SDAccel Environment

Platform Development Guide

UG1164 (v2016.4) March 9, 2017

Revision History

The following table shows the revision history for this document.

Date	Version	Changes
03/09/2017	2016.4	Minor editorial updates. Added comments related to the DONE pin for configuration in SDAccel Kernel Programming.
11/30/2016	2016.3	Initial public release.

Table of Contents

Chapter 1 Overview

Introduction

The Xilinx® SDAccel™ Development Environment lets you compile OpenCL™ C, C and C++ programs to run on Xilinx programmable platforms. The programmable platform is composed of the SDAccel Xilinx Open Code Compiler (XOCC), a Device Support Archive (DSA) which describes the hardware platform, a software platform, an accelerator board, and the SDAccel OpenCL runtime. The XOCC is a command line compiler that takes user source code and runs it through the Xilinx implementation tools to generate the bitstream and other files that are needed to program the FPGA based accelerator boards. It supports kernels expressed in OpenCL C, C++ and RTL (SystemVerilog, Verilog or VHDL)

This document describes the steps required to create hardware platform designs for use with the SDAccel Development Environment. The hardware platform consists of a Vivado® IP Integrator subsystem design with all the required board interface IP cores configured and connected to the device I/Os and the programmable region. The programmable region defines the programmable device logic partition that accepts the software kernel from the SDAccel Development Environment.

This guide is written from the perspective of a hardware designer, using terminology common to the Vivado Design Suite. An overview of the OpenCL Design flow to create the programmable region, written from the perspective of a software developer, is provided in the *SDAccel Development Environment User Guide* (UG1023).

The SDX software installation includes the compatible version of Vivado needed to create or modify the hardware platform design. It can be accessed on the Linux command line or Windows Program Start Menu. For more information on using the Vivado Design Suite, refer to the online compatible version Vivado documentation and collateral.

Using Available Hardware Platforms

The process of creating a hardware platform DSA for use with the SDAccel Development Environment can be time consuming, depending on the device selected and the interfaces involved. If you are just learning the SDAccel solution, or are interested in experimenting with your own OpenCL kernels, you should consider using one of the standard hardware platforms offered by Xilinx. Refer to the *SDx Environments Release Notes, Installation, and Licensing Guide* (UG1238) for a list of standard platforms supported by the tool.

In addition, developing your own hardware platform is much simpler when starting with an existing hardware platform reference design and modifying it as needed to meet your design objectives.

Xilinx Developer Board for Acceleration with KU115

The Xilinx supplied hardware acceleration platform (**XIL-ACCEL-RD-KU115)** is an excellent basis to experiment with the capabilities of the SDAccel Environment. There is also the *SDAccel Platform Reference Design User Guide: Developer Board for Acceleration with KU115* (UG1234). This is the manual that describes understand the use of the hardware platform in the SDAccel design flow.

Using the Nimbix HPC Cloud-based SDAccel Environment

The entire SDAccel Acceleration capability is available as a cloud based service. There is a board farm with several hardware platform options available. Refer to https://www.nimbix.net/xilinx/ for more information.

Working with a Programmable Device

The Xilinx® SDAccel OpenCL boards are PCIe® based accelerator cards that plug into a standard PCIe slot in x86_64 host or server type architectures.

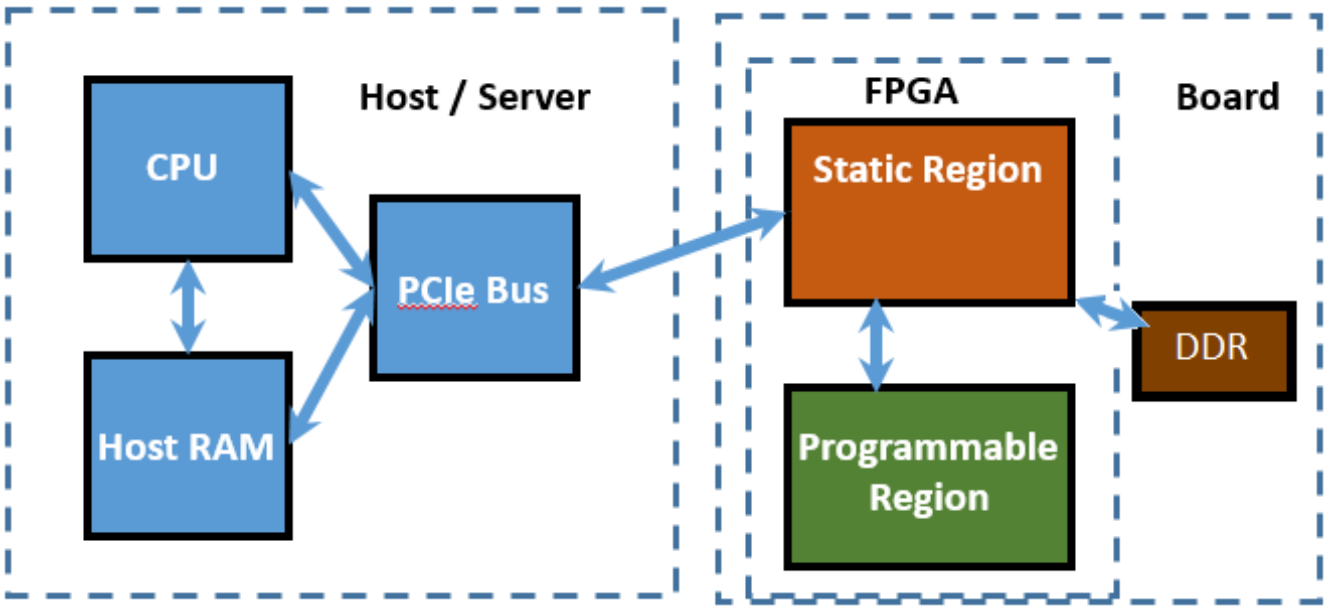

Figure 1: Programmable Device Block Diagram

The Xilinx PCIe hardware device consists of two regions, as shown in Figure 1, the Static Region and the Programmable Region. The Static Region provides the connectivity framework to the Programmable Region, which will execute the hardware functions as defined in the software kernel. The diagram above is an example of a standard partial reconfiguration hardware platform. The expanded partial reconfiguration hardware platform would incorporate most of the static region into the programmable region.

Terminology

Hardware Platform – Represents the physical board interface and the programmable region. The hardware platform consists of a Vivado IP Integrator design with a target device and all interface IPs configured and connected to the device I/Os and the programmable region. It also contains the interface representation of the programmable region.

Software Platform – Consists of the runtime, drivers, and software stack that are needed to enable interaction with the hardware platform.

Hardware Function – Blanket term to include either the OpenCL, C, C++, kernels or RTL IP that defines the programmable region.

Static Region – Represents the fixed logic portion of the programmable device that manages the design state before, during, and after partial reconfiguration of the programmable region. This logic is not re-implemented with the programmable region.

Programmable Region – Describes the partition region that accepts the hardware functions from the SDAccel Development Environment. The term also describes the physical resources available on the programmable device to implement the hardware functions. Special parameters and design considerations are required for signals that cross between the static region and the programmable region.

Device Support Archive (DSA) – Contains all of the design and metadata needed for an SDAccel hardware function to interact with the physical design. It is the output product of the hardware platform design process described in this guide.

Understanding Partial Reconfiguration

The hardware platform design process requires familiarity with the underlying design flow used in the SDAccel Development Environment. The SDAccel programming flow uses the partial reconfiguration capabilities unique to Xilinx to enable downloading partial bitstreams into the programmable device, or hardware platform while it remains online and connected to the host computer's PCIe bus. Refer to the *Vivado Design Suite User Guide: Partial Reconfiguration* (UG909) for more information about partial reconfiguration.

The partial reconfiguration flow requires the use of a decoupler IP core to hold the design in a safe state while the device is partially reconfigured with the hardware function. Refer to Using the Decoupler IP in Chapter 3.

The partial reconfiguration flow also requires several modifications to the design including floorplanning and partition pin placement constraints to lock down the interconnect route points of the interface signals. Refer to Applying Partial Reconfiguration Constraints in Chapter 4.

Understanding Regular and Expanded PR Hardware Platforms

There are two methods for using partial reconfiguration in hardware platform designs: partial reconfiguration (PR) and expanded partial reconfiguration (XPR). The hardware platform development process varies based on which of these methods is used, as it affects the logic hierarchy and design preparation techniques.

The XPR flow expands the contents of the reconfigurable module to encompass the programmable region as well as a large portion of the hardware platform design, as shown in Figure 2. The XPR method is used to maximize the available device resources to improve results, and is also used with SSI devices. Better performance can be achieved with the XPR flow by allowing the implementation tools greater flexibility when placing the kernel and interface logic onto the programmable device. Both the standard PR and XPR methods are described in the following sections and are referred to throughout this document.

Figure 2 shows the underlying design processes used to describe and implement hardware functions on the programmable device. It illustrates both the regular PR and expanded PR hardware platform design options.

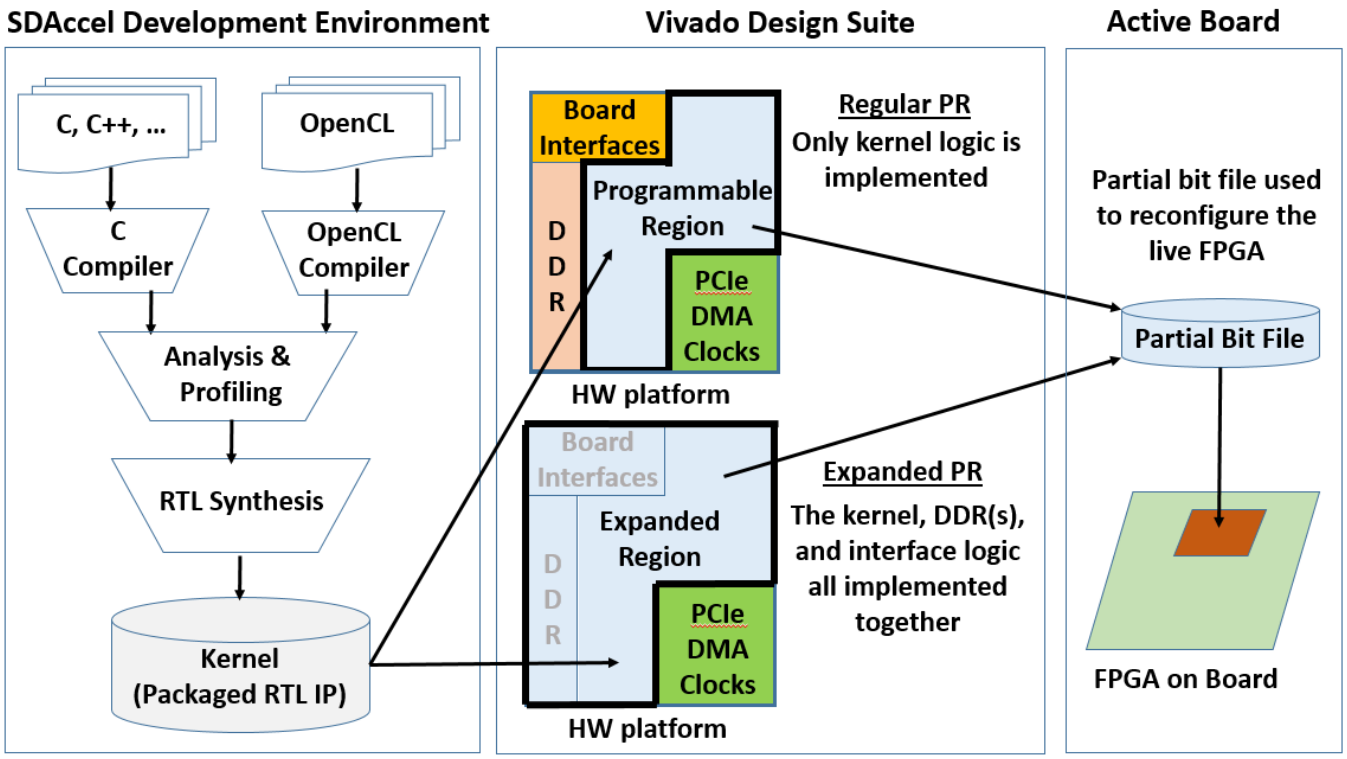

Figure 2: The SDAccel Device Programming Flow Diagram

Using Expanded Partial Reconfiguration

You can see from the diagram above, that the XPR method includes board interface logic and DDR memory that are a part of the hardware platform design as part of the reconfigurable module. Implementing these along with the kernel from the SDAccel Environment enables the most effective use of the programmable device resources across the entire design. This is primarily done to improve the quality of the results, and maximize the performance of the overall design. With XPR, a partial bit file is created for the entire expanded region and is used to reconfigure the active device.

The smaller static region contains the minimum logic needed to keep the hardware platform online and connected while waiting to be reconfigured with the hardware function of the programmable region. Typically with the XPR method, only the Xilinx DMA Subsystem for PCI Express, basic control interfaces, and clock sources are contained within the "static region". This region is floorplanned to use as little of the device area as possible in order to maximize the available area for kernel resources. The logic hierarchy must be designed to separate the static region logic from the expanded programmable region, in the Vivado IP Integrator block design. Refer to Chapter 3, Logic Design using IP Integrator.

Programming the Device using Partial Reconfiguration

The FPGA is a peripheral on a PCIe bus, and the system host configures the FPGA through the PCIe connection.

Power-On Reset Programming

After power-on reset of the FPGA, the entire device is configured with a full bitstream (.bit) file from flash memory. The test kernel used to create the top-level bitstream file within the DSA should be designed for the desired power on design state.

SDAccel Kernel Programming

As each kernel is executed on the FPGA, the SDAccel software controls the loading of partial bitstreams for the programmable region. Partial bitstreams are fully self-contained. All addressing, header, and footer details are contained within these bitstreams, just as they would be for full configuration bitstreams. Dedicated PR features such as per-frame CRC checks (to ensure bitstream integrity) and automatic initialization (so the region starts in a known state) are available, as well as full bitstream features such as encryption and compression.

As part of the static region of the design, the internal Partial Reconfiguration control circuitry operates without interruption throughout the Partial Reconfiguration process. This allows the portion of the FPGA logic not being reconfigured to continue functioning while the reconfigurable portion is modified. The Decoupler IP is used to maintain the board state while it is partially reconfigured. Refer to Using the Decoupler IP in Chapter 3 for more information.

The partial bitstreams are retrieved and then delivered through the PCIe connection and onto the FPGA configuration logic. For 7 Series devices, an Internal Configuration Access Port (ICAP) is used to configure the partial bitstreams. For UltraScale devices, the Media Configuration Access Port (MCAP) of the PCIe block is used to configure the partial bitstreams. For more information on the use of ICAP or

MCAP please refer to *7 Series FPGAs Configuration User Guide,* (UG470) and *UltraScale Architecture Configuration User Guide,* (UG570) respectively.

The DONE pin pulls LOW when reconfiguration begins and pulls HIGH again when partial reconfiguration successfully completes. For this reason, do not use the DONE pin to directly generate PCIe reset. See the *Vivado Design Suite User Guide: Partial Reconfiguration* (UG909) for more information.

TIP: *With UltraScale devices, the DONE and PRDONE pins pull LOW at the beginning of the clearing bitstream and remain low until the end of the partial bitstream because the two bitstreams together constitute a complete partial reconfiguration sequence. The DONE/PRDONE pin does NOT return high at the end of the clearing bitstream.*

The following are some special notes regarding pins and I/O banks for use in Partial Reconfiguration:

- *DONE – This pin will pull low when partial reconfiguration begins and return high when it successfully completes. For this reason, do not use the DONE pin to directly generate PCIe reset. For 7 series and Zynq-7000 devices, this will only occur when RESET_AFTER_RECONFIG is used; this option is always on for UltraScale devices and newer.*
- *INIT_B – This pin pulls low when a configuration error occurs when loading a partial bitstream, just as it does for a power-on configuration.*
- *Reconfigured I/O Banks – If an IO bank is included within a reconfigurable partition, all the IO within the bank will be held in tristate until reconfiguration completes.*
- *DCI_Cascade – Neighboring banks cannot use the DCI_Cascade feature if one is static and one is reconfigurable (PR) or if one is in stage 1 and one is in stage 2 (Tandem Configuration)*

Understanding the Device Support Archive (DSA)

Compiling OpenCL, C and C++ kernel programs to run as a partial bitstream on a Xilinx programmable device requires a specific set of data to properly model the programmable region interface to the active device. The Device Support Archive, or DSA, contains all of the data required to enable the design and programmability of your OpenCL C or C++ kernel directly with the running FPGA.

The DSA is output from the hardware platform design project in the Vivado Design Suite for use with the SDAccel Development Environment. See Chapter 6, Generating DSA Files.

Several example DSAs are shipped with the SDAccel Development Environment, or you can create your own using the methodology described in this document.

Hardware Platform Design Flow

Using the Vivado IP Integrator environment, you can model the hardware platform, define and configure the static and programmable regions, configure the device interfaces, define physical constraints to implement and validate the hardware platform, and test it with a sample kernel.

The diagram below illustrates the hardware platform design and validation flow. Each of the displayed steps is described in greater detail in this document.

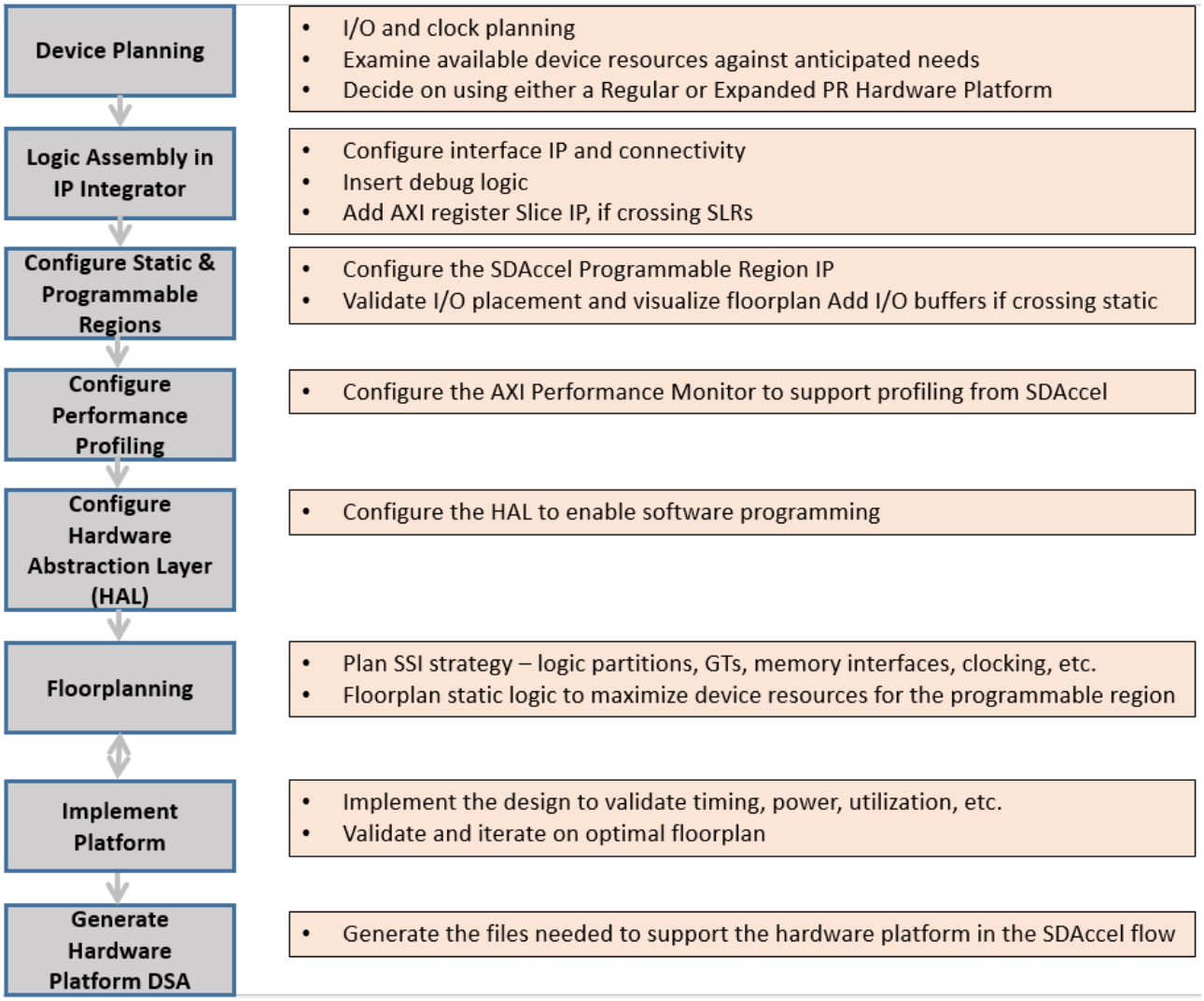

Figure 3: Hardware Platform Creation and Validation Flow Diagram

Chapter 2 Device Planning

Introduction

To assure the best possible performance of the overall design, you should carefully plan the assignment of the required device interfaces. You should study and understand the device package footprint, the package pin assignments, and the internal die layout of the programmable device to determine the best I/Os to use for the various hardware interfaces, especially the external processor and memory interfaces.

You should plan the intended logic flow of the hardware platform, while paying attention to clock distribution, the expected logic size of the interfaces, and the hardened device resources such as PCIe, BRAMs, and DSPs. This is true not only for the static region logic, but also for the programmable region logic. You should focus on maximizing the available resources for the programmable region logic.

Optionally, a Platform Board file can be created to support the hardware platform design. The board file defines I/O characteristics and the interface IP configuration options. A board file facilitates the use of Connectivity Assistance in the IP Integrator. This can greatly improve the ease of platform design creation. Also, if variants of the hardware platform are planned, consider using a Platform Board file to enable faster configuration of IP interfaces.

As you define the hardware platform logic in the Vivado Design Suite, you can use the I/O planning features to assign the I/O and clock constraints. You can also use floorplanning techniques to ensure consistent results and optimal performance of the hardware platform design. Refer to the *Vivado Design Suite User Guide: I/O and Clock Planning* (UG899) and the *Vivado Design Suite User Guide: Design Analysis and Closure Techniques* (UG906) for more information.

Working with SSI Devices

If your target device is a Stacked Silicon Interconnect (SSI) device, you must also analyze and plan the data flow across super-logic region (SLR) boundaries. Although the increased availability of device resources in SSI devices is beneficial, the delay penalty for routes crossing between SLRs can be significant, and make timing closure especially difficult in highly connected designs.

AXI data paths used in many platforms are wide and generally include combinatorial paths to implement the AXI-4 Memory Mapped protocol. It is imperative that these data paths are well designed and contain as few logic levels as possible when crossing SLR boundaries.

I/O interfaces, IP logic distribution across SLRs, and clock isolation should also be studied carefully. As the hardware platform design is assembled and implemented, you may need multiple iterations of the floorplan to achieve the required results.

I/O and Clock Planning

You need to pay special attention to the I/O interfaces and clocking topology used with SSI devices. You should isolate clocks within a single SLR whenever possible. The logic associated with various I/O interface IP should be floorplanned to reduce SLR crossings. You should also assign I/Os with the objective of optimizing data flow, and to account for the programmable region logic coming from the SDAccel Development Environment. Refer to the *Vivado Design Suite User Guide: I/O and Clock Planning* (UG899) for more information.

Logic Partitioning across SLRs

To avoid unnecessary SLR boundary crossing, use floorplanning techniques to assign specific IP cores to specific SLRs. Plan the intended logic flow while paying attention to clock distribution, the expected logic size of the interfaces, and the utilization of hardened device resources such as PCIe, BRAMs and DSPs.

When planning logic content for the design, the total resources available in SSI devices should not be the only factor. Each SLR has a specific logic count, and over packing a single SLR can lead to poor performance for the whole design. You should take care to balance the resource utilization across all SLRs as much as possible.

The logic for the programmable region needs to be planned as well. Optimally, the programmable region imported from SDAccel would be implemented within a single SLR. Consistent timing performance is more challenging when the programmable region spans more than one SLR. See Configuring the Programmable Region in Chapter 3 for more information.

Consider using an Expanded PR hardware platform to maximize device resources and to allow the implementation tools flexibility when placing timing sensitive logic across the SLR boundaries.

Adding Pipeline Registers across SLRs

If a critical interface must cross an SLR boundary, consider adding pipeline registers to the interconnect to help prevent performance degradation. These registers can be floorplanned to assign them to a specific SLR resource, or allowed to float to let the Vivado tool place them.

Figure 4 shows a simplified representation of the logic partitioning described above. The red horizontal line represents the boundary between the two SLRs. The small white squares represent pipeline stages that are floorplanned for assignment to a specific SLR. The small gray squares in the figure represent pipeline stages connecting to the programmable region that are not floorplanned, and may be placed on either side of the SLR boundary as determined by the Vivado placer.

Note: The pipeline stages are actually sub-modules of SmartConnect IP instances, but for clarity are shown as separate entities in the simplified figure.

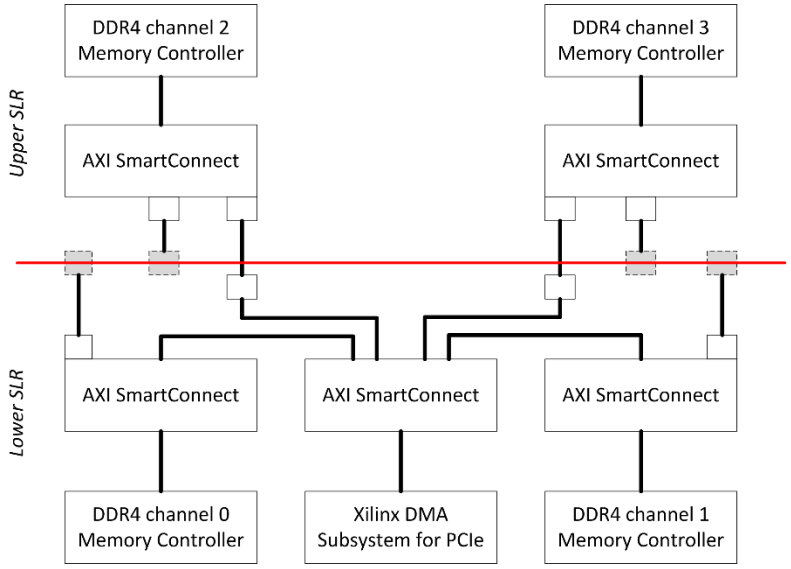

Figure 4: Simplified representation of IP partitioning for controlled SLR crossing

Define the static region logic interconnect with careful floorplanning of IP logic and signal interfaces and then create any necessary AXI Register SLR crossings in order to effectively utilize both SLRs.

Using Platform Board Files

Optionally, a Platform Board file can be created to support the hardware platform design. The board file defines I/O characteristics and the interface IP configuration options.

The Vivado Design Suite enables the use of platform boards that define the components on the board, the I/O interfaces between the components, and configuration options for IP cores as the design is being assembled in the Vivado IP Integrator. The Vivado Design Suite includes board interface files for several supplied reference boards that are delivered as part of the tool. You can also get predefined boards from supported third-party providers, and define your own board interface files as described at this link in the *Vivado Design Suite User Guide: System-Level Design Entry* (UG895).

When you define a new project in the Vivado tool you can select a target Xilinx device, or a platform board. A board file facilitates the use of Connectivity Assistance in the IP Integrator and enables quick variations of the hardware platform. It simplifies the process of adjusting IP configurations and making connections in the Vivado IP Integrator block design.

When you use the platform board flow, the Vivado IP Integrator shows all the interfaces to the board in a separate window called the Board window, as shown in Figure 5. When you use this window to select the desired components, you can easily connect the IP in your block design to the board components of your choosing. All the interface connections and I/O constraints are automatically generated as a part of using the platform board flow.

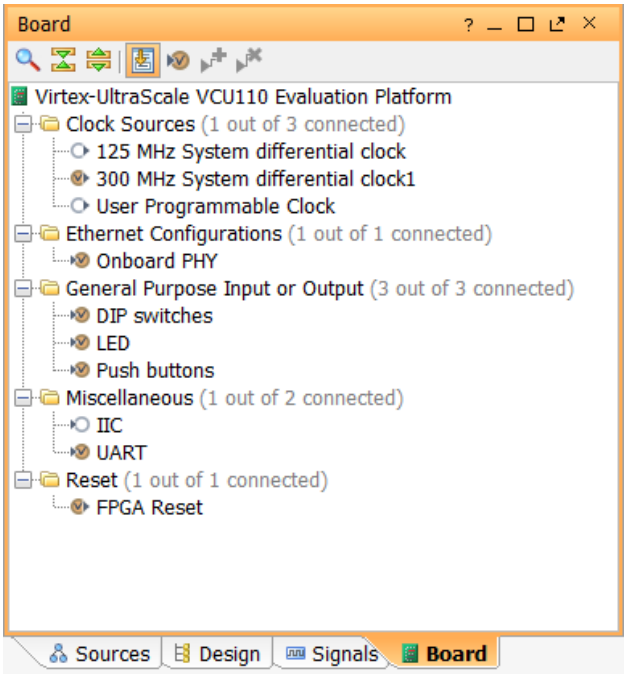

Figure 5: Board Interface Configuration Options

Chapter 3 Logic Design using IP Integrator

You will use the Vivado IP Integrator feature to create the hardware platform logic design. This feature of the Vivado Design Suite provides the necessary board information, interface IP, and the interface to the programmable region which will be imported from the SDAccel Development Environment. This chapter describes the process for creating the hardware platform. For more information on working with the Vivado IP Integrator, refer to the *Vivado Design Suite User Guide: Designing IP Subsystems Using IP Integrator* (UG994).

Creating the Design Hierarchy

As previously discussed, the logic hierarchy of the hardware platform must be carefully planned to properly support the partial reconfiguration flow (PR or XPR) for the programmable region of the design, to define the static base region, and to plan the signal flow across SLRs of SSI devices, and to other elements of the hardware platform, such as components on the platform board or IP cores in the subsystem design. Proper levels of hierarchy need to be established within the Vivado IP Integrator block design.

Defining the Static Region for Regular PR Hardware Platforms

The static region is the area of the hardware design that is fixed on the hardware platform, does not get implemented with the programmable region, and is not affected by the PR flow. Remember the static region logic is used to keep the board in a stable state and to manage the device reconfiguration process. As shown in Figure 6, it is composed of the main connectivity framework IP: the PCIe Endpoint, DMA Controller, AXI Interconnects, and Memory Controller. There are other smaller interface IP cores that are not shown in the figure below, but that are part of the static region as described in this chapter.

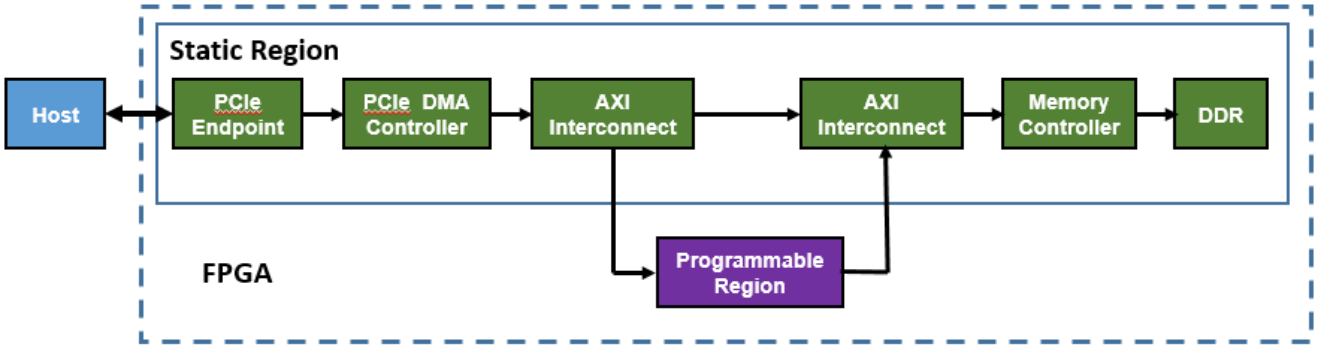

Figure 6: Static Region with Programmable Region (detail from Figure 1)

The Vivado IP integrator provide flexibility for capturing the overall structure of your design. Levels of hierarchy can be created for easier design navigation and comprehension. When using a regular PR hardware platform, it is recommended to place the SDAccel OpenCL Programmable Region IP on the top-level of the design, as shown in Figure 7 below. This IP is used to configure the Programmable Region interface. The rest of the design is considered the static region with a regular PR hardware platform design. To make the connectivity between the static and programmable regions more visible, the hierarchy can be created to isolate the static and programmable regions of the design, as shown below.

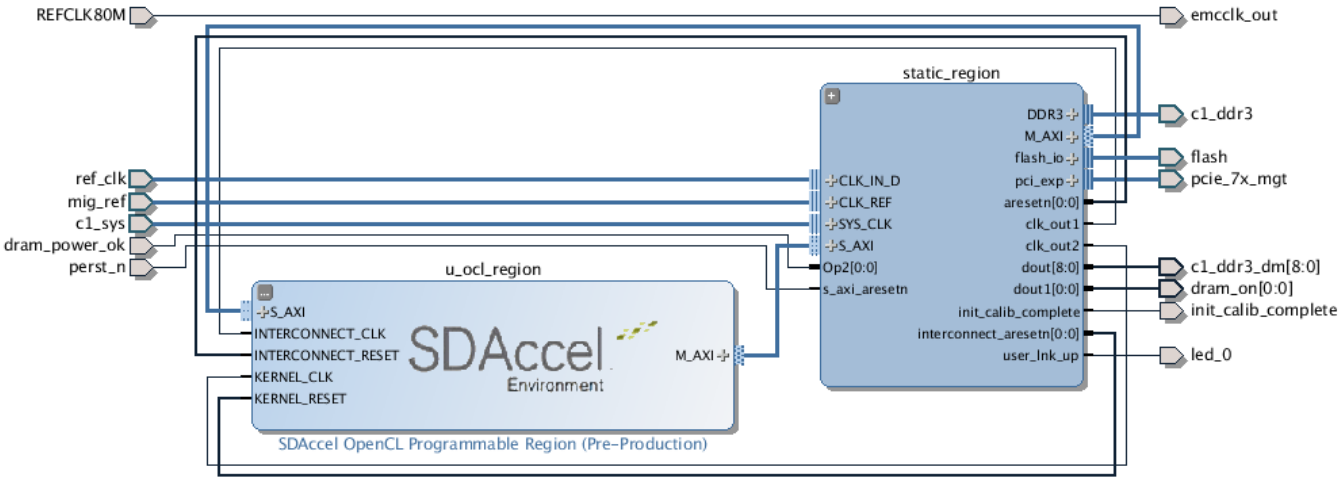

Figure 7: Define the Static and Programmable Regions on the Top-level of Design

Defining the Static Region for Expanded PR Hardware Platforms

As seen in Figure 8 below, when using expanded partial reconfiguration (XPR), the expanded region and the static region logic must be separately partitioned in the hierarchy of the hardware platform.

The figure below shows the top-level view of the IP Integrator block diagram of an example hardware platform showing the static base region and the reconfigurable expanded region.

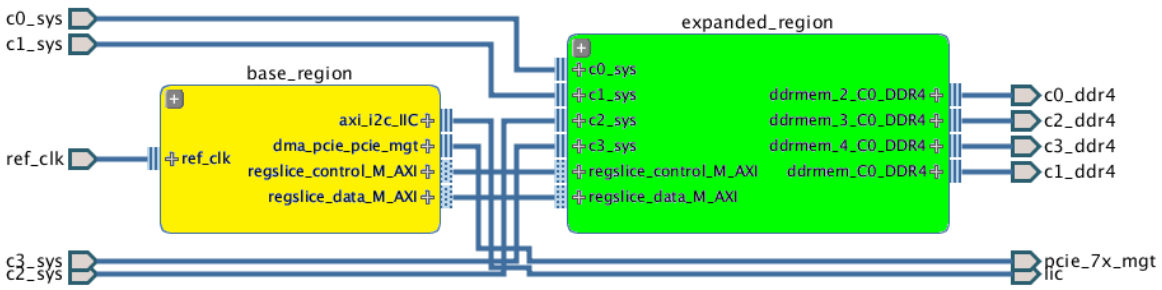

Figure 8: Top-Level Logic Hierarchy for Expanded Region

The SDAccel OpenCL Programmable Region IP is contained inside the expanded region hierarchy. Figure 9 below shows an example of the inside of the expanded region logic hierarchy.

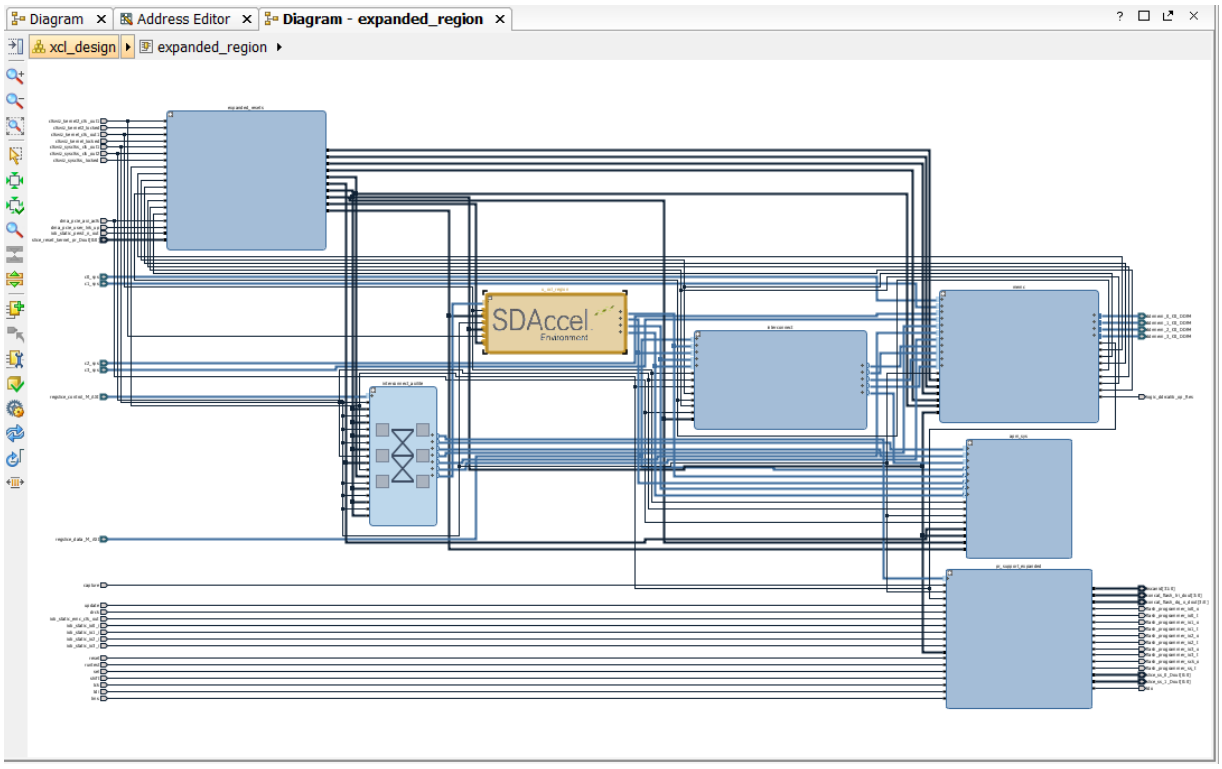

Figure 9: Programmable region inside of the Expanded PR Region Hierarchy

Including I/O Buffers for Static I/O Crossings

With partial reconfiguration, all routing must be contained within the programmable region. If the floorplan for the static region includes any I/O ports that are used by signals within the expanded region, an IBUF or OBUF needs to be placed in the static region logic design to connect it through the static region to the expanded region. Currently, this involves preparing a small RTL design containing the I/O buffer logic and adding it to the design using the Vivado IP Integrator **Add Module** command. An example of such RTL code is included with the sources supplied with the Xilinx Platform Development Board.

Creating and Configuring the Logic Design

Use the features of the Vivado IP Integrator to instantiate IP from the Vivado IP catalog, configure and connect all of the interface IP associated with the hardware platform. Figure 10 below shows an example hardware platform design. This could represent either the top-level design for a regular PR hardware platform design or the level of hierarchy under the expanded region in the XPR flow. Notice the SDAccel module that defines the programmable region of the OpenCL, C, or C++ kernel.

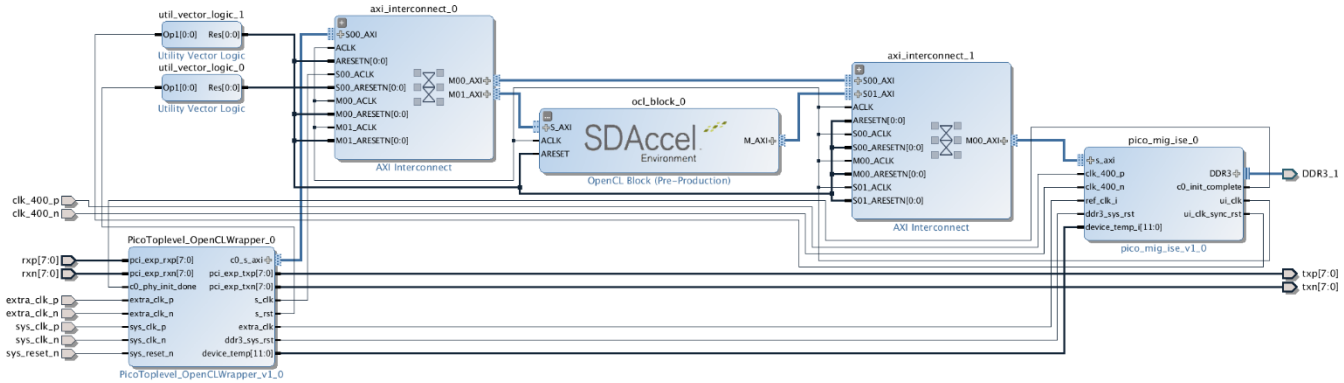

Figure 10: Example Hardware Platform Design

Configuring the Programmable Region

The programmable region is represented by the SDAccel OpenCL Programmable Region IP core in the block design. You must configure this module to properly represent the interfaces between the programmable region and the static logic design. The OpenCL compiler uses the IP integrator framework to swap out the default programmable region logic (for the training or placeholder kernel) and replace it with the compiled kernel generated from the OpenCL device code, and the interconnect topology it requires.

To add the SDAccel IP core to the Vivado IP Integrator block design:

1. Use the **Add IP** command to add the "SDAccel OpenCL Programmable Region" IP to the design canvas. The instance name is ocl_block_0.
2. Right-click on the IP and use the **Customize Block** command to open the Re-Customize IP dialog box to see the features of the IP core, as shown in Figure 11.

The SDAccel OpenCL Programmable Region IP is a container for compiled OpenCL kernels from the SDAccel Development Environment. You can specify the number of training kernels the core contains. When used with SDAccel, the OpenCL code and the XOCC determines the number of kernels in the programmable region. The IP has one AXI slave interface which is used to send commands to the OpenCL kernels. It also has one or more AXI master interfaces which are connected to device RAM. A clock and a reset signal are also wired to the SDAccel OpenCL Programmable Region.

You can customize the address and data bit width for the AXI master interface, but you should utilize the maximum value that the memory controller can support in order to maximize bandwidth and minimize data width converters on the datapath.

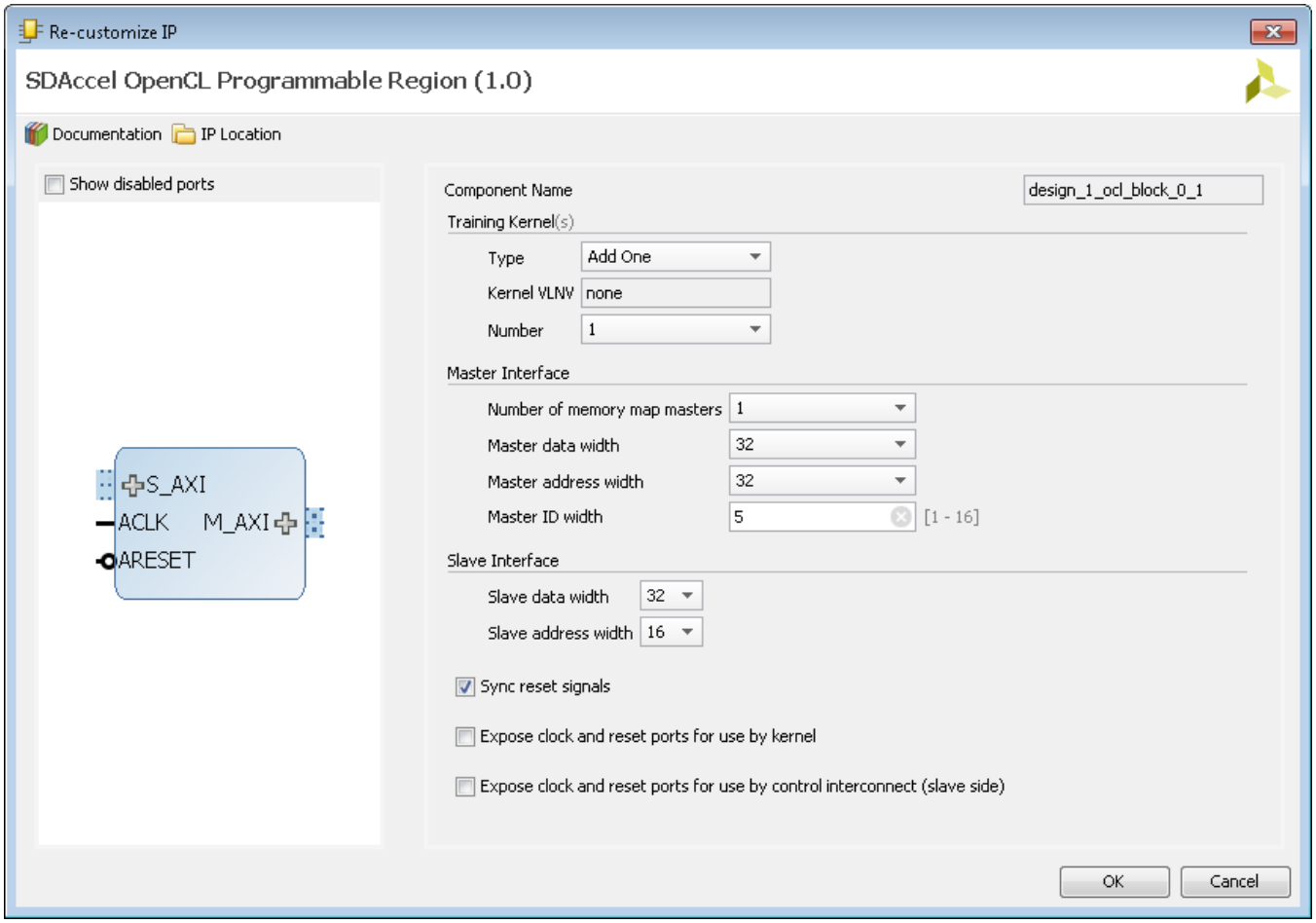

Figure 11: SDAccel OpenCL Programmable Region IP Configuration Options

The SDAccel OpenCL Programmable Region is a hierarchical IP with placeholders for compiled kernels. You can view the hierarchical block design by right-clicking on the IP and using the **View Block Design** command.

Supporting Multiple Kernels

The SDAccel OpenCL Programmable Region can support up to 16 compiled kernels on an FPGA device, located between the slave and master AXI interconnects. Figure 12 shows the placeholder kernels in the hierarchical block design.

You must set the **Master ID width** on the Re-customize IP dialog box of the SDAccel OpenCL Programmable Region in order to support multiple kernels, as seen in Figure 11. The Master ID width field supports values from 1 to 16.

Connecting Directly to External I/Os

If direct connections are required from the programmable region to external FPGA I/Os, you must instantiate either an IBUF or OBUF into the static logic region. Refer to the Including I/O Buffers for Static I/O Crossings section for more information.

Configuring Programmable Region Kernels

The SDAccel OpenCL Programmable Region IP can be configured to support up to 16 kernels, as shown in Figure 13. These are essentially placeholders to later accept compiled SDAccel kernels. An example of the internal IP connectivity is shown with three kernels in the following figure.

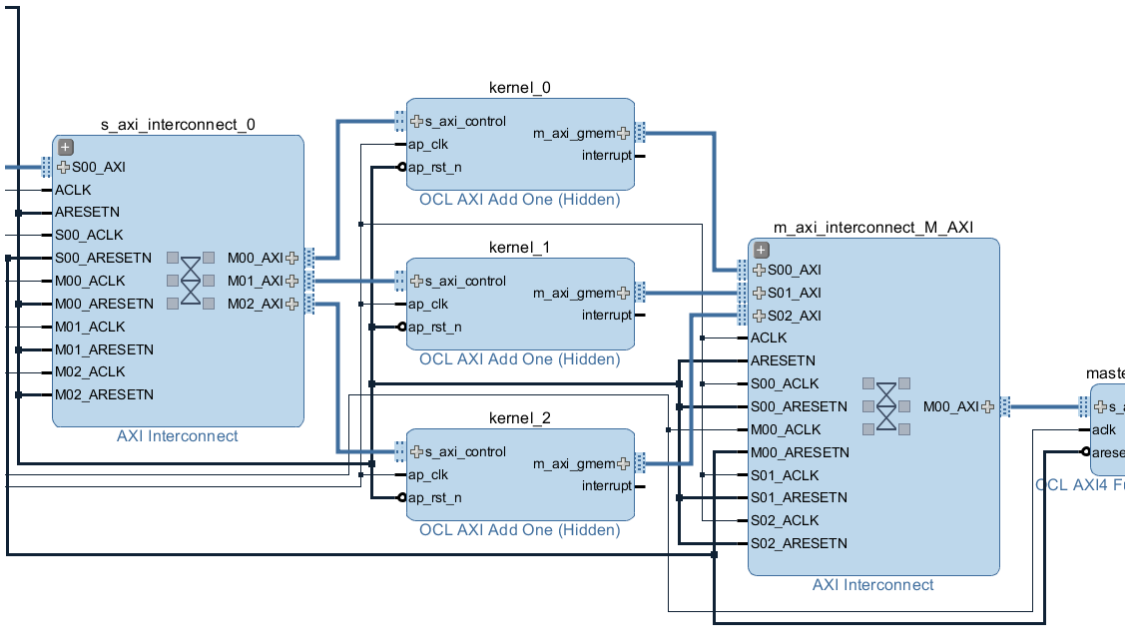

Figure 12: IP Configuration with Placeholder Kernels

Defining Training Kernels

Training Kernels can be defined using the IP Customization view. Xilinx supplies several training kernels or a custom VLNV training kernel can be defined, as shown in Figure 13.

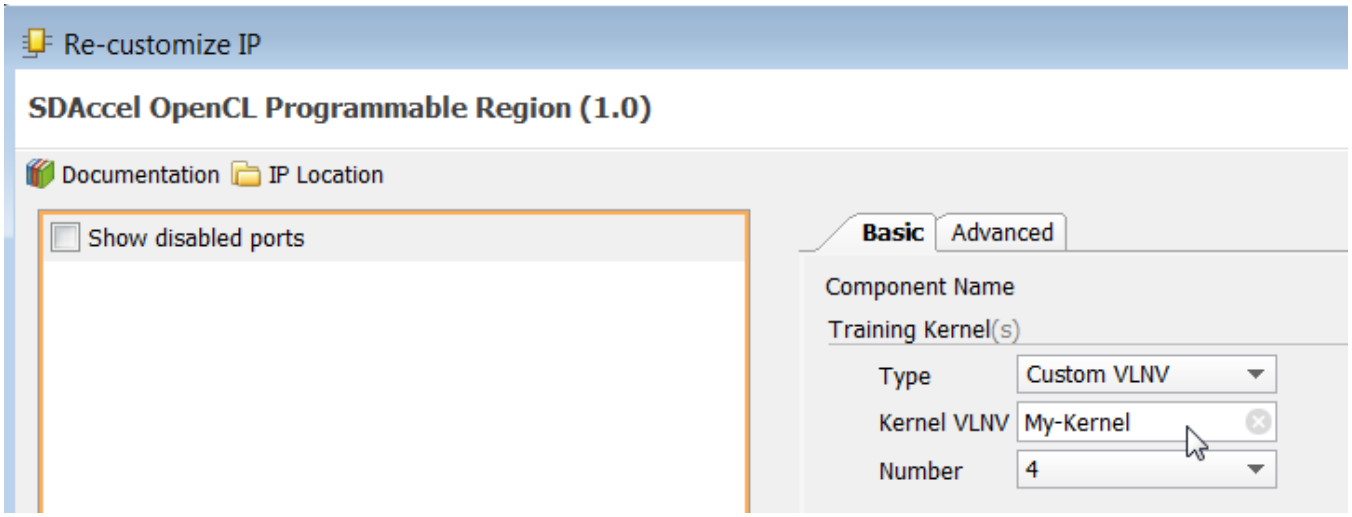

Figure 13: Defining a Training Kernel

Working with AXI Interconnect

As shown in Figure 12 above, the programmable region has two AXI interconnects. The left AXI interconnect is used to connect the SDAccel OpenCL Programmable Region slave control port and provide a path to read/write DDR from the PCIe bus. The right AXI Interconnect is used to connect two bus masters to the Memory Controller: the PCIe DMA controller and SDAccel OpenCL Programmable Region IP. The two AXI interconnects realize a path that allows you to bypass the Programmable Region and directly access the card resident RAM.

TIP: *When the SDAccel OpenCL Programmable Region is customized to expose more than one AXI master interface, a matching number of AXI Interconnects is used within the programmable region.*

Configuring the AXI Address Space

The device relies on AXI memory mapped bus for addressing the OpenCL kernel, the device dedicated RAM, and any other peripherals. There are two AXI masters in the design: the PCIe Endpoint/DMA Controller and the OpenCL kernel. There are two AXI slave end points in the design: The AXI-Lite control port of the Programmable Region (*ocl_block_0*) and the MIG controller.

Defining the Address Space

Open the Address Editor tab as shown in Figure 14. It shows the DMA Controller with one AXI master and 2 slave AXI endpoints. It should also show the SDAccel OpenCL Programmable Region with one slave AXI endpoint. Select **Auto Assign Address** for the DMA Controller and the OCL REGION. The Master interface of the DMA controller would have two address ranges for the two Slave interfaces it is connected to:

1. Slave Port of Memory Controller: Select the full range of the addressable memory which is typically 4G with Offset Address 0x000000000 and High Address 0x0FFFFFFFF.
2. Slave Port of OCL Region: Select a range of 8K with the Offset Address 0x100000000 and High Address of 0x100001FFF. If the DMA master has several master interfaces, the address ranges can be different.
3. The Programmable Region master interface should use the same offset address to access the external addressable memory, but the range will be limited (in the current release of the tools) to a 64 bit address, making it limited to 4GB.
4. You can validate the design by clicking **Validate Design**.

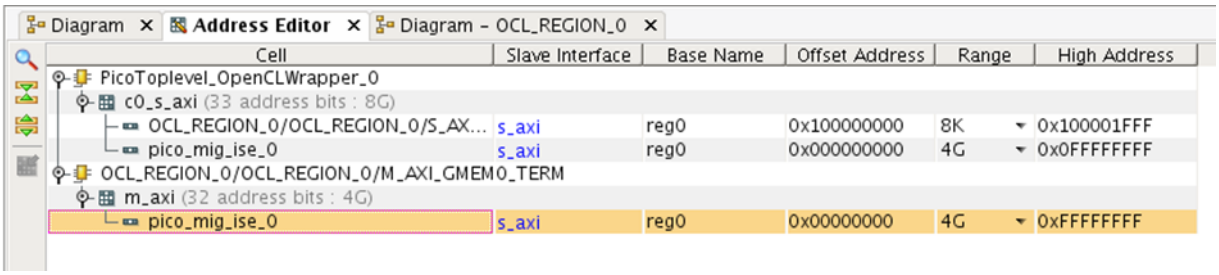

Figure 14: AXI Address Spaces

Adding Custom IP

Custom RTL can be added to the design in the IP Integrator using one of two ways. RTL only logic can be added using the **Add Module** command. Designs that contain IP or other logic structures can be packaged as IP and added to the design using the IP Packager.

Using the Add Module Command

Custom RTL can easily be incorporated into the design by using the **Add Module** command. RTL sources must first be added to the Vivado project using the **Add Sources** command. It is the available in the Add Module dialog box to be added to the design..

Using the IP Packager

The Vivado IP Packager can be used to define custom IP to add to the hardware platform design. The Vivado IP Packager supports standard interfaces such as AXI, DDR, and PCIe signals. For more

information on packaging custom IP refer to the *Vivado Design Suite User Guide: Creating and Packaging IP* (UG1118).

Once packaged, the custom IP can be added to the IP Catalog, and made available in the Vivado IP Integrator. For SDAccel, you would typically use this feature for defining and importing a custom DMA or memory controller, custom RTL, and I/O Buffer logic.

To add a custom IP repository to the IP Catalog:

1. Open the **Tools > Project Settings** dialog box and select the IP tab.
2. Click the **Repository Manager** tab, select the **Add** command, and select the folders that contain the custom packaged IP.

Configuring the Design for SDAccel

The platform design should contain a DMA Subsystem master IP, memory controller interfaces (DDR3, DDR4 IP) for global memory, clocking via the Clocking Wizard IP and resets via the Processor System Reset IP. Refer to the *SDAccel Platform Reference Design User Guide: Developer Board for Acceleration with KU115* (UG1234) for more information.

Use the Xilinx DMA Subsystem for PCI Express (XDMA) IP, or use a board vendor specific IP for the DMA functionality. This IP provides PCIe DMA functionality between the card resident RAM and host RAM. The XDMA IP requires the addition of two master AXI interfaces:

- One to master data to the Memory Controller.
- One to control the programmable region (to configure the OCL kernels) and to control other peripherals in the static region.

Add the DMA Subsystem for PCI Express (XDMA) IP to the IPI design. The configuration options are displayed in the following figure.

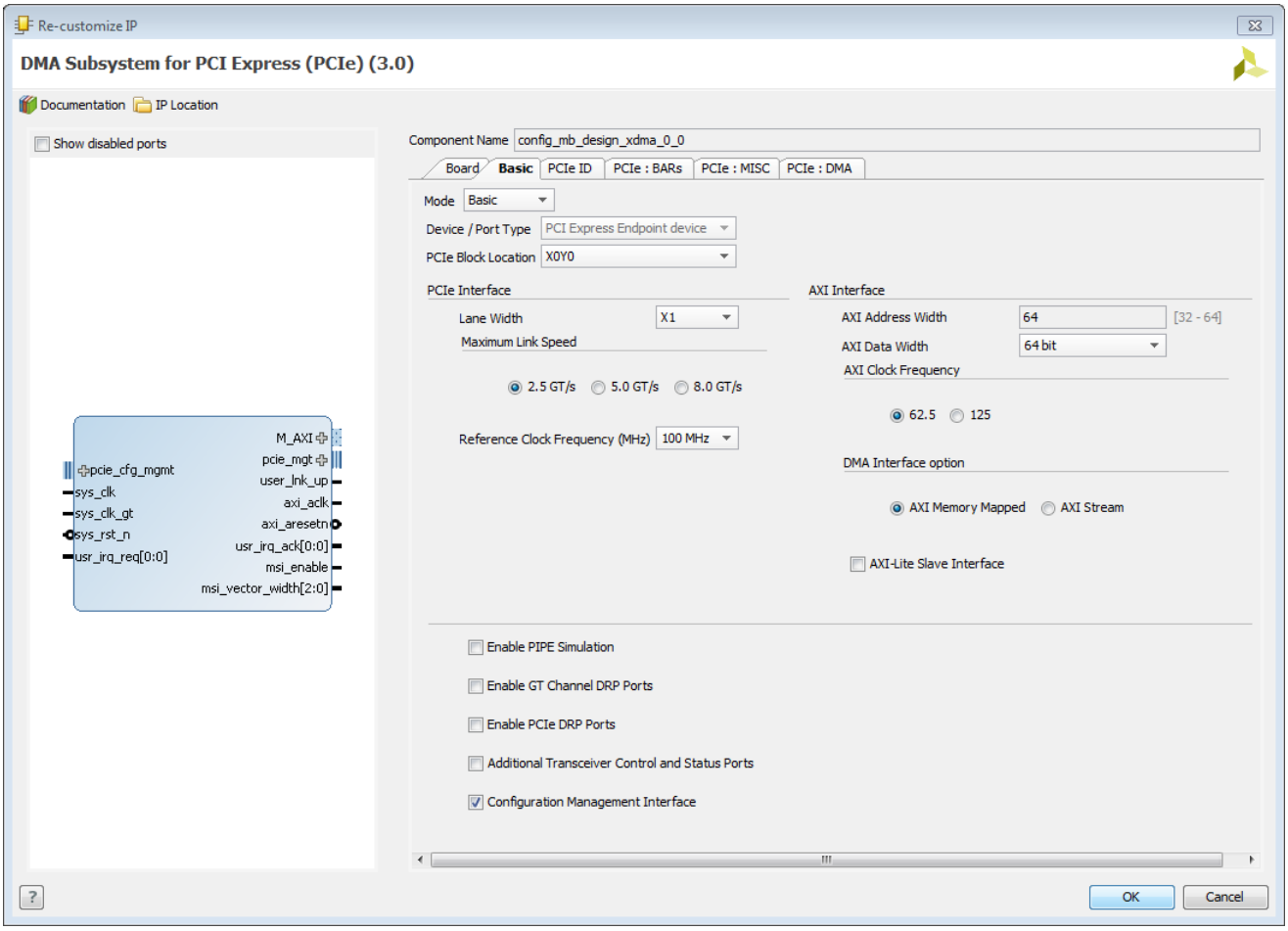

Figure 15: DMA Subsystem for PCIe Configuration Options

Using the Decoupler IP

The Partial Reconfiguration Decoupler IP may also be used in the static region to hold the design in a safe state while the device is partially reconfigured with the hardware function. A single reset within the IP will put the entire Programmable Region into reset. A pair of registers supply back pressure on the AXI interfaces to ensure they maintain state. The Decoupler IP can be instantiated and configured into the static region design using the IP Integrator. Figure 15 and Figure 16 show the Decoupler IP configuration options.

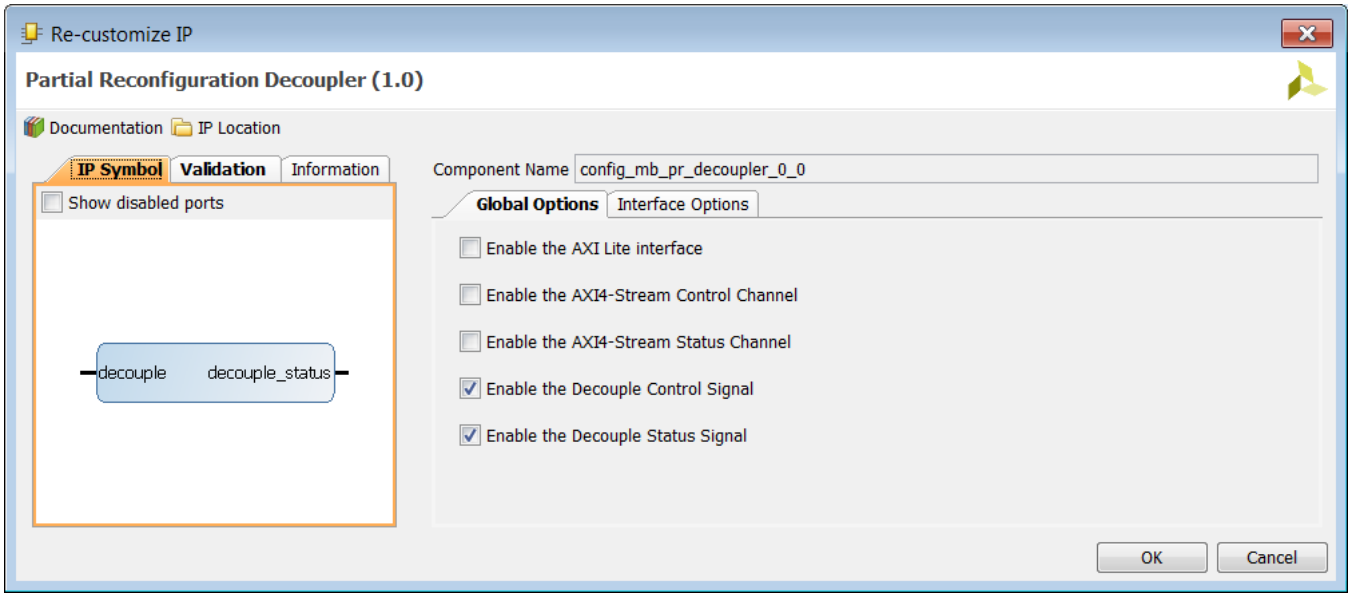

Figure 15: Partial Reconfiguration Decoupler IP - Global Options

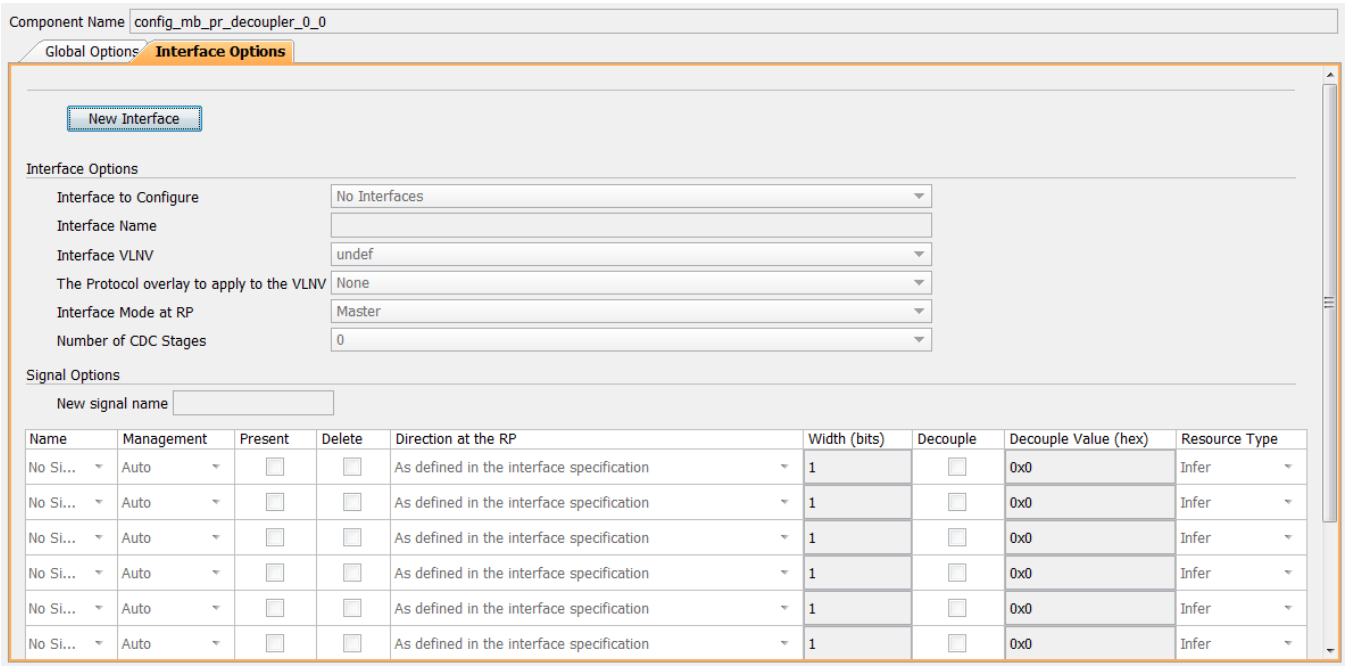

Figure 16: Partial Reconfiguration Decoupler IP – Interface Options

The Global Options are used to set the general AXI interface and Decoupler IP configuration. The Interface Options are used to create custom interfaces and to set the required Signal Options.

Using the Flash Programmer

This module allows firmware update via PCIe without JTAG.

Configuring Debug Logic

Debug logic can be added to the design to enable hardware validation and debug using the Vivado Hardware Manager. The System ILA IP should be configured to add debug logic to the design. It can be used to monitor and validate the running hardware platform on the Xilinx device. Refer to this link in the *Vivado Design Suite User Guide: Designing IP Subsystems Using IP Integrator* (UG994) for more information.

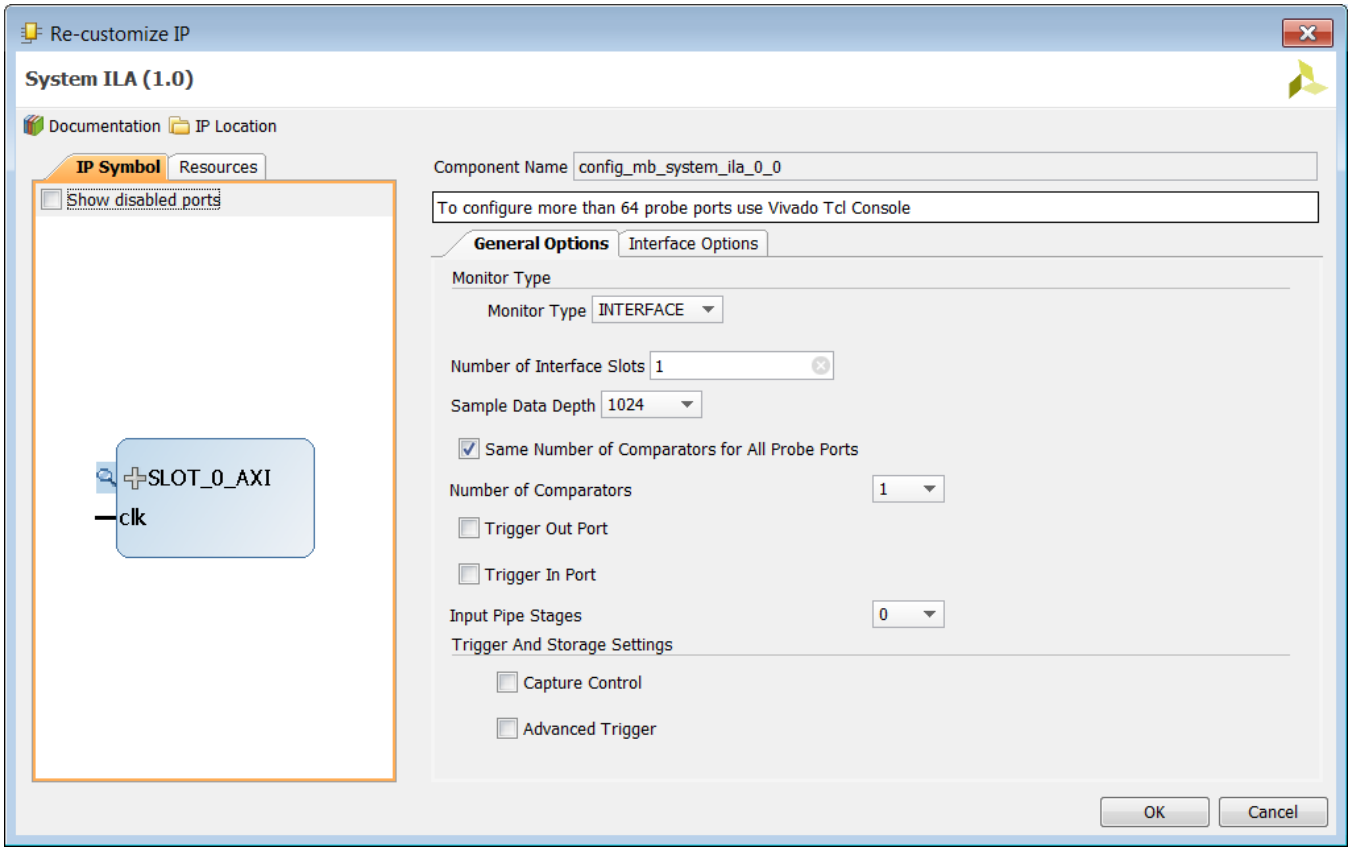

Figure 17: System ILA Debug Logic Configuration Options

Designing for Performance Profiling

The SDAccel OpenCL runtime has integrated device profiling features to enable real-time performance analysis of the board as they are being utilized. Performance statistics from the board can be unified with statistics from the host to get a clearer picture of the performance of the entire system. This empowers you to analyze your device performance and then quickly optimize your host code and kernel source.

To take advantage of this capability, the hardware platform must have the following:

- A performance monitor framework must be added to the hardware platform.
- The framework must follow a specified address mapping.
- The specified format for the data must be followed.
- A shim for the XCL HAL driver API must be implemented. See Using the OpenCL Hardware Abstraction Layer (XCL HAL) Driver API for more information.

The flow presented here is for an example hardware platform and may need to be adapted to meet other design or board specific characteristics.

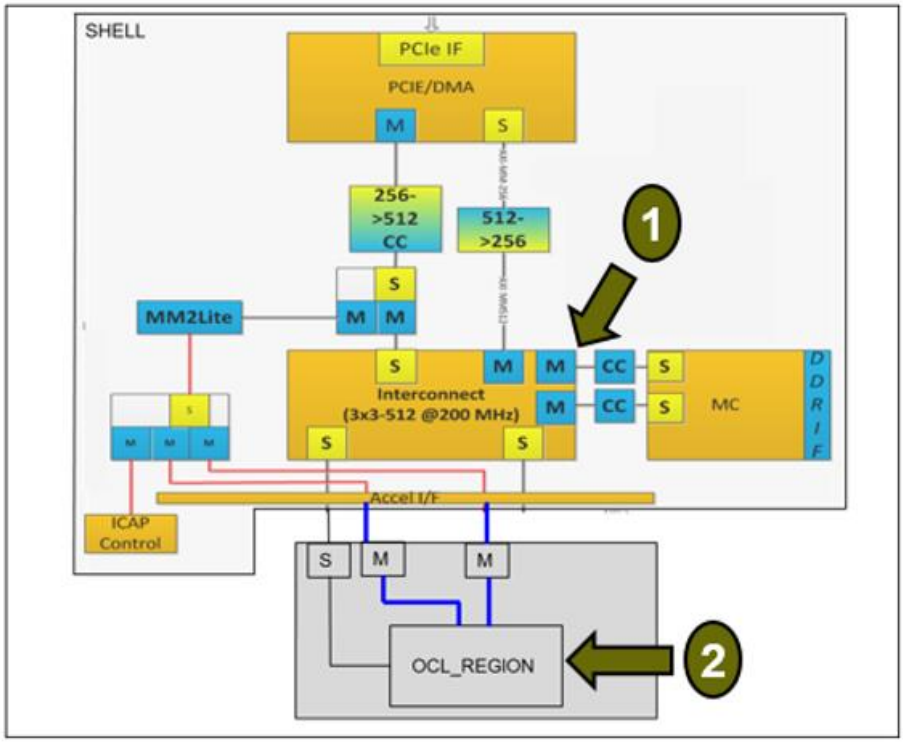

Figure 18: Performance Monitor Points on Hardware Platform

Monitoring the Framework

A performance monitor framework can be inserted into the base hardware platform at the points of interest as shown in Figure 18. The monitor framework is designed from standard Xilinx IP and can be configured and accessed through the hardware platform by the host machine.

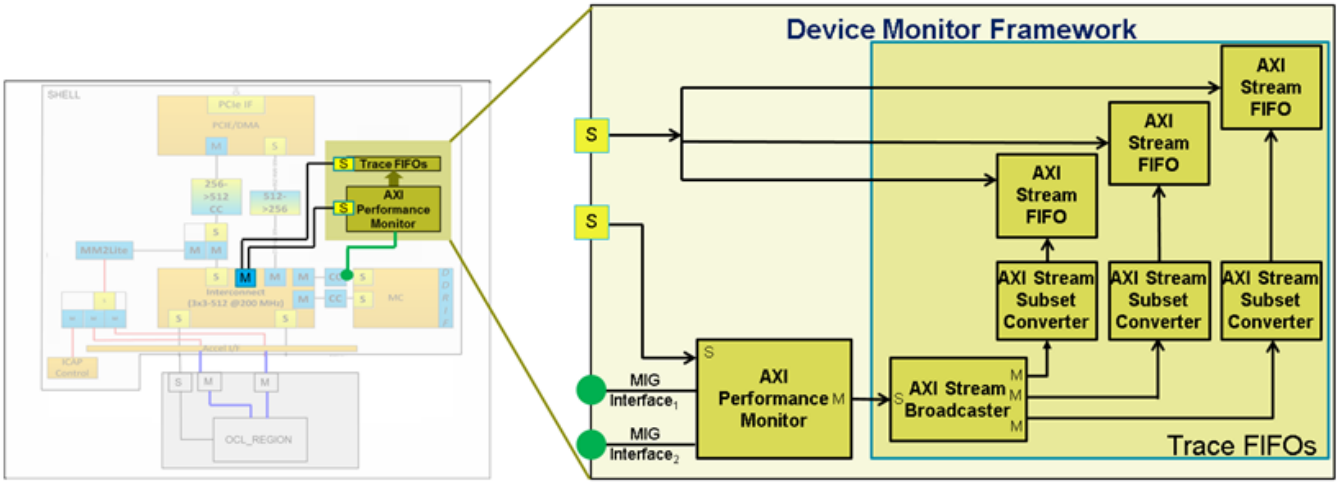

Figure 19: Device Monitor Framework

Figure 19 shows a block diagram for the device monitor framework. It is shown connected to the MIG interface on an example device, but will be similar on other devices. An AXI Performance Monitor (APM) is the IP core used to monitor the AXI transactions and provide performance statistics and events of every monitored AXI connection. See the *AXI Performance Monitor LogiCORE IP Product Guide* (PG037) for more information. For the APM shown in Figure 19, the monitored connections are the two slave interfaces on the MIG memory controller which communicates with the DDR DRAM.

The APM contains two modules: profile metric counters, and trace. The profile metric counters are aggregated counters used to calculate average throughput and latency for both write and read channels. A single AXI-Lite interface is used to configure the APM and read the metric counters. Xilinx recommends that you use the sampled metric counters, which are guaranteed to provide samples on the same clock cycle. This is done by first reading from the sample interval register, then reading the appropriate sampled metric counters.

The APM trace module provides timestamps of AXI transaction events. These events are captured in a collection of individual flags and are output by the APM using an AXI Stream in one of the following ways:

- AXI Lite – slower offload but easier to close timing
- Full AXI – faster offload as it offloads trace at kernel clock rate

AXI Lite

Figure 19 shows how this trace stream is stored using multiple AXI Stream FIFOs (slower offload approach). An AXI Stream broadcaster is used to broadcast the stream to three AXI Stream subset converters. These subset converters split the stream into three different 32-bit values, which are then written to AXI Stream FIFOs. Note that the use of multiple AXI Stream FIFOs is specific to the current available IP: the AXI stream FIFO IP has a 32 bit data path width but the implementation needs 96 bit width.

AXI-4 Memory Mapped (Full AXI)

Figure 20 shows an alternative implementation of the device monitor framework which supports faster trace offload. The AXI Stream FIFO is configured to have a Data Interface set to "AXI4". Besides the AXI Stream slave, this configures the core to have two other slave interfaces:

- AXI4-Lite interface for control
- AXI4 interface for data

Use the AXI4-Lite slave for register accesses such as reading the number of data bytes in the FIFO and resetting the core. In addition, you can use the AXI4 slave interface to offload the data in the FIFO. The data width of this interface can be set to a value up to 512 bits. For example, a PCIe interface such as an Alpha Data card could use the maximum bit width of 256 or 512. This AXI4 slave would then be connected to a DMA or similar master, enabling burst reads and thus a much faster trace offload path.

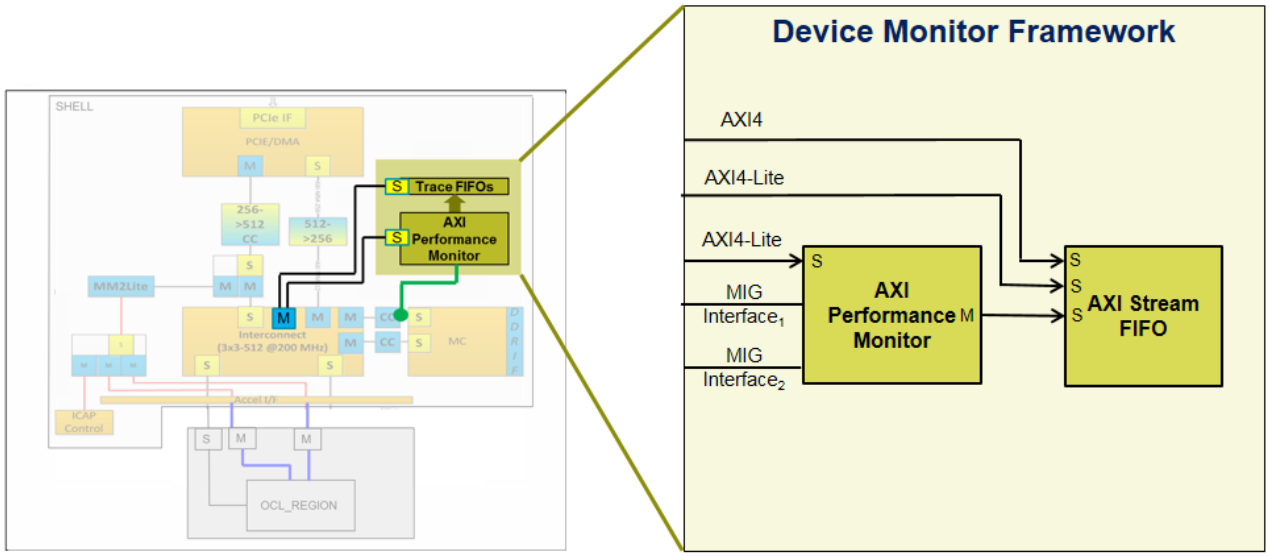

Figure 20: Device Monitor Framework with Faster Trace Offload

Figure 18 shows the two monitor points of interest on an example hardware platform. This is where the above-mentioned monitor framework will be inserted in the base design. Table 1 below lists the interfaces that are being monitored. The MIG DDR interface ports are monitored with one APM. A second APM will monitor activity in the programmable region, including kernel stalls. This second APM is optionally inserted during compilation.

Table 1: Example Monitor Points on Alpha Data Card

Number	Device Interface	Statistics	Source
1	MIG DDR interface ports	Profile: Write/read throughput and latency Trace: Timestamps of AXI transactions	APM in static region
2	Kernel start/done	Trace: Timestamps of start/done events	APM in OCL region

Using Performance Monitor with AXI Lite

In this example design, which uses AXI Lite and multiple FIFOs, AXI Performance Monitor (APM) IP cores are not contained within their own level of hierarchy. However, the following IP cores logically constitute an "apm_sys", which monitors transactions for the purposes of recording and reporting system performance details: xilmonitor_apm, xilmonitor_broadcast, xilmonitor_fifo0, xilmonitor_fifo1, xilmonitor_fifo2, xilmonitor_subset0, xilmonitor_subset1, xilmonitor_subset2.

Address Mapping

In order to access this performance monitoring circuitry, it is important to know the address mapping used for the APM and AXI Stream trace FIFOs.

Table 2: Address Mapping of Device Profile Monitor Framework with AXI Lite Offload

Base Address (Hex)	Description
0x100000	AXI Performance Monitor (APM)
0x110000	AXI Stream FIFO 0 (device trace bits 31:0)
0x111000	AXI Stream FIFO 1 (device trace bits 63:32)
0x112000	AXI Stream FIFO 2 (device trace bits 95:64)

Table 2 above lists the address mapping for the monitor framework connected to the MIG slave ports. The APM slave interface is used to both configure the core as well as read the profile counters. The interfaces to the AXI Stream FIFOs is used to configure the cores, reset the FIFOs, and read/pop values off the data queue.

The address mapping for this example project is as follows.

Cell	Slave Interface	Base Name	Offset Address	Range	High Address
u_ocl_region					
M_AXI (32 address bits : 4G)					
static_region/mig_0	C0_DDR3_S_AXI	C0_DDR3_ADDRESS_BLOCK	0x0000_0000	4G	0xFFFF_FFFF
static_region/dma_pcie_0					
m_axi (64 address bits : 16E)					
static_region/mig_0	C0_DDR3_S_AXI	C0_DDR3_ADDRESS_BLOCK	0x0000_0000_0000_0000	8G	0x0000_0001_FFFF_FFFF
m_axil (22 address bits : 4M)					
static_region/flash_programmer	S_AXI	reg0	0x04_0000	64K	0x04_FFFF
static_region/pr_isolation/axi_gpio_0	S_AXI	Reg	0x03_0000	4K	0x03_0FFF
static_region/axi_hwicap_0	S_AXI_LITE	Reg	0x02_0000	64K	0x02_FFFF
static_region/clk_wiz_1	s_axi_lite	Reg	0x05_0000	64K	0x05_FFFF
static_region/id_field	S_AXI	Reg	0x03_1000	4K	0x03_1FFF
static_region/mig_0	C0_DDR3_S_AXI_CTRL	C0_REG	0x06_0000	64K	0x06_FFFF
u_ocl_region	S_AXI	Reg0	0x00_0000	128K	0x01_FFFF
static_region/xilmonitor_apm	S_AXI	Reg	0x10_0000	64K	0x10_FFFF
static_region/xilmonitor_fifo0	S_AXI	Mem0	0x11_0000	4K	0x11_0FFF
static_region/xilmonitor_fifo1	S_AXI	Mem0	0x11_1000	4K	0x11_1FFF
static_region/xilmonitor_fifo2	S_AXI	Mem0	0x11_2000	4K	0x11_2FFF

Figure 21: Example address mapping for a project

Using Performance Monitor with AXI Full

In this example design, which uses AXI Full and a single FIFO, APM cores are not contained within their own level of hierarchy. However, the following IP cores logically constitute an "apm_sys", which monitors transactions for the purposes of recording and reporting system performance details: xilmonitor_apm, xilmonitor_broadcast, xilmonitor_fifo0, xilmonitor_subset0.

Address Mapping

In order to access this performance monitoring circuitry, it is important to know the address mapping used for the APM and AXI Stream trace FIFO.

Table 3: Address Mapping of Device Profile Monitor Framework with AXI Full Offload

Base Address (Hex)	Description
0x100000	AXI Performance Monitor (APM)
0x110000	AXI Stream FIFO 0 (contains all device trace bits)

Table 3 above lists the address mapping for the monitor framework connected to the MIG slave ports. The APM slave interface is used to both configure the core as well as read the profile counters. The interfaces to the AXI Stream FIFO is used to configure the cores and reset the FIFOs. The reading/popping of values off the data queue is performed using AXI Full.

The address mapping for this example project is as follows.

Cell	Slave Interface	Base Name	Offset Address	Range	High Address
base_region/dma_pcie					
M_AXI (64 address bits : 16E)					
expanded_region/memc/ddrmem_0	C0_DDR4_S_AXI	C0_DDR4_ADDRESS_BLOCK	0x0000_0000_0000_0000	4G	0x0000_0000_FFFF_FFFF
expanded_region/memc/ddrmem_1	C0_DDR4_S_AXI	C0_DDR4_ADDRESS_BLOCK	0x0000_0001_0000_0000	4G	0x0000_0001_FFFF_FFFF
expanded_region/memc/ddrmem_2	C0_DDR4_S_AXI	C0_DDR4_ADDRESS_BLOCK	0x0000_0002_0000_0000	4G	0x0000_0002_FFFF_FFFF
expanded_region/memc/ddrmem_3	C0_DDR4_S_AXI	C0_DDR4_ADDRESS_BLOCK	0x0000_0003_0000_0000	4G	0x0000_0003_FFFF_FFFF
expanded_region/apm_sys/xilmonitor_fifo0	S_AXI_FULL	Mem1	0x0000_0020_0000_0000	4G	0x0000_0020_FFFF_FFFF
M_AXI_LITE (32 address bits : 4G)					
expanded_region/u_ocl_region	S_AXI	Reg0	0x0000_0000	32K	0x0000_7FFF
expanded_region/u_ocl_region	S_AXI	Reg1	0x0000_8000	32K	0x0000_FFFF
expanded_region/u_ocl_region	S_AXI	Reg2	0x0001_0000	32K	0x0001_7FFF
expanded_region/u_ocl_region	S_AXI	Reg3	0x0001_8000	32K	0x0001_FFFF
base_region/pr_isolation_expanded/gate_pr	S_AXI	Reg	0x0003_0000	4K	0x0003_0FFF
base_region/featureid/gpio_featureid	S_AXI	Reg	0x0003_1000	4K	0x0003_1FFF
base_region/pr_isolation_expanded/ddr_calib_status	S_AXI	Reg	0x0003_2000	4K	0x0003_2FFF
expanded_region/pr_support_expanded/flash_programmer	AXI_LITE	Reg	0x0004_0000	4K	0x0004_0FFF
base_region/axi_i2c	S_AXI	Reg	0x0004_1000	4K	0x0004_1FFF
base_region/base_clocking/clkwiz_kernel	s_axi_lite	Reg	0x0005_0000	4K	0x0005_0FFF
base_region/base_clocking/clkwiz_kernel2	s_axi_lite	Reg	0x0005_1000	4K	0x0005_1FFF
expanded_region/memc/ddrmem_0	C0_DDR4_S_AXI_CTRL	C0_REG	0x0006_0000	64K	0x0006_FFFF
expanded_region/memc/ddrmem_2	C0_DDR4_S_AXI_CTRL	C0_REG	0x0007_0000	64K	0x0007_FFFF
expanded_region/memc/ddrmem_3	C0_DDR4_S_AXI_CTRL	C0_REG	0x0008_0000	64K	0x0008_FFFF
base_region/sys_rmgmt_wiz	S_AXI_LITE	Reg	0x000A_0000	64K	0x000A_FFFF
expanded_region/apm_sys/xilmonitor_apm	S_AXI	Reg	0x0010_0000	64K	0x0010_FFFF
expanded_region/apm_sys/xilmonitor_fifo0	S_AXI	Mem0	0x0011_0000	4K	0x0011_0FFF
expanded_region/u_ocl_region					
M_AXI (32 address bits : 4G)					
M00_AXI (34 address bits : 16G)					
expanded_region/memc/ddrmem_0	C0_DDR4_S_AXI	C0_DDR4_ADDRESS_BLOCK	0x0_0000_0000	4G	0x0_FFFF_FFFF
M01_AXI (34 address bits : 16G)					
expanded_region/memc/ddrmem_1	C0_DDR4_S_AXI	C0_DDR4_ADDRESS_BLOCK	0x1_0000_0000	4G	0x1_FFFF_FFFF
M02_AXI (34 address bits : 16G)					
expanded_region/memc/ddrmem_2	C0_DDR4_S_AXI	C0_DDR4_ADDRESS_BLOCK	0x2_0000_0000	4G	0x2_FFFF_FFFF
M03_AXI (34 address bits : 16G)					
expanded_region/memc/ddrmem_3	C0_DDR4_S_AXI	C0_DDR4_ADDRESS_BLOCK	0x3_0000_0000	4G	0x3_FFFF_FFFF

Figure 22: Example address mapping for a project

Monitoring Data Format

It is also important to understand the format of the data that is read from the monitor framework. This includes both the device profile counters as well as the device trace data.

```
typedef struct {
    float SampleIntervalUsec;
    unsigned int WriteBytes[XAPM_MAX_NUMBER_SLOTS];
    unsigned int WriteTranx[XAPM_MAX_NUMBER_SLOTS];
    unsigned int WriteLatency[XAPM_MAX_NUMBER_SLOTS];
    unsigned short WriteMinLatency[XAPM_MAX_NUMBER_SLOTS];
    unsigned short WriteMaxLatency[XAPM_MAX_NUMBER_SLOTS];
    unsigned int ReadBytes[XAPM_MAX_NUMBER_SLOTS];
    unsigned int ReadTranx[XAPM_MAX_NUMBER_SLOTS];
    unsigned int ReadLatency[XAPM_MAX_NUMBER_SLOTS];
    unsigned short ReadMinLatency[XAPM_MAX_NUMBER_SLOTS];
    unsigned short ReadMaxLatency[XAPM_MAX_NUMBER_SLOTS];
} xclCounterResults;
```

Figure 23: Data Structure Used to Store Device Profiling Counter Results

Figure 23 shows the data structure used to store device counters results. For the AXI Performance Monitor core, `XAPM_MAX_NUMBER_SLOTS` = 8. This data structure is populated using the function `xclPerfMonReadCounters` as described in Using the OpenCL Hardware Abstraction Layer (XCL HAL)

Driver API. There are several counters per monitored AXI port: byte count, transaction count, total latency, and minimum/maximum latency. There are counters for both write and read transactions. Also note that some are 16 or 32 bits values. These counters are then used to calculate average throughput and latency for each channel and each port. This calculation and subsequent reporting is performed by the SDAccel runtime.

```
typedef struct {
    unsigned char LogID; /* 0: event flags, 1: host timestamp */
    unsigned char Overflow;
    unsigned short Timestamp;
    unsigned int HostTimestamp;
    unsigned char EventFlags[XAPM_MAX_NUMBER_SLOTS];
    unsigned char ExtEventFlags[XAPM_MAX_NUMBER_SLOTS];
    unsigned char WriteAddrLen[XAPM_MAX_NUMBER_SLOTS];
    unsigned char ReadAddrLen[XAPM_MAX_NUMBER_SLOTS];
    unsigned short WriteAddrId[XAPM_MAX_NUMBER_SLOTS];
    unsigned short ReadAddrId[XAPM_MAX_NUMBER_SLOTS];
} xclTraceResults;

typedef struct {
  unsigned int mLength;
  unsigned int mNumSlots;
  xclTraceResults mArray[MAX_TRACE_NUMBER_SAMPLES];
} xclTraceResultsVector;
```

Figure 24: Data Structure and Vector Used to Store Device Profiling Trace Results

The Figure 24 shows the data structure used to store device trace results. This data structure is populated using the function `xclPerfMonReadTrace` as described in Using the OpenCL Hardware Abstraction Layer (XCL HAL) Driver API.

`LogID` is a single bit that signals the type of data this event contains, while `Timestamp` contains the number of device clock cycles since the last event. `Overflow` is active high only when `Timestamp` has been greater than 216. `HostTimestamp` contains the host timestamp only if `LogID` = 1.

The event flags are a collection of single-bit flags that signal AXI transaction events, while the external event flags are single-bit flags that signal external events (e.g., kernel start/done). These are both further defined in the *AXI Performance Monitor LogiCORE IP Product Guide* (PG037).

Table 4: Bitwise Definition of Device Trace Word

95:70	69:62	61:54	53:47	46:44	43:36	35:28	27:21	20:18	17	16:1	0
Zero padding	Slot 1 AWLEN	Slot 1 ARLEN	Slot flags	Slot 1 ext events	Slot 0 AWLEN	Slot 0 ARLEN	Slot 0 flags	Slot 0 ext events	Over-flow	Time-stamp	Log ID

As previously mentioned, each trace word is stored in three AXI Stream FIFOs. The values from these FIFOs is analyzed then written into the trace results data structure. The table above lists the bitwise definition of the entire trace word. The data shown is for the APM connected to the MIG ports, however, and could be extrapolated to other connections and devices. The APM is connected to two ports and is configured to output AXI lengths as well as the standard event flags and external (ext) events.

This data format is specified by the APM core and further described in the *AXI Performance Monitor LogiCORE IP Product Guide* (PG037). The reporting infrastructure in the SDAccel OpenCL runtime parses the trace data structure for AXI transaction and kernel events and then reports in the device trace log.

Chapter 4 Applying Physical Design Constraints

This section discusses the various types of physical constraints that are needed to support the reconfigurable hardware platform. There are different requirements depending on whether you are using the regular or expanded partial reconfiguration flow.

I/O and Clock Planning

One of the key considerations in the design of a DSA is to identify the I/Os necessary for the board requirements. The Static I/Os are expected to operate and remain live during the device reprogramming of OpenCL kernels. The physical I/O locations will influence performance and must be considered as part of the floorplanning process, especially when SSI devices are used.

It is recommended that the static I/Os are assigned physically outside of the programmable region. There may be trade-offs made between the size and shape of the programmable region to accommodate an area free of static I/Os and overall system performance.

If the floorplan for the static region includes any I/O ports that are used by signals within the programmable region, an IBUF or OBUF needs to be placed in the static region logic design to connect it through the static region to the programmable region.

Refer to the *Vivado Design Suite User Guide: I/O and Clock Planning* (UG899) for more information on I/O and clock planning.

Applying Partial Reconfiguration Constraints

The use of Xilinx Partial Reconfiguration technology requires the use of several types of physical constraints. These include:

- Definition of the reconfigurable region (Partition Definition)
- Floorplanning with physical blocks (Pblock)
- Partition Pin Placement (partPin)

For more information about the partial reconfiguration design flow, and constraint requirements, refer to the *Vivado Design Suite User Guide: Partial Reconfiguration* (UG909).

Definition of the Reconfigurable Region

The programmable region of the hardware platform must be identified in order for the tools to process the logic correctly with partial reconfiguration and in order to create a partial bitstream file. This process

is done differently depending on whether you are using the standard partial reconfiguration (PR) flow or the expanded partial reconfiguration (XPR) flow to define the hardware platform.

Defining the Programmable Region for the Standard PR Slow

For the standard PR flow, the reconfigurable region is identified by enabling the **Use Partial Reconfiguration** checkbox on the *SDAccel OpenCL Programmable Region* IP in the IP Integrator block design, as shown in Figure 25 below.

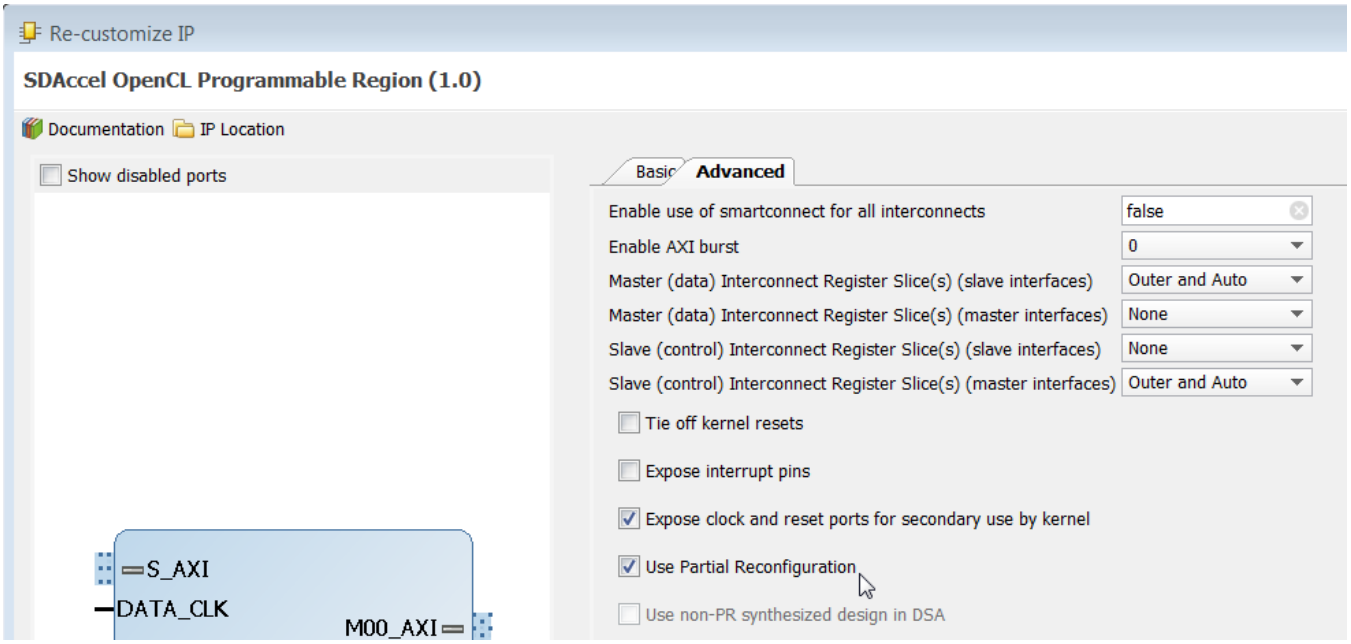

Figure 25: Setting the Reconfigurable Region for a PR Hardware Platform

This causes the HD. RECONFIGURABLE property to be set to TRUE for the *SDAccel OpenCL Programmable Region* IP. The partial reconfiguration capabilities of the Vivado Design Suite use this property to identify the programmable region when processing the design.

Defining the Programmable Region for the Expanded PR Flow

For the expanded PR flow, you will directly assign the HD. RECONFIGURABLE property on the hierarchical module that defines the expanded region. The HD.RECONFIGURABLE property is set to TRUE in the XDC constraints file on the expanded region hierarchical block as shown below.

```
# Floorplanning
# ------------------------------------------------------------------------------
set_property DONT_TOUCH true [get_cells xcl_design_i/expanded_region]
set_property HD.RECONFIGURABLE true [get_cells xcl_design_i/expanded_region]
```

Figure 26: Defining Properties for the Expanded Region

IMPORTANT: *Notice the DONT_TOUCH property is also set on the expanded region hierarchical module to prevent optimization across the boundary of the module.*

Floorplanning

When using Xilinx Partial Reconfiguration technology, floorplanning is required to separate the static and programmable regions of the design. The floorplanning strategy can dramatically affect the performance of the design. It can be an iterative process to find the optimal floorplan for your specific design and device. Refer to the *Vivado Design Suite User Guide: Design Analysis and Closure Techniques* (UG906) for more information on defining floorplan regions.

Floorplanning is performed by assigning logic modules or cells to physical block ranges (Pblocks) on the device canvas. Floorplanning can be performed interactively by opening the Synthesized design in the Vivado IDE or with constraints in the XDC file. Occasionally, in order to optimize performance or device resources, multiple Pblocks are used.

Pblocks can be defined as rectangular regions, or by combining multiple rectangles to define a non-rectangular shape. Pblocks can also be nested to enable lower levels of logic to be further constrained to specific regions of the device. If nested Pblocks are desired, ensure the lower level Pblock region is completely within the upper level Pblock region.

TIP: *Overlapping Pblocks should be avoided.*

In both standard PR and expanded PR hardware platform designs, the programmable region must be entirely contained within a Pblock to facilitate partial reconfiguration.

Floorplanning PR Hardware Platforms

When designing a standard PR hardware platform, the static region includes all the logic on the hardware platform except the programmable region. You can use Pblocks to constrain the programmable region hierarchy, leaving the rest of the design for the static logic. In some cases, floorplanning the static logic can also be done to improve performance. Multiple logic modules from the static region can be assigned to a single Pblock, but the programmable region Pblock can only be assigned a single programmable region module.

Pblock size is important for the overall system performance. Rather than packing a Pblock as tightly as possible, it is often necessary to leave the Pblock loosely utilized to enable the programmable region logic to be optimally placed within it for performance reasons. Iterations are often needed to find the optimal Pblock sizes.

Individual logic elements, such as BRAMs and DSPs from the static region, can also be locked onto specific sites on the device fabric by manually placing these elements onto the device and fixing their location. Refer to the *Vivado Design Suite User Guide: Design Analysis and Closure Techniques* (UG906) for more information on manual placement.

Floorplanning Expanded PR Hardware Platforms

When designing an expanded PR hardware platform, only a small static region is needed for the PCIe/DMA base logic. The rest of the hardware platform logic, including the programmable region, gets implemented as part of a single expanded PR region hierarchy. Multiple Pblock rectangles should be used for the expanded region logic to reserve and use as much of the device as possible. Ensure as many of the DSP and BRAM device resources are included in the expanded region Pblock area.

You will adjust the Pblock shapes to make the static region as small as possible near the I/O banks assigned to it. Multiple iterations may be required to see how tightly you can pack the static region.

Partition Pin Assignment

Partition pins need to be assigned for the programmable region interface pins to act as anchor points for the interface signal routing. They are automatically assigned by the implementation tools within the programmable region, along the border of the programmable and static regions.

In order to ensure uncongested routing, consider the number and expected location of partition pins when shaping Pblocks. It is often necessary to include enough of an edge along the sides of the Pblock region to enable sparse placement of the partition pins. There are only a limited number of partition pins available in any given device resource tile. You should take care to ensure enough area is provided in the Pblock region to accommodate partition pin placement.

Defining Partition Pin Ranges

If routing congestion is encountered, you can manually define placement ranges for partition pin assignments using one of the following methods:

Case 1: Define a single PartPin Range for all pins

```
set_property HD.PARTPIN_RANGE {SLICE_X1Y1:SLICE_X2Y2} [get_pins <rp_cell>/*]
```

Case 2: Define multiple unique PartPin Ranges for various interface pins of the programmable region:

```
set_property HD.PARTPIN_RANGE {SLICE_X1Y1:SLICE_X2Y2} \
[get_pins <rp_cell>/AXIM_*]
set_property HD.PARTPIN_RANGE {SLICE_X1Y1:SLICE_X2Y2} \
[get_pins <rp_cell>/AXIS_*]
```

IMPORTANT: *The assignment of partition pins is done automatically by the tool. You should only manually define partition pin ranges when necessary to improve the quality of the routing results.*

You can identify partition pins in the design using the example Tcl command below:

```
select_objects [get_pins -of [get_cells -hier -filter HD.RECONFIGURABLE]]
```

Refer to the *Vivado Design Suite User Guide: Partial Reconfiguration* (UG909) for more information on working with partition pins.

Chapter 5 Implementing the Hardware Platform Design

The hardware platform design should be implemented and validated to ensure it works as expected in the SDAccel flow. The first step in that validation process should be to ensure the hardware platform design itself is performing as expected. This can be done using test kernel logic to populate the programmable region.

Simulating the Design

The Vivado Design Suite has extensive logic simulation capabilities to enable block or system level validation of the design. Available third party FPGA simulation tools are also supported. Refer to the *Vivado Design Suite User Guide: Logic Simulation* (UG900) for more information.

Implementation and Timing Validation

The design should be synthesized and implemented to ensure desired performance is achieved. It is often required to iterate on floorplanning and implementation strategies to ensure optimal performance.

It is often important to implement, analyze, and iterate on the hardware platform design to ensure that it continues to meet timing during kernel implementation. Using a test kernel, implement the design and then check that the design meets timing by opening the Implemented Design.

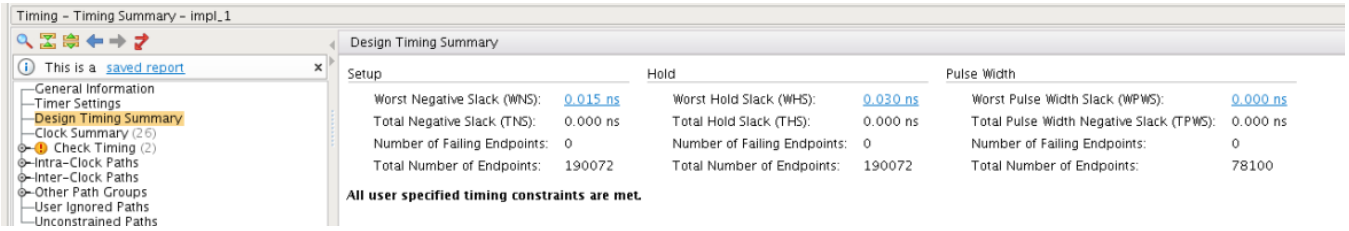

Figure 27 Checking Implementation Results

The floorplan can be examined and modified if need be to optimize implementation results. Refer to the *Vivado Design Suite User Guide: Design Analysis and Closure Techniques* (UG906) for more information.

Chapter 6 Generating DSA Files

After completing your hardware platform design and generating a valid bitstream using the Vivado Design Suite, you are ready to create a Device Support Archive (DSA) file for use with the SDAccel Development Environment. A DSA is a single-file capture of the hardware platform, to be handed off by the platform developer for use with the SDAccel Environment.

Using Example DSAs

Several example DSAs are shipped with the SDAccel Development Environment that can be found in the following location:

<*SDx Environments Installation*>/SDx/2016.3/platforms

Xilinx recommends adhering to the directory structure and file formats used with the example designs when creating your own hardware platforms for use with the SDAccel Environment. See Validating the DSA.

Setting Vivado Properties for Generating a DSA

Prior to creating a DSA file for the hardware platform, you must define various properties in the hardware platform design so that they are included in the DSA. These include metadata properties to help identify the DSA, as well as design properties to define system configuration.

IMPORTANT: *These properties are not saved in the Vivado project so they must be set each time a DSA is created. You can create a TCL script to run prior to creating the DSA.*

The following three properties are required by the `write_dsa` Tcl command, and must be defined prior to generating a DSA. If these properties are not found, the `write_dsa` command will issue an error and stop. The values shown are example values.

```
set_property dsa.vendor "xilinx" [current_project]
set_property dsa.name "1ddr" [current_project]
set_property dsa.boardId "adm-board" [current_project]
```

The following property can be used to override the SPI Flash type associated with a DSA, and influences the generation of the MCS file from the bitstream. The default value is `bpix16`. For more information on flash memory support and the MCS file, refer to the *Vivado Design Suite User Guide: Programming and Debug* (UG908).

```
set_property dsa.flash_interface_type "spix8" [current_project]
```

The following properties have default values but can be defined to specify the DSA file version, enable the partial reconfiguration (PR) flow, or capture the synthesis checkpoint within the DSA.

```
set_property dsa.version "1.0" [current_project]
```

- Default value is 0.0

```
set_property dsa.uses_pr true [current_project]
```

- Default value is true
- Possible values are true and false

```
set_property dsa.static_synth_checkpoint false [current_project]
```

- Default value is false
- Possible values are true and false

The following properties can be used to set the PCIe Id and Board attributes for the Board section if a board file is not yet available.

```
set_property dsa.pcie_id_vendor "0x10ee" [current_project]
set_property dsa.pcie_id_device "0x8038" [current_project]
set_property dsa.pcie_id_subsystem "0x0011" [current_project]
set_property dsa.board_name "alpha-data.com:adm-pcie3-ku3:1.0" [current_project]
set_property dsa.board_interface_type "gen3x8" [current_project]
set_property dsa.board_memories {{ddr3 8GB} {ddr4 16GB} [current_project]
set_property dsa.board_interface_name "PCIe" [current_project]
set_property dsa.board_vendor "alpha-data.com" [current_project]
```

If defined, these property values will take precedence over the values that come from the board files.

The default value for these properties is an empty string. Using a local board repository can be avoided by using these properties.

Setting Parameter for Expanded Region

When using an expanded partial reconfiguration hardware platform, the following parameter is also required:

```
set_param dsa.expandedPRRegion 1
```

The parameter can be set in the Vivado IDE prior to opening the Vivado project for the hardware platform, or it can be set in the `init.tcl` that is sourced each time the Vivado Design Suite is launched. For more information on the `init.tcl` file refer to the *Vivado Design Suite Tcl Command Reference Guide* (UG835).

Generating the DSA File

IMPORTANT: *After creating the DSA file you should retain the source Vivado Design Suite project files so you can recreate or update the DSA file as needed.*

Once the design has been implemented, and the required properties have been set, you can generate a DSA file using the `write_dsa` command. This creates an archive of the hardware platform that contains all the relevant files and data needed by the SDAccel Development Environment.

The `write_dsa` command will also create a bitstream file if one has not yet been created.

After all parameters are properly set up for the DSA, run the following Tcl command in the Vivado Tcl Console:

```
write_dsa <filename>.dsa -include_bit
```

Where <*filename*> should be something meaningful that references the board and its characteristics. However, the filename is optional. If omitted, a filename is constructed from the DSA properties as:

```
${DSA.VENDOR}_${DSA.BOARD_ID}_${DSA.NAME}_${DSA.VERSION}.DSA
```

TIP: *The DSA file format is a zip file and can be opened with the appropriate tools in order to check the contents and assist further development.*

Validating the DSA

You can use the `validate_dsa` command to validate a custom DSA file to ensure it contains the proper content and metadata needed to support the hardware platform in the SDAccel environment. Use the following command to validate a DSA file:

```
validate_dsa <dsa file> -verbose
```

Assembling the Platform

Once the DSA file has been created for the hardware platform, it needs to be associated with the software platform configuration files and drivers. Xilinx recommends using the same directory structure as used with the supplied example hardware platforms.

Platform Directory Structure

Each hardware platform should be contained in a directory with a name that corresponds to the DSA name. Naming the folder after the DSA file is just a recommendation, but it does help identify the hardware platforms from the directory names.

The hardware platform directory contains a `<platform name>.xpfm` file and subdirectories called `./sw` and `./hw`, as shown in Figure 28. Notice the DSA file is stored in the `hw` folder.

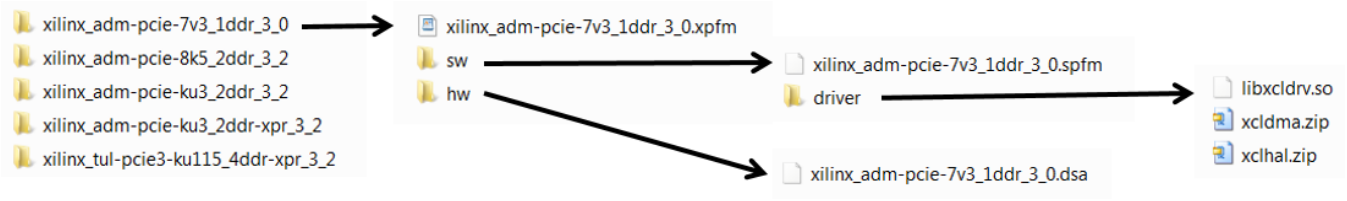

Figure 28: Platform Directory Structure

Configuring the Hardware Platform .xpfm File

The `<platform name>.xpfm` file, as shown in the figure below, defines the location of the hardware and software components of the hardware platform. The example XPFM files can be copied to use as a template for a custom hardware platform. The file can be edited to reflect an alternate directory structure.

```
<?xml version="1.0" encoding="UTF-8"?>
<sdx:platform sdx:vendor="xilinx"
              sdx:library="adm-pcie-7v3"
              sdx:name="1ddr"
              sdx:version="3.0"
              xmlns:sdx="http://www.xilinx.com/sdx">
  <sdx:description>
  This is a place-holder for a detailed description of the
hardware/software platform. The platform creator can provide
detailed information about the hardware platform. Which Fpga
family does this platform use? Is it PCIe based? Is it a 1ddr,
2ddr or 4ddr configuration? What are the target domains?
Acceleration, ADAS, Vision etc.
  </sdx:description>

  <sdx:hardwarePlatforms>
    <sdx:hardwarePlatform sdx:path="hw" sdx:name="xilinx_adm-
pcie-7v3_1ddr_3_0.dsa"/>
  </sdx:hardwarePlatforms>

  <sdx:softwarePlatforms>
    <sdx:softwarePlatform sdx:path="sw" sdx:name="xilinx_adm-
pcie-7v3_1ddr_3_0.spfm"/>
  </sdx:softwarePlatforms>

</sdx:platform>
```

Figure 29: XPFM File Contents

TIP: *Avoid using absolute paths in the XPFM file to ensure portability of the platform.*

Populating the Hardware Platform Directory

The hardware platform is defined in the DSA file. The DSA is a ZIP file that can be extracted for examination.

Copy or move the DSA file into the `./hw` folder of the hardware platform directory structure.

The DSA file contains the following information:

- `dsa.xml` – contains DSA hardware platform settings and configuration from when the `write_dsa` Tcl command was run, and the DSA file was created.
- *`<platform_name>`*`.bit` – Full bitstream file for initial system configuration.
- *`<platform_name>`*`_clear.bit` – For UltraScale architecture only, this is a file for clearing the full bitstream from the device.
- *`<platform_name_pblock_name>`*`.bit` – Placeholder partial bitstream of the test kernel for the programmable region logic. The partial bitstream for the programmable region is dynamically created by the SDAccel Development Environment and will replace the one included in the DSA file.
- *`<platform_name>`*`.dcp` – Routed version of the hardware platform design.

  NOTE: The DCP file could also be a synthesized checkpoint file if you are not using the partial reconfiguration flow.
- *`<platform_name>`*`.png` – Optional picture of the board to display in Vivado and SDx projects. Multiple images at different resolutions can be included.

Populating the Software Platform Directory

The software platform is contained in the `./sw` folder of the hardware platform directory structure. The `./sw` folder contains the *`<platform name>`*`.spfm` file and a `./driver` subdirectory, as shown in Figure 28.

The *`<platform name>`*`.spfm` file, as shown in the figure below, defines information for the software platform.

```
<?xml version="1.0" encoding="UTF-8"?>
<sdx:platform sdx:vendor="xilinx"
              sdx:library="adm-pcie-7v3"
              sdx:name="1ddr"
              sdx:version="3.0"
              xmlns:sdx="http://www.xilinx.com/sdx">
    <sdx:description>
      TBD
    </sdx:description>
    <sdx:systemConfigurations>
        <sdx:configuration sdx:name="linux_x86"
                           sdx:displayName="Linux on x86">
            <sdx:description>Linux on x86</sdx:description>
        </sdx:configuration>
    </sdx:systemConfigurations>
</sdx:platform>
```

Figure 30: Software Platform .spfm File

The `./driver` subdirectory contains the driver configurations defined for the software platform, as shown in Figure 28. Refer to the Software Platform Design section form more information on defining drivers.

Using Older DSA Versions

The format of the DSA file was changed in the 2016.3 release to decouple the hardware platform from the software platform. Previous versions had both the hardware and software files combined within a single DSA file. Previous DSA versions are still supported and work as defined.

IMPORTANT: *If you use an old DSA version file in the new directory structure, as defined above, it will enable the separation of the software and hardware platforms. The DSA will be read for the hardware platform, and the software platform will be looked for in the `./sw` folder.*

Updating an Older Version DSA

To update an older format DSA to the new DSA format, open the original Vivado hardware platform project, upgrade the IP, implement the design, and generate a new DSA.

Chapter 7 Software Platform Design

Introduction

The SDAccel runtime software is layered on top of a common low-level software interface called the Hardware Abstraction Layer (HAL). The HAL driver provides APIs to runtime software which abstract the xcldma driver details. The xcldma driver is a kernel mode DMA driver which interfaces to the memory-mapped platform over PCIe.

Figure 31 shows the layers of the software platform:

- The Linux Kernel Mode DMA Driver for PCI Express (xcldma)
- The OpenCL Hardware Abstraction Layer (XCL HAL)
- The Xilinx SDAccel Runtime Software

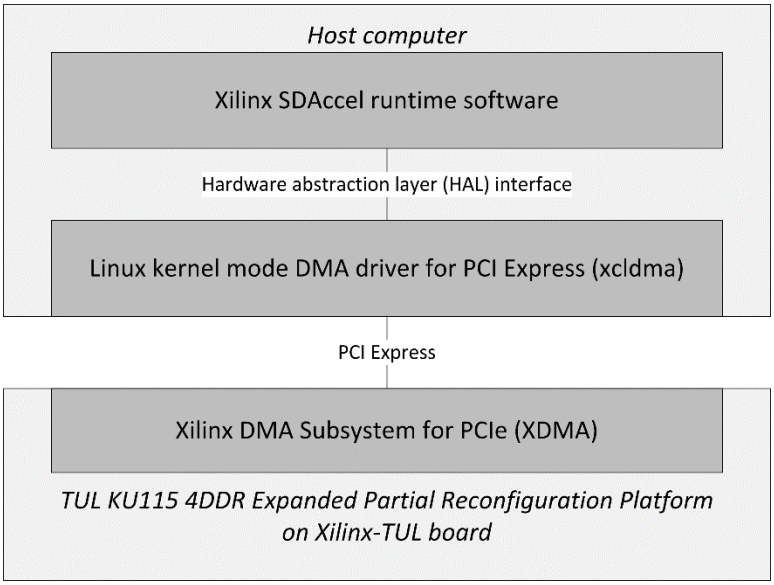

Figure 31: Software Platform Layers

Using the Kernel Mode Driver (xcldma)

The Linux kernel mode driver (xcldma) driver is included with the SDx installation. It manages the PCIe DMA engine and provides other functionality.

When the hardware platform is implemented as a DSA and used with the SDAccel Development Environment, the hardware abstraction layer (HAL) driver insulates the SDAccel runtime software from the implementation details of the xcldma driver.

TIP: *Users who do not use SDAccel or the provided HAL driver can still interact with the hardware platform's memory-mapped IP cores by using the provided xcldma driver.*

The Linux kernel mode driver (xcldma) supports all the boards released by Xilinx and is available for use under the GNU GPL License. If you are creating your own hardware platform, you will start by modifying the Xilinx xcldma driver.

The xcldma kernel mode driver is used to manage the device (the accelerator), and provide essential services such as:

- Provide DMA from host to device and device to host.
- Allow the clients to memory map the device registers which are exposed on PCIe Base Address Register (BAR).
- Provide bitstream download capabilities.
- Enable kernel clock scaling.
- Provide access to sensors on the device via SysMon (temperature, voltage, etc.).
- Reports MIG calibration status.
- Enable resetting and rebooting the board from PROM.

The DMA operations are provided by Linux pread/pwrite system call interface. Other services are provided by Linux ioctl interface which are defined in header file `xdma-ioctl.h` which is included in `xcldma.zip`. The archive can be found in `plaforms/<dsa>/sw/driver` of the SDx installation area.

The xcldma kernel mode driver, which is used with the Xilinx DMA Subsystem for PCI Express (XDMA) IP, has the following standard system call interfaces which are used by the XDMA HAL driver:

```
// perform IO Control operations
int ioctl(int d, unsigned long request, ...);

//reading data from the device
ssize_t pread(int fd, void *buf, size_t count, off_t offset);

// sending data to the device
ssize_t pwrite(int fd, const void *buf, size_t count, off_t offset);
```

The driver exposes 2 sets of read channels and 2 sets of write channels via device nodes:

```
$ tree /dev/xdma/
/dev/xdma/
├── xdma0_c2h_0
├── xdma0_c2h_1
```

```
├── xdma0_control
├── xdma0_h2c_0
├── xdma0_h2c_1
└── xdma0_user
```

- `c2h` implies card-to-host transfer channel and `h2c` implies host-to-card transfer. 0 and 1 are channel numbers respectively.
- `pread` and `pwrite` system calls are used to DMA buffers to and back from the device.
- `xmda_control` node exposes the DMA registers, please note these should not be used to directly program from the user space (application should use pread and pwrite to perform DMA).
- `xdma_user` node is used to memory map PCIe user BAR which exposes OCL region's control port, AXI gate, APMs, clock wizard, and other peripherals.
- `ioctl` system calls are used to perform the following operations:
    - XDMA_IOCINFO - Get Device Information - legacy
    - XDMA_IOCINFO2 – Additional device Information - recommended
    - XDMA_IOCICAPDOWNLOAD - Download xclbin via ICAP
    - XDMA_IOCMCAPDOWNLOAD - Download xclbin via MCAP
    - XDMA_IOCHOTRESET - Perform PCIe Hot Reset of the device
    - XDMA_IOCOCLRESET - Perform OCL Region reset
    - XDMA_IOCFREQSCALING – Perform clock frequency scaling – legacy
    - XDMA_IOCFREQSCALING2 - Perform multiple clock frequency scaling – recommend
    - XDMA_IOCREBOOT – Perform a reboot of the device and load from PROM

XDMA Source Files

The files that comprise the xcldma driver are listed below, with a brief description of the file:

```
driver
|-- include
|   `-- xclbin.h – Defines the XCLBIN container format
`-- xcldma
    |-- include
    |   |-- mcap_registers.h – Defines the MCAP registers used to program UltraScale devices
    |   |-- perfmon_parameters.h – Defines APM registers
    |   |-- xbar_sys_parameters.h – Defines registers and constants for various peripherals
    |   `-- xdma-ioctl.h – defines the IOCTL commands and structures
    `-- kernel
        |-- 10-xcldma.rules – Linux udev rule (typically resides in /etc/udev/rules.d/)
        |-- makefile – the makefile to build the drive
        |-- xdma-bit.c – routines for downloading bitstreams
        |-- xdma-core.c – DMA programming routines
        |-- xdma-core.h – header file for the DMA programming routines
        |-- xdma-ioctl.c – defines IOCTRL routines
        |-- xdma-sgm.c – DMA programming routines
        `-- xdma-sgm.h – DMA programming routines
```

Using the OpenCL Hardware Abstraction Layer (XCL HAL) Driver API

Xilinx OpenCL Hardware Abstraction Layer or XCL HAL Driver API is required by the OpenCL runtime to communicate with the hardware platform. It is used for downloading Xilinx FPGA bitstreams, allocating/de-allocating OpenCL buffers, migrating OpenCL buffers between host memory and hardware platform memory, and communicating with the OpenCL kernel on its control port.

The API supports address spaces which may be used for accessing device peripherals with their own specific memory mapped ranges. A hardware platform may optionally have a flat memory space which can be used to address all peripherals on the card.

A HAL Driver is typically layered on top of a Linux kernel mode PCIe driver. The API has a C-style interface and is defined in the header file `xclhal.h`. XCL HAL driver should be multi-threading safe.

User modifications to the xcldma driver or replacement of the kernel mode driver may necessitate a new or modified HAL driver.

Understanding the HAL Driver API

The HAL Driver API is comprised of C-header and source files which are delivered with the SDx installation. The files are located in <install>/ SDx/2016.3/data/sdaccel/pcie/src/ in the `xclhal.zip` file.

Device Access Operations

- `unsigned xclProbe()`

  xclProbe should return a count of physical devices present in system that are supported by the HAL driver. The same driver may optionally support devices of different kinds.

- `xclDeviceHandle xclOpen(unsigned deviceIndex, const char *logFileName, xclVerbosityLevel level)`

  xclOpen opens the device specified by deviceIndex. If logFileName is not NULL then HAL driver is supposed to log messages into the log file. The verbosity of messages is specified by level. The API returns a device handle that should be used for all other API operations. If the open fails, the driver should return `nullptr`. Note that once a device is opened the driver should lock the device so that if another process tries to open the same device it should get a `nullptr` back. xclVerbosityLevel can have the following values:

  - XCL_QUIET - Do not log any messages
  - XCL_INFO - Log all info, warning and error messages
  - XCL_WARN - Only Log warning and error messages
  - XCL_ERROR - Only log error messages

- `void xclClose(xclDeviceHandle handle)`

  xclClose is used to close the device which was previously opened. The driver should ensure that if any OpenCL Compute Units are still running when the device is closed, then OCL Region is reset.

- `int xclResetDevice(xclDeviceHandle handle, xclResetKind kind)`

  xclResetDevice is used to reset the device. Possible kinds of reset are:

  XCL_RESET_KERNEL: Performs a soft reset which resets the OCL Region. This stops any running kernels. Buffers allocated on the device are released.

  XCL_RESET_FULL: Resets the full device including the Memory Controller and the DMA controller. This is typically done via PCIe hot reset, and may be useful if the device `seems hung`.

- `int xclGetDeviceInfo2(xclDeviceHandle handle, xclDeviceInfo2 *info)`

  xclGetDeviceInfo2 is used to get information about the device. Caller provides a `struct` which is populated by the API. For details of the `struct` see the xclhal.h in the xclhal.zip. This supersedes the xclGetDeviceInfo, which will be deprecated in a future release.

- int xclLockDevice(xclDeviceHandle handle)

  xclLockDevice is used to get exclusive ownership of the device. The lock is necessary before performing buffer migration, register access or bitstream downloads. If the device is already locked by another user xclLockDevice will return non-zero status.

- int xclReClock2(xclDeviceHandle handle, unsigned short region, const unsigned short *targetFreqMHz);

  xclReClock2 is used to change the frequency of the clocks driving the OCL region (up to two clocks is supported).

Bitstream Loading Operation

Bitstreams are packaged in xclbin file format by SDAccel compiler. The file format is defined in header file `xclbin.h`. For more details on the xclbin format. There are two APIs which may be used by OpenCL runtime to load bitstreams:

- `int xclLoadBitstream(xclDeviceHandle handle, const char *fileName)`

  xclLoadBitstream is used by the runtime to pass the name of a file whose contents are in xclbin format. This API should extract the bitstream from the file and download it to the FPGA using one of ICAP or MCAP interfaces. This function will be deprecated in a future release, you should use the xclLoadXclBin instead.

- `int xclLoadXclBin(xclDeviceHandle handle, const xclBin *buffer)`

  xclLoadXclBin is used by the runtime to directly pass a xclbin buffer in memory. This is more efficient since an additional file IO operation is avoided.

Buffer Management

Buffer management APIs are used for managing device memory. The DSA vendors are required to provide a memory management with the following 4 APIs.

- `uint64_t xclAllocDeviceBuffer(xclDeviceHandle handle, size_t size)`

  xclAllocDeviceBuffer allocates a buffer of the given size on the card and return the offset of the buffer in the RAM as return value. The offset acts as buffer handle. Note that the runtime will subsequently pass the returned handle to OpenCL kernel which would use it to perform bus master read and write operations on the allocated buffer in the RAM. The host will not write to this memory directly. If there are no free blocks remaining, the function should return 0xffffffffffffffff.

- `uint64_t xclAllocDeviceBuffer2(xclDeviceHandle handle, size_t size, xclMemoryDomains domain, unsigned flags)`

  xclAllocDeviceBuffer2 allocates a buffer of the given size on the specified DDR bank on the k card and returns the offset of the buffer in the RAM as a return value. The offset acts as buffer handle. Note that the runtime subsequently passes the returned handle to the OpenCL kernel that would use it to perform bus master read and write operations on the allocated buffer in the RAM. The host will not write to this memory directly. If there are no free blocks remaining, the function should return 0xffffffffffffffff. xclAllocDeviceBuffer2 helps improve throughput between compute units and DDR by placing buffers in different DDR banks which can be read or written in parallel.

- `void xclFreeDeviceBuffer(xclDeviceHandle handle, uint64_t buf)`

  xclFreeDeviceBuffer frees the memory previously allocated by xclAllocDeviceBuffer or xclAllocDeviceBuffer2. The freed memory may be reallocated in a subsequent call to xclAllocDeviceBuffer or xclAllocDeviceBuffer2. It is an error to pass a buffer handle which was previously not allocated by xclAllocDeviceBuffer or xclAllocDeviceBuffer2.

- `size_t xclCopyBufferHost2Device (xclDeviceHandle handle, uint64_t dest, const void *src, size_t size, size_t seek)`

  xclCopyBufferHost2Device copies the contents of host buffer into the DDR resident destination buffer. The term `src` refers to host buffer pointer and the term `dest` refers to the device buffer handle. It is an error to pass a `dest` handle which was previously not allocated by xclAllocDeviceBuffer. The term `seek` specifies how many bytes to skip at the beginning of the destination before copying `size` bytes of the host buffer. It is an error to pass `size` where `size` plus `seek` is greater than the size of device buffer previously allocated. It is assumed that DSA would use PCIe DMA to migrate the buffer.

- `size_t xclCopyBufferDevice2Host(xclDeviceHandle handle, void *dest, uint64_t src, size_t size, size_t skip)`

  xclCopyBufferDevice2Host copies the contents from DDR resident buffer to host buffer. The term `src` refers to device buffer handle and the term `dest` refers to the host buffer pointer. It is an error to pass a `src` handle which was previously not allocated by xclAllocDeviceBuffer. The term `skip` specifies how many bytes to skip from the beginning of the source before copying `size` bytes of the device buffer. It is an error to pass `size` where `size` plus `skip` is greater than the size of the device buffer previously allocated. It is assumed that DSA would use PCIe DMA to migrate the buffer.

Device Profiling

For additional details refer to Designing for Performance Profiling.

The following `enum` values define the type of device profiling that is requested for each function below:

```
enum xclPerfMonType {
   XCL_PERF_MON_MEMORY = 0,
   XCL_PERF_MON_HOST_INTERFACE = 1,
```

```
    XCL_PERF_MON_OCL_REGION = 2
};
```

- `double xclGetDeviceClockFreqMHz(xclDeviceHandle handle)`

  xclGetDeviceClockFreqMHz provides the clock frequency (in MHz) of the device. The value returned is a double. This is used for converting device trace timestamps to host timestamps.

- `double xclGetReadMaxBandwidthMBps(xclDeviceHandle handle)`

  xclGetReadMaxBandwidthMBps provides the maximum read bandwidth (in MBps) between the host and the device. This is used in profile reporting.

- double xclGetWriteMaxBandwidthMBps(xclDeviceHandle handle)

  xclGetWriteMaxBandwidthMBps provides the maximum write bandwidth (in MBps) between the host and the device. This is used in profile reporting.

- size_t xclPerfMonClockTraining(xclDeviceHandle handle , xclPerfMonType type)

  xclPerfMonClockTraining performs clock training from the host to the device. A 64-bit timestamp will be written to the software-written data register of the AXI Performance Monitor (APM)[1] as two 32-bit words. Since this register is 32 bits, the higher 32 bits will be written first, and then the lower 32 bits. The return value is the number of bytes transferred to/from the device.

- size_t xclPerfMonStartCounters(xclDeviceHandle handle, xclPerfMonType type)

  xclPerfMonStartCounters starts the device profile counters. If an APM is used on a hardware device, then this involves:

  1. Cycling the reset on the metric counters.
  2. Enabling the metric counters.
  3. Specifying the metric counters to not reset after reading.
  4. Reading the sample register to reset the sample interval. The return value is the number of bytes transferred to/from the device.

- size_t xclPerfMonStopCounters(xclDeviceHandle handle, xclPerfMonType type)

  xclPerfMonStopCounters stops the device profile counters. They will not restart until the next call to xclPerfMonStartCounters. If an APM is used on a hardware device, then this involves disabling the metric counters. The counters should not be reset so that a final read can be performed. The return value is the number of bytes transferred to/from the device.

- size_t xclPerfMonReadCounters(xclDeviceHandle handle, xclPerfMonType type, xclCounterResults& counterResults)

[1] *AXI Performance Monitor LogiCORE IP Product Guide* (PG037).

xclPerfMonReadCounters reads the device profile counters. The unsigned int pointer hostBuf is the host buffer where the results will be written. The counterResults data structure is shown in Figure 23. The return value is the number of bytes transferred to/from the device.

- size_t xclPerfMonStartTrace(uint32_t startTrigger, xclPerfMonType type)

  xclPerfMonStartTrace starts the device trace. If an APM is used on a hardware device, then this involves: 1. Cycling the reset on the APM trace stream FIFO; 2. Cycling the reset on the AXI stream FIFOs; 3. Performing clock training by invoking a call to xclPerfMonClockTraining; and 4. Write to the trigger register (not currently implemented). The unsigned int startTrigger will contain the value to write to the trigger register. The return value is the number of bytes transferred to/from the device.

- size_t xclPerfMonStopTrace(xclDeviceHandle handle, xclPerfMonType type)

  xclPerfMonStopTrace stops the device trace. Trace will not restart until the next call to xclPerfMonStartTrace. If an APM is used on a hardware device, then this involves:

  1. Disabling the trace; and
  2. Cycling the reset on the AXI stream FIFOs.

  The return value is the number of bytes transferred to/from the device.

- uint32_t xclPerfMonGetTraceCount(xclDeviceHandle handle, xclPerfMonType type)

  xclPerfMonGetTraceCount obtains the number of trace words available for reading. The return value is the number of words in the device trace AXI stream FIFOs.

- size_t xclPerfMonReadTrace(xclDeviceHandle handle, xclPerfMonType type, xclTraceResultsVector& traceVector)

  xclPerfMonReadTrace reads the device trace. The traveVector is a vector of trace data structures as shown in Figure 24. The return value is the number of bytes transferred to/from the device.

Device Read and Write Operations

Device Read or Write operations are used to access device peripherals usually on AXI-Lite interface. These operations can use flat addressing or relative addressing. The following `enum` values define address spaces which can be extended to add more peripherals:

```
enum xclAddressSpace {
        XCL_ADDR_SPACE_DEVICE_FLAT = 0,
        XCL_ADDR_SPACE_DEVICE_RAM = 1,
        XCL_ADDR_KERNEL_CTRL = 2,
        XCL_ADDR_SPACE_DEVICE_PERFMON = 3,
        XCL_ADDR_SPACE_MAX = 8
};
```

- `size_t xclWrite(xclDeviceHandle handle, xclAddressSpace space, uint64_t offset, const void *hostBuf, size_t size)`

  xclWrite copies the contents of host buffer hostBuf to a specific location in the card address map. It is used to program card peripherals. For example, the runtime uses this API to send the arguments to the OpenCL kernel. The offsets are relative to the address space. All the sizes and offsets are 32 bits (4 bytes) aligned.

- `size_t xclRead(xclDeviceHandle handle, xclAddressSpace space, uint64_t offset, void *hostbuf, size_t size)`

  xclRead copies data from a specific location in the card address map to host buffer hostBuf. It will be used to read status of card peripherals. For example, the runtime uses this API to determine if the OpenCL kernel is finished running. The offsets are relative to the address space. All the sizes and offsets are 32 bits (4 bytes) aligned.

Device Flash Operations

These APIs can be used to update the flash on your board and forcing a boot from the flash.

- `int xclUpgradeFirmware2(xclDeviceHandle handle, const char *file1, const char* file2)`

  xclUpgradeFirmware2 is used to flash a new firmware to the board. This is done over PCIe so no JTAG cables or physical access to the board is required. For boards using QSPI you must specify primary (file1) and secondary (file2) mcs files. For BPI flash specify file1, file2 should be NULL.

- `int xclBootFPGA(xclDeviceHandle handle)`

  xclBootFPGA will force a card to boot from the flash. The driver utilizes the PCIe AER to boot the FPGA without requiring a system reboot.

Using the OpenCL Runtime Flow

The SDAccel OpenCL runtime provides OpenCL 1.2 embedded profile conformant runtime API. The runtime is layered on top of common low-level software interface also called the Xilinx OpenCL Hardware Abstraction Layer or XCL HAL. The SDAccel runtime source code is not provided.

The SDAccel OpenCL runtime uses a layered hierarchy to stitch together the components which make up the OpenCL system. The diagram in Figure 32 explains the roles of the different layers.

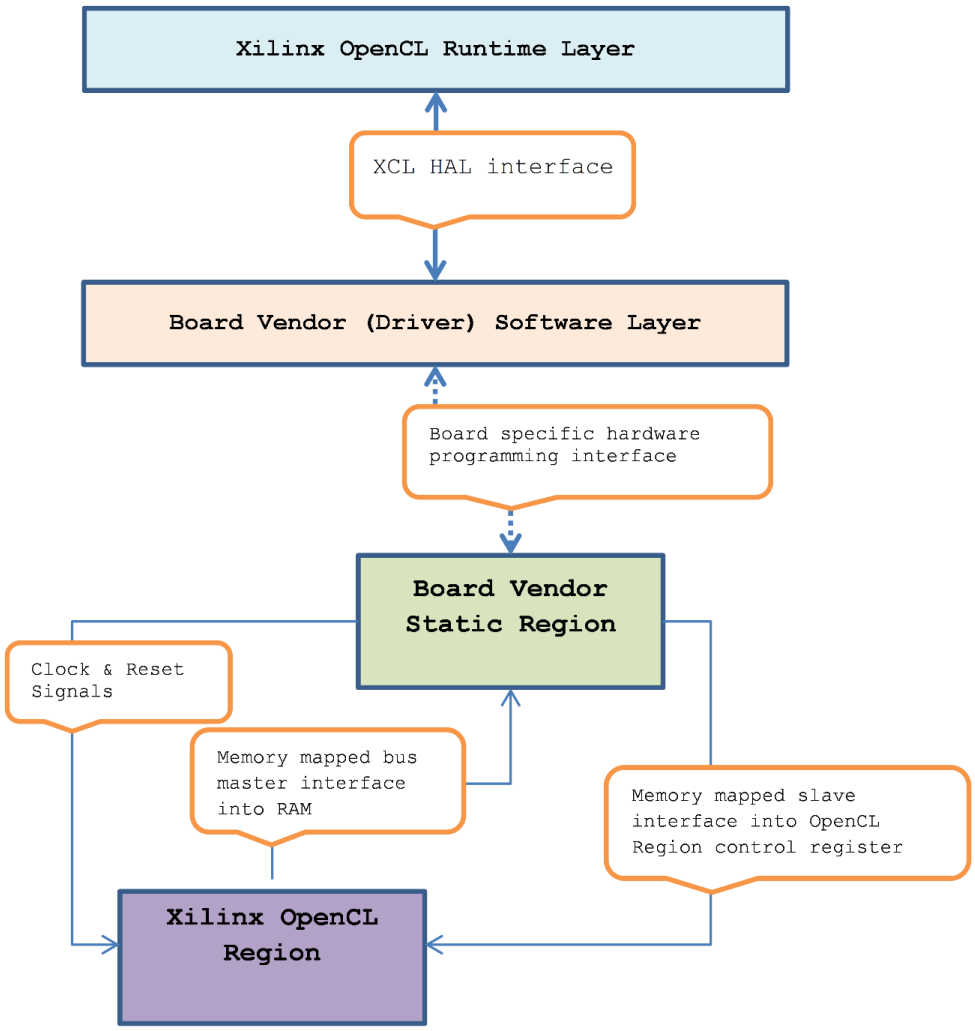

Figure 32: OpenCL DSA Layers

OpenCL Kernel Execution Steps

1. Initialize runtime.
2. OpenCL application downloads bitstream to the device.
3. OpenCL application allocates OpenCL buffers in host memory.
4. Runtime effects PCIe DMA transfer to send the buffer from host memory to device memory.
5. Runtime signals the kernel to start via the AXI Slave port.
6. Runtime starts polling the device to monitor for done signal.
7. Kernel executes and loads/stores data from device memory.
8. When kernel is finished running, it changes the status to done.
9. Runtime effects PCIe DMA transfer to read updated buffers from device memory to host memory.
10. OpenCL application reads the buffer received from memory.

Runtime Library and Driver Interaction

The table below shows the interaction between the different layers of the OpenCL stack. The sequence of the operations is typical of an OpenCL application timeline.

Table 5: Bitwise Definition of Device Trace Word

	Runtime Library	HAL Driver	XDMA Kernel Driver	Hardware
1	clGetPlatformId	xclProbe	open	
2	clGetDeviceId	xclOpen	open, mmap	
3	clCreateContext	xclLockDevice	flock	
4	clCreateProgramWithBinary	xclLoadXclBin, xclReclock2	ioctl	ICAP/MCAP, ClockWiz
5	clCreateBuffer	xclAllocDeviceBuffer		
6	clEnqueueMigrateMemObjects	xclCopyBufferHost2Device	pwrite	DMA, MIG, DDR
7	clEnqueueNDRange	xclWrite,	mmap IO	OCL Ctrl Port
8	clFinish	xclRead	mmap IO	OCL Ctrl Port
9	clEnqueReadBuffer	xclCopyBufferDevice2Host	pread	DMA, MIG, DDR
10	clReleaseDevice	xclClose	close	

1. Application initializes runtime
   The driver enumerates the devices(s)
2. Application opens the device.
   HAL driver opens the DMA channel nodes and memory maps the PCIe BAR space, which exposes OCL Blocks Control Ports.
3. Application creates context on the device.
   The HAL driver locks the device so that another user cannot use the same device until it is unlocked.
4. Application loads binary program.

   The xclbin file is sent to the HAL driver, which forwards it to the XDMA kernel mode driver via ioctl. The driver then downloads the bitstream using ICAP for 7 Series and MCAP for UltraScale.
5. Application allocates OpenCL buffers in host memory.

   The Device memory manager in the HAL driver allocates memory on the device DDR.
6. Application writes to the buffer.

   Runtime updates the cache of the buffer in host memory, and may effect PCIe DMA transfer to send the host buffer to device memory.
7. Application starts a kernel

   Runtime ensures all buffers specified as kernel arguments have been transferred to the device. If not it transfers all the necessary buffers to the device memory via PCIe DMA transfer. It would then kick off the kernel via write to the status register of the kernel.
8. Application waits for kernel to finish.

   Runtime starts polling the kernel status register to monitor done signal.
9. Application reads from the output buffer.

   Runtime transfers the buffer from device memory to buffer cache in the host. It then returns the contents of the cache to the client.
10. Application releases the device.

    Runtime closes the device handle

Kernel Control Register Map

OpenCL compiler generates a custom register map for every kernel compiled. The register map is used to pass addresses of OpenCL buffers in the device memory, scalar arguments to the kernel function and to control the kernel through control signals. It is also used to pass the group id and group offset required by the OpenCL specification. Please refer to *Vivado Design Suite User Guide High-Level Synthesis* (UG902), which describes register maps generated by HLS. Below is a sample register map generated by compiler.

```
// control
// 0x00 : Control signals
//        bit 0  - ap_start (Read/Write/COH)
//        bit 1  - ap_done (Read/COR)
//        bit 2  - ap_idle (Read)
//        bit 3  - ap_ready (Read)
//        bit 7  - auto_restart (Read/Write)
//        others - reserved
// 0x04 : Global Interrupt Enable Register
//        bit 0  - Global Interrupt Enable (Read/Write)
//        others - reserved
// 0x08 : IP Interrupt Enable Register (Read/Write)
//        bit 0  - Channel 0 (ap_done)
//        bit 1  - Channel 1 (ap_ready)
//        others - reserved
// 0x0c : IP Interrupt Status Register (Read/TOW)
//        bit 0  - Channel 0 (ap_done)
//        bit 1  - Channel 1 (ap_ready)
//        others - reserved
// 0x10 : Data signal of group_id_x
//        bit 31~0 - group_id_x[31:0] (Read/Write)
// 0x14 : reserved
// 0x18 : Data signal of group_id_y
//        bit 31~0 - group_id_y[31:0] (Read/Write)
// 0x1c : reserved
// 0x20 : Data signal of group_id_z
//        bit 31~0 - group_id_z[31:0] (Read/Write)
// 0x24 : reserved
// 0x28 : Data signal of global_offset_x
//        bit 31~0 - global_offset_x[31:0] (Read/Write)
// 0x2c : reserved
// 0x30 : Data signal of global_offset_y
//        bit 31~0 - global_offset_y[31:0] (Read/Write)
// 0x34 : reserved
// 0x38 : Data signal of global_offset_z
//        bit 31~0 - global_offset_z[31:0] (Read/Write)
// 0x3c : reserved
// 0x40 : Data signal of matrix
//        bit 31~0 - matrix[31:0] (Read/Write)
// 0x44 : reserved
// 0x48 : Data signal of maxIndex
//        bit 31~0 - maxIndex[31:0] (Read/Write)
// 0x4c : reserved
// 0x50 : Data signal of s1
//        bit 31~0 - s1[31:0] (Read/Write)
// 0x54 : reserved
// 0x58 : Data signal of s2
//        bit 31~0 - s2[31:0] (Read/Write)
// 0x5c : reserved
// (SC = Self Clear, COR = Clear on Read, TOW = Toggle on Write, COH = Clear on
Handshake)
```

Chapter 8 Bring-up Tests

Bring-up tests are unit tests which help isolate issues during a new DSA Bring-up phase. They help test individual XCL HAL driver API implementations, hardware functionality, hardware efficiency and the working of the OpenCL runtime. Each hardware platform should deliver a set of bring-up test to be performed after installation and setup.

With the board installed, DSA programmed, and SDx utilities indicating compatibility and readiness of the device, it can be useful to compile and execute bring-up tests which utilize the SDAccel compilation flow. This process provides the next level of confidence that the platform is compatible and operational with partial reconfiguration, the SDAccel flow and its implemented kernels, and the software stack.

Xilinx Bring-up Tests

Xilinx provides source code with build scripts for the Xilinx provided DSAs delivered with SDAccel. This can be a good place to start in developing bring-up tests specific to your hardware platform. The set of bring-up tests should exercise low-level functionality of the accelerator device through their implementation as kernels in the SDAccel compilation flow, download as partial bitstreams to the operating platform, and communication with the host computer. When compiled and run on the host computer containing the operational Xilinx Development Board for Acceleration with programmed DSA, each test will run host executable code, communicate with the implemented kernel on the accelerator device, and report a pass/fail status.

Refer to the bring-up test README file, packaged with the reference designs, for more information on building and running the tests delivered.

Installing a Hardware Platform Board

Install the board in a PCIe Gen3 x8-compatible slot on the host computer. Then use the `xbinst` utility included with the SDx installation to prepare board installation files, and program the configuration memory using the Vivado Hardware Manager. Finally, install the device drivers on the host computer. As a reference see the *SDAccel Development Environment User Guide* (UG1023) for detailed instructions on the installation and setup process for supported accelerator cards.

TIP: *When using the `xbinst` utility with a platform and DSA not included with the SDx installation such as one built from the reference design, the `-f` switch must be used to specify the location of the platform .xpfm which contains the `.dsa` file.*

To test for basic compatibility of the host computer with the installed board, use the Linux `lspci` command to see if the board is detected and registered.

Here is an example of output from a Xilinx supported board:

```
22:00.0 Memory controller: Xilinx Corporation Device 8438
```

Another Linux utility that may be helpful is `dmesg`.

Xilinx Delivered Debug Tools

For DSAs delivered with the Xilinx SDx Development Environments two utilities are provided:

- Xilinx Board Swiss Army Knife (`xbsak`)
- The SDx system check utility (sdxsyschk)

To determine the present operational status of the platform in more detail, including the HAL driver version, PCIe IDs, global memory details, FPGA temperature and voltage, compute unit status, and more, use the `xbsak` utility included with the SDx installation. Note that the PCIe IDs – Vendor, Device, SDevice (Subsystem) – must match those indicated in the host connectivity for compatibility with the xcldma driver.

To test for basic compatibility of the host computer with the installed board, programmed device, and installed drivers – and to confirm that DMA access to DDR memory is enabled by the platform – use the `sdxsyschk` (might need to use with `sudo`) utility included with the SDx installation. Currently, the sdxsyschk utility only works with DSAs delivered with SDx.

Currently these utilities only work with the DSAs delivered with SDx. Please contact your Xilinx representative to enquire about getting source code for the `xbsak` tool. The `sdxsyschk` is a Python utility and can be copied and modified as needed.

Xilinx Board Swiss Army Knife (xbsak)

Xilinx Board Swiss Army Knife (`xbsak`) utility is a standalone command line tool that can perform the following board administration and debug tasks independent of SDx runtime library:

- Board administration tasks
  - Flash PROM
  - Reboot boards without rebooting the host
  - Reset hung boards
  - Query board status, sensors and PCIe AER registers
  - Setting clock frequency feature for kernel
- Debug operations
  - Download SDAccel binary (.xclbin) to FPGA
  - DMA test for PCIe bandwidth
  - Show status of compute units

The `xbsak` tool is found in the runtime/bin directory of the SDx installation.

Command Options

The following are the xbsak command line format and details on the options:

```
xbsak <command> [options]
```

help

Print help messages

list

List all supported devices installed on the server in the format of `[device_id]:device_name`.

The following is an example output where the device ID and device Name are specified:

```
[0] xilinx:xil-accel-rd-ku115:4ddr-xpr:3.2
```

query [-d device_id] [-r region_id]

Query the specified device and programmable region on the device to get detailed status information.

`-d device_id` : specify the target device. Default=0 if not specified. (Optional)

`-r region_id` : specify the target region. Default=0 if not specified. (Optional)

boot [-d device_id]

Boot device from PROM and retrain the PCIe link without rebooting the host.

`-d device_id` : specify the target device. Default=0 if not specified. (Optional)

clock [-d device_id] [-r region_id] [-f clock1_freq] [-g clock2_freq]

Set frequencies of clocks driving the computing units

`-d device_id` : Specify the target device. Default=0 if not specified. (Optional)

`-r region_id` : Specify the target region. Default=0 if not specified. (Optional)

`-f clock1_freq` : Specify clock frequency in MHz for the first clock.
All platforms have this clock. (**Required**)

`-g clock2_freq` : Specify clock frequency in MHz for the second clock. Some platforms have this clock to support IP based kernels. (Optional)

`dmatest [-d device_id] [-b blocksize]`

Test throughput of data transfer between the host machine and global memory on the device.

`-d device_id` : Specify the target device. Default=0 if not specified. (Optional)

`-b blocksize` : Specify the test block size in KB. Default=65536 (64MB) if not specified. The block size can be specified in both decimal or hexadecimal formats. e.g. Both "-b 1024" and "-b 0x400" set the block size to 1024KB (1MB). (Optional)

`flash [-d device_id] -m primary_mcs [-n secondary_mcs]`

Program PROMs on the device with specified configuration files.

`-d device_id` : Specify the target device. Default=0 if not specified. (Optional)

`-m primary_mcs` : Specify the primary configuration file. All platforms have at least one PROM. (**Required**)

`-n secondary_mcs` : specify the secondary configuration file. Some platform have two PROMs and the secondary configuration file is required for the second PROM. (Optional)

IMPORTANT: *The flash programming function requires certain hardware features in the platform to work, so it only works with devices already programmed with the supported DSAs as noted in the following table:*

Board	DSAs
ADM-PCIE-7V3	xilinx:adm-pcie-7v3:1ddr:3.0 or newer
ADM-PCIE-KU3	xilinx:adm-pcie-ku3:2ddr-xpr:3.2 or newer xilinx:adm-pcie-ku3:2ddr:3.2 or newer xilinx:adm-pcie-ku3:1ddr:3.2 or newer
XIL-ACCEL-RD-KU115	xilinx:xil-accel-rd-ku115:4ddr-xpr:3.2 or newer

`program [-d device_id] [-r region_id] -p xclbin`

Download the OpenCL binary to the programmable region on the device

`-d device_id` : Specify the target device. Default=0 if not specified. (Optional)

`-r region_id` : Specify the target region. Default=0 if not specified. (Optional)

`-p xclbin` : Specify the OpenCL binary file. (**Required**)

`reset [-d device_id] [-r region_id]`

Reset the programmable region on the device. All running compute units in the region will be stopped and reset.

`-d device_id` : Specify the target device. Default=0 if not specified. (Optional)

`-r region_id` : Specify the target region. Default=0 if not specified. (Optional)

SDx system check utility (sdxsyschk)

The `sdxsyschk` command is a utility script created in the Python language intended to perform analysis on system and hardware setup on the devices and platforms supported by Xilinx SDAccel Development Environment.

When the script is executed, it compiles and generates an informative status report. It also provides basic debug and troubleshooting capabilities to the users by issuing suggestive messages in descriptive manners.

The `sdxsyschk.py` script is located in the `bin` directory of SDx installation.

There are two general section categories from the diagnosis:

- System and Environment diagnosis
- PCIe Platform Setup Diagnosis (critical status of a target Xilinx PCIe device will be analyzed)

Requirements:

The `sdxsyschk` script has some requirements which must be met:

1. Use Python version 2.7.5 and above to run this script
2. Have `lspci` package installed either under `/sbin` or `/usr/bin` directory

Example Usages:

- To query system and Xilinx PCIe hardware statistic status

```
sudo python ./sdxsyschk.py
```

- To query status and redirect the output to a text file specified with argument

```
sudo python ./sdxsyschk.py -f <path_and_file_name>
```

- To query basic system status plus environment variables info (must be run without `sudo` access)

```
python ./sdxsyschk.py -e
```

Chapter 9 Code Samples

The SDx Environments software installation includes several example header files for reference. They can easily be copied and modified for custom hardware platform designs. The code examples are located at the following location:

```
<SDx_Install_Area>/SDx/2016.3/runtime/driver/include
```

The examples include:

- `xclbin.h` Header file - defines the `xclbin` file format
- `xclhal.h` and `xclhal2.h` – defines the XCL HAL driver API
- `xclperf.h` - defines the AXI Performance Monitor API

Legal Notices

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

AUTOMOTIVE PRODUCTS (IDENTIFIED AS “XA” IN THE PART NUMBER) ARE NOT WARRANTED FOR USE IN THE DEPLOYMENT OF AIRBAGS OR FOR USE IN APPLICATIONS THAT AFFECT CONTROL OF A VEHICLE (“SAFETY APPLICATION”) UNLESS THERE IS A SAFETY CONCEPT OR REDUNDANCY FEATURE CONSISTENT WITH THE ISO 26262 AUTOMOTIVE SAFETY STANDARD (“SAFETY DESIGN”). CUSTOMER SHALL, PRIOR TO USING OR DISTRIBUTING ANY SYSTEMS THAT INCORPORATE PRODUCTS, THOROUGHLY TEST SUCH SYSTEMS FOR SAFETY PURPOSES. USE OF PRODUCTS IN A SAFETY APPLICATION WITHOUT A SAFETY DESIGN IS FULLY AT THE RISK OF CUSTOMER, SUBJECT ONLY TO APPLICABLE LAWS AND REGULATIONS GOVERNING LIMITATIONS ON PRODUCT LIABILITY.