KCU105 PCI Express Memory-Mapped Data Plane TRD User Guide

KUCon-TRD02

UG919 (Vivado Design Suite v2015.2) June 30, 2015

XILINX
ALL PROGRAMMABLE™

Revision History

The following table shows the revision history for this document.

Date	Version	Revision
06/30/2015	2015.2	Released with Vivado Design Suite 2015.2 without changes from the previous version.
05/04/2015	2015.1	Updated for Vivado Design Suite 2015.1. Updated information about resource utilization for the base and the user extension design. Added information about Windows 7 driver support of the reference design. Deleted section on QuestaSim simulation and added support for Vivado Simulator. Added Appendix E, APIs Provided by the XDMA Driver in Windows and Appendix F, Recommended Practices and Troubleshooting in Windows.
02/26/2015	2014.4.1	Initial Xilinx release.

Table of Contents

Introduction

This document describes the features and functions of the PCI Express® (PCIe®) memory-mapped data plane targeted reference design (TRD). The TRD comprises two designs - a base design and a user extension design. The user extension design is an extension of the base design which shows how to add custom logic on top of the base design and achieve the design requirement. In the pre-built user extension design, a video image processing algorithm is presented. This document also describes how to set up and test the base and the user extension designs.

PCIe Memory-Mapped Data Plane TRD Overview

The PCIe memory-mapped data plane TRD targets the Kintex® UltraScale™ XCKU040-2FFVA1156E FPGA running on the KCU105 evaluation board.

The PCIe memory-mapped data plane TRD provides a platform for high speed data transfer between host system memory and DDR4 memory on the KCU105 board through the KU040 FPGA.

Figure 1-1 shows the top-level block diagram of the PCIe memory-mapped data plane base design.

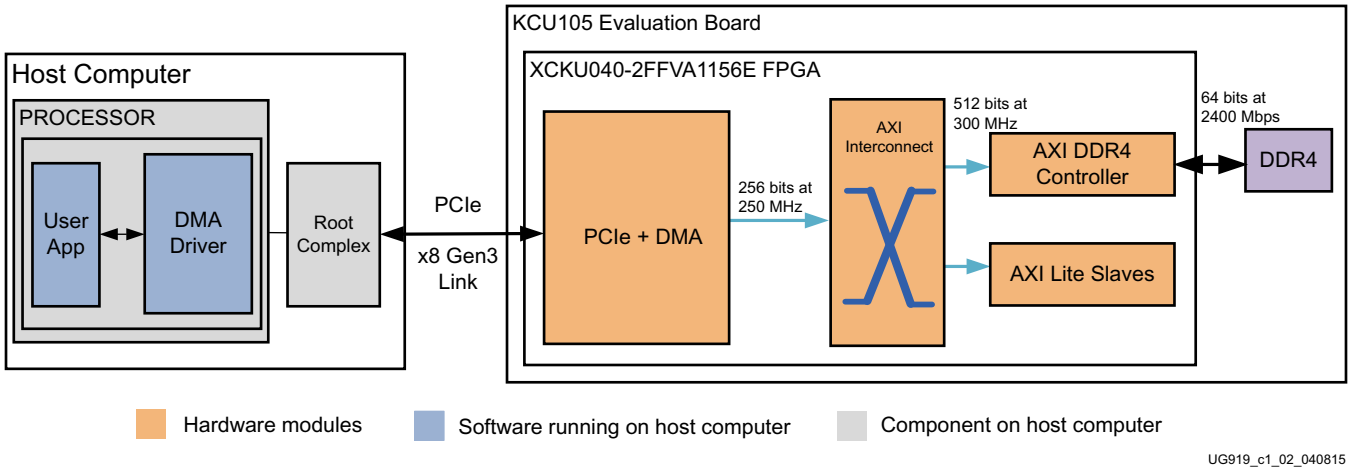


Figure 1-1: **KCU105 PCI Express Memory-Mapped Data Plane Base Design**

This TRD uses an integrated Endpoint block for PCIe in x8 Gen3 configuration along with high performance Expresso DMA from Northwest Logic for data transfers between host system memory and endpoint memory (DDR4 memory on KCU105 board).

The AXI Bridge from Northwest Logic provides protocol conversion between PCIe TLPs and AXI transactions. The DDR4 memory controller is provided through MIG IP.

The downstream AXI Lite slaves include a power monitor module, user space registers, and an AXI performance monitor.

In system-to-card (S2C) direction, the DMA block moves data from the host memory to the FPGA through the PCIe integrated Endpoint block and then writes the data into the DDR4 through the DDR4 controller. In card to system (C2S) direction, DMA reads the data from DDR4 and writes to host system memory through the PCIe integrated Endpoint block.

The base design provides an AXI memory-mapped interface which makes it easy for you to add custom logic to the design. This base platform can be extended to variety of applications such as video processing and Ethernet packet processing.

The user extension design delivered with this TRD (shown in Figure 1-2) provides an example of off-loading a video image processing algorithm like edge detection through a Sobel filter to the FPGA. Video frames (full high definition) are transferred from the host system to the card memory (DDR4). AXI Video DMA reads the video frames from DDR4 and presents them to the Sobel filter. The processed video frames are written back to DDR4 by AXI VDMA. The DMA engine reads the processed video frames from DDR4 and writes to host system memory over the PCIe integrated Endpoint block.

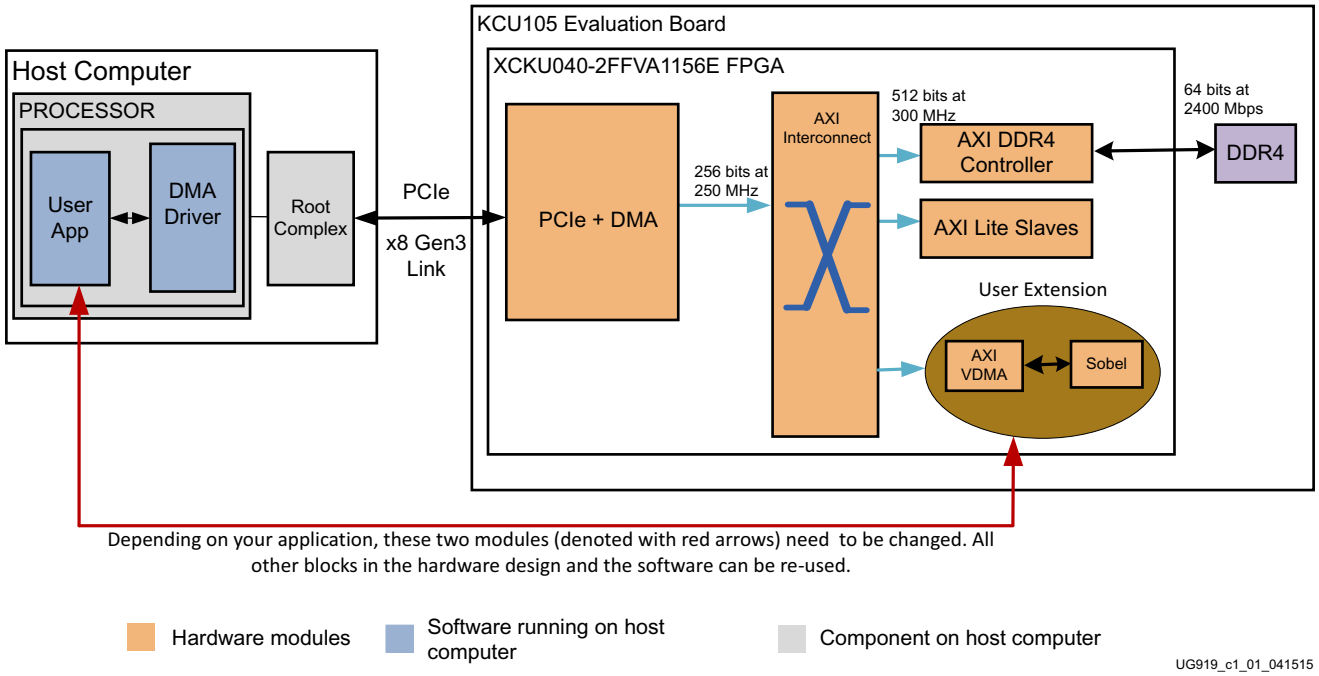


Figure 1-2: **KCU105 PCI Express Memory-Mapped Data Plane User Extension Design**

The designs delivered as part of this TRD use the Vivado® IP Integrator to build the system. IP Integrator provides intelligent IP integration in a graphical, Tcl based, correct-by-construction IP, and system-centric design development flow. For further details

see the *Vivado Design Suite Tutorial: Designing IP Subsystems Using IP Integrator* (UG995) [Ref 1].

Note: See Chapter 5, Targeted Reference Design Details and Modifications for detailed descriptions of the hardware and software designs.

Components, Features, and Functions

The PCIe memory mapped data plane reference design includes:

- Hardware design:
  - An 8-lane integrated integrated Endpoint block for PCIe:
    - Each lane operates at 8 GT/s per lane, per direction
    - 256-bit @ 250 MHz
  - An AXI-PCIe Bridge IP with Expresso DMA from Northwest Logic:
    - Scatter Gather enabled
    - Two channels: One System-to-Card (S2C) and one Card-to-System (C2S)
    - Support for AXI3 interface
    - Two ingress and two egress translation region support

      Note: The IP is capable of supporting a higher number of translation regions, the netlist used here is built to support only two such regions. Contact Northwest Logic for further customizing of IP.
  - DDR4 operating four components:
    - Each 16-bit@ 2,400 Mb/s
    - MIG IP with AXI interface operating 512 bits @ 300 MHz
  - SYSMON-based power monitor
    - Pre-packaged reusable IP for power and die temperature monitoring using the SYSMON block
- Software Design:
  - 64-bit Linux kernel space drivers for DMA and a raw data driver
  - 64-bit Windows 7 drivers for DMA and a raw data driver
  - User space application
  - Control and monitoring graphical user interface (GUI)

Resource Utilization

Table 1-1 and Table 1-2 list the resources used by the PCIe memory-mapped data plane base and pre-built user extension designs, respectively, after synthesis. Place and route can alter these numbers based on placements and routing paths, so use these numbers as a rough estimate of resource utilization. These numbers might vary based on the version of the PCIe memory-mapped data plane and the tools and implementation settings used to regenerate the design.

Table 1-1: **Resource Utilization of Base Design**

Resource Type	Used	Available	Usage (%)
CLB registers	128,079	484,800	26.42
CLB LUTs	84,229	242,400	34.75
Block RAM	64	600	10.67
MMCME3_ADV	2	10	20
Global clock buffers	10	240	4.17
BUFG_GT	5	120	4.17
SYSMONE1	1	1	100
IOB	136	520	26.15
GTHE3_CHANNEL	8	20	40
GTHE3_COMMON	2	5	40

Table 1-2: **Resource Utilization of Pre-built User Extension Design**

Resource Type	Used	Available	Usage (%)
CLB Registers	162,226	484,800	33.46
CLB LUTs	101,865	242,400	42.02
Block RAM	102	600	17
MMCME3_ADV	2	10	20
Global Clock Buffers	10	240	4.17
BUFG_GT	5	120	4.17
SYSMONE1	1	1	100
IOB	136	520	26.15
GTHE3_CHANNEL	8	20	40
GTHE3_COMMON	2	5	40

Setup

This chapter lists the requirements and describes how to do all preliminary setup of the KCU105 board, control computer, and software before bringing up the PCIe memory-mapped data plane reference design.

IMPORTANT: *Perform the procedures described in this chapter before performing the bring up procedures described in Chapter 3, Bringing Up the Design.*

Requirements

Hardware

The hardware listed here is included in the KCU105 Connectivity Kit.

- KCU105 board with the Kintex® UltraScale™ XCKU040-2FFVA1156E FPGA
- USB cable, standard-A plug to micro-B plug (Digilent cable)
- Power Supply: 100 VAC–240 VAC input, 12 VDC 5.0A output
- ATX Power supply
- ATX Power supply adapter

Computers

A control computer is required for running the Vivado® Design Suite and configuring the FPGA. It can be a laptop or desktop computer with a Vivado supported operating system such as Redhat Linux or Microsoft Windows 7.

The reference design test configuration requires a host computer comprised of the chassis, a motherboard with a PCI Express® slot, a monitor, keyboard, and mouse.

- For Linux: A DVD drive is also required.
- For Windows: The 64-bit Windows 7 OS and the Java SE Development Kit 7 must be installed.

Design Tools and Software

- Vivado Design Suite 2015.2: Logic Edition (full seat, device-locked to the XCKU040-2FFVA1156E FPGA)
- Fedora 20 Live DVD media (this is the Linux OS with the reference design driver and GUI files support, built by adding additionally required packages to the Fedora 20 kernel available online)

Note: The Fedora 20 Live DVD is only required if you are using the Linux flow.

Download and installation instructions for each required software is described in Preliminary Setup.

Preliminary Setup

Complete these tasks before bringing up the design described in Chapter 3, Bringing Up the Design.

Install Vivado Design Suite

Install Vivado Design Suite 2015.2 on the control computer (laptop). Follow the installation instructions provided in *Vivado Design Suite User Guide Release Notes, Installation, and Licensing* (UG973) [Ref 2].

Download Targeted Reference Design Files

1. Download the PCIe memory-mapped data plane reference design ZIP file (`rdf0306-kcu105-trd02-2015-2.zip`).
2. Unzip the contents of the file to a working directory.
3. The unzipped contents will be located at `<working_dir>/kcu105_aximm_dataplane`.

The PCIe memory-mapped data plane reference design TRD directory structure is described in Appendix A, Directory Structure.

Setup and Driver Installation on Windows 7 Host Computer

Note: If you are using the Linux flow, skip this section and proceed to Set DIP Switches on the KCU105 Board, page 15.

This section includes steps to setup and install KUCon-TRD drivers in a host computer running the Windows 7 64-bit OS.

Disable Driver Signature Enforcement

Note: Windows allows only drivers with valid signatures obtained from trusted certificate authorities to load in Windows 7 64 bit OS. Windows drivers provided for this reference design do not have a valid signature. Therefore, you have to perform disable driver signature enforcement on the host computer, as follows:

1. Power up the host system. Press **F8** to go to the *Advanced Boot Options* menu.
2. Select the **Disable Driver Signature Enforcement** option as shown in Figure 2-1, and press **Enter**.

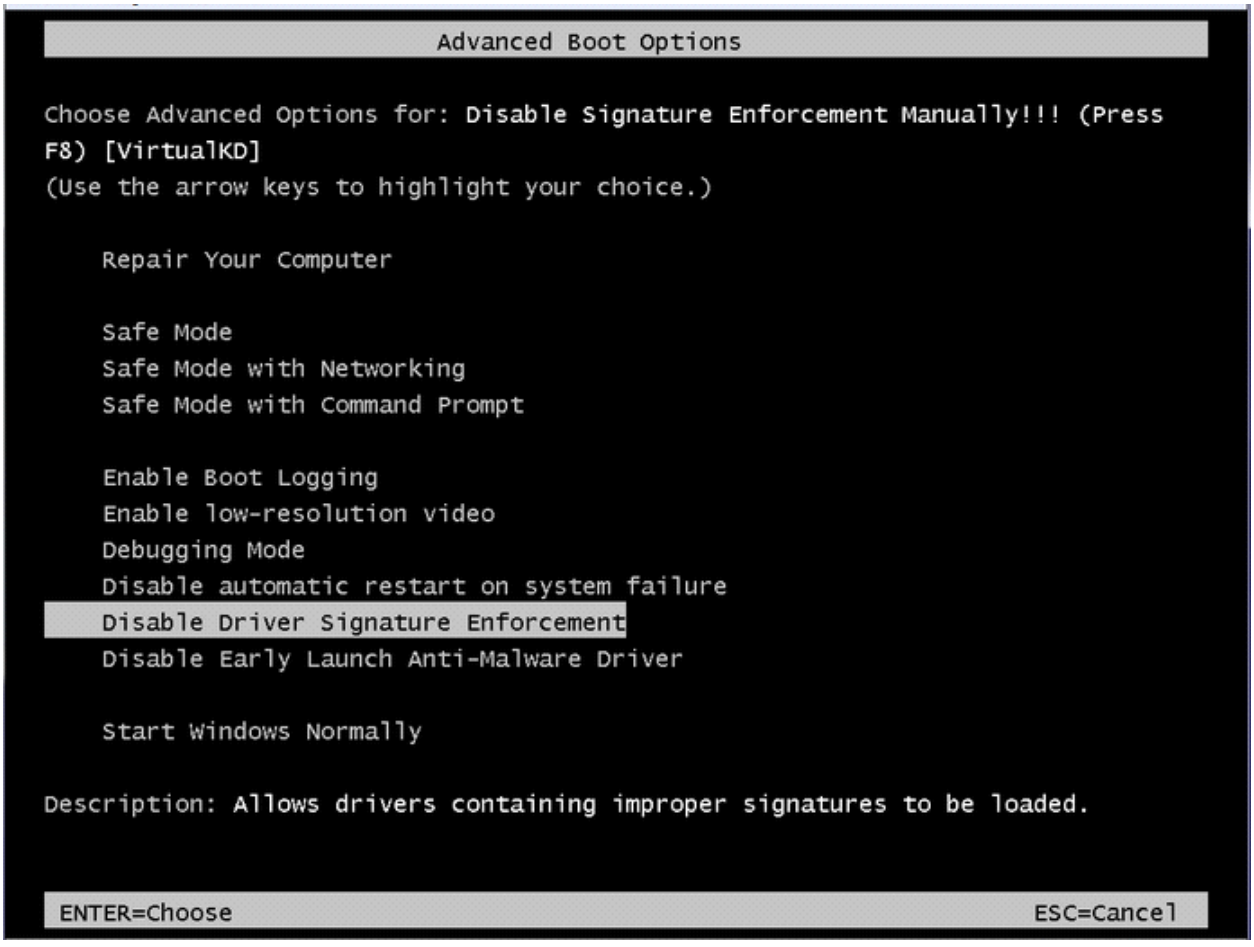


Figure 2-1: **Disable Driver Signature Enforcement**

Installing Drivers

1. From the Windows explorer, navigate to the folder in which the reference design is downloaded (`<dir>\kcu105_aximm_dataplane\software\windows\`) and run the setup file with Administrator privileges, as shown in Figure 2-2.

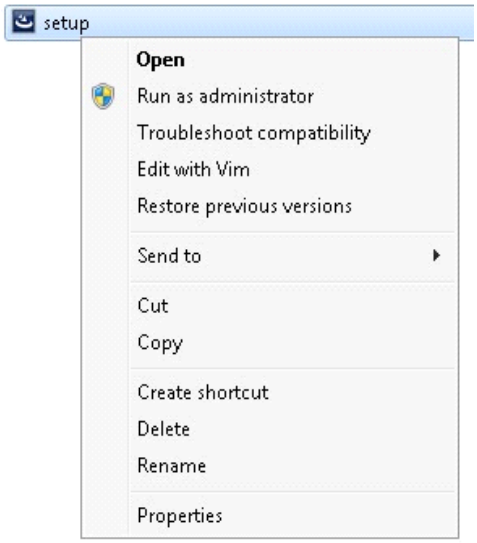


Figure 2-2: **Run Setup File to Install Drivers**

2. Click **Next** after the Install Shield Wizard opens, as shown in Figure 2-3.

Figure 2-3: **InstallShield Wizard**

3. Click **Next** to install to the default folder or click **Change** to install to a different folder (shown in Figure 2-4).

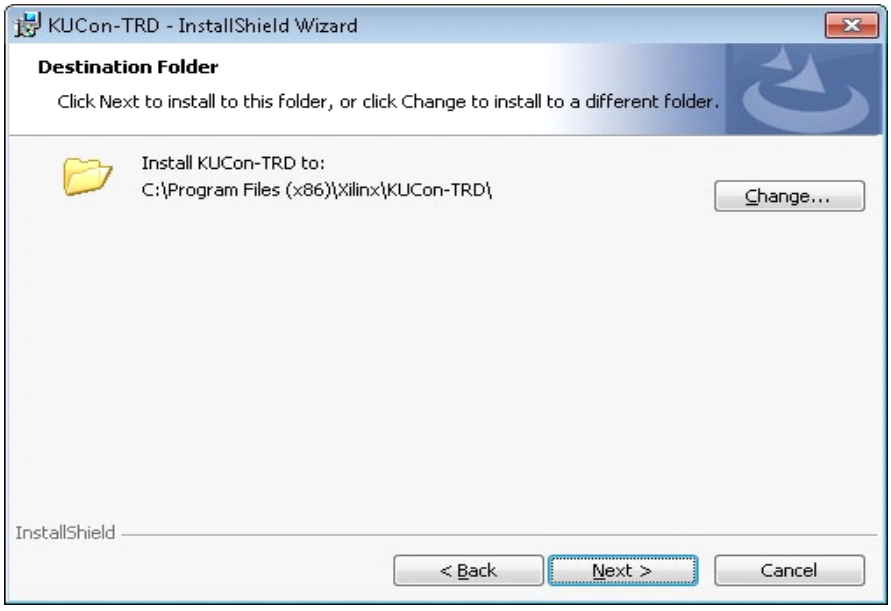


Figure 2-4: **InstallShield - Destination Folder**

4. Click Install to begin driver installation (see Figure 2-5).

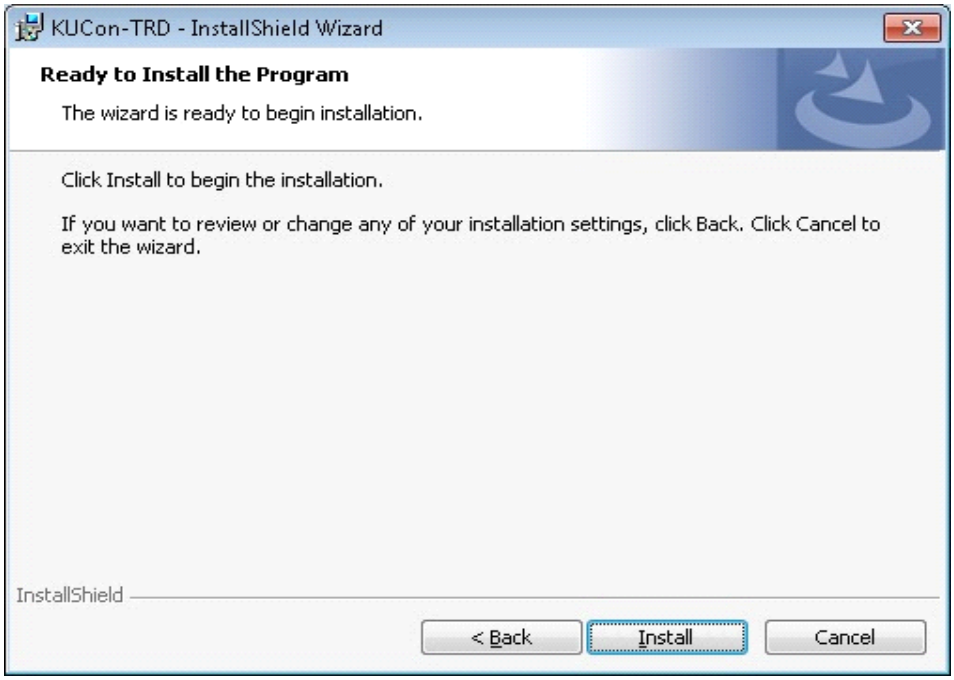


Figure 2-5: **Install Drivers**

5. A warning screen is displayed while installing the drivers, as they are not signed by a trusted certificate authority yet. To install the drivers, ignore the warning message shown in Figure 2-6 and click **Install this driver software anyway**. This warning message pops up two times so repeat this step accordingly.

Figure 2-6: **Ignore Windows Security Alert**

6. After installation is complete, click **Finish** to exit the InstallShield Wizard, as shown in Figure 2-7.

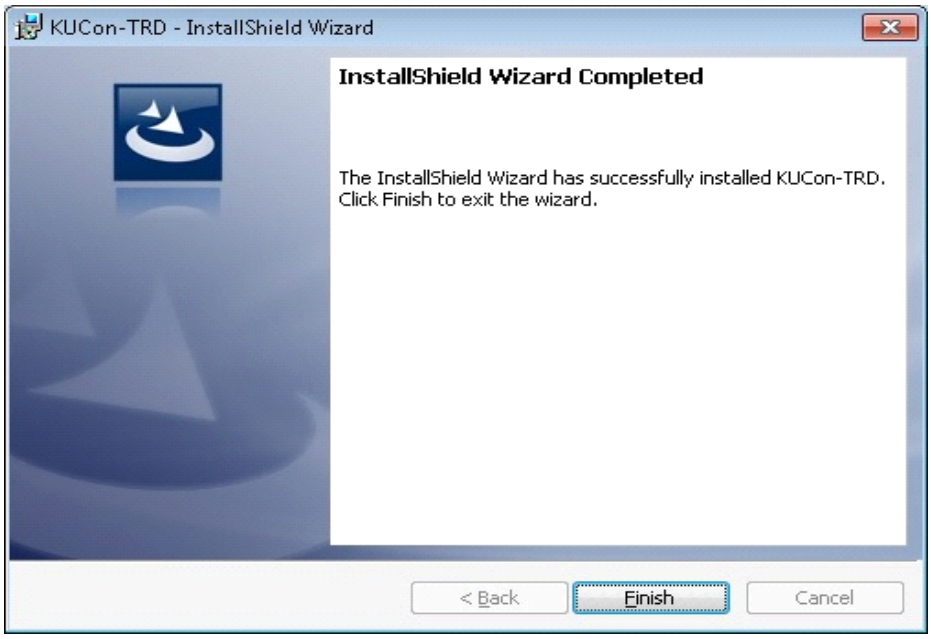


Figure 2-7: **Finish InstallShield Wizard**

Set DIP Switches on the KCU105 Board

Set the DIP switches and jumpers on the KCU105 board to the factory default settings as described in the *Kintex UltraScale FPGA KCU105 Evaluation Board User Guide* (UG917) [Ref 3].

Ready to Bring Up the Design

After all procedures in this chapter are complete, go to Chapter 3, Bringing Up the Design.

Bringing Up the Design

This chapter describes how to bring up the PCIe memory-mapped data plane.

IMPORTANT: *Perform the preliminary setup procedures described in Chapter 2, Setup before performing the bring up procedures described in this chapter.*

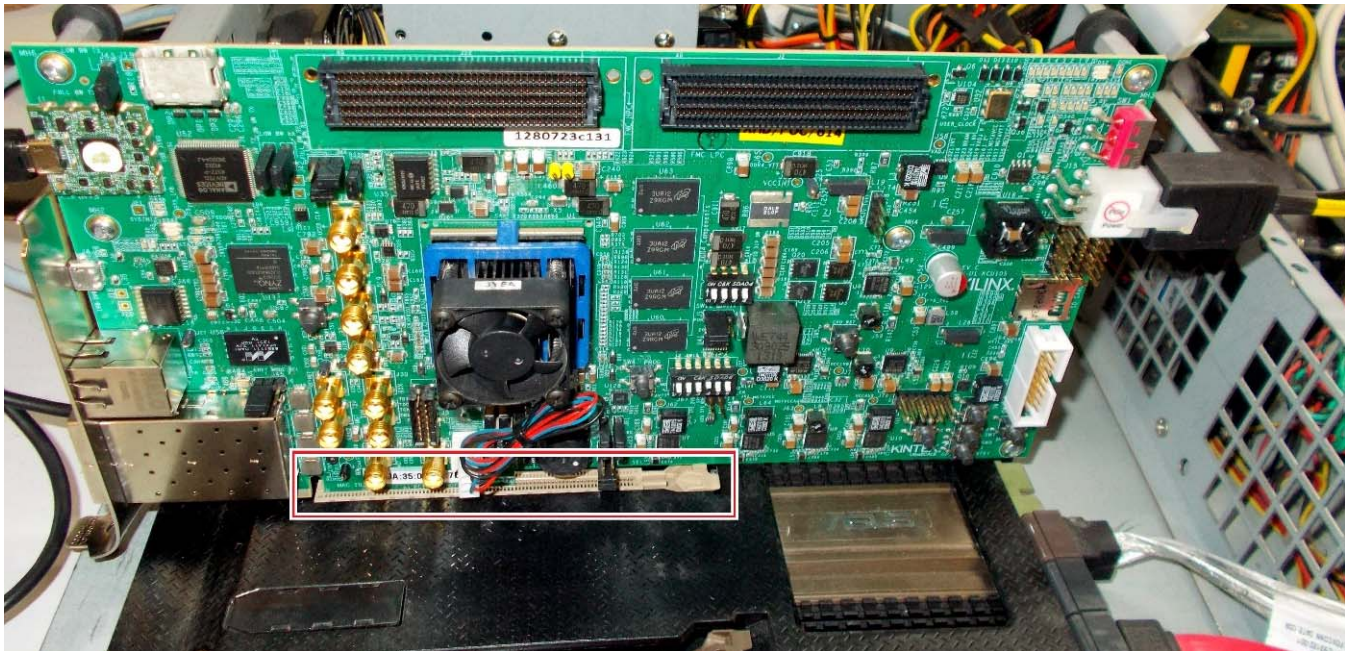
Install the KCU105 Board into the Host Chassis

1. Remove all rubber feet and standoffs from the KCU105 board
2. Power down the host chassis and disconnect the power cord.

CAUTION! *The power cord must be removed to prevent electrical shock or damage to the KCU105 board or other components.*

3. Ensure that the host computer is powered off.
4. Open the host chassis and select a vacant PCIe Gen3-capable expansion slot and remove the expansion cover at the back of the host chassis.

5. Plug the KCU105 board into the PCIe connector on this slot as shown in Figure 3-1.



UG918_c3_01_091314

Figure 3-1: **KCU105 Board Installed into a PCIe x8 Connector**

6. Connect the ATX power supply to the KCU105 board using the ATX power supply adapter cable as shown in Figure 3-2.

   Note: A 100 VAC–240 VAC input, 12 VDC 5.0A output external power supply can be substituted for the ATX power supply.

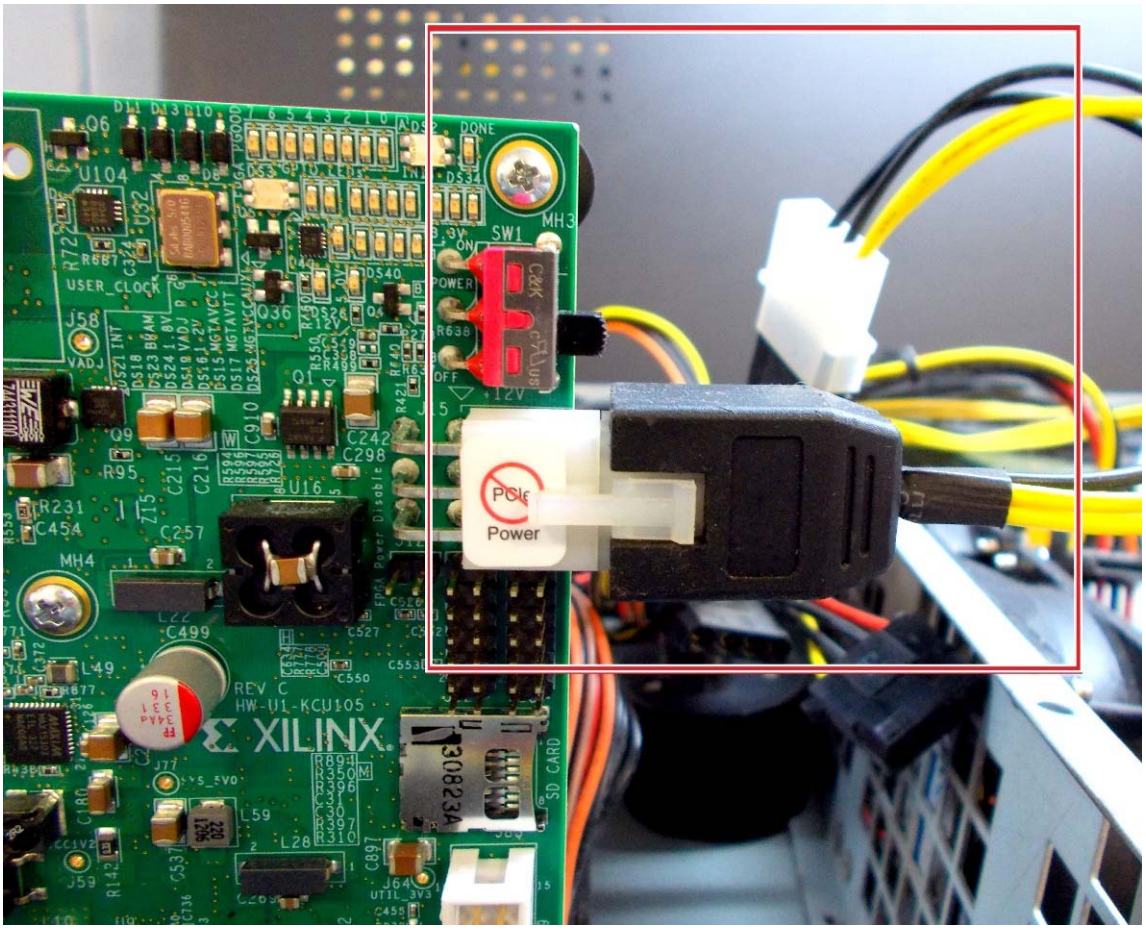


Figure 3-2: **Power Supply Connection to KCU105 Board**

6. Slide KCU105 board power switch SW1 to the **ON** position.

Set the Host System to Boot from the Live DVD

This section applies only to the Linux Flow. Proceed with Configure the FPGA if you are using Windows.

1. Power on the host system and stop it in BIOS to select options to boot from the DVD Drive. BIOS options are entered by depressing **DEL** or **F12** or **F2** keys on most computers.

   Note: If an external power supply is used instead of the ATX power supply, the FPGA can be configured first. Then power on the host system.

2. Place the Fedora 20 live DVD into the host system DVD drive.

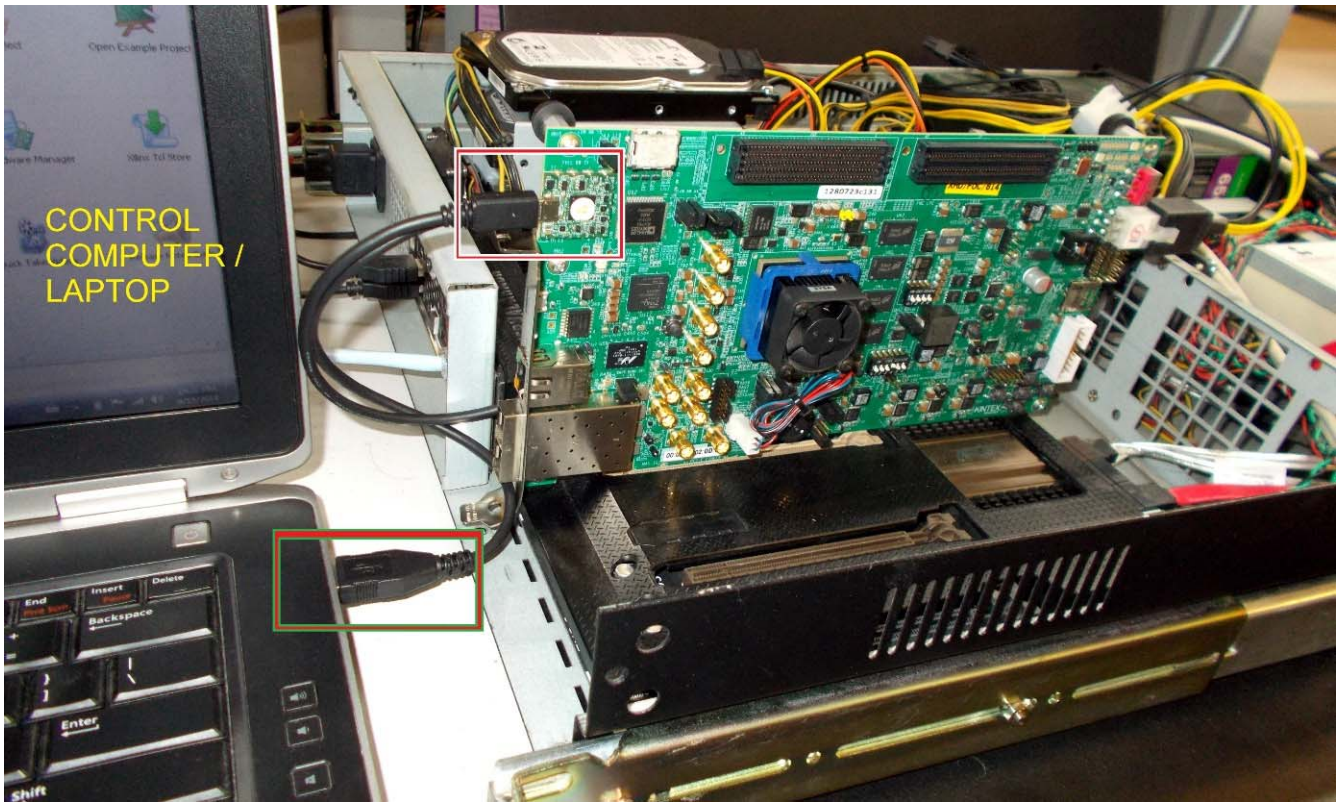
3. Select the option to boot from DVD.

Configure the FPGA

While in BIOS, program the FPGA with the bitfile:

1. Connect the standard-A plug to micro-B plug USB cable to the JTAG port of the KCU105 board and control PC as shown in Figure 3-3.

   Note: Figure 3-3 shows a Rev C board. The USB JTAG connector is on the PCIe panel for production boards.



UG918_c3_03_091314

Figure 3-3: **Connect the USB Cable to the KCU105 Board and Control Computer**

2. Launch the Vivado Integrated Design Environment (IDE) on the control computer:
   a. Select **Start > All Programs > Xilinx Design Tools > Vivado 2015.2 > Vivado 2015.2**.
   b. On the getting started page, click **Open Hardware Manager** (Figure 3-4).

Figure 3-4: **Vivado IDE Getting Started Page, Open Hardware Manager**

3. Open the connection wizard to initiate a connection to the KCU105 board:
   a. Click **Open a new hardware target** (Figure 3-5).

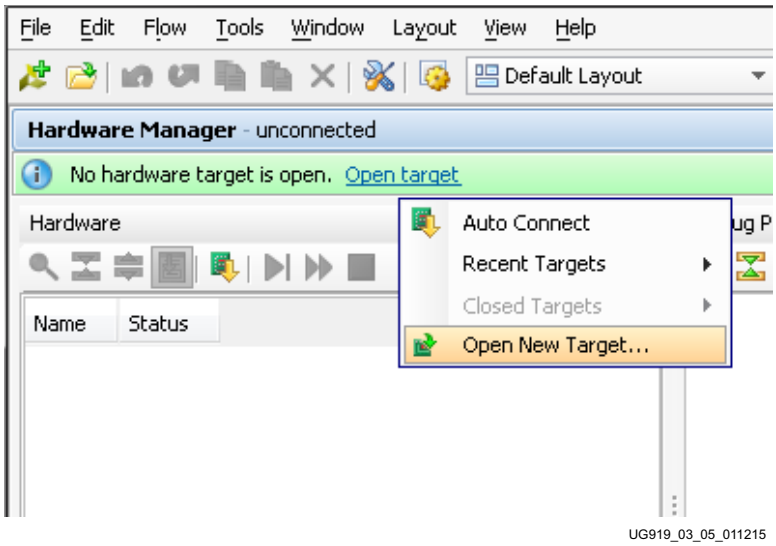


Figure 3-5: **Using the User Assistance Bar to Open a Hardware Target**

4. Configure the wizard to establish connection with the KCU105 board by selecting the default value on each wizard page. Click **Next > Next > Next > Finish**.

   a. In the hardware view, right-click **xcku040** and select **Program Device** (Figure 3-6).

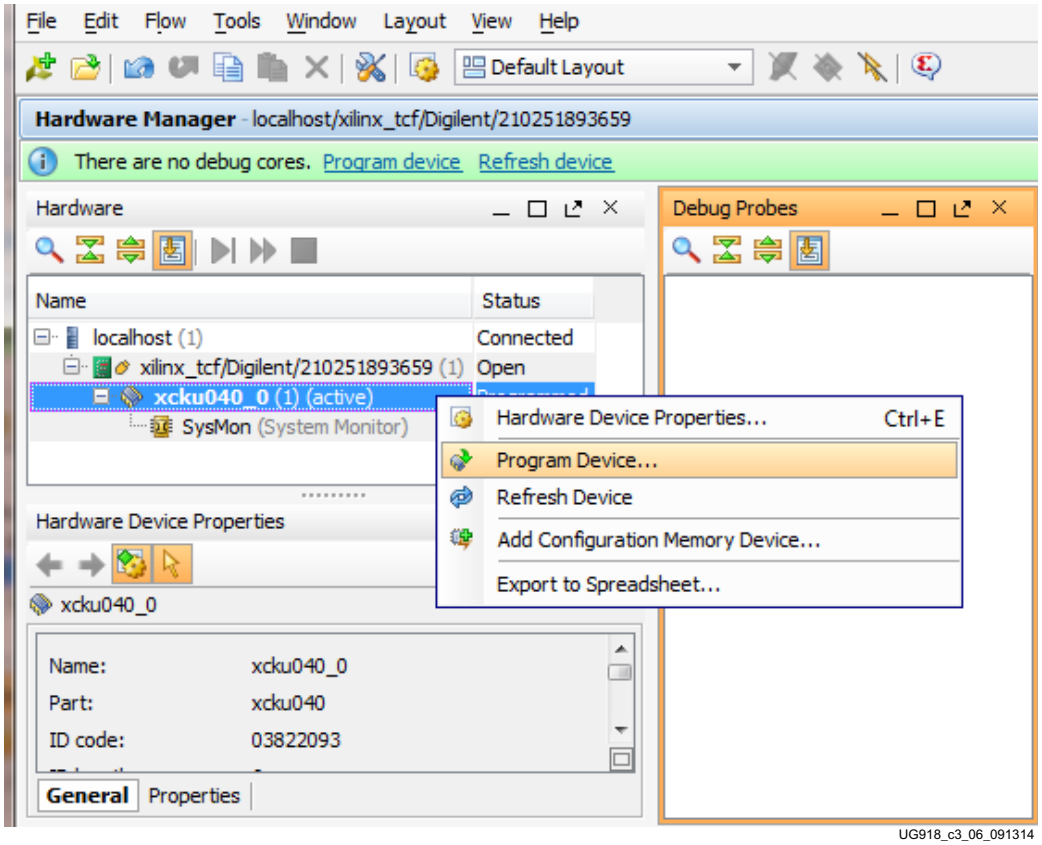


Figure 3-6: **Select Device to Program**

b. In the **Bitstream file** field, browse to the location of the BIT file and click **Program** (see Figure 3-7):

```
<working_dir>/kcu105_aximm_dataplane/ready_to_test/trd02.bit
```

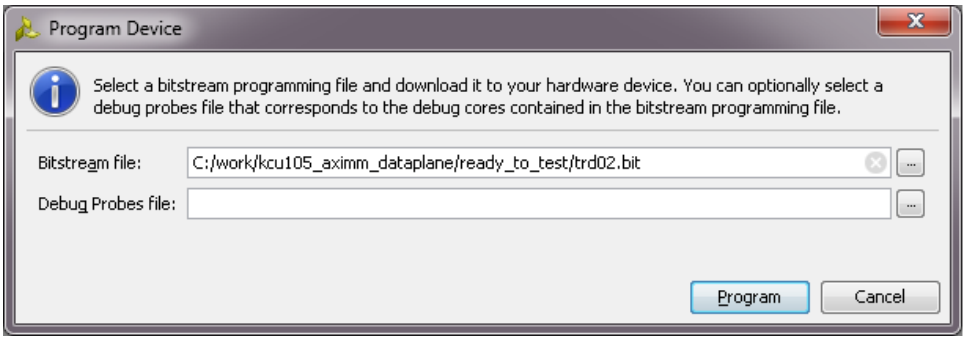


Figure 3-7: **Program Device Window**

5. Check the status of the design by observing the GPIO LEDs positioned at the top right corner of the KCU105 board (see Figure 3-8). After FPGA configuration, the LED status from left to right indicate
   - LED position **4**: ON if the DDR4 calibration is successful
   - LED position **3**: ON if the link speed is Gen3, flashing indicates a link speed error
   - LED position **2**: ON if the lane width is x8, flashing indicates a lane width error
   - LED position **1**: Heart beat LED, flashes if the PCIe user clock is present
   - LED position **0**: ON if the PCIe link is UP

   Note: The LED position numbering used here matches with the LED positions on the board.

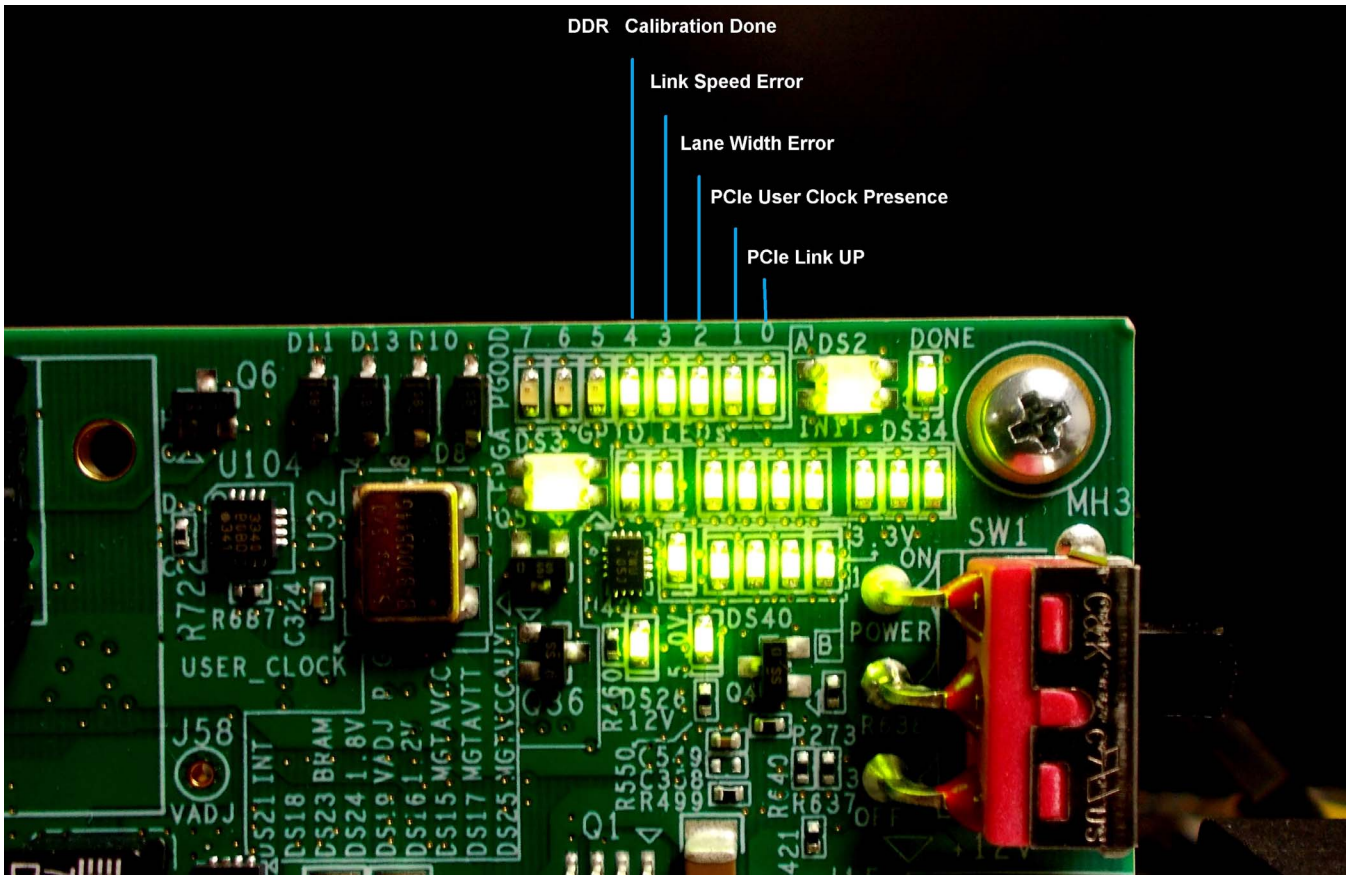


Figure 3-8: **GPIO LED Indicators**

Setup and Testing the Reference Design

Setup on a Linux System

Proceed as follows to test the reference design using the Linux drivers and Fedora 20 live DVD.

Figure 3-9 shows the different stages of the Fedora 20 boot. After you reach the third screen as shown in Figure 3-9, click the **Try Fedora** option, then click **Close**. It is recommended that you run the Fedora operating system from the DVD.

CAUTION! *If you want to install Fedora 20 on the hard drive connected to the host system, click the **Install to Hard Drive** option. BE CAREFUL! This option erases any files on the hard disk!*

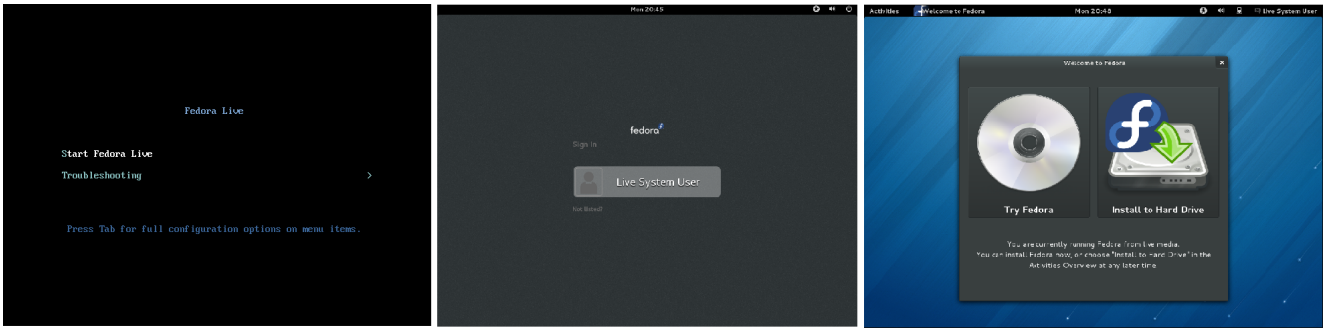


Figure 3-9: **Fedora 20 Boot Stages**

1. Check for PCIe Devices: After the Fedora 20 OS boots, open a terminal and enter **lspci** to see a list of PCIe devices detected by the host:

   $ **lspci | grep -i xilinx**

   The following is displayed:

   ```
   01:00.0 Memory controller: Xilinx Corporation Device 8083
   ```

2. Copy the reference design zip file to any desired directory (`/home/liveuser` is used here). This zip file contains the hardware design, software drivers, and application GUI executables. Unzip this file which should extract the software folder with a quickstart script. (Refer to Appendix A, Directory Structure for the directory structure.)

3. Navigate to the /home/liveuser/kcu105_aximm_dataplane folder and open a terminal. Enter:

```
$ cd /home/liveuser/kcu105_aximm_dataplane

$ su        --> command to login as super user

$ chmod +x quickstart.sh

$ ./quickstart.sh
```

4. The TRD setup screen is displayed (Figure 3-10) and indicates detection of a PCIe device with an ID of 8083, an AXI-MM dataplane design selection, and a Performance/PCIe-DMA-DDR4 driver mode selection. Click **Install** and the drivers are installed. This takes you to the Control and Monitoring GUI as shown in Figure 3-14, page 29.

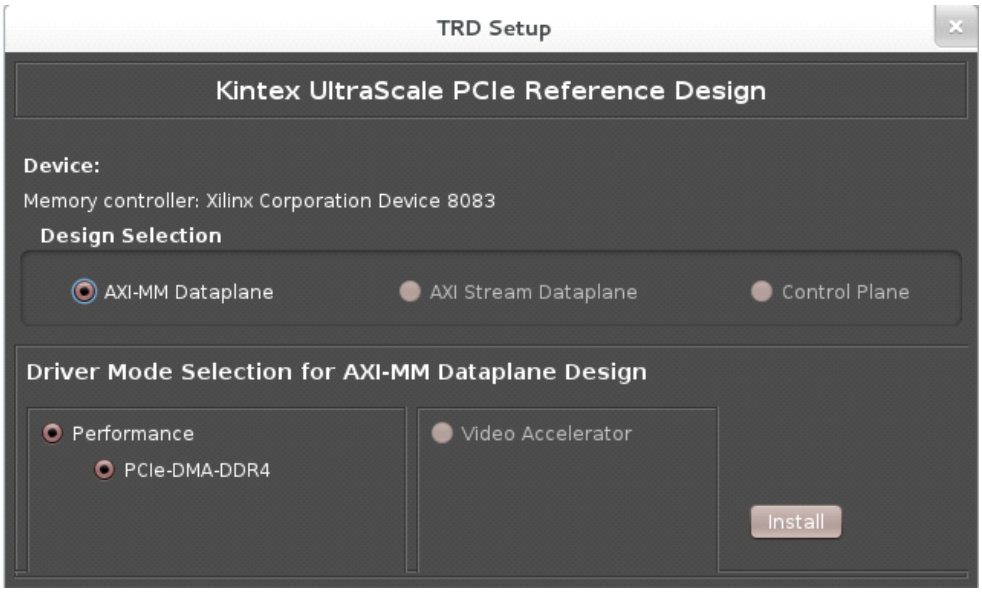


Figure 3-10: **TRD Setup Screen on Linux**

Setup on a Windows System

After booting the Windows OS, follow these steps:

1. Repeat the steps in section Disable Driver Signature Enforcement, page 11.
2. Open Device Manager (click **Start** > **devmgmt.msc** then press **Enter**) and look for the Xilinx PCI Express Device as shown in Figure 3-12.

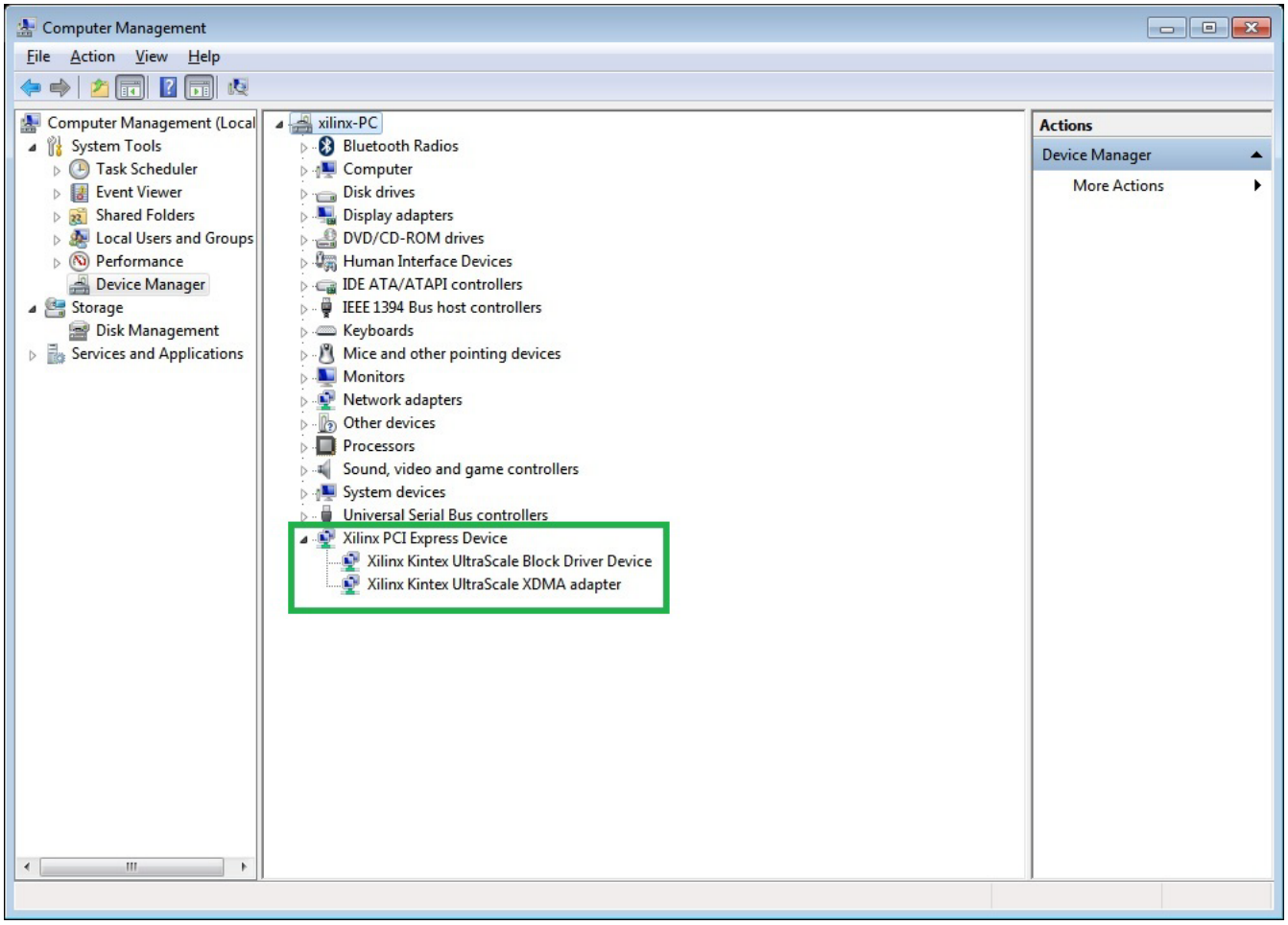


Figure 3-11: **Xilinx PCI Express Device in Device Manager**

3. Open the command prompt with administrator privilege (as shown in Figure 3-12):

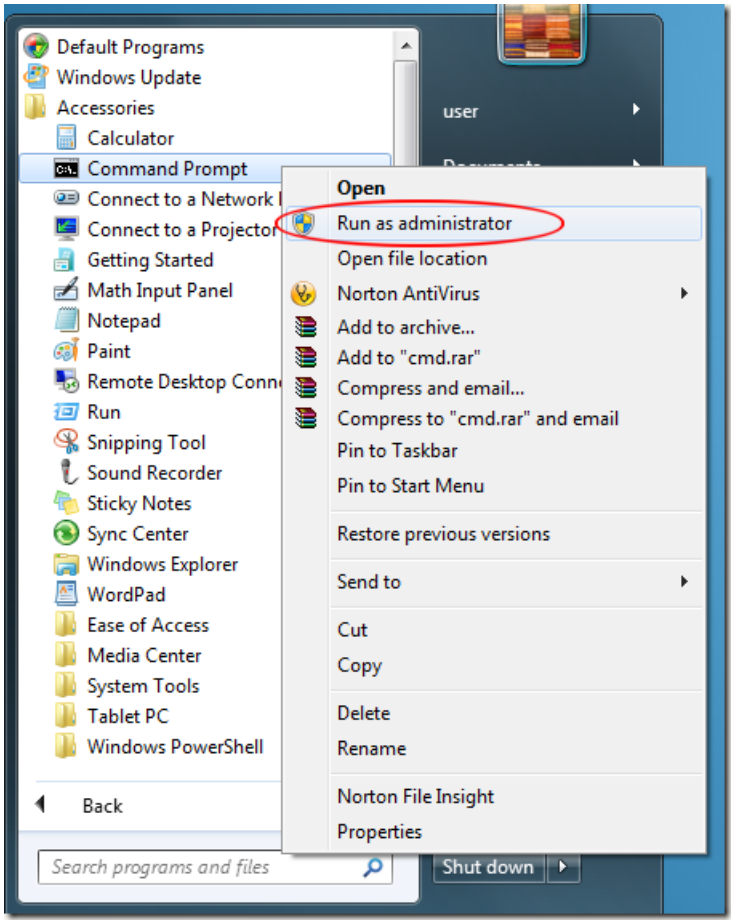


Figure 3-12: **Command Prompt with Administrator Privileges**

4. Navigate to the folder where the reference design is copied:

```
cd <dir>\kcu105_aximm_dataplane
```

5. Run the batch script quickstart_win7.bat:

```
quickstart_win7.bat
```

6. Figure 3-13 shows the TRD Setup GUI. Click **Proceed** to test the reference design. This step takes you to the Control and Monitoring GUI as shown in Figure 3-14, page 29.

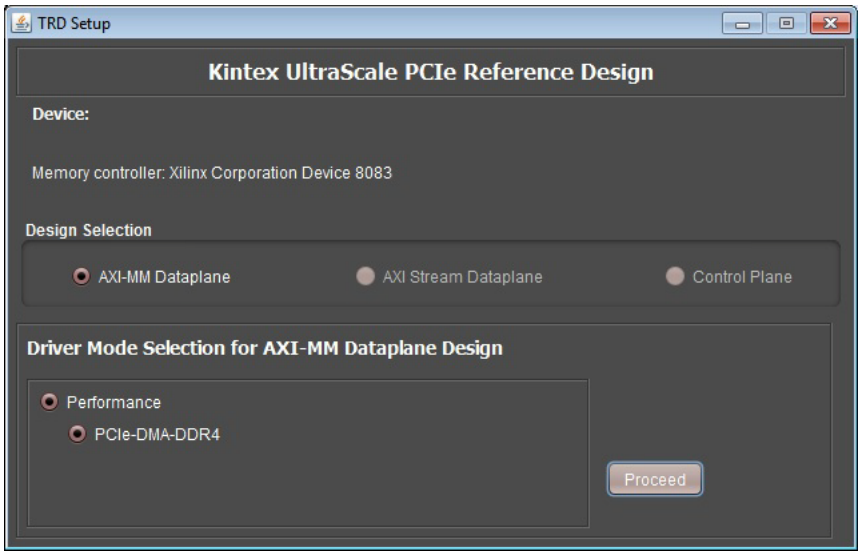


Figure 3-13: **TRD Setup Screen on Windows**

Testing the Reference Design

1. The Control and Monitoring GUI shown in Figure 3-14 can be used to provide information on power and FPGA die temperature, PCI Express endpoint link status, host system initial flow control credits, PCIe write and read throughput numbers, and AXI performance numbers showing AXI throughput.

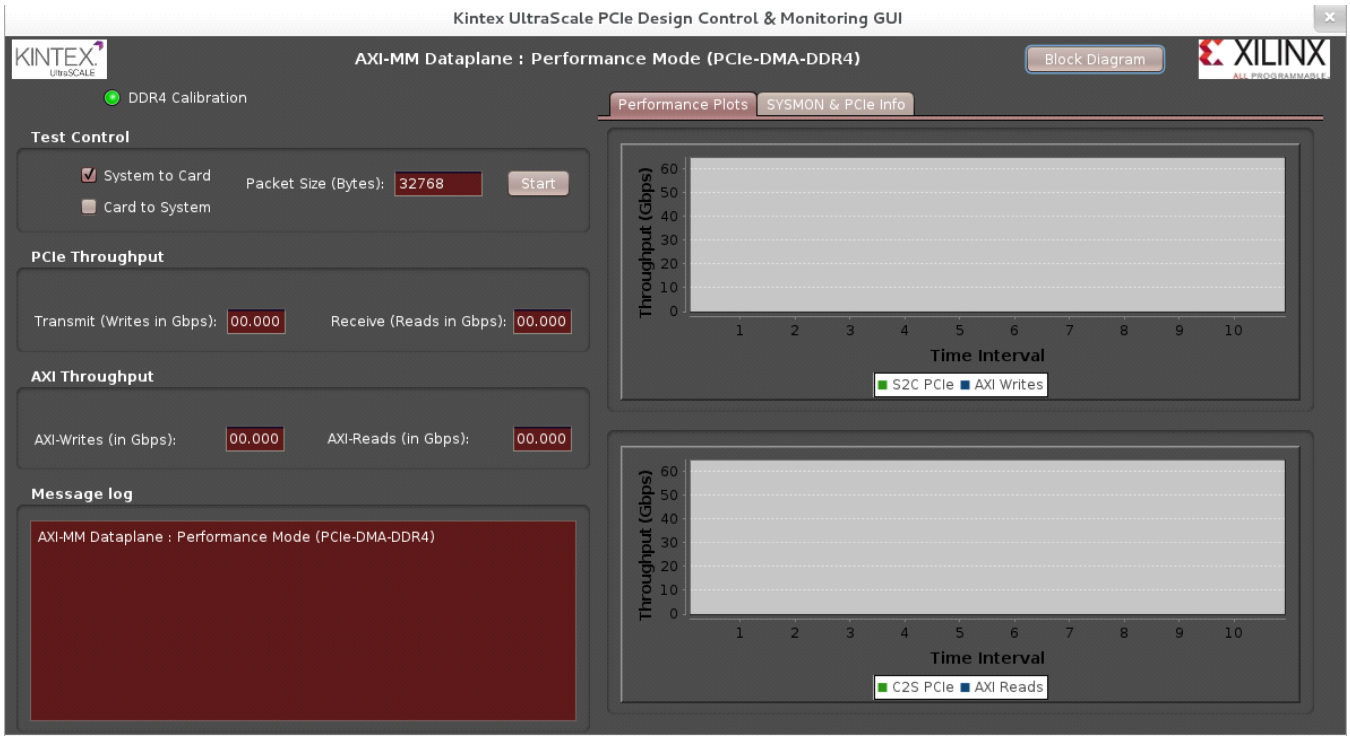


Figure 3-14: **Control and Monitoring GUI**

The following performance plots can be observed using Control and Monitor Interface GUI selections:

a. Data transfer from the Host computer to the FPGA can be started by selecting **System to Card** (S2C) Test Control mode, as shown in Figure 3-15. Click **Start**. Start button changes to **Stop** after the test starts so that you can stop the test when so desired. To stop the test, click **Stop**.

Figure 3-15: **System to Card Performance**

b. Data transfer from the FPGA to the Host computer can be started by deselecting the **System to Card** (S2C) Test Control mode and selecting the **Card to System** (C2S) Test Control mode (Figure 3-16). Click **Start**. To stop the test, click **Stop**.

Figure 3-16: **Card to System Performance**

c. You can send data from the Host to the FPGA and FPGA to the Host at the same time by selecting both **System to Card** (S2C) and **Card to System** (C2S) Test Control modes (Figure 3-17). Click **Start**. To stop the test, click **Stop**.

Figure 3-17: **System to Card and Card to System Performance**

2. You can view the block diagram by clicking **Block Diagram** in top right corner of the screen (Figure 3-18).

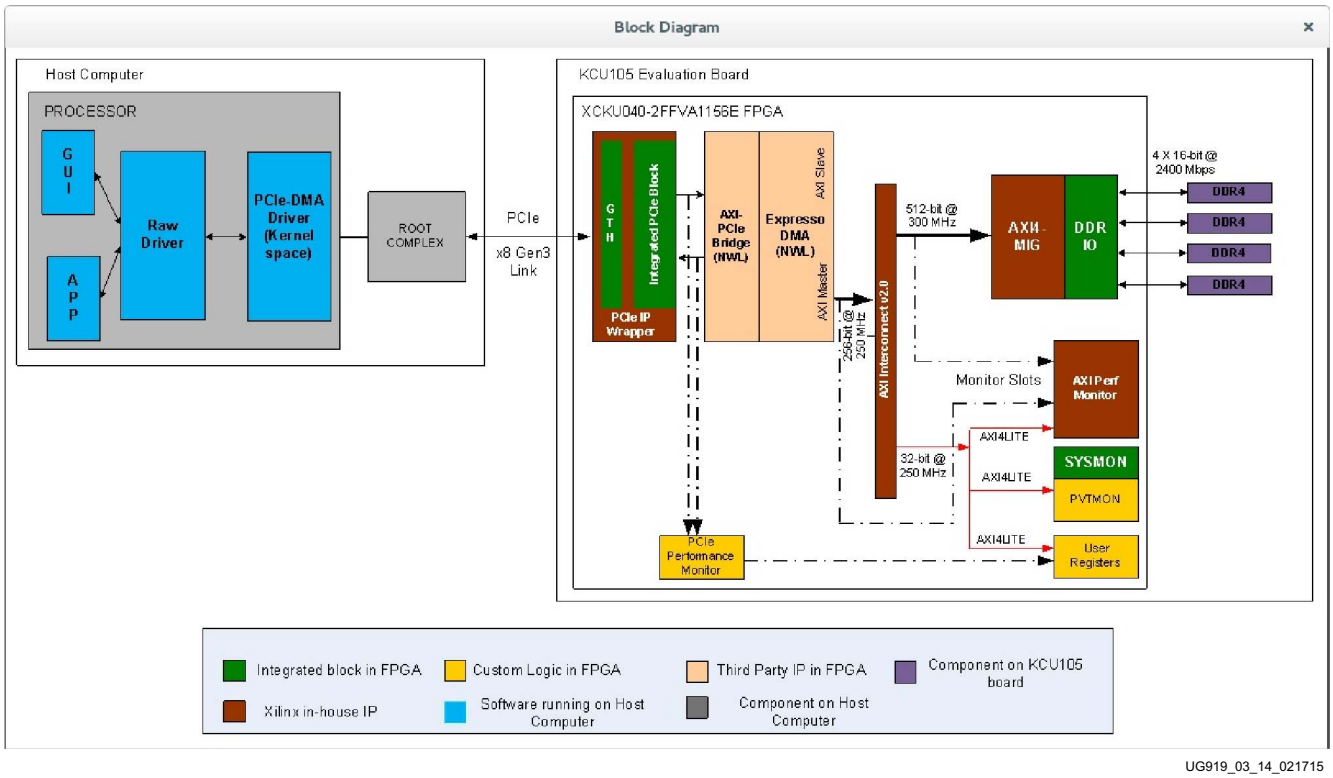


Figure 3-18: **Block Diagram View**

3. PCIe endpoint Status, Host system Initial Flow control credits can be viewed in the middle portion of the GUI. Power and Die temperature monitoring performed by the SYSMON block in the FPGA can be viewed by clicking the **SYSMON & PCIe Info** tab (Figure 3-19).

UG919_03_15_013015

Figure 3-19: **SYSMON and PCIe Information**

4. Click **X** on top right hand corner of the GUI to close the GUI.

- On a Linux host computer, this step uninstalls the drivers and returns the GUI to the TRD Setup screen.
- On a Windows host computer, this step just returns to the TRD Setup screen. Close the TRD Setup Screen and power off the host machine and then the KCU105 board.

Remove Drivers from a Windows Host Computer

To remove the drivers from a Windows host computer, follow these steps:

- Power on the host computer
- After it boots, from the Windows explorer, navigate to the folder in which the reference design is downloaded (`<dir>\kcu105_aximm_dataplane\software\windows\`) and run the setup file with administrator privileges, as shown in Figure 2-2, page 12.

5. Click **Next** after the Install Shield Wizard opens up, as shown in Figure 3-20.

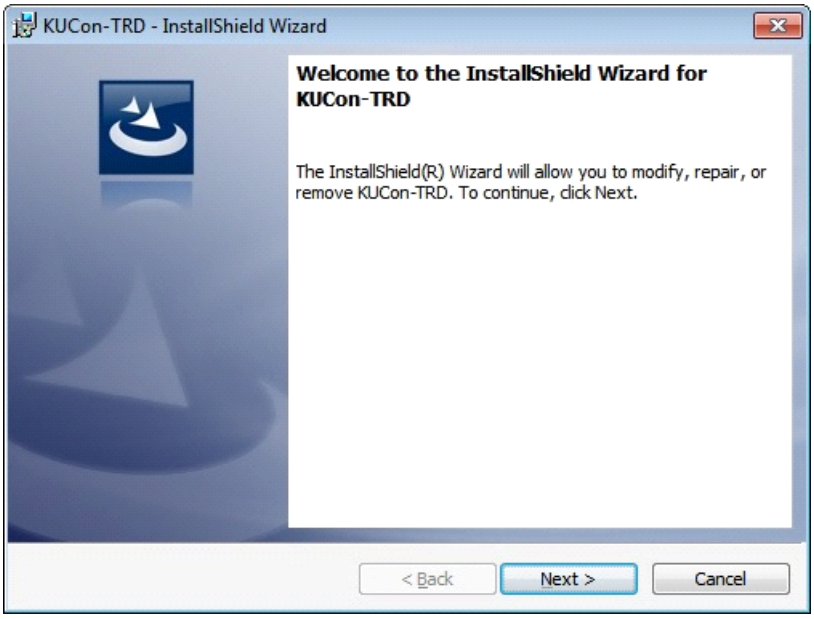


Figure 3-20: **Successful Removal of the Drivers**

3. Select **Remove** and click **Next** to proceed (see Figure 3-21).

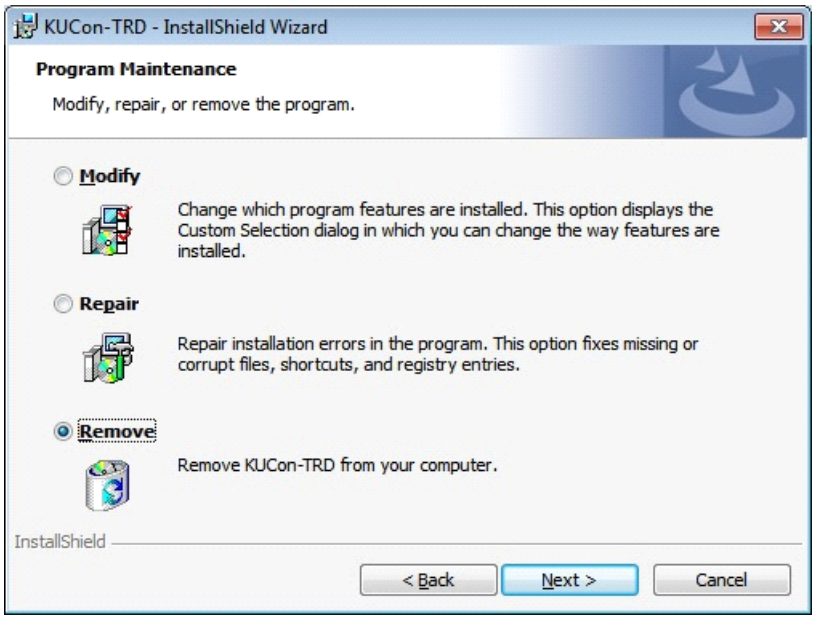


Figure 3-21: **InstallShield Wizard - Remove Drivers Selection**

4. Click **Remove** to remove drivers from the host system as shown in Figure 3-22.

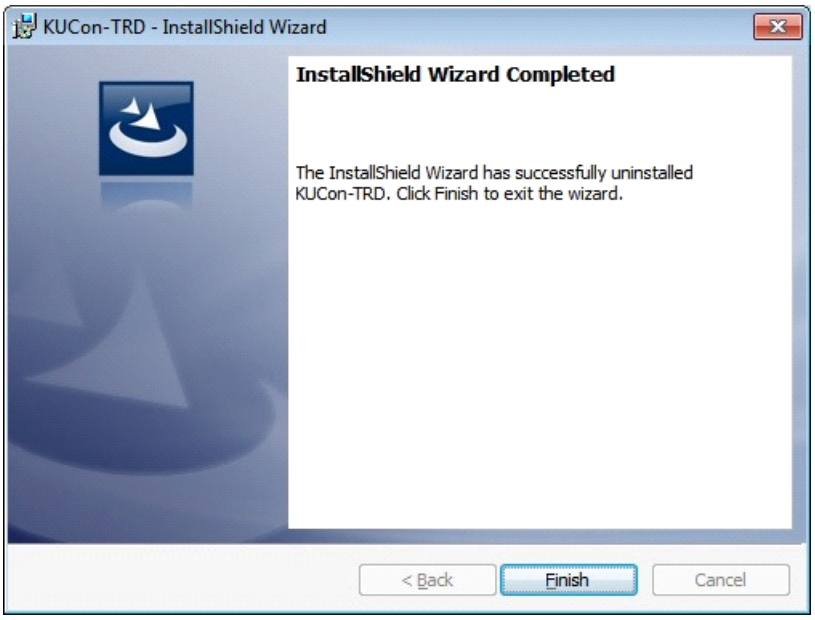


Figure 3-22: **InstallShield Wizard - Remove Drivers**

5. Click **Finish** to exit the wizard (see Figure 3-23).

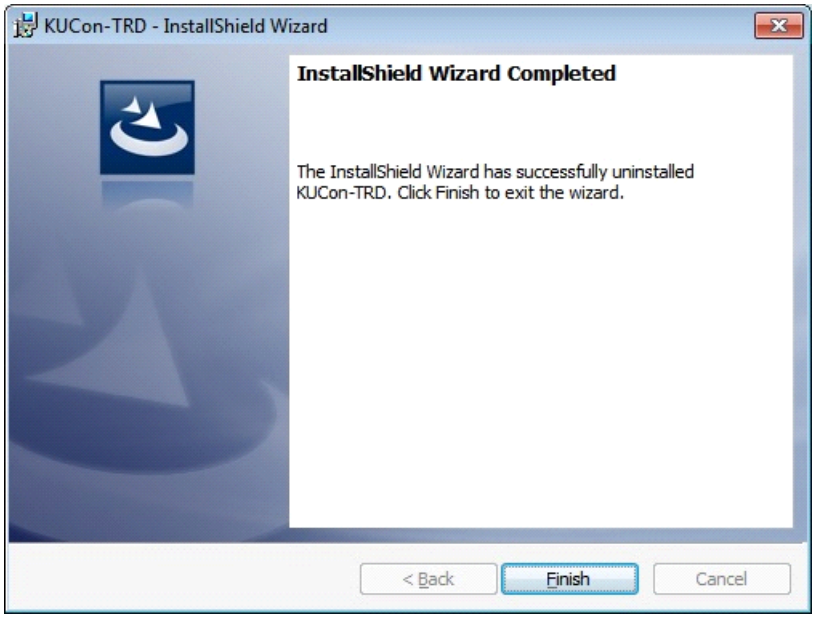


Figure 3-23: **Finish InstallShiled Wizard**

Implementing and Simulating the Design

This chapter describes how to implement and simulate the PCIe memory-mapped data plane reference design. The time required to build the design and run simulation and implementation can vary from system to system depending on the system hardware configuration.

Note: All of the steps presented in this chapter to run simulation and implementation should be run on the control PC that has the Vivado tools installed.

Note: In Windows, if the path length is more than 260 characters, then design implementation or simulation using Vivado Design Suite might fail. This is due to a Windows OS limitation. Refer to the following AR for more details: KCU105 Evaluation Kit Master Answer Record (AR 63175).

Implementing the Base Design

1. Copy the reference design ZIP file to a desired directory on the control PC. Unzip the ZIP file.
2. Open a terminal (in Linux and setup Vivado environment) or a Vivado Tcl shell (on Windows).
3. Navigate to the `kcu105_aximm_dataplane/hardware/vivado/scripts/base` folder.
4. To run the implementation flow, enter:

```
$ vivado -source trd02_base.tcl
```

This opens the Vivado Integrated Design Environment (IDE), loads the block diagram and adds required top file and XDC file to the project (see Figure 4-1).

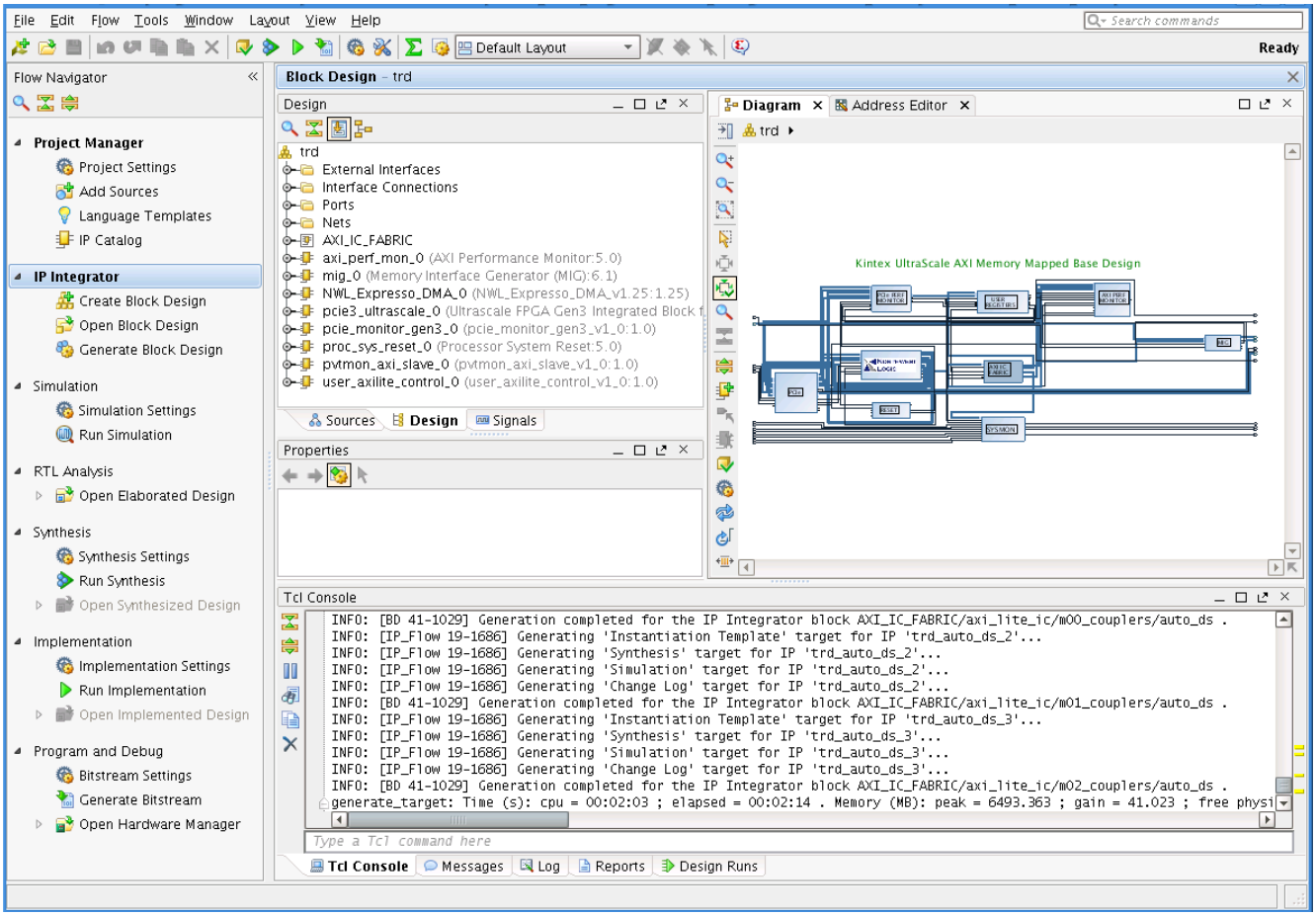


Figure 4-1: **Base Design – Project View**

5. In the Flow Navigator panel, click the **Generate Bitstream** option which runs synthesis and implementation, and generates the bit file (see Figure 4-2). The generated bitstream can be found under the following directory:
   `kcu105_aximm_dataplane/hardware/vivado/runs_base/trd02.runs/impl_1/`

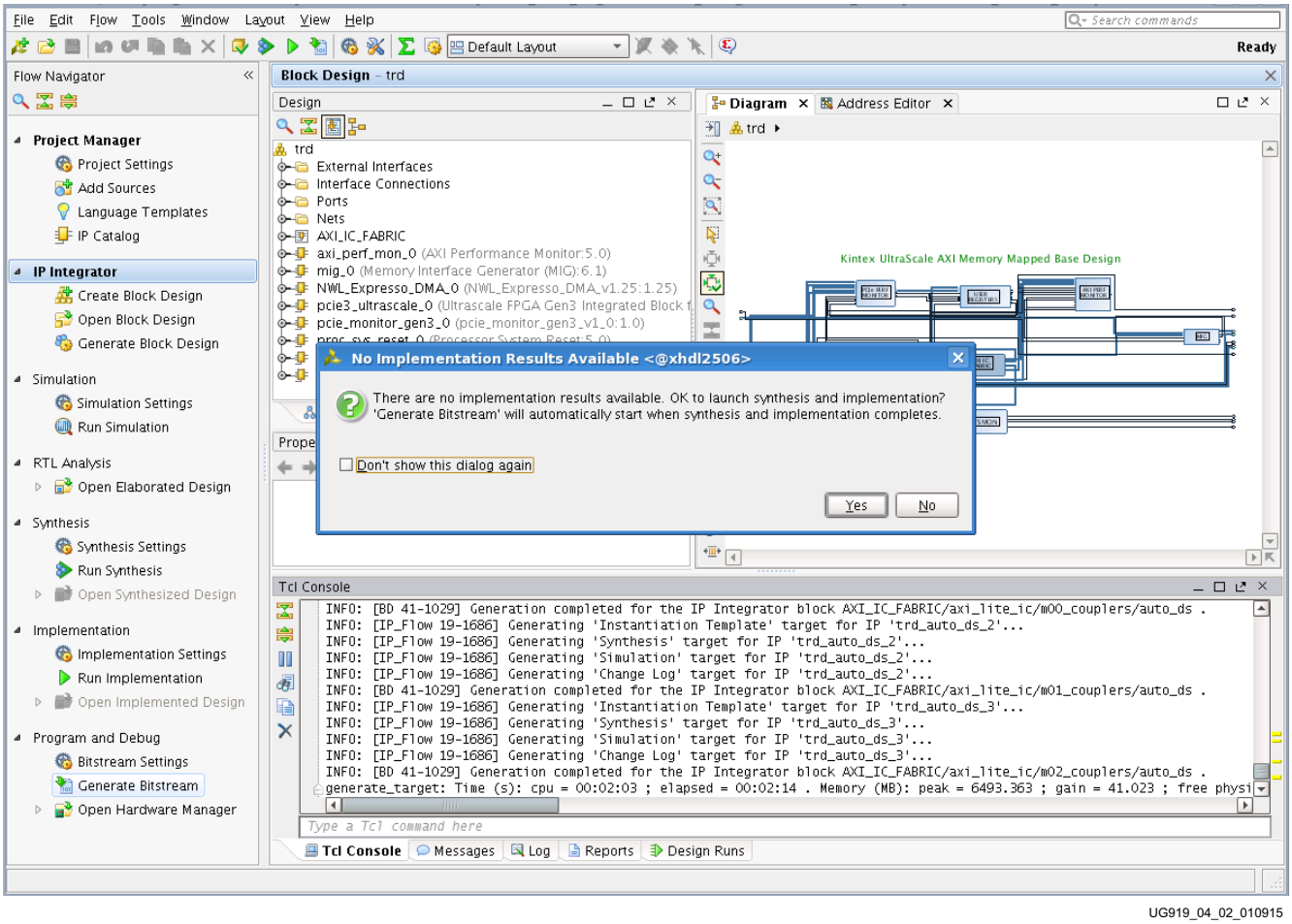


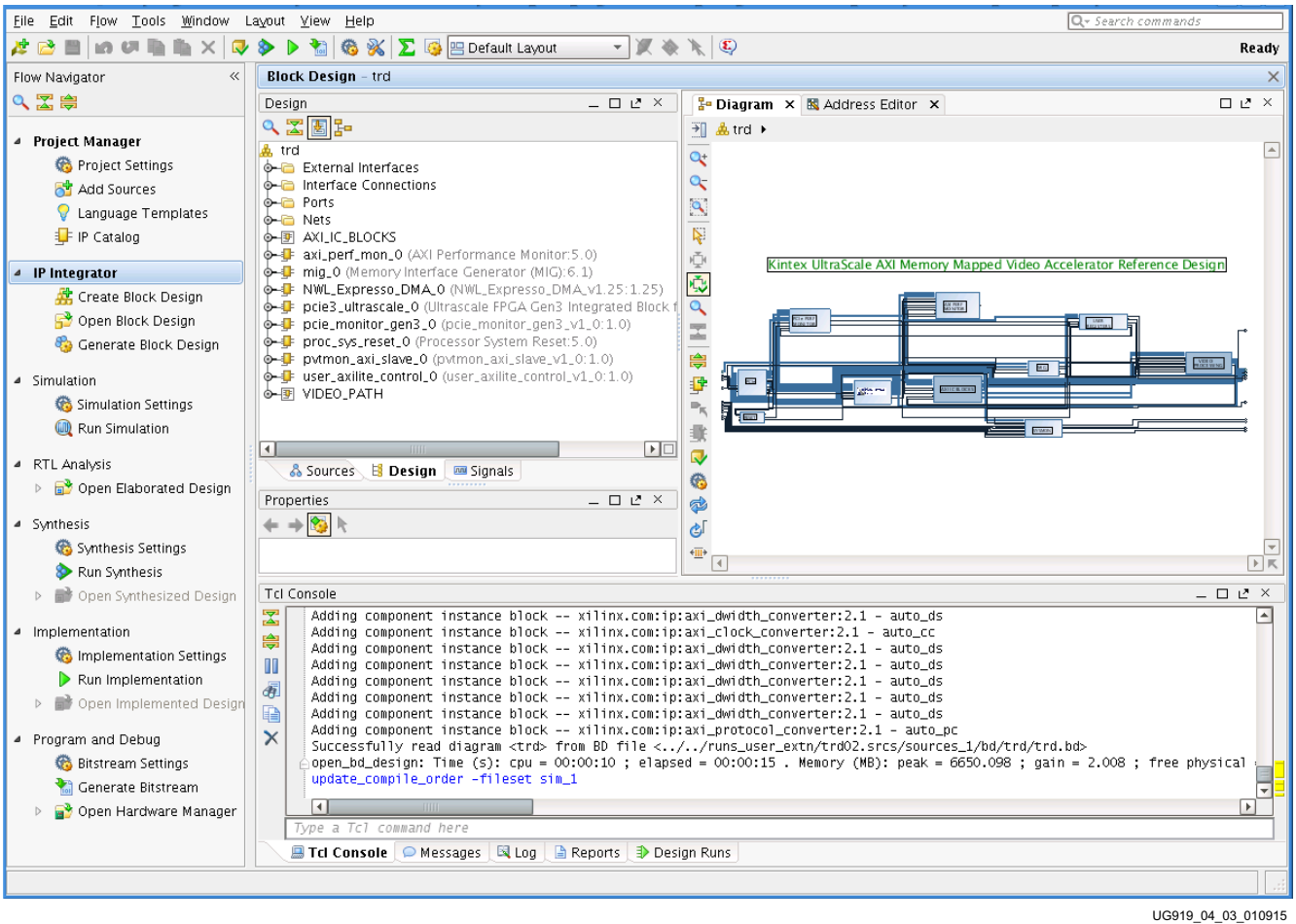
Figure 4-2: **Base Design Generate Bitstream**

Implementing the User Extension Design

1. Open a terminal (in Linux and setup Vivado environment) or a Vivado Tcl shell (on Windows).
2. Navigate to the `kcu105_aximm_dataplane/hardware/vivado/scripts/user_extn` folder.
3. To run the implementation flow enter:

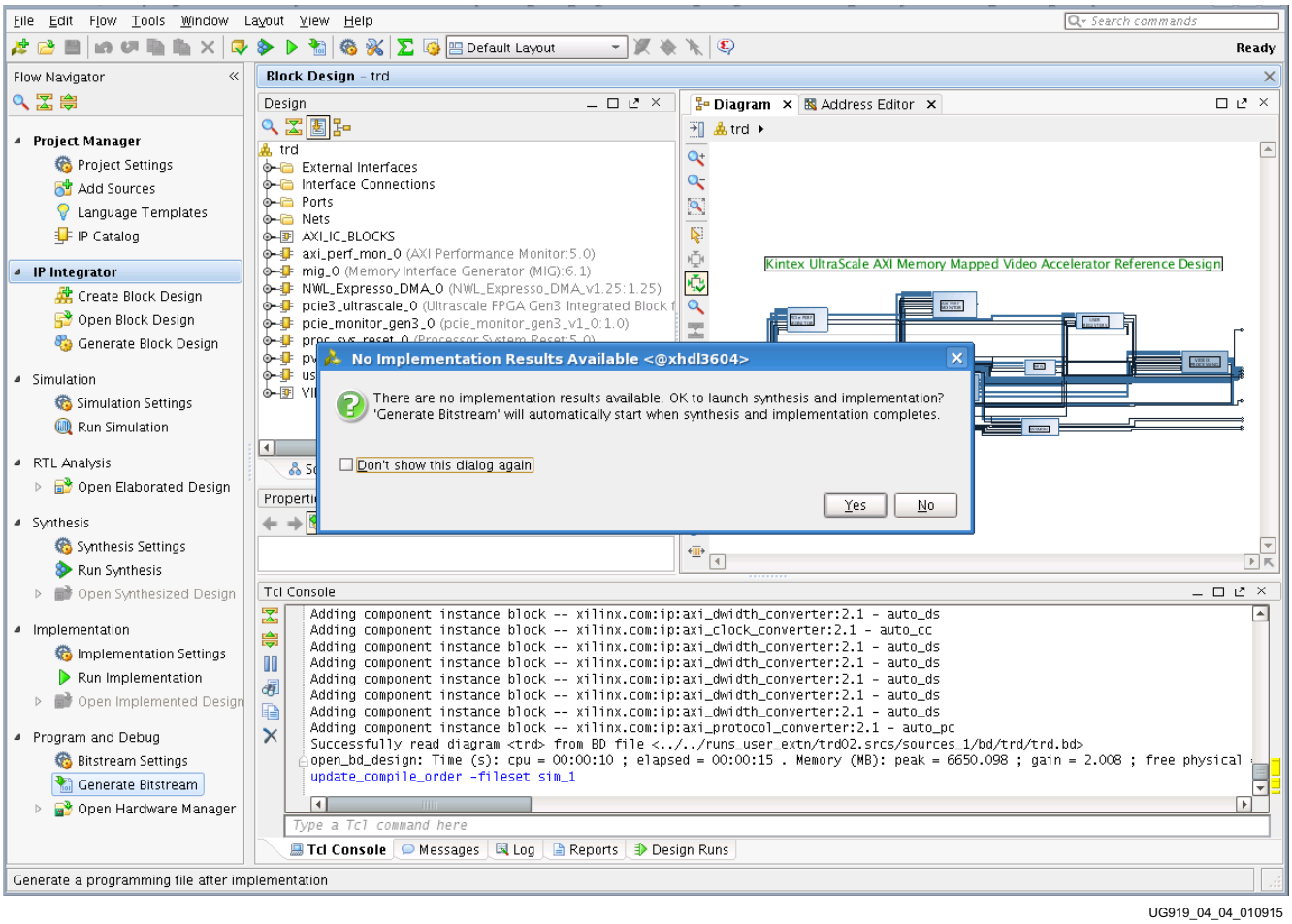
```
$ vivado -source trd02_user_extn.tcl
```

This opens the Vivado IDE, loads the block diagram, and adds the required top and XDC files to the project (see Figure 4-3). In the Flow Navigator panel, click the **Generate Bitstream** option which runs synthesis and implementation, and generates the bit file (see Figure 4-4). The generated bitstream can be found under the following directory: `kcu105_aximm_dataplane/hardware/vivado/runs_user_extn/trd02.runs/impl_1/`



UG919_04_03_010915

Figure 4-3: **User Extension Design – Project View**



UG919_04_04_010915

Figure 4-4: **User Extension – Project Generate Bitstream**

Simulating the Base Design Using Vivado Simulator

The PCI Express memory-mapped base reference design can be simulated using the Vivado Simulator.

Note: The testbench and the endpoint PCIe IP block are configured to use PHY Interface for PCI Express (PIPE) mode simulation.

The test bench initializes the bridge and DMA, sets up the DMA for system-to-card (S2C) and card-to-system (C2S) data transfer. The testbench configures the DMA to transfer one 64 byte packet from host memory (basically an array in the testbench) to card memory (DDR4 model) and readback the data from the card memory. The testbench then compares the data read back from the DDR4 model with the transmitted packet.

Simulation setup is provided only for the base design and not for the pre-built user extension design.

The simulation testbench requires a DDR4 model. The DDR4 model is obtained by generating MIG IP example design. There is a place holder for the DDR4 model under the `kcu105_aximm_dataplane/hardware/vivado/scripts/base/ddr4_model` directory. Copy the files under DDR4 model obtained from MIG IP example design to above mentioned directory before executing the simulation script.

Refer to the *UltraScale Architecture-Based FPGAs Memory Interface Solutions Product Guide* (PG150) [Ref 7] for steps to generate the MIG IP example design.

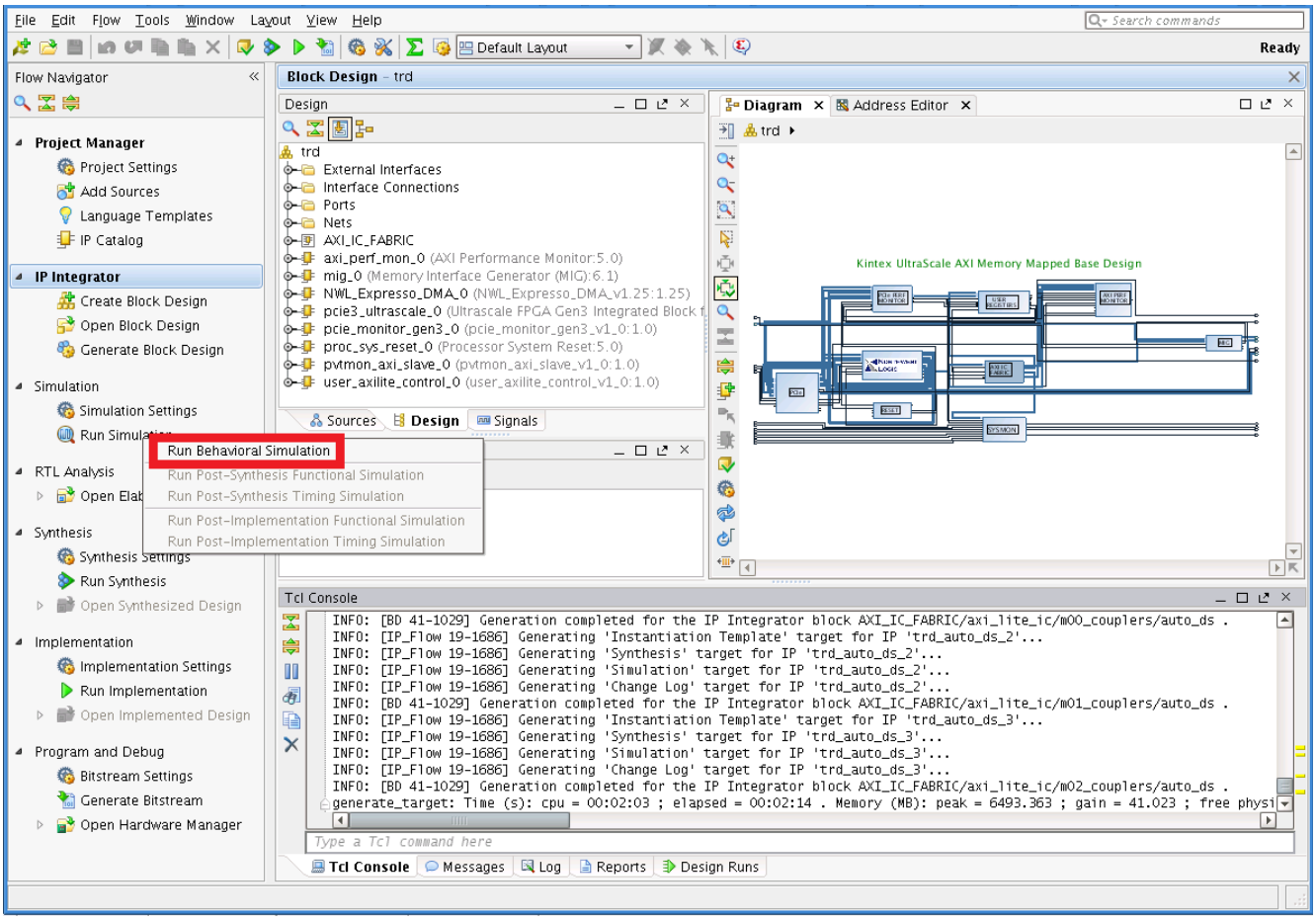
Running Simulation using the Vivado Simulator

1. Open a terminal window on a Linux system and setup Vivado environment, or open a Vivado Tcl shell on a Windows system.
2. Navigate to the `kcu105_aximm_dataplane/hardware/vivado/scripts/base` folder.
3. To run simulation enter:

   ```
   $ vivado -source trd02_base.tcl
   ```

   This opens the Vivado IDE (Figure 4-1, page 38) with the target simulator set to the Vivado simulator.

4. In the Flow Navigator panel, under Simulation, click **Run Simulation** and select **Run Behavioral Simulation** (Figure 4-5).



UG919_04_07_010915

Figure 4-5: **Base Design Behavioral Simulation using the Vivado Simulator**

Targeted Reference Design Details and Modifications

This chapter describes the PCIe memory-mapped data plane hardware design and software components in detail, and provides modifications to add a video accelerator design.

Hardware

Figure 5-1 shows a block-level overview of the PCIe memory-mapped data plane.

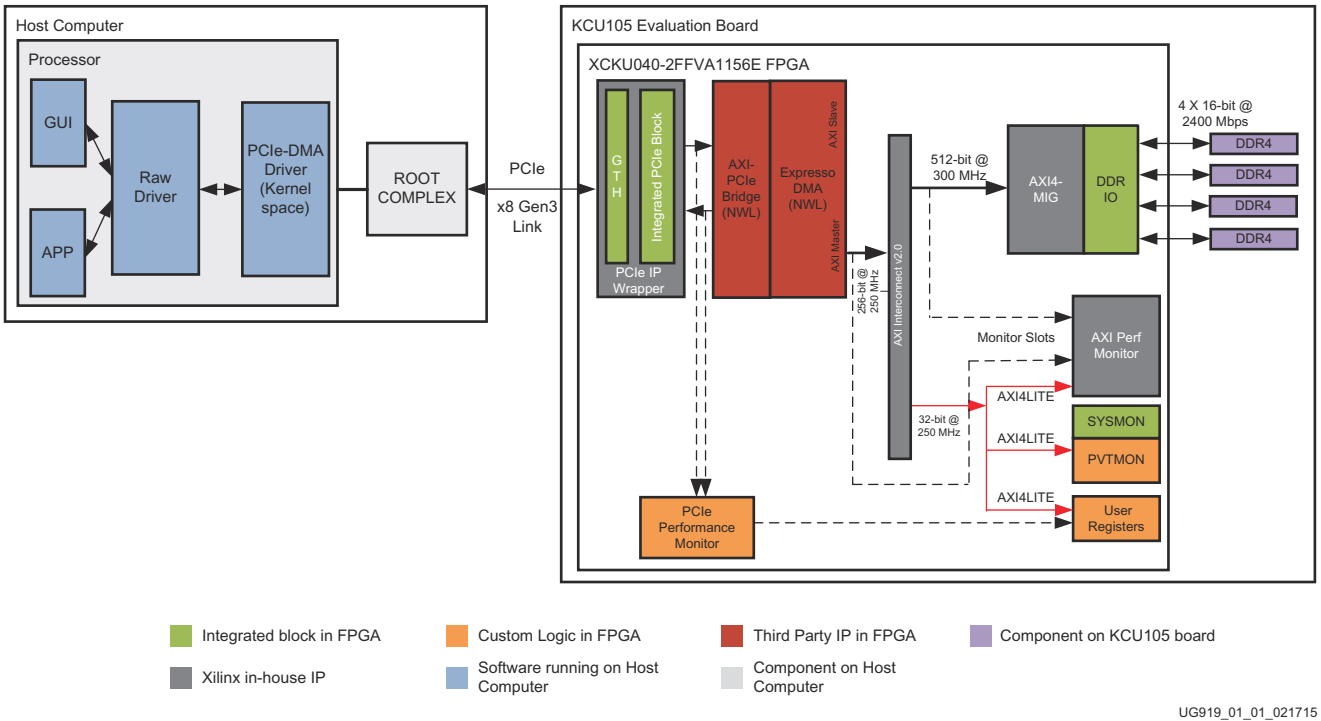


Figure 5-1: **PCIe Memory Mapped Data Plane TRD Block Diagram**

The details of the hardware architecture are provided in these sections:

PCI Express IP Core

The PCI Express IP core is used in the following configuration:

- X8 Gen3 Line rate (8 GT/s/lane/direction)
- Three 64-bit BARs each of 1 MB size
- MSI-X Capability

See *LogiCORE IP UltraScale FPGAs Gen3 Integrated Block for PCI Express Product Guide* (PG156) [Ref 5] for more information.

NWL Expresso DMA IP Core

The Expresso DMA IP from NWL includes AXI-PCIe Bridge and Expresso DMA in one netlist bundle. See the Northwest Logic Expresso DMA Bridge Core website to obtain a user guide.

Note: The IP netlist used in the reference design supports a fixed configuration where the number of DMA channels and translation regions is fixed; for higher configurations of the IP, contact NWL.

Note: The Northwest Logic Expresso IP provided with the design is an evaluation version of the IP. It times out in hardware after 12 hours. To obtain a full license of the IP, please contact Northwest Logic.

AXI-PCIe Bridge

This IP is a protocol conversion unit between PCIe and AXI3 domains. It has the following features:

- Protocol conversion from PCIe transactions to AXI3 and vice-versa
- Support for two ingress translation regions to convert PCIe BAR mapped transactions to AXI3 domain transactions

- Single ingress translation is enabled (0x800)
- The address translation is setup as shown in Table 5-1.

For example, assume that the PCIe BAR2 physical address is 0x2E000000. Any memory read request targeted to address 0x2E000000 is translated to 0x44A00000.

Table 5-1: **Bridge Address Translation**

Ingress Source Base	Ingress Destination Base	Aperture Size
BAR2	0x44A00000	1M

- Bridge Register Initialization
  - Bridge Base Low (0x210) is programmed to (BAR0 + 0x8000)
  - Bridge Control register (0x208) is programmed to set bridge size and enable translation
- After bridge translation has been enabled, the ingress registers can be accessed with Bridge Base + 0x800
- The read and write request size for Master AXI read and write transactions is programmed to be 512B (cfg_axi_master register at offset 0x08 from [BAR0 + 0x8000])

Expresso DMA

Key features of Expresso DMA are:

- High-Performance Scatter Gather DMA, designed to achieve full bandwidth of AXI and PCIe
- Separate source and destination Scatter-Gather queues with separate source and destination DMA completion Status queues
- DMA channels merge the source and destination Scatter Gather information

DMA Operation

This section provides a summary of DMA operation.

The Expresso DMA has four queues per channel:

- SRC-Q, which provides data buffer source information and corresponding STAS-Q, which indicates SRC-Q processing completion by DMA
- DST-Q, which provides destination buffer information and corresponding STAD-Q, which indicates DST-Q processing completion by DMA

The Q elements' layout is depicted in Figure 5-2.

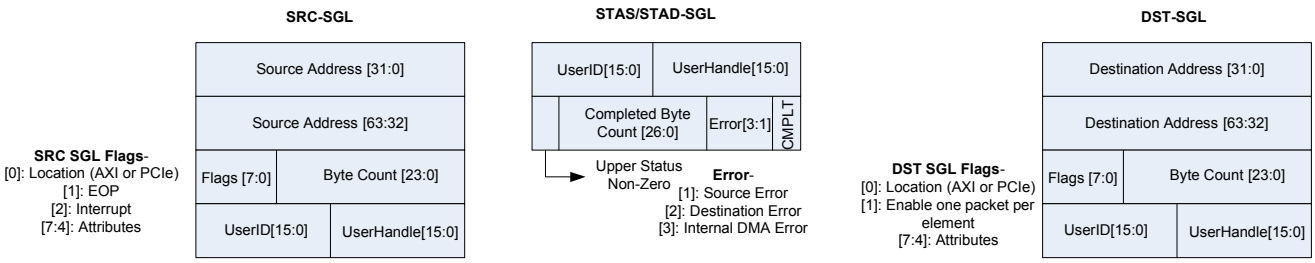


Figure 5-2: **SGL Q-Element Structure**

These Qs can be resident either in host memory or AXI memory and Q elements are required to be in contiguous location for DMA to fetch multiple SRC/DST-Q elements in a burst fetch. The software driver sets up the Q elements in contiguous location and DMA takes care of wrap-around of Q. Every DMA channel has the following registers pertaining to each Q:

- Q_PTR: The starting address of the Q
- Q_SIZE: The number of SGL elements in the Q
- Q_LIMIT: Index of the first element still owned by the software; DMA hardware wraps around to start element location when Q_LIMIT is equal to Q_SIZE

Figure 5-3 summarizes DMA operation.

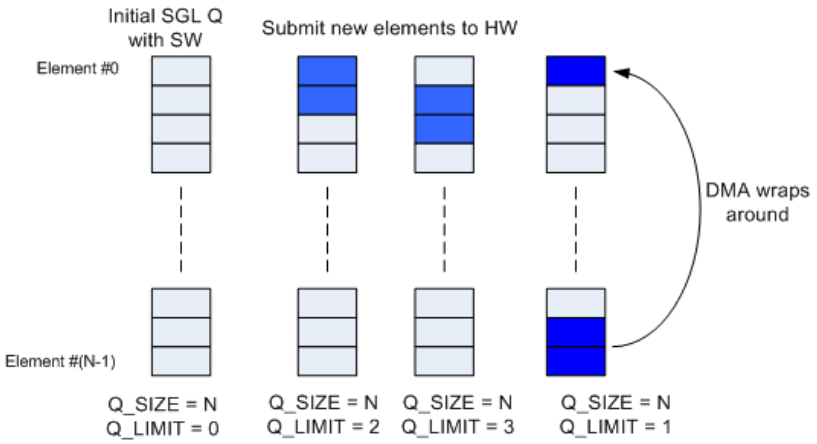


Figure 5-3: **DMA Operation**

The Expresso DMA supports multi CPU DMA operation (a single DMA channel can be managed by both the host CPU as well as the AXI CPU). The use-model in this reference design is of a host software driver managing all DMA Qs for a given channel, with all Qs

resident in host memory. The host software is made aware of the AXI domain addresses needed for DMA operation.

System to Card Flow

1. Software sets up SRC-Q with buffer address in host and appropriate buffer size in host memory
2. Software sets up DST-Q with buffer address in AXI domain and appropriate buffer size in host memory
3. Software sets up STAS-Q and STAD-Q in host memory
4. On enabling DMA, DMA fetches SRC and DST elements over PCIe
5. DMA fetches the buffer pointed to by SRC elements (upstream memory read) and provides it on AXI interface (as AXI write transaction) targeting AXI address provided in DST-Q
6. On completion of DMA transfer (encountering EOP), STAS-Q and STAD-Q are updated in host memory

Card to System Flow

1. Software sets up SRC-Q with buffer address pointing to AXI domain and appropriate buffer size in AXI memory
2. Software sets up DST-Q with buffer address in host and appropriate buffer size
3. Software sets up STAS-Q and STAD-Q in host memory
4. On enabling DMA, DMA fetches SRC and DST elements over PCIe
5. DMA fetches the buffer pointed to by SRC elements over AXI (through AXI read transaction) and writes it into address provided in DST-Q in host memory (upstream memory write)
6. On completion of DMA transfer (encountering EOP), STAS-Q and STAD-Q are updated in host memory

Status Updates

The status elements are updated only on EOP and not every SGL element. This section describes the status updates and use of the User handle field.

Relation between SRC-Q and STAS-Q:

As depicted in Figure 5-4, packet-0 spans across three SRC-Q elements; the third element indicates EOP=1 with UserHandle=2. On EOP, DMA updates STAS-Q with UserHandle=2 which corresponds to a handle value in SRC-Q element with EOP=1. Similarly, packet-1 spans two elements and in STAS-Q an updated handle value corresponds to EOP =1

element. This UserHandle mechanism allows software to associate number of SRC-Q elements corresponding to a STAS-Q update.

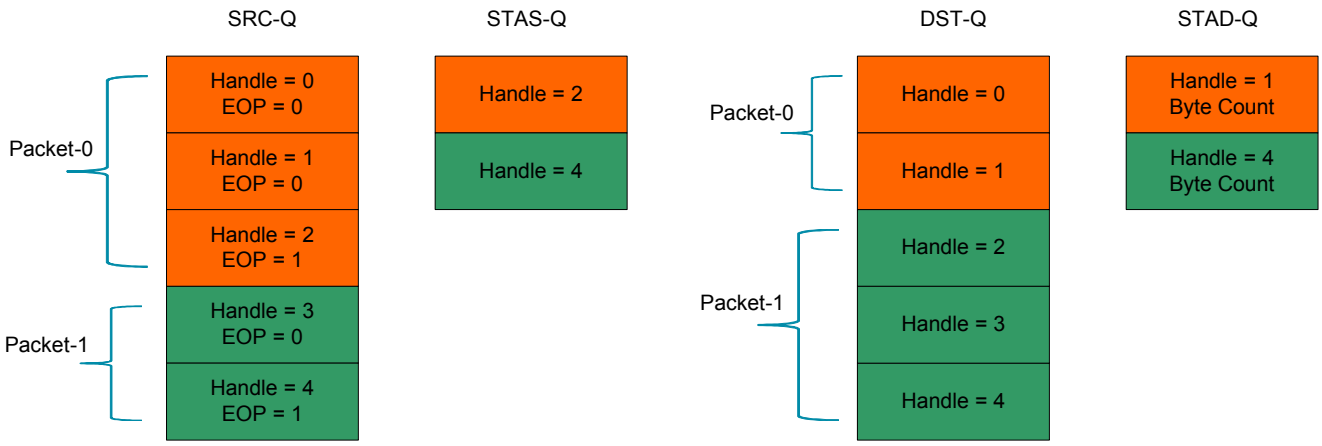


Figure 5-4: **SRC-Q and STAS-Q**

Relation between DST-Q and STAD-Q:

Software sets up DST-Q elements with predefined UserHandle values and pointing to empty buffers. In Figure 5-4, packet-0 spans across two DST-Q elements; one STAD-Q element is updated with a Handle value of the last DST-Q element used by the packet and corresponding packet length. Software thus maintains the number of DST-Q elements used (buffers used and appropriate buffer fragment pointers) for a particular status completion.

AXI Interconnect

The AXI Interconnect is used to connect the various IPs together in a memory-mapped system. The interconnect is responsible for:

- Converting AXI3 transactions from the AXI-PCIe Bridge into AXI4 transactions for various slaves
- Decoding addresses to target the appropriate slave

There are three slaves connected to the AXI Lite Interconnect, and the AXI Interconnect directs the read/write requests to appropriate slaves based on the address shown in Table 5-2.

Table 5-2: **AXI LITE Slaves Address Decoding**

AXI Lite Slave	Address Range	Size
Reserved	0x44A00000 - 0x44A00FFF	4K
User Space registers	0x44A01000 - 0x44A01FFF	4K

Table 5-2: **AXI LITE Slaves Address Decoding *(Cont'd)***

AXI Lite Slave	Address Range	Size
PVTMON	0x44A02000 - 0x44A02FFF	4K
AXI Performance Monitor	0x44A10000 - 0x44A1FFFF	64K

See *LogiCORE IP AXI Interconnect Product Guide* (PG059) [Ref 6] for more information.

AXI-MIG Controller and DDR4

The design has DDR4 operating at 64-bit@2400 Mb/s. The AXI data width of the AXI-MIG controller is 512-bit@300 MHz. The reference clock for the AXI MIG Controller IP is on-board 300 MHz differential clock.

See *LogiCORE IP Memory Interface Solutions Product Guide* (PG150) [Ref 7] for more information.

PCIe Performance Monitor

The PCIe Performance Monitor snoops the interface between PCIe and Expresso DMA blocks. It calculates the number of bytes flowing in and out of the PCIe block per second. This gives the throughput of the PCIe block in transmit and receive directions.

AXI Performance Monitor

The AXI Performance Monitor is used to calculate the write and read byte throughput on the AXI Master interface of DMA and on the memory-mapped interface of the AXI MIG. This measures the payload performance and does not include any PCIe overhead, DMA setup, or SGL element overheads.

See *LogiCORE IP AXI Performance Monitor Product Guide* (PG037) [Ref 8] for more information.

User Space Registers

User Space registers implement the AXI Lite interface for sample interval and scaling factor programming of the PCIe Performance Monitor. The transmit and receive byte counts computed by the PCIe performance block, and the initial flow control credit information of the root complex are written to registers inside this block. The GUI running on the host machine interacts with the device driver to get those values from user space registers and displays it on the GUI.

Power and Temperature Monitoring

The design uses a SYSMON block to provide system power and die temperature monitoring capabilities.

The System Monitor block (17 channel, 200 ksps) provides analog-to-digital conversion and monitoring capabilities. It enables reading of voltage and current on different power supply rails (supported on the KCU105 board) which are then used to calculate power.

A lightweight PicoBlaze™ controller is used to setup the SYSMON registers in continuous sequence mode and read various rails data periodically. The output from PicoBlaze is made available in block RAM and an FSM reads various rails from the block RAM (as shown in Figure 5-5) and updates the user space registers. These registers can be accessed over PCIe through a BAR mapped region.

The AXI4 Lite IPIF core is used in the design and the interface logic between the block RAM and the AXI4 Lite IPIF reads the power and temperature monitor registers from block RAM. Providing an AXI4 Lite slave interface adds the flexibility of using the module in other designs.

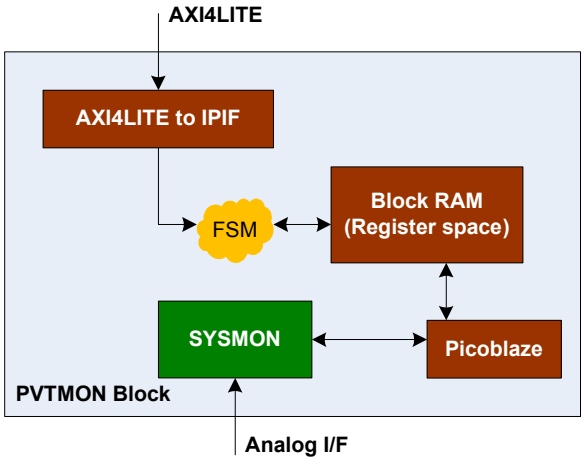


Figure 5-5: **PVTMON Functional Block Diagram**

See *UltraScale Architecture System Monitor User Guide* (UG580) [Ref 9] for more information.

Data Flow

The data transfer between host and card DDR has the following data flow:

1. S2C traffic flow
    a. The host maintains SRC-Q, STAS-Q, DST-Q, STAD-Q (resident on host) – the host manages the card address provided in DST-Q.
    b. DMA fetches SRC-Q, DST-Q and buffer pointed to by SRC-Q from the host over PCIe.
    c. DMA provides data on the AXI-MM interface with addresses as indicated by the buffer address in DST-Q SGL element.
    d. DMA updates STAS-Q and STAD-Q after transfer completion.
2. C2S traffic flow
    a. The host maintains DST-Q, STAD-Q, SRC-Q and STAS-Q – the host manages card address for SRC-Q.
    b. DMA fetches DST-Q, SRC-Q.
    c. DMA fetches buffer pointed to by SRC-Q from AXI domain.
    d. DMA writes buffer to address pointed by DST-Q and then updates STAS-Q and STAD-Q after completion of transfer (over PCIe in host memory).

Note: The address regions to be used on card memory can be pre-defined or advertised by the user logic registers.

Reference Design Modifications, page 65 explains how the steps defined above can be extended to include a hardware accelerator block.

When using further downstream user logic on card (user hook functionality), the sequence of operations is follows:

1. S2C Data Flow
    a. The host sets up SRC-Q, STAS-Q, DST-Q, STAD-Q (resident on host) – the host manages card address in DST-Q.
    b. On STAD-Q completion status, the host sets up user hook logic to read data from the target location.
    c. After the host gets user hook logic read completion status, it releases the DST-Q buffer.
2. C2S Data Flow
    a. The host sets up user hook logic to receive user application data and DST-Q and STAD-Q.

b. After user hook logic indicates completion of reception, the host sets up SRC-Q, STAS-Q.

c. On completion indication in STAD-Q, the user hook logic receive address can be freed for reuse.

Software

Expresso DMA Driver Design

Summary

The section describes the design of the PCIe Expresso DMA driver (hereafter referred to as the *XDMA driver*) with the objective of enabling you to use the same name in your software stack.

Prerequisites

It is assumed that you are aware of the Expresso DMA hardware design and have a basic understanding of PCIe protocol, software engineering fundamentals, and Windows and Linux OS internals.

Frequently Used Terms

1. **XDMA driver**: Low level driver to control the Expresso DMA. The driver is agnostic of applications stacked on top of it and serves the basic purpose of ferrying data across PCIe link.
2. **Application driver**: Device driver layer stacked on XDMA driver and hooks up with a protocol stack or user space application (Ethernet driver, video driver, user traffic generator).
3. **HOST/system**: Typically server/desktop type of a machine with PCIe connectivity.
4. **Endpoint (EP)/Card**: PCIe endpoint, typically an Ethernet card attached to PCIe slots of a server/desktop.
5. **SGL**: Scatter gather list. This is a software array whose elements have a format as prescribed Expresso DMA. This list is used to point to I/O buffers from/to which Expresso DMA transfers data across the PCIe link. Note that although the I/O buffers might be on the host or EP the SGL (source and destination) are always on the host memory. Expresso DMA is capable of deploying SGL on the host or EP, however in context of this reference design, the model of all SGL on host memory is explored.

6. **Source SGL**: SGL used to point to the source I/O buffers. Expresso DMA takes data from source SGL I/O buffers and drains into I/O buffers pointed to by destination SGL I/O buffers. Source SGL is resident in host memory.
7. **Destination SGL**: SGL used to point to the destination I/O buffers. Expresso DMA takes data from source SGL I/O buffers and drains into I/O buffers pointed to by destination SGL I/O buffers. Destination SGL is resident in host memory.
8. **S2C**: System-to-PCIe Card. Data transfer from host I/O buffers (source) to EP I/O buffers (destination).
9. **C2S**: PCIe Çard-to-system. Data transfer from EP I/O buffers (source) to host I/O buffers (destination).

Design Goals

1. To provide a driver to control the Expresso DMA facilitating I/Os and other features specific to Expresso DMA.
2. To present the application drivers a set of APIs that enable the drivers to perform high speed I/Os between host and EP.
3. To abstract the inner working of the Expresso DMA such that it can be treated as a *black box* by the application drivers through the APIs.
4. To create a common driver for different scenarios wherein the source and destination SGL can be resident on host, EP, or both host and EP. In this document the scenario of all SGL on host memory is explored.

All SGL Qs on Host Model

In this model of operation all eight SGLs corresponding to the four DMA channels (two lists per channel) are resident in host memory. Note that a real-world use case might not require all four channels to be operational. For every channel, one list is used to point to I/O buffers in host memory while the other list is used to point to I/O buffers or FIFO inside EP memory. Based on the application logic's requirement each channel (and two SGLs corresponding to the channels) is configured as IN/OUT. This means an Ethernet application can configure channel 0 to send Ethernet packets from host to EP (say EP is Ethernet HBA card) for transmitting to the external world (S2C direction) while channel 1 is configured to get incoming Ethernet packets from the external world into host memory (C2S direction).

In this model the application driver must be aware of the memory map of the PCIe EP as it provides the address of source/destination I/O buffers/FIFO in the EP. This model is typically used where there is no processor (software logic) or hardware logic in the EP that is aware of Expresso DMA's SGL format and operating principal.

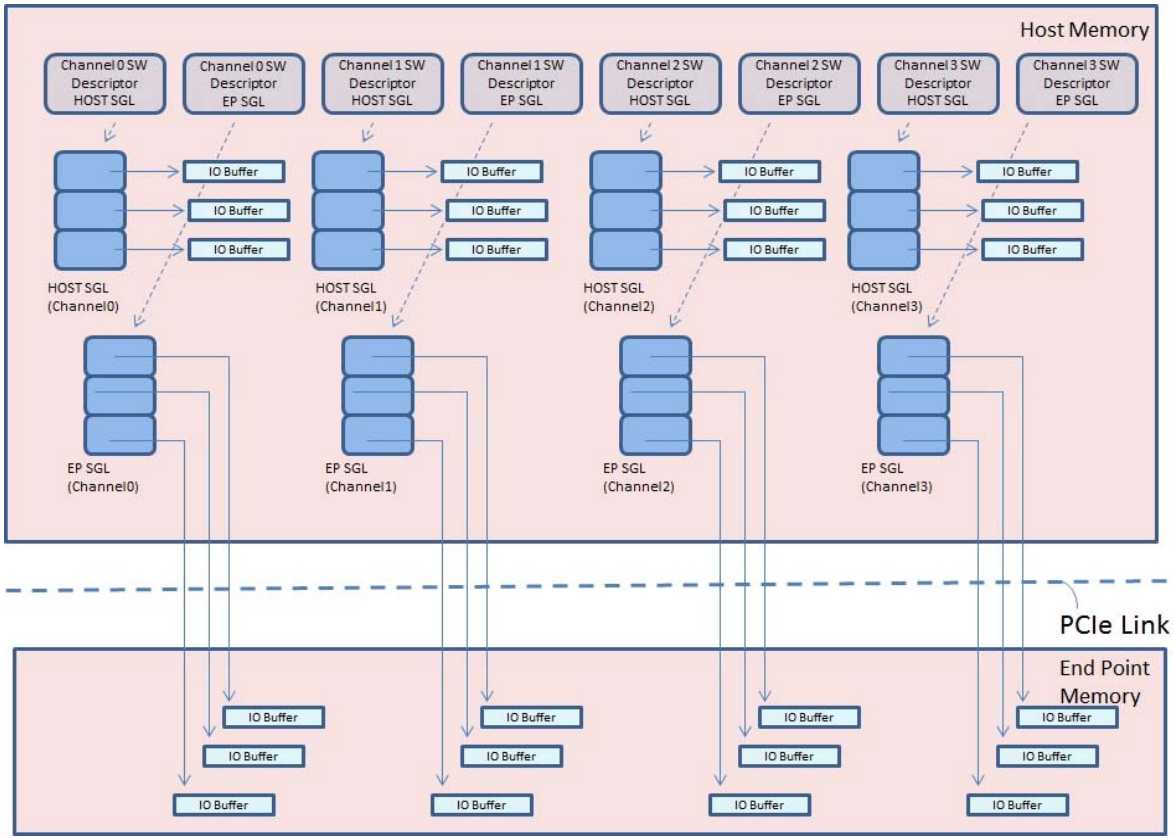


Figure 5-6: **All SGL-Qs Managed by Host Model**

As shown in Figure 5-6, the SGLs consists of:

1. Descriptors: There are eight SGL descriptors instantiated (assuming all four channels are in use). One pair of SGL descriptors must be created for each channel (host SGL and EP SGL) to use. The driver creates an SGL corresponding to each SGL descriptor when the SGL descriptor is created by the application driver using an API call. The application driver passes a pointer to the SGL descriptor and APIs when it wants to perform any operation on the SGL (e.g., perform I/O, activate SGL, stop I/O, etc.) after creation of the SGL descriptor. During creation of the host SGL descriptor, the direction parameter passed by the application driver is used by the XDMA driver to determine if the corresponding SGL is a source or destination SGL.
2. Host SGL: There is one host SGL per channel and it is used by the application driver by invoking APIs to perform operations on the SGL (e.g., perform I/O, activate SGL, stop I/O, etc.). Host SGL list elements are used to point to I/O buffers that are the source or sink (destination) of data depending on the direction in which the channel is used for data transfer. In the S2C direction, the host SGL elements point to source I/O buffers and in the C2S direction the host SGL elements are used to point to the sink (destination) I/O buffers. The I/O buffers are resident on the host. An important attribute of the host SGL,

loc_axi, when set to *false* indicates to the Expresso DMA that the I/O buffer being pointed to by the SGL element is present on the host.

3. EP SGL: There is one EP SGL per channel and it is used by the application driver by invoking APIs to perform operations on the SGL (e.g., perform I/O, activate SGL, stop I/O, etc.). EP SGL list elements are used to point to I/O buffers that are the source or sink (destination) of data depending on the direction in which the channel is used for data transfer. In the S2C direction, the EP SGL elements point to source I/O buffers (or FIFOs) and in the C2S direction point to sink (destination) I/O buffers (or FIFOs). The I/O buffers/FIFOs are resident on the EP. An important attribute of the EP SGL, loc_axi, when set to *true* indicate to the Expresso DMA that the I/O buffer/FIFO being pointed to by the SGL element is present on the EP, even though the SGL is actually resident in host memory.
4. I/O Buffer/FIFO: These are the memory locations from/to which the Expresso DMA actually performs data transfers. The Expresso DMA makes use of source and destination SGL elements and the loc_axi flag in these elements to determine the location of the source and destination of I/O buffers/FIFOs for data transfer. The XDMA driver sets up the SGL and marks the loc_axi flag appropriately to facilitate correct data transfer.

XDMA Driver Stack and Design

The XDMA driver provides APIs to the application drivers. Most of these APIs require passing of pointers to the SGL descriptors. The application driver must create two SGL descriptors for each channel it intends to use.

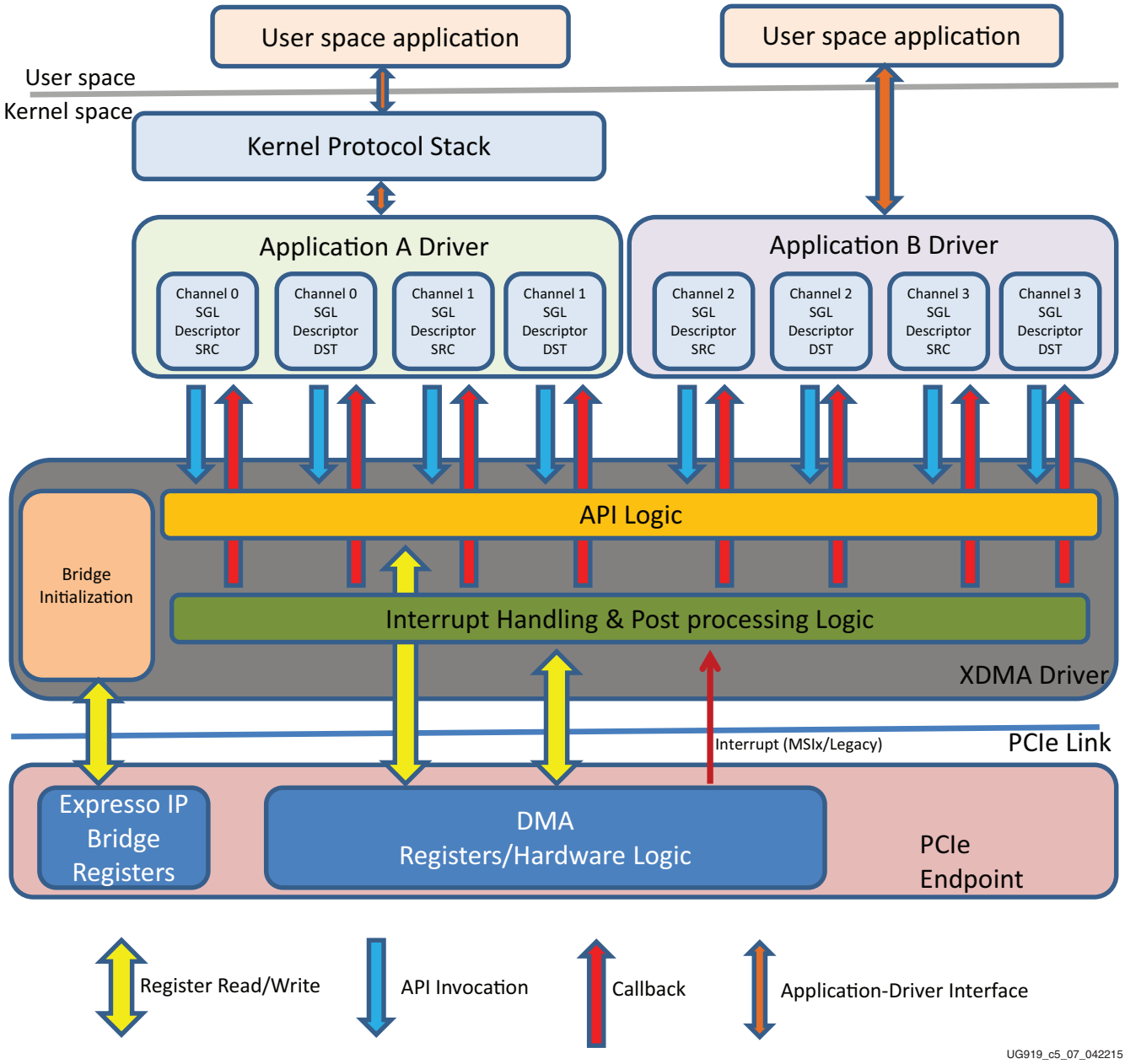


Figure 5-7: **XDMA Driver Stack and Design**

As shown in Figure 5-7, the XDMA driver consists of:

1. API Logic: APIs are exported to application driver. This logic executes in context of the calling process. APIs can be invoked from process and bottom half context.
2. Interrupt Handling and Post Processing Logic: This logic performs post processing after an I/O buffer is submitted and the Expresso DMA has processed it. For source buffers, this logic kicks in when the data has been taken from the source buffers and copied into the destination buffers. For destination buffers, this logic kicks in when data is copied from the source buffers. Post processing logic performs cleanups after a source or destination buffer is utilized (as source/sink of data) by DMA. Post processing logic invokes callbacks which have been provided by the application driver.
3. Bridge Initialization: The Expresso IP contains bridge for protocol conversion (AXI to PCIe and PCIe to AXI) which needs initialization. The details on bridge initialization are documented under AXI-PCIe Bridge, page 45.

Steps to Perform I/O using the XDMA Driver in Linux

The following steps must be followed for an application driver to make use of an XDMA driver to perform data transfer between host and EP. Passing of an OUT direction parameter indicates to the XDMA driver that the SGL is a source-side SGL. Passing of an IN parameter indicates to the XDMA driver that the SGL is a destination-side SGL.

S2C I/O

Channel 0 is used in these steps. Assume that data from the host I/O buffer pointed to by a virtual address of `0x12340000` must be transferred to a fixed AXI address location of `0xC0000000` in the EP.

1. Create a source SGL SW descriptor using XDMA API xlnx_get_dma_channel (). Specify the direction as OUT. Let us call this descriptor `s2c_chann0_out_desc`. This is the descriptor for the source SGL.
2. Allocate Qs for the SGL using XDMA API `xlnx_alloc_queues ()`.
3. Activate the DMA channel using XDMA API `xlnx_activate_dma_channel ()`.
4. Create a destination SGL SW descriptor using XDMA API `xlnx_get_dma_channel ()`. Specify the direction as IN. Let us call this descriptor `s2c_chann0_in_desc`. This is the descriptor for the destination SGL. Repeat steps 2 and 3.
5. Invoke XDMA API `xlnx_data_frag_io ()` passing parameters `s2c_chann0_out_desc`, `0x12340000`, address type as VIRT_ADDR. A pointer to a callback function can also be passed (optional).
6. Invoke XDMA API `xlnx_data_frag_io ()` passing parameters `s2c_chann0_in_desc`, `0xC0000000`, address type as EP_PHYS_ADDR. A pointer to a callback function can also be passed (optional).

7. These steps cause the data to be drained from the host I/O buffer pointed by `0x12340000` to an EP AXI address location of `0xC000000`. Optional callbacks are invoked for both SGL descriptors so that the application drivers can do the necessary cleanups after data transfer.

C2S I/O

Channel 0 is used in these steps. Assume data from the EP I/O buffer pointed to by an AXI address of `0x40000` must be transferred to a host I/O buffer pointed to by a physical address of `0x880000`.

1. Create a destination SGL SW descriptor using XDMA API `xlnx_get_dma_channel ()`. Specify the direction as IN. Let us call this descriptor `c2s_chann1_in_desc`. This is the descriptor for the destination SGL.
2. Allocate Qs for the SGL using XDMA API `xlnx_alloc_queues ()`.
3. Activate the DMA channel using XDMA API `xlnx_activate_dma_channel ()`.
4. Create a source SGL SW descriptor using XDMA API `xlnx_get_dma_channel ()`. Specify the direction as OUT. Let us call this descriptor `c2s_chann1_out_desc`. This is the descriptor for the source SGL. Repeat steps 2 and 3.
5. Invoke XDMA API `xlnx_data_frag_io ()` passing parameters `c2s_chann1_in_desc`, `0x880000`, address type as PHYS_ADDR. A pointer to a callback function can also be passed (optional).
6. Invoke XDMA API `xlnx_data_frag_io ()` passing parameters `c2s_chann1_out_desc`, `0x40000`, address type as EP_PHYS_ADDR. A pointer to a callback function can also be passed (optional).
7. These steps cause the data to be drained from the EP I/O buffer pointed to by `0x40000` to a host I/O buffer physical address location of `0x880000`. Optional callbacks are invoked for both SGL descriptors so that the application drivers can do the necessary cleanups after data transfer.

Steps to Perform DMA Transfers using the XDMA Driver in Windows

The following steps must be followed to enable an application driver to use an XDMA driver to perform data transfer between host and EP.

Application Driver Registration

The application driver must register itself with the XDMA driver to be able to communicate through the DMA channel of its choice. The DMA driver exports an interface for other drivers to enable this registration process.

The application driver must open the interface exported by XDMA driver and invoke the DmaRegister along with the relevant information such as number of buffer descriptors, coalesce count, and callback routines to be invoked when transfers are complete.

If the channel requested by the application driver is free, the DMA driver accepts the registration of the Application Driver and provides a handle to it for initiating transfers.

Initiating Transfers

The application driver can initiate transfers using the handle provided by the XDMA driver after registration. Using XDMADataTransfer API, an application driver can update either the source or destination SGL of the channel by specifying the relevant value in the QueueToBeServiced parameter.

Callback Invocation

After completion of either a transmit or receive DMA transaction for a particular channel, relevant callback provided by the application driver during registration phase is invoked. The application driver can take an appropriate action based on the information provided by the DMA driver, such as number of bytes completed or notification of errors (if any) during the transfer.

Driver Configuration for Linux

Note: This driver configuration is applicable only for Linux and not for Windows.

The driver for Expresso DMA is developed to be highly configurable depending on the design of the end application using the Expresso DMA. Supported configurations are detailed in Table 5-3.

Table 5-3: **Supported Configurations**

Operating Mode	Details	Driver Macro to Enable (ps_pcie_pf.h)
No Endpoint Processor (Firmware) based DMA Operation	• In this mode, there is no firmware operation on the scatter gather queues of the Expresso DMA inside the PCIe Endpoint. The Expresso driver running on the host manages source as well as destination-side scatter gather queues. The host driver must be aware of the source/destination address on the Endpoint from/to which data is to be moved into host/Endpoint. • In this mode the driver must be built for deployment on the host only.	PFORM_USCALE_NO_EP_ PROCESSOR
Endpoint Hardware Logic-based DMA Operation	• In this mode, hardware logic inside the Endpoint operates the source/destination side scatter gather queues of a DMA channel (HW-SGL), while the host side DMA driver manages the destination/source-side scatter gather queue of that channel. • In this mode the Endpoint hardware logic is aware of the Expresso DMA. • In this mode the driver must be built for deployment on the host only.	HW_SGL_DESIGN

The macros PFORM_USCALE_NO_EP_PROCESSOR and HW_SGL_DESIGN are mutually exclusive (only one can be enabled). Enabling more than one of the above can lead to unpredictable results.

MSIx Support

The driver, when built for the host machine, supports MSIx mode. To disable MSIx mode, comment out macro `USE_MSIX` in file `ps_pcie_dma_driver.h`.

API List

APIs are provided by the Expresso DMA (XDMA) driver to facilitate I/Os over the Expresso DMA channels. Refer to Appendix D, APIs Provided by the XDMA Driver in Linux for more information on the APIs for Linux and Appendix E, APIs Provided by the XDMA Driver in Windows for more information on the APIs for Windows.

User Space Application Software Components

The user space software component comprises the application traffic generator block and the GUI. The function of the traffic generator block is to feed I/O buffers to the XDMA driver for demonstrating raw I/O performance capable by the Expresso DMA.

The user space software interfaces with the XDMA driver through an application data driver in the I/O path and directly with the XDMA driver to fetch performance data from hardware performance monitoring registers.

The application data driver provides a character driver interface to the user space software components.

Graphical User Interface

The user-space GUI is a Java based graphical user interface that provides the following features:

- Installs/uninstalls selected mode device drivers.
- Gathers statistics such as power, PCIe reads/writes, and AXI reads/writes from the DMA driver and displays it on the JAVA GUI.
- Controls test parameters such as packet size and mode (S2C/C2S).
- Displays PCIe information such as host system credits, etc.
- Graphical representation of performance numbers.
- JNI layer of the GUI interacts with the application traffic generator and driver interface. It is written in C++.

Application Traffic Generator

The application traffic generator is a multi-threaded application, which generates traffic. It constructs a packet according to the format of the application driver:

- *TX thread* for allocating and formatting packets according to test parameters.
- *TX done thread* polls for packet completion.
- *RX thread* provides buffers to the DMA ring.
- *RX done* thread polls for receipt of packets.
- Apart from the above threads there will be *main thread* which spawns all of these threads according to test parameters from the GUI.

GUI and Application Traffic Generator Interface

In Linux, the GUI and application traffic generator communicate through UNIX sockets. The GUI opens a UNIX socket and sends messages to the application traffic generator. The main thread of the application traffic generator waits for messages from the GUI and spawns threads according to test mode.

In Windows, the application traffic generator is a part of the GUI itself and hence there is no requirement for any socket communication. It is implemented in a dynamic linked library which is invoked by the GUI when it starts. GUI spawns threads according to the test mode.

GUI and XDMA Driver Interface

The XDMA driver polls power monitor and performance monitor statistics periodically and stores the statistics in internal data structures. The GUI periodically polls these statistics from the XDMA driver by opening the driver interface and sending IOCTL system calls.

Application Traffic Generator and Application Driver Interface

According to test parameters from GUI, the application traffic generator spawns threads which prepare packets. These packets are sent to the application driver by opening the application driver and issuing write/ read system calls.

Control and data path interfaces are shown in Figure 5-8.

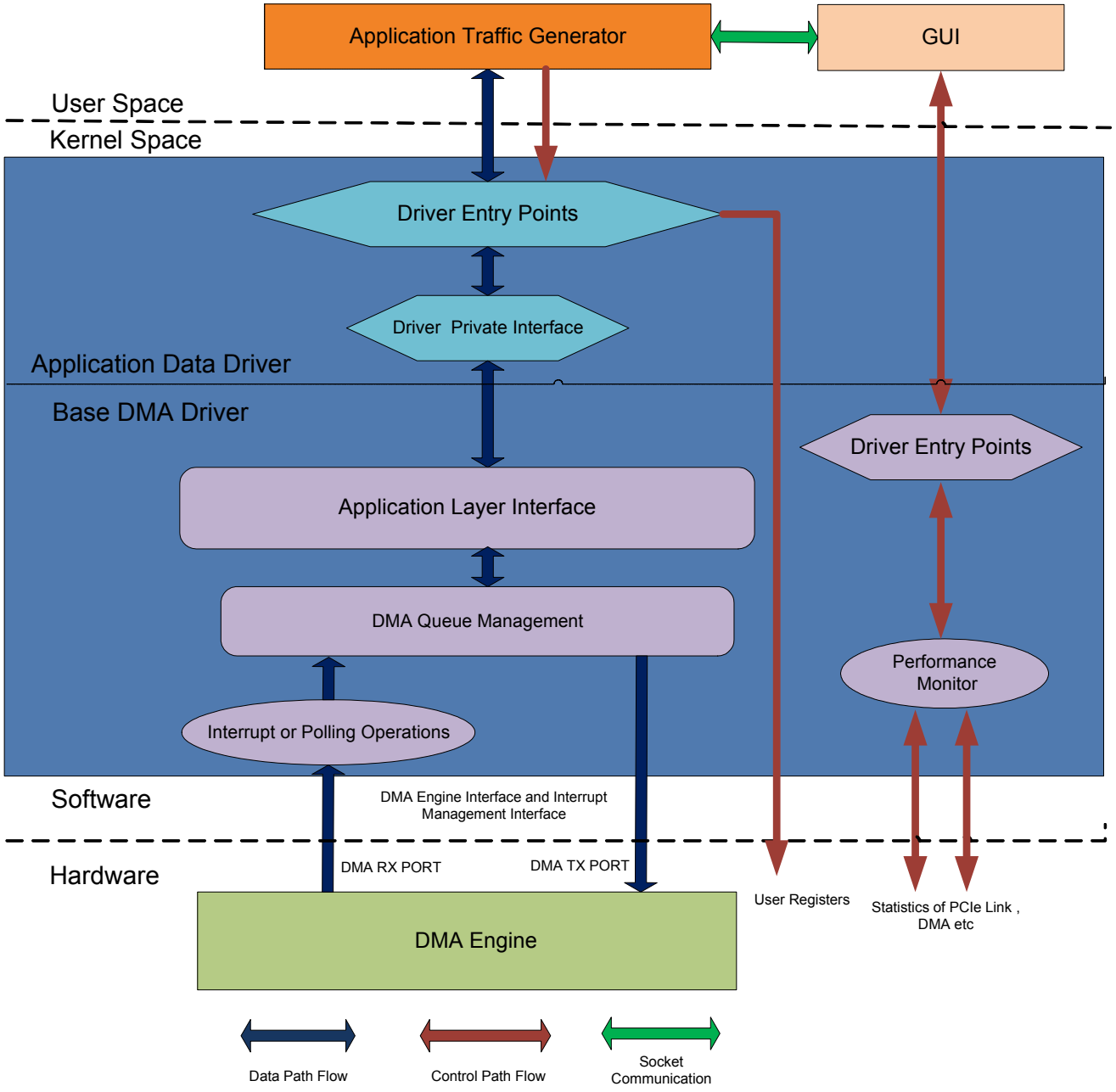


Figure 5-8: **Control and Data Path Interfaces**

Application Data Driver

The user space application opens the application driver interface and operates write/read system calls.

The application driver performs the following steps for each direction of data transfer.

S2C

- Receives a WRITE system call from the user application along with the user buffer.
- Pins the user buffer pages to protect it from swapping.
- Converts the pages to physical addresses.
- Stores all page address and buffer information in a local data structure for post-processing usage.
- Calls relevant data transfer API of the XDMA driver for transfer of packet.
- In the process of registration with the XDMA driver, a call back function is registered and invoked when the XDMA driver receives the completion of the packet.
- The completed packet is queued in the driver.
- The application periodically polls for completion of the packet by reading the driver's queues.

C2S

- The application sends free buffers from user space by issuing a READ system call.
- Pins the user buffer pages to protect it from swapping.
- Converts the pages to physical addresses.
- Stores all page address and buffer information in a local data structure for post-processing usage.
- Calls relevant data transfer API of the XDMA driver for transfer of packet.
- In the process of registration with the XDMA driver, a call back function is registered and invoked when the XDMA driver receives a packet.
- The received packet is queued in the driver.
- The application periodically polls for receipt of packet by reading the driver's queues.

Reference Design Modifications

A video accelerator design is provided as a part of the reference design, which is an extension of the base design. The following section describes the setup procedure required to test the video accelerator design.

IMPORTANT: *The pre-built user extention design can be tested only on Linux and not on Windows. There is no support for the user extension design in the Windows platform.*

Note: Use the `trd02_user_etxn.bit` file in the `ready_to_test` folder of the reference design zip file to configure the FPGA.

Setup Procedure for the Video Accelerator Design

IMPORTANT: *Follow the preliminary setup procedures in Chapter 3, Bringing Up the Design, up to Step 2 of Testing the Reference Design. Then proceed as follows:*

1. Copy the software directory and quickstart.sh script from the reference design zip file to /tmp.
2. Login as super user by entering `su` on the terminal.
3. Enter `cd /tmp`.
4. Enter `chmod +x quickstart.sh`.

   Note: The script installs VLC player and associated packages for the first time, and then invokes the TRD Setup installer screen of the GUI as shown in Figure 3-10, page 26. This step takes some time, as this step involves VLC player installation.
5. Performance mode is selected under the **AXI-MM Dataplane** Design Selection by default. Change the Driver Mode Selection for AXI-MM Dataplane Design to **Video Accelerator** mode, as shown in Figure 5-9.

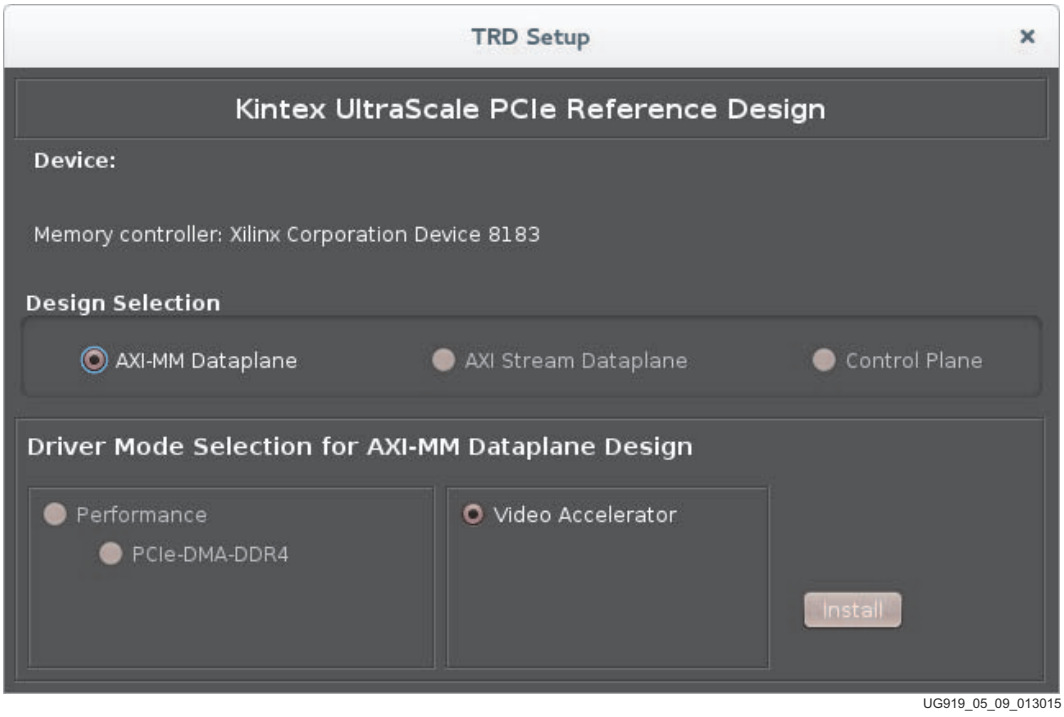


Figure 5-9: **Installer Screen for Video Accelerator Design**

6. Click **Install**. This installs the drivers for running the video accelerator design (shown in Figure 5-10).

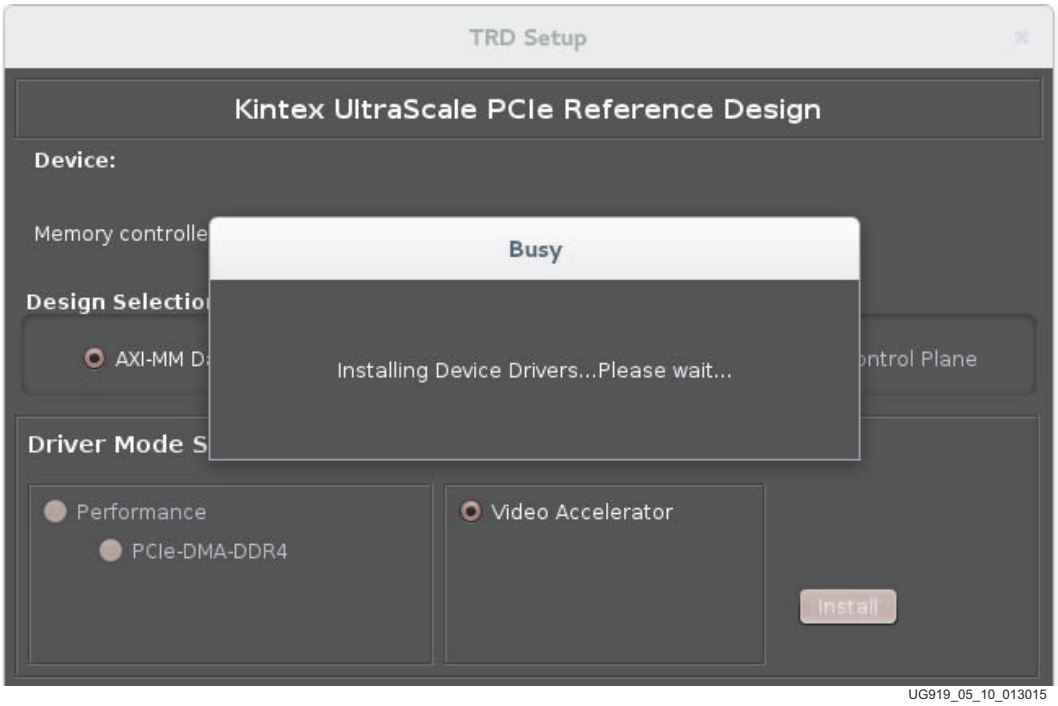


Figure 5-10: **Installing Device Drivers for Video Accelerator Mode**

7. After the device drivers are installed, the Design Control and Monitoring Interface GUI window pops up, as shown in Figure 5-11.

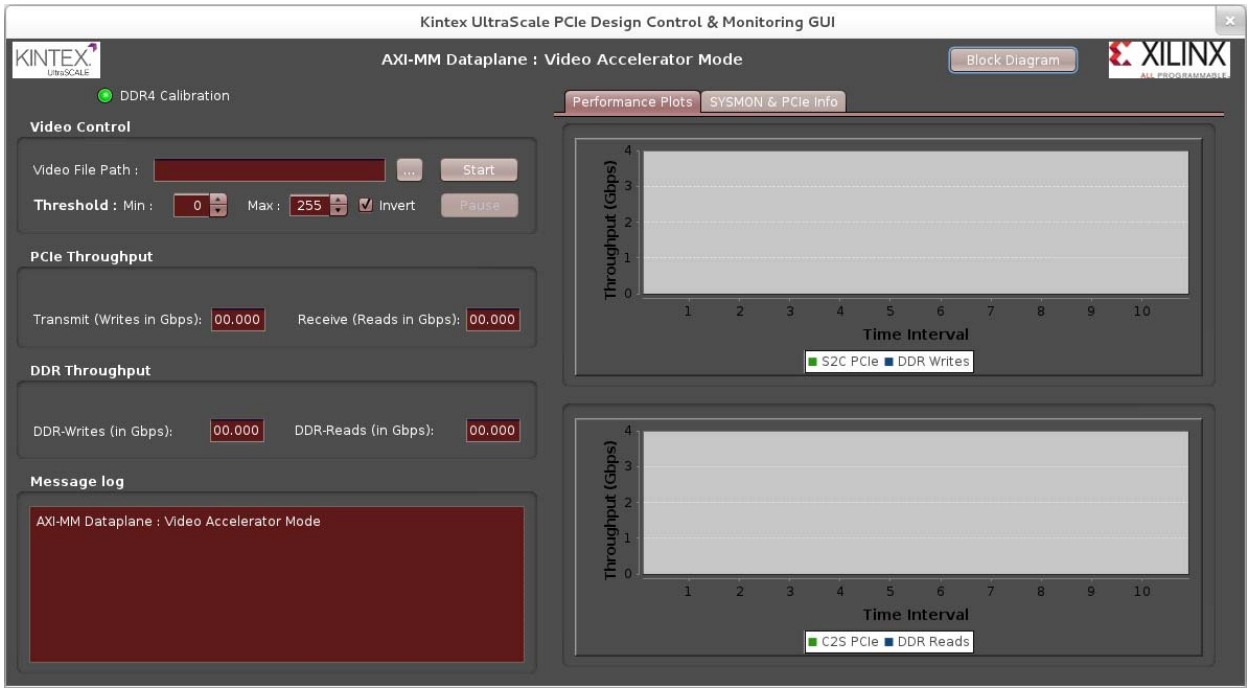


Figure 5-11: **Design Control and Monitoring GUI**

8. Download the full high definition (HD) video.
9. Copy the downloaded video to a USB stick or pendrive.
10. In the GUI, choose the path to the downloaded video file (browse to the USB stick).
11. Click **Start**. This opens the VLC player's Privacy and Network Policies window. Click **OK** to see the source video and the image processed video in a video panel as shown in Figure 5-12.

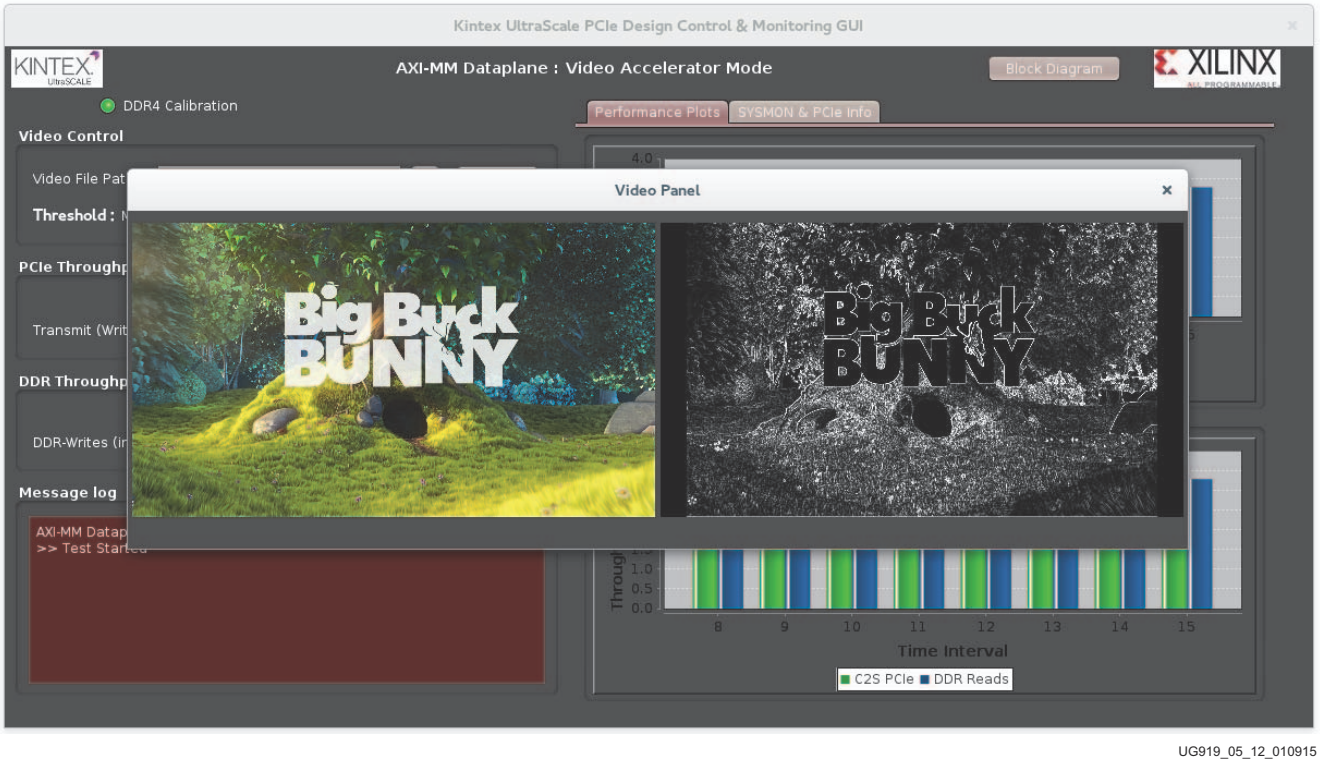


Figure 5-12: **Sobel Operation in Hardware**

12. While the video frames are transmitted, two options are available, as shown in Figure 5-13:
    a. Min and Max Threshold (valid range is 0 to 255)
    b. Invert (inverts the pixels)

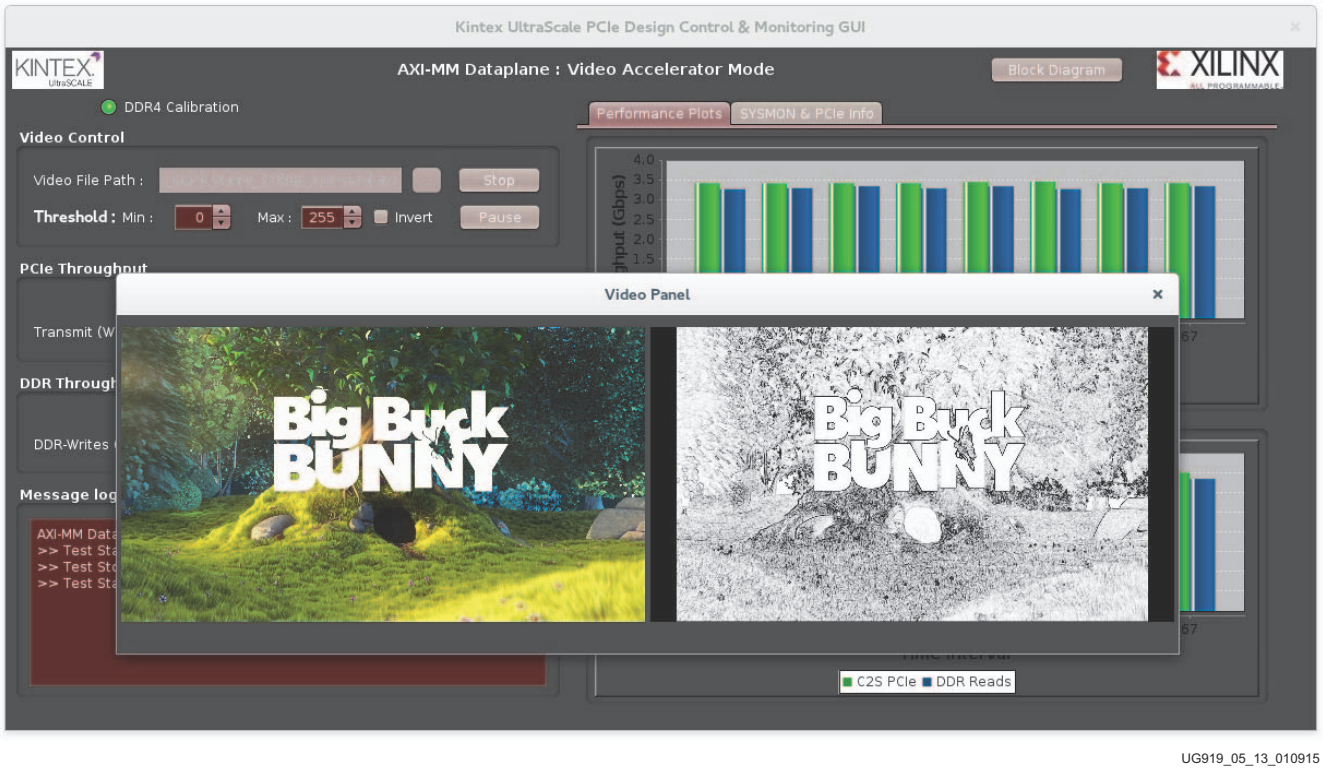


Figure 5-13: **Sobel Filter Options**

13. Click **Block Diagram** in the top right corner of the screen to display the block diagram of the design, as shown in Figure 5-14.

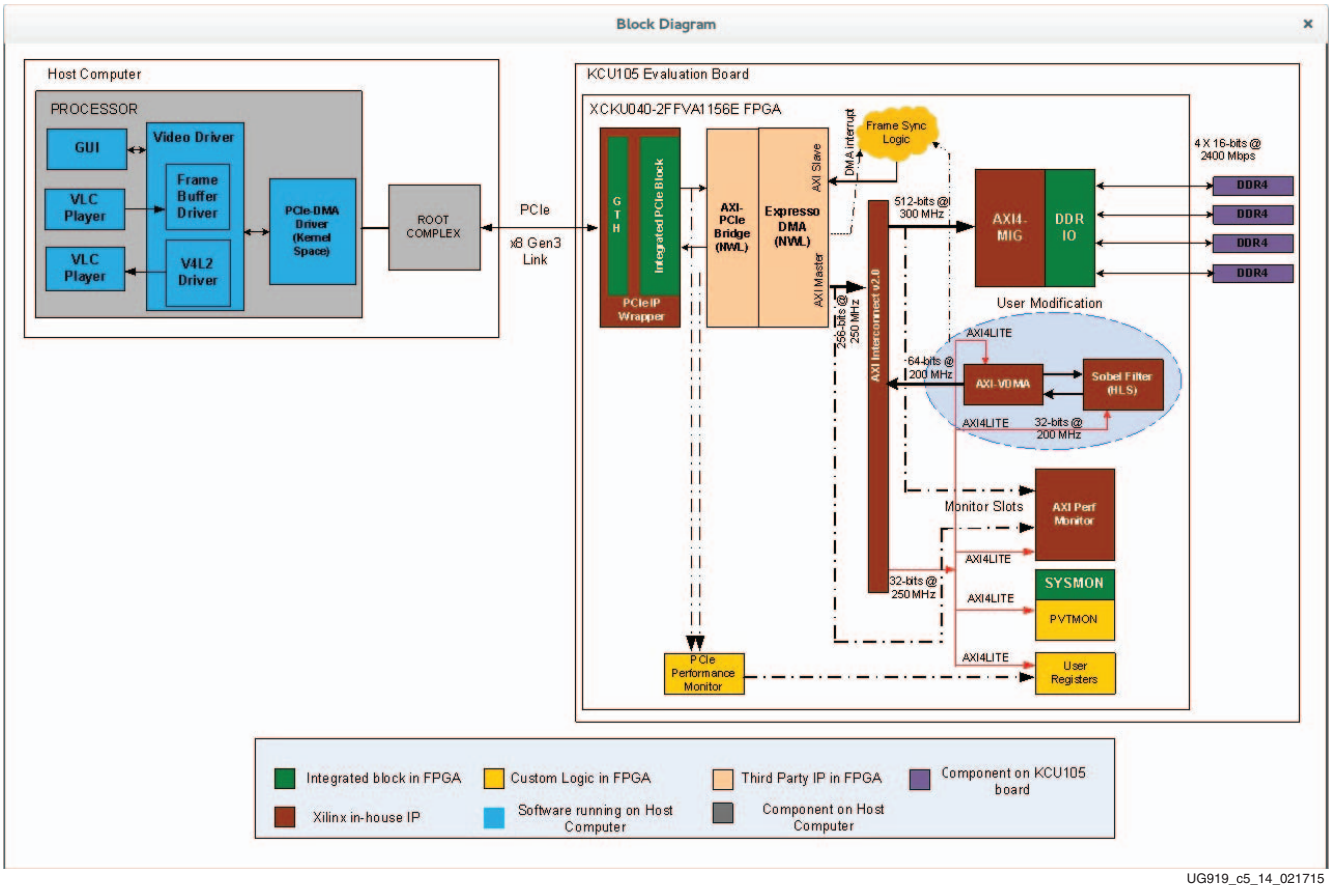


Figure 5-14: **Video Accelerator Design, Block Diagram**

14. Close the block diagram and click **Stop** in the Video Control panel to stop the test. Click on **x** mark on top right hand corner to close the Video Accelerator GUI. The drivers are uninstalled (click **Yes** at the prompt as shown in Figure 5-15) and the TRD Setup screen (Figure 3-10, page 26) is displayed.

Figure 5-15: **Uninstalling Device Drivers for Video Accelerator Design**

Prebuilt Modification: Adding Video Accelerator Block

This section describes how the base design can be modified to build a PCI Express based video accelerator design.

This design demonstrates the use of an endpoint for PCI Express as an accelerator card, offloading computer-intensive tasks from the host processor to the accelerator card, resulting in freeing up of host CPU bandwidth. In this model, the host driver manages card memory and is responsible for setting up the accelerator on the card for processing.

In the example provided, edge detection is offloaded to the endpoint. DDR4 memory on the endpoint is used as a frame buffer memory that holds both the processed and unprocessed video frames. The accelerator block consists of AXI Video DMA (VDMA) and a Sobel Edge detection block. Both these blocks are controlled by host software.

The video accelerator design uses 1080p video at 24f/s.

AXI-VDMA and Sobel Filter control interfaces are added to the AXI4Lite interconnect with the offsets listed in Table 5-4.

Table 5-4: **AXI LITE Slaves Address Decoding for Video Path**

AXILITE SLAVE	Address Range	Size
AXI VDMA	0x44A20000 – 0x44A2FFFF	64K
SOBEL Filter	0x44A30000 – 0x44A3FFFF	64K

The PCIe-based AXI memory-mapped data plane video accelerator design block diagram is shown in Figure 5-16.

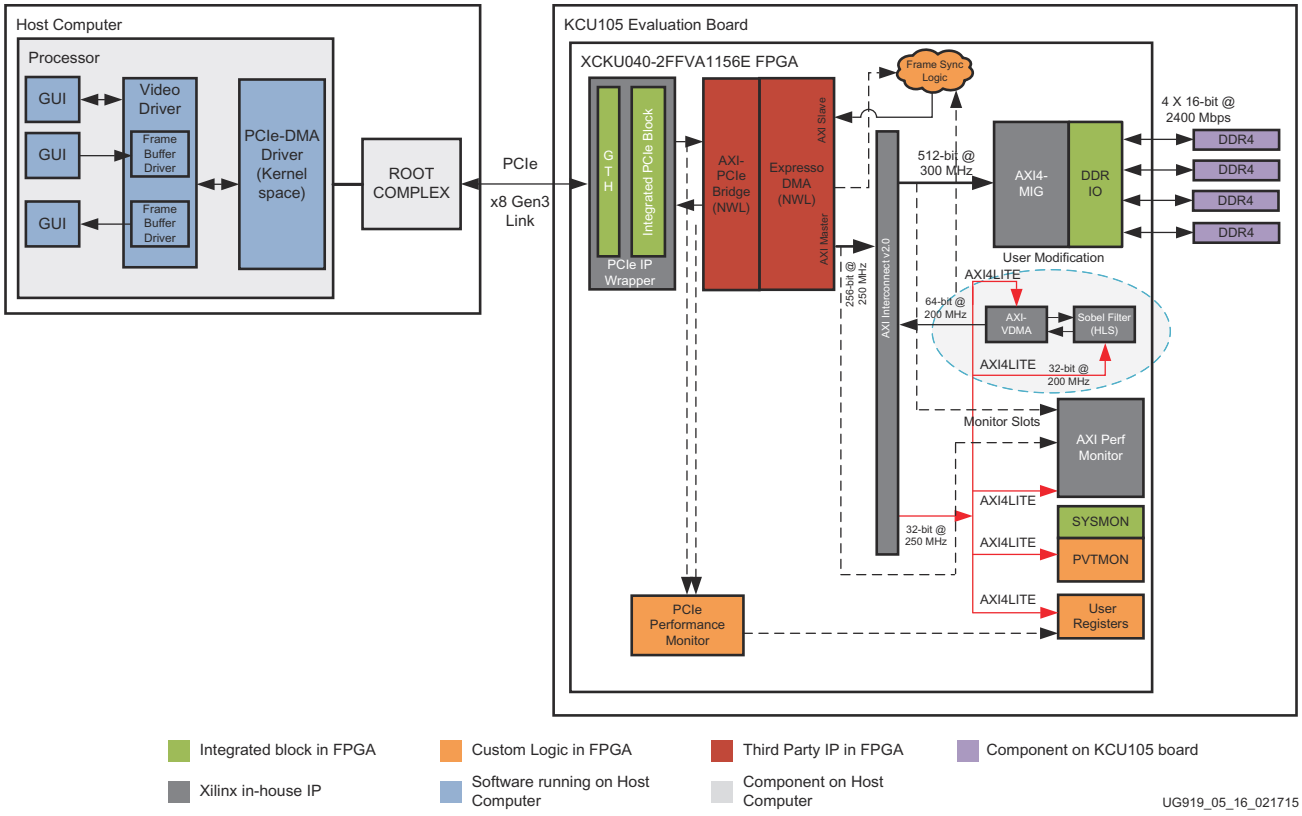


Figure 5-16: **PCIe-based AXI Memory-Mapped Data Plane – Video Accelerator Design**

AXI-VDMA

The AXI VDMA provides high bandwidth direct memory access between memory and AXI4-Stream video interface (DDR4 and Sobel filter, respectively, in this case).

This IP is used in following mode:

1. S2MM and MM2S channels enabled
2. Circular mode of operation controlled by fsync
3. FSYNC options:
   a. For S2MM channel, use `s2mm_tuser`

b. For MM2S channel, use `mm2s_fsync` port (driven by frame sync logic based on doorbell interrupt from host to endpoint)

4. Three frame buffers in each direction

VDMA Operation

VDMA is configured to use three frame buffers in each S2MM and MM2S direction. These frame buffers reside in card memory and have pre-defined static addresses. In the MM2S direction, the VDMA reads video data from frame buffers allocated to the MM2S direction one after the other. VDMA is programmed in a circular mode of operation, so VDMA reads the frame from the first buffer after it has finished reading the third. The same approach is followed in the S2MM direction. The S2MM and MM2S channels of VDMA are configured for the free running mode with the external FSYNC signal controlling video timing. VDMA fetches frames from predefined addresses in DDR4 (defined at VDMA initialization time) and passes them on to the Sobel filter. The FSYNC output from VDMA propagates over to the Sobel filter (over the tuser port of the streaming interface), which maintains the video frame synchronization.

The static buffer addresses (pointers to card DDR4 memory for buffers) can be programmed in VDMA during the initialization phase. All VDMA specific programming will be done in the initialization phase. Refer to Appendix B, VDMA Initialization Sequence for information on the VDMA initialization sequence followed in the design.

Frame synchronization information is needed to start VDMA operations after the frame buffer is transferred from host memory to card memory. This information is provided by the host software driver (on STAD-Q completion) through an AXI interrupt by programming the AXI_INTERRUPT_ASSERT register.

See *LogiCORE IP AXI Video Direct Memory Access Product Guide* (PG020) [Ref 10] for more information.

Frame Synchronization Logic

The frame synchronization logic does the following:

- Generates FSYNC to start VDMA operation for each frame in the MM2S direction. FSYNC generation is based on a software-generated interrupt (via the int_dma port of the Expresso DMA) and mm2s_all_lines_xfred (which indicates that VDMA has finished reading the frame from DDR4).
- Clears the AXI_INTERRUPT_STATUS register, so that software driver can raise the next interrupt, after the frame has made it to the card memory from host memory.
- In the S2MM direction, generates an interrupt per frame to the host (PCIe domain), by writing to the PCIE_INTERRUPT_ASSERT register. This interrupt is generated after VDMA has finished writing the frame into DDR4 (this is signaled by VDMA by asserting s2mm_all_lines_xfred).

The following sequence is followed:

1. Software sets up AXI-VDMA at initialization with three buffer addresses in both the S2MM and MM2S directions. These addresses are fixed in DDR4 memory each of 8 MB size, contiguous.
2. Other registers in AXI-VDMA are programmed and used in FSYNC mode in the MM2S direction and TUSER mode in the S2MM direction.
3. After a video frame is transferred by DMA into DDR4 memory from host memory, software writes to the doorbell register of that channel to initiate an interrupt in the AXI domain. FSYNC is issued to the VDMA by frame synchronization logic after receiving an interrupt in the AXI domain. VDMA asserts an indication on the axi_vdma_tstvec[1] (mm2s_all_lines_xfred) signal when the frame has been read from DDR4 memory and presented on the streaming interface to the Sobel block. If there is another interrupt from host before the VDMA has finished reading the frame from card DDR4 memory, the frame synchronization logic latches onto the interrupt until it receives an indication from VDMA.
4. For S2MM transfers, VDMA provides axi_vdma_tstvec[32] which results in s2mm_all_lines_xfred. The s2mm_all_lines_xfred port gets asserted after the line transfer ends on the stream interface. An interrupt is generated by the frame synchronization logic after every video frame is written into DDR4, by writing into the doorbell register in the PCIe domain.

Sobel Edge Detection Filter

The AXI Sobel filter is an HLS IP. It operates in free running mode. The filter coefficients are set by the host processor during initialization. The low/high threshold and inversion can be programmed dynamically, if desired. The GUI, which is the control and monitor interface of the design, provides options for changing the low and high threshold values of the filter, and also to invert the output of the Sobel filter.

The Sobel interrupt output is not used, as AXI-VDMA raises an interrupt on completion of S2MM operation which can be used to signal processed frame availability in card memory for upstream transmission.

Refer to Appendix C, Sobel Filter Registers for details on Sobel filter register programming.

Rebuilding Hardware

A prebuilt design script is provided for the user modification design which can be run to generate bitstream.

Refer to Chapter 4, Implementing and Simulating the Design to get details on how to re-build the video accelerator design.

Software for the Video Application

The kernel driver that registers with the frame buffer and V4L2 subsystems supports video transfers from system-to card and card-to-system. It interacts with the DMA driver using the APIs (see Software, page 53) to initiate transfers.

The two salient features of the kernel driver are associated with video transfers:

1. The driver register with frame buffer framework provides a standard transmission interface for the VLC media player for video media. DMA channel 0 is used to achieve S2C transfers wherein the video data is transferred from the host to the card.
2. The same driver gets registered with V4L2 framework to provide a standard receive interface for the VLC media player to display the received video frames. DMA channel 1 is used to achieve C2S transfers wherein the processed video data is transferred from the card back to the host.

S2C Transfers

All S2C transfers are initiated by the VLC media player when a media file is played. The S2C transfer sequence is as follows:

1. The VLC media player extracts the raw video format for each frame of the video file.
2. The raw video frames are sent down to the frame buffer interface exposed by the video kernel driver.
3. The raw video frames are sent over to the FPGA, where the video is processed and sent back to the host.

C2S Transfers

The hardware sends an interrupt after the processed video is available in card memory. All C2S transfers are initiated by the video driver after receiving scratch pad (software) interrupt from hardware. The C2S transfer sequence is as follows:

1. The VLC media player is configured to display the output from the V4L2 interface exposed by the video driver.
2. The VLC media player negotiates with the driver to obtain information regarding the video format and frame rate of the video stream coming in from the V4L2 interface.
3. The VLC media player then waits for actual frames to be available at the V4L2 interface for display.
4. The video driver initiates the transfer of the processed frames from card memory to host memory and V4L2 framework informs the player about the availability of raw video frames for display.
5. The processed frame is then displayed by the VLC media player.

Note: Other than full HD video (1080p), you can also run 720p resolution by defining a macro -DRES_720P in the Makefile under `kcu105_aximm_dataplane/software/linux_driver_app/driver/video_driver`.

VDMA Configuration

Refer to Appendix B, VDMA Initialization Sequence for information on VDMA configuration by the software driver.

Sobel Configuration

The Sobel filter register map is described in Appendix C, Sobel Filter Registers. The graphical user interface allows changing the Sobel filter threshold values and inversion of pixels.

Directory Structure

The directory structure for the PCIe memory-mapped data plane reference design is shown in Figure A-1.

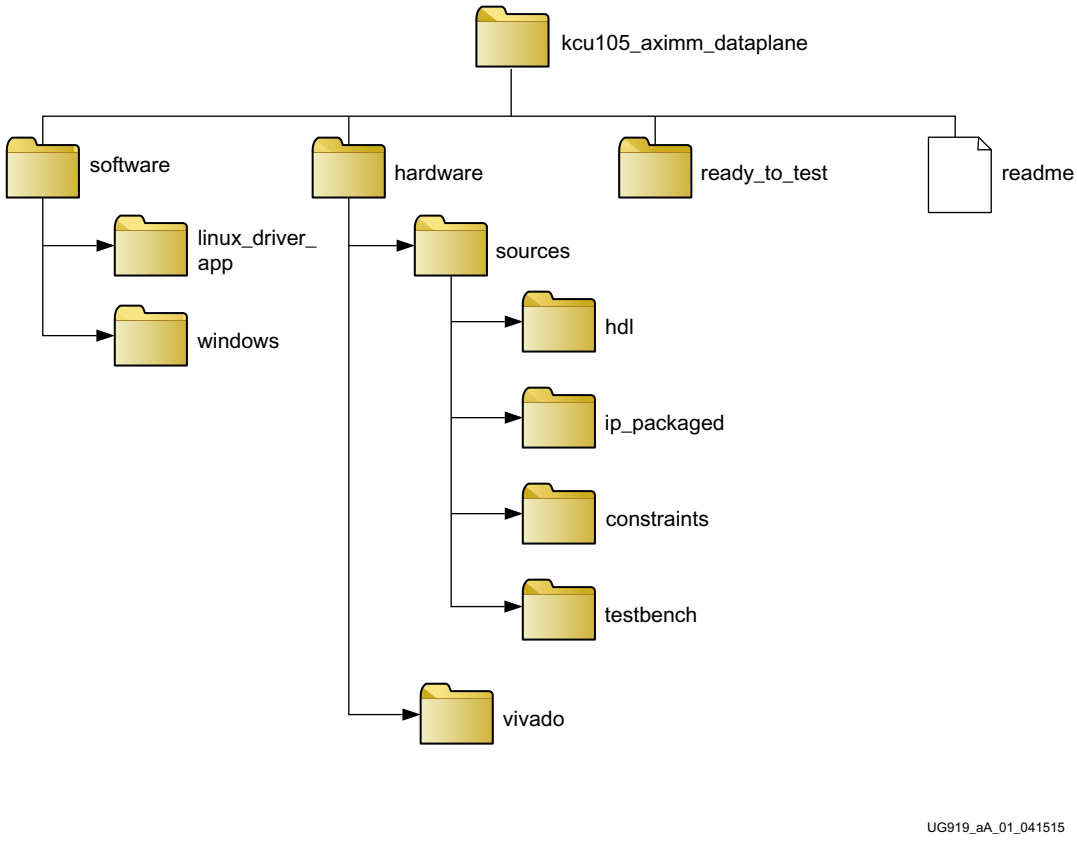


Figure A-1: **Reference Design Directory Structure**

Directory Content Summary

The files and folders contained in the PCIe memory-mapped data plane reference design are shown in Figure A-1. The top-level folder is `kcu105_aximm_dataplane`. For detailed description of each folder, refer to `readme` file which accompanies the reference design zip file

Table A-1: **Directory Structure Details**

Folder	Description
readme	A TXT file that includes revision history information, steps to implement and simulate the design, required Vivado® tool software version, and known limitations of the design (if any).
hardware	Contains hardware design deliverables
sources	
hdl	Contains HDL files
constraints	Contains constraint files
ip_package	Contains custom IP packages
testbench	Contains test bench files
vivado	Contains scripts to create a Vivado Design Suite project and outputs of Vivado runs
ready to test	Contains the bitfile to program the KCU105 PCIe memory-mapped data plane reference design application
software linux_driver_app windows	Contains software design deliverables for Linux and Windows

VDMA Initialization Sequence

Table B-1 identifies the AXI-VDMA registers. See *LogiCORE IP AXI Video Direct Memory Access Product Guide* (PG020) [Ref 10] for the AXI-VDMA register address map and register descriptions.

Table B-1: **AXI-VDMA Registers**

Register Name	Offset	Value
MM2S_VDMACR	0x00	0x0001_0003
MM2S_START_ADDR1	0x5C	0xC000_0000
MM2S_START_ADDR2	0x60	0xC07E_9000
MM2S_START_ADDR3	0x64	0xC0FD_2000
MM2S_FRMDLY_STRIDE	0x58	0x1E00
MM2S_HSIZE	0x54	0x1E00
MM2S_VSIZE	0x50	0x438
S2MM_VDMACR	0x30	0x0001_0043
S2MM _START_ADDR1	0xAC	0xC17B_B000
S2MM _START_ADDR2	0xB0	0xC1FA_4000
S2MM _START_ADDR3	0xB4	0xC278_D000
S2MM _FRMDLY_STRIDE	0xA8	0x1E00
S2MM _HSIZE	0xA4	0x1E00
S2MM _VSIZE	0xA0	0x438

Sobel Filter Registers

The Sobel filter registers are used to configure and control various internal features of the Sobel filter logic. The base address of these registers is `0x44A3_0000`. Register descriptions follow.

Control and Status Register (Offset: 0x0)

Table C-1: **Control and Status Register (Offset: 0x0)**

Bit Position	Mode	Default Value	Description
[31:8], [6:4]	-	-	Reserved
7	RW	0	Auto restart
3	R	1	IP is ready to accept new data
2	R	1	IP is idle
1	Clear on Read	0	Frame processing is done
0	RW	0	Start processing the frame

Number of Rows Register (Offset: 0x14)

Table C-2: **Control and Status Register (Offset: 0x14)**

Bit Position	Mode	Default Value	Description
[31:0]	RW	-	Number of rows in a frame

Number of Columns Register (Offset: 0x1C)

Table C-3: **Number of Columns Register (Offset: 0x1C)**

Bit Position	Mode	Default Value	Description
[31:0]	RW	-	Number of columns in a frame

XR0C0 Coefficient Register (Offset: 0x24)

Table C-4: **XR0C0 Coefficient Register (Offset: 0x24)**

Bit Position	Mode	Default Value	Description
[31:0]	RW	0	XR0C0 coefficient of Sobel filter

XR0C1 Coefficient Register (Offset: 0x2C)

Table C-5: **XR0C1 Coefficient Register (Offset: 0x2C)**

Bit Position	Mode	Default Value	Description
[31:0]	RW	0	XR0C1 coefficient of Sobel filter

XR0C2 Coefficient Register (Offset: 0x34)

Table C-6: **XR0C2 Coefficient Register (Offset: 0x34)**

Bit Position	Mode	Default Value	Description
[31:0]	RW	0	XR0C2 coefficient of Sobel filter

XR1C0 Coefficient Register (Offset: 0x3C)

Table C-7: **XR1C0 Coefficient Register (Offset: 0x3C)**

Bit Position	Mode	Default Value	Description
[31:0]	RW	0	XR1C0 coefficient of Sobel filter

XR1C1 Coefficient Register (Offset: 0x44)

Table C-8: **XR1C1 Coefficient Register (Offset: 0x44)**

Bit Position	Mode	Default Value	Description
[31:0]	RW	0	XR1C1 coefficient of Sobel filter

XR1C2 Coefficient Register (Offset: 0x4C)

Table C-9: **XR1C2 Coefficient Register (Offset: 0x4C)**

Bit Position	Mode	Default Value	Description
[31:0]	RW	0	XR1C2 coefficient of Sobel filter

XR2C0 Coefficient Register (Offset: 0x54)

Table C-10: **XR2C0 Coefficient Register (Offset: 0x54)**

Bit Position	Mode	Default Value	Description
[31:0]	RW	0	XR2C0 coefficient of Sobel filter

XR2C1 Coefficient Register (Offset: 0x5C)

Table C-11: **XR2C1 Coefficient Register (Offset: 0x5C)**

Bit Position	Mode	Default Value	Description
[31:0]	RW	0	XR2C1 coefficient of Sobel filter

XR2C2 Coefficient Register (Offset: 0x64)

Table C-12: **XR2C2 Coefficient Register (Offset: 0x64)**

Bit Position	Mode	Default Value	Description
[31:0]	RW	0	XR2C2 coefficient of Sobel filter

YR0C0 Coefficient Register (Offset: 0x6C)

Table C-13: **YR0C0 Coefficient Register (Offset: 0x6C)**

Bit Position	Mode	Default Value	Description
[31:0]	RW	0	YR0C0 coefficient of Sobel filter

YR0C1 Coefficient Register (Offset: 0x74)

Table C-14: **YR0C1 Coefficient Register (Offset: 0x74)**

Bit Position	Mode	Default Value	Description
[31:0]	RW	0	YR0C1 coefficient of Sobel filter

YR0C2 Coefficient Register (Offset: 0x7C)

Table C-15: **YR0C2 Coefficient Register (Offset: 0x7C)**

Bit Position	Mode	Default Value	Description
[31:0]	RW	0	YR0C2 coefficient of Sobel filter

YR1C0 Coefficient Register (Offset: 0x84)

Table C-16: **YR1C0 Coefficient Register (Offset: 0x84)**

Bit Position	Mode	Default Value	Description
[31:0]	RW	0	YR1C0 coefficient of Sobel filter

YR1C1 Coefficient Register (Offset: 0x8C)

Table C-17: **YR1C1 Coefficient Register (Offset: 0x8C)**

Bit Position	Mode	Default Value	Description
[31:0]	RW	0	YR1C1 coefficient of Sobel filter

YR1C2 Coefficient Register (Offset: 0x94)

Table C-18: **YR1C2 Coefficient Register (Offset: 0x94)**

Bit Position	Mode	Default Value	Description
[31:0]	RW	0	YR1C2 coefficient of Sobel filter

YR2C0 Coefficient Register (Offset: 0x9C)

Table C-19: **YR2C0 Coefficient Register (Offset: 0x9C)**

Bit Position	Mode	Default Value	Description
[31:0]	RW	0	YR2C0 coefficient of Sobel filter

YR2C1 Coefficient Register (Offset: 0xA4)

Table C-20: **YR2C1 Coefficient Register (Offset: 0xA4)**

Bit Position	Mode	Default Value	Description
[31:0]	RW	0	YR2C1 coefficient of Sobel filter

YR2C2 Coefficient Register (Offset: 0xAC)

Table C-21: **YR2C2 Coefficient Register (Offset: 0xA4)**

Bit Position	Mode	Default Value	Description
[31:0]	RW	0	YR2C2 coefficient of Sobel filter

High Threshold Register (Offset: 0xB4)

Table C-22: **High Threshold Register (Offset: 0xB4)**

Bit Position	Mode	Default Value	Description
[31:0]	RW	0	YR2C2 coefficient of Sobel filter

High Threshold Register (Offset: 0xB4)

Table C-23: **High Threshold Register (Offset: 0xB4)**

Bit Position	Mode	Default Value	Description
[31:8]	-	-	Reserved
[7:0]	RW	0	High threshold value of Sobel filter

Low Threshold Register (Offset: 0xBC)

Table C-24: **Low Threshold Register (Offset: 0xBC)**

Bit Position	Mode	Default Value	Description
[31:8]	-	-	Reserved
[7:0]	RW	0	Low threshold value of Sobel filter

Invert Output Register (Offset: 0xC4)

Table C-25: **Low Threshold Register (Offset: 0xC4)**

Bit Position	Mode	Default Value	Description
[31:1]	-	-	Reserved
[0]	RW	0	Invert output of Sobel filter

APIs Provided by the XDMA Driver in Linux

Table D-1 describes the APIs provided by the XDMA driver in Linux.

Table D-1: **APIs Provided by the XDMA Driver in Linux**

API Prototype	Details	Parameters	Return
ps_pcie_dma_desc_t* **xlnx_get_pform_dma_desc (void** *prev_desc, unsigned short vendid, **unsigned short** devid) ;	Returns pointer to an instance of DMA descriptor. XDMA driver creates one instance of DMA descriptor corresponding to each Expresso DMA Endpoint plugged into PCIe slots of system. The Host side API can be called successively passing the previously returned descriptor pointer until all instance pointer are returned. This API is typically called by an application driver (stacked onto the XDMA driver) during initialization. On the Endpoint side this API must be called just once as a single XDMA instance is supported.	**prev_desc** - Pointer to XDMA descriptor instance returned in previous call to the API. When called for the first time NULL is passed **vendid** - PCI vendor Id **devid** - PCI device id **NOTE:** Currently the XDMA driver does not support multiple Endpoints on the Host side, therefor this API is always called with all parameters as 0.	API returns pointer to XDMA software descriptor. This pointer can be used as parameter to other XDMA driver API.
int xlnx_get_dma_channel(**ps_pcie_dma_desc_t** *ptr_dma_desc, **u32** channel_id, **direction_t** dir, **ps_pcie_dma_chann_desc_t** **pptr_chann_desc, f**unc_ptr_chann_health_cbk_no_block** ptr_chann_health**)**;	The API is used to get a pointer to a DMA channel descriptor. XDMA has four DMA channels. This API is typically called during initialization by the application driver to acquire a DMA channel. Once the channel is acquired, it cannot be acquired until it is relinquished.	**ptr_dma_desc** - XDMA descriptor pointer acquired from call to xlnx_get_pform_dma_desc **channel_id** - Channel number **dir** - IN/OUT. Specifies direction of data movement for the channel. On the host system OUT is specified if the channel is used to move data to the Endpoint. On Endpoint side, IN is specified if the channel is used to move data from the host to the Endpoint. **pptr_chann_desc** - Pointer to place holder to store the pointer to the XDMA channel software descriptor populated by the API. The returned channel descriptor pointer is used by other APIs. **ptr_chann_health** - Callback that notifies application driver about changes in state of the XDMA channel or DMA at large. The application driver can choose to keep this NULL. The application driver should check the channel's state to get more information.	API returns XLNX_SUCCESS on success. On failure, a negative value is returned. Check the "/* Xilinx DMA driver status messages */" in the ps_pcie_dma_driver.h for more information on error codes.
int **xlnx_rel_dma_channel (** p**s_pcie_dma_chann_desc_t** *ptr_chann_desc**)**;	The API is used to relinquish an acquired DMA channel.	**ptr_chann_desc** - Channel descriptor pointer to be released. Should have been acquired by an earlier call to xlnx_get_dma_channel.	API returns XLNX_SUCCESS on success. On failure, a negative value is returned. Check the "/* Xilinx DMA driver status messages */" in ps_pcie_dma_driver.h for more information on error codes.

Table D-1: **APIs Provided by the XDMA Driver in Linux *(Cont'd)***

API Prototype	Details	Parameters	Return
int xlnx_alloc_queues(**ps_pcie_dma_chann_desc_t** *ptr_chann_desc, **unsigned int** *ptr_data_q_addr_hi, **unsigned int** *ptr_data_q_addr_lo, **unsigned int** *ptr_sta_q_addr_hi, **unsigned int** *ptr_sta_q_addr_lo, **unsigned** int q_num_elements**);**	Allocates Source/Destination side Data & Status Queues. Based on direction of channel (specified during call to get DMA channel) appropriate Queues are created.	**ptr_chann_desc** - Channel descriptor pointer acquired by an earlier call to xlnx_get_dma_channel. **ptr_data_q_addr_hi** - Pointer to placeholder for upper 32 bits of data queue address. **ptr_data_q_addr_lo** - Pointer to placeholder for lower 32 bits of data queue address. **ptr_sta_q_addr_hi** - Pointer to placeholder for upper 32 bits of status queue address. **ptr_sta_q_addr_lo** - Pointer to placeholder for lower 32 bits of status queue address. **q_num_elements** - Number of queue elements requested. This determines the number of buffer descriptors.	API returns XLNX_SUCCESS on success. On failure, a negative value is returned. Check the "/* Xilinx DMA driver status messages */" in ps_pcie_dma_driver.h for more information on error codes.
int xlnx_dealloc_queues(**ps_pcie_dma_chann_desc_t** *ptr_chann_desc**)**	De-allocates source/destination side data and status queues.	**ptr_chann_desc** - Channel descriptor pointer.	Returns XLNX_SUCCESS on success. On failure, a negative value is returned. Check the "/* Xilinx DMA driver status messages */" in ps_pcie_dma_driver.h for more information on error codes.
int xlnx_activate_dma_channel(**ps_pcie_dma_desc_t** *ptr_dma_desc, **ps_pcie_dma_chann_desc_t** *ptr_chann_desc, **unsigned int** data_q_addr_hi, **unsigned int** data_q_addr_lo, **unsigned int** data_q_sz, **unsigned int** sta_q_addr_hi, **unsigned int** sta_q_addr_lo, **unsigned int** sta_q_sz, **unsigned char** coalesce_cnt, **bool** warm_activate **);**	Activate acquired DMA channel for usage.	**ptr_chann_desc** - Channel descriptor pointer acquired by an earlier call to xlnx_get_dma_channel. **data_q_addr_hi** - Upper 32 bits of data queue address. **data_q_addr_lo** - Lower 32 bits of data queue address. **data_q_sz** - Number of elements in data queue. This is typically same as **q_num_elements** passed in call to xlnx_alloc_queues. **sta_q_addr_hi** - Upper 32 bits of status queue address. **sta_q_addr_lo** - Lower 32 bits of status queue address. **sta_q_sz** - Number of elements in status queue. This is typically same as **q_num_elements** passed in call to xlnx_alloc_queues. **coalesce_cnt** - Interrupt coalesce count. **warm_activate** - Set to True if API is called after an earlier call xlnx_deactivate_dma_channel	API returns XLNX_SUCCESS on success. On failure, a negative value is returned. Check the "/* Xilinx DMA driver status messages */" in ps_pcie_dma_driver.h for more information on error codes.

Table D-1: **APIs Provided by the XDMA Driver in Linux *(Cont'd)***

API Prototype	Details	Parameters	Return
int xlnx_deactivate_dma_channel(**ps_pcie_dma_chann_desc_t** *ptr_chann_desc)	Deactivate activated DMA channel.	**ptr_chann_desc** - Channel descriptor pointer.	API returns XLNX_SUCCESS on success. On failure, a negative value is returned. Check the "/* Xilinx DMA driver status messages */" in ps_pcie_dma_driver.h for more information on error codes.

Table D-1: **APIs Provided by the XDMA Driver in Linux *(Cont'd)***

API Prototype	Details	Parameters	Return
int **xlnx_data_frag_io (** **ps_pcie_dma_chann_desc_t** *ptr_chan_desc, **unsigned char *** addr_buf, **addr_type_t** at, **size_t** sz, **func_ptr_dma_chann_cbk_noblock** cbk, **unsigned short** uid, **bool** last_frag, **void** *ptr_user_data**)**;	API is invoked to DMA a data fragment across the PCIe link. Channel lock has to be held while invoking this API. For multi-fragment buffer, the channel lock must be held until all fragments of the buffer are submitted to DMA. Lock should not be taken if API is to be invoked in callback context.	**ptr_chann_desc** - Channel descriptor pointer. **addr_buf** - Pointer to start. memory location for DMA **at** - Type of address passed in parameter 'addr_buf'. Valid types can be virtual memory (VIRT_ADDR), physical memory (PHYS_ADDR) or physical memory inside Endpoint (EP_PHYS_ADDR). **sz** - Length of data to be transmitted/received. **cbk** - Callback registered to notify completion of DMA. Application can unmap, free buffers in this callback. Also application can invoke the API to submit new buffer/buffer-fragment. Callback is invoked by XDMA driver with channel lock held. May be NULL. **uid** - UserId passed to identify transactions spanning across multiple (typically 2) DMA channels. In a transaction type of application this field is used to match request/response. This can never be 0 and has to be set to a non-zero value even if the field is unused. **last_frag** - Set to true to indicate last fragment of a buffer. **ptr_user_data** - Pointer to application specific data that is passed as parameter when callback is invoked. Can be NULL.	API returns XLNX_SUCCESS on success. On failure, a negative value is returned. Check the "/* Xilinx DMA driver status messages */" in ps_pcie_dma_driver.h for more information on error codes.
void **xlnx_register_doorbell_cbk(** **ps_pcie_dma_chann_desc_t** *ptr_chann_desc, **func_doorbell_cbk_no_block** ptr_fn_drbell_cbk**)**;	Register callback to receive doorbell (scratchpad) notifications.	**ptr_chann_desc** - Channel descriptor pointer. **ptr_fn_drbell_cbk** - Function pointer supplied by application drive. It is invoked when a software interrupt is invoked (typically after populating scratchpad) to notify the host/Endpoint.	Void.

Appendix E

APIs Provided by the XDMA Driver in Windows

Table E-1 describes the APIs provided by the XDMA driver in Windows.

Table E-1: **APIs Provided by the XDMA Driver in Windows**

API Prototype	Details	Parameters	Return
XDMA_HANDLE XDMARegister(IN PREGISTER_DMA_ENGINE_REQUEST LinkReq, OUT PREGISTER_DMA_ENGINE_RETURN LinkRet)	This API is made available to child drivers present on the DMA driver's virtual bus. It is used by child drivers to register themselves with DMA driver for a particular channel on DMA. Only after successful registration will the child drivers be able to initiate I/O transfers on the channel.	PREGISTER_DMA_ENGINE _REQUEST DMA Engine Request provides all the required information needed by XDMA driver to validate the child driver's request and provide access to the DMA channel. It contains the following information: 1. Channel number 2. Number of queues the child driver wants the DMA driver to allocate and maintain. (It can be either 2 or 4 depending on the application's use case.) 3. Number of Buffer Descriptors in Source, Destination SGL queues and the corresponding Status Queues. 4. Coalesce count for that channel 5. Direction of the channel. 6. Function Pointers to be invoked to intimate successful completion of I/O transfers. PREGISTER_DMA_ENGINE _RETURN DMA Engine Return provides a structure which contains the following information: 1. DMA function pointer for initiating data transfer. 2. DMA function pointer for cancelling transfers and doing DMA reset.	XDMA_HANDLE The handle is a way for DMA driver to identify the channel registered with Child driver. All function calls to DMA driver after registration will have this as its input argument. If the DMA Registration is not successful, XDMA_HANDLE will be NULL
int XDMAUnregister(IN XDMA_HANDLE UnregisterHandle)	This API is made available to child drivers present on the DMA driver's virtual bus. It is used by child drivers to unregister themselves with DMA driver. After successful invocation of XDMAUnregister the DMA channel can be reused by any other child driver, by invoking XDMARegister	XDMA_HANDLE This is the same parameter obtained by child driver upon successful registration.	Always returns zero.

Table E-1: **APIs Provided by the XDMA Driver in Windows *(Cont'd)***

API Prototype	Details	Parameters	Return
NTSTATUS XlxDataTransfer(IN XDMA_HANDLE XDMAHandle, IN PDATA_TRANSFER_PARAMS pXferParams)	This API can be invoked using the function pointer obtained as part of PREGISTER_DMA_ENGINE_RETURN after successful registration of Child driver. This is the API which programs the buffer descriptors and initiates DMA transfers based on the information provided in PDATA_TRANSFER_PARAMS	XDMA_HANDLE This is the same parameter obtained by child driver upon successful registration. PDATA_TRANSFER_PARAMS This parameter contains all the information required by the DMA driver to initiate data transfer. It contains the following information: 1. Information regarding which Queue of DMA channel has to be updated. (It can either be SRC Q or DST Q). 2. Number of bytes to be transferred. 3. Memory location information, which helps the DMA driver to know whether the address provided is the Card DDR address or if it is a host SGL list.	STATUS_SUCCESS is retuned if the call is successful otherwise relevant STATUS message will be set.
NTSTATUS XlxCancelTransfers(IN XDMA_HANDLE XDMAHandle)	This API can be invoked using the function pointer obtained as part of PREGISTER_DMA_ENGINE_RETURN after successful registration of Child driver. Using this API the child driver can cancel all pending transfers and reset the DMA.	XDMA_HANDLE This is the same parameter obtained by child driver upon successful registration.	STATUS_SUCCESS is returned every time.

Recommended Practices and Troubleshooting in Windows

Recommended Practices

RECOMMENDED: *Make a backup of the system image and files using the Backup and Restore utility of the Windows 7 operating system before installing reference design drivers. (As a precautionary measure, a fresh installation of the Windows 7 OS is recommended for testing the reference design.)*

Troubleshooting

Problem: The TRD Setup screen of the GUI does not detect the board.

Corrective Actions:

1. If the GUI does not detect the board, open **Device Manager** and see if the drivers are loaded under **Xilinx PCI Express Device**.
2. If the drivers are not loaded, check the PCIe Link Up LED on the board (see Figure 3-8).
3. If the drivers are loaded but the GUI is not detecting the board, remove non-present devices from Device Manager using the following steps.
   a. Open a command prompt with Administrator privileges.
   b. At the command prompt, enter the following bold text:

      set devmgr_show_nonpresent_devices=1

      start devmgmt.msc

   c. Click the **View** menu and select **Show hidden devices** on the Device Manager window.
   d. Non-present devices are indicated by a lighter shade of text.
   e. Look for all the Non Present/Hidden devices. Right-click each one, and select **Uninstall**. Remove the driver if prompted for it.
4. Invoke the GUI of the reference design and check if it detects the board.

Additional Resources and Legal Notices

Xilinx Resources

For support resources such as Answers, Documentation, Downloads, and Forums, see Xilinx Support.

For continual updates, add the Answer Record to your myAlerts.

Solution Centers

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

References

The most up-to-date information for this design is available on these websites:

KCU105 Evaluation Kit website

KCU105 Evaluation Kit documentation

KCU105 Evaluation Kit Master Answer Record (AR 63175)

These documents and sites provide supplemental material:

1. *Vivado Design Suite Tutorial: Designing IP Subsystems Using IP Integrator* (UG995)
2. *Vivado Design Suite User Guide Release Notes, Installation, and Licensing* (UG973)
3. *Kintex UltraScale FPGA KCU105 Evaluation Board User Guide* (UG917)
4. *Vivado Design Suite User Guide: Logic Simulation* (UG900)
5. *LogiCORE IP UltraScale FPGAs Gen3 Integrated Block for PCI Express v3.0 Product Guide* (PG156)

6. *LogiCORE IP AXI Interconnect Product Guide* (PG059)
7. *UltraScale Architecture-Based FPGAs Memory Interface Solutions Product Guide* (PG150)
8. LogiCORE IP AXI Performance Monitor Product Guide (PG037)
9. *UltraScale Architecture System Monitor User Guide* (UG580)
10. PG020, LogiCORE IP AXI Video Direct Memory Access Product Guide (PG020)

Please Read: Important Legal Notices

The information disclosed to you hereunder (the "Materials") is provided solely for the selection and use of Xilinx products. To the maximum extent permitted by applicable law: (1) Materials are made available "AS IS" and with all faults, Xilinx hereby DISCLAIMS ALL WARRANTIES AND CONDITIONS, EXPRESS, IMPLIED, OR STATUTORY, INCLUDING BUT NOT LIMITED TO WARRANTIES OF MERCHANTABILITY, NON-INFRINGEMENT, OR FITNESS FOR ANY PARTICULAR PURPOSE; and (2) Xilinx shall not be liable (whether in contract or tort, including negligence, or under any other theory of liability) for any loss or damage of any kind or nature related to, arising under, or in connection with, the Materials (including your use of the Materials), including for any direct, indirect, special, incidental, or consequential loss or damage (including loss of data, profits, goodwill, or any type of loss or damage suffered as a result of any action brought by a third party) even if such damage or loss was reasonably foreseeable or Xilinx had been advised of the possibility of the same. Xilinx assumes no obligation to correct any errors contained in the Materials or to notify you of updates to the Materials or to product specifications. You may not reproduce, modify, distribute, or publicly display the Materials without prior written consent. Certain products are subject to the terms and conditions of Xilinx's limited warranty, please refer to Xilinx's Terms of Sale which can be viewed at http://www.xilinx.com/legal.htm#tos; IP cores may be subject to warranty and support terms contained in a license issued to you by Xilinx. Xilinx products are not designed or intended to be fail-safe or for use in any application requiring fail-safe performance; you assume sole risk and liability for use of Xilinx products in such critical applications, please refer to Xilinx's Terms of Sale which can be viewed at http://www.xilinx.com/legal.htm#tos.

Automotive Applications Disclaimer

XILINX PRODUCTS ARE NOT DESIGNED OR INTENDED TO BE FAIL-SAFE, OR FOR USE IN ANY APPLICATION REQUIRING FAIL-SAFE PERFORMANCE, SUCH AS APPLICATIONS RELATED TO: (I) THE DEPLOYMENT OF AIRBAGS, (II) CONTROL OF A VEHICLE, UNLESS THERE IS A FAIL-SAFE OR REDUNDANCY FEATURE (WHICH DOES NOT INCLUDE USE OF SOFTWARE IN THE XILINX DEVICE TO IMPLEMENT THE REDUNDANCY) AND A WARNING SIGNAL UPON FAILURE TO THE OPERATOR, OR (III) USES THAT COULD LEAD TO DEATH OR PERSONAL INJURY. CUSTOMER ASSUMES THE SOLE RISK AND LIABILITY OF ANY USE OF XILINX PRODUCTS IN SUCH APPLICATIONS.

Fedora Information

Xilinx obtained the Fedora Linux software from Fedora (http://fedoraproject.org/), and you may too. Xilinx made no changes to the software obtained from Fedora. If you desire to use Fedora Linux software in your product, Xilinx encourages you to obtain Fedora Linux software directly from Fedora (http://fedoraproject.org/), even though we are providing to you a copy of the corresponding source code as provided to us by Fedora. Portions of the Fedora software may be covered by the GNU General Public license as well as many other applicable open source licenses. Please review the source code in detail for further information. To the maximum extent permitted by applicable law and if not prohibited by any such third-party licenses, (1) XILINX DISCLAIMS ANY AND ALL EXPRESS OR IMPLIED WARRANTIES, INCLUDING, BUT NOT LIMITED TO, THE IMPLIED WARRANTIES OF MERCHANTABILITY AND FITNESS FOR A PARTICULAR PURPOSE; AND (2) IN NO EVENT SHALL XILINX BE LIABLE FOR ANY DIRECT, INDIRECT, INCIDENTAL, SPECIAL, EXEMPLARY, OR CONSEQUENTIAL DAMAGES (INCLUDING, BUT NOT LIMITED TO, PROCUREMENT OF SUBSTITUTE GOODS OR SERVICES; LOSS OF USE,DATA, OR PROFITS; OR BUSINESS INTERRUPTION) HOWEVER CAUSED AND ON ANY THEORY OF LIABILITY, WHETHER IN CONTRACT, STRICT LIABILITY, OR TORT (INCLUDING NEGLIGENCE OR OTHERWISE) ARISING IN ANY WAY OUT OF THE USE OF THIS SOFTWARE, EVEN IF ADVISED OF THE POSSIBILITY OF SUCH DAMAGE.

Fedora software and technical information is subject to the U.S. Export Administration Regulations and other U.S. and foreign law, and may not be exported or re-exported to certain countries (currently Cuba, Iran, Iraq, North Korea, Sudan, and Syria) or to persons or entities prohibited from receiving U.S. exports (including those (a) on the Bureau of Industry and Security Denied Parties List or Entity List, (b) on the Office of Foreign Assets Control list of Specially Designated Nationals and Blocked Persons, and (c) involved with missile technology or nuclear, chemical or biological weapons). You may not download Fedora software or technical information if you are located in one of these countries, or otherwise affected by these restrictions. You may not provide Fedora software or technical information to individuals or entities located in one of these countries or otherwise affected by these restrictions. You are also responsible for compliance with foreign law requirements applicable to the import and use of Fedora software and technical information.

© Copyright 2015 Xilinx, Inc. Xilinx, the Xilinx logo, Artix, ISE, Kintex, Spartan, Virtex, Vivado, Zynq, and other designated brands included herein are trademarks of Xilinx in the United States and other countries. All other trademarks are the property of their respective owners.