigger stages. This value must be an integer from 1 to M , where M is set by the **Max trigger stages** parameter of the **FPGA Data Capture Component Generator** tool. When you specify the **Max trigger stages** parameter, consider the maximum number of trigger stages in which you plan to configure the trigger conditions to capture data.

Trigger position — Position of trigger detection cycle within capture buffer
0 (default) | integer up to *window depth*-1

By default, the clock cycle when the trigger is detected is the first sample of the capture buffer. You can change the relative position of the trigger detection cycle within the capture buffer. A nondefault trigger position means that some samples are captured before the trigger occurs. You can set this parameter to any number between 0 and *window depth*-1, inclusive. When the trigger position is equal to the *window depth*-1, the last sample corresponds to the cycle when the trigger occurs. If **Number of capture windows** is greater than one, the same trigger position applies to all windows. See “Triggers” on page 6-7.

Signal — Trigger component signal name
character vector

This parameter is read-only. The signal names you specified at generation time are listed in the drop-down menu at the bottom. Click the + button to add a signal to the trigger condition.

Operator — Operator to compare signals within trigger condition
== | != | < | > | <= | >=

To compare signals, select one of these operators: ==, !=, <, >, <=, or >=. To compare signals containing X or x (don't-care value), specify either == or != operator.

Value — Value to compare signal to as part of overall trigger condition
decimal | binary | hexadecimal | Low | High | Rising edge | Falling edge | Both edges

The trigger condition can be composed of value comparisons of one or more signals. This parameter specifies the value to match for each signal.

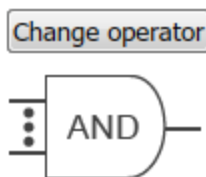
For a multi-bit signal, specify a decimal, binary, or a hexadecimal value within the range of the data type associated with the signal. While providing hexadecimal or binary values, you can provide values with a combination of X or x (don't care value) to enable bit masking. While comparing the values, the trigger condition discards place values with X or x and provides the output.

To separate a group of bits for better readability, you can use _ between bits. For example, you can represent a 32-bit binary value as 0b1010_XXXX_1011_XXXX_1110_XXXX_1111XXXX and a 32-bit hexadecimal value as 0xAB_CDEFX.

For boolean signals, select a level or edge condition. See “Triggers” on page 6-7.

Trigger combination operator — Logical operator to combine comparisons of individual signals into overall trigger condition
AND (default) | OR

This parameter is indicated by the logic gate icon. Click the **Change operator** button to toggle between AND and OR.



The trigger condition can be composed of value comparisons of one or more signals. Combine these value comparisons with only one type of logical operator. Suppose three signals, A, B, and C, make up the trigger condition. The options are:

A == 10 AND B == 'Falling edge' AND C == 0

or

A == 10 OR B == 'Falling edge' OR C == 0

You cannot mix and match the combination operators. See “Triggers” on page 6-7.

Trigger time out — Maximum number of data capture IP core clock cycles within which trigger condition must occur in a trigger stage

1 (default) | integer from 1 to 65,536

Within this many data capture IP core clock cycles, the trigger condition must occur in a trigger stage in which you are enabling this parameter. You can specify any integer value from 1 to 65,536 according to your requirements. Select this parameter to enable trigger time out in a trigger stage. A trigger time out is not allowed in **Trigger Stage 1**.

Time out — Number of seconds to wait before aborting data capture, if the trigger condition is not met

10 (default) | positive integer

If a trigger condition is enabled, but the HDL IP core does not detect the condition, the data capture request times out after this many seconds. No data is returned to Simulink.

Capture Condition

Enable capture condition logic — Option to enable capture condition logic

off (default) | on

Select this parameter to enable capture condition logic in the data capture IP core. Enable capture condition logic to use a capture condition to control which data to capture from the FPGA. The data capture IP core evaluates the capture condition at each clock cycle and captures only the data that satisfies the capture condition. For more information on capture conditions, see “Capture Conditions” on page 6-13.

Dependencies

To enable this parameter, in the **FPGA Data Capture Component Generator** tool, select **Include capture condition logic**.

Signal — Capture component signal name

character vector

This parameter is read-only. The signal names you specified as triggers at generation time are listed in the drop-down menu at the bottom. Click the + button to add a signal to the capture condition.

Dependencies

To enable this parameter, select **Enable capture condition logic**.

Operator — Operator to compare signals within capture condition

== | != | < | > | <= | >=

To compare signals, select one of these operators: ==, !=, <, >, <=, or >=. To compare signals containing X or x (don't-care value), specify either == or != operator.

Dependencies

To enable this parameter, select **Enable capture condition logic**.

Value — Value to compare signal to as part of overall capture condition

decimal | binary | hexadecimal | Low | High | Rising edge | Falling edge | Both edges

The capture condition can be composed of value comparisons of one or more signals. This parameter specifies the value to match for each signal.

For a multi-bit signal, specify a decimal, binary, or a hexadecimal value within the range of the data type associated with the signal. While providing hexadecimal or binary values, you can provide values with a combination of X or x (don't care value) to enable bit masking. While comparing the values, the capture condition discards place values with X or x and provides the output.

To separate a group of bits for better readability, you can use _ between bits. For example, you can represent a 32-bit binary value as 0b1010_XXXX_1011_XXXX_1110_XXXX_1111XXXX and a 32-bit hexadecimal value as 0xAB_CDEFX.

For boolean signals, select a level or edge condition. See “Capture Conditions” on page 6-13.

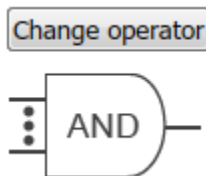
Dependencies

To enable this parameter, select **Enable capture condition logic**.

Capture condition combination operator — Logical operator to combine comparisons of individual signals into overall capture condition

AND (default) | OR

This parameter is indicated by the logic gate icon. Click the **Change operator** button to toggle between AND and OR.



The capture condition can be composed of value comparisons of one or more signals. Combine these value comparisons with only one type of logical operator. You cannot mix and match the combination operators. See “Capture Conditions” on page 6-13.

Dependencies

To enable this parameter, select **Enable capture condition logic**.

Data Types

Signal Name — Name of output port

character vector

This parameter is read-only. It reflects the name of the **Capture_Window** output port, the name of the **Trigger_Position** output port, and the signal names you specify at generation time.

Bit Width — Number of bits in signal
positive integer

This parameter is read-only. It reflects the value you specified at generation time.

Data Type — Data type for captured data
built-in type | numeric type

The **Data Type** menu provides data type suggestions that match the bit width of the captured signal. This size is the width you specified for the port on the generated IP. You can type in this field to specify a custom data type. If the signal is 8, 16, or 32 bits, the default is `uint`. If the signal has one bit, the default is `boolean`. If the signal is a different width, the default is `numeric type(0, bitWidth, 0)`.

The block supports these data types, depending on the signal bit width: `boolean`, `uint8`, `int8`, `uint16`, `int16`, `half`, `uint32`, `int32`, `single`, `uint64`, `int64`, `double`, and `numeric type`.

If your development board has multiple FPGAs or multiple JTAG connections, the data capture software cannot detect the location of your FPGA in the JTAG chain. Specify these advanced parameters to locate the FPGA that contains the data capture IP core.

Advanced Board Setup

JTAG cable name — Name of JTAG cable used for data capture
`auto` (default) | character vector

Name of the JTAG cable used for data capture, specified as a character vector. Use this parameter when the board is connected to two JTAG cables of the same type.

Dependencies

To enable this parameter, in the **FPGA Data Capture Component Generator** tool, set the **Connection type** parameter to `JTAG`.

JTAG cable type — Type of JTAG cable used for communication with FPGA board
`auto` (default) | `FTDI`

Specify this parameter if more than one JTAG cable is connected to the host computer. When not specified, the FPGA Data Reader block will auto-detect the JTAG cable type, in the following order:

- The FPGA Data Reader block first searches for a Digilent cable.
- If it does not find a Digilent JTAG cable, it searches for an FTDI cable.
- If it finds two cables of the same type, the object returns an error. Set this parameter to resolve it.
- If it finds two cables of different types, it will prioritize the Digilent cable. To use an FTDI cable, set this parameter to `FTDI`.

Dependencies

To enable this parameter, in the **FPGA Data Capture Component Generator** tool, set the **Connection type** parameter to `JTAG`.

JTAG chain position — Position of FPGA in JTAG scan chain
`0` (default) | positive integer

Position of the FPGA in the JTAG scan chain, specified as a positive integer.

Dependencies

To enable this parameter, in the **FPGA Data Capture Component Generator** tool, set the **Connection type** parameter to JTAG.

Instruction registers before FPGA — Instruction register lengths before FPGA

0 (default) | nonnegative integer

Number of instruction register lengths before the FPGA, specified as a nonnegative integer.

Dependencies

To enable this parameter, in the **FPGA Data Capture Component Generator** tool, set the **Connection type** parameter to JTAG.

Instruction registers after FPGA — Instruction register lengths after FPGA

0 (default) | nonnegative integer

Number of instruction register lengths after the FPGA, specified as a nonnegative integer.

Dependencies

To enable this parameter, in the **FPGA Data Capture Component Generator** tool, set the **Connection type** parameter to JTAG.

JTAG clock frequency in MHz — JTAG clock frequency

15 (default) | integer

Specify the JTAG clock frequency, in MHz. For Xilinx FPGAs, the JTAG clock frequency is 33 or 66 MHz. The JTAG frequency depends on the type of cable and the maximum clock frequency supported by the FPGA board.

Dependencies

To enable this parameter, in the **FPGA Data Capture Component Generator** tool, set the **Connection type** parameter to JTAG.

Device IP address — IP address of Ethernet port on FPGA board

192.168.0.2 (default) | dotted-quad value

Specify the internet protocol (IP) address of the Ethernet port on the FPGA board as a dotted-quad value. The device IP address must be a set of four numbers consisting of integers in the range from 0 to 255 that are separated by three dots.

Dependencies

To enable this parameter, in the **FPGA Data Capture Component Generator** tool, set the **Connection type** parameter to Ethernet.

Device Port address — UDP port number of FPGA board

50101 (default) | integer from 255 to 65,535

Specify the user datagram protocol (UDP) port number of the FPGA board as an integer from 255 to 65,535.

Dependencies

To enable this parameter, in the **FPGA Data Capture Component Generator** tool, set the **Connection type** parameter to Ethernet.

Version History

Introduced in R2017a

See Also

Apps

FPGA Data Capture

Objects

`hdlverifier.FPGADataReader`

Topics

“Data Capture Workflow” on page 6-2

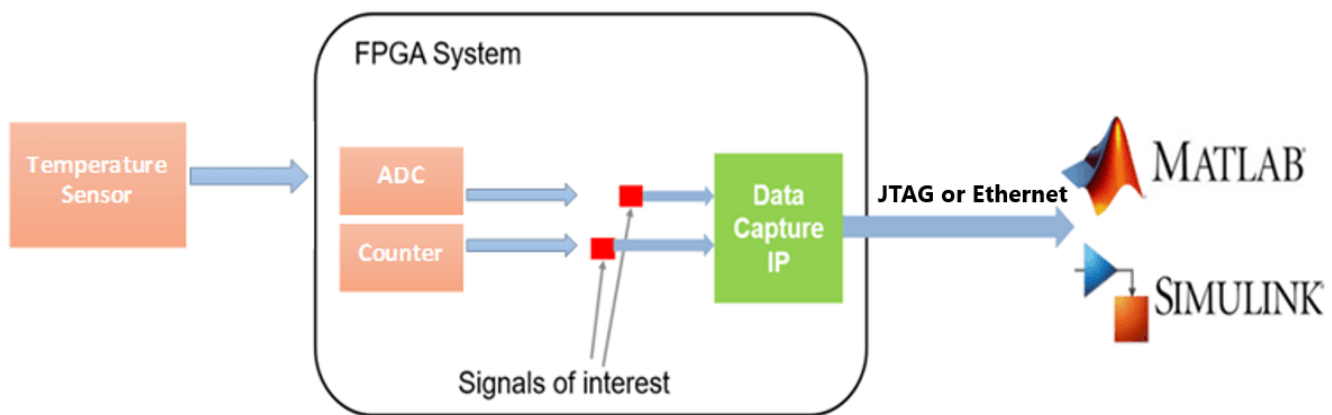
“Triggers” on page 6-7

“Capture Conditions” on page 6-13

HDL Verifier Support Package for Xilinx FPGA Boards Examples

Capture Temperature Sensor Data from Xilinx FPGA Board Using FPGA Data Capture

This example shows how to use FPGA data capture with existing HDL design to capture FPGA internal signals over a JTAG or Ethernet connection. Start with an existing FPGA design that implements Xilinx® XADC IP to read the on-chip temperature sensor data. The XADC IP exposes a dynamic reconfiguration port (DRP) interface for read and write internal registers. This FPGA design contains logic that reads out the temperature sensor register from the XADC IP. To obtain the temperature reading for further analysis, use the FPGA data capture feature to read the raw sensor data into the MATLAB® workspace. Then, MATLAB converts the raw temperature data to Celsius.



Requirements and Prerequisites

- MATLAB
- HDL Verifier™
- HDL Verifier Support Package for Xilinx FPGA Boards
- Fixed-Point Designer™
- Xilinx Vivado® Design Suite, with a supported version listed in “Supported EDA Tools and Hardware” on page 1-6
- For JTAG connection: ZedBoard™ or Xilinx Virtex®-7 VC707 development board
- For Ethernet connection: Xilinx Virtex-7 KC705 development board
- JTAG cable and/or Ethernet cable

Prepare Example Resources

Set up the Xilinx Vivado Design Suite. This example assumes that the Xilinx Vivado executable is located in the file C:\Xilinx\Vivado\2020.2\bin\vivado.bat. If the location of your executable is different, use your path instead.

```
hdlsetuptoolpath('ToolName','Xilinx Vivado','ToolPath', ...
                'C:\Xilinx\Vivado\2020.2\bin\vivado.bat');
```

Set Up FPGA Development Board

1. Confirm that the power switch is off.

2. Connect the AC power cord to the power plug and plug the power supply adapter cable into the FPGA development board.
3. Use the JTAG download cable to connect the FPGA development board with the host computer.
4. Connect the Ethernet connector on the FPGA development board directly to the Ethernet adapter on your host machine using the crossover Ethernet cable.
5. Turn on the power switch on the FPGA board.

Generate FPGA Data Capture Components

Launch the **FPGA Data Capture Component Generator** tool by executing this command in MATLAB.

```
generateFPGADataCaptureIP
```

This example monitors two signals from the existing HDL code for the temperature sensor system: 16 bit **temperature** and 8 bit **counter**. The **temperature** signal is the reading of register 0x00 from the XADC, which stores the converted raw temperature sensor. It has 16 bits, but only the 12-bit most significant bits (MSBs) are the raw temperature sensor reading. The last signal **counter** is an 8 bit free-running counter. To configure the data capture components to operate on these two signals over a JTAG or Ethernet connection, follow these steps.

1. Add one row to the **Ports** table by clicking the **Add** button once.
2. Name the first signal to **temperature** and the second signal to **counter**.
3. Change the bit widths of the two signals to 16 and 8, respectively.
4. Select **FPGA vendor** as Xilinx.
5. Select **Generated IP language** as Verilog.
6. Select **Sample depth** as 1024. This is the number of samples of each signal that the data capture tool returns to MATLAB each time a trigger is detected.
7. Select **Max trigger stages** as 2. This value is the maximum number of trigger stages that you can add during data capture to provide multiple trigger conditions.
8. Select **Connection type** as one of these options.
 - JTAG - To capture data over a JTAG connection
 - Ethernet - To capture data over an Ethernet connection

FPGA Data Capture Component Generation

Description
 Generate an HDL IP for integration into your FPGA design.
 Specify port names and bit-widths for the interface of the generated IP, and specify how many samples the IP captures at a time.

Generate data capture IP → Integrate with existing FPGA design → Capture data

[Read more about the data capture workflow](#)

Ports

Port Name	Bit Width	Use As
temperature	16	Both triqger and data
counter	8	Both triqger and data

Target

Generated IP name: datacapture
 FPGA vendor: Xilinx
 Generated IP language: Verilog
 Connection type: JTAG
 Destination folder: hdlsrc

Capture

Sample depth: 1024
 Max trigger stages: 2
 Include capture condition logic

Generate Cancel Help

9. Ethernet connection only: Set **IP address** to 192.168.0.2 and **Port address** to 50101. Select **Interface type** as GMII. This is the type of Ethernet interface of your target FPGA development board.

This figure shows these tool settings.

FPGA Data Capture Component Generation

Description
Generate an HDL IP for integration into your FPGA design.

Specify port names and bit-widths for the interface of the generated IP, and specify how many samples the IP captures at a time.

Generate data capture IP → Integrate with existing FPGA design → Capture data

[Read more about the data capture workflow](#)

Ports

Port Name	Bit Width	Use As
temperature	16	Both triqger and data
counter	8	Both triqger and data

Target

Generated IP name: datacapture
 FPGA vendor: Xilinx
 Generated IP language: Verilog
 Connection type: Ethernet
 Destination folder: hdlsrc

Capture

Sample depth: 1024
 Max trigger stages: 2
 Include capture condition logic

Ethernet settings

IP address: 192.168.0.2
 Port address: 50101
 Interface type: GMII

Generate Cancel Help

To generate the FPGA data capture component, click **Generate**. A report shows the results of the generation.

Integrate FPGA Data Capture HDL IP

You must include the generated HDL IP core into the example FPGA design. You can copy the module instance code from the generated report. In this example, we connect the generated HDL IP with the temperature sensor output from XADC IP and an 8-bit free-running counter.

For JTAG Connection

If you are using ZedBoard, open the `top.v` file provided with this example. If you are using VC707, open the `top_vc707.v` file provided with this example. Uncomment this code.

```
datacapture u0 (
    .clk(clk),
    .clk_enable(1'b1),
    .ready_to_capture(),
    .temperature(do_out),
    .counter(counter[7:0]));
```

Save the file you modified, compile the modified FPGA design, and create an FPGA programming file.

If you are using ZedBoard, execute this command in MATLAB.

```
system('vivado -mode batch -source data_capture_xadc_zedboard.tcl &')
```

If you are using VC707, execute this command in MATLAB.

```
system('vivado -mode batch -source data_capture_xadc_vc707.tcl &')
```

The above mentioned Tcl scripts that are included in this example perform these steps.

1. Create a new Vivado project.
2. Add example HDL files and the generated FPGA data capture HDL files to the project.
3. Compile the design.
4. Program the FPGA.

Wait until the Vivado process successfully finishes before going to the next step. This process takes approximately 5 to 10 minutes.

For Ethernet Connection

1. Create a Vivado project for KC705 board by executing this command in MATLAB. This command takes about one minute to run. When the execution completes, a Vivado project named `data_capture_xadc_kc705.xpr` appears in your current directory.

```
system('vivado -mode batch -source create_project_kc705.tcl &')
```

2. Open the generated Vivado project in GUI mode by double-clicking the project in a file browser or executing this command in MATLAB.

```
system('vivado data_capture_xadc_kc705.xpr &')
```

3. Navigate to the `hdlsrc` folder by executing this command in the Vivado Tcl console.

```
cd ./hdlsrc
```

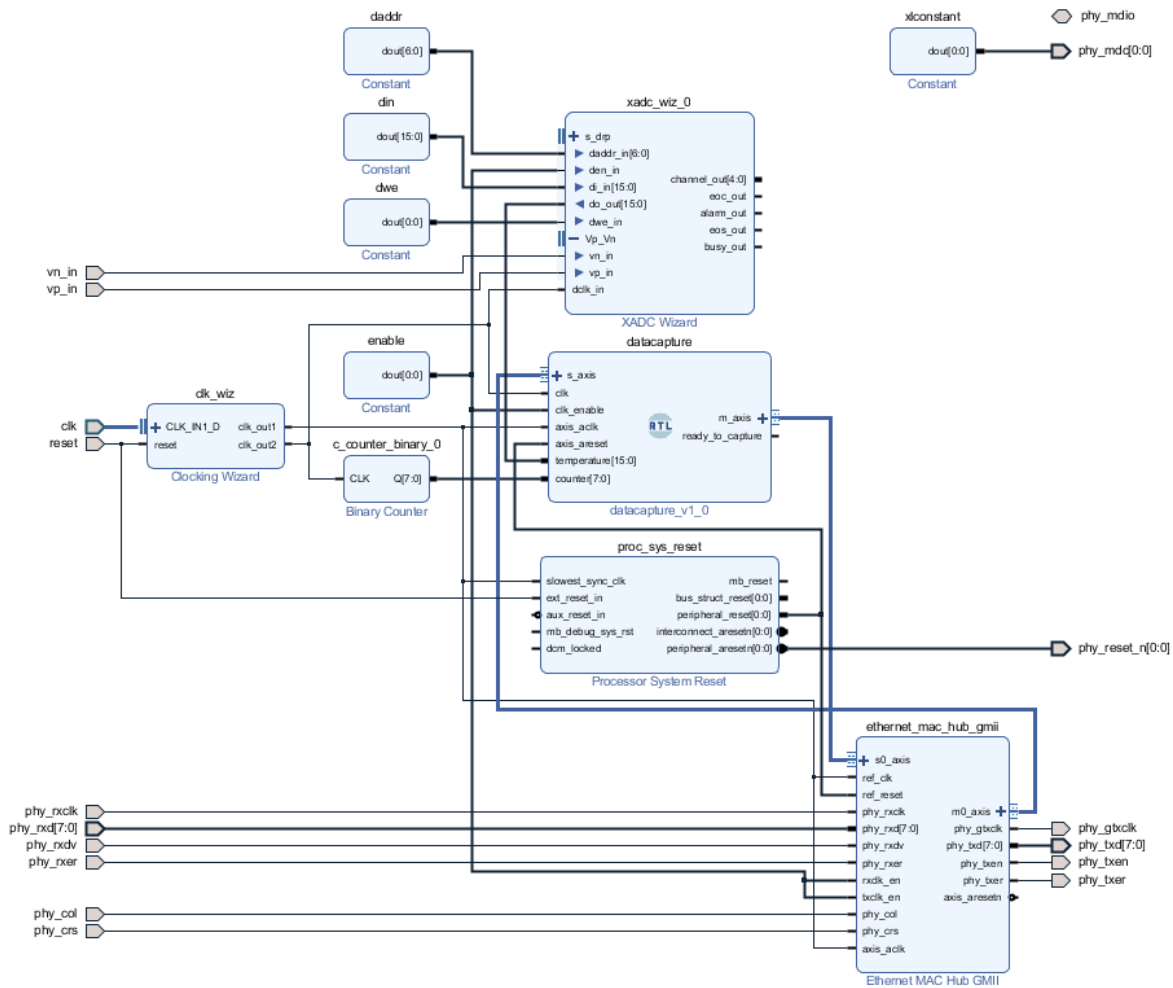
4. Insert the Ethernet MAC Hub IP and FPGA data capture IP into Vivado project by executing this command in the Vivado Tcl console. The **FPGA Data Capture Component Generator** tool generates the `insertEthernet.tcl` script.

```
source ./insertEthernet.tcl
```

5. Complete the design by connecting IPs in the Vivado project, compile the design, and program the FPGA by executing this command in the Vivado Tcl console.

```
source ../ethernet_data_capture_xadc_kc705.tcl
```

6. This figure shows the block diagram in Vivado.



Wait until the Vivado process successfully finishes before going to the next step. This process takes approximately 5 to 10 minutes.

Capture Data

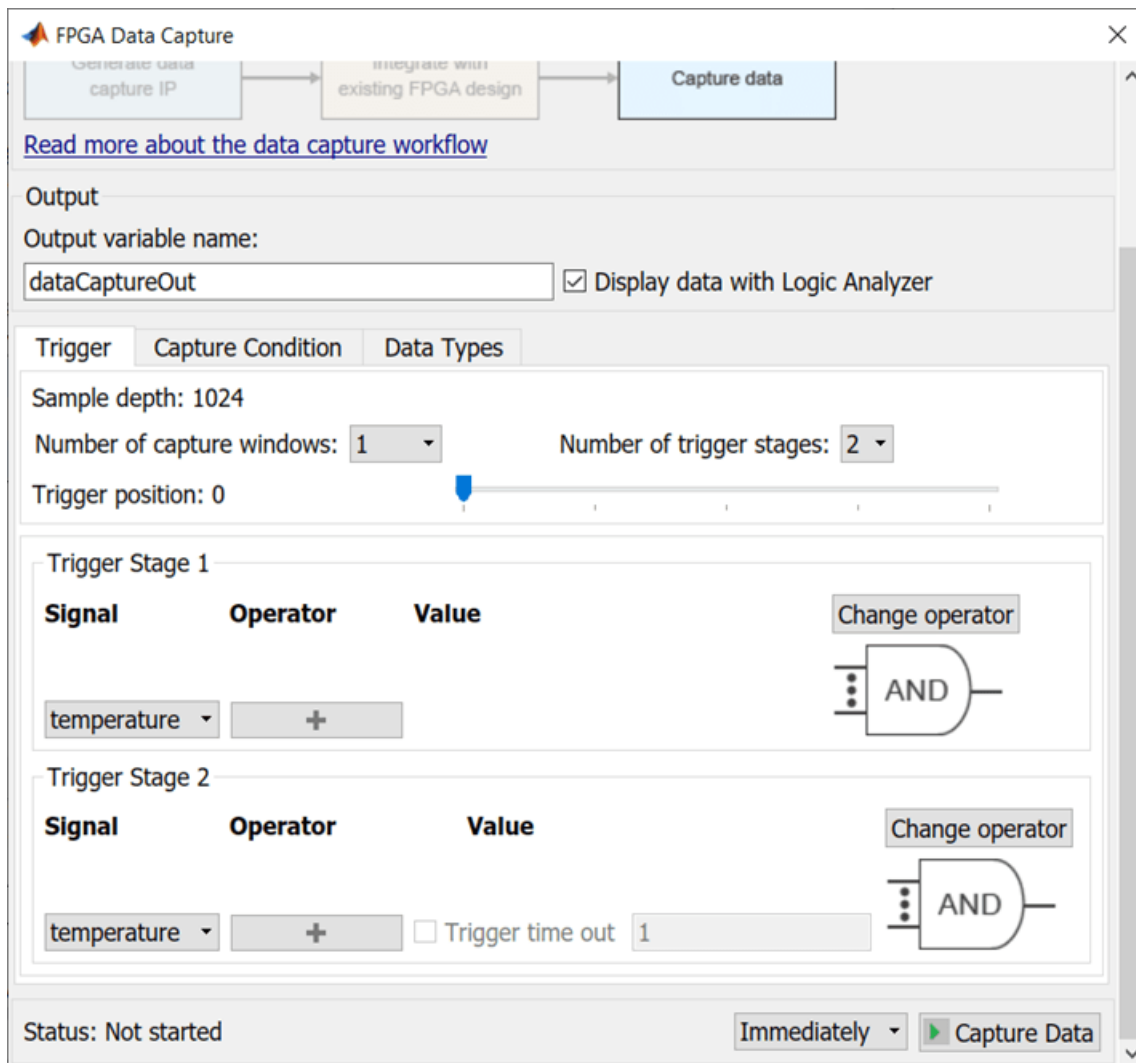
Navigate to the directory where the FPGA data capture component is generated in MATLAB.

```
cd hdlsrc
```

Launch the **FPGA Data Capture** tool. This tool is customized for your data capture signals.

```
launchDataCaptureApp
```

To start data capture click **Capture Data**. This action requests one buffer of captured data from the FPGA. The default trigger setting is to capture immediately, without waiting for a trigger condition.



The captured data is saved into a structure, `dataCaptureOut`, in the MATLAB workspace. If you have the DSP System Toolbox™ software, the captured data is also displayed as signal waveforms in the **Logic Analyzer** tool.

The captured temperature sensor data is in raw format. The sensor data sheet gives the formula for converting raw format data to Celsius units. Calculate and report the average temperature over all of the samples returned.

```
CelsiusTemp = (double(dataCaptureOut.temperature))/(2^4)*503.975/4096 - 273.15;
sprintf('The FPGA Temperature is %fC\n',mean(CelsiusTemp))
```

Narrow Scope of Data Capture Using Triggers

To capture data from the FPGA around a particular event, you can configure trigger conditions in the **FPGA Data Capture** app. For example, capture the temperature data only after a counter reaches a certain value.

Select **Number of trigger stages** as 1. In the **Trigger Stage 1** section, select **Signal** as counter. Enable this trigger signal by clicking the + button. Select the corresponding trigger condition value

(**Value**) to 10. The trigger mode automatically changes to **On trigger**. This change tells the FPGA to wait for the trigger condition before capturing and returning data. This figure shows these tool settings.

FPGA Data Capture

Capture data from a design running on your FPGA board.

Specify data types for the returned data structure, and specify a logical trigger condition that defines when the data is captured.

Generate data capture IP → Integrate with existing FPGA design → Capture data

[Read more about the data capture workflow](#)

Output

Output variable name:
dataCaptureOut Display data with Logic Analyzer

Trigger Capture Condition Data Types

Sample depth: 1024
Number of capture windows: 1 Number of trigger stages: 1
Trigger position: 0

Trigger Stage 1

Signal	Operator	Value
counter	==	10
temperature	+	

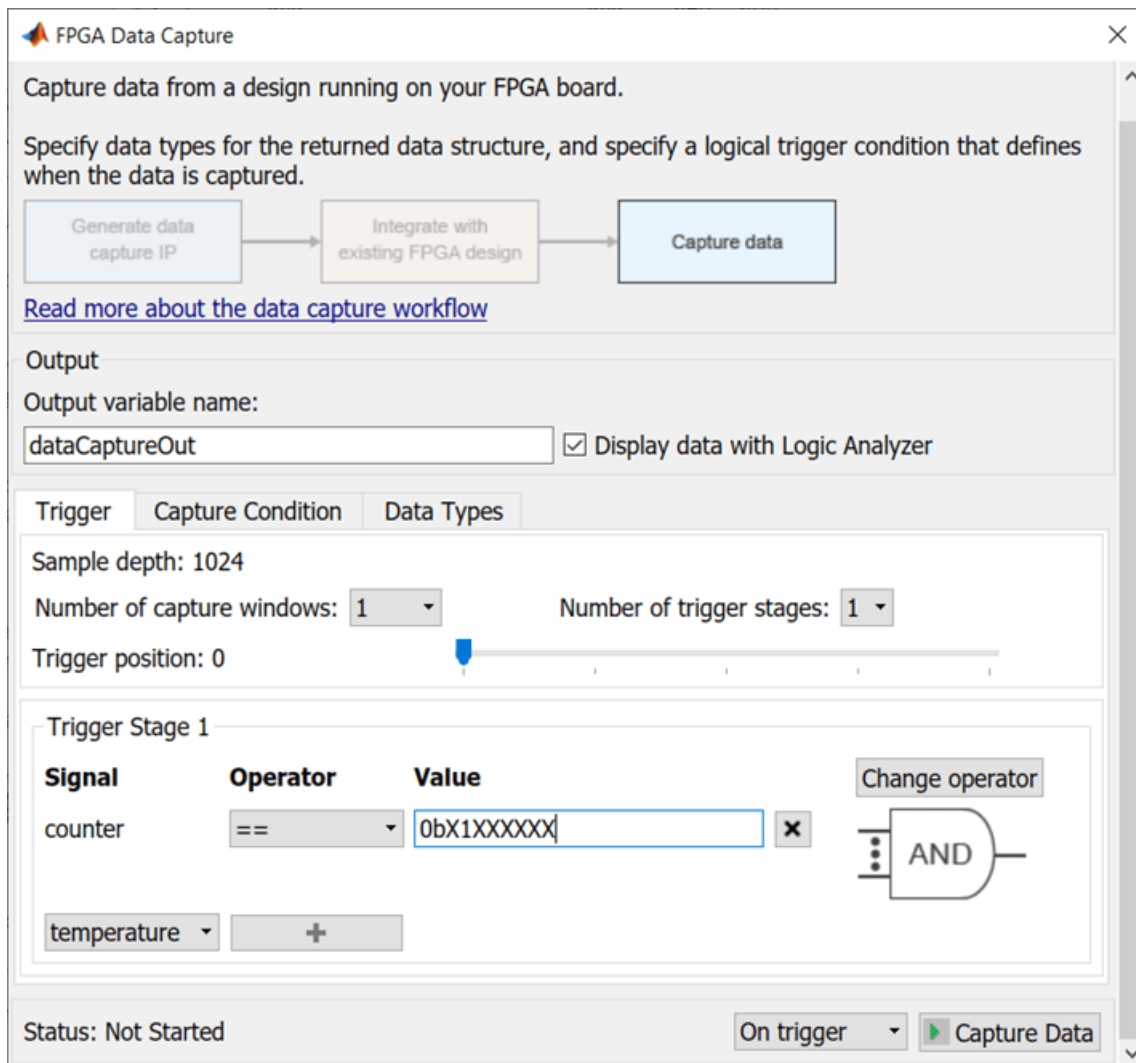
Change operator AND

Status: Not Started On trigger Capture Data

Click **Capture Data** again. This time, the data capture IP returns 1024 samples, captured when it detects that the counter equals 10.

To capture data from the FPGA for specific bits in the trigger value, irrespective of other bits, you can configure the trigger condition with a bit-masked value.

For example, to capture the temperature data only when the seventh bit of the counter is 1, set the trigger condition value (**Value**) to 0bX1XXXXXX as this figure shows.

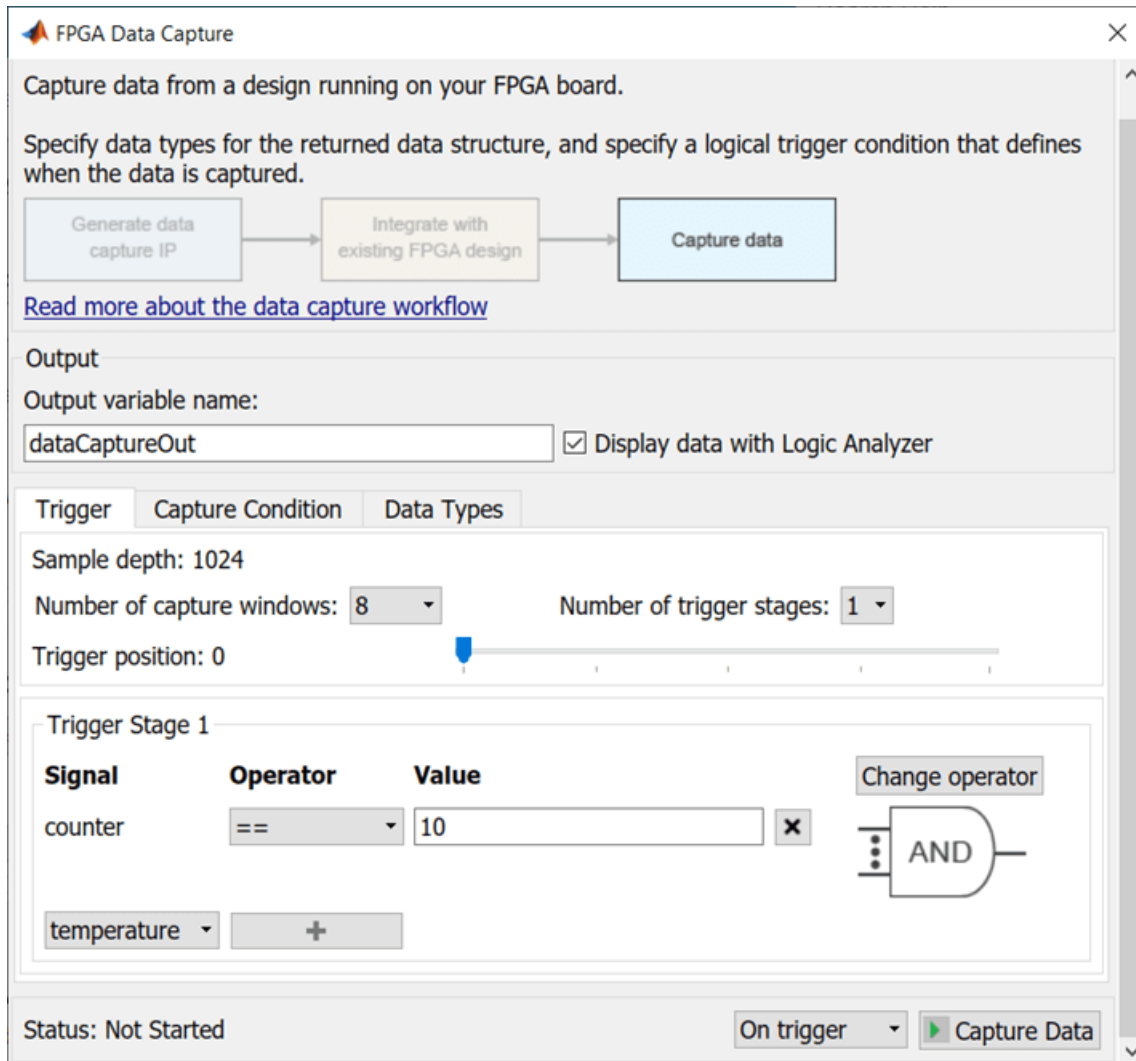


Click **Capture Data** again. The data capture IP triggers to capture the temperature data for counter values in the range [64, 127] and [192, 255].

Capture Multiple Occurrences of Event

To capture a recurring event from the FPGA, configure **Number of capture windows** in the **FPGA Data Capture** tool.

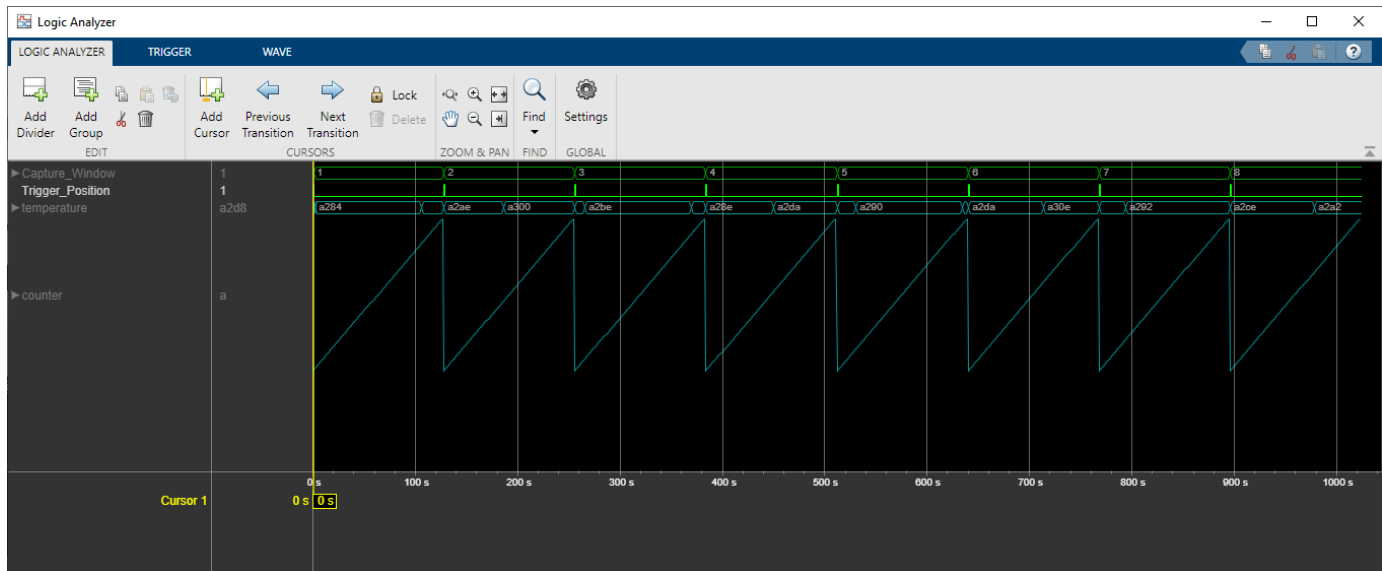
For example, to capture the temperature data at eight different time slots, select **Number of capture windows** as 8. This figure shows the updated tool settings.



Click **Capture Data**. The data capture IP returns eight windows of 128 samples each, which amounts to a total sample depth of 1024.

Window depth = Sample depth/Number of capture windows;

The **Logic Analyzer** tool shows this result as eight occurrences of the trigger, with the temperature logged for 128 samples each.



The signals Capture Window and Trigger Position indicate the corresponding window number and trigger position, respectively.

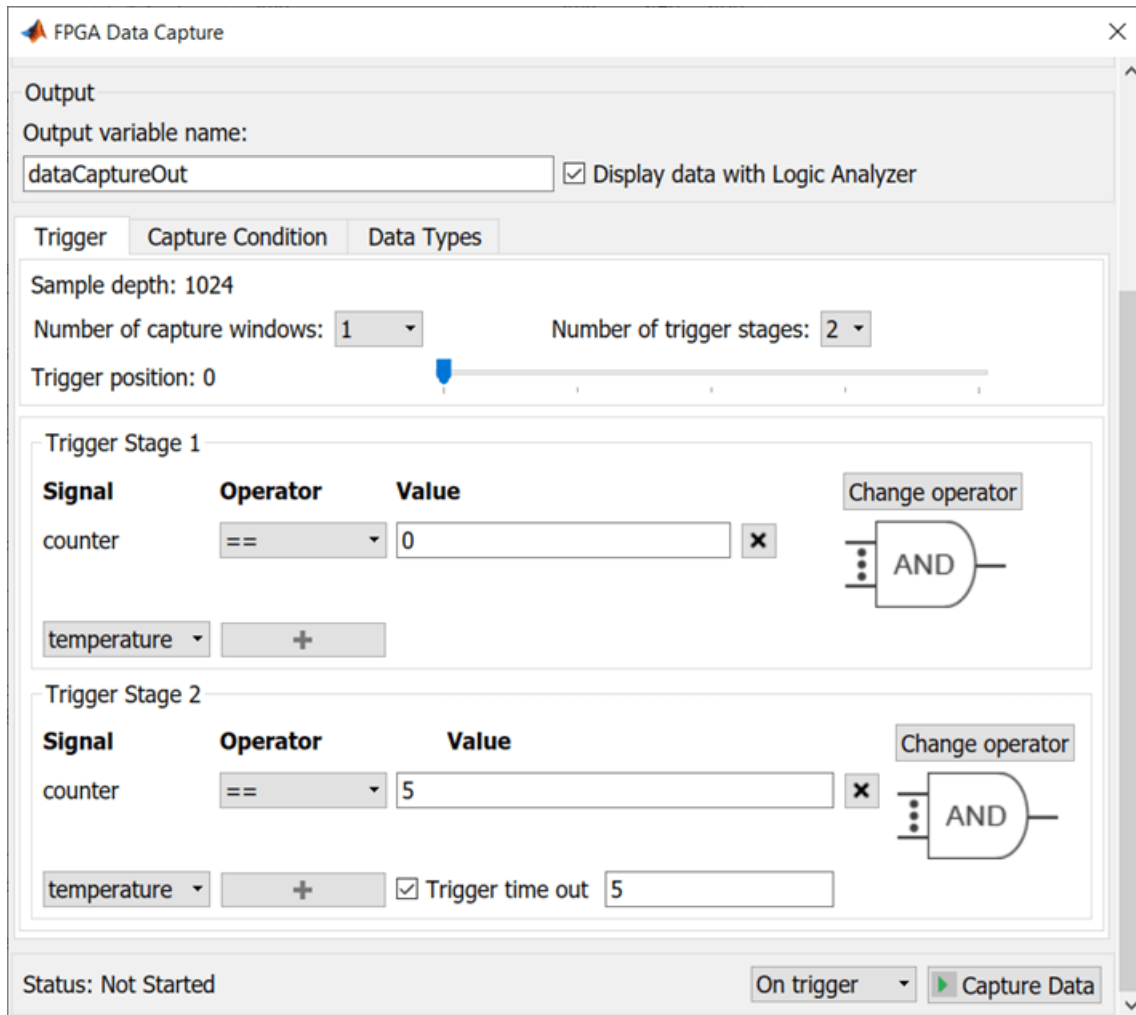
Capture Data in Multiple Trigger Stages

This scenario explains how to capture data by providing a sequence of trigger conditions in multiple trigger stages. For this action, you must select **Number of trigger stages** as a value greater than 1 in the **FPGA Data Capture** tool.

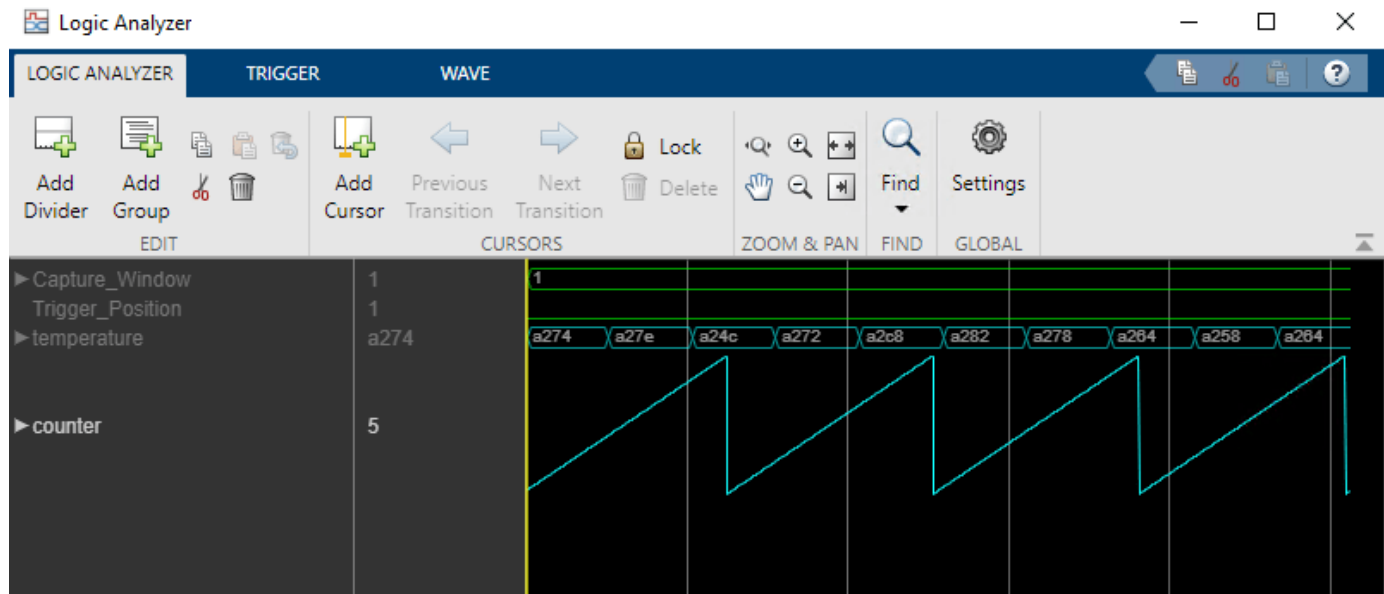
For example, to capture an temperature data when the counter value reaches from 0 to 5 in 5 clock cycles:

1. Select **Number of trigger stages** as 2.
2. In the **Trigger Stage 1** section, select **Signal** as counter. Enable this trigger signal by clicking the + button. Select the corresponding trigger condition value (**Value**) as 0.
3. In **Trigger Stage 2** section, select **Signal** as counter. Enable this trigger signal by clicking the + button. Select the corresponding trigger condition value (**Value**) as 5. Select **Trigger time out** and set it to 5.

This figure shows the updated tool settings.



Click **Capture Data**. The data capture IP captures 1024 samples when it detects the trigger condition in trigger stage 2 within 5 clock cycles, preceded by the trigger condition detected in trigger stage 1.



See Also

FPGA Data Capture Component Generator | FPGA Data Capture

More About

- “Data Capture Workflow” on page 6-2
- “Triggers” on page 6-7
- “Troubleshooting”

Access FPGA Memory Using JTAG-Based AXI Manager

Use JTAG-based AXI manager to access the memories connected to the FPGA. In the FPGA, there is a Xilinx® DDR memory controller and BRAM controller exist for accessing the DDR memories and the BRAM, respectively. These memory controllers provide an AXI4 subordinate interface for write and read operations by other components in the FPGA. The JTAG-based AXI manager feature provides an AXI manager component that you can use to access any AXI subordinate IPs in the FPGA. This example demonstrates how to integrate AXI Manager IP into a Xilinx Vivado® project and how to write and read data from the DDR memory and the BRAM using MATLAB®. This example simulates the design using the Vivado simulator and then programs the FPGA and performs write and read operations from the MATLAB console.

Requirements

- Xilinx Vivado Design Suite with supported version listed in “Supported EDA Tools and Hardware” on page 1-6
- Artix®-7 35T Arty FPGA Evaluation Kit
- HDL Verifier™ Support Package for Xilinx FPGA Boards
- JTAG cable

Setup

1. Set up the FPGA board. Connect the Arty board to the host computer via USB-JTAG cable.
2. Prepare the example in MATLAB. Set up the Xilinx Vivado tool path. Use your own Xilinx Vivado installation path when executing the command.

```
hdlsetuptoolpath('ToolName','Xilinx Vivado','ToolPath', ...
                'C:\Xilinx\Vivado\2020.2\bin\vivado.bat');
```

Create a Vivado project for this example. This project contains the IP Integrator block diagram and constraint file for this example.

```
system('vivado -mode batch -source jtagAXIMcreateproject.tcl')
```

This command takes about one minute to run. When the execution completes, a Vivado project named `arty.xpr` appears in your current directory.

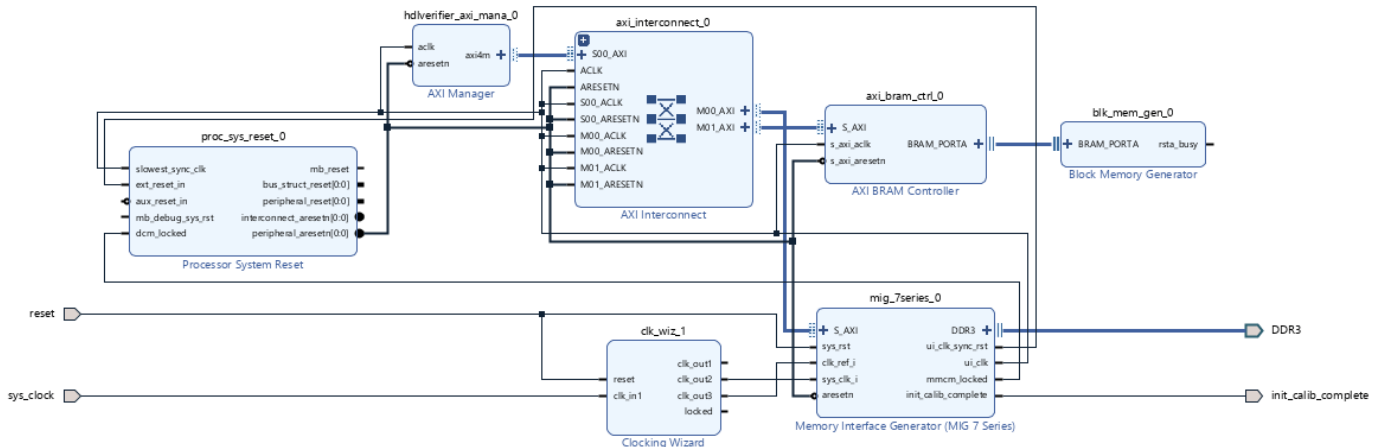
3. Configure the Vivado project with a Vivado IP. To use the AXI Manager IP inside the Vivado IP Integrator, add the folder that contains the IP to the Vivado IP repository path setting for the Vivado project. Add the path to the project by executing this command in MATLAB.

```
setupAXIManagerForVivado arty.xpr
```

Open the generated Vivado project in GUI mode by double-clicking the project in a file browser or executing this command in MATLAB.

```
system('vivado arty.xpr &')
```

4. Add AXI Manager IP to the FPGA design. In the Vivado GUI, open the block diagram design file `jtagAXIMdesign_1.bd`. You can find the design file in the source file subwindow.



Set the address of `mig_7series_0` (DDR controller) to `0x0000_0000` and `axi_bram_ctrl_0` (BRAM controller) to `0xc000_0000` in the address editor as shown in this figure.

Name	Interface	Slave Segment	Master Base Address	Range	Master High Address
Network 0					
/hdlverifier_axi_mana_0					
/hdlverifier_axi_mana_0/axi4m (32 address bits : 4G)					
/axi_bram_ctrl_0/S_AXI	S_AXI	Mem0	0xc000_0000	8K	0xc000_1FFF
/mig_7series_0/memmap	S_AXI	memaddr	0x0000_0000	256M	0x0FFF_FFFF

Alternatively, you can complete these steps by executing Tcl commands in Vivado.

```
source ./jtagAXIMmodifydesign.tcl
```

Write and Read Operations in Simulation

Click **Run Simulation** on the Vivado window to launch the simulation.

When simulation starts, the first few minutes are required to simulate the calibration process of the DDR3.

After the calibration, these lines in the `jtagAXIMtestbench.sv` file writes and reads 256 consecutive words (32-bit) of data from the address 0 using Increment mode.

```
wdata = new[256];
for(integer i =0;i<256;i=i+1)
wdata[i] = i+1;
wrapper.jtagAXIMdesign_1_i.hdlverifier_axi_mana_0.inst.writememory ...
(0,wdata,hdlverifier::HDLV_AXI_BURST_TYPE_INCR);
wrapper.jtagAXIMdesign_1_i.hdlverifier_axi_mana_0.inst.readmemory ...
(0,256,hdlverifier::HDLV_AXI_BURST_TYPE_INCR,rdata);
```

These commands write and read 256 consecutive word of data from the address `0xc000_0000` using Increment mode.

```
wdata = new[256];
for(integer i =0;i<256;i=i+1)
wdata[i] = i+1;
wrapper.jtagAXIMdesign_1_i.hdlverifier_axi_man_0.inst.writememory ...
(3221225472,wdata,hdlverifier::HDLV_AXI_BURST_TYPE_INCR);
wrapper.jtagAXIMdesign_1_i.hdlverifier_axi_man_0.inst.readmemory ...
(3221225472,256,hdlverifier::HDLV_AXI_BURST_TYPE_INCR,rdata);
```

Similarly, these lines show how to write a single word of value 100 into memory at address 0 and read it back.

```
wdata = new[1];
wdata[0] = 100;
wrapper.jtagAXIMdesign_1_i.hdlverifier_axi_man_0.inst.writememory ...
(0,wdata,hdlverifier::HDLV_AXI_BURST_TYPE_FIXED);
wrapper.jtagAXIMdesign_1_i.hdlverifier_axi_man_0.inst.readmemory ...
(0,1,hdlverifier::HDLV_AXI_BURST_TYPE_FIXED,rdata);
```

Generate FPGA Bitstream and Program FPGA

Click **Generate Bitstream** on the Vivado window to generate the FPGA programming file. Vivado might prompt you to save the project before proceeding to the next step. Generating the bitstream file takes about 5 to 10 minutes for Vivado to generate the bitstream file.

After Vivado generates the bitstream, program the FPGA by executing this command in MATLAB.

```
filProgramFPGA('Xilinx Vivado','arty.runs\impl_1\design_1.bit',1)
```

Write and Read Operations to FPGA

After programming the FPGA, you can write and read from the AXI subordinates connected to the AXI Manager IP. This example writes data to the DDR memory connected to the FPGA and the BRAM and then retrieves data into MATLAB.

Create the AXI manager object in MATLAB to write and read from the DDR memory. Write single location and then read data from the same location. In this case, the read data is 100.

```
h = aximanager('Xilinx');
writememory(h,'00000000',100);
readmemory(h,'00000000',1);
```

Use any address in the range from c000_0000 to c000_1FFF to write and read from the BRAM. Write single location and then read data from the same location. In this case, the read data is 1000.

```
writememory(h,'c0000000',1000);
readmemory(h,'c0000000',1);
```

See Also

[aximanager](#) | [writememory](#) | [readmemory](#)

More About

- “Set Up AXI Manager” on page 3-2
- “Troubleshooting”

Perform Large Matrix Multiplication on FPGA External DDR Memory Using Ethernet-Based AXI Manager

This example shows how to use Ethernet-based AXI manager to access external memory connected to an FPGA. This example also shows how to:

- 1 Generate an HDL IP core with an interface.
- 2 Access large matrices from the external DDR3 memory on the Xilinx® Kintex®-7 KC705 board using the Ethernet-based AXI manager interface.
- 3 Perform matrix vector multiplication in the HDL IP core and write the output result back to the DDR3 memory using the Ethernet-based AXI manager interface.

Requirements

To run this example, you must have this software and hardware installed and set up.

- Xilinx Vivado® Design Suite, with a supported version listed in the “Supported EDA Tools and Hardware” on page 1-6
- Xilinx Kintex-7 KC705 Evaluation Kit
- JTAG cable and Ethernet cable for connecting to KC705 FPGA
- HDL Coder™ Support Package for Xilinx FPGA Boards
- HDL Verifier™ Support Package for Xilinx FPGA Boards

Introduction

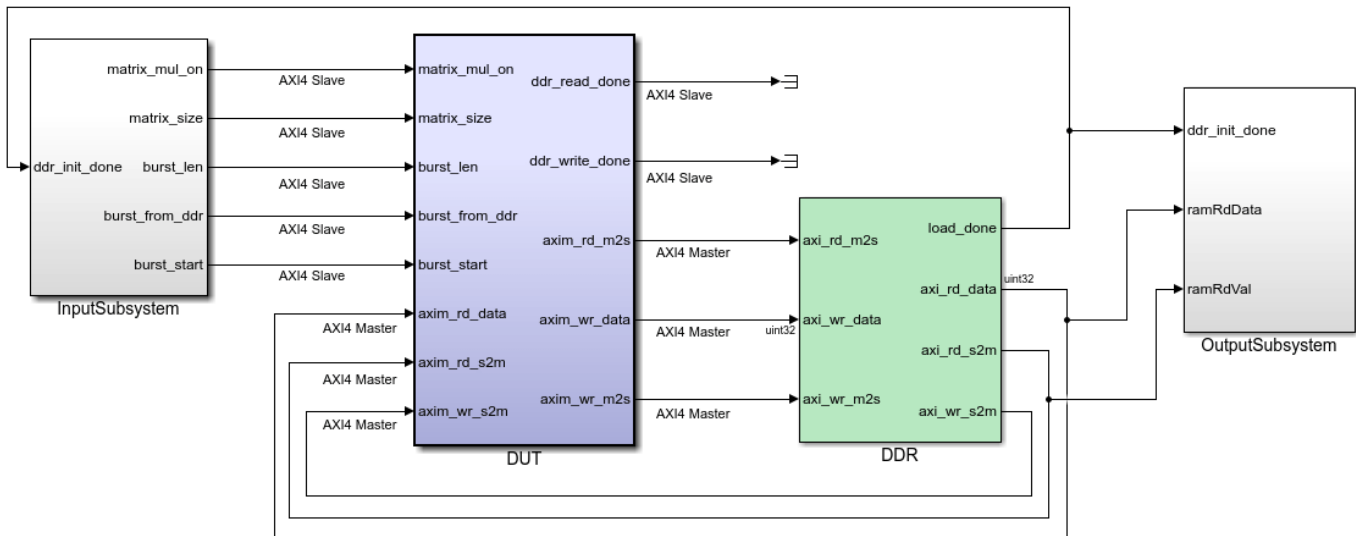
This example models a matrix vector multiplication algorithm and implements the algorithm on the Xilinx Kintex-7 KC705 board. Large matrices might not map efficiently to block RAMs on the FPGA fabric. Instead, store the matrices in the external DDR3 memory on the FPGA board. The Ethernet-based AXI manager interface can access the data by communicating with vendor-provided memory interface IP cores that interface with the DDR3 memory. This capability enables you to model algorithms that involve large data processing and requires high-throughput DDR access, such as matrix operations and computer vision algorithms.

The matrix vector multiplication module supports fixed-point matrix vector multiplication, with a configurable matrix size ranging from 2 to 4000. The size of the matrix is run-time configurable through the AXI4 accessible register.

Open the model by entering this command at the MATLAB® command prompt.

```
modelName = 'hdlcoder_external_memory_axi_master';  
open_system(modelName);
```

Using IP Core Generation Workflow: External Memory Access



This example shows how to use HDL Coder to generate a custom IP core which perform large matrix operations on FPGAs using external memory.

In MATLAB, type the following:
`hdladvisor('hdlcoder_external_memory_axi_master/DUT')`

Launch HDL Workflow Advisor

Run Demo

Copyright 2017 The MathWorks, Inc.

Model Algorithm

This example model includes an FPGA implementable design under test (DUT) block, a DDR functional behavior block, and a test environment to drive inputs and verify the expected outputs.

The DUT subsystem contains an AXI4 Master read and write controller along with a matrix vector multiplication module. Using the AXI4 Master interface, the DUT subsystem reads data from the external DDR3 memory, feeds the data into the `Matrix_Vector_Multiplication` module, and then writes the output data to the external DDR3 memory using the Ethernet-based AXI manager interface. The DUT module has several parameter ports. These ports are mapped to AXI4 accessible registers, so you can adjust these parameters from MATLAB even after you implement the design onto the FPGA.

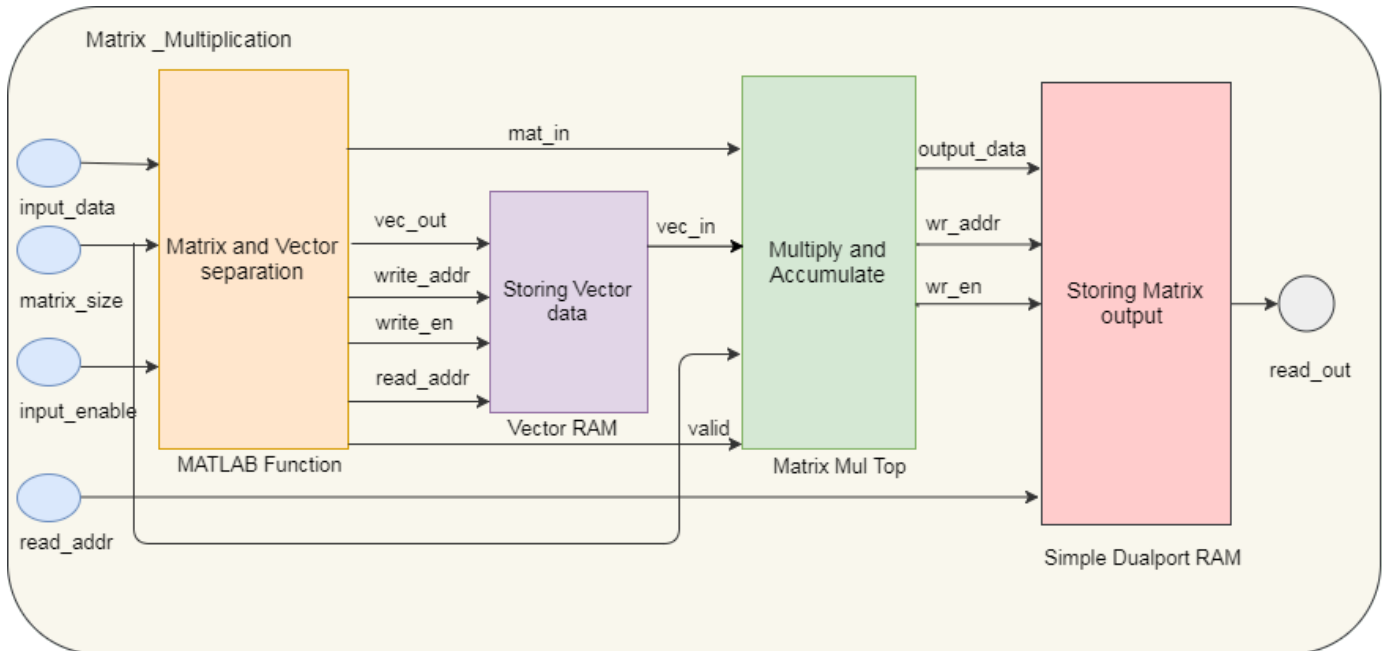
The `matrix_mul_on` port controls whether to run the `Matrix_Vector_Multiplication` module. When the input to `matrix_mul_on` is true, the DUT subsystem performs matrix vector multiplication, as described earlier in this example. When the input to `matrix_mul_on` is false, the DUT subsystem performs a data loop back mode. In this mode, the DUT subsystem reads data from the external DDR3 memory, writes it into the `Internal_Memory` module, and then writes the same data back to the external DDR3 memory. The data loop back mode is a way to verify the functionality of the AXI4 Master external DDR3 memory access.

Also inside the DUT subsystem, the `Matrix_Vector_Multiplication` module uses a multiply-add block to implement a streaming dot-product computation for the inner-product of the matrix vector multiplication.

If A is a matrix of size N -by- N and B is a vector of size N -by-1, then the matrix vector multiplication output is $Z = A \times B$, which is of size N -by-1.

The first N values from the DDR are treated as the N-by-1 size vector, followed by N-by-N size matrix data. The first N values (vector data) are stored into a RAM. From N+1 values onward, data is directly streamed as matrix data. Vector data is read from the Vector_RAM in parallel. Both matrix and vector inputs are fed into the Matrix_mul_top subsystem. The first matrix output is available after N clock cycles and is stored in output RAM. Again, the vector RAM read address is reinitialized to 0 and starts reading the same vector data corresponding to the new matrix stream. This operation is repeated for all of the rows of the matrix.

This diagram shows the architecture of the Matrix_Vector_Multiplication module.



Generate HDL IP core with Ethernet-Based AXI Manager

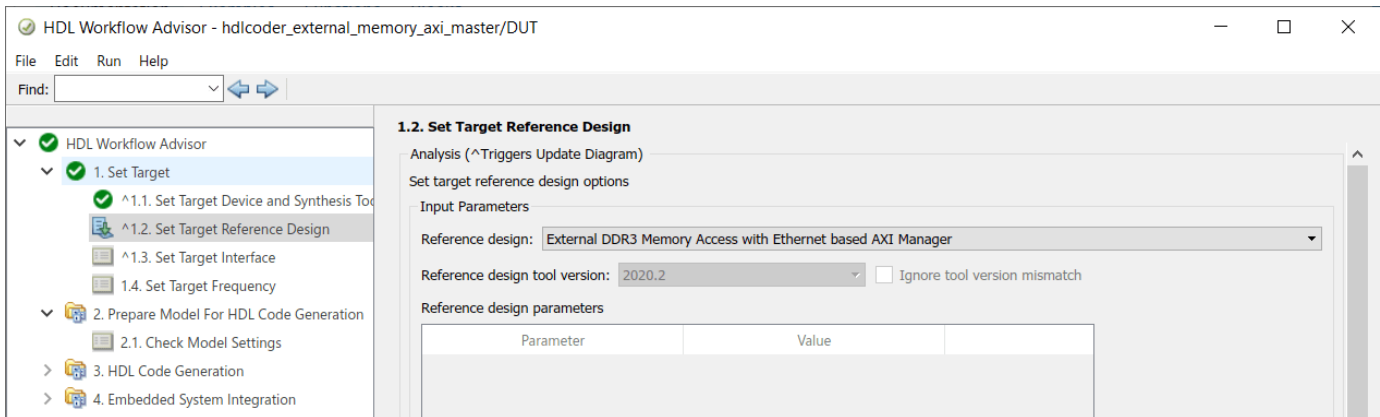
Start the HDL Workflow Advisor and use the IP Core Generation workflow to deploy this design on the Xilinx Kintex-7 hardware.

1. Set up the Xilinx Vivado synthesis tool path by entering this command at the MATLAB command prompt. Use your own Vivado installation path when you run the command.

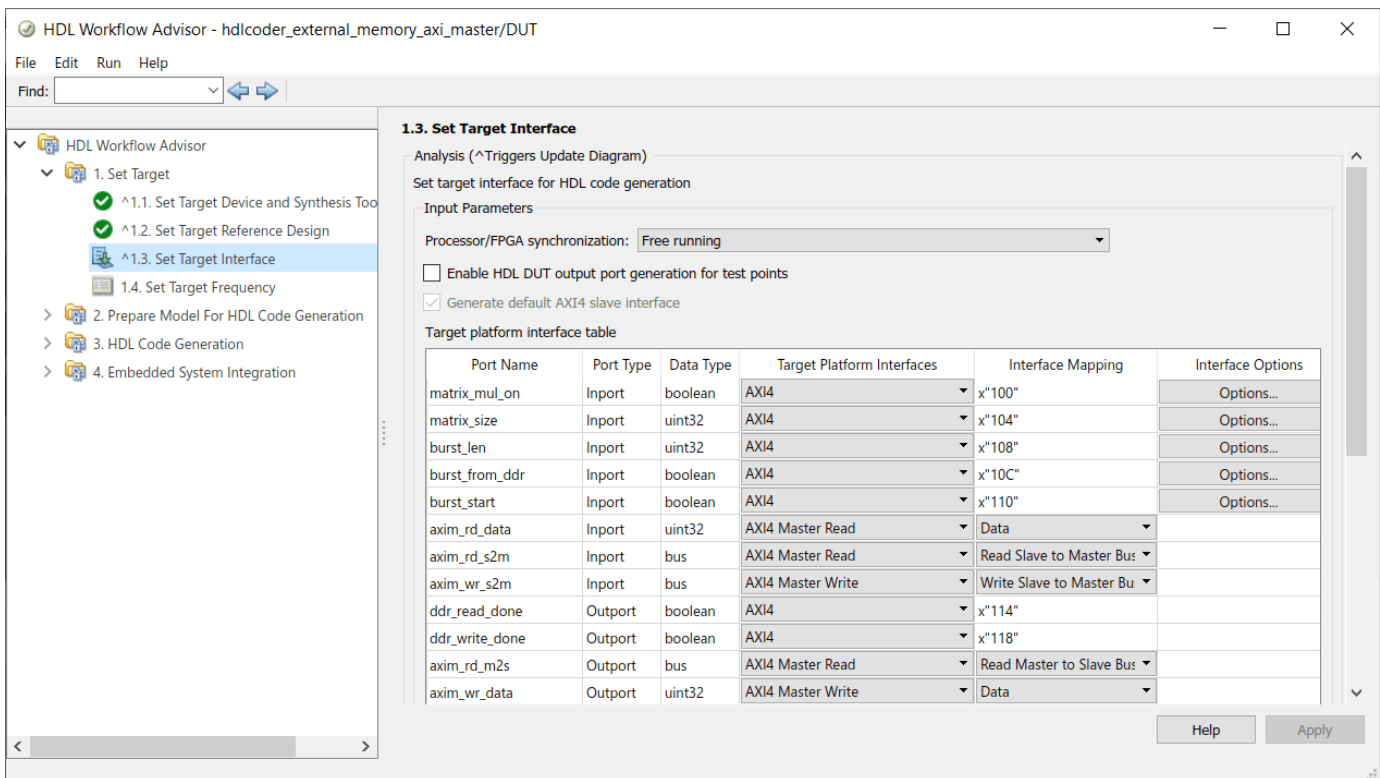
```
hdlsetuptoolpath('ToolName','Xilinx Vivado', ...
                'ToolPath','C:\Xilinx\Vivado\2020.2\bin\vivado.bat')
```

2. Start the HDL Workflow Advisor from the DUT subsystem `hdlcoder_external_memory_axi_master/DUT`. The target interface settings are saved on the model. In step 1.1, the target workflow is the IP Core Generation workflow and the target platform is the Xilinx Kintex-7 KC705 development board.

3. In step 1.2, select the **Reference design** as External DDR3 Memory Access with Ethernet based AXI Manager.



4. Review the target platform interface table settings.



In this example, the input parameter ports, such as **matrix_mul_on**, **matrix_size**, **burst_len**, **burst_from_dds**, and **burst_start**, are mapped to the AXI4 interface. The HDL Coder product generates AXI4 interface accessible registers for these ports. You can use MATLAB to tune these parameters at run-time when the design is running on the FPGA board.

The Ethernet-based AXI manager interface has separate read and write channels. The read channel ports, such as **axim_rd_data**, **axim_rd_s2m**, and **axim_rd_m2s**, are mapped to the AXI4 Master Read interface. The write channel ports, such as **axim_wr_data**, **axim_wr_s2m**, and **axim_wr_m2s**, are mapped to the AXI4 Master Write interface.

The AXI manager feature is used to initialize the external DDR3 memory with input vector and matrix data and to clear the output DDR memory location.

The DUT calculation is started by controlling the AXI4 accessible registers. The DUT IP core reads input data from the DDR memory, performs matrix vector multiplication, and then writes the result back to the DDR memory.

Finally, the output result is read back to MATLAB and compared with the expected value. In this way, the hardware results are verified in MATLAB.

```
>> hdlcoder_external_memory_axi_master_hw_run
Initializing external DDR3 memory (data size 250500) ...
Starting DUT IP core processing ...
Verifying result ...
PASSED: Matrix vector multiplication output matches with the expected data.
```

See Also

More About

- “Set Up AXI Manager” on page 3-2
- “Ethernet AXI Manager” on page 3-10
- “Troubleshooting”

Access FPGA Memory Using Ethernet-Based AXI Manager

This example shows how to use Ethernet-based AXI manager to access internal and external memories of FPGA through different UDP ports. In the FPGA, Xilinx® DDR memory controller and BRAM controller exist for accessing the DDR memories and the BRAM, respectively. These memory controllers provide an AXI4 slave interface for write and read operations by other components in the FPGA. Ethernet-based AXI manager provides an AXI manager component that you can use to access AXI subordinate IPs in the FPGA. This example demonstrates how to integrate Ethernet-based AXI manager into a Xilinx Vivado® project and how to write and read from the DDR memory and the BRAM using MATLAB®.

Requirements

- Xilinx Vivado Design Suite, with supported version listed in “Supported EDA Tools and Hardware” on page 1-6
- Xilinx Kintex®-7 KC705 Evaluation Kit
- HDL Verifier™ Support Package for Xilinx FPGA Boards
- Ethernet cable and JTAG cable

Setup

1. Set up the FPGA board. Connect the Xilinx KC705 board to the host computer via Ethernet and JTAG cables. The JTAG cable is used for programming the device.
2. Prepare the example in MATLAB. Set up the Xilinx Vivado tool path. Use your own Xilinx Vivado installation path when executing the command.

```
hdlsetuptoolpath('ToolName','Xilinx Vivado', ...
                 'ToolPath','C:\Xilinx\Vivado\2020.2\bin\vivado.bat');
```

Create a Vivado project for this example. This project contains the IP Integrator block diagram and constraint file.

```
system('vivado -mode batch -source createProjectEthAxim.tcl')
```

This command takes about one minute to run. When the execution completes, a Vivado project named `ethernetaximaster.xpr` appears in your current directory.

3. Configure the Vivado project with a Vivado IP. To use the UDP AXI Manager IP inside the Vivado IP Integrator, add the folder that contains the IP to the IP repository path setting for the Vivado project. Add the path to the project by executing this command in MATLAB.

```
setupAXIManagerForVivado ethernetaximaster.xpr
```

Open the generated Vivado project in GUI mode by double-clicking the project in a file browser or executing this command in MATLAB.

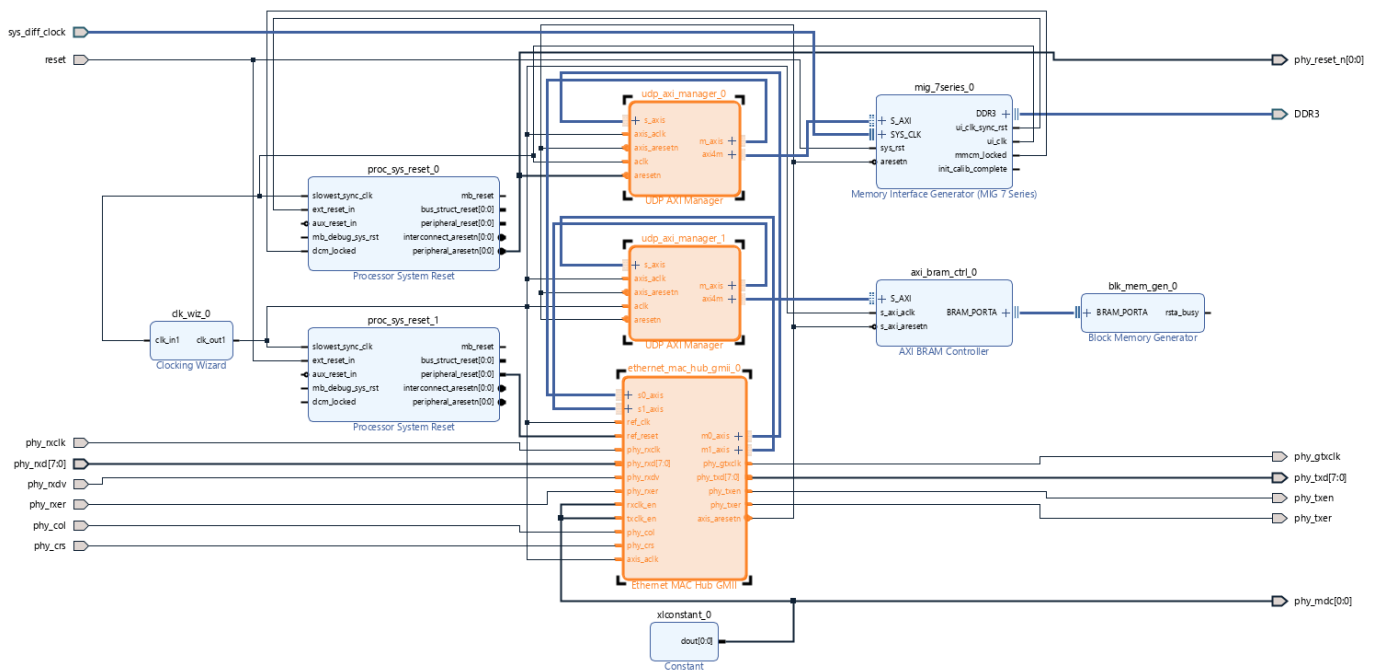
```
system('vivado ethernetaximaster.xpr &')
```

4. In the Vivado GUI, open the block diagram design file `design_1.bd`. You can find the block diagram in the source file subwindow. Add the Ethernet MAC Hub GMII and UDP AXI Manager IPs to the FPGA design. The Ethernet MAC Hub GMII IP has a default target IP Address of 192.168.0.2 and default UDP port value of 50101. Change the number of AXI Stream channels of Ethernet MAC Hub

to 2. You can change these values by double-clicking the `ethernet_mac_hub_gmii_0` IP in the block design. Modify the `ethernet_mac_hub_gmii_0` parameters as shown in this figure.

Component Name <code>ethernet_mac_hub_gmii_0</code>	
Number of AXI Stream Channels	2
MAC Address	0x000A3502218A
IP Address Byte1	192
IP Address Byte2	168
IP Address Byte3	0
IP Address Byte4	2
UDP Port For Channel 1	50101
UDP Port For Channel 2	50102
UDP Port For Channel 3	50103
UDP Port For Channel 4	50104
UDP Port For Channel 5	50105
UDP Port For Channel 6	50106
UDP Port For Channel 7	50107
UDP Port For Channel 8	50108

Make the connections among IPs as shown in this figure. You can access DDR memory and BRAM through UDP ports 50101 and 50102, respectively.



Set the address of mig_7series_0 (DDR controller) and axi_bram_ctrl_0 (BRAM controller) to 0x0000_0000 and 0x1000_0000, respectively, as shown in this figure.

Name	Interface	Slave Segment	Master Base Address	Range	Master High Address
Network 0					
/udp_axi_manager_0					
/udp_axi_manager_0/axi4m (32 address bits : 4G)					
/mig_7series_0/memmap	S_AXI	memaddr	0x0000_0000	1G	0x3FFF_FFFF
Network 1					
/udp_axi_manager_1					
/udp_axi_manager_1/axi4m (32 address bits : 4G)					
/axi_bram_ctrl_0/S_AXI	S_AXI	Mem0	0x1000_0000	8K	0x1000_1FFF

Alternatively, you can complete the above setup steps by executing Tcl commands in Vivado.

```
source ./modifyDesignEthAxim.tcl
```

5. Generate the FPGA programming file and program the FPGA. Click **Generate Bitstream** on the Vivado window to generate the FPGA programming file. Vivado might prompt you to save the project before proceeding to the next step. Generating the bitstream file takes about 5 to 10 minutes for Vivado to generate the bitstream file.

After Vivado generates the bitstream, verify that FPGA board is connected with Digilent® JTAG and Ethernet cables. Program the FPGA by executing this command in MATLAB.

```
filProgramFPGA('Xilinx Vivado', 'ethernetaximaster.runs\impl_1\design_1.bit', 1)
```

6. Make sure that the host network connection must be on the same subnet as the hardware board. For this example, set the host network IP address to 192.168.0.x, where x is any number in the range 1 through 255, apart from 2. 192.168.0.2 is the IP address of the hardware board in this example.

FPGA Write and Read Operations

After programming the FPGA, you can write and read from the AXI subordinates that are connected to the UDP AXI Manager IP. This example writes data to the DDR memory connected to the FPGA and the BRAM and then retrieves data into MATLAB.

Create the AXI manager object in MATLAB to write and read from the DDR memory. The default port address is 50101. Write single location and then read data from the same location. In this case, the read back data is 100.

```
hDDR = aximanager('Xilinx','interface','UDP', ...  
                 'DeviceAddress','192.168.0.2');  
writememory(hDDR,'00000000',100);  
readmemory(hDDR,'00000000',1);
```

Release the AXI manager object to open communication through other UDP ports.

```
release(hDDR);
```

Create a new AXI manager object with a different UDP port to write and read from the BRAM. Write single location and then read data from the same location. In this case, the read back data is 1000.

```
hBRAM = aximanager('Xilinx','interface','UDP', ...  
                  'DeviceAddress','192.168.0.2','Port','50102');  
writememory(hBRAM,'10000000',1000);  
readmemory(hBRAM,'10000000',1);  
release(hBRAM);
```

See Also

[aximanager](#) | [writememory](#) | [readmemory](#) | [release](#)

More About

- “Set Up AXI Manager” on page 3-2
- “Ethernet AXI Manager” on page 3-10
- “Troubleshooting”

Access FPGA External Memory Using AXI Manager over PCI Express

This example shows how to use AXI manager over PCI Express® (PCIe) to access the external memory connected to an FPGA. The FPGA includes a Xilinx® DDR memory controller for accessing the DDR memory. This memory controller provides an AXI4 slave interface for write and read operations by other components in the FPGA. The PCIe AXI manager feature provides an AXI manager object that you can use to access any memory mapped location in the FPGA. This example shows how to integrate PCIe AXI manager into a Xilinx Vivado® project and write and read data from the DDR memory using MATLAB®.

Requirements

- Xilinx Vivado Design Suite, with supported version listed in “Supported EDA Tools and Hardware” on page 1-6
- Xilinx Kintex® UltraScale+ FPGA KCU116 Evaluation Kit
- HDL Verifier™ Support Package for Xilinx FPGA Boards
- Host machine (PC) with PCIe slot
- USB-JTAG cable

Setup

1. Set up the FPGA board. Connect the Xilinx KCU116 board to the host computer via PCIe and JTAG cables. The JTAG cable is used for programming the device.
2. Prepare the example in MATLAB. Set up the Xilinx Vivado tool path. Use your own Xilinx Vivado installation path when executing the command.

```
hdlsetuptoolpath('ToolName','Xilinx Vivado', ...
                 'ToolPath','C:\Xilinx\Vivado\2020.2\bin\vivado.bat');
```

Create a Vivado project for this example. The following MATLAB command creates a Vivado project named `pcieaximaster.xpr` and contains the IP Integrator block diagram and constraint files.

```
system('vivado -mode batch -source pcieAXIMcreateproject.tcl')
```

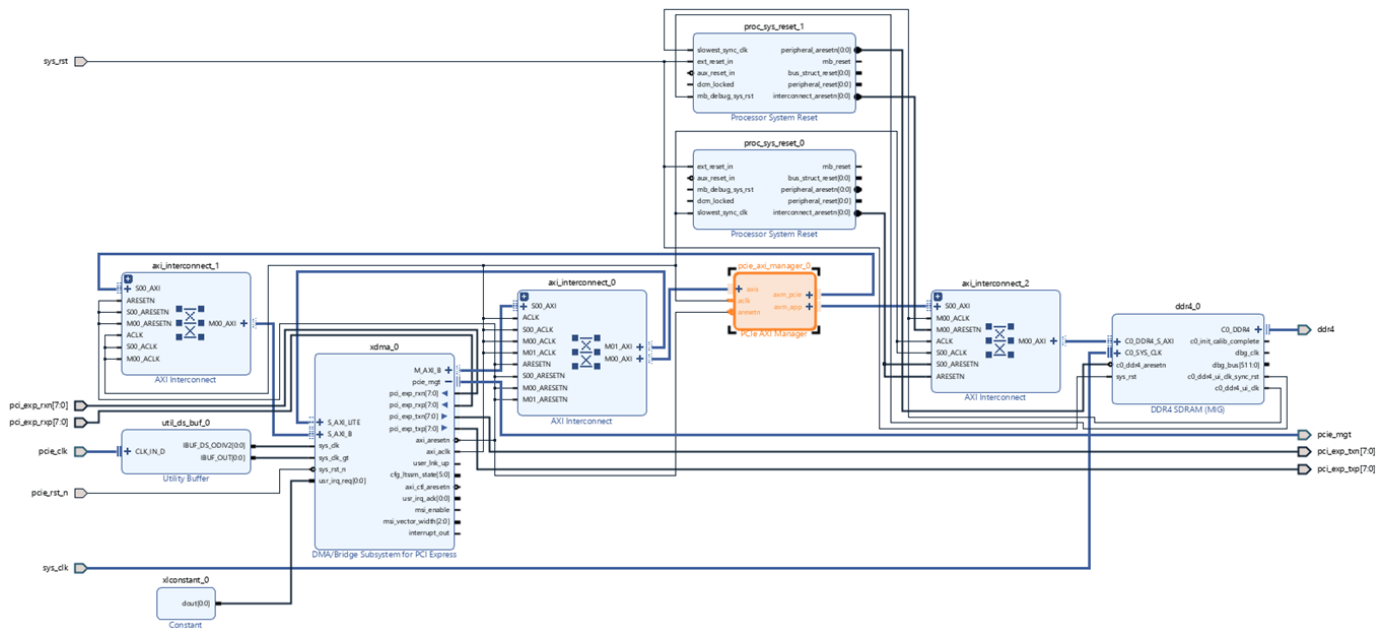
3. Configure the Vivado project with a Vivado IP. To use the PCIe as AXI Manager IP inside the Vivado IP Integrator, add the folder that contains the IP to the IP repository path setting for the Vivado project. Add the path to the project by executing this command in MATLAB.

```
setupAXIManagerForVivado pcieaximaster.xpr
```

Open the generated Vivado project in GUI mode by double-clicking the project in a file browser or by executing this command in MATLAB.

```
system('vivado pcieaximaster.xpr &')
```

4. Add PCIe AXI Manager IP to the FPGA design. In the Vivado GUI, open the block diagram design file `pcieAXIMdesign_1.bd`. You can find the design in the source file subwindow.



Set the address of xdma_0 (AXI Bridge Subsystem for PCI Express) and ddr4_0 (memory controller) as shown in this figure.

Diagram | Address Editor | Address Map |

Assigned (4) | Unassigned (0) | Excluded (0) | Hide All

Name	Interface	Slave Segment	Master Base Address	Range	Master High Address
Network 0					
/pcie_axi_manager_0					
/pcie_axi_manager_0/axm1 (32 address bits : 4G)					
/xdma_0/S_AXI_B	S_AXI_B	BAR0	0x0000_0000	4G	0xFFFF_FFFF
Network 1					
/pcie_axi_manager_0					
/pcie_axi_manager_0/axm2 (32 address bits : 4G)					
/DDR4_0/C0_DDR4_MEMORY_MAP	C0_DDR4_S_AXI	C0_DDR4_ADDRESS_BLOCK	0x0000_0000	1G	0x3FFF_FFFF
Network 2					
/xdma_0					
/xdma_0/M_AXI_B (32 address bits : 4G)					
/pcie_axi_manager_0/axis	axis	reg0	0x0000_4000	16K	0x0000_7FFF
/xdma_0/S_AXI_LITE	S_AXI_LITE	CTL0	0x0000_0000	16K	0x0000_3FFF

Alternatively, you can complete the above setup steps by executing Tcl commands in Vivado.

```
source ./pcieAXIMmodifydesign.tcl
```

5. Generate the FPGA programming file and program the FPGA. Click **Generate Bitstream** on the Vivado window to generate the FPGA programming file. Vivado might prompt you to save the project before proceeding to the next step. Generating the bitstream file takes about 5 to 10 minutes for Vivado to generate the bitstream file.

After Vivado generates the bitstream, program the FPGA by executing this command in MATLAB.

```
filProgramFPGA('Xilinx Vivado','pcieaximaster.runs\impl_1\pcieAXIMdesign_1.bit',1)
```

6. Reboot the host machine after programming the FPGA.

FPGA Write and Read Operations

Once the design is running on the FPGA board, you can write and read from the AXI subordinates that are connected to the PCIe AXI Manager IP. This example writes data to the DDR memory connected to the FPGA and then retrieves data into MATLAB.

Create the AXI manager object in MATLAB.

```
h = aximanager('Xilinx','interface','pcie');
```

Write and read from the memory locations on the FPGA. The following two lines use the AXI manager object `h` to write 100 to address 0 and then read from address 0 of the DDR memory.

```
writememory(h,0,100) readmemory(h,0,1)
```

See Also

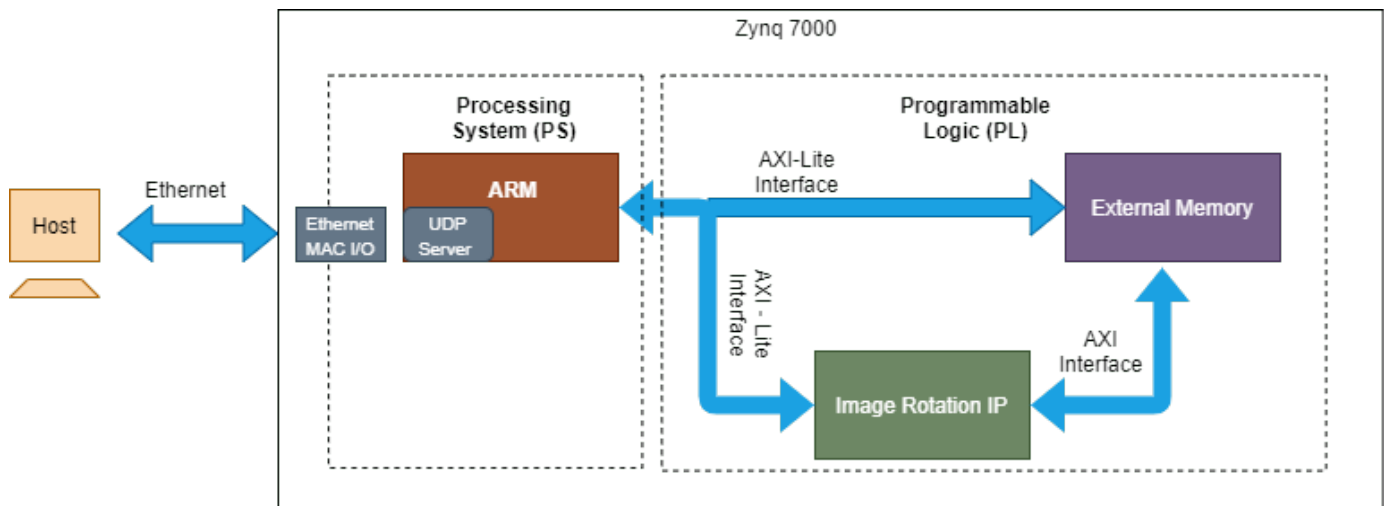
[aximanager](#) | [writememory](#) | [readmemory](#) | [release](#)

More About

- “Set Up AXI Manager” on page 3-2
- “PCI Express AXI Manager” on page 3-6
- “Troubleshooting”

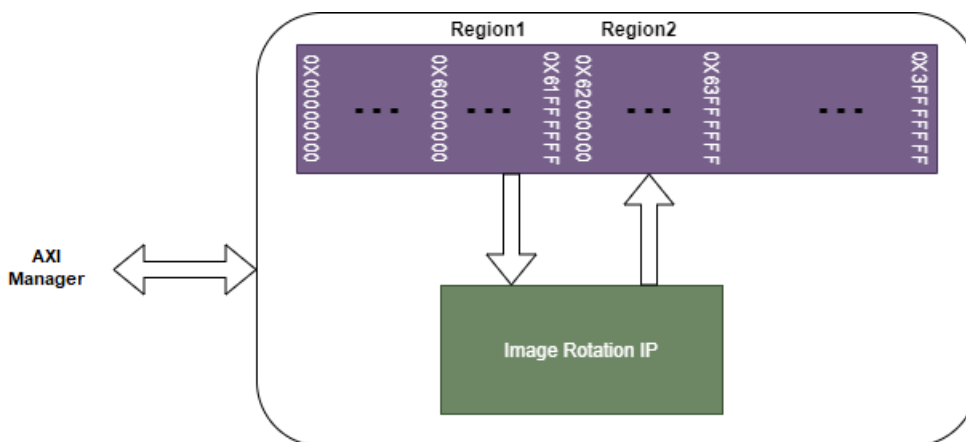
Leverage Built-In Ethernet on Zynq to Perform Memory Access Using AXI Manager

This example shows how to use an Ethernet-based AXI manager to access the external memory and FPGA IPs on the Xilinx® Zynq®-7000 ZC706 board over Ethernet. In Xilinx Zynq-based designs, MATLAB® acts as an AXI manager and communicates with the external memory controller and FPGA IPs through an AXI Lite interface by using the user datagram protocol (UDP) server in the processing system (PS). This block diagram shows the communication between a host and a Xilinx Zynq-7000 board over Ethernet.



The example demonstrates operations as outlined in these steps and this figure.

- 1 Write an ASCII image of size 24-by-64 to **Region1**.
- 2 Configure **Image Rotation IP** to read the image from **Region1**, rotate the image, and write the rotated image to **Region2**.
- 3 Read the image from **Region2**.



Hardware and Software Requirements

To run this example, you must have this software and hardware installed and set up.

- Xilinx Vivado® Design Suite, with a supported version listed in the “HDL Language Support and Supported Third-Party Tools and Hardware” (HDL Coder)
- Xilinx Zynq ZC706 evaluation kit
- Ethernet cable to connect the ZC706 FPGA
- HDL Coder™ Support Package for Xilinx Zynq Platform
- HDL Verifier™ Support Package for Xilinx FPGA Boards
- SD card

Setup

Step 1: Set up the FPGA board

Verify that the Xilinx Zynq-7000 ZC706 board is connected to the host computer through an Ethernet cable. The Ethernet cable is used to program and communicate with the board.

Step 2: Set up the SD card

For details, see steps 1 through 4 in “Ethernet AXI Manager for Xilinx Zynq SoC Devices” on page 3-16.

Step 3: Create a Vivado project

Set up the Xilinx Vivado tool path. Use your Xilinx Vivado installation path when executing the command in MATLAB. For example, enter this command at the MATLAB command prompt.

```
hdlsetuptoolpath('ToolName','Xilinx Vivado','ToolPath', ...  
                'C:\Xilinx\Vivado\2019.2-mw-0\Win\Vivado\2020.2\bin');
```

Create a folder outside of the scope of your MATLAB installation folder into which you can copy the example files. The folder must be writable. This example assumes that the folder is located at C:\MyTests.

Unzip the `imagerotation_ip.zip` file to add the image rotation IP to the **User Repository** in your Vivado project.

```
unzip(fullfile('ipcore','imagerotation_ip_v1_0.zip'), ...  
       fullfile('ipcore','imagerotation_ip_v1_0'));
```

Create a Vivado project using this command. This project contains an IP Integrator block diagram and a constraint file.

```
system('vivado -mode batch -source createproject.tcl')
```

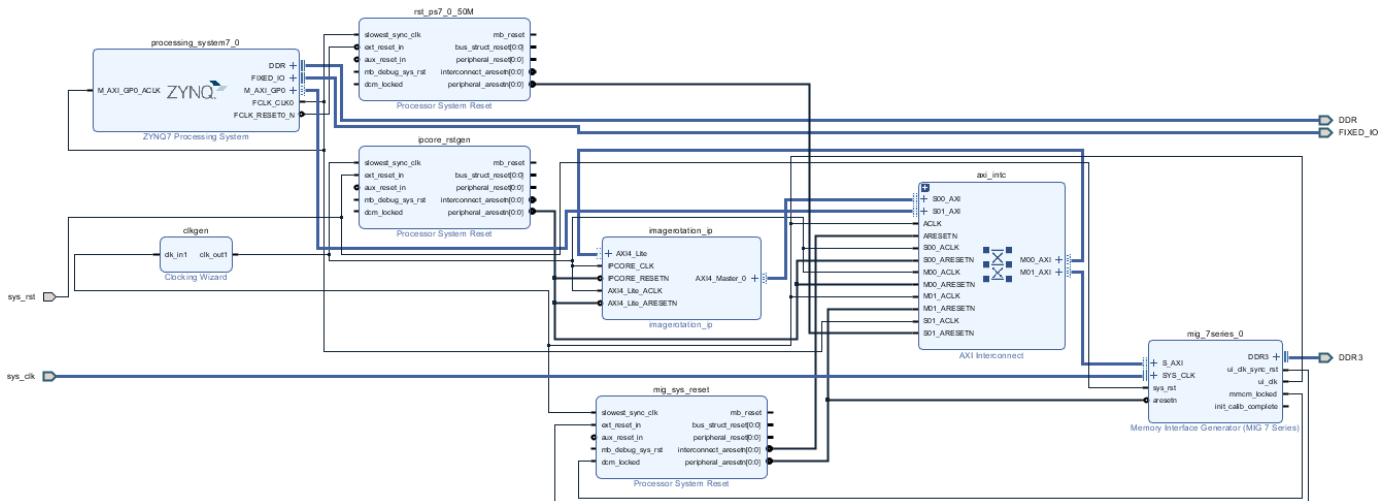
This process completes in about a minute. A Vivado project named `ethernetaximasterzynq.xpr` is created in your current directory.

Open the generated Vivado project in GUI mode by double-clicking the project in a file browser or by entering this command at the MATLAB command prompt.

```
system('vivado ethernetaximasterzynq.xpr &')
```

Step 4: Open the block diagram and Address Editor window

In the Vivado GUI, open the block diagram design file `design_1.bd`. You can find this file in the source file subwindow.



Open the Address Editor window to view the address mapping of the **mig_7series_0** (memory controller) and **imagerotation_ip** IPs.

Cell	Slave Interface	Slave Segment	Offset Address	Range	High Address
processing_system7_0					
Data (32 address bits : 0x40000000 [1G])					
imagerotation_ip	AXI4_Lite	reg0	0x4000_0000	64K	0x4000_FFFF
mig_7series_0	S_AXI	memaddr	0x6000_0000	64M	0x63FF_FFFF
imagerotation_ip					
AXI4_Master_0 (32 address bits : 4G)					
mig_7series_0	S_AXI	memaddr	0x6000_0000	64M	0x63FF_FFFF

These are the **imagerotation_ip** registers and their corresponding physical locations.

- IP Core reset register ['40000000'] — Write 1 to reset the image rotation IP.

- IP Core enable register ['40000004'] — Write 1 to enable the image rotation IP.
- Read base address register ['40000008'] — Contains starting address of **Region1** (for example, 0X60000000).
- Write base address register ['4000000C'] — Contains starting address of **Region2** (for example, 0X62000000).
- Start register ['40000100'] — Write 1 to start the image rotation operation.

Note: To use the pregenerated bitstream and the device tree blob (DTB) file copied to the current working directory, skip steps 5 and 6.

Step 5: Generate the FPGA programming file

To generate the FPGA programming file, click **Generate Bitstream** in the Vivado window. Vivado might prompt you to save the project before moving forward. Vivado generates the bitstream file in about 5 to 10 minutes.

Step 6: Generate the DTB file

To compile a DTB file, you need a device tree compiler (DTC) on a Linux machine. If a DTC is not installed, execute these commands in a Linux console window.

```
sudo apt-get update -y
sudo apt-get install -y device-tree-compiler
```

After successful installation of a DTC, to generate a DTB file, open the board-specific DTS file to provide the FPGA memory information as the code in this figure shows.

```
mw_imagerotation_ip@0 {
compatible = "mathworks,mwipcore-v2.00";
reg = <0x40000000 0x10000>;
linux,phandle = <0x36>;
phandle = <0x36>;
};
.....
mw_mig_7series_ip@0 {
compatible = "mathworks,mwipcore-v2.00";
reg = <0x60000000 0x1000000>;
linux,phandle = <0x37>;
phandle = <0x37>;
};
```

Alternatively, you can use the DTS file provided for this example: C:\MyTests\devicetree_zc706_image_rotation.dts.

Generate a DTB file from the DTS file by executing this command in a Linux console window.

```
dtc -I dts -O dtb devicetree_zc706_image_rotation.dts -o
devicetree_zc706_image_rotation.dtb
```

Step 7: Program the FPGA

Program the FPGA in MATLAB by entering this command at the MATLAB command prompt.

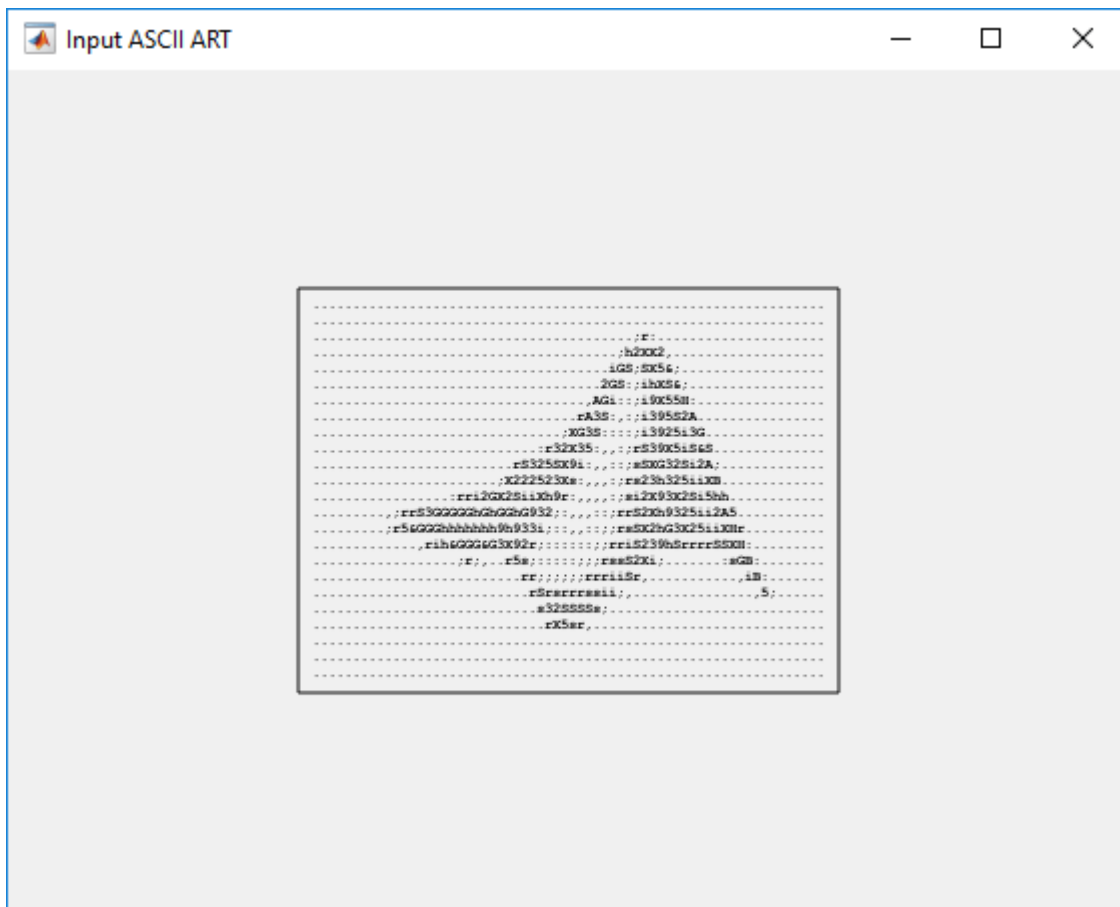
```
loadBitstream('ZC706','ethernetaximaster_zynq_zc706.bit', ...
             'devicetree_zc706_image_rotation.dtb');
```

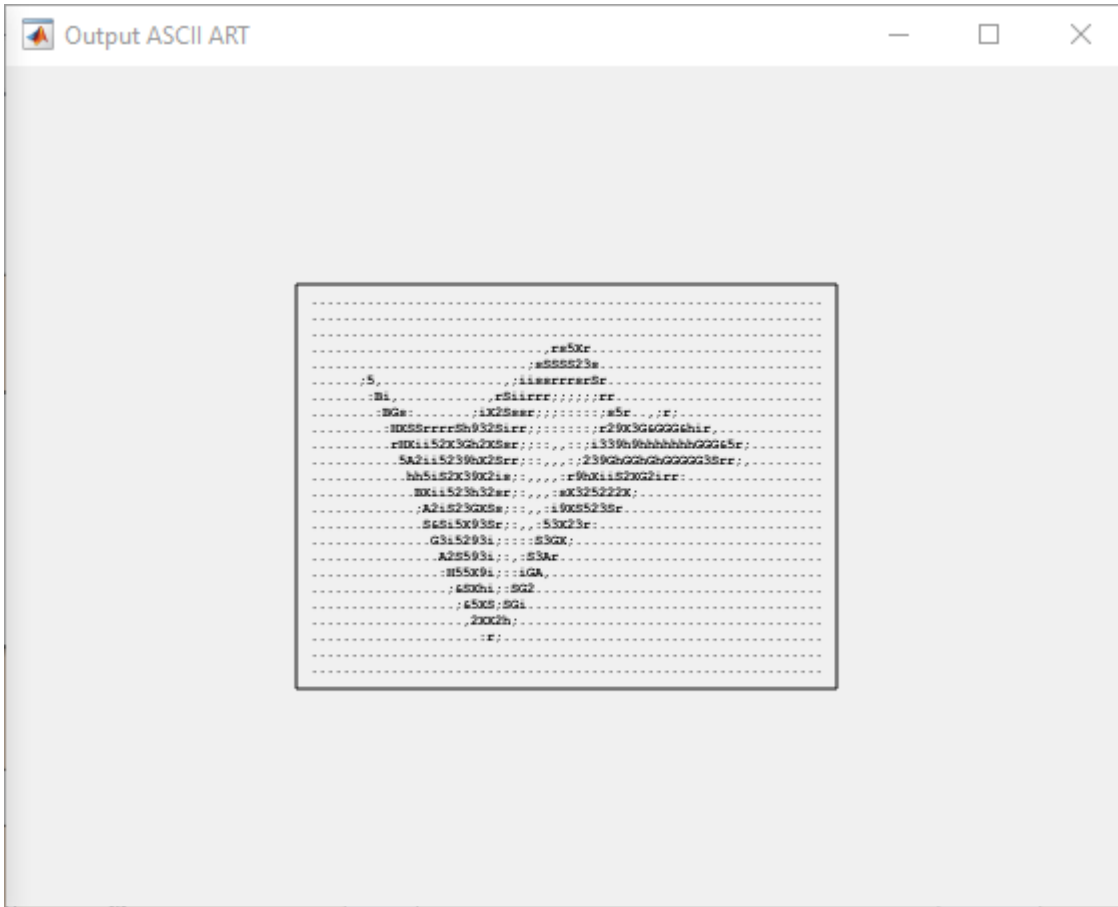
Run Design

To run the design, run the `ethernetaximasterzynq_tb.m` file from the current working folder at MATLAB command prompt.

```
ethernetaximasterzynq_tb
```

The script runs the design from MATLAB by creating an AXI manager object and plots the input and output images as these figures show.





See Also

More About

- “Set Up AXI Manager” on page 3-2
- “Ethernet AXI Manager” on page 3-10
- “Troubleshooting”

Access DUT Registers on Xilinx Pure FPGA Board Using IP Core Generation Workflow

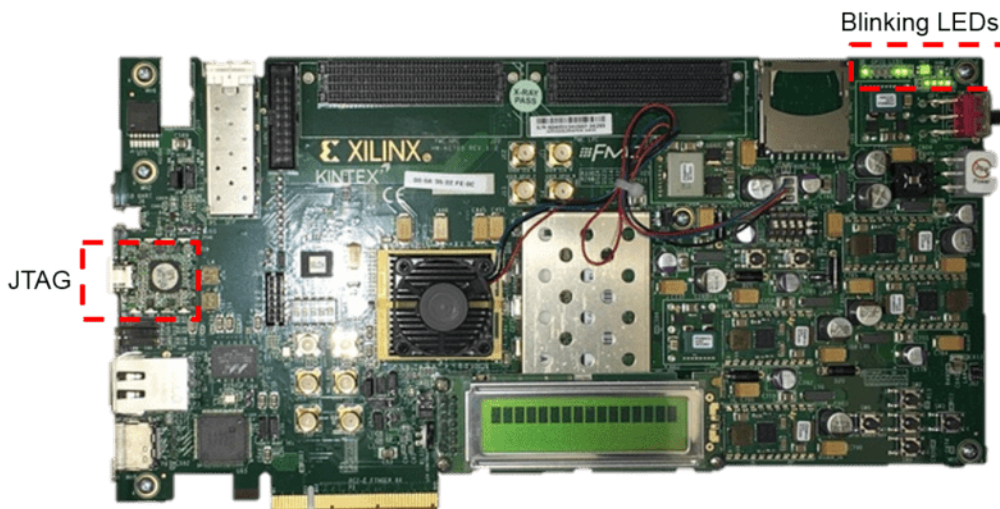
This example shows how to use the HDL Coder™ IP core generation workflow to develop reference designs for Xilinx® parts that do not use an embedded ARM® processor present but that still utilize the HDL Coder generated AXI interface to control the design under test (DUT). This example uses the HDL Verifier™ AXI Manager IP to access the HDL Coder generated DUT registers from MATLAB®. Alternatively, you can use the Xilinx JTAG AXI Master to access the DUT registers using Vivado® Tcl console by writing Tcl commands. For the Xilinx JTAG AXI Master, you must create a custom reference design. The FPGA design is implemented on the Xilinx Kintex®-7 KC705 board.

Requirements

- Xilinx Vivado Design Suite, with a supported version listed in “HDL Language Support and Supported Third-Party Tools and Hardware” (HDL Coder)
- Xilinx Kintex-7 KC705 development board
- HDL Coder Support Package for Xilinx FPGA Boards
- HDL Verifier Support Package for Xilinx FPGA Boards

Xilinx Kintex-7 KC705 Development Board

This figure shows the Xilinx Kintex-7 KC705 development board.



Example Reference Designs

Designs that can benefit from using the HDL Coder IP core generation workflow without using either an embedded ARM processor or an Embedded Coder™ support package but still leverage the HDL Coder generated AXI4-Lite registers can include one of these IP sets.

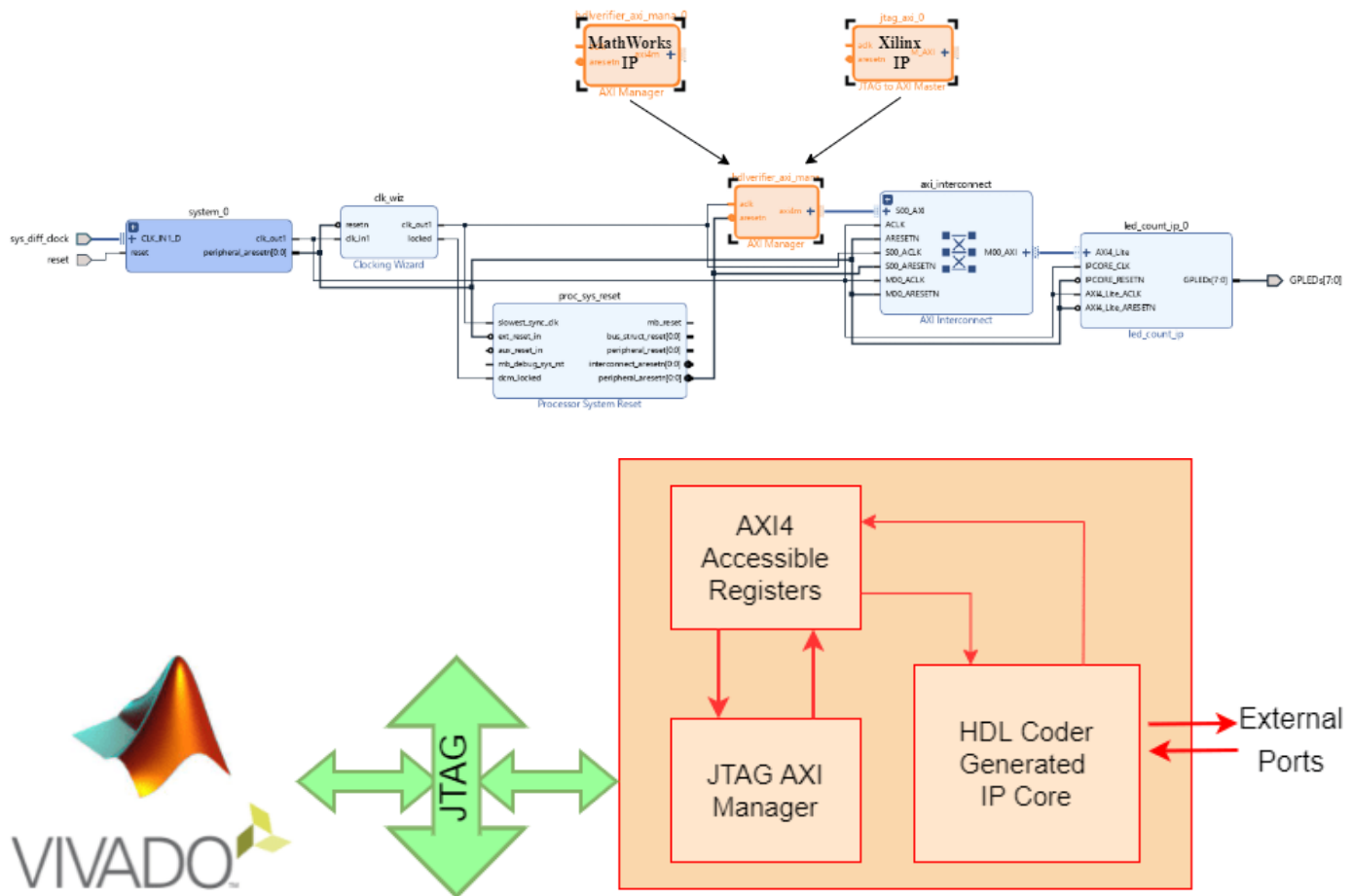
- HDL Verifier AXI Manager + HDL Coder IP Core
- Xilinx JTAG Master + HDL Coder IP Core
- MicroBlaze™ + HDL Coder IP Core

- PCIe Endpoint + HDL Coder IP Core

This example includes two reference designs.

- The Default System reference design uses MathWorks® IP and a MATLAB command line interface for issuing read and write commands. To use this design, you must have the HDL Verifier product.
- The Xilinx JTAG to AXI Master reference design uses Vivado IP for the JTAG to AXI Master and requires using the Vivado Tcl console to issue read and write commands.

The two reference designs differ by only the JTAG manager IP that they use, as this figure shows.



HDL Verifier AXI Manager Reference Design

In the IP core generation workflow of the HDL Workflow Advisor, in the **Set Target Reference Design** step, set the **Insert AXI Manager (HDL Verifier required)** parameter to an interface that communicates between your host machine and the target hardware. This option adds AXI manager IP for your interface automatically into the reference design and connects the added IP to the DUT IP using the AXI4-slave interface. The next section details the steps to auto-insert the JTAG AXI Manager IP in the reference design.

Execute IP Core Workflow

Follow these steps to execute the IP core workflow for the Default System reference design, which uses JTAG AXI Manager IP. Using this reference design, you can generate an HDL IP core that blinks LEDs on the KC705 board. To generate an HDL IP core, follow these steps.

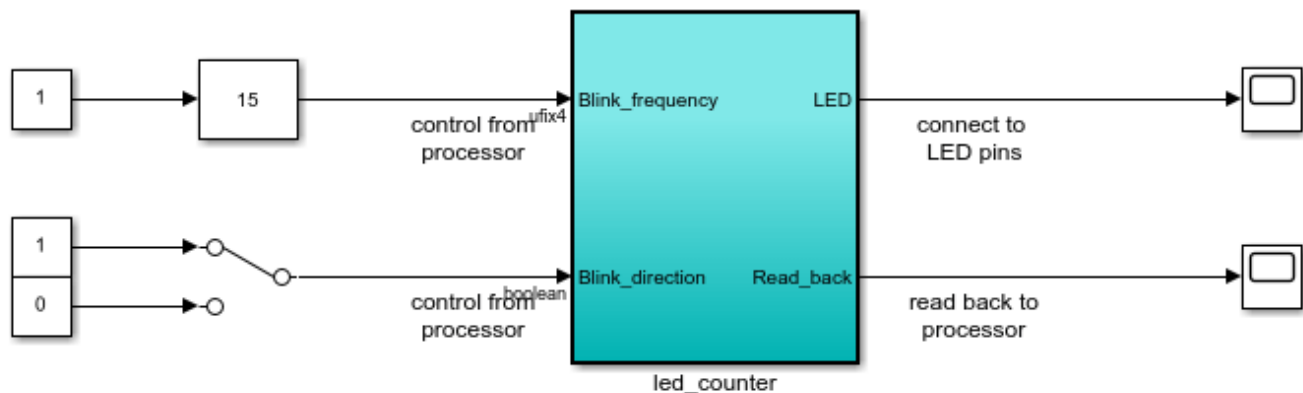
1. Set up the Xilinx Vivado tool path by executing this command in MATLAB. Use your own Xilinx Vivado installation path when executing the command.

```
hdlsetuptoolpath('ToolName','Xilinx Vivado','ToolPath', ...
    'C:\Xilinx\Vivado\2020.2\bin\vivado.bat');
```

2. Open the Simulink model that implements LED blinking by executing this command in MATLAB.

```
open_system('hdlcoder_led_blinking')
```

Using IP Core Generation Workflow: LED Blinking



This example shows how to use HDL Workflow Advisor to generate a custom IP core which blink LEDs on FPGA board.

In MATLAB, type the following:
`hdladvisor('hdlcoder_led_blinking/led_counter')`

Launch HDL Workflow Advisor

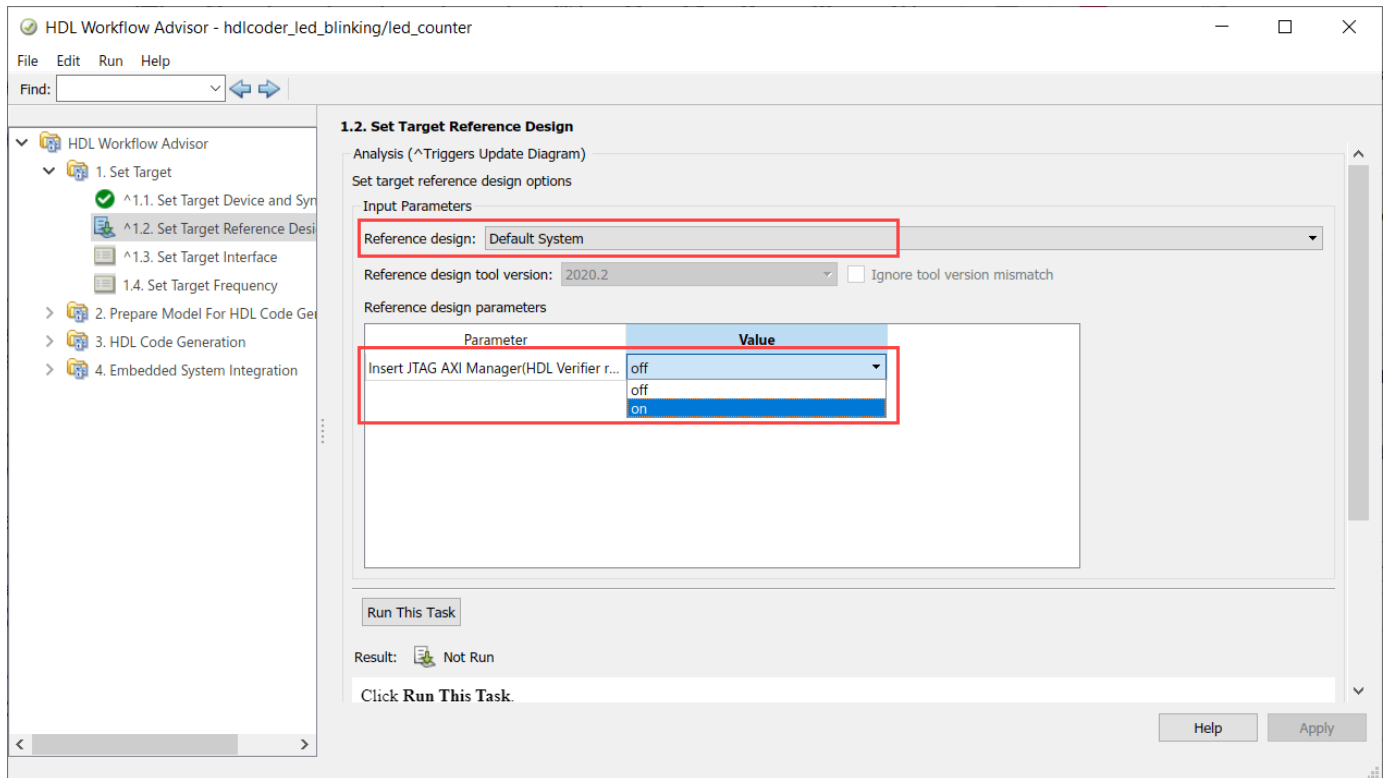
Run Demo

Copyright 2012 The MathWorks, Inc.

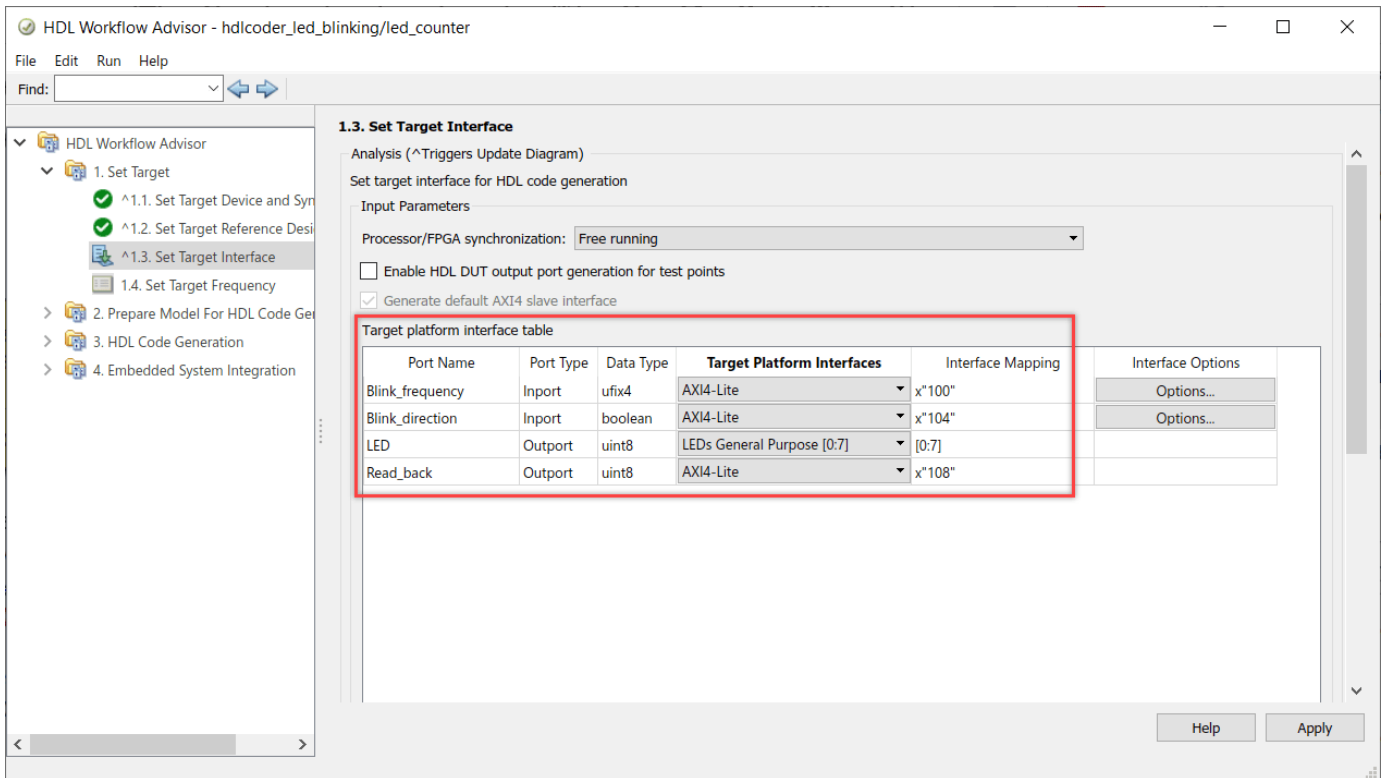
3. Launch HDL Workflow Advisor from the `hdlcoder_led_blinking/led_counter` subsystem by right-clicking the `led_counter` subsystem and selecting **HDL Code** followed by **HDL Workflow Advisor**.

4. In step 1.1, set **Target workflow** to IP Core Generation and **Target platform** to Xilinx Kintex-7 KC705 development board. Click **Run This Task**.

5. In step 1.2, set **Reference design** to Default System. Under **Reference design parameters**, set **Insert AXI Manager (HDL Verifier required)** to JTAG. Click **Run This Task**.

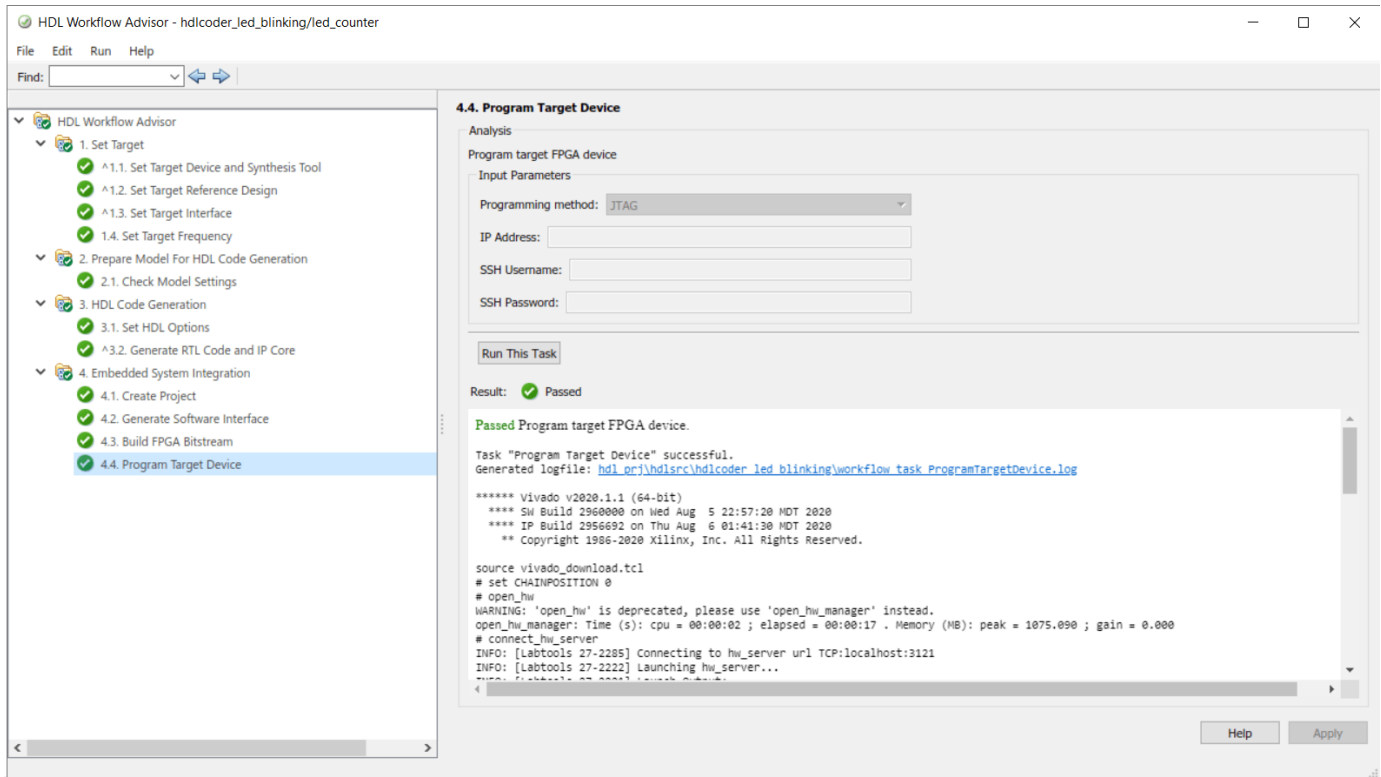


6. In step 1.3, set the interface of the **Blink_frequency**, **Blink_direction**, and **Read_back** ports to AXI4-Lite. Set the interface of the **LED** port to LEDs General Purpose [0:7].



7. Run the remaining steps in the workflow to generate a bitstream and program the target device.

Unlike the Zynq-based reference design, a **Generate Software Interface Model** task does not exist, as this figure shows.



Determine Addresses from IP Core Report

The base address for an HDL Coder IP core is defined as $0x40000000$ for the Default System reference design, which uses the AXI Manager IP. You can see address setting in the generated IP core report as shown in this figure.

Code Generation Report

Find: Match Case

Contents

- Summary
- [Clock Summary](#)
- [Code Interface Report](#)
- Timing And Area Report
 - [High-level Resource Report](#)
- Optimization Report
 - [Distributed Pipelining](#)
 - [Streaming and Sharing](#)
 - [Delay Balancing](#)
 - [Adaptive Pipelining](#)
 - [Hierarchy Flattening](#)
 - [Target Code Generation](#)
 - IP Core Generation Report**
 - [Traceability Report](#)
- Generated Source Files
 - [led_count_ip_src_led_counter_pk](#)
 - [led_count_ip_src_led_counter.vhd](#)
- Referenced Models

Use JTAG AXI Master to control the IP core from MATLAB

In 1.2 Step "Set Target Reference design", "Insert JTAG AXI Manager" is turned "on". This adds Matlab as an "AXI Manager" to control the DUT IP core using AXI4 interface as shown.

JTAG Interface **AXI4 Interface**

The diagram illustrates the connection between MATLAB, the Reference Design, and the DUT IP Core. MATLAB is connected to the Reference Design via a JTAG interface. The Reference Design contains a MATLAB JTAG AXI Master IP, which is connected to the DUT IP Core via an AXI4 interface. The DUT IP Core is connected to the LE (Logic Element) block.

Requires a HDL Verifier license to use this feature. After that use MATLAB® Command line interface to access the DUT IP core registers. **The Base Address of AXI4 Slave is 0x40000000.**

OK Help

The IP core report register address mapping table shows the offsets.

The screenshot shows a window titled "Code Generation Report" with a search bar and "Match Case" option. The left sidebar contains a "Contents" list with "IP Core Generation Report" highlighted. The main content area is titled "Register Address Mapping" and contains the following text:

The following AXI4-Lite bus accessible registers were generated for this IP core:

Register Name	Address Offset	Description
IPCore_Reset	0x0	write 0x1 to bit 0 to reset IP core
IPCore_Enable	0x4	enabled (by default) when bit 0 is 0x1
IPCore_Timestamp	0x8	contains unique IP timestamp (yymmddHHMM): 2201201659
Blink_frequency_Data	0x100	data register for Inport Blink_frequency
Blink_direction_Data	0x104	data register for Inport Blink_direction
Read_back_Data	0x108	data register for Output Read_back

Following are the AXI4 slave Base address and Master address space specified in the reference design:

Default System.
 AXI4 Slave Base Address: **0x40000000**
 AXI4 Slave Master connection: **hdlverifier_axi_mana/axi4m**
 Use the AXI4 Slave Base Address plus Address offset to access the IP Core registers shown in Register Address Mapping table

The AXI4 slave write register readback is OFF for the IP core.
 The register address mapping is also in the following C header file for you to use when programming the processor:
[include\led_count_ip_addr.h](#)
 The IP core name is appended to the register names to avoid name conflicts.

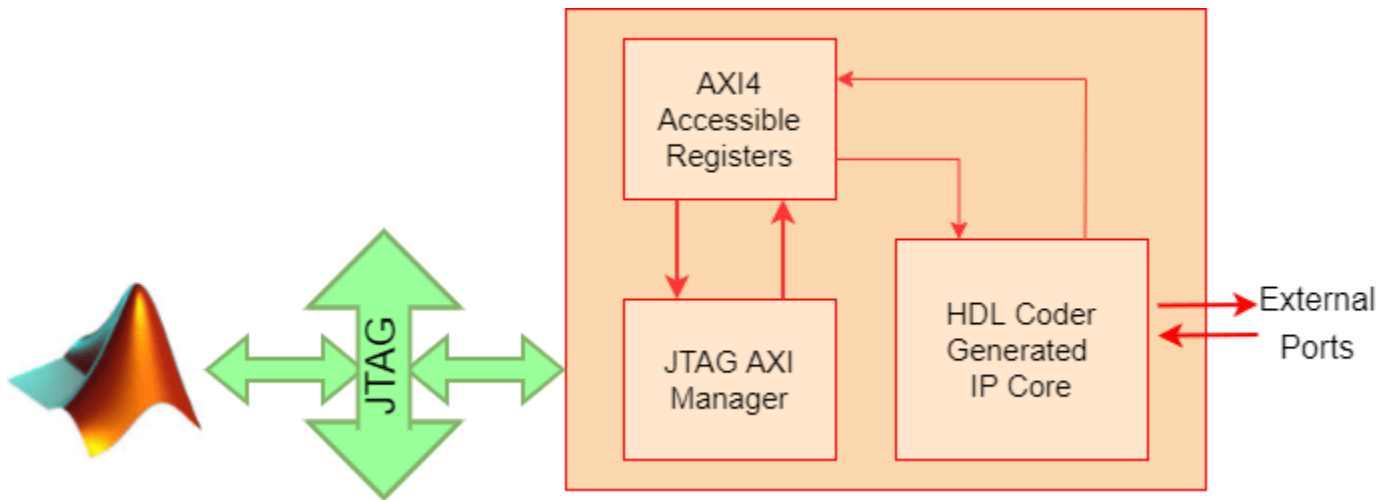
IP Core User Guide

Theory of Operation

At the bottom of the window are "OK" and "Help" buttons.

HDL Verifier Command Line Interface

If you have the HDL Verifier support package for Xilinx FPGA boards, select the AXI Manager reference design, then you can use MATLAB command line interface to access the IP core that is generated by the HDL Coder product.



To write and read from the DDR memory, follow these steps.

1. Create an AXI manager object.

```
h = aximanager('Xilinx')
```

2. Issue a write command. For example, disable the DUT.

```
h.writememory('40000004',0)
```

3. Re-enable the DUT.

```
h.writememory('40000004',1)
```

4. Issue a read command. For example, read the current counter value.

```
h.readmemory('40000108',1)
```

5. Delete the object to free up the JTAG resource. If you do not delete the object, other JTAG operations, such as programming the FPGA, fail.

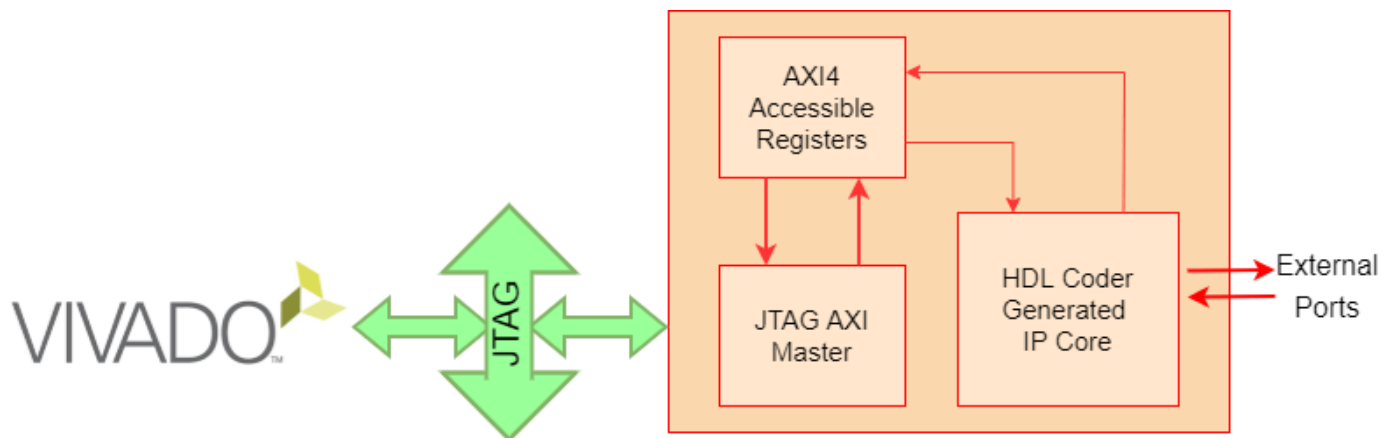
```
delete(h)
```

Xilinx JTAG to AXI Master Reference Design

Create a custom reference design to use the Xilinx JTAG to AXI Master IP in the reference design, and then add the reference design files to the MATLAB path using the `addpath` command.

Access the HDL Coder IP core registers using the Xilinx JTAG to AXI Master IP by using the base address that is defined in reference design plugin file.

Vivado Tcl Commands for AXI Read and Write



This example uses the standalone Vivado Tcl console for basic commands to issue reads and writes. You can use these commands to open the JTAG device and set up an "enable" and "disable" write to the DUT. You can enter these commands directly into the Vivado Tcl console or save them in a Tcl file and source them later. For simplicity, copy these Tcl commands into a file `open_jtag.tcl`.

```
# Open connection to the JTAG Master
open_hw
connect_hw_server
open_hw_target
refresh_hw_device [lindex [get_hw_devices] 0]

# Create some reads/writes
create_hw_axi_txn wr_enable [get_hw_axis hw_axi_1] ...
  -address 44a0_0004 -data 0000_0001 -type write
create_hw_axi_txn wr_disable [get_hw_axis hw_axi_1] ...
  -address 44a0_0004 -data 0000_0000 -type write
```

Launch the Vivado Tcl console, sourcing the file you just created.

```
system('vivado -mode tcl -source open_jtag.tcl&')
```

The screenshot shows a Windows command prompt window titled "C:\windows\system32\cmd.exe - vivado -mode tcl". The output of the Tcl script is displayed as follows:

```
WARNING: Default location for XILINX_VIVADO_HLS not found:
***** Vivado v2015.4 (64-bit)
***** SW Build 1412921 on Wed Nov 18 09:43:45 MST 2015
***** IP Build 1412160 on Tue Nov 17 13:47:24 MST 2015
***** Copyright 1986-2015 Xilinx, Inc. All Rights Reserved.
Vivado%
```

When you are done using the JTAG Master, close the connection by using these Tcl commands.

```
# Close and disconnect from the JTAG Master
close_hw_target;
disconnect_hw_server;
```

Summary

You can use the JTAG AXI Manager IP to interface with HDL Coder IP core registers in systems that do not have an embedded ARM processor, such as the Kintex-7. You can use this IP as a first step to debug standalone HDL Coder IP cores, prior to hand coding software for soft processors, (such as MicroBlaze), or as a way to tune parameters on a running system.

See Also

[aximanager](#) | [writememory](#) | [readmemory](#)

More About

- “Set Up AXI Manager” on page 3-2
- “Getting Started with the HDL Workflow Advisor” (HDL Coder)
- “Custom IP Core Generation” (HDL Coder)
- “Troubleshooting”

Verify OFDM Transmit and Receive using FPGA Data Capture

This example shows how to verify a wireless HDL IP that you generate in “OFDM Transmit and Receive Using Analog Devices AD9361/AD9364” (Communications Toolbox Support Package for Xilinx Zynq-Based Radio). The example also shows how to monitor and analyze the internal signals of an IP core using FPGA data capture on real hardware.

You use the **FPGA Data Capture** tool to capture the hardware signals over JTAG for debugging analysis. FPGA data capture offers many capabilities to capture signals of interest using appropriate trigger and capture conditions. You validate the behavior of the `whd\LOFDMRx Model` subsystem in the `zynqRadioHWSWOFDMAD9361AD9364SL` model by performing these steps.

- Use capture control to capture valid data constellation points.
- Use trigger conditions and multiple windows to capture header constellation points for frames with a failed CRC.
- Use trigger conditions and the trigger position to capture synchronizing sequence (SS) correlation data for the peak search and validate the timing offset.

Requirements

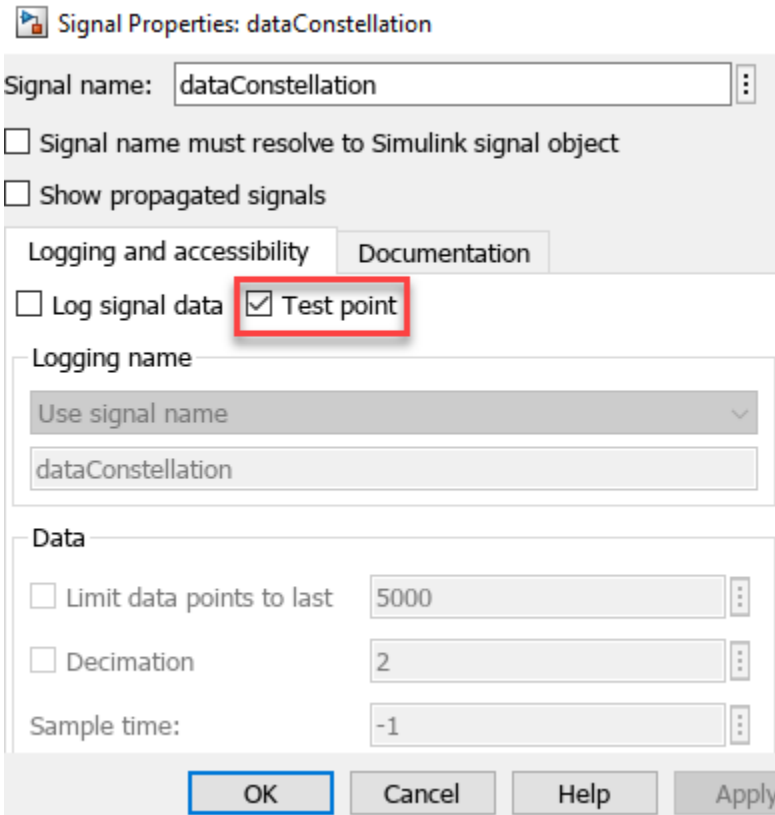
To run this example, you need a Xilinx Zynq ZC706 evaluation kit and FMCOMMS2/3/4 radio transmitter hardware.

Hardware Setup

- Follow the steps in “Guided Host-Radio Hardware Setup” (Communications Toolbox Support Package for Xilinx Zynq-Based Radio) to configure the host computer and ZC706 board with FMCOMMS2/3/4 radio hardware.
- Use the JTAG cable to connect the board to the host computer.

Simulink Model Setup

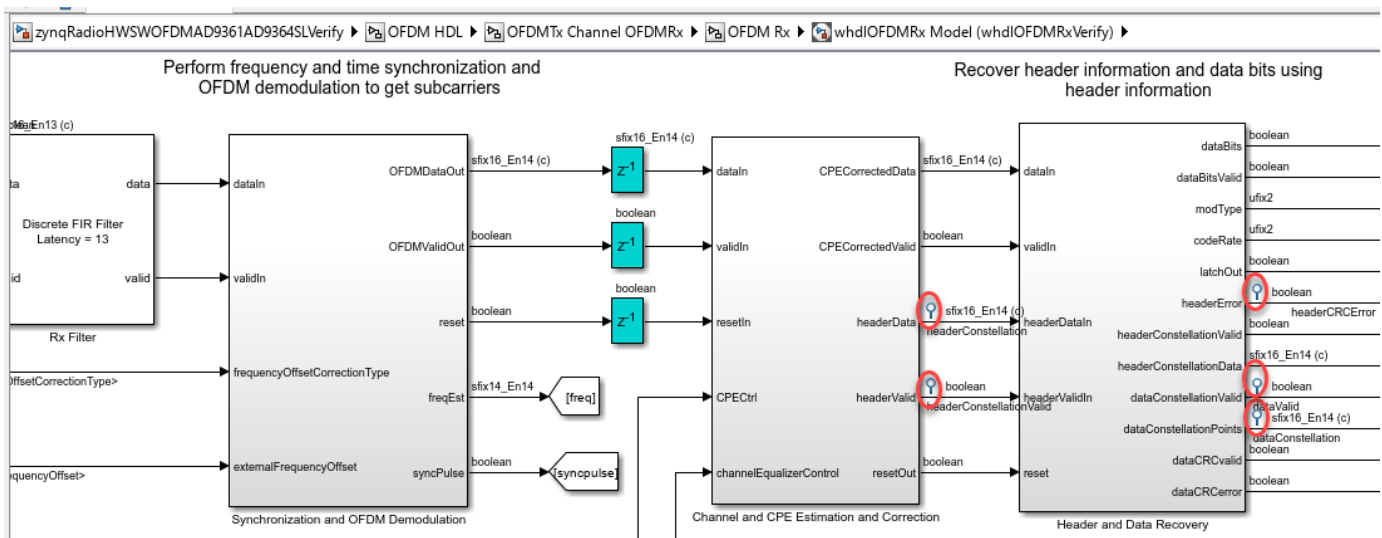
- 1 Open the `zynqRadioHWSWOFDMAD9361AD9364SL` model.
- 2 Mark the signals that you want to analyze through data capture as test points. To mark a signal as a test point, right-click the signal and then click **Properties**. On the **Signal Properties** dialog box, on the **Logging and accessibility** tab, select the **Test point** checkbox, as this figure shows.



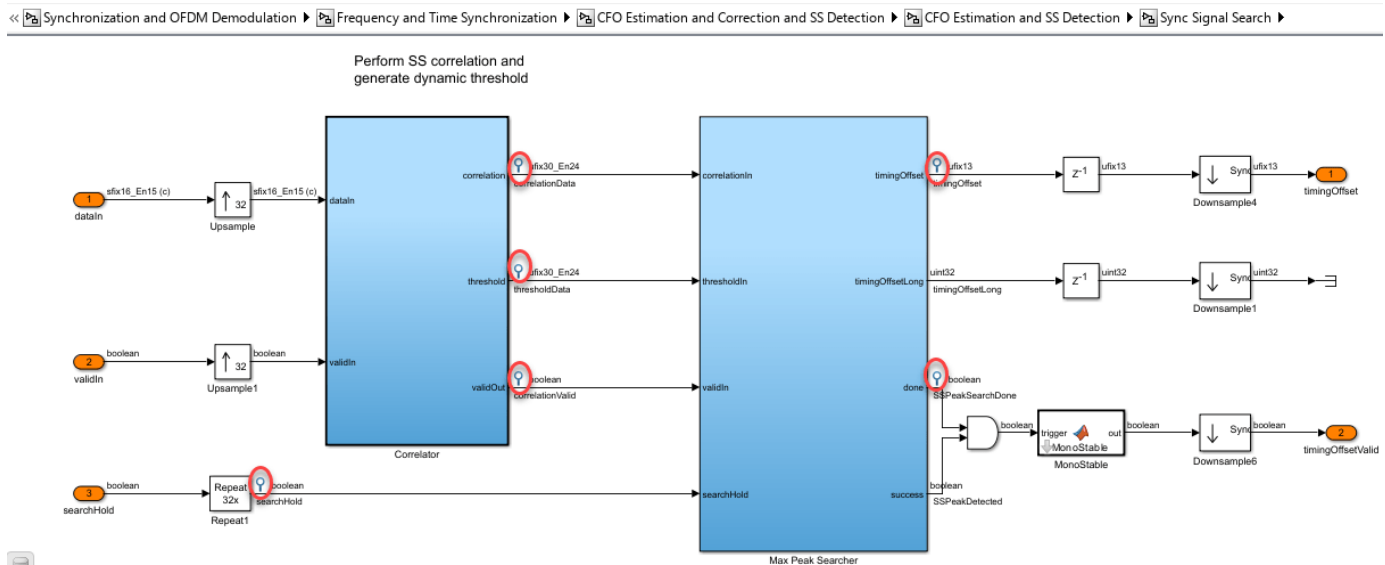
Alternatively, use the `hOFDMTxRxAddTestPoints` helper function to mark the required test points and save the model as `zynqRadioHWSWOFDMAD9361AD9364SLVerify`. The function is attached to the example as a supporting file.

hOFDMTxRxAddTestPoints

This figure shows the test points marked in the `whd1OFDMRx Model` subsystem for capturing the header and data constellation points.

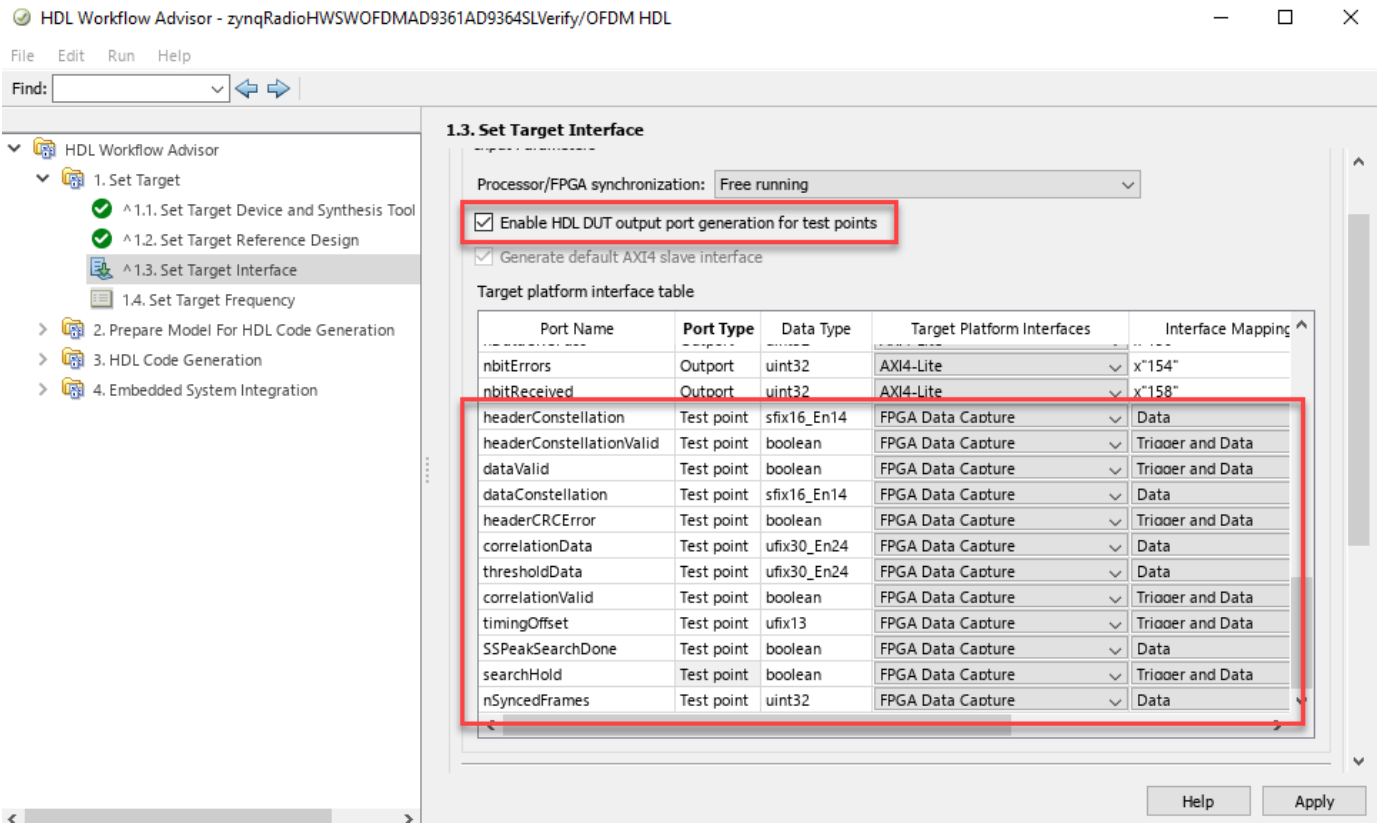


This figure shows the test points marked in the Sync Signal Search subsystem for capturing the SS correlation and timing offset signals.

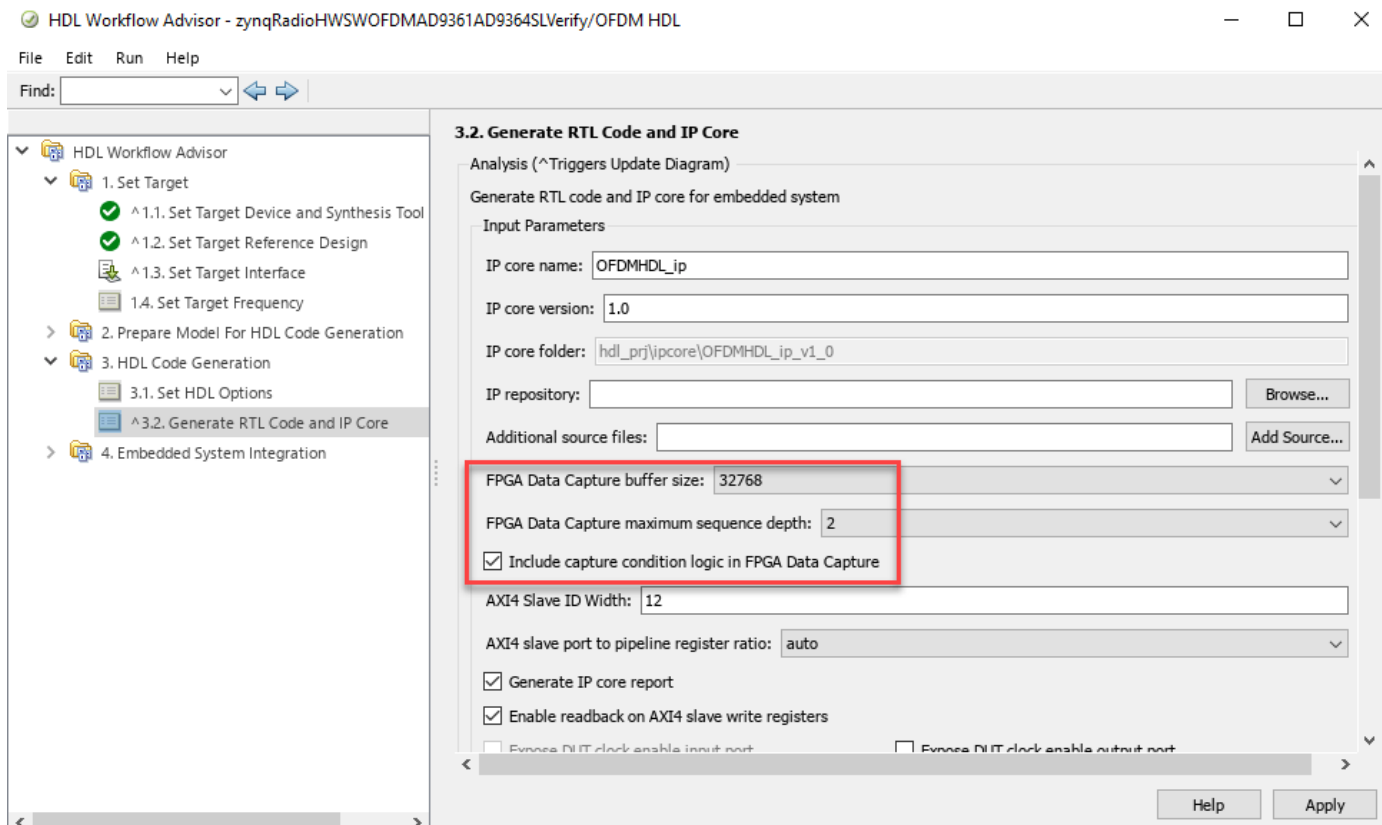


Generate HDL IP Core with Data Capture and Load Bitstream

- Start the targeting workflow by right-clicking the OFDM HDL subsystem and selecting **HDL Code > HDL Workflow Advisor**.
- In step 1.1, set **Target workflow** to IP Core Generation and **Target platform** to ZC706 and FMCOMMS2/3/4.
- In step 1.2, set **Reference design** to Receive and Transmit path. For this example, you can use the default values of the reference design parameters.
- In step 1.3, select **Enable HDL DUT output port generation for test points** to update the interface table with all the test points. Then, in the **Target platform interface table**, map the test point signals to FPGA Data Capture. For information about mapping IO ports, see “OFDM Transmit and Receive Using Analog Devices AD9361/AD9364” (Communications Toolbox Support Package for Xilinx Zynq-Based Radio).



- Run step 1.4 to step 3.1 by following the Generate IP Core section of the “OFDM Transmit and Receive Using Analog Devices AD9361/AD9364” (Communications Toolbox Support Package for Xilinx Zynq-Based Radio) example.
- In step 3.2, set **FPGA data capture buffer size** to 32768 and **FPGA data capture maximum sequence depth** to 2. Select **Include capture condition logic in FPGA Data Capture** to insert the capture control logic into the generated FPGA data capture component, as this figure shows.



- Run step 4.1 and follow the steps in the Generate Software Interface Model and Block Library section of the “OFDM Transmit and Receive Using Analog Devices AD9361/AD9364” (Communications Toolbox Support Package for Xilinx Zynq-Based Radio) example to generate a software interface model.
- To generate and download the bitstream onto the board, follow the Generate and Load Bitstream section of the “OFDM Transmit and Receive Using Analog Devices AD9361/AD9364” (Communications Toolbox Support Package for Xilinx Zynq-Based Radio) example.

Alternatively, run the `hOFDMTxRxRunHDLWATasks` helper function to run the tasks from step 1.1 to step 4.3. The function is attached to the example as a supporting file.

`hOFDMTxRxRunHDLWATasks`

Capture and Analyze Data from IP Core

Capture the test points of the generated IP core and map them to FPGA data capture. To run the software interface model while the FPGA data capture waits for a trigger, launch the **FPGA Data Capture** tool in nonblocking mode.

```
cd(fullfile('hdl_prj','hdlsrc', ...
    'zynqRadioHWSWOFDMAD9361AD9364SLVerify','fpga_data_capture'));
fdc = FPGADataCapture;
fdc.CaptureMode = 'nonBlocking';
fdc.launchApp;
```

Open the OFDM software interface model. Run the model in Monitor & Tune mode to control the configuration from the Simulink model.


```
open_system('zynqRadioHWSWOFDMAD9361AD9364SL_interface');
```

Capture Data Constellation Points

Use the frequency-domain, channel-equalized, and common-phase-error (CPE) corrected data subcarriers from the Channel and CPE Estimation and Correction subsystem to plot the data constellation diagram. To capture only valid data constellation points, select **Enable capture condition logic** on the **Capture Condition** tab of the **FPGA Data Capture** tool and add `tp_dataValid_1` as a signal with value High, as this figure shows. Click **Capture Data**.

The screenshot shows the 'FPGA Data Capture' tool window. The 'Output' section has 'dataCaptureOut' as the output variable name and 'Display data with Logic Analyzer' checked. The 'Capture Condition' tab is active, with 'Enable capture condition logic' checked. A table defines the capture condition:

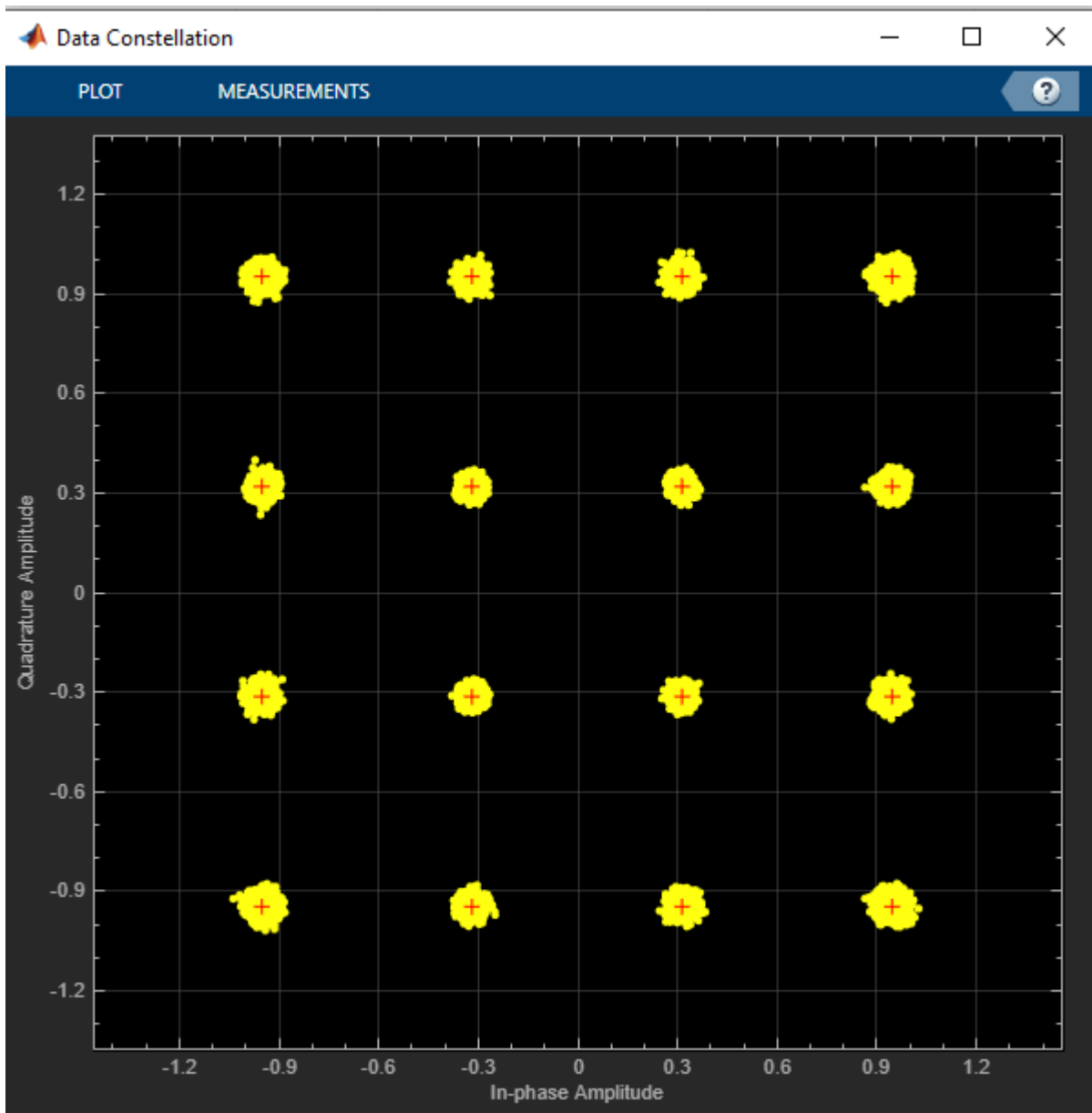
Signal	Operator	Value
tp_dataValid_1	==	High
tp_headerConstellation_1_re	+	

The status bar indicates 'Status: Not started' and a 'Capture Data' button is available.

After data capture is complete, plot the constellation diagram using the data that the board captures.

When `enableInternalLoopback` in the software interface model is `false`, the radio hardware transmits and receives the OFDM signals through the Tx and Rx antennas, respectively. The channel impairments modify the signal as it travels over the air in real time. In this scenario, use the constellation plot to analyze the received signal quality. You can also modify `modType` in the software interface model and verify the data constellation diagram appears as you expect.

h0FDMTxRxPlotDataConstellationFDC



Capture Header Constellation Points for Failed CRC

When the signal-to-noise ratio (SNR) of the channel is low, the valid header data after channel equalization and CPE correction often has CRC failures. You can emulate these conditions using the software interface model.

In the software interface model, set `enableInternalLoopback` to `true` and `snrdb` to 3. Use these trigger settings to capture the I/Q data of two header frames when the CRC fails.

FPGA Data Capture

Generate data capture IP → Integrate with existing FPGA design → Capture data

[Read more about the data capture workflow](#)

Output

Output variable name: dataCaptureOut Display data with Logic Analyzer

Trigger | Capture Condition | Data Types

Sample depth: 32768

Number of capture windows: 2 | Number of trigger stages: 1

Trigger position: 16383

Trigger Stage 1

Signal	Operator	Value
tp_headerCRCError_1	==	High
tp_headerConstellation_1_im	+	

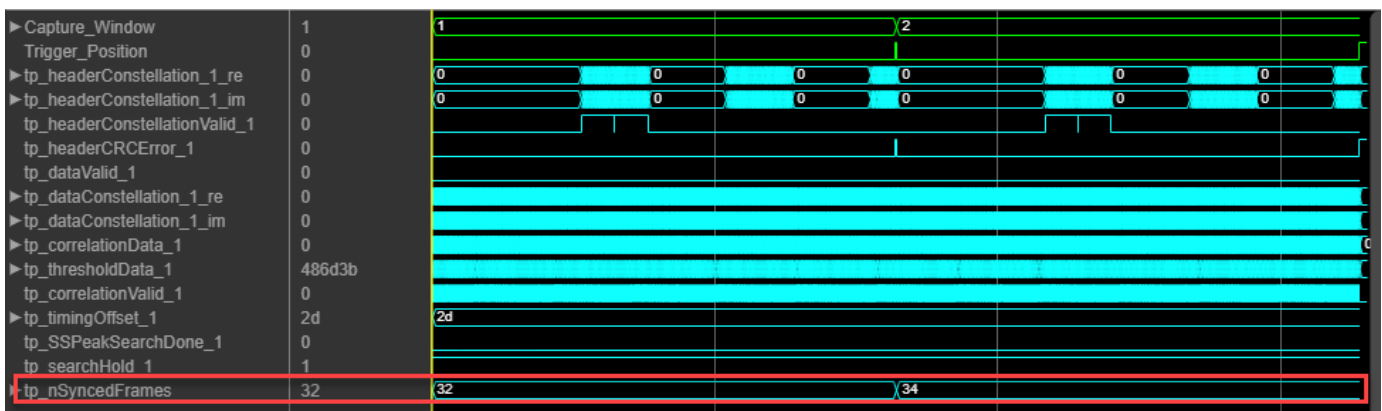
Change operator

AND

Status: Successfully captured 2 windows data from FPGA

On trigger | Capture Data

Click **Capture Data**. If you have a DSP System Toolbox™ license, the **Logic Analyzer** app plots the captured data as multiple signal waveforms. The signal `tp_nSyncedFrames` in the captured data indicates the number of the frame with a failed CRC.



Validate Functionality of Sync Signal Search Subsystem

The Sync Signal Search subsystem implements the SS correlation. The subsystem performs SS detection by continuously cross-correlating the received signal with the SS signal in the time domain. In addition, the subsystem computes the energy of the signal in the span of the correlator at each time step and then scales the value to generate a threshold value. The Max Peak Searcher subsystem searches for the maximum correlation peak in each OFDM frame duration. In the Sync Signal Search subsystem, the SearchHold signal disables maximum peak search until it estimates the carrier frequency offset. The SSPeakSearchDone signal goes high after the completion of peak search in each OFDM frame duration. For an FFT Length of 128, the frame duration is 5760 samples. The position of the maximum peak from the start of the frame duration gives the timing offset. You can capture valid samples of correlation in each OFDM frame duration to visualize the correlation data and record the timing offset.

On the **Trigger** tab, add `tp_searchHold_1` as a signal with the value High in **Trigger Stage 1**. To capture correlation data for each frame duration, add `tp_SSPeakSearchDone_1` as a signal with the value High in **Trigger Stage 2** and set **Trigger position** to 5759. These settings ensure that the tool captures 5760 valid correlation samples before the signals satisfy these triggers. Samples 5761 to 11520 correspond to correlation of the next OFDM frame duration, and so on.

The screenshot shows the FPGA Data Capture tool interface. The top section displays the sample depth (32768), the number of capture windows (1), and the number of trigger stages (2). The trigger position is set to 5759. Below this, the trigger configuration is shown for two stages:

Trigger Stage 1

Signal	Operator	Value
<code>tp_searchHold_1</code>	<code>==</code>	High

Trigger Stage 2

Signal	Operator	Value
<code>tp_SSPeakSearchDone_1</code>	<code>==</code>	High

The status bar at the bottom indicates: "Status: Successfully captured 1 windows data from FPGA". The "On trigger" dropdown is set to "On trigger", and the "Capture Data" button is visible.

Select **Enable capture condition logic** on the **Capture Condition** tab to capture valid correlation data in each frame duration. Add the `tp_correlationValid_1` as a signal with the value High, as this figure shows.

FPGA Data Capture

dataCaptureOut Display data with Logic Analyzer

Trigger Capture Condition Data Types

Enable capture condition logic

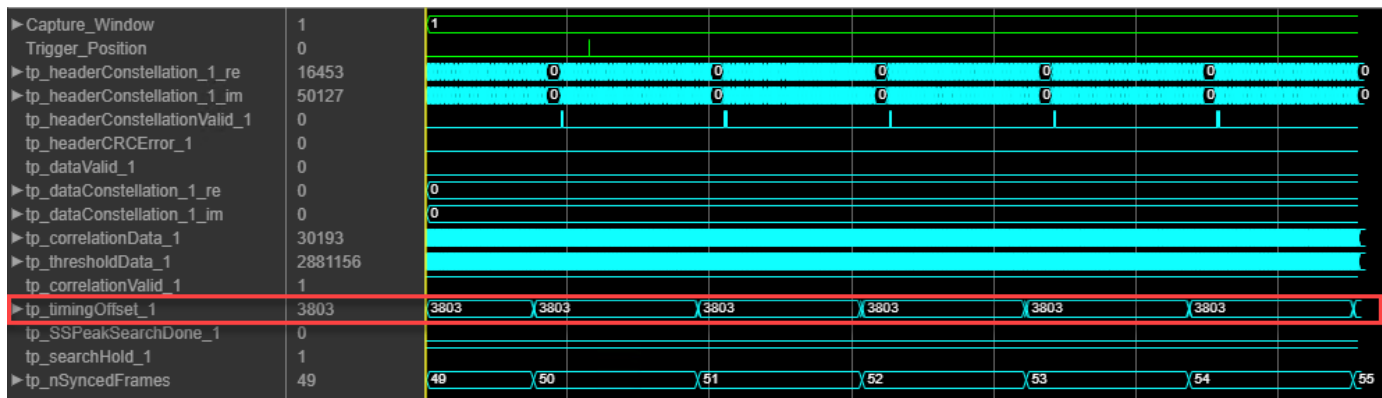
Signal	Operator	Value
tp_correlationValid_1	==	High
tp_headerConstellation_1_re	+	

Change operator

AND

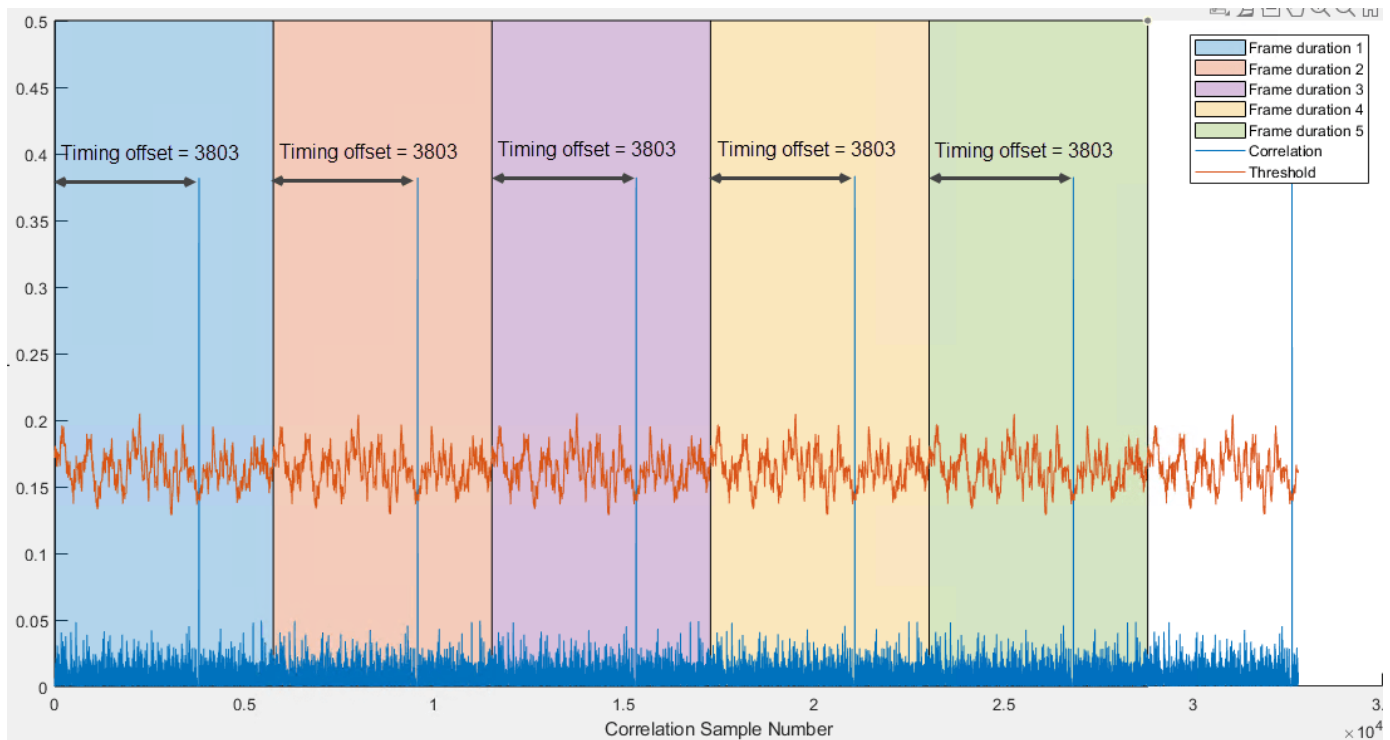
Status: Not started On trigger Capture Data

Click **Capture Data**. The **Logic Analyzer** displays the captured data as multiple signal waveforms.



Run the `hOFDMTxRxPlotSSCorrelation` helper function to visualize the peak by plotting the captured correlation and threshold values. The index of the peak in each frame relative to start of the frame duration gives the timing offset.

```
hOFDMTxRxPlotSSCorrelation
```



Conclusion

This example shows how to map the internal signals of an IP core to FPGA data capture and then visualize the signals after you deploy the design to an FPGA. You use the trigger and capture control configuration of the **FPGA Data Capture** tool for capturing the signals of interest. You can use this approach to analyze and debug your own HDL IP core.

See Also

FPGA Data Capture | `hdlverifier.FPGADataReader`

Related Examples

- “OFDM Transmit and Receive Using Analog Devices AD9361/AD9364” (Communications Toolbox Support Package for Xilinx Zynq-Based Radio)

More About

- “Data Capture Workflow” on page 6-2
- “Troubleshooting”