

## XUP Vitis Labs (2019.2)

1. Setup Vitis	2. Introduction to Vitis	3. Improving Performance	4. Optimization	5. RTL Kernel Wizard	6. Debugging	7. Vision Application	8. PYNQ Lab
----------------------	--------------------------------	-----------------------------	--------------------	----------------------------	-----------------	--------------------------	-------------------

# Hardware/Software Debugging

## Introduction

This lab is a continuation of the previous ([RTL-Kernel Wizard Lab](#)) lab. You will use ChipScope to monitor signals at the kernel interface level and perform software debugging using Vitis. Note that this lab is not currently supported on Nimble as the Xilinx Virtual Cable (XVC is not supported)

## Objectives

After completing this lab, you will be able to:

- Add ChipScope cores to a design created using Vitis
- Use ChipScope to monitor signals at the kernel interface
- Debug a software application in Vitis

## Steps

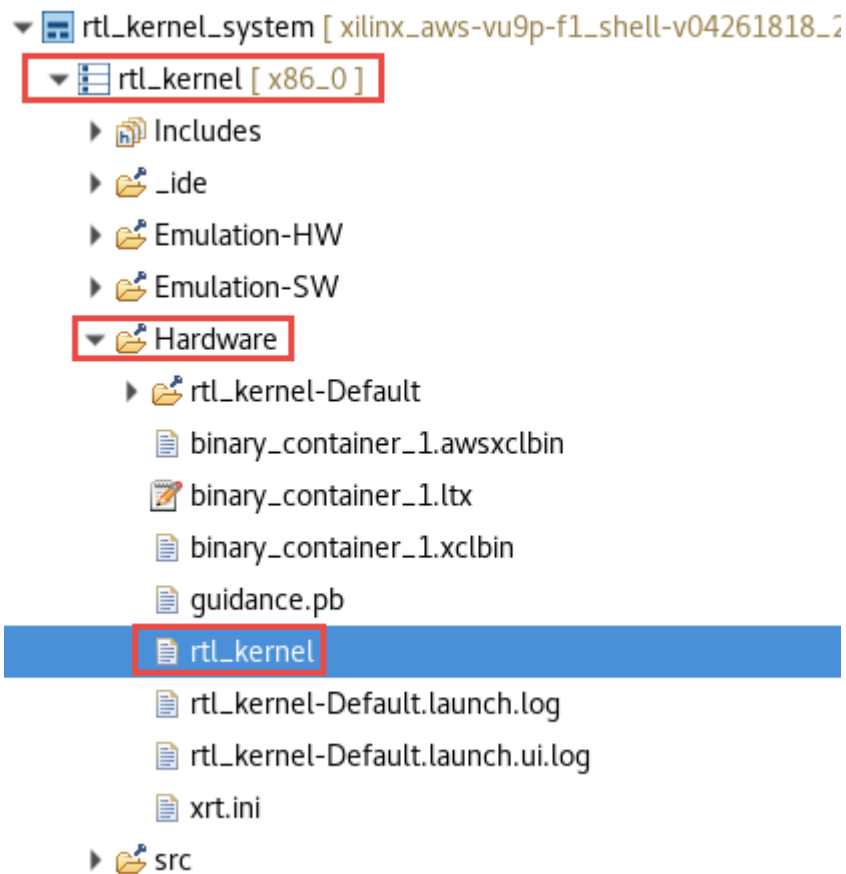
### Open Vitis and import the project

To save time on compilation, a precompiled project will be provided with the ChipScope debug cores already included in the design. See Appendix-I to learn how to add ChipScope debug cores

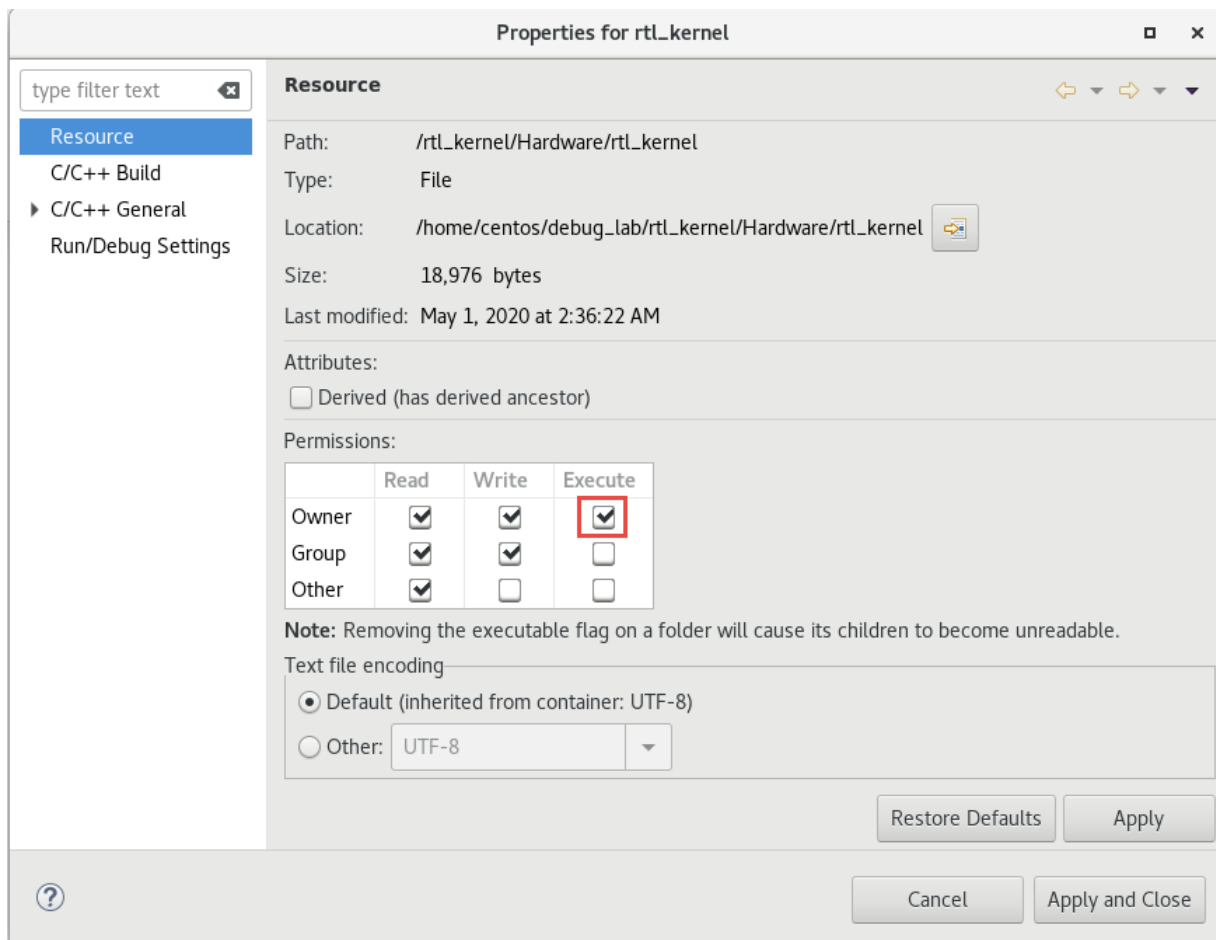
1. Open Vitis and select a new workspace at `~/debug_lab`
2. Since this is new workspace, click on **Add Custom Platform** link in the *Welcome* page, and add the target platform by clicking on "+" sign, browsing to the directory where the target platform is defined
3. From the *Welcome* page, click on the **Import Project** link
4. In the *Import Projects*, select **Vitis project exported zip file** and click **Next**
5. Click on **Browse** and select **debug\_lab** from `~/compute_acceleration/sources/` and click **OK**  
Make sure that the three hierarchical options are checked
6. Click **Next**

### Set permissions on imported executable

1. In the *Explorer* view, expand the hierarchy and double-click on **rtl\_kernel.prj**
2. Set **Hardware** as an *Active build configuration*
3. Expand **rtl\_kernel > Hardware**
4. Right click on *rtl\_kernel* and select **Properties**



5. Tick the box to add **Execute** to the *Owner* permissions, and click **Apply and Close**



6. If you don't see an option to set the permissions, open a terminal, browse to the directory containing the `rtl_kernel` (in Hardware directory), and run the following command to change the permissions to make the file executable:

```
chmod +x rtl_kernel
```

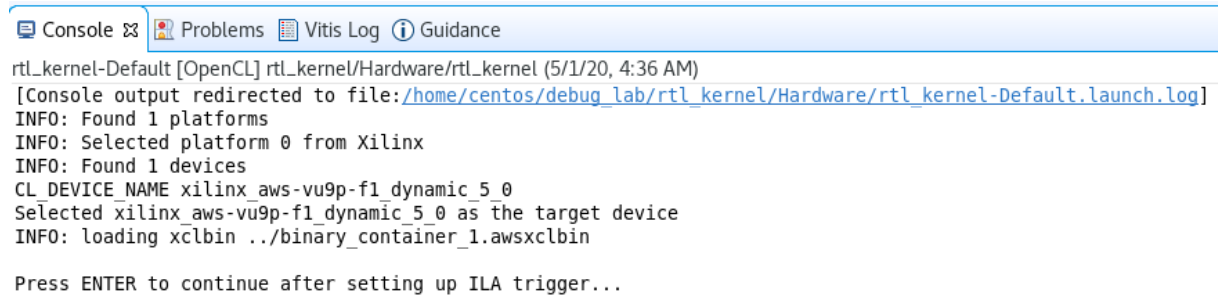
## Hardware Debugging

Review Appendix-I section to understand how to add the ChipScope Debug bridge core and build the project. The debug core has been included in the precompiled sources provided

### Run the application

1. In *Assistant* view, select **Hardware > Run > Run Configurations...**
2. Expand *OpenCL* and select *debug-Default*
3. For Alveo, in the *Arguments* tab make sure **Automatically add binary container(s) to arguments** is selected. For AWS, make sure the `binary_container_1.awsxcclbin` file is listed as an argument
4. Click **Run**

The host application will start executing, load the bitstream, and wait for user input (press any key to continue)



```
Console Problems Vitis Log Guidance
rtl_kernel-Default [OpenCL] rtl_kernel/Hardware/rtl_kernel (5/1/20, 4:36 AM)
[Console output redirected to file:/home/centos/debug_lab/rtl_kernel/Hardware/rtl_kernel-Default.launch.log]
INFO: Found 1 platforms
INFO: Selected platform 0 from Xilinx
INFO: Found 1 devices
CL_DEVICE_NAME xilinx_aws-vu9p-f1_dynamic_5_0
Selected xilinx_aws-vu9p-f1_dynamic_5_0 as the target device
INFO: loading xclbin ../binary_container_1.awsxcclbin

Press ENTER to continue after setting up ILA trigger...
```

## Set up the Xilinx Virtual Cable (XVC)

The Xilinx Virtual Cable (XVC) is a virtual device that gives you JTAG debug capabilities over PCIe to the target device. XVC will be used to debug the design.

### For Alveo U200

For an Alveo board, you need to determine the XVC device in your system. XVC is installed as part of the Vitis and XRT installation.

```
ls /dev/xvc_pub*
```

This will report something similar to the output below:

```
/dev/xvc_pub.u513
```

Each computer may have a different value for `xvc_pub.*` so you will need to check the name for your computer.

- In a terminal window, start a virtual jtag connection

Run the following command (where *u513* should be the value your obtained from the previous command):

```
debug_hw --xvc_pcie /dev/xvc_pub.u513 --hw_server
```

The Virtual JTAG XVC Server will start listening to TCP port **10200** in this case. This is the port you will need to [connect to from Vivado](#). Note the *hw\_server* is listening to TCP port 3121. See example output below

```
launching xvc_pcie...
xvc_pcie -d /dev/xvc_pub.u513 -s TCP::10200
launching hw_server...
hw_server -sTCP::3121

*****
*** Press Ctrl-C to exit ***
*****
```

Skip the next section and continue with [Connecting Vivado to the XVC](#)

## For AWS

Open a new terminal window and run the following script which will manage setup of the XVC:

```
sudo fpga-start-virtual-jtag -P 10200 -S 0
```

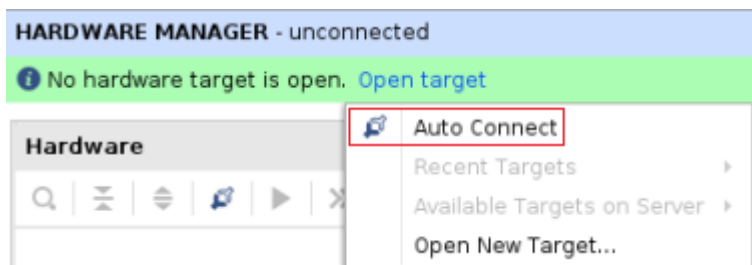
```
[centos@ip-10-243-0-23 ~]$ sudo fpga-start-virtual-jtag -P 10200 -S 0
Starting Virtual JTAG XVC Server for FPGA slot id 0, listening to TCP port 10200.
Press CTRL-C to stop the service.
```

## Connecting Vivado to XVC

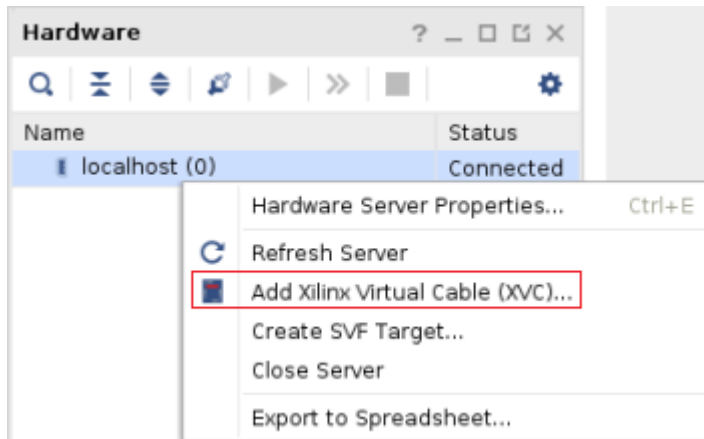
1. Start Vivado from another terminal

```
vivado
```

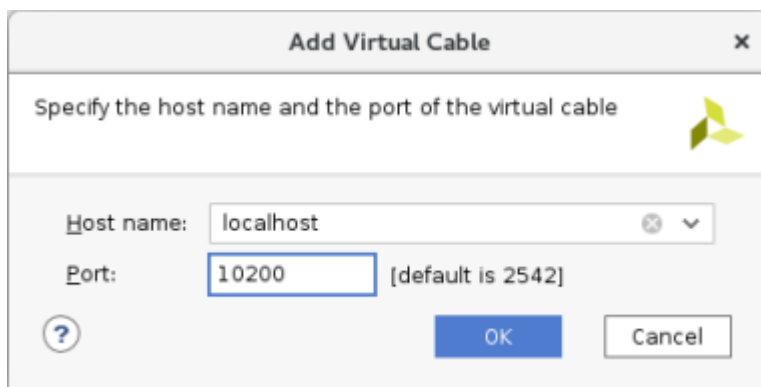
1. Click on **Open Hardware Manager** link
2. Click **Open Target > Auto Connect**



3. Right click on *localhost (0)* and select **Add Xilinx Virtual Cable (XVC)**

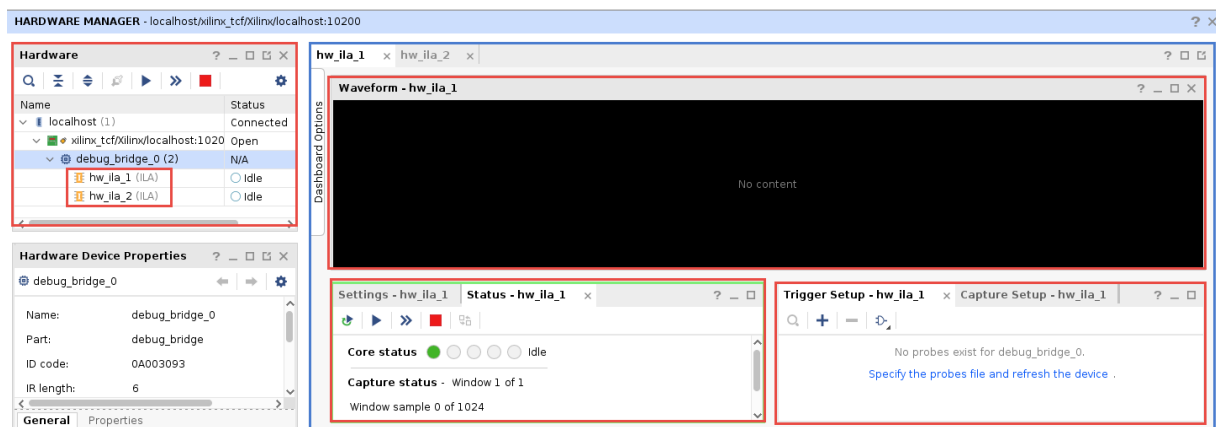


- Enter **localhost** as the *host name*, and **10200** as the port (or the *port number* for your machine obtained previously) and click **OK**



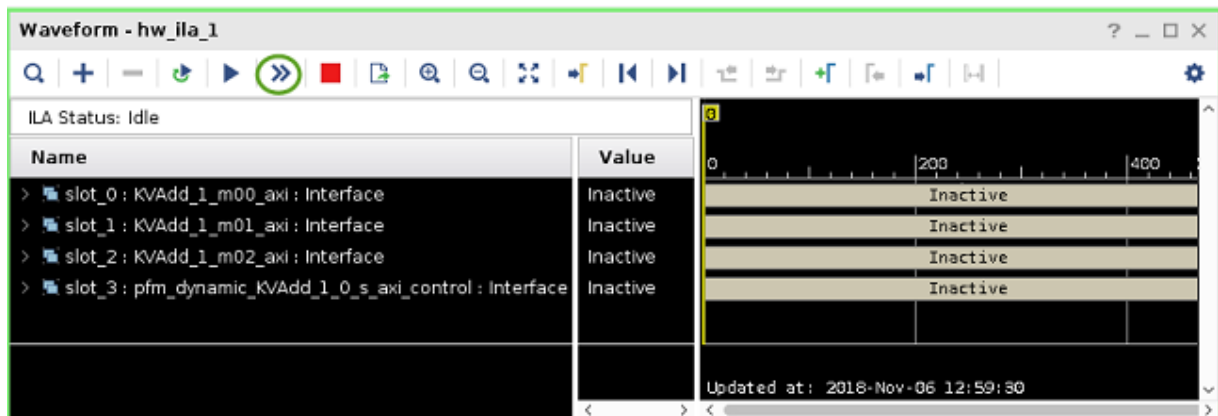
- Right click on the *debug\_bridge* and select **Refresh Device**.

The Vivado Hardware Manager should open showing *Hardware*, *Waveform*, *Settings-hw*, *Trigger-Setup* windows. The *Hardware* window also shows the detected ILA cores (*hw\_ila\_\**), inserted in the design. The Alveo design will have one ILA. The AWS design will have two ILAs, one monitoring the AWS shell interface.

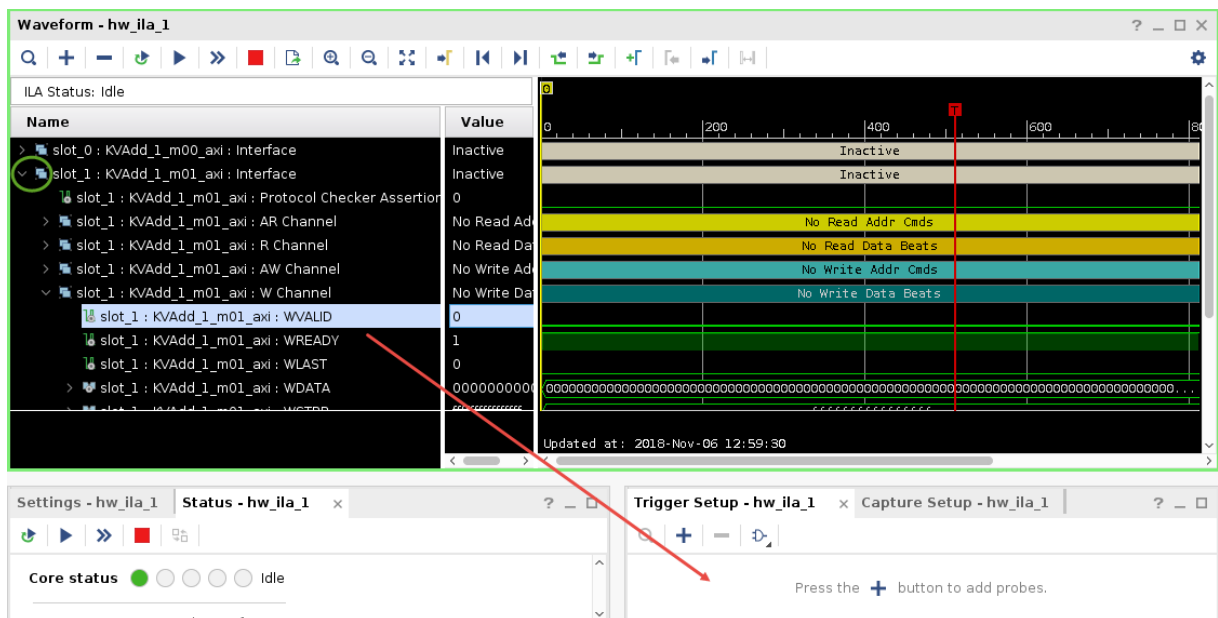


- Select the *debug\_bridge* in the Hardware panel
- In the *Hardware Device Properties* view, click on the browse button beside **Probes file**
- Browse to the project's **~/debug\_lab/rtl\_kernel/Hardware** folder, select the **.ltx** file and click **OK**
- Select the *hw\_ila\_1* tab, and notice four (Slot\_0 to Slot\_3) probes are filled in the Waveform window

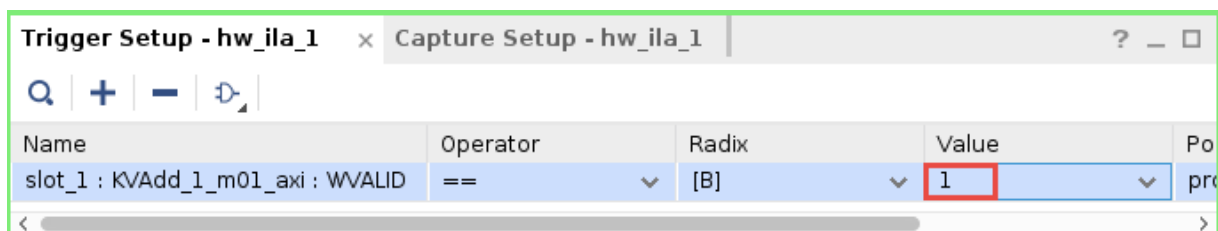
- Click on the **Run Trigger immediate** button and observe the waveform window is fills with data showing that the four channels were *Inactive* for the duration of the signal capture.




- Expand **slot\_1 : KVAdd\_1\_m01\_axi : Interface**, then find and expand **slot\_1 : KVAdd\_1\_m01\_axi : W Channel** in the Waveform window.
- Select the **WVALID** signal and drag it to the Trigger Setup - hw window



- Click on drop-down button of the Value field and select trigger condition value as **1**



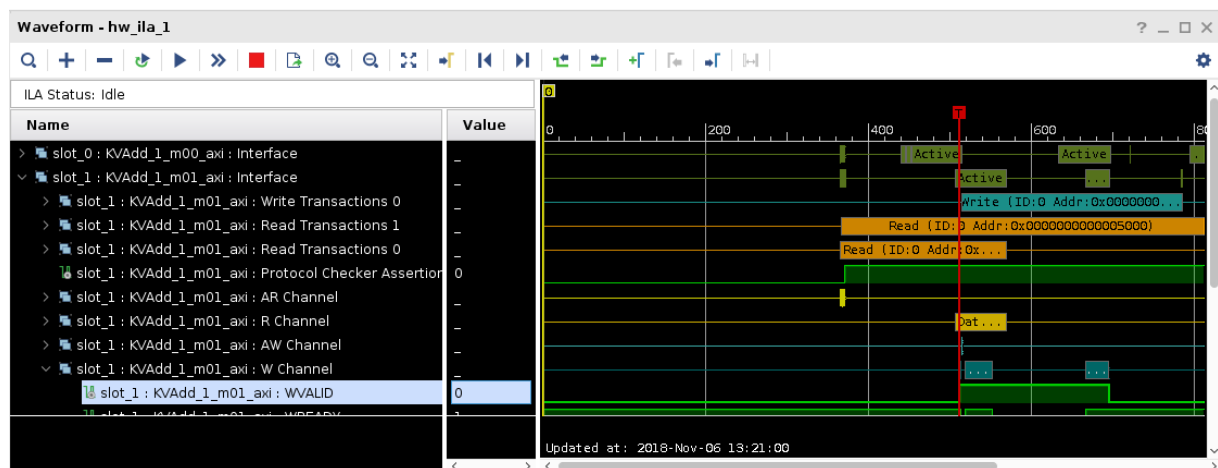
- Click on the **Run trigger** button  and observe the *hw\_ila\_1* probe is waiting for the trigger condition to occur

Hardware		
Name	Status	
localhost (1)	Connected	
xilinx_tcf/Xilinx/localhost:1...	Open	
debug_bridge_0 (2)	Programmed	
hw_ila_1 (WRAPPER_IN...	Waiting For Trigger	

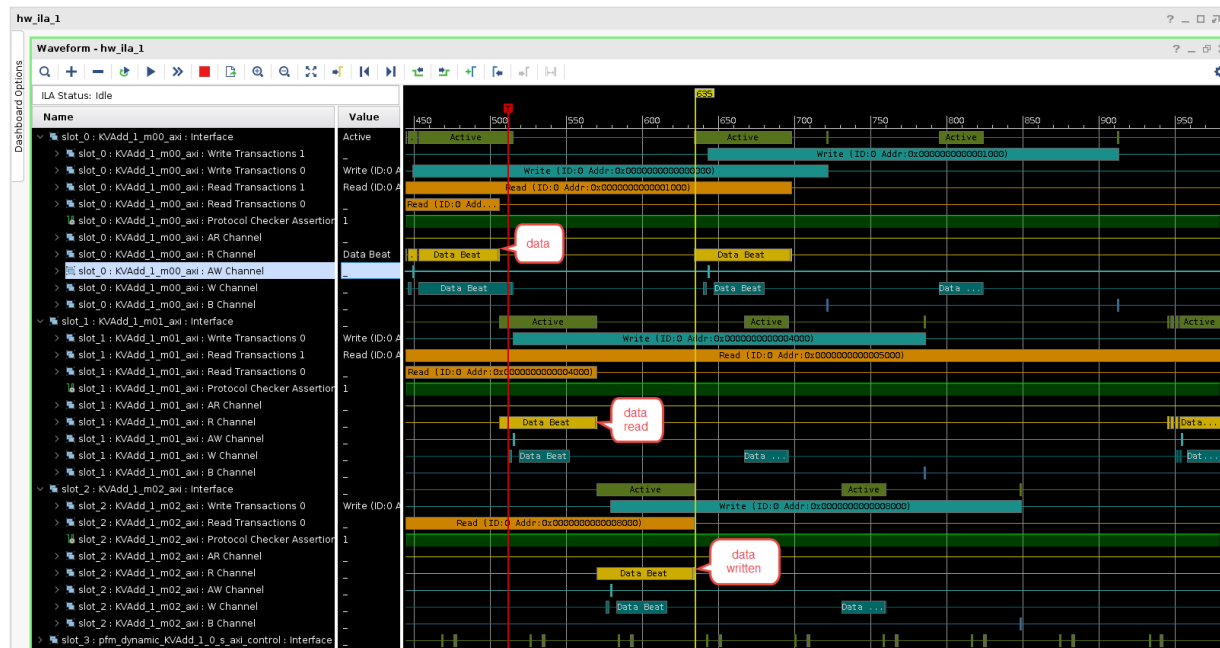
15. Switch to the Vitis IDE window select the *Console* window and press the **Enter** key to allow the program to continue executing

Observe that the program completes displaying **INFO: Test completed successfully** in the Console window

16. Switch back to Vivado and notice that because the trigger condition was met, the waveform window has been populated with new captured data.



17. Expand **Slot\_0**, **slot\_1**, and **slot\_2** groups, zoom in to the region around samples 450 to 1000, and observe the data transfers taking place on each channels. Also note the addresses from where data are read and where the results are written to.



18. Zoom in on one of the transactions and hover your mouse at each successive sample and notice the data content changing

19. When you are finished, close Hardware Manager by selecting **File > Close Hardware Manager**

20. Click **OK** and then close Vivado by selecting **File > Exit**

21. Close the jtag probe by switching to its terminal window and pressing **Ctrl-C**

## Perform Software Debugging

1. Switch to the Vitis GUI

2. In the *Assistant* view, right click on Hardware and select **Debug > Debug Configurations...**

3. Make sure that the **Arguments** tab shows **../binary\_container\_1.awsxcclbin**

4. Click **Apply** if needed, and then click **Debug**

5. Click **Yes** when prompted to switch to the *Debug perspective*

The bitstream will be downloaded to the FPGA and the host application will start executing, halting at **main()** entry point

6. In *host\_example.cpp* view scroll down to line ~272 and double-click on the left border to set a breakpoint At this point, three buffers would have been created



```

251 mem_ext.flags = 1;
252 d A = clCreateBuffer(context, CL_MEM_READ_WRITE | CL_MEM_EXT_PTR_XILINX, sizeof(cl_uint) * number_of_words, &mem_
253 if (err != CL_SUCCESS) {
254     std::cout << "Return code for clCreateBuffer flags=" << mem_ext.flags << ": " << err << std::endl;
255 }
256
257
258 mem_ext.flags = 2;
259 d B = clCreateBuffer(context, CL_MEM_READ_WRITE | CL_MEM_EXT_PTR_XILINX, sizeof(cl_uint) * number_of_words, &mem_
260 if (err != CL_SUCCESS) {
261     std::cout << "Return code for clCreateBuffer flags=" << mem_ext.flags << ": " << err << std::endl;
262 }
263
264
265 mem_ext.flags = 3;
266 d Res = clCreateBuffer(context, CL_MEM_READ_WRITE | CL_MEM_EXT_PTR_XILINX, sizeof(cl_uint) * number_of_words, &me
267 if (err != CL_SUCCESS) {
268     std::cout << "Return code for clCreateBuffer flags=" << mem_ext.flags << ": " << err << std::endl;
269 }
270
271
272 if (!(d_A&&d_B&&d_Res)) {
273     printf("Error: Failed to allocate device memory!\n");
274     printf("Test failed\n");
275     return EXIT_FAILURE;
276 }

```

7. Click on the **Resume** button or press **F8**

8. When prompted click in the console and press *Enter*

The program will resume executing and stop when it reaches the breakpoint.

At this point you can click on the various monitoring tabs (*Variables*, *Command Queue*, *Memory Buffers* etc.) and see the contents currently in scope.

Vitis debug allows command queues and memory buffers to be examined as the program execution progresses

9. Click back to select *Debug.exe* > *#Thread 1* in the Debug panel

10. Click on the **Step Over** button or press **F6**

The execution will progress one statement at a time

11. Continue pressing **F6** until you reach line ~332 at which point kernel will finish executing

12. Select the **Memory Buffers** tab

Notice that three buffers are allocated, their IDs, DDR memory address, and sizes

Memory Command Queue Memory Buffers Platform Debug	
Name	Value
▼ Mem-0x6799f0	
Mem	0x6799f0
MemID	1
Device Memory Address	0x800004000
Bank	bank0
Size	16384
HostAddress	0
▼ Mem-0x679bc0	
Mem	0x679bc0
MemID	2
Device Memory Address	0x800008000
Bank	bank0
Size	16384
HostAddress	0
▼ Mem-0x6826a0	
Mem	0x6826a0
MemID	0
Device Memory Address	0x800000000
Bank	bank0
Size	16384
HostAddress	0

13. Select the **Command Queue** tab and notice that there no commands enqueued.

Memory Command Q... Platform Debug	
Name	
▼ Queue-0x676b10	
Queued(0)	
Submitted(0)	

Lines ~340-344 creates commands to read the data and results

```
err |= clEnqueueReadBuffer( ... );
```

1. Press **F6** to execute the first `clEnqueueReadBuffer()` to create a read buffer command for reading operand `d_A`  
Notice the Command Queue tab shows one command submitted

Memory Command Queue Memory Buffers Platform De...	
Name	Value
Queue-0x676b10	
Queued(0)	
Submitted(1)	
Event-0x69d160	
name	Event-0x69d160
Uid	5
Status	Complete
Type	CL_COMMAND_READ_BUFFER
WaitingOn	None
Description	Transfer 16384 bytes from cl_

Line 340  
executed

2. Press **F6** to execute the next `clEnqueueReadBuffer()` for `d_B`

Notice the Command Queue tab shows two commands submitted

Memory Command Queue Memory Buffers Platform Debug	
Name	Value
Queue-0x676b10	
Queued(0)	
Submitted(2)	
Event-0x69d160	
name	Event-0x69d160
Uid	5
Status	Complete
Type	CL_COMMAND_READ_BUFFER
WaitingOn	None
Description	Transfer 16384 bytes from cl_mem 0x6826a0+0
Event-0x69e420	
name	Event-0x69e420
Uid	6
Status	Complete
Type	CL_COMMAND_READ_BUFFER
WaitingOn	None
Description	Transfer 16384 bytes from cl_mem 0x6799f0+0

Line 342 executed

3. Set a breakpoint at line ~398 ( `clReleaseKernel()` ) and press **F8** to resume the execution

Notice that the Command Queue tab still shows entries

4. Press **F6** to execute `clReleaseKernel()`

Notice the Memory Buffers tab is empty as all memories are released

5. Click **F8** to complete the execution

6. Close the Vitis program

## Conclusion

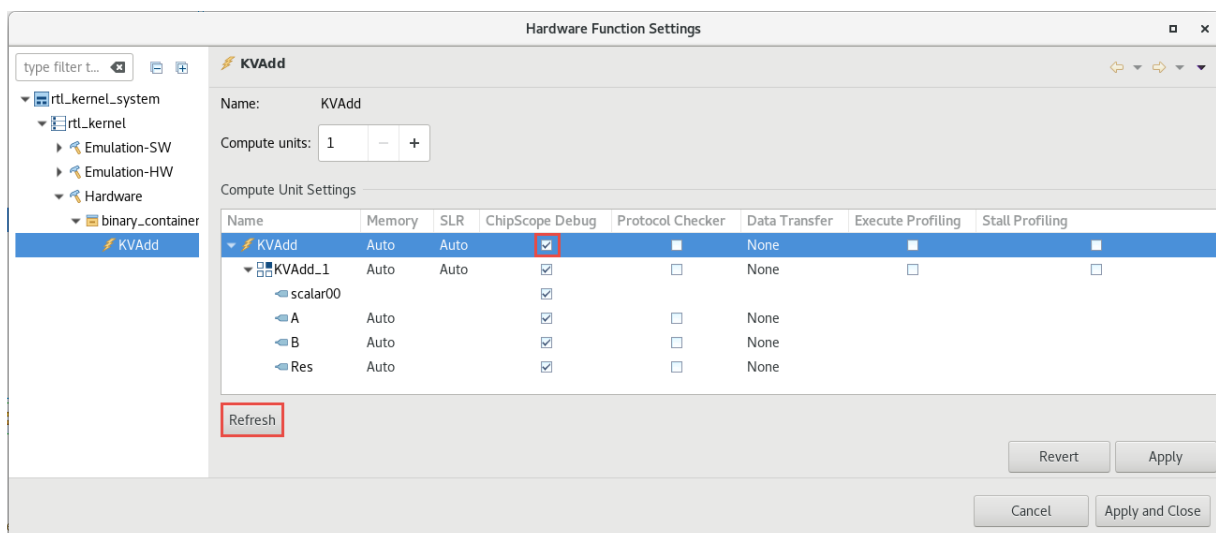
In this lab, you used the ChipScope Debug bridge and cores to perform hardware debugging. You also performed software debugging using the Vitis GUI.

Start the next lab: [Vision Lab](#)

## Appendix-I

### Steps to Add ChipScope Debug core and build the design

1. In the *Assistant* view, expand **System > binary\_container\_1 > KVAdd**
2. Select **KVAdd**, right-click and select **Settings...**
3. In the **Hardware Function Settings** window, click **Refresh**, and then click on the *ChipScope Debug* option for the *KVAdd* kernel



4. Click **Apply and close**

5. In the **Project** tab, expand **rtl\_kernel > src > vitis\_rtl\_kernel > KVAdd** and double-click on the **host\_example.cpp** to open it in the editor window

6. Around line 243 (after the `clCreateKernel` section) enter the following lines of code and save the file. This will pause the host software execution after creating kernel but before allocating buffer

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

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

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

```
// Create structs to define memory bank mapping  
cl_mem_ext_ptr_t d_bank_ext[4];
```

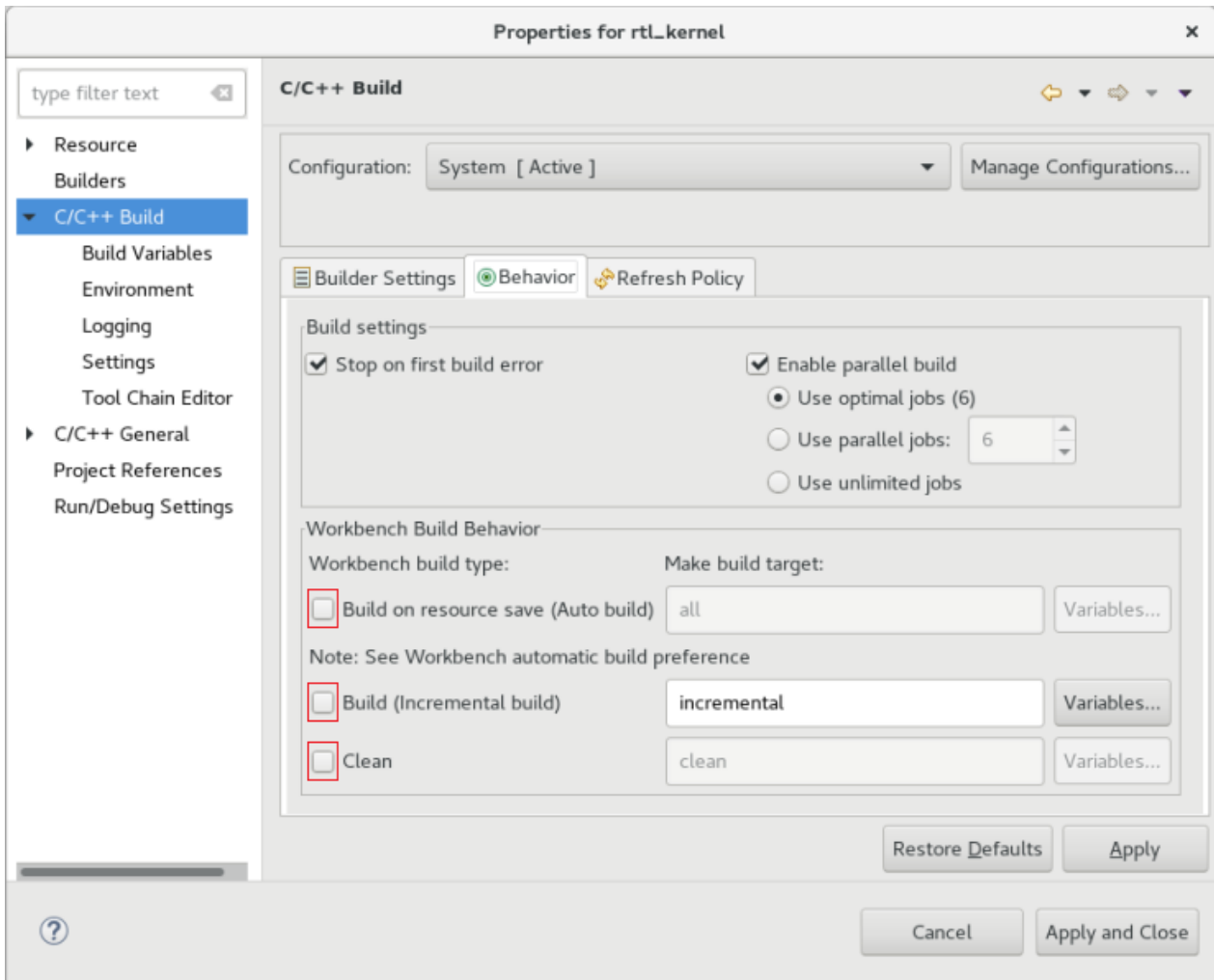
1. Build the design

## Disable automatic rebuilding of the design

When you export a project, and re-import it, the file modified dates may change and cause Vitis to make the output executable and hardware kernel "out-of-date". This may cause the design to be automatically recompiled when an attempt is made to run the application from the GUI.

- To disable automatic rebuilding, right click on the project folder, and select **C/C++ Build Settings**
- Select **C/C++ Build** and click on the **Behavior** tab
- Uncheck the following:
  - Build on resource save (Auto Build)
  - Build (Incremental build)
  - Clean

When you export a project, and re-import it, these settings stop the bitstream being automatically rebuilt.



If you need to rebuild the project, you can re-enable these settings. If you only need to update the host application, you can run the following command in a terminal in the project folder to rebuild the executable file only (where *debug.exe* is the name of the executable):

```
cd ./workspace/rtl_kernel/Hardware
make rtl_kernel
```

## References

[Vitis Debugging Applications and Kernels](#)