SDAccel Environment Debugging Guide

UG1281 (v2019.1) May 22, 2019

See all versions of this document

XILINX

Revision History

The following table shows the revision history for this document.

Section	Revision Summary
05/22/2019 Version 2019.1	
Entire document	Minor editorial changes.
01/24/2019 Version 2018.3	
Entire document	Minor editorial changes.
12/05/2018 Version 2018.3	
Defensive Programming	Added an explanation of a defensive programming technique.
Typical Errors Leading to Application Hangs	Added a new section about the causes of application hangs.
Debugging a MicroBlaze Processor (RTL Kernels Only)	Added a new section about how to enable MicroBlaze debugging in RTL kernel block designs.
Connecting to a MicroBlaze Processor in an RTL Kernel over XVC	New section.
Complete Command Line Debug Example	Added a new example section to illustrate command line debug.
10/02/2018 Version 2018.2.xdf	
Throughout document	Changed `xbsak` to `xbutil`.
07/02/2018 Version 2018.2	
Entire document	Minor editorial changes.
06/06/2018 Version 2018.2	
General updates	Initial Xilinx release.

Table of Contents

Introduction to Debugging in SDAccel

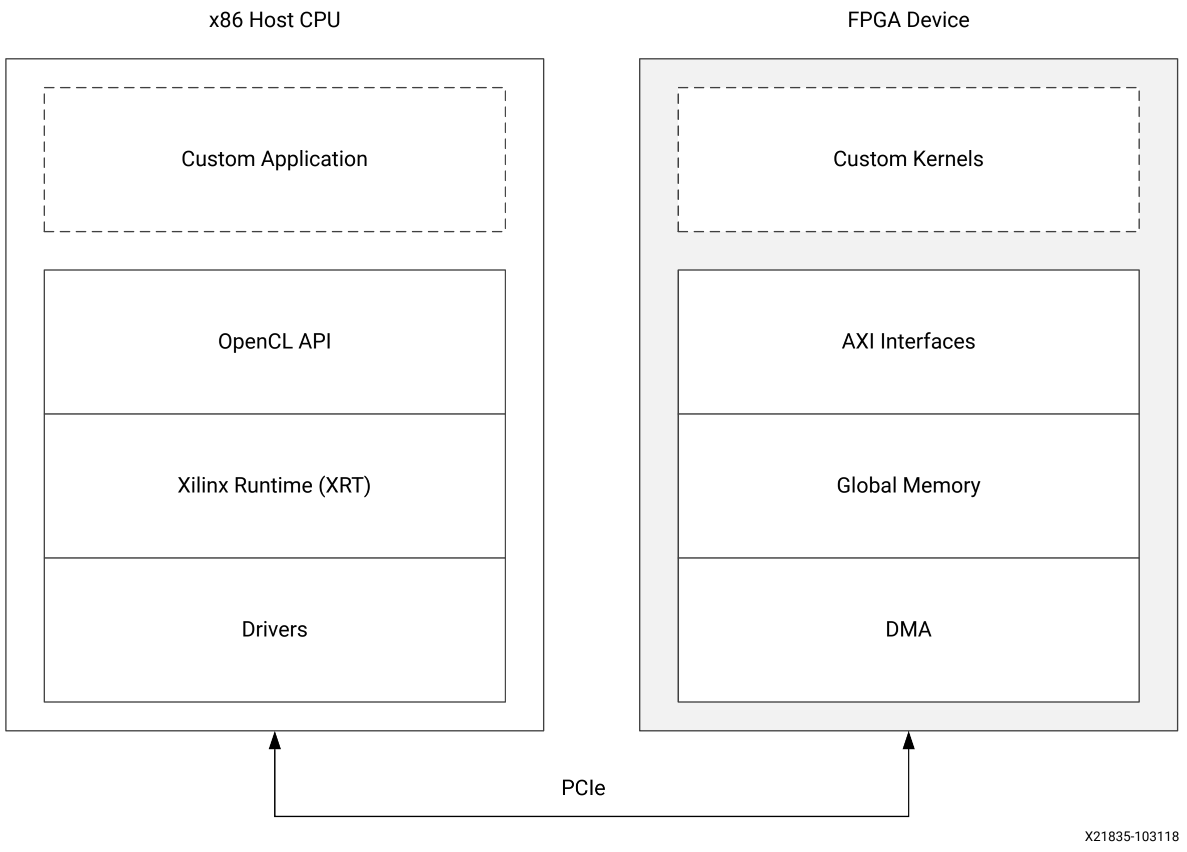
This document is intended to introduce the debugging capabilities of the SDAccel™ environment. The goal is to provide detailed instructions on how to analyze any failure encountered within the SDAccel flow. If no tool problem is encountered and the behavior of the design is deemed functionally correct, look for answers in the *SDAccel Environment Profiling and Optimization Guide* (UG1207) to determine if the performance of the design can be further improved.

SDAccel Execution Model

In the SDAccel framework, an application program is split between a host application and hardware accelerated kernels with a communication channel between them. The host application, written in C/C++ and using API abstractions like OpenCL, runs on an x86 server while hardware accelerated kernels run within the Xilinx FPGA. The API calls, managed by the Xilinx runtime (XRT), are used to communicate with the hardware accelerators. Communication between the host x86 machine and the accelerator board, including control and data transfers, occurs across the PCIe bus. While control information is transferred between specific memory locations in hardware, global memory is used to transfer data between the host application and the kernels. Global memory is accessible by both the host processor and hardware accelerators, while host memory is only accessible by the host application.

For instance, in a typical application, the host will first transfer data, to be operated on by the kernel, from host memory into global memory. The kernel would subsequently operate on the data, storing results back to the global memory. Upon kernel completion, the host would transfer the results back into the host memory. Data transfers between the host and global memory introduce latency which can be costly to the overall acceleration. To achieve acceleration in a real system, the benefits achieved by hardware acceleration kernels must outweigh the extra latency of the data transfers. The general structure of this acceleration platform is shown in the following figure.

Figure 1: **Architecture of an SDAccel Application**



The FPGA hardware platform, on the right-hand side, contains the hardware accelerated kernels, global memory, and the DMA to transfer. Kernels can have one or more global memory interfaces and are programmable. The SDAccel execution model can be broken down into these steps:

1. The host application writes the data needed by a kernel into the global memory of the attached device through the PCIe interface.
2. The host application programs the kernel with its input parameters.
3. The host application triggers the execution of the kernel function on the FPGA.
4. The kernel performs the required computation while reading and writing data from global memory, as necessary.
5. The kernel writes data back to global memory and notifies the host that it has completed its task.
6. The host application reads data back from global memory into the host memory and continues processing as needed.

The FPGA can accommodate multiple kernel instances at one time; this can occur between different types of kernels or multiple instances of the same kernel. The XRT transparently orchestrates the communication between the host application and the kernels in the accelerator. The number of instances of a kernel is determined by compilation options.

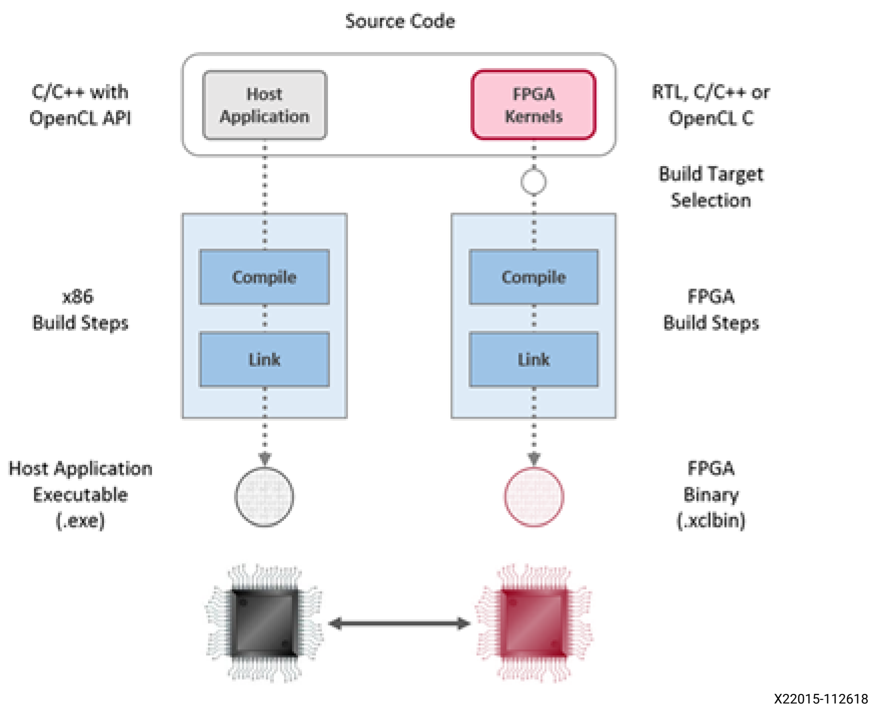
SDAccel Build Process

The SDAccel environment offers all of the features of a standard software development environment:

- Optimized compiler for host applications
- Cross-compilers for the FPGA
- Robust debugging environment to help identify and resolve issues in the code
- Performance profilers to identify bottlenecks and optimize the code

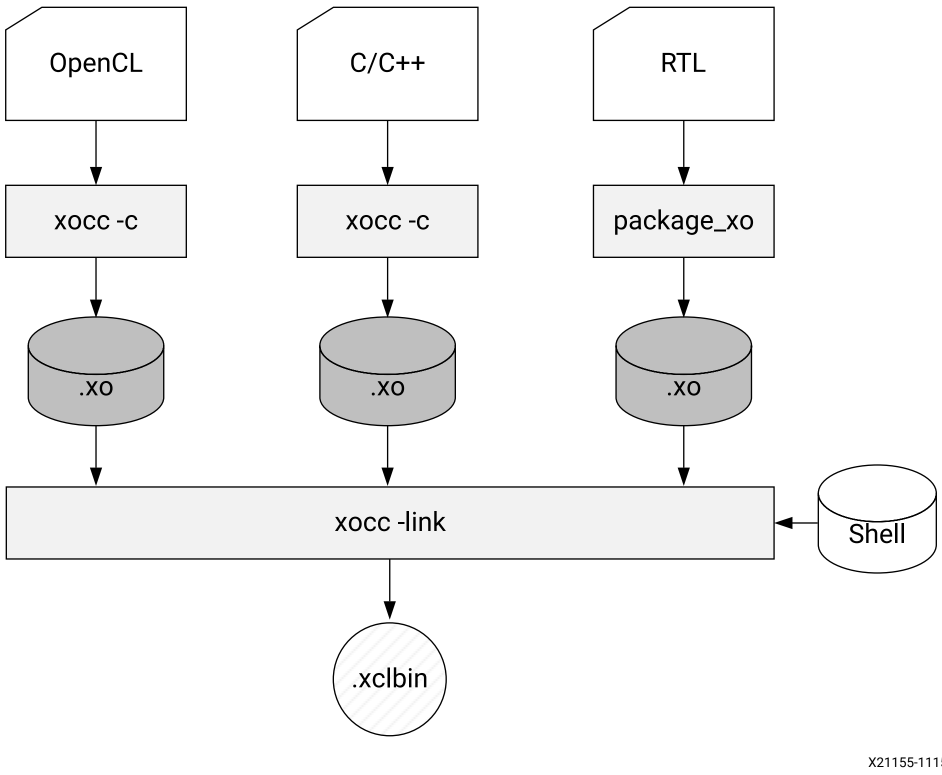
Within this environment, the build process uses a standard compilation and linking process for both the software elements, and the hardware elements of the project. As shown in the following figure, the host application is built through one process using standard GCC compiler, and the FPGA binary is built through a separate process using the Xilinx `xocc` compiler.

Figure 2: **Software/Hardware Build Process**



1. Host application build process using GCC:
   - Each host application source file is compiled to an object file (`.o`).
   - The object files (`.o`) are linked with the Xilinx SDAccel runtime shared library to create the executable (`.exe`).
2. FPGA build process is highlighted in the following figure:
   - Each kernel is independently compiled to a Xilinx object (`.xo`) file.
     - C/C++ and OpenCL C kernels are compiled for implementation on an FPGA using the `xocc` compiler. This step leverages the Vivado® HLS compiler. Pragmas and attributes supported by Vivado HLS can be used in C/C++ and OpenCL C kernel source code to specify the desired kernel micro-architecture and control the result of the compilation process.
     - RTL kernels are compiled using the `package_xo` utility. The RTL kernel wizard in the SDAccel environment can be used to simplify this process.
   - Regardless of kernel designed with RTL, OpenCL, or C/C++, the kernel `.xo` files are linked with the hardware platform (shell) to create the FPGA binary (`.xclbin`). Important architectural aspects are determined during the link step. In particular, this is where connections from kernel ports to global memory banks are established and where the number of instances for each kernel is specified.
     - When the build target is software or hardware emulation, as described below, `xocc` generates simulation models of the device contents.
     - When the build target is the system (actual hardware), `xocc` generates the FPGA binary for the device leveraging the Vivado Design Suite to run synthesis and implementation.

Figure 3: **FPGA Build Process**



Note: The `xocc` compiler automatically uses the Vivado HLS and Vivado Design Suite tools to build the kernels to run on the FPGA platform. It uses these tools with predefined settings which have proven to provide good quality of results. Using the SDAccel environment and the `xocc` compiler does not require knowledge of these tools; however, hardware-savvy developers can fully leverage these tools and use all their available features to implement kernels.

Build Targets

The SDAccel tool build process generates the host application executable (`.exe`) and the FPGA binary (`.xclbin`). The SDAccel build target defines the nature of FPGA binary generated by the build process.

The SDAccel tool provides three different build targets, two emulation targets used for debug and validation purposes, and the default hardware target used to generate the actual FPGA binary:

- **Software Emulation (`sw_emu`):** Both the host application code and the kernel code are compiled to run on the x86 processor. This allows iterative algorithm refinement through fast build-and-run loops. This target is useful for identifying syntax errors, performing source-level debugging of the kernel code running together with application, and verifying the behavior of the system.

- **Hardware Emulation (`hw_emu`):** The kernel code is compiled into a hardware model (RTL) which is run in a dedicated simulator. This build and run loop takes longer but provides a detailed, cycle-accurate, view of kernel activity. This target is useful for testing the functionality of the logic that will go in the FPGA and for getting initial performance estimates.
- **System (`hw`):** The kernel code is compiled into a hardware model (RTL) and is then implemented on the FPGA device, resulting in a binary that will run on the actual FPGA.

SDAccel Debug Flow Overview

This section presents the general debug flow of the SDAccel environment by detailing the general steps of a proven development process. This process allows you to focus rapidly on potential errors in the design. This sets the baseline for developers indicating where to start if an error occurs in their adopted development steps.

The debug flow described here assumes that an SDAccel platform board is installed and the initial setup checks have passed. It is possible to configure the SDAccel environment to work with custom hardware platforms that require a platform shell which defines the foundational components of the board.

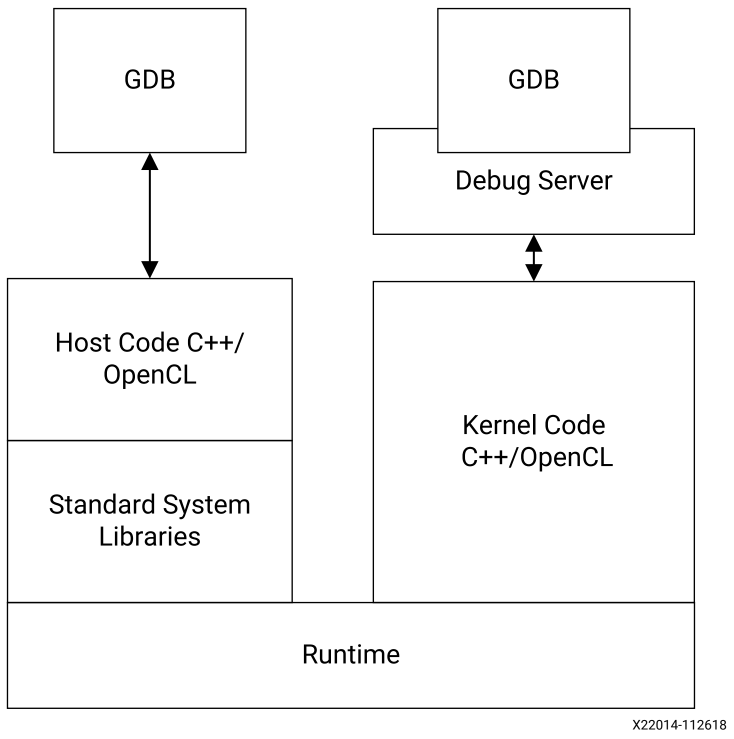
The SDAccel environment provides application-level debug features which allow the host code, the kernel code, and the interactions between them to be efficiently debugged. The recommended application-level debugging flow consists of three levels of debugging: software emulation, hardware emulation, and hardware execution.

This three-tiered approach allows debugging of the host and kernel code and their interactions at different levels of abstraction. Each of the execution models described below is supported through the SDAccel IDE as well as through a batch flow using basic compile time and runtime setup options.

Software Emulation

- **Purpose:** Algorithm verification
- **Execution Model:** During software emulation, all processes are running pure C/C++ models. OpenCL kernel models are transformed to execute concurrently.

Figure 4: **Software Emulation**

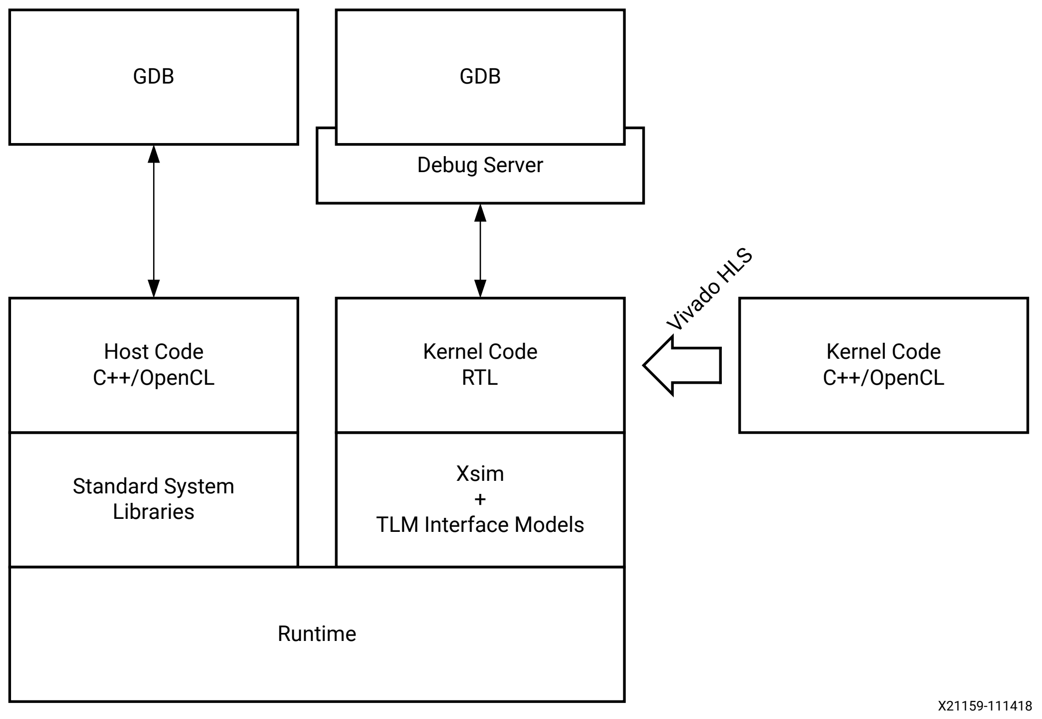


Verify that both the host and kernel code are functionally correct by running software emulation. Because software emulation compiles and executes quickly, spend time here to iterate through the code until the host and kernel code function correctly. Both hardware emulation and hardware execution take more time to compile and execute.

Hardware Emulation

- **Purpose:** RTL debugging, finding protocol violations.
- **Execution Model:** During hardware emulation the host code is executed concurrently with a simulation of the RTL model of the kernel, directly imported, or created through Vivado HLS from the C/C++/OpenCL kernel code.

Figure 5: **Hardware Emulation**



Verify the host code and the kernel hardware implementation is correct by running hardware emulation on a data set. Hardware emulation performs detailed verification using an accurate model of the hardware (RTL) together with the host code C/OpenCL model. The hardware emulation flow invokes the hardware simulator in the SDAccel environment to test the functionality of the logic that is to be executed on the FPGA compute fabric. The interface between the models is represented by a transaction-level model (TLM) to limit impact of interface model on the overall execution time. The execution time for hardware emulation is longer than for software emulation.

TIP: *Xilinx recommends that you use small data sets for debug and validation.*

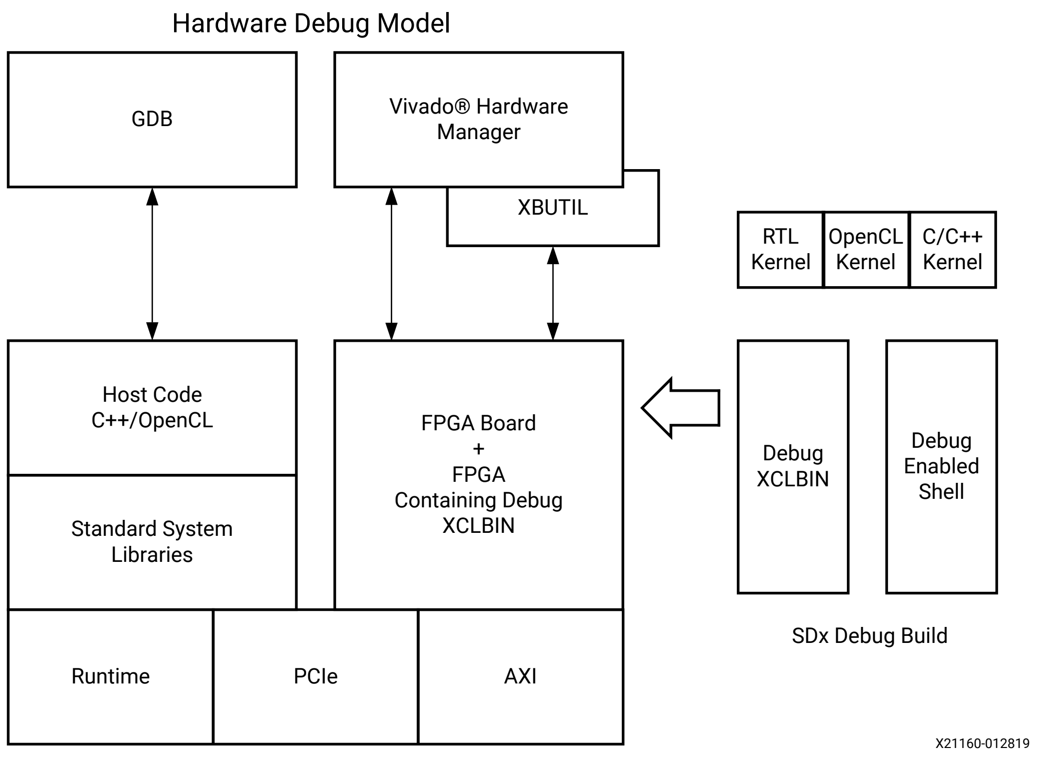
During the hardware emulation stage you can optionally modify the kernel code to improve performance. Iterate in hardware emulation until the functionality is correct and the estimated kernel performance is sufficient. See *SDAccel Environment Profiling and Optimization Guide* (UG1207) for more information.

Hardware Execution

- **Purpose:** Final verification of the complete system, finding protocol violations (hardware hangs), and debugging system performance.

- **Execution Model:** During hardware execution, the actual hardware platform is used to execute the kernels. The difference between this debug configuration and the final compilation of the kernel code is the inclusion of special hardware logic in the platform, such as ILA and VIO debug cores, and AXI performance monitors for debug purposes.

Figure 6: **Hardware Execution**



At this stage, a system image (`xclbin`) is compiled and executed on the actual hardware platform. Refer to the *SDAccel Environment User Guide* (UG1023) for more information on generating the `xclbin` file. At this point, the kernels are confirmed to be executing correctly on the actual FPGA hardware, and your focus can shift from debugging to performance tuning. See the *SDAccel Environment Profiling and Optimization Guide* (UG1207).

Nevertheless, the hardware execution model might not be functional due to protocol issues, or issues with the hardware configuration. Towards that end, the SDAccel environment provides specific hardware debug capabilities which include ChipScope™ debug cores (such as System ILAs), which can be viewed in Vivado hardware manager, with waveform analysis, kernel activity reports, and memory access analysis to localize these critical hardware issues.

IMPORTANT! *Debugging the kernel on the platform hardware requires additional logic to be incorporated into the overall hardware model. This means that if hardware debugging is enabled, there is some impact on resource use of the FPGA, as well as some impact on the kernel performance.*

Chapter 2

SDAccel Debug Features

In this chapter, different features of the SDAccel™ environment supporting debugging efforts are examined. This chapter introduces the debugging tools available to analyze the project and perform debugging. The next chapter illustrates debug techniques using the features described here.

Defensive Programming

The SDAccel environment is capable of creating very efficient implementations. In some cases, however, implementation issues can occur. One such case is if a write request is emitted before there is enough data available in the process to complete the write transaction. This can cause deadlock conditions when multiple concurrent kernels are affected by this issue and the write request of a kernel depends on the input read being completed.

To avoid such situations, a conservative mode is available on the adapter. In principle, it delays the write request until it has all of the data necessary to complete the write. This mode is enabled during compilation by applying the following `--xp` option to the `xocc` compiler:

```
--xp param:compiler.axiDeadLockFree=yes
```

Because enabling this mode can impact performance, you might prefer to use this as a defensive programming technique where this option is inserted during development and testing and then removed during optimization. You might also want to add this option when the accelerator hangs repeatedly.

SDAccel Software Debug

The SDAccel environment supports typical software-like debugging for the host as well as kernel code. This flow is supported during software and hardware emulation and allows the use of break points and the analysis of variables as commonly done during software debugging.

Note: The host code can still be debugged in this mode even when the actual hardware is executed.

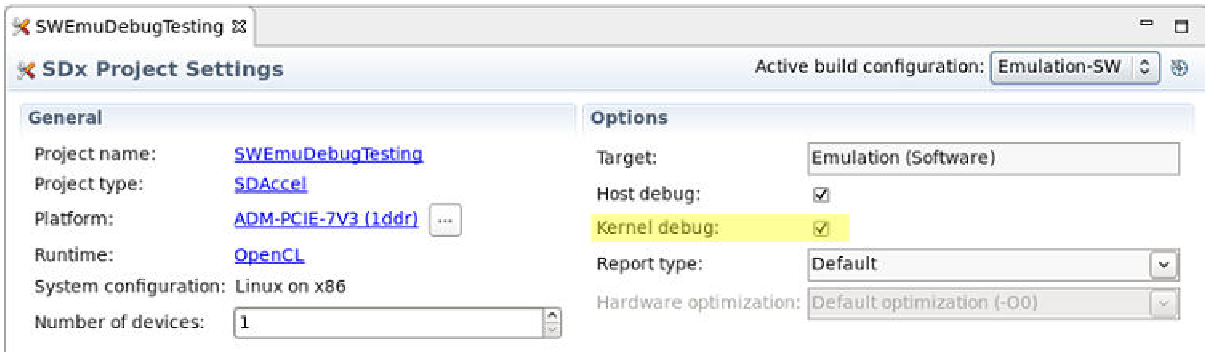
IDE Debug Flow

The SDAccel integrated design environment (IDE) flow provides easy access to the debug capabilities. Setting up an executable for debugging requires many steps when performed manually. These steps are handled by the IDE when you use the IDE debug flow.

***Note*:** The SDAccel debug flow relies on shell scripts during debugging. This requires that the setup files such as `.bashrc` or `.cshrc` do not interfere with the SDAccel setup, such as the `LD_LIBRARY_PATH`.

Preparing the executable for debugging requires that you change the build configurations to enable the application of debug flags. You can set these options through the Project Settings in the SDx™ environment. There are two check boxes provided in the Options section for the Active build configuration. One enables host debug builds while the other enables debugging of the kernels. The checkboxes are named Host debug and Kernel debug respectively.

Figure 7: **Software Project Settings Options**



Active build configuration: Emulation-SW

Project name:	SWEmuDebugTesting
Project type:	SDAccel
Platform:	ADM-PCIE-7V3 (1ddr)
Runtime:	OpenCL
System configuration:	Linux on x86
Number of devices:	1

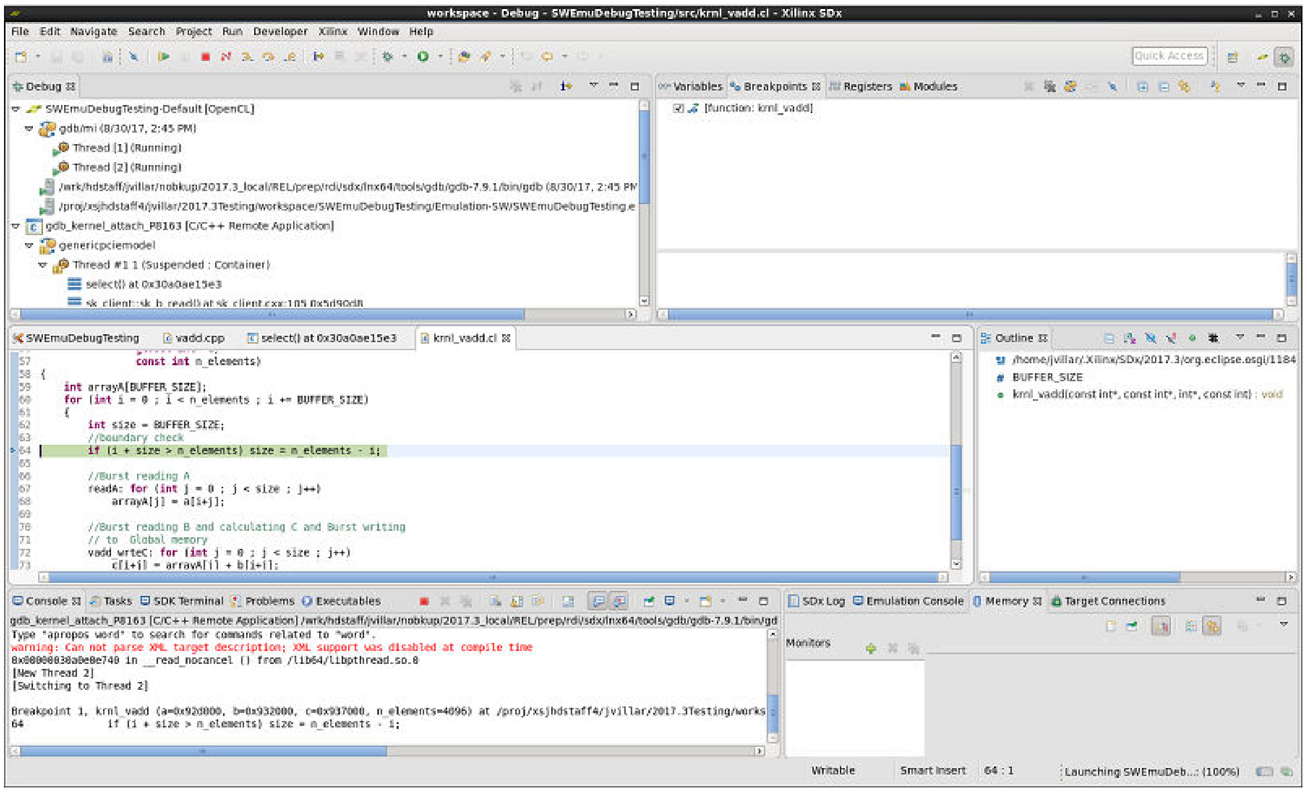
Target:	Emulation (Software)
Host debug:	✓
Kernel debug:	✓
Report type:	Default
Hardware optimization:	Default optimization (-O0)

A more intuitive way to set these build options is through the context menu settings. To do this, right-click on the build configuration in the Assistant view and select **Settings**. Alternatively, you can double-click on the build configuration. The same two checkboxes are presented. While you can enable host debug on all targets, kernel debug is only supported for software emulation and hardware emulation build targets. This completes the setup; cleaning the build directory and rebuilding the application ensure that the project is ready for running in the GDB debug environment.

Running a GDB session from the IDE takes care of all the setup required. It automatically manages the environment setup for hardware or software emulation. It configures the SDAccel runtime to ensure debug support by the runtime environment, and manages the different consoles required for the execution of the kernel model, the host model, and the debug server.

As a result, when initiating the debug session, the SDAccel environment asks to switch into the debug perspective, which presents several windows to manage the different debug consoles and source code windows.

Figure 8: **GDB Console**



```
SWEmuDebugTesting-Default [OpenCL]
    gdb/mi (8/30/17, 2:45 PM)
        Thread [1] (Running)
        Thread [2] (Running)
    /wrk/hdstaff/jvillar/nobkup/2017.3_local/REL/prep/rdi/sdx/lnx64/tools/gdb/gdb-7.9.1/bin/gdb (8/30/17, 2:45 PM
    /proj/xsjhdstaff4/jvillar/2017.3Testing/workspace/SWEmuDebugTesting/Emulation-SW/SWEmuDebugTesting.e
gdb_kernel_attach_P8163 [C/C++ Remote Application]
    genericpciemodel
        Thread #1 1 (Suspended : Container)
            select() at 0x30a0ae15e3
            sk_client::sk_b_read() at sk_client.cxx:105 0x5d90d8
```

```
[function: krnl_vadd]
```

```
                   const int n_elements)
{
    int arrayA[BUFFER_SIZE];
    for (int i = 0 ; i < n_elements ; i += BUFFER_SIZE)
    {
        int size = BUFFER_SIZE;
        //boundary check
        if (i + size > n_elements) size = n_elements - i;

        //Burst reading A
        readA: for (int j = 0 ; j < size ; j++)
            arrayA[j] = a[i+j];

        //Burst reading B and calculating C and Burst writing
        // to  Global memory
        vadd_wrteC: for (int j = 0 ; j < size ; j++)
            c[i+j] = arrayA[j] + b[i+j];
```

```
/home/jvillar/.Xilinx/SDx/2017.3/org.eclipse.osgi/1184
BUFFER_SIZE
krnl_vadd(const int*, const int*, int*, const int) : void
```

```
gdb_kernel_attach_P8163 [C/C++ Remote Application] /wrk/hdstaff/jvillar/nobkup/2017.3_local/REL/prep/rdi/sdx/lnx64/tools/gdb/gdb-7.9.1/bin/gd
Type "apropos word" to search for commands related to "word".
warning: Can not parse XML target description; XML support was disabled at compile time
0x00000038a0e0e740 in __read_nocancel () from /lib64/libpthread.so.0
[New Thread 2]
[Switching to Thread 2]

Breakpoint 1, krnl_vadd (a=0x92d000, b=0x932000, c=0x937000, n_elements=4096) at /proj/xsjhdstaff4/jvillar/2017.3Testing/works
64              if (i + size > n_elements) size = n_elements - i;
```

After starting the application, by default the application is stopped right at the beginning of the `main` function body in the host code. As with any GDB graphical front end, you can now set breakpoints and inspect variables in the host code. The SDAccel environment enables the same capabilities for the accelerated kernel implementation in a transparent way.

Note: In hardware emulation, because the C/C++/OpenCL™ kernel code is translated for efficient implementation, breakpoints cannot be placed on all statements. Mostly, untouched loops and functions are available for breakpoints, and similarly only preserved variables can be accessed.

Related Information

Xilinx OpenCL Runtime GDB Extensions

Command Line Debug Flow

The command line debug flow in the SDAccel environment provides tools to debug the host and kernel application running in all modes: software emulation, hardware emulation, or hardware execution.

Note: The host code can be debugged using this feature in the hardware execution mode only.

There are four steps to debugging in the SDAccel environment using the command line flow:

1. General environment setup.
2. Prepare the host code for debug.
3. Prepare the kernel code for debug.
4. Launch GDB Standalone to debug.

IMPORTANT! *The SDAccel environment supports host program debugging in all modes, but kernel debugging is only supported in the emulation flows with `gdb`. In addition, more hardware-centric debugging support, such as waveform analysis, is provided for the kernels.*

General Environment Setup

Running software or hardware emulation requires first the tool setup followed by the selection of the emulation mode.

1. To set up the tool environment and run the SDx tool, source the file below so that SDx command settings are in the `PATH`:
    - C Shell: `source <SDX_INSTALL_DIR>/settings64.csh`
    - Bash: `source <SDX_INSTALL_DIR>/settings64.sh`
2. To set up the runtime environment reponsible for the interaction between the software and hardware implementation, source the file below:
    - C Shell: `source /opt/xilinx/xrt/setup.csh`
    - Bash: `source /opt/xilinx/xrt/setup.sh`

Table 1: **Select Emulation Mode**

Environment Variable	Value
`XCL_EMULATION_MODE`	`sw_emu` or `hw_emu` These environment settings are used by the runtime library to correctly execute the desired emulation. This is required in addition to building the executable for the specific emulation flow.

Preparing the Host Code

The host program needs to be compiled with debugging information generated in the executable by adding the `-g` option to the `xcpp` command line option, as follows:

```
xcpp -g ...
```

TIP: *Because `xcpp` is simply a wrapper around the system compiler (`gcc`), the `-g` option enables the compiler to generate debug information.*

Preparing the Kernel

Kernel code can be debugged together with the host program in either software emulation or hardware emulation. Debugging information needs to be generated first in the binary container by passing the `-g` option to the `xocc` command line executable:

```
xocc -g -t [sw_emu | hw_emu | hw] ...
```

The `-t` (or `-target`) option is used to specify the compilation target as either software emulation (`sw_emu`), hardware emulation (`hw_emu`), or hardware execution (`hw`).

In the software emulation flow, additional runtime checks can be performed for OpenCL based kernels. The runtime checks include:

- Checking out-of-bound access made by kernel interface buffers (option: `address`)
- Checking uninitialized memory access initiated by kernel local to kernel (option: `memory`)

The options are enabled through the `--xp` option and the `param:compiler.fsanitize` directive, and need to be enabled during the link stage (`-l`) as shown in the following examples:

```
xocc -l -t sw_emu --xp param:compiler.fsanitize=address -o bin_kernel.xclbin
xocc -l -t sw_emu --xp param:compiler.fsanitize=memory -o bin_kernel.xclbin
xocc -l -t sw_emu --xp param:compiler.fsanitize=address,memory -o
bin_kernel.xclbin
```

When applied, the emulation run produces a debug log with emulation diagnostic messages such as `<project_dir>/Emulation-SW/<proj_name>-Default>/emulation_debug.log`.

Launching GDB Host Code Debug

You can launch GDB standalone to debug the host program if the code is built with debug information (built with the `-g` flag). This flow should also work while using a graphical front-end for GDB, such as the Data Display Debugger (DDD) available from GNU. The following steps are the instructions for launching GDB.

1. To set up the environment to run the SDx tool, source the file below so that SDx command settings are in the `PATH`:
    - C Shell: `source <SDX_INSTALL_DIR>/settings64.csh`
    - Bash: `source <SDX_INSTALL_DIR>/settings64.sh`
2. To set up the runtime environment responsible for the interaction between the software and hardware implementation, source the file below:
    - C Shell: `source /opt/xilinx/xrt/setup.csh`
    - Bash: `source /opt/xilinx/xrt/setup.sh`
3. Ensure that the environment variable `XCL_EMULATION_MODE` is set to the correct mode.

4. The application debug feature must be enabled at runtime using an attribute in the `sdaccel.ini` file. Create an `sdaccel.ini` file in the same directory as your host executable, and include the following lines:

   ```
   [Debug]
   app_debug=true
   ```

   This informs the runtime library that the kernel is debug enabled.

5. Start `gdb` through the Xilinx® wrapper:

   ```
   xgdb --args host.exe test.xclbin
   ```

   The `xgdb` wrapper performs the following setup steps under the hood:

   - Launches GDB on the host program:

     ```
     gdb --args host.exe test.xclbin
     ```

   - Sets up the environment variables `PYTHONHOME` and `PYTHONPATH` to Python installation. Currently, the `gdb` in the SDx environment expects Python 2.6 or Python 2.7. For example, if the Python available on the machine is Python 2.6, set the environment as shown (Bash shell shown):

     ```
     export PYTHONHOME=/usr
     export PYTHONPATH=/usr/lib64/python2.6/:/usr/lib64/python2.6/lib-
     dynload/
     ```

   - Sources the Python script in the GDB console to enable the Xilinx GDB extensions:

     ```
     gdb> source ${XILINX_SDX}/scripts/appdebug.py
     ```

Launching Host and Kernel Debug

In software emulation, to better mimic the hardware being emulated, kernels are spawned off as separate processes. If you are using GDB to debug the host code, breakpoints set on kernel lines are not hit because the kernel code is not run within that process. To support the concurrent debugging of the host code and the kernel code, the SDAccel environment provides a mechanism to attach to spawned kernels through the use of `sdx_server`.

1. You must start three different terminals in the command line flow. In the first terminal, start the `sdx_server` using the following command:

   ```
   ${XILINX_VIVADO}/bin/sdx_server --sdx-url
   ```

2. In a second terminal, run the host code in `xgdb` as described in Launching GDB Host Code Debug.

At this point, the first terminal running the `sdx_server` should provide a `GDB listener port NUM` on standard out. Keep track of the number returned by the `sdx_server` as the GDB listener port is used by GDB to debug the kernel process. When the GDB listener port is printed, the spawned kernel process has attached to the `sdx_server` and is waiting for commands from you. To control this process, you must start a new instance of GDB and connect to the `sdx_server`.

IMPORTANT! *If the `sdx_server` is running, then all spawned processes compiled for debug connect and wait for control from you. If no GDB ever attaches or provides commands, the kernel code appears to hang.*

3. In a third terminal, run the `xgdb` command, and at the GDB prompt, run the following commands:

   - For software emulation:

     ```
     "file ${XILINX_SDX}/data/emulation/unified/cpu_em/generic_pcie/model/
     genericpciemodel"
     ```

   - For hardware emulation:

     1. Locate the `sdx_server` temporary directory:`/tmp/sdx/$uid`.
     2. Find the `sdx_server` process id (PID) containing the DWARF file of this debug session.
     3. At the `gdb` command line, run: `file /tmp/sdx/$uid/$pid/NUM.DWARF`.

   - In either case, connect to the kernel process:

     ```
     target remote :NUM
     ```

     Where `NUM` is the number returned by the `sdx_server` as the GDB listener port.

TIP: *When debugging software/hardware emulation kernels in the SDAccel IDE, these steps are handled automatically and the kernel process is automatically attached, providing multiple contexts to debug both the host code and kernel code simultaneously.*

After these commands are executed, you can set breakpoints on your kernels as needed, run the `continue` command, and debug your kernel code. When the all kernel invocations have finished, the host code continues, and the `sdx_server` connection drops.

For both software and hardware emulation flows, there are restrictions with respect to the accelerated kernel code debug interactions. Because this code is preprocessed in the software emulation flow, and then translated in the hardware emulation flow into a hardware description language (HDL) and simulated during debugging, it is not always possible to set breakpoints at all locations. Especially with hardware emulation, only a limited number of breakpoints such as on preserved loops and functions are supported. Nevertheless, this mode is useful for debugging the kernel/host interface.

Utilities for Hardware Debugging

In some cases, the normal SDAccel IDE and command line debug features are limited in their ability to isolate an issue. This is especially true when the software or hardware appears not to make any progress (hangs). These kinds of system issues are best analyzed with the help of the utilities mentioned in this section.

Using the Linux dmesg Utility

Well-designed Linux kernels and modules report issues through the kernel ring buffer. This is also true for SDAccel environment modules that allow you to debug the interaction with the accelerator board on the lowest Linux level.

***Note*:** This utility intended for use in hardware debug only.

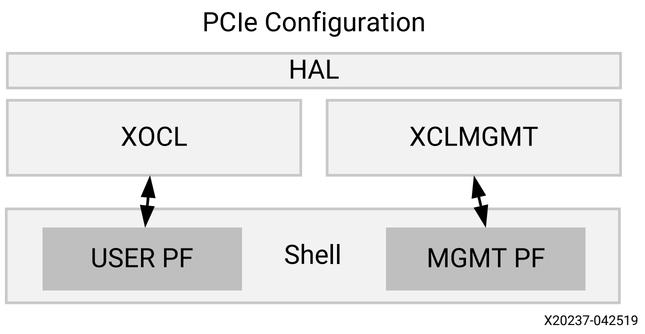
TIP: *In most cases, it is sufficient to work with the less verbose `xbutil` feature to localize a problem. Refer to the SDx Command and Utility Reference Guide (UG1279) for more information on the `xbutil` command.*

The `dmesg` utility is a Linux tool that lets you read the kernel ring buffer. The kernel ring buffer holds kernel information messages in a circular buffer. A circular buffer of fixed size is used to limit the resource requirements by overwriting the oldest entry with the next incoming message.

In the SDAccel tool, the `xocl` module and `xclmgmt` driver modules write informational messages to the ring buffer. Thus, for an application hang or crash, or for that matter any unexpected behavior (like being unable to program the bitstream, and so on), the `dmesg` tool should be used to check the ring buffer.

The following image shows the layers of the software platform associated with the SDAccel board platform.

Figure 9: **Software Platform Layers**



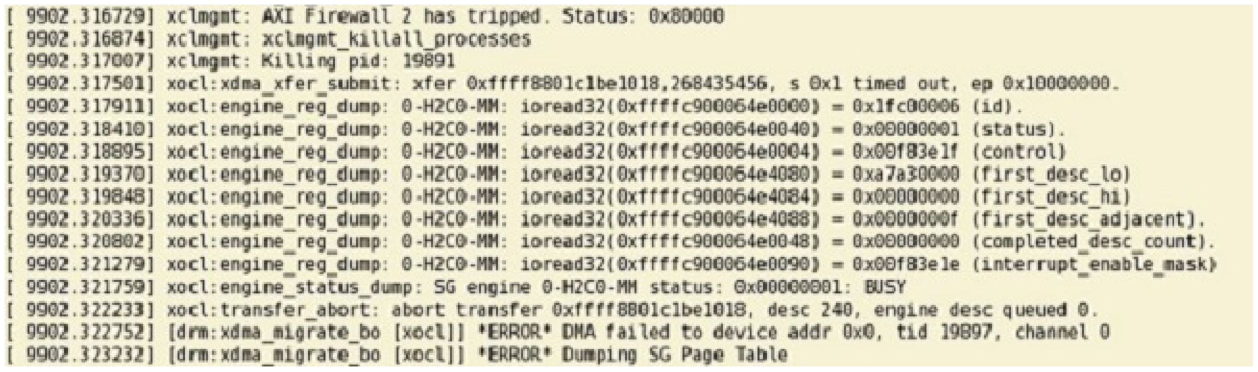
To review messages from the Linux tool, you should first clear the ring buffer:

```
sudo dmesg -c
```

This flushes all messages from the ring buffer and make it easier to spot messages from the `xocl` and `xclmgmt`. After that, start your application and run `dmesg` in another terminal.

```
sudo dmesg
```

The `dmesg` utility prints a record such as the following module reports:



```
[ 9902.316729] xclmgmt: AXI Firewall 2 has tripped. Status: 0x80000
[ 9902.316874] xclmgmt: xclmgmt_killall_processes
[ 9902.317007] xclmgmt: Killing pid: 19891
[ 9902.317501] xocl:xdma_xfer_submit: xfer 0xffff8801c1be1018,268435456, s 0x1 timed out, ep 0x10000000.
[ 9902.317911] xocl:engine_reg_dump: 0-H2C0-MM: ioread32(0xffffc900064e0000) = 0x1fc00006 (id).
[ 9902.318410] xocl:engine_reg_dump: 0-H2C0-MM: ioread32(0xffffc900064e0040) = 0x00000001 (status).
[ 9902.318895] xocl:engine_reg_dump: 0-H2C0-MM: ioread32(0xffffc900064e0004) = 0x00f83e1f (control)
[ 9902.319370] xocl:engine_reg_dump: 0-H2C0-MM: ioread32(0xffffc900064e4080) = 0xa7a30000 (first_desc_lo)
[ 9902.319848] xocl:engine_reg_dump: 0-H2C0-MM: ioread32(0xffffc900064e4084) = 0x00000000 (first_desc_hi)
[ 9902.320336] xocl:engine_reg_dump: 0-H2C0-MM: ioread32(0xffffc900064e4088) = 0x0000000f (first_desc_adjacent).
[ 9902.320802] xocl:engine_reg_dump: 0-H2C0-MM: ioread32(0xffffc900064e0048) = 0x00000000 (completed_desc_count).
[ 9902.321279] xocl:engine_reg_dump: 0-H2C0-MM: ioread32(0xffffc900064e0090) = 0x00f83e1e (interrupt_enable_mask)
[ 9902.321759] xocl:engine_status_dump: SG engine 0-H2C0-MM status: 0x00000001: BUSY
[ 9902.322233] xocl:transfer_abort: abort transfer 0xffff8801c1be1018, desc 240, engine desc queued 0.
[ 9902.322752] [drm:xdma_migrate_bo [xocl]] *ERROR* DMA failed to device addr 0x0, tid 19897, channel 0
[ 9902.323232] [drm:xdma_migrate_bo [xocl]] *ERROR* Dumping SG Page Table
```

In the example shown above, the AXI Firewall 2 has tripped, which is better examined using the `xbutil` utility.

Using the Xilinx xbutil Utility

The Xilinx board utility (`xbutil`) is a powerful standalone command line utility that can be used to debug lower level hardware/software interaction issues. A full description of this utility can be found in the *SDx Command and Utility Reference Guide* (UG1279).

With respect to debugging, the following `xbutil` options are of special interest:

- `query`: Provides an overall status of an SDAccel environment platform.
- `program`: Downloads a binary (`xclbin`) to the programmable region of the Xilinx device.
- `status`: Extracts the status of the SDx environment Performance Monitors (`spm`) and the Lightweight AXI Protocol Checkers (`lapc`).

Hardware Debugging Using ChipScope

After the final system image (`xclbin`) is generated and executed on the SDAccel environment platform, the entire system including the host application running on the CPU, and the accelerated kernels on the Xilinx FPGA, can be confirmed to be executing correctly on the actual hardware. At this stage you can validate the functioning of the host code and kernel in the target hardware, and debug any issues found. Some of the conditions that can be looked for or analyzed are listed as follows:

- System hangs that could be due to protocol violations:
  - These violations can take down the entire system.
  - These violations can cause the kernel to get invalid data or to hang.
  - It is hard to determine where or when these violations originated.
  - To debug this condition, you should use an ILA triggered off of the AXI protocol checker, which needs to be configured on the SDAccel platform in use.
- Problems inside the RTL kernel:
  - These problems are sometimes caused by the implementation: timing issues, race condition, and bad design constraint.
  - Functional bugs that hardware emulation did not show.
- Performance problems:
  - For example, the frames per second processing is not what you expect.
  - You can examine data beats and pipelining.
  - Using an ILA with trigger sequencer, you can examine the burst size, pipelining, and data width to locate the bottleneck.

Checking the FPGA Board for Hardware Debug Support

Supporting hardware debugging requires the platform to support several IP components, most notably the Debug Bridge. Talk to your platform designer to determine if these components are included in the platform shell. If a Xilinx platform is used, debug availability can be verified using the `platforminfo` utility to query the platform. Debug capabilities are listed under the `chipscope_debug` objects.

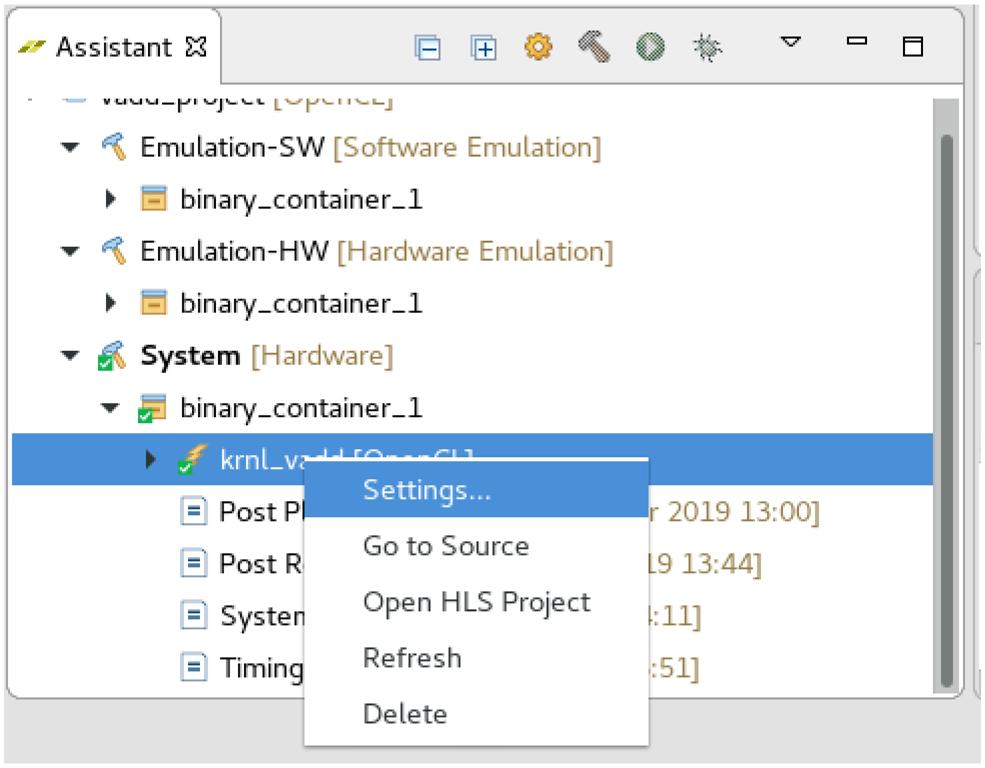
For example, to query the a platform for hardware debug support, the following `platforminfo` command can be used. A response can be seen showing that the platform contains a user and management debug network, and also supports debugging a MicroBlaze™ processor.

```
$ platforminfo --json="hardwarePlatform.extensions.chipscope_debug" --
platform xilinx_u200_xdma_201830_1
{
    "debug_networks": {
        "user": {
            "name": "User Debug Network",
            "pcie_pf": "1",
            "bar_number": "0",
            "axi_baseaddr": "0x000C0000",
            "supports_jtag_fallback": "false",
            "supports_microblaze_debug": "true",
            "is_user_visible": "true"
        },
        "mgmt": {
            "name": "Management Debug Network",
            "pcie_pf": "0",
            "bar_number": "0",
            "axi_baseaddr": "0x001C0000",
            "supports_jtag_fallback": "true",
            "supports_microblaze_debug": "true",
            "is_user_visible": "false"
        }
    }
}
```

Enabling ChipScope from the SDx IDE

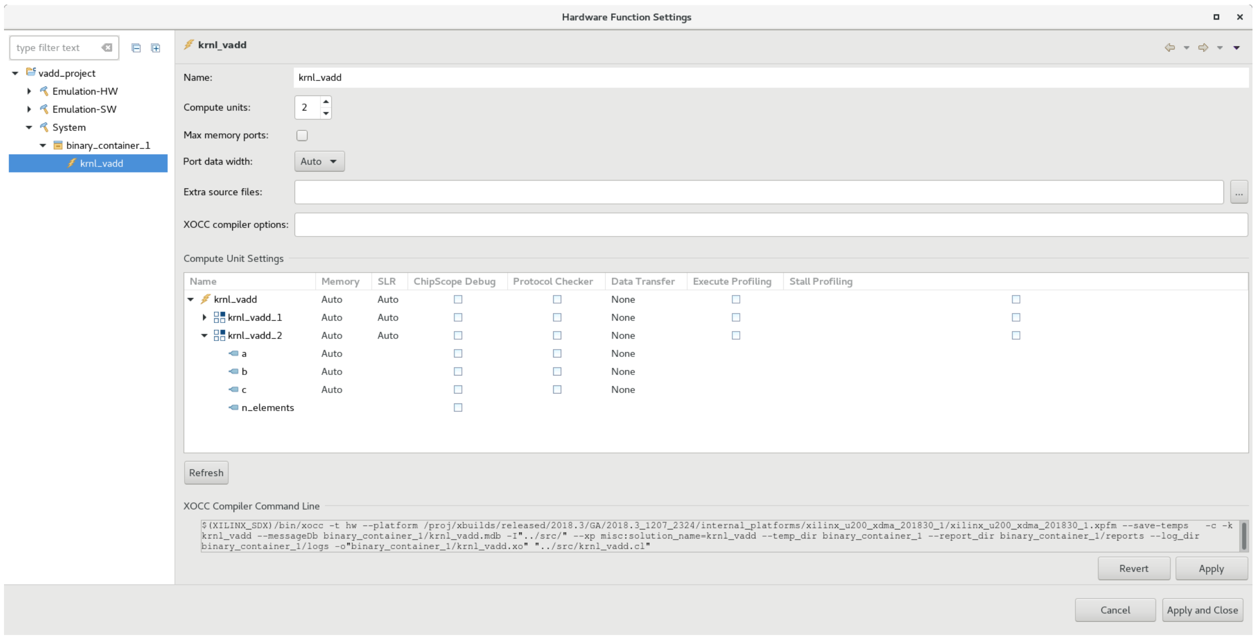
The SDx IDE provides options to enable the ChipScope™ debug feature on all the interface ports of the compute units in the design. When enabling this option on a compute unit, the SDAccel environment compiler adds a System ILA debug core to monitor the interface ports of the compute unit. This ensures that you can debug the interface signals on the SDAccel environment platform hardware while the kernel is running. You can access this through the Settings command by right-clicking on a kernel in the system build configuration in the Assistant window as shown below.

Figure 10: **SDx Assistant View**



This brings up the Hardware Function Settings dialog box as shown in the following figure. You can use the Debug and Profiling Settings table in this dialog box to enable the ChipScope Debug checkbox for specific compute units of the kernel, which enables the monitoring of all the interfaces/ports on the compute unit.

Figure 11: **SDx Hardware Function Settings**



Name: krnl_vadd

Compute units: 2

Max memory ports: ☐

Port data width: Auto

Extra source files:

XOCC compiler options:

Compute Unit Settings

Name	Memory	SLR	ChipScope Debug	Protocol Checker	Data Transfer	Execute Profiling	Stall Profiling
krnl_vadd	Auto	Auto	☐	☐	None	☐	☐
krnl_vadd_1	Auto	Auto	☐	☐	None	☐	☐
krnl_vadd_2	Auto	Auto	☐	☐	None	☐	☐
a	Auto		☐	☐	None		
b	Auto		☐	☐	None		
c	Auto		☐	☐	None		
n_elements			☐				

XOCC Compiler Command Line

```
$(XILINX_SDX)/bin/xocc -t hw --platform /proj/xbuilds/released/2018.3/GA/2018.3_1207_2324/internal_platforms/xilinx_u200_xdma_201830_1/xilinx_u200_xdma_201830_1.xpfm --save-temps  -c -k
krnl_vadd --messageDb binary_container_1/krnl_vadd.mdb -I"../src/" --xp misc:solution_name=krnl_vadd --temp_dir binary_container_1 --report_dir binary_container_1/reports --log_dir
binary_container_1/logs -o"binary_container_1/krnl_vadd.xo" "../src/krnl_vadd.cl"
```

TIP: *Enabling the* ***ChipScope Debug*** *option on larger designs with multiple kernels and/or compute units can result in overuse of the FPGA device resources. Xilinx recommends using the* `xocc --dk list_ports` *option on the command line to determine the number and type of interfaces on the compute units. If you know which ports need to be monitored for debug as the design runs in hardware, the recommended methodology is to use the* `--dk` *option documented in the following topic.*

Command Line Flow

The full SDAccel kernel code compilation and linking command line flow can be found in the *SDAccel Environment User Guide* (UG1023), Chapter 8. The following section covers the `xocc` linker options that can be used to list the available kernel ports as well as enable the System Integrated Logic Analyzer core on the selected ports. You should only use this flow if you are already familiar with the steps to build an SDAccel kernel at the command line.

The System Integrated Logic Analyzer debug core provides transaction-level visibility into an accelerated kernel or function running on hardware. AXI traffic of interest can also be captured and viewed using the System ILA core. The ILA core can be instantiated in the overall hardware of an existing RTL IP design to enable debugging features within that design, or it can be inserted automatically by the compiler. The `xocc` compiler provides the `--dk` option to attach System ILA cores at the interfaces to the kernels for debugging and performance monitoring purposes.

The --dk option to enable ILA IP core insertion has the following syntax:

```
 --dk <[chipscope|list_ports]<:compute_unit_name><:interface_name>>
```

In general, the <interface_name> is optional. If not specified, all ports are expected to be analyzed. The chipscope option requires the explicit name of the compute unit to be provided for the <compute_unit_name> and <interface_name>. The list_ports option generates a list of valid compute units and port combinations in the current design and must be used after the kernel has been compiled.

Before using the --dk option, the kernel must be compiled into an .xo file. For a complete description of each xocc command line option as well as the complete SDAccel command line build flow, refer to the *SDAccel Environment User Guide* (UG1023).

The first command compiles the kernel source files into an .xo file:

```
xocc -c -k <kernel_name> --platform <platform> -o <kernel_xo_file>.xo
<kernel_source_files>
```

After the kernel has been compiled into an .xo file, --dk list_ports can be added to the command line options used during the xocc linking process. This causes the xocc compiler to print the list of valid compute units and port combinations. See the following example:

```
xocc -l --platform <platform> --nk
<kernel_name>:<compute_units>:<kernel_nameN> --dk list_ports
<kernel_xo_file>.xo
```

Finally, ChipScope debug can be enabled on the desired ports by replacing list_ports with the appropriate --dk chipscope command:

```
xocc -l --platform <platform> --nk
<kernel_name>:<compute_units>:<kernel_nameN> --dk
chipscope:<compute_unit_name>:<interface_name> <kernel_xo_file>.xo
```

***Note*:**

Multiple --dk option switches can be specified in a single command line to additively increase interface monitoring capability.

Refer to the *SDx Command and Utility Reference Guide* (UG1279) for more information on any xocc option. When the design is built, you can debug the design using the Vivado® hardware manager as described in *Vivado Design Suite User Guide: Programming and Debugging* (UG908).

Chapter 3

Debug Techniques

This section closely examines different styles of debugging techniques. It classifies the different approaches into software-based debugging techniques and hardware-oriented techniques. In the software-based approaches, you are not required to fully understand the ultimate mapping of the kernel code onto the FPGA. However, this concept can only be extended to a certain amount of detail, at which point the more detailed hardware-based analysis is required.

The section is structured along the different debug stages in the SDAccel™ environment. It starts with functional verification during software emulation (a purely software-based approach). Next is hardware emulation, where the kernel code is converted into actual hardware representation providing more details of the final implementation. Hardware debugging as well as software debugging concepts can be applied during debugging in the hardware emulation stage. The last stage is system verification, where the actual hardware is executed. In this stage, software debugging concepts can only be applied to the host while the kernel must deploy hardware debugging concepts.

Functional Verification (Software Emulation)

Functional verification is the process during which the software representing the system is verified towards the ultimate implementation goal by ensuring that the software behaves as intended on the given data. This is a very common task during software development and many different concepts are available.

If your software does not perform as intended, you can use the debugger to identify the root cause of the issue, or if necessary, dump datapoints during software execution. This section introduces these concepts applied to an SDx™ environment project.

Using printf() to Debug Kernels

The simplest approach to debugging algorithms is to verify key data values throughout the execution of the program. For application developers, printing checkpoint values in the code is a tried and trusted way of identifying problems within the execution of a program. Because part of the algorithm is now running on an FPGA, even this debugging technique requires additional support.

The SDAccel development environment supports the OpenCL™ `printf()` built-in function within the kernels in all development flows: software emulation, hardware emulation, and running the kernel in actual hardware. The following is an example of using `printf()` in the kernel, and the output when the kernel is executed with `global` size of 8:

```
__kernel __attribute__ ((reqd_work_group_size(1, 1, 1)))
void hello_world(__global int *a)
{
    int idx = get_global_id(0);

     printf("Hello world from work item %d\n", idx);
     a[idx] = idx;
}
```

The output is as follows:

```
Hello world from work item 0
Hello world from work item 1
Hello world from work item 2
Hello world from work item 3
Hello world from work item 4
Hello world from work item 5
Hello world from work item 6
Hello world from work item 7
```

IMPORTANT! *`printf()` messages are buffered in the global memory and unloaded when kernel execution is completed. If `printf()` is used in multiple kernels, the order of the messages from each kernel display on the host terminal is not certain. Please note, especially when running in hardware emulation and hardware, the hardware buffer size might limit `printf` output capturing.*

***Note*:** This feature is only supported for OpenCL kernels in all development flows.

For C/C++ kernel models `printf()` is only supported during software emulation and should be excluded from the Vivado® HLS synthesis step. In this case, any `printf()` statement should be surrounded by the following compiler macros:

```
#ifndef __SYNTHESIS__
    printf("text");
#endif
```

GDB-Based Debugging

This section shows how host and kernel debugging can be performed with the help of GDB. Because this flow should be familiar to software developers, this section focuses on the extensions of host code debugging capabilities specifically for FPGAs, and the current status of kernel-based hardware emulation support.

Host Code Debugging

Except for the method of launching the debugging environment described in the previous chapter, there is no difference between the SDAccel host code debugging and the commonly used GDB application debugging flow and features.

After `gdb` is launched, you can step through the host code in GDB and examine the C/C++/ OpenCL objects to verify that their contents are as expected at any point in the code.

However, as stated in the introduction especially in the case of hardware emulation, it is common to look for issues regarding protocol synchronization between the host and the kernel. The SDAccel environment provides special GDB extensions to examine the content of the OpenCL runtime environment from the application host. These commands are described in more detail in the next section.

Xilinx OpenCL Runtime GDB Extensions

The Xilinx OpenCL runtime Debug Environment introduces new GDB commands that give visibility from the host application into the OpenCL runtime library.

Note: If you run GDB outside of the SDAccel environment, these commands need to be enabled as described in Launching GDB Host Code Debug.

There are two kinds of commands which can be called from the `gdb` command line:

- Commands that give visibility into the OpenCL runtime data structures (`cl_command_queue`, `cl_event`, and `cl_mem`). The arguments to `xprint queue` and `xprint mem` are optional. The application debug environment keeps track of all the OpenCL objects and automatically prints all valid queues and `cl_mem` objects if the argument is not specified. In addition, the commands do a proper validation of supplied command `queue`, `event`, and `cl_mem` arguments.

```
xprint queue [<cl_command_queue>]
xprint event <cl_event>
xprint mem [<cl_mem>]
xprint kernel
xprint all
```

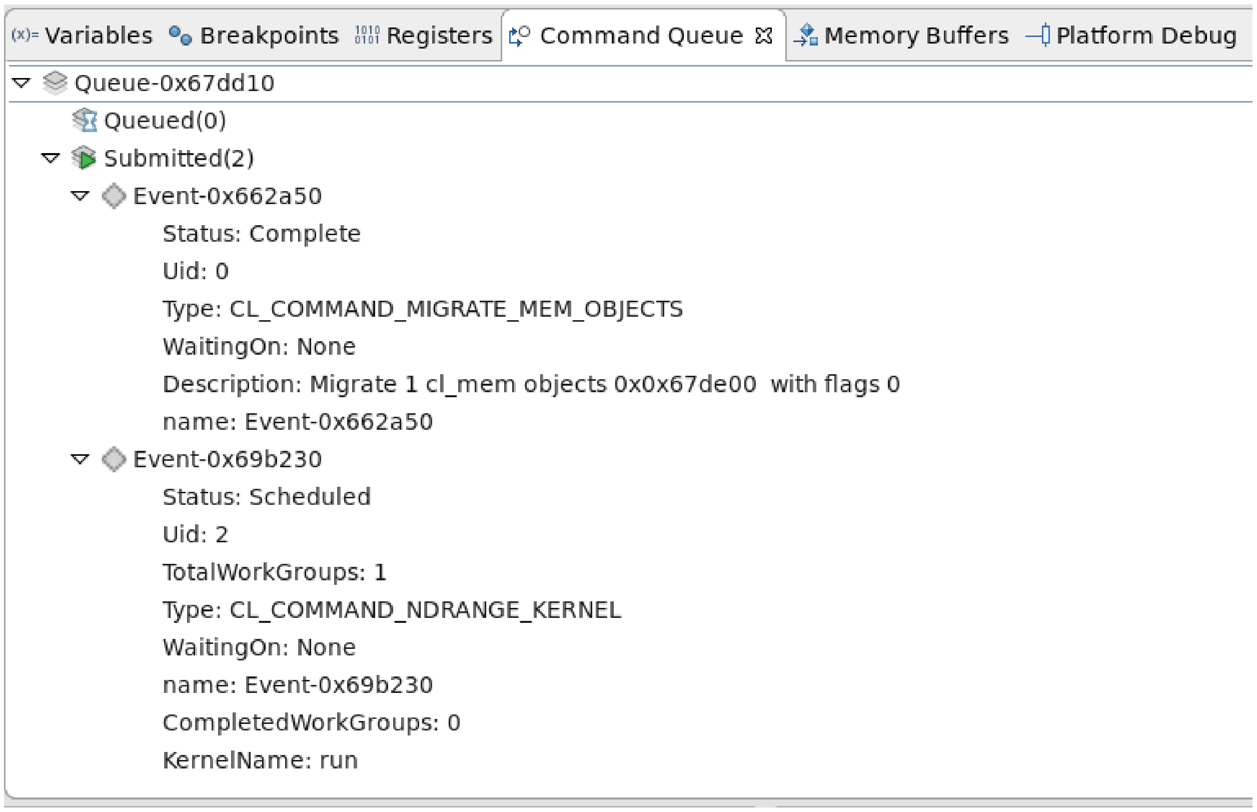
- Commands that give visibility into the IP on the SDAccel platform. This functionality is only available in the system flow (hardware execution) and not in any of the emulation flows.

```
xstatus all
xstatus --<ipname>
```

You can get help information about the commands by using `help <command>`.

A typical example for using these commands is if you are seeing the host application hang. In this case, the host application is likely to be waiting for the command queue to finish or waiting on an event list. Printing the command queue using the `xprint` command can tell you what events are unfinished, letting you analyze the dependencies between the events.

The output of both of these commands is automatically tracked when debugging with the SDAccel IDE. In this case three tabs are provided next to the common tabs for Variables, Breakpoints, and Registers in the left upper corner of the debug perspective. These are labeled Command Queue, Memory Buffers, and Platform Debug, showing the output of `xprint queue`, `xprint mem`, and `xstatus` respectively.



```
Queue-0x67dd10
  Queued(0)
  Submitted(2)
    Event-0x662a50
      Status: Complete
      Uid: 0
      Type: CL_COMMAND_MIGRATE_MEM_OBJECTS
      WaitingOn: None
      Description: Migrate 1 cl_mem objects 0x0x67de00  with flags 0
      name: Event-0x662a50
    Event-0x69b230
      Status: Scheduled
      Uid: 2
      TotalWorkGroups: 1
      Type: CL_COMMAND_NDRANGE_KERNEL
      WaitingOn: None
      name: Event-0x69b230
      CompletedWorkGroups: 0
      KernelName: run
```

***Note*:** The information presented in these views is only visible to the application developer while actually debugging the host code. This is the reason why this debug technique is also applicable when actual system execution (hardware) is performed.

GDB Kernel-Based Debugging

GDB kernel debugging is supported for the software emulation and hardware emulation flows. When the GDB executable is connected to the kernel in the IDE or command line flows, you can set breakpoints and query the content of variables in the kernel, similar to normal host code debugging. This is fully supported in the software emulation flow because the kernel GDB processes attach to the spawned software processes.

However, during hardware emulation, the kernel source code is transformed into RTL, created by Vivado HLS, and executed. As the RTL model is simulated, all transformations for performance optimization and concurrent hardware execution are applied. For that reason, not all C/C++/ OpenCL lines can be uniquely mapped to the RTL code, and only limited breakpoints are supported and at only specific variables can be queried. Today, the GDB tool therefore breaks on the next possible line based on requested breakpoint statements and clearly states if variables can not be queried based on the RTL transformations.

Related Information

Command Line Debug Flow

Debugging in Hardware Emulation

During hardware emulation, it is possible to deep dive into the implementation of the kernels. The SDAccel environment allows you to perform typical hardware-like debugging in this mode as well as some software-like GDB-based analysis on the hardware implementation.

GDB-Based Debugging

Debugging using a software-based GDB flow is fully supported during hardware emulation. Except for the execution of the actual RTL code representing the kernel code, there is no difference to the user because the GDB flow maps the RTL back into the source code description. This limits the breakpoint and observability of the variables in some cases, because during the RTL generation (HLS), variables and loops might have been dissolved.

For a detailed description of the debug feature itself please see the description in the Chapter 2: SDAccel Debug Features chapter, and the extensions to GDB as presented in the GDB-Based Debugging section.

Waveform-Based Kernel Debugging

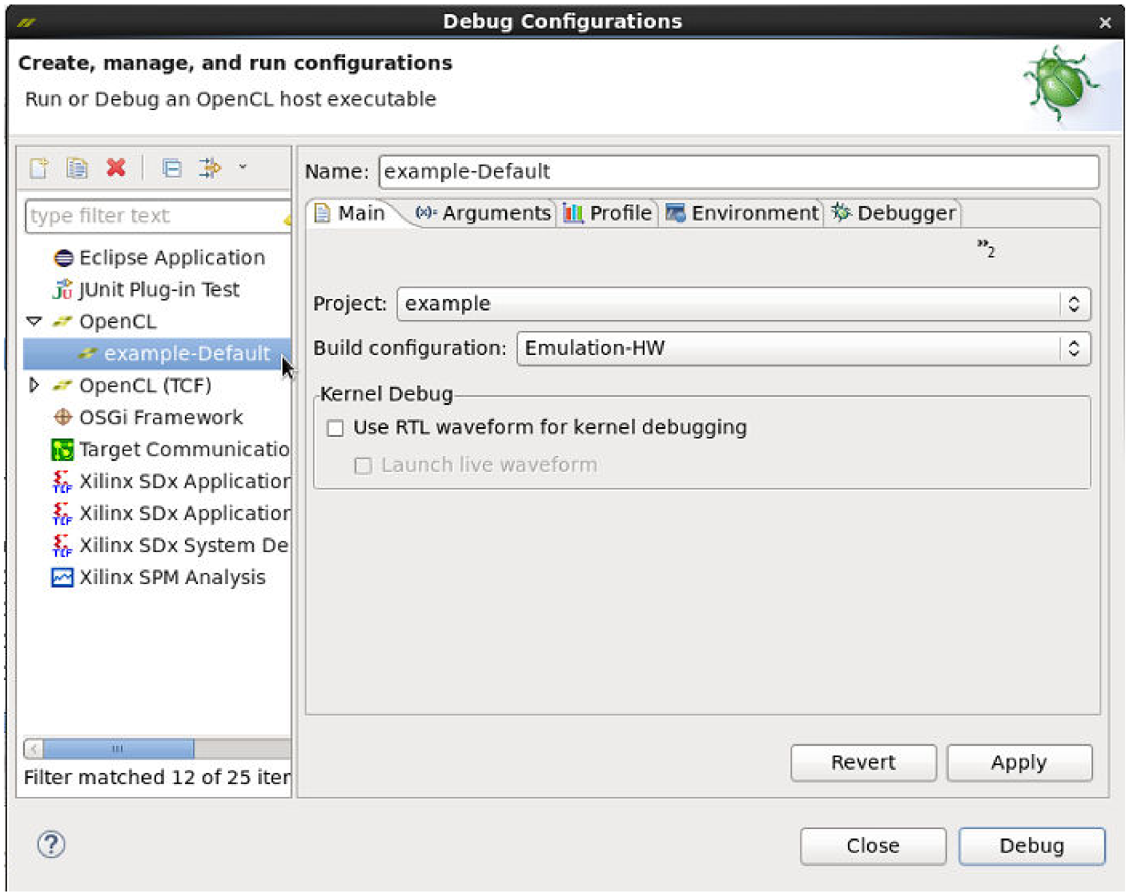
The C/C++ and OpenCL kernel code is synthesized using Vivado High Level Synthesis (HLS) to transform it into a Hardware Description Language (HDL) and later implement it onto the FPGA (`xclbin`).

Another debugging approach is based on simulation waveforms. Hardware-centric algorithm programmers are likely to be familiar with this approach. This waveform-based HDL debugging is best supported by the SDAccel environment through the IDE flow during hardware emulation.

TIP: *For most debugging, the HDL model does not need to be analyzed. Waveform debugging is considered an advanced debugging capability.*

Run the Waveform-Based Kernel Debugging Flow

1. Start the SDx environment, and perform the regular setup.
2. Select **Run → Debug Configurations** to open the Debug Configurations.
3. On the Debug Configurations window, select the current launch configuration from the OpenCL list, as shown in the following figure.



Debug Configurations

Create, manage, and run configurations

Run or Debug an OpenCL host executable

type filter text

- Eclipse Application
- JUnit Plug-in Test
- OpenCL
  - example-Default
- OpenCL (TCF)
- OSGi Framework
- Target Communicatio
- Xilinx SDx Application
- Xilinx SDx Application
- Xilinx SDx System De
- Xilinx SPM Analysis

Filter matched 12 of 25 iter

Name: example-Default

Main | Arguments | Profile | Environment | Debugger

Project: example

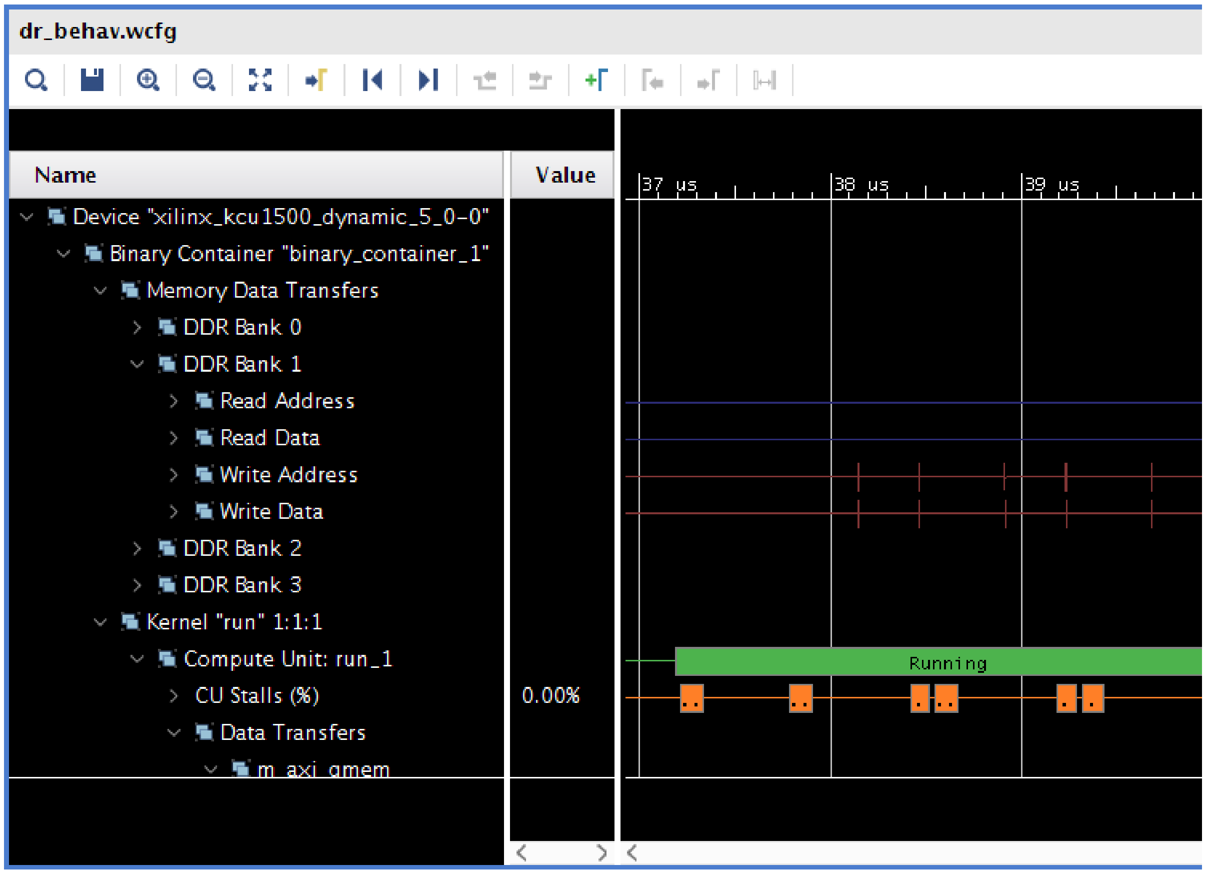
Build configuration: Emulation-HW

Kernel Debug

- [ ] Use RTL waveform for kernel debugging
- [ ] Launch live waveform

4. On the Main tab, two kernel debug options are displayed. Select both **Use RTL waveform for kernel debugging** and **Launch live waveform**, and close the configuration window. A debug session starts automatically. Selecting the Use RTL waveform for kernel debugging option ensures that a simulation waveform database is generated, while the Launch live waveform option spawns the Waveform viewer during the actual simulation, allowing you full control of the simulation engines and waveform display.

   If the live waveform viewer is activated, the waveform viewer automatically opens when running the executable. By default, the waveform viewer shows all interface signals and the following debug hierarchy:



- Memory Data Transfers: Shows data transfers from all compute units funnel through these interfaces.

TIP: *These interfaces could be a different bit width from the compute units. If so, then the burst lengths would be different. For example, a burst of sixteen 32-bit words at a compute unit would be a burst of one 512-bit word at the OCL master.*

- `Kernel` <kernel name><workgroup size> `Compute Unit`<CU name>
  - **CU Stalls (%)**: This section shows a summary of stalls for the entire compute unit (CU). A bus of all lowest-level stall signals is created, and the bus is represented in the waveform as a percentage (%) of those signals that are active at any point in time.
  - **Data Transfers**: This section shows the data transfers for all AXI masters on the CU.
  - **User Functions**: This section lists all of the functions within the hierarchy of the CU.
    - **Function**: `<function name>`
      - **Dataflow/Pipeline Activity**: This section shows the function-level loop dataflow/ pipeline signals for a CU.
      - **Function Stalls**: This section lists the three stall signals within this function.
      - **Function I/O**: This section lists the I/O for the function. These I/O are of protocol `-m_axi`, `ap_fifo`, `ap_memory`, or `ap_none`.

TIP: *As with any waveform debugger, additional debug data of internal signals can be added by selecting the instance of interest from the scope menu and the signals of interest from the object menu. Similarly, debug controls such as HDL breakpoints, as well as HDL code lookup and waveform markers are supported. Refer to the Vivado Design Suite User Guide: Logic Simulation (UG900) for more information on working with the waveform viewer.*

Enable Waveform Debugging through the XOCC Command Line

The waveform debugging process can also be enabled through the XOCC command line. Use the following instructions to enable it:

1. Turn on debug code generation during kernel compilation.

   ```
   xocc -g ...
   ```

2. Create an `sdaccel.ini` file in the same directory as the host executable with the contents below:

   ```
   [Emulation]
   launch_waveform=batch

   [Debug]
   profile=true
   timeline_trace=true
   data_transfer_trace=fine
   ```

3. Execute hardware emulation. The hardware transaction data is collected in the file named `<hardware_platform>-<device_id>-<xclbin_name>.wdb`. This file can directly be opened through the SDAccel IDE.

TIP: *If the `launch_waveform` option is set to `gui` in the emulation section: `[Emulation] launch_waveform=gui`, a live waveform viewer is spawned during the execution of the hardware emulation.*

System Verification and Hardware Debug

Application Hangs

This section discusses debugging issues related to the interaction of the host code and the accelerated kernels. Problems with these interactions manifest as issues such as machine hangs or application hangs. Although the GDB debug environment might help with isolating the errors in some cases (`xprint`), such as hangs associated with specific kernels, these issues are best debugged using the `dmesg` and `xbutil` commands as shown here.

If the process of hardware debugging does not resolve the problem, it is necessary to perform hardware debugging using the ChipScope™ feature.

Related Information
Utilities for Hardware Debugging
Debugging with ChipScope

AXI Firewall Trips

The AXI firewall should prevent host hangs. This is why Xilinx recommends the AXI Protocol Firewall IP to be included in SDAccel environment platforms. When the firewall trips, one of the first checks you perform should be to see if the host code and kernels are set up to use the same memory banks. The following steps detail one of the simplest methods to perform this check.

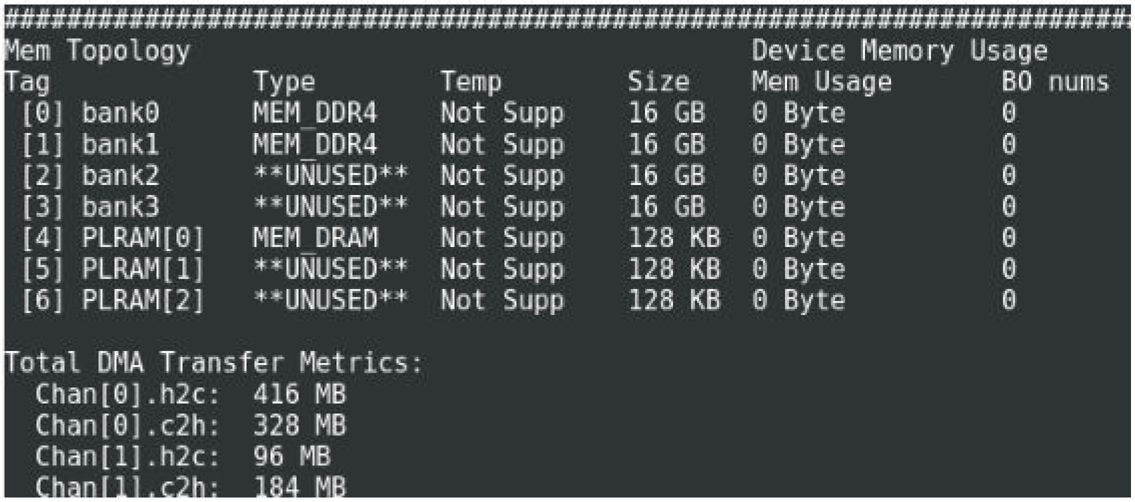
1. Use `xbutil` to program the FPGA:

   ```
   xbutil program -p <xclbin>
   ```

2. Run the `xbutil` query option to check memory topology:

   ```
   xbutil query
   ```

   In the following example, there is no memory bank associated with the kernels:



```
#########################################################################
Mem Topology                                      Device Memory Usage
Tag             Type        Temp        Size     Mem Usage       BO nums
 [0] bank0      MEM_DDR4    Not Supp    16 GB    0 Byte          0
 [1] bank1      MEM_DDR4    Not Supp    16 GB    0 Byte          0
 [2] bank2      **UNUSED**  Not Supp    16 GB    0 Byte          0
 [3] bank3      **UNUSED**  Not Supp    16 GB    0 Byte          0
 [4] PLRAM[0]   MEM_DRAM    Not Supp    128 KB   0 Byte          0
 [5] PLRAM[1]   **UNUSED**  Not Supp    128 KB   0 Byte          0
 [6] PLRAM[2]   **UNUSED**  Not Supp    128 KB   0 Byte          0

Total DMA Transfer Metrics:
  Chan[0].h2c:  416 MB
  Chan[0].c2h:  328 MB
  Chan[1].h2c:  96 MB
  Chan[1].c2h:  184 MB
```

3. If the host code expects any DDR banks/PLRAMs to be used, this report should indicate an issue. In this case, it is necessary to check kernel and host code expectations. If the host code is using the Xilinx OpenCL extensions, it is necessary to check which DDR banks should be used by the kernel. These should match the `xocc -sp` arguments provided.

Kernel Hangs due to AXI Violations

It is possible for the kernels to hang due to bad AXI transactions between the kernels and the memory controller. To debug these issues, it is required to instrument the kernels.

1. The SDAccel environment provides two options for instrumentation to be applied during XOCC linking (`-l`). Both of these add hardware to your implementation, and based on utilization it might be necessary to limit instrumentation.

a. Add Lightweight AXI Protocol Checkers (`lapc`). These protocol checkers are added using the `--dk` option. The following syntax is used:

```
--dk <[protocol|list_ports]<:compute_unit_name><:interface_name>>
```

In general, the `<interface_name>` is optional. If not specified, all ports are expected to be analyzed. The `protocol` option is used to define the protocol checkers to be inserted. This option can accept a special keyword, `all`, for `<compute_unit_name>` and/or `<interface_name>`. The `list_ports` option generates a list of valid compute units and port combinations in the current design.

***Note*:** Multiple `--dk` option switches can be specified in a single command line to additively add interface monitoring capability.

b. Adding SDx environment Performance Monitors (`spm`) enables the listing of detailed communication statistics (counters). Although this is most useful for performance analysis, it provides insight during debugging on pending port activities. The Performance Monitors are added using the `profile_kernel` option. The basic syntax for `profile_kernel` option is:

```
--profile_kernel data:<krnl_name|all>:<cu_name|all>:<intrfc_name|
all>:<counters|all>
```

Three fields are required to determine the precise interface to which the performance monitor is applied. However, if resource use is not an issue, the keyword `all` enables you to apply the monitoring to all existing kernels, compute units, and interfaces with a single option. Otherwise, you can specify the `kernel_name`, `cu_name`, and `interface_name` explicitly to limit instrumentation.

The last option, `<counters|all>`, allows you to restrict the information gathering to just `counters` for large designs, while `all` (default) includes the collection of actual trace information.

***Note*:** Multiple `--profile_kernel` option switches can be specified in a single command line to additively add performance monitoring capability.

```
--profile_kernel data:kernel1:cu1:m_axi_gmem0
--profile_kernel data:kernel1:cu1:m_axi_gmem1
--profile_kernel data:kernel2:cu2:m_axi_gmem
```

2. When the application is rebuilt, rerun the host application using the `xclbin` with the added SPM IP and LAPC IP.
3. When the application hangs, you can use `xbutil status` to check for any errors or anomalies.
4. Check the SPM output:
   - Run `xbutil status --spm` a couple of times to check if any counters are moving. If they are moving then the kernels are active.

TIP: *Testing SPM output is also supported through GDB debugging using the command extension* `xstatus spm`*.*

- If the counters are stagnant, the outstanding counts greater than zero might mean some AXI transactions are hung.

5. Check the LAPC output:

   - Run `xbutil status --lapc` to check if there are any AXI violations.

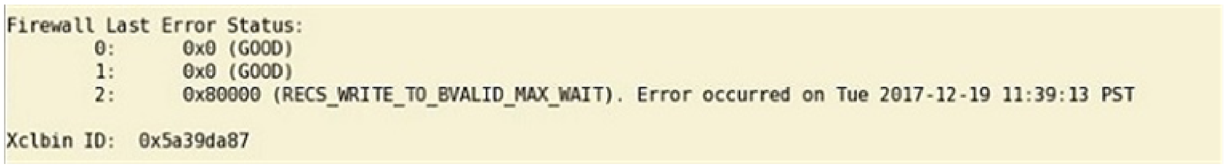
TIP: *Testing LAPC output is also supported through GDB debugging using the command extension `xstatus lapc`.*

   - If there are any AXI violations, it implies that there are problems in the kernel implementation.

Host Application Hangs when Accessing Memory

Application hangs can also be caused by incomplete DMA transfers initiated from the host code. This does not necessarily mean that the host code is wrong; it might also be that the kernels have issued illegal transactions and locked up the AXI.

1. If the platform has an AXI firewall, such as in the SDAccel platforms, it is likely to trip. The driver issues a `SIGBUS` error, kills the application, and resets the device. You can check this by running `xbutil query`. The following figure shows such an error in the firewall status:



```
Firewall Last Error Status:
        0:      0x0 (GOOD)
        1:      0x0 (GOOD)
        2:      0x80000 (RECS_WRITE_TO_BVALID_MAX_WAIT). Error occurred on Tue 2017-12-19 11:39:13 PST

Xclbin ID:  0x5a39da87
```

TIP: *If the firewall has not tripped, the Linux tool, `dmesg`, can provide additional insight.*

2. When you know that the firewall has tripped, it is important to determine the cause of the DMA timeout. The issue could be an illegal DMA transfer, or kernel misbehavior. However, a side effect of the AXI firewall tripping is that the health check functionality in the driver resets the board after killing the application; any information on the device that might help with debugging the root cause is lost. To debug this problem, you can disable the health check thread in the `xclmgmt` kernel module to capture the error. This uses common Unix kernel tools in the following sequence:

   a. `sudo modinfo xclmgmt`: This command lists the current configuration of the module and indicates if the `health_check` parameter is on or off. It also returns the path to the `xclmgmt` module.

   b. `sudo rmmod xclmgmt`: This removes and therefore disables the `xclmgmt` kernel module.

   c. `sudo insmod <path to module>/xclmgmt.ko health_check=0`: This reinstalls the `xclmgmt` kernel module with the health check disabled.

TIP: *The path to this module is reported in the output of the call to `modinfo`.*

3. With the health check disabled, rerun the application. You can use the kernel instrumentation to isolate this issue as previously described.

Related Information

Kernel Hangs due to AXI Violations

Using the Linux dmesg Utility

Typical Errors Leading to Application Hangs

The user errors that typically create application hangs are listed below:

- Read-before-write in 5.0+ shells causes an MIG ECC (Memory Interface Generator error correction code) error. This is typically a user error. For example, this error might occur when a kernel is expected to write 4KB of data in DDR, but it produces only 1KB of data, and you then try to transfer the full 4KB of data to the host. It can also happen if you supply a 1KB buffer to a kernel, but the kernel tries to read 4KB of data.
- An ECC read-before-write error also occurs if no data has been written to a memory location since the last bitstream download which results in MIG initialization, but a read request is made for that same memory location. ECC errors stall the affected MIG because kernels are usually not able to handle this error. This can manifest in two different ways:
  1. The CU might hang or stall because it cannot handle this error while reading or writing to or from the affected MIG. The `xbutil` query shows that the CU is stuck in a `BUSY` state and is not making progress.
  2. The AXI Firewall might trip if a PCIe® DMA request is made to the affected MIG, because the DMA engine is unable to complete the request. AXI Firewall trips result in the Linux kernel driver killing all processes which have opened the device node with the `SIGBUS` signal. The `xbutil` query shows if an AXI Firewall has indeed tripped, and includes a timestamp.

  If the above hang does not occur, the host code might not read back the correct data. This incorrect data is typically 0s, and is located in the last part of the data. It is important to review the host code carefully. One common example is compression, where the size of the compressed data is not known up front, and an application might try to migrate more data to the host than was produced by the kernel.

Debugging with ChipScope

You can use the ChipScope debugging environment and the Vivado hardware manager to help you debug your host application and kernels quickly and more effectively. In order to do this, at least one of the following must be true:

- Your SDAccel application project has been instrumented with debug cores, using the `--dk` compiler switch (as described in Hardware Debugging Using ChipScope).
- The RTL kernels used in your project must have been instantiated with debug cores (as described in Adding Debug IP to RTL Kernels).

These tools enable a wide range of capabilities from logic to system level debug while your kernel is running in hardware.

***Note*:** Debugging on the kernel platform requires additional logic to be incorporated into the overall hardware model, which might have an impact on resource use and kernel performance.

Running XVC and HW Servers

The following steps are required to run the XVC (Xilinx Virtual Cable) and HW servers, host applications, and finally trigger and arm the debug cores in Vivado hardware manager.

1. Add debug IP to the kernel.
2. Instrument the host application to pause at appropriate point in the host execution where you want to debug. See Debugging through the Host Application.
3. Set up the environment for hardware debug. You can do this manually, or by using a script that automates this for you. The following steps are described in Manual Setup for Hardware Debug and Automated Setup for Hardware Debug:
    a. Run the required XVC and HW servers.
    b. Execute the host application and pause at the appropriate point in the host execution to enable setup of ILA triggers.
    c. Open Vivado hardware manager and connect to the XVC server.
    d. Set up ILA trigger conditions for the design.
    e. Continue with host application.
    f. Inspect results in the Vivado hardware manager.
    g. Rerun iteratively from step b (above) as required.

Adding Debug IP to RTL Kernels

IMPORTANT! *This debug technique requires familiarity with the Vivado Design Suite, and RTL design.*

You need to instantiate debug cores like the Integrated Logic Analyzer (ILA) and Virtual Input/Output (VIO) in your RTL kernel code to debug the kernel logic. From within the Vivado Design Suite, edit the RTL kernel to instantiate an ILA IP customization, or a VIO IP, into the RTL code, similar to using any other IP in Vivado IDE. Refer to the *Vivado Design Suite User Guide: Programming and Debugging* (UG908) to learn more about using the ILA or other debug cores in the RTL Insertion flow and to learn about using the HDL generate statement technique to enable/disable debug core generation.

TIP: *The best time to add debug cores to your RTL kernel is when you create it. Refer to the Debugging section in the UltraFast Design Methodology Guide for the Vivado Design Suite (UG949) for more information.*

You can also add the ILA debug core using a Tcl script from within an open Vivado project as shown in the following code example:

```
create_ip -name ila -vendor xilinx.com -library ip -version 6.2 -
module_name ila_0
set_property -dict [list CONFIG.C_PROBE6_WIDTH {32} CONFIG.C_PROBE3_WIDTH
{64} \
CONFIG.C_NUM_OF_PROBES {7} CONFIG.C_EN_STRG_QUAL {1}
CONFIG.C_INPUT_PIPE_STAGES {2} \
CONFIG.C_ADV_TRIGGER {true} CONFIG.ALL_PROBE_SAME_MU_CNT {4}
CONFIG.C_PROBE6_MU_CNT {4} \
CONFIG.C_PROBE5_MU_CNT {4} CONFIG.C_PROBE4_MU_CNT {4}
CONFIG.C_PROBE3_MU_CNT {4} \
CONFIG.C_PROBE2_MU_CNT {4} CONFIG.C_PROBE1_MU_CNT {4}
CONFIG.C_PROBE0_MU_CNT {4}] [get_ips ila_0]
```

The following is an example of an ILA debug core instantiated into the RTL kernel source file of the RTL Kernel Debug example design on GitHub. The ILA monitors the output of the combinatorial adder as specified in the `src/hdl/krnl_vadd_rtl_int.sv` file.

```
    // ILA monitoring combinatorial adder
    ila_0 i_ila_0 (
        .clk(ap_clk),                  // input wire        clk
        .probe0(areset),               // input wire [0:0]  probe0
        .probe1(rd_fifo_tvalid_n),     // input wire [0:0]  probe1
        .probe2(rd_fifo_tready),       // input wire [0:0]  probe2
        .probe3(rd_fifo_tdata),        // input wire [63:0] probe3
        .probe4(adder_tvalid),         // input wire [0:0]  probe4
        .probe5(adder_tready_n),       // input wire [0:0]  probe5
        .probe6(adder_tdata)           // input wire [31:0] probe6
    );
```

After the RTL kernel has been instrumented for debug with the appropriate debug cores, you can analyze the hardware in the Vivado hardware manager features as described in the previous topic.

Debugging through the Host Application

To debug the host application working with the kernel code running on the SDAccel platform, the application host code must be modified to ensure that you can set up the ILA trigger conditions *after* the kernel has been programmed into the device, but *before* starting the kernel.

Pausing a C++ Host Application

The following code example is from the `src/host.cpp` code from the RTL Kernel example on GitHub:

```
....
    std::string binaryFile = xcl::find_binary_file(device_name,"vadd");

    cl::Program::Binaries bins = xcl::import_binary_file(binaryFile);
    devices.resize(1);
    cl::Program program(context, devices, bins);
    cl::Kernel krnl_vadd(program,"krnl_vadd_rtl");
```

```
     wait_for_enter("\nPress ENTER to continue after setting up ILA
trigger...");

    //Allocate Buffer in Global Memory
    std::vector<cl::Memory> inBufVec, outBufVec;
    cl::Buffer buffer_r1(context,CL_MEM_USE_HOST_PTR | CL_MEM_READ_ONLY,
            vector_size_bytes, source_input1.data());
    ...

    //Copy input data to device global memory
    q.enqueueMigrateMemObjects(inBufVec,0/* 0 means from host*/);

    //Set the Kernel Arguments
    ...

    //Launch the Kernel
    q.enqueueTask(krnl_vadd);
```

The addition of the conditional `if (interactive)` test and the use of the `wait_for_enter` function pause the host application to give the ILA time to set up the required triggers and prepare to capture data from the kernel. After the Vivado hardware manager is set up and configured properly, you can press `Enter` to continue running the host application.

Pausing the Host Application Using GDB

Instead of making changes to the host application to pause before a kernel execution, you can run a GDB session from the SDx IDE. You can then set a breakpoint prior to the kernel execution in the host application. When the breakpoint is reached, you can set up the debug ILA triggers in Vivado hardware manager, arm the trigger, and then resume the kernel execution in GDB.

Automated Setup for Hardware Debug

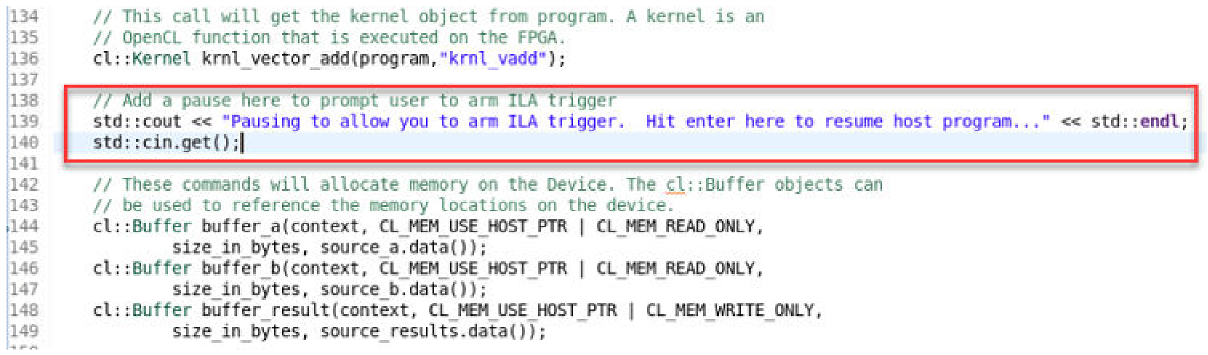
Note: A full SDx environment install is required to complete the following task. See the *SDAccel Environment Release Notes, Installation, and Licensing Guide* (UG1238) for more information about installation.

1. Set up your SDx environment by sourcing the appropriate `settings64.sh/.csh` file found in your SDx install area.
2. Start `xvc_pcie` and `hw_server` apps using the `sdx_debug_hw` script.

   ```
   sdx_debug_hw --xvc_pcie /dev/xvc_pub.m1025 --hw_server
   launching xvc_pcie...
   xvc_pcie -d /dev/xvc_pub.m1025 -s TCP::10200
   launching hw_server...
   hw_server -sTCP::3121
   ```

   Note: The `/dev/xvc_*` character device will differ depending on the platform. In this example, the character device is `/dev/xvc_pub.m1025`, though on your system it is likely to differ.
3. In the SDx IDE, modify the host code to include a pause statement *after* the kernel has been created/downloaded and *before* the kernel execution is started, then recompile the host program.

- For C++ host code, add a pause after the creation of the `cl::Kernel` object. The following snippet is from the Vector Add template design C++ host code:

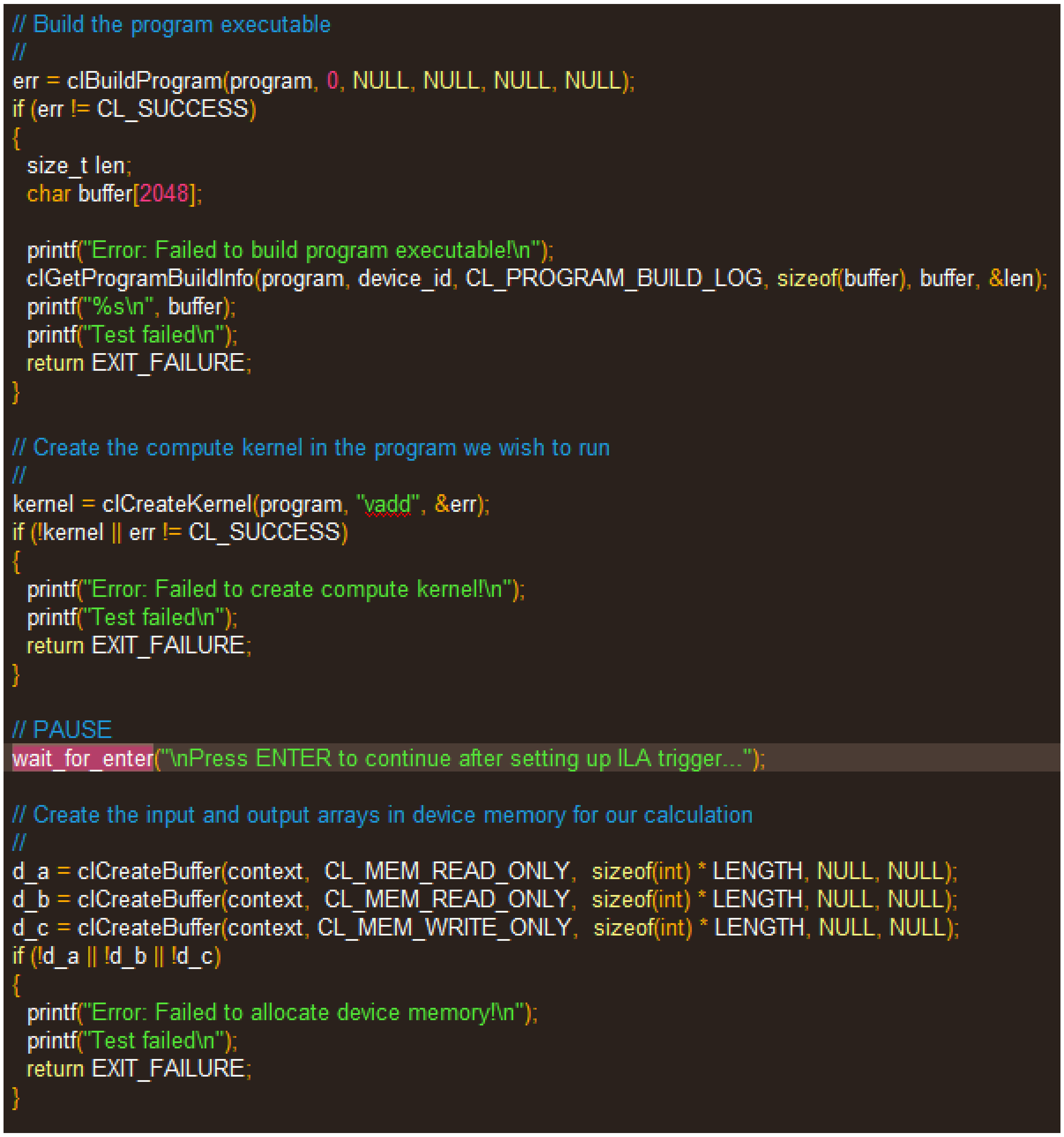


```
    // This call will get the kernel object from program. A kernel is an
    // OpenCL function that is executed on the FPGA.
    cl::Kernel krnl_vector_add(program,"krnl_vadd");

    // Add a pause here to prompt user to arm ILA trigger
    std::cout << "Pausing to allow you to arm ILA trigger.  Hit enter here to resume host program..." << std::endl;
    std::cin.get();

    // These commands will allocate memory on the Device. The cl::Buffer objects can
    // be used to reference the memory locations on the device.
    cl::Buffer buffer_a(context, CL_MEM_USE_HOST_PTR | CL_MEM_READ_ONLY,
            size_in_bytes, source_a.data());
    cl::Buffer buffer_b(context, CL_MEM_USE_HOST_PTR | CL_MEM_READ_ONLY,
            size_in_bytes, source_b.data());
    cl::Buffer buffer_result(context, CL_MEM_USE_HOST_PTR | CL_MEM_WRITE_ONLY,
            size_in_bytes, source_results.data());
```

- For C-language host code, add a pause after the `clCreateKernel()` function call:



```
// Build the program executable
//
err = clBuildProgram(program, 0, NULL, NULL, NULL, NULL);
if (err != CL_SUCCESS)
{
  size_t len;
  char buffer[2048];

  printf("Error: Failed to build program executable!\n");
  clGetProgramBuildInfo(program, device_id, CL_PROGRAM_BUILD_LOG, sizeof(buffer), buffer, &len);
  printf("%s\n", buffer);
  printf("Test failed\n");
  return EXIT_FAILURE;
}

// Create the compute kernel in the program we wish to run
//
kernel = clCreateKernel(program, "vadd", &err);
if (!kernel || err != CL_SUCCESS)
{
  printf("Error: Failed to create compute kernel!\n");
  printf("Test failed\n");
  return EXIT_FAILURE;
}

// PAUSE
wait_for_enter("\nPress ENTER to continue after setting up ILA trigger...");

// Create the input and output arrays in device memory for our calculation
//
d_a = clCreateBuffer(context,  CL_MEM_READ_ONLY,  sizeof(int) * LENGTH, NULL, NULL);
d_b = clCreateBuffer(context,  CL_MEM_READ_ONLY,  sizeof(int) * LENGTH, NULL, NULL);
d_c = clCreateBuffer(context, CL_MEM_WRITE_ONLY,  sizeof(int) * LENGTH, NULL, NULL);
if (!d_a || !d_b || !d_c)
{
  printf("Error: Failed to allocate device memory!\n");
  printf("Test failed\n");
  return EXIT_FAILURE;
}
```

4. Run your modified host program.

```
vadd_test.exe ./binary_container_1.xclbin
Loading: './binary_container_1.xclbin'
Pausing to allow you to arm ILA trigger. Hit enter here to resume host
program...
```

5. Launch Vivado Design Suite using the `sdx_debug_hw` script located in your SDAccel installation directory.

```
> sdx_debug_hw --vivado --host xcoltlab40 --ltx_file ../workspace/
vadd_test/System/pfm_top_wrapper.ltx
```

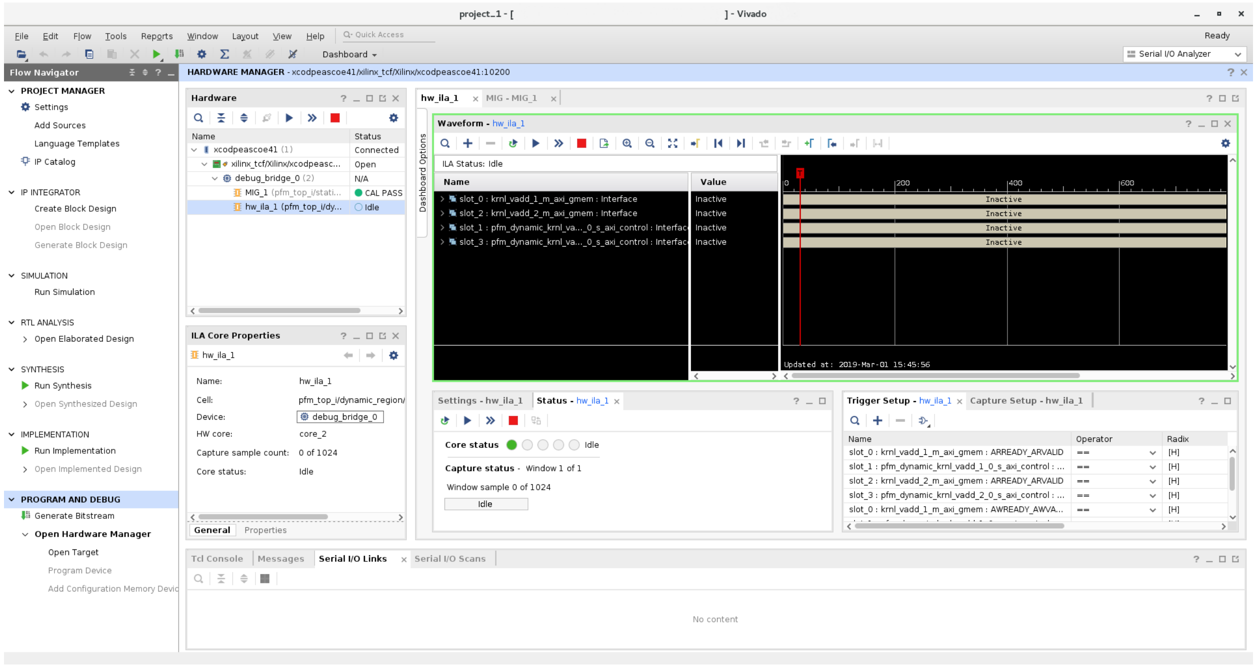
The command window displays the following:

```
launching vivado... ['vivado', '-source', 'sdx_hw_debug.tcl', '-
tclargs', '/tmp/sdx_tmp/project_1/project_1.xpr', 'workspace/vadd_test/
System/pfm_top_wrapper.ltx', 'xcoltlab40', '10200', '3121']

****** Vivado v2018.2 (64-bit)
  **** SW Build 2245749 on Wed May 30 12:36:19 MDT 2018
  **** IP Build 2245576 on Wed May 30 15:12:50 MDT 2018
    ** Copyright 1986-2018 Xilinx, Inc. All Rights Reserved.

start_gui
```

6. In Vivado Design Suite, run the ILA trigger.



HARDWARE MANAGER - xcodpeascoe41/xilinx_tcf/Xilinx/xcodpeascoe41:10200

Name	Status
xcodpeascoe41 (1)	Connected
xilinx_tcf/Xilinx/xcodpeasc...	Open
debug_bridge_0 (2)	N/A
MIG_1 (pfm_top_i/stati...	● CAL PASS
hw_ila_1 (pfm_top_i/dy...	○ Idle

ILA Core Properties

Name:	hw_ila_1
Cell:	pfm_top_i/dynamic_region/
Device:	debug_bridge_0
HW core:	core_2
Capture sample count:	0 of 1024
Core status:	Idle

Waveform - hw_ila_1

ILA Status: Idle

Name	Value
slot_0 : krnl_vadd_1_m_axi_gmem : Interface	Inactive
slot_2 : krnl_vadd_2_m_axi_gmem : Interface	Inactive
slot_1 : pfm_dynamic_krnl_va..._0_s_axi_control : Interface	Inactive
slot_3 : pfm_dynamic_krnl_va..._0_s_axi_control : Interface	Inactive

Updated at: 2019-Mar-01 15:45:56

Status - hw_ila_1

Core status: Idle

Capture status - Window 1 of 1

Window sample 0 of 1024

Idle

Trigger Setup - hw_ila_1

Name	Operator	Radix
slot_0 : krnl_vadd_1_m_axi_gmem : ARREADY_ARVALID	==	[H]
slot_1 : pfm_dynamic_krnl_vadd_1_0_s_axi_control : ...	==	[H]
slot_2 : krnl_vadd_2_m_axi_gmem : ARREADY_ARVALID	==	[H]
slot_3 : pfm_dynamic_krnl_vadd_2_0_s_axi_control : ...	==	[H]
slot_0 : krnl_vadd_1_m_axi_gmem : AWREADY_AWVA...	==	[H]

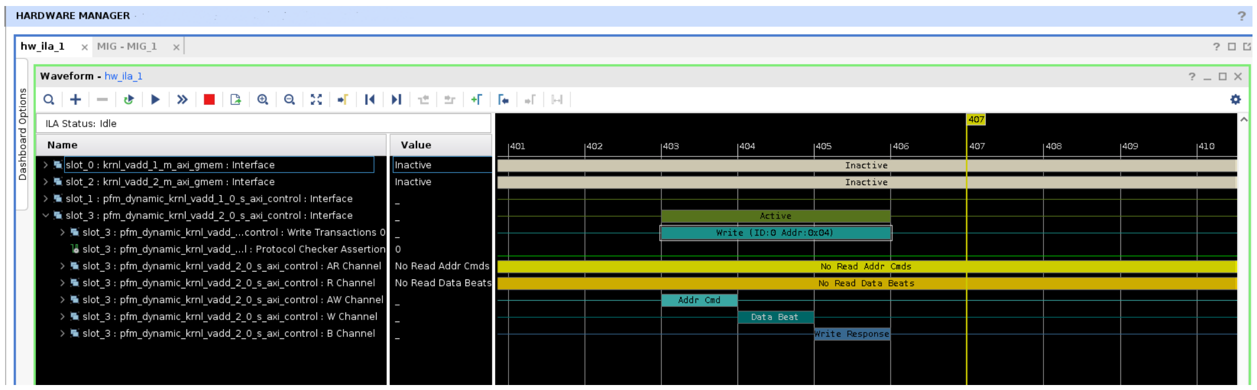
Serial I/O Links: No content

7. Press **Enter** to un-pause the host program.

```
vadd_test.exe ./binary_container_1.xclbin
Loading: './binary_container_1.xclbin'
Pausing to allow you to arm ILA trigger.  Hit enter here to resume host
program...

TEST PASSED
```

8. In the Vivado Design Suite, see the interface transactions on the kernel compute unit slave control interface in the Waveform view.



```
ILA Status: Idle
Name                                                              Value
slot_0 : krnl_vadd_1_m_axi_gmem : Interface                       Inactive
slot_2 : krnl_vadd_2_m_axi_gmem : Interface                       Inactive
slot_1 : pfm_dynamic_krnl_vadd_1_0_s_axi_control : Interface      _
slot_3 : pfm_dynamic_krnl_vadd_2_0_s_axi_control : Interface      _
  slot_3 : pfm_dynamic_krnl_vadd_...control : Write Transactions 0  _
  slot_3 : pfm_dynamic_krnl_vadd_...l : Protocol Checker Assertion  0
  slot_3 : pfm_dynamic_krnl_vadd_2_0_s_axi_control : AR Channel   No Read Addr Cmds
  slot_3 : pfm_dynamic_krnl_vadd_2_0_s_axi_control : R Channel    No Read Data Beats
  slot_3 : pfm_dynamic_krnl_vadd_2_0_s_axi_control : AW Channel   _
  slot_3 : pfm_dynamic_krnl_vadd_2_0_s_axi_control : W Channel    _
  slot_3 : pfm_dynamic_krnl_vadd_2_0_s_axi_control : B Channel    _
```

Manual Setup for Hardware Debug

Manually Starting Debug Servers

***Note*:** The following steps are also applicable when using Nimbix and other cloud platforms.

There are two steps required to start the debug servers prior to debugging the design in Vivado hardware manager.

1. Source the SDx environment setup script, `settings64.csh` or `settings64.sh`, and launch the `xvc_pcie` server. The filename passed to `xvc_pcie` must match the character driver file installed with the kernel device driver.

   ```
   >xvc_pcie -d /dev/xvc_pub.m1025
   ```

   ***Note*:** The `xvc_pcie` server has many useful command line options. You can issue `xvc_pcie -help` to obtain the full list of available options.

2. Start the XVC server on port 10201 and the `hw_server` on port 3121.

   ```
   >hw_server "set auto-open-servers xilinx-xvc:localhost:10201" -e "set
   always-open-jtag 1"
   ```

Starting Debug Servers on an Amazon F1 Instance

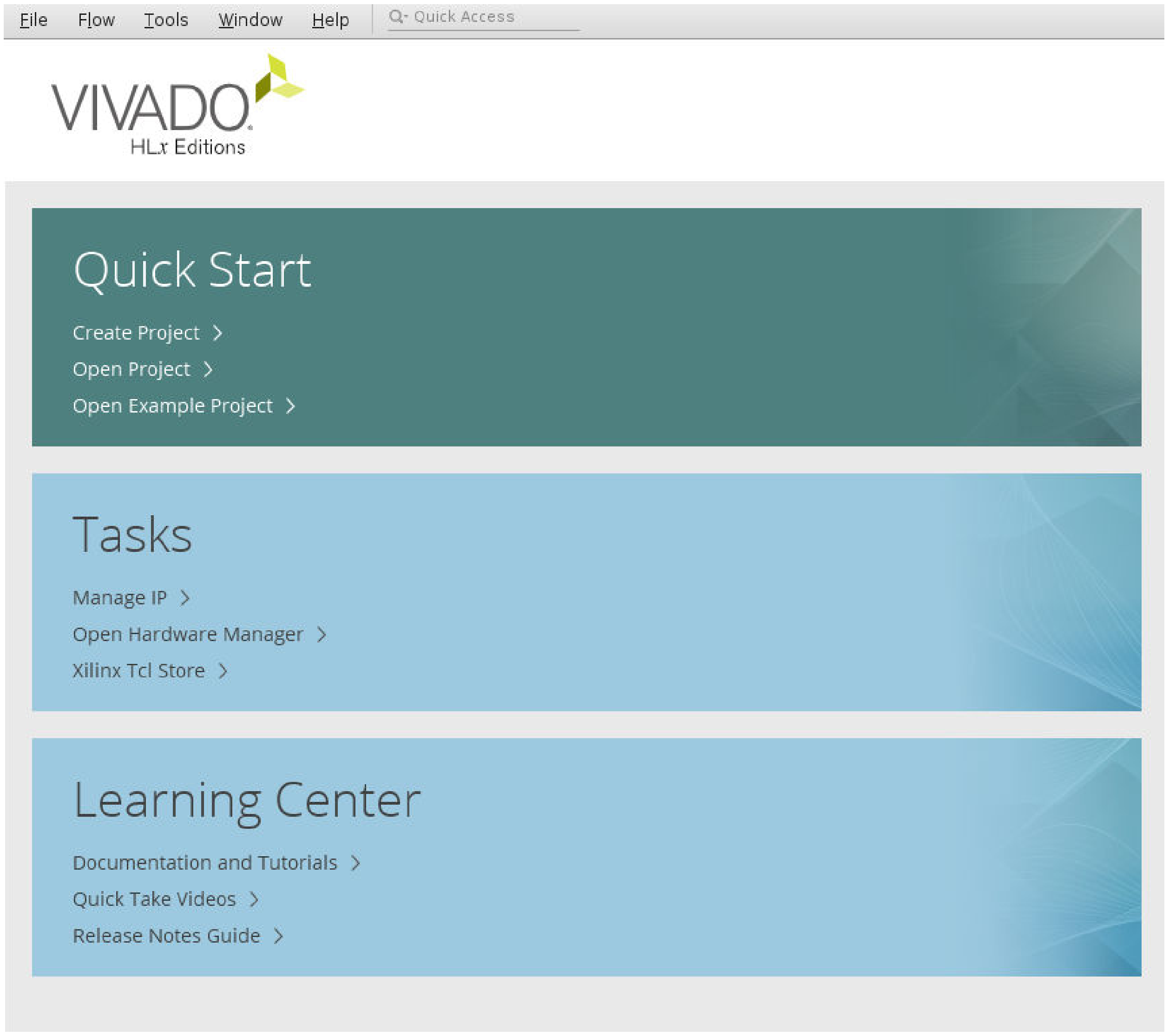
Instructions to start the debug servers on an Amazon F1 instance can be found here: https://github.com/aws/aws-fpga/blob/master/hdk/docs/Virtual_JTAG_XVC.md

Debugging Designs using Vivado Hardware Manager

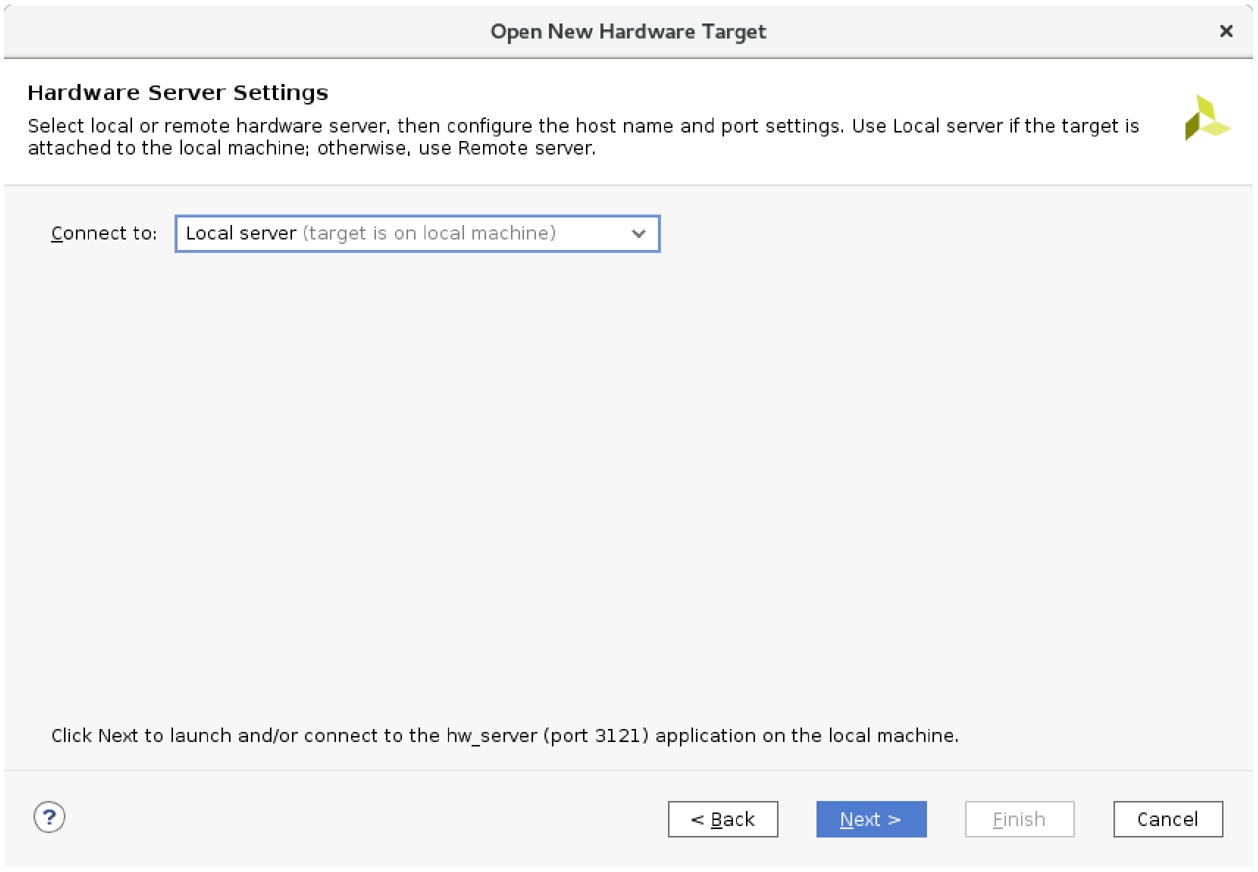
Traditionally, a physical JTAG connection is used to debug FPGAs. The SDAccel platforms have leveraged XVC for a debug flow that enables debug in the cloud. To take advantage of this capability, SDAccel enables running the XVC server. The XVC server is an implementation of Xilinx Virtual Cable (XVC) protocol, which allows the Vivado Design Suite to connect to a local or remote target FPGA for debug, using standard Xilinx debug cores like the Integrated Logic Analyzer IP (ILA), or the Virtual Input/Output IP (VIO), and others.

The Vivado hardware manager (Vivado Design Suite or Vivado Lab Edition) can be running on the target instance or it can be running remotely on a different host. The TCP port on which the XVC server is listening must be accessible to the host running Vivado hardware manager. To connect the Vivado hardware manager to XVC server on the target, the following steps should be followed on the machine hosting the Vivado tools:

1. Launch the Vivado Lab Edition, or the full Vivado Design Suite.
2. Select **Open Hardware Manager** from the Tasks menu, as shown in the following figure.



3. Connect to the Vivado tools `hw_server`, specifying a local or remote connection, and the **Host name** and **Port**, as shown below.



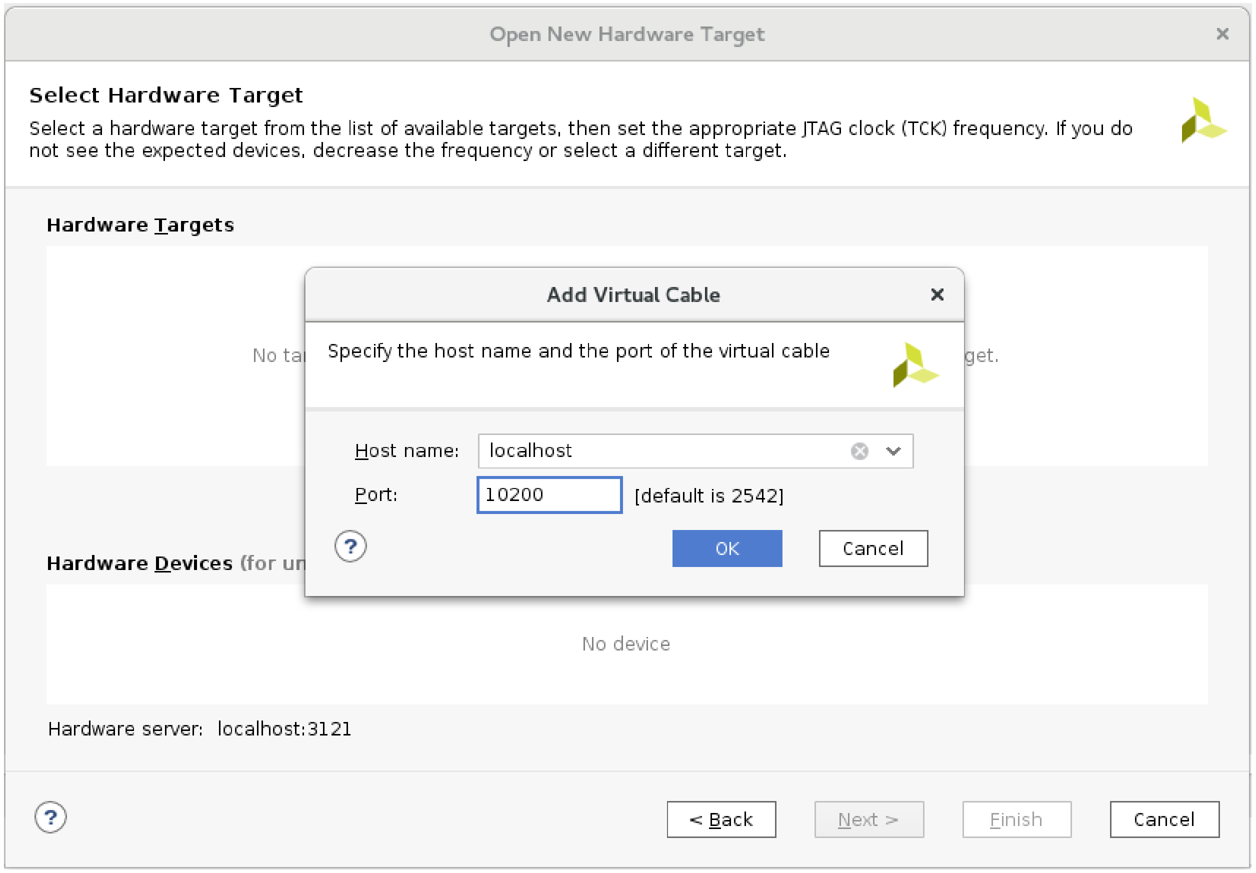
Hardware Server Settings

Select local or remote hardware server, then configure the host name and port settings. Use Local server if the target is attached to the local machine; otherwise, use Remote server.

Connect to: Local server (target is on local machine)

Click Next to launch and/or connect to the hw_server (port 3121) application on the local machine.

4. Connect to the target instance Virtual JTAG XVC server.


Open New Hardware Target

Select Hardware Target

Select a hardware target from the list of available targets, then set the appropriate JTAG clock (TCK) frequency. If you do not see the expected devices, decrease the frequency or select a different target.

Hardware Targets

No ta ... get.

Add Virtual Cable

Specify the host name and the port of the virtual cable

Host name: localhost

Port: 10200 [default is 2542]

Hardware Devices (for un

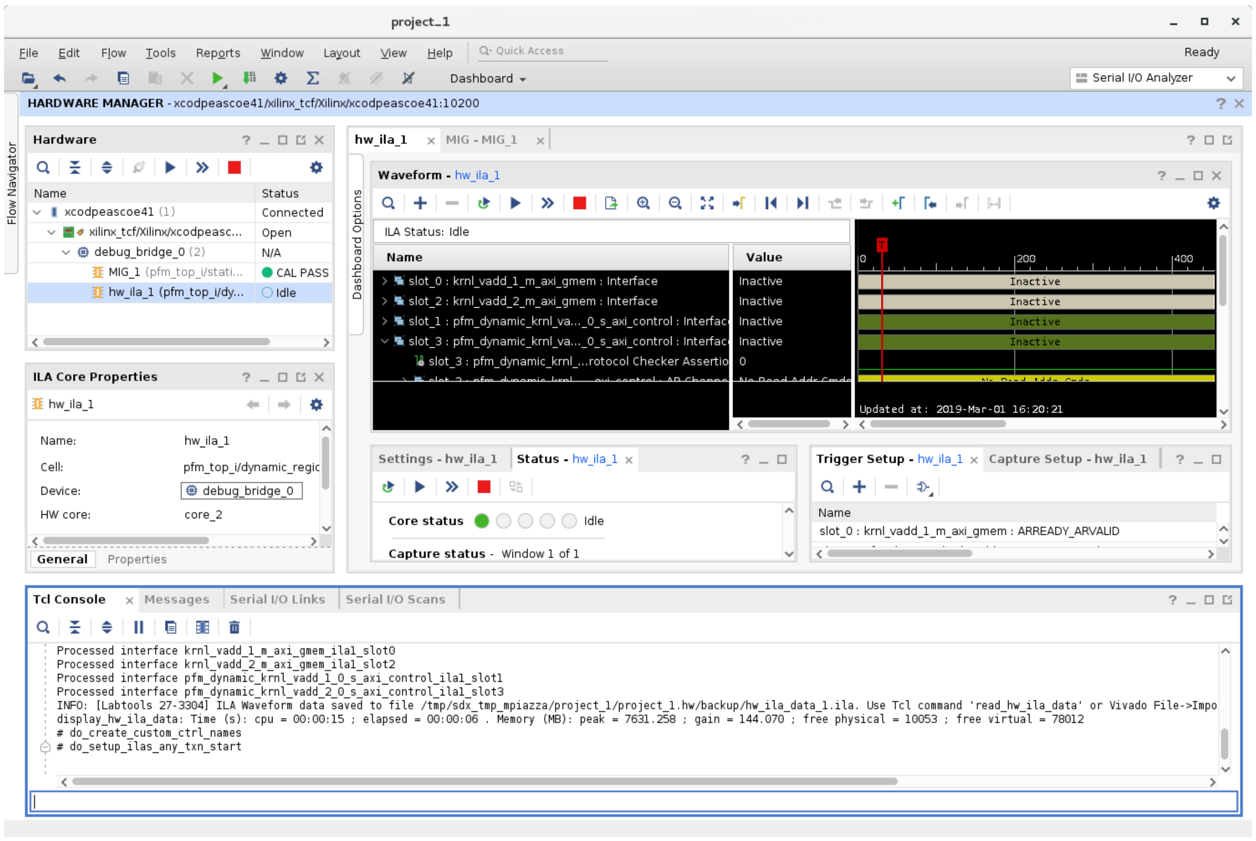
No device

Hardware server: localhost:3121

5. Select the debug bridge instance from the Hardware window of the Vivado hardware manager.
6. In the Hardware Device Properties window select the appropriate probes file for your design by clicking the icon next to the Probes file entry, selecting the file, and clicking **OK**. This refreshes the hardware device, and it should now show the debug cores present in your design.

TIP: *The probes file (`.ltx`) is written out during the implementation of the kernel by the Vivado tool, if the kernel has debug cores as specified in* Hardware Debugging Using ChipScope.

7. The Vivado hardware manager can now be used to debug the kernels running on the SDAccel platform. Refer to the *Vivado Design Suite User Guide: Programming and Debugging* (UG908) for more information on working with the Vivado hardware manager.



Name	Status
xcodpeascoe41 (1)	Connected
xilinx_tcf/Xilinx/xcodpeasc...	Open
debug_bridge_0 (2)	N/A
MIG_1 (pfm_top_i/stati...	● CAL PASS
hw_ila_1 (pfm_top_i/dy...	○ Idle

ILA Status: Idle

Name	Value
slot_0 : krnl_vadd_1_m_axi_gmem : Interface	Inactive
slot_2 : krnl_vadd_2_m_axi_gmem : Interface	Inactive
slot_1 : pfm_dynamic_krnl_va..._0_s_axi_control : Interface	Inactive
slot_3 : pfm_dynamic_krnl_va..._0_s_axi_control : Interface	Inactive
slot_3 : pfm_dynamic_krnl_...rotocol Checker Assertio	0

Updated at: 2019-Mar-01 16:20:21

```
Processed interface krnl_vadd_1_m_axi_gmem_ila1_slot0
Processed interface krnl_vadd_2_m_axi_gmem_ila1_slot2
Processed interface pfm_dynamic_krnl_vadd_1_0_s_axi_control_ila1_slot1
Processed interface pfm_dynamic_krnl_vadd_2_0_s_axi_control_ila1_slot3
INFO: [Labtools 27-3304] ILA Waveform data saved to file /tmp/sdx_tmp_mpiazza/project_1/project_1.hw/backup/hw_ila_data_1.ila. Use Tcl command 'read_hw_ila_data' or Vivado File->Impo
display_hw_ila_data: Time (s): cpu = 00:00:15 ; elapsed = 00:00:06 . Memory (MB): peak = 7631.258 ; gain = 144.070 ; free physical = 10053 ; free virtual = 78012
# do_create_custom_ctrl_names
# do_setup_ilas_any_txn_start
```

Debugging a MicroBlaze Processor (RTL Kernels Only)

Note: This technique requires familiarity with the Vivado Design Suite, RTL design, the MicroBlaze™ processor, and standard MicroBlaze debugging techniques.

In RTL kernel block designs, a MicroBlaze processor is included under the control hierarchy. To debug the software applications running on the MicroBlaze processor, a MicroBlaze Debug Module (MDM) can optionally be included in the RTL kernel block design, allowing standard MicroBlaze debugging techniques to take place over XVC. To enable MicroBlaze debugging, both of the following must be true:

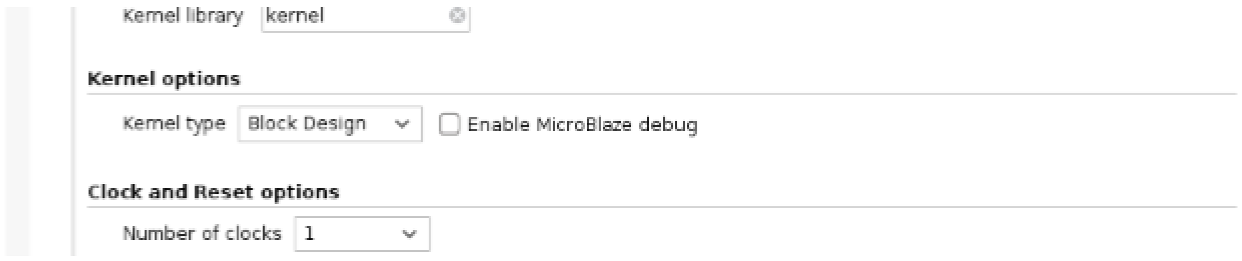
- The SDAccel environment platform must support MicroBlaze debugging over XVC.
- The RTL kernel must contain a MicroBlaze processor and MicroBlaze Debug Module (MDM).

The following platforms support hardware debug of a MicroBlaze processor:

- xilinx_u200_xdma_201830_1
- xilinx_u250_xdma_201830_1
- xilinx_vcu1525_xdma_201830_1

MicroBlaze debugging can optionally be enabled in the RTL Kernel Wizard user interface. When generating the RTL kernel, if the platform supports MicroBlaze debug, a checkbox appears in the wizard allowing the feature to be enabled. When this box is checked, the optional MicroBlaze Debug Module (MDM) is included in the control block of the RTL kernel. The following steps detail how to enable MicroBlaze debug on your RTL kernel during the generation of the kernel.

1. Launch the RTL Kernel Wizard by clicking **Xilinx → RTL Kernel Wizard**. When the RTL Kernel Wizard launches, click **Next**.
2. On the General Settings page, select **Block Design** as the kernel type, and check the box to **Enable MicroBlaze Debug**, as seen in the following figure:



Kernel library: kernel

Kernel options

Kernel type: Block Design ☐ Enable MicroBlaze debug

Clock and Reset options

Number of clocks: 1

Connecting to a MicroBlaze Processor in an RTL Kernel over XVC

When the RTL kernel has been generated with MicroBlaze debug and an `.xclbin` binary has been created, you can connect to the MicroBlaze processor embedded in the kernel while it is running in hardware to view hardware registers and perform standard MicroBlaze debugging techniques.

1. Set up your environment by sourcing the appropriate `settings64.sh/.csh` file found in your install area.
2. Start the `xvc_pcie` and `hw_server` apps using the `sdx_debug_hw` script, as shown in the following example:

```
sdx_debug_hw --xvc_pcie /dev/xvc_pub.m1025 --hw_server
launching xvc_pcie...
xvc_pcie -d /dev/xvc_pub.m1025 -s TCP::10200
launching hw_server...
hw_server -sTCP::3121
```

Note: The `/dev/xvc_*` character device differs depending on the platform. In this example, the character device is `/dev/xvc_pub.m1025`, though on your system it is likely to differ.

3. Launch the Xilinx Software Command Line Tool (XSCT):

```
$ xsct

****** Xilinx Software Commandline Tool (XSCT) v2018.3
  **** SW Build 2373407 on Thu Oct 25 21:12:35 MDT 2018
    ** Copyright 1986-2018 Xilinx, Inc. All Rights Reserved.


xsct%
```

4. Connect to the hardware server and XVC server to list the available targets:

```
xsct% connect -url tcp:localhost:3121 -xvc-url tcp:localhost:10200
tcfchan#0
xsct% targets
  1  debug_bridge
     2  00000000
     3  Legacy Debug Hub
     4  MicroBlaze Debug Module at USER1.1.2.2
        5  MicroBlaze #0 (Running)
xsct%
```

   Note: While this example uses a both a local hardware server and local XVC server, this is not a requirement. If you wish to use XSCT on a remote machine, replace `localhost` in the above example with the IP address or host name of the host on which `sdx_debug_hw` is running.

5. As can be seen, the MicroBlaze processor is listed as target number 5. It can be connected to by issuing the `targets -set` command. Listing the targets again shows that the MicroBlaze processor has been selected as the active target:

```
xsct% targets -set 5
xsct% targets
  1  debug_bridge
     2  00000000
     3  Legacy Debug Hub
    4  MicroBlaze Debug Module at USER1.1.2.2
        5* MicroBlaze #0 (Running)
```

6. At this point, standard MicroBlaze debugging techniques can be applied as described in the *MicroBlaze Processor Reference Guide* (UG984). For example, to list the contents of the MicroBlaze registers, `rrd` can be issued:

```
xsct% rrd
 r0: 0000000000000000   r1: 00000000000115e8   r2: 0000000000010960
 r3: 0000000000000006   r4: 0000000000000006   r5: 0000000000000000
 r6: 0000000000000000   r7: 0000000000000000   r8: 0000000000000000
 r9: 0000000000000000  r10: 0000000000000000  r11: 0000000000000000
r12: 0000000000000000  r13: 0000000000010a60  r14: 0000000000000000
r15: 0000000000010348  r16: 0000000000000000  r17: 0000000000000000
r18: 00000000ffffffff  r19: 00000000000115e8  r20: 0000000000000000
r21: 0000000000000000  r22: 0000000000000000  r23: 0000000000000000
r24: 0000000000000000  r25: 0000000000000000  r26: 0000000000000000
r27: 0000000000000000  r28: 0000000000000000  r29: 0000000000000000
r30: 0000000000000000  r31: 0000000000000000   pc: 00000000000106bc
msr:         00000010  ear: 0000000000000010  esr:         00000010
btr: 0000000000000010  edr:         00000010  dcr:         00000009
dsr:         21010000

xsct%
```

Examples

This chapter presents examples to illustrate particular debug techniques and flows.

Complete Command Line Debug Example

To get familiar with the command line flow in the SDx™ environment, take the IDCT example available from the Xilinx GitHub repository and compile it manually (no makefile) for debugging.

1. In a terminal, set up your environment by sourcing the SDx environment setting file to build the accelerated application:
   - C Shell: `source <SDX_INSTALL_DIR>/settings64.csh`
   - Bash: `source <SDX_INSTALL_DIR>/settings64.sh`

   Starting from 2018.3, debugging requires you to source the runtime environment, which is installed separately:
   - C Shell: `source /opt/xilinx/xrt/setup.sh`
   - Bash: `source /opt/xilinx/xrt/setup.sh`
2. Clone the complete SDAccel Examples GitHub repository to acquire the example code:

   ```
   git clone https://github.com/Xilinx/SDAccel_Examples.git
   ```

   This creates an `SDAccel_Examples` directory which includes the IDCT example. Move into the example directory:

   ```
   cd SDAccel_Examples/vision/idct/
   ```

   The host code is fully contained in `src/idct.cpp` and the kernel code is part of `src/krnl_idct.cpp`.
3. Compile the kernel software, using the option `-t sw_emu` to specify compilation for software emulation. In general, no additional options are required for hardware emulation, except for changing the `-t` option to `hw_emu`.
   a. The next step is to compile the kernel object file for debugging. The kernel is compiled using the `xocc` compiler:

   ```
   xocc -g -c -k krnl_idct -t sw_emu --platform <DEVICE> -o krnl_idct.xo
   src/krnl_idct.cpp
   ```

The `-g` option ensures that the code is compiled for debugging. The `-c` option instructs the compilation of the kernel, which is implemented by the `krnl_idct` function (`-k`). The target `sw_emu` (`-t`) determines that the output of this compilation is used for software emulation. The output of the compilation is intended to run on the `<DEVICE>` specified through `--platform` option. The generated Xilinx object file is called `krnl_idct.xo` and is specified with the `-o` option. The file to be compiled is the last argument.

b. Link the kernel object file. Linking allows multiple kernels to be combined, and provides the means to specify implementation directives. The following is the example link line for the IDCT:

```
xocc -g -l -t sw_emu --platform <DEVICE> --xp
"prop:solution.hls_pre_tcl=src/hls_config.tcl" --sp
krnl_idct_1.m_axi_gmem0:bank0 --sp krnl_idct_1.m_axi_gmem1:bank0 --sp
krnl_idct_1.m_axi_gmem2:bank1 --nk krnl_idct:1 -o krnl_idct.xclbin
krnl_idct.xo
```

Similarly to the compile line, the `-g` option is provided for debugging followed by the `-l` option to instruct `xocc` to perform object linking. The target and platform need to be presented again and need to be aligned with the compile step. The provided `--xp` option is an example of how to control the downstream tools such as HLS with arguments (in most cases, this is not required).

The `--sp` option is used to specify port bindings to specific DDR banks and PLRAMs. For optimization purposes, it is good to consider port binding for any larger designs. Each port can be bound individually to a DDR/PLRAM, and you are required to adhere to this same binding in the host code when allocating buffers.

The `--nk` option is used to specify multiple instances of a kernel. In this case, only one instance of `krnl_idct` is implemented in the final bitstream. The name of the bitstream is defined by the `-o` option before the different kernel object files are listed as the last argument.

4. Compile and link the host code for debugging. The host code is compiled with the GNU compiler chain, although it is wrapped under `xcpp`. Thus, separate compile and linking phases can also be performed. The host compilation is completely independent of the final target.

a. Compile host code C++ files:

```
xcpp -c -I${XILINX_XRT}/include -g -o idct.o src/idct.cpp
```

The `-c` option specifies a compile-only run, which creates an object file. The name of the object file is specified by the `-o` option. The `-I` option is using the runtime environment variable `XILINX_XRT` to specify the location of the common header files used by the host code. The `-g` option states that a debug compile is initiated. The final argument is the source file to be compiled in this step.

b. Link the object files:

```
xcpp -g -lOpenCL -lpthread -lrt -lstdc++ -L${XILINX_XRT}/lib/ -o idct
idct.o
```

Linking is performed again using the `-g` option to ensure debug information is included. Because the example uses the OpenCL™ interfaces and the runtime library, several additional libraries are included in the link process (`-l`) which are picked up in addition to the default library path from the path specified by `-L` option. Finally, the name of the executable is specified by the `-o` option and the previously generated object file is provided through the last argument.

5. Prepare the emulation environment. The following command is required for emulation runs:

```
emconfigutil --platform <device>
```

The actual emulation mode (`sw_emu` or `hw_emu`) then needs to be set through the `XCL_EMULATION_MODE` environment variable. In C-shell this would be as follows:

```
setenv XCL_EMULATION_MODE sw_emu
```

6. Run the debugger on host and kernel. As stated in the earlier chapter, running the debugger is best performed in the IDE. The following steps guide you through the command line debug process which requires three separate terminals, all prepared by sourcing the SDAccel™ environment as described in the first section of this description.

a. In the first terminal, start the SDx debug server:

```
${XILINX_VIVADO}/bin/sdx_server --sdx-url
```

b. In a second terminal, set the emulation mode:

```
setenv XCL_EMULATION_MODE sw_emu
```

Create an `sdaccel.ini` file in the `run` directory with the following content:

```
[Debug]
app_debug=true
```

Run GDB by executing the following:

```
xgdb --args idct krnl_idct.xclbin
```

Enter the following on the `gdb` prompt:

```
run
```

c. In the third terminal, attach the software emulation or hardware emulation model to GDB to allow stepping through the design. Here, there is a difference between running software emulation and hardware emulation. In either flow, start up another `xgdb`:

```
xgdb
```

- For software emulation:
  - Type the following on the `gdb` prompt:

    ```
    file <XILINX_SDX>/data/emulation/unified/cpu_em/generic_pcie/
    model/genericpciemodel
    ```

    ***Note*:** Because GDB does not expand the environment variable, it is easiest to replace `<XILINX_SDX>` with the actual value of `$XILINX_SDX`.

- For hardware emulation:
  1. Locate the `sdx_server` temporary directory: `/tmp/sdx/$uid`.
  2. Find the `sdx_server` process ID (PID) containing the DWARF file of this debug session.
  3. At the `gdb` prompt, run:

     ```
     file /tmp/sdx/$uid/$pid/NUM.DWARF
     ```

- In either case, connect to the kernel process:

  ```
  target remote :NUM
  ```

  Here, `NUM` is the number returned by the `sdx_server` as the GDB listener port.

At this point, debugging of the `sw_emu` and `hw_emu` can be done as usual with GDB. The only difference is that the host code and the kernel code are debugged in two different GDB sessions. This is common when dealing with different processes. It is most important to understand that a breakpoint in one process might be hit before the next breakpoint in the current process is hit. In these cases, the debugging session appears to hang, while the second terminal is waiting for input.

Appendix A

Additional Resources and Legal Notices

Documentation Navigator and Design Hubs

Xilinx® Documentation Navigator (DocNav) provides access to Xilinx documents, videos, and support resources, which you can filter and search to find information. DocNav is installed with the SDSoC™ and SDAccel™ development environments. To open it:

- On Windows, select **Start → All Programs → Xilinx Design Tools → DocNav**.
- At the Linux command prompt, enter `docnav`.

Xilinx Design Hubs provide links to documentation organized by design tasks and other topics, which you can use to learn key concepts and address frequently asked questions. To access the Design Hubs:

- In DocNav, click the **Design Hubs View** tab.
- On the Xilinx website, see the Design Hubs page.

Note: For more information on DocNav, see the Documentation Navigator page on the Xilinx website.

References

1. *SDAccel Environment Release Notes, Installation, and Licensing Guide* (UG1238)
2. *SDAccel Environment Profiling and Optimization Guide* (UG1207)
3. *SDAccel Environment Getting Started Tutorial* (UG1021)
4. SDAccel™ Development Environment web page
5. Vivado® Design Suite Documentation
6. *Vivado Design Suite User Guide: Designing IP Subsystems using IP Integrator* (UG994)
7. *Vivado Design Suite User Guide: Creating and Packaging Custom IP* (UG1118)

8. *Vivado Design Suite User Guide: Partial Reconfiguration* (UG909)
9. *Vivado Design Suite User Guide: High-Level Synthesis* (UG902)
10. *UltraFast Design Methodology Guide for the Vivado Design Suite* (UG949)
11. *Vivado Design Suite Properties Reference Guide* (UG912)
12. Khronos Group web page: Documentation for the OpenCL standard
13. Xilinx® Virtex® UltraScale+™ FPGA VCU1525 Acceleration Development Kit
14. Xilinx® Kintex® UltraScale™ FPGA KCU1500 Acceleration Development Kit
15. Xilinx® Alveo™ web page

Please Read: Important Legal Notices

Copyright