Debugging on Alveo U280s in OCT

This tutorial will show you how to debug FPGA applications on Alveo U280s in the Open Cloud Testbed (OCT).

Why Debugging

- **Find and Fix Issues**: Debugging allows you to detect and fix errors in your FPGA design, such as logic flaws or timing violations to ensure that the system behaves as expected.
- **Improve Performance**: By analyzing real-time behavior, you can refine the design to improve efficiency, speed, and resource utilization.
- **Save Time**: Spotting and addressing issues before the production stage is faster and more efficient than addressing them after deployment.

What We Use

In OCT, we use the **Xilinx Virtual Cable (XVC)** for FPGA debugging. This method involves the following components:

- **XVC Server**: A lightweight server application running on the OCT node. It enables remote JTAG communication over TCP/IP or PCIe.
- **Vivado Hardware Manager**: A debugging tool provided by AMD Xilinx for interacting with the FPGA. It connects to the XVC server to send and receive JTAG commands.
- **FPGA Target**: The hardware device (Alveo U280 in our case) that requires debugging.
- **PCIe Interface**: Facilitates communication between the XVC server and the debugger.

Why XVC

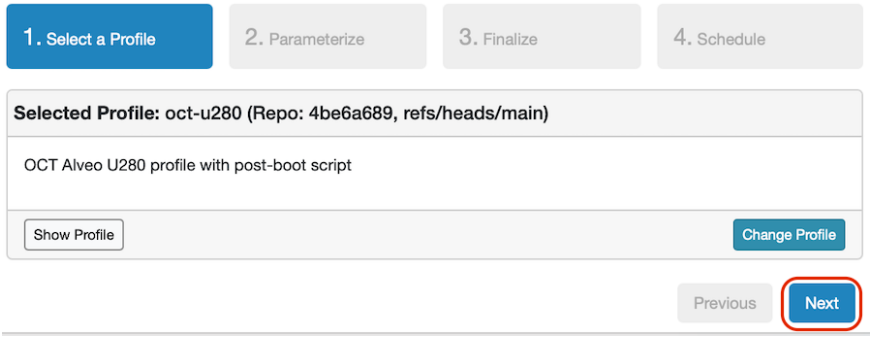
- **No Direct JTAG Access:** OCT users do not have direct JTAG access to FPGA boards in the testbed. JTAG debugging through traditional means (e.g., USB cables) is not possible.
- **Remote Debugging:** XVC allows users to debug their FPGAs remotely, making it practical for cloud-based setups where the hardware is located in data centers.

Note: *In this tutorial, we use local debugging rather than remote debugging. Local debugging means that the debugger will run directly on the same OCT node where the FPGA is physically connected, rather than using a remote machine to control the debugging process. The reason we use local debugging is that we are working with a host executable and need to step through its execution. This requires access to the Xilinx Runtime (XRT), which is required to interact with the FPGA and is expected to be installed on the same machine as the FPGA. While remote debugging with Vivado is possible, when using a remote node for Vivado, the host code must still be executed on the OCT node where the FPGA is connected.*

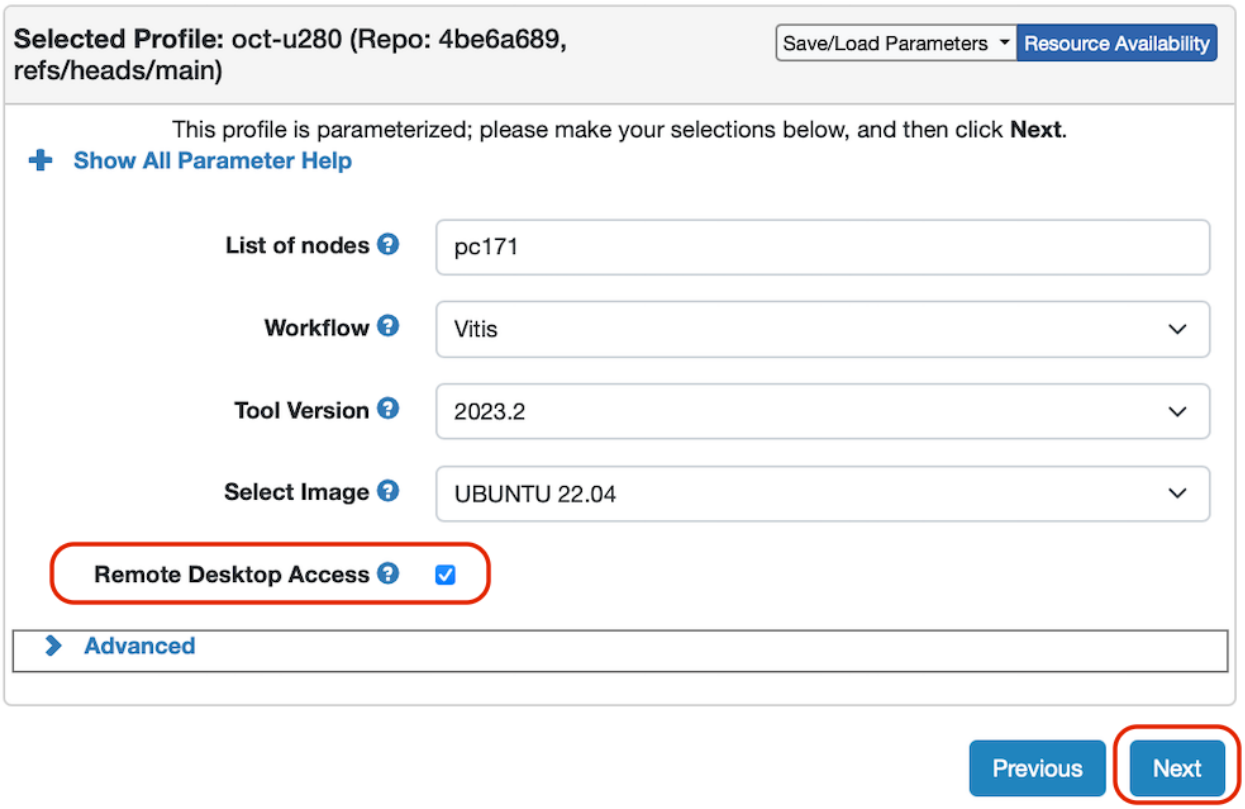
The tutorial has two main parts: **1. Setting up your OCT experiment in CloudLab,** and **2. Building and debugging the application**. In the first part, you will create an experiment in CloudLab to set up an OCT node for debugging. Once the node is ready, you will connect to the node through VNC to access its GUI. From there, you will use the Vitis IDE and create a project using the vector addition example (vadd) in the installed examples repository. You will also add debug capabilities into your design by adding ChipScope ILA (Integrated Logic Analyzer) cores. These cores are used to capture internal signals, helping you see what's happening inside the FPGA during operation. You will build an FPGA bitstream and a host executable. After loading the bitstream you will step through the host code, launch the vadd kernel, and observe the FPGA signals.

Part 1: Experiment Setup

Under project profiles, click the profile oct-u280.



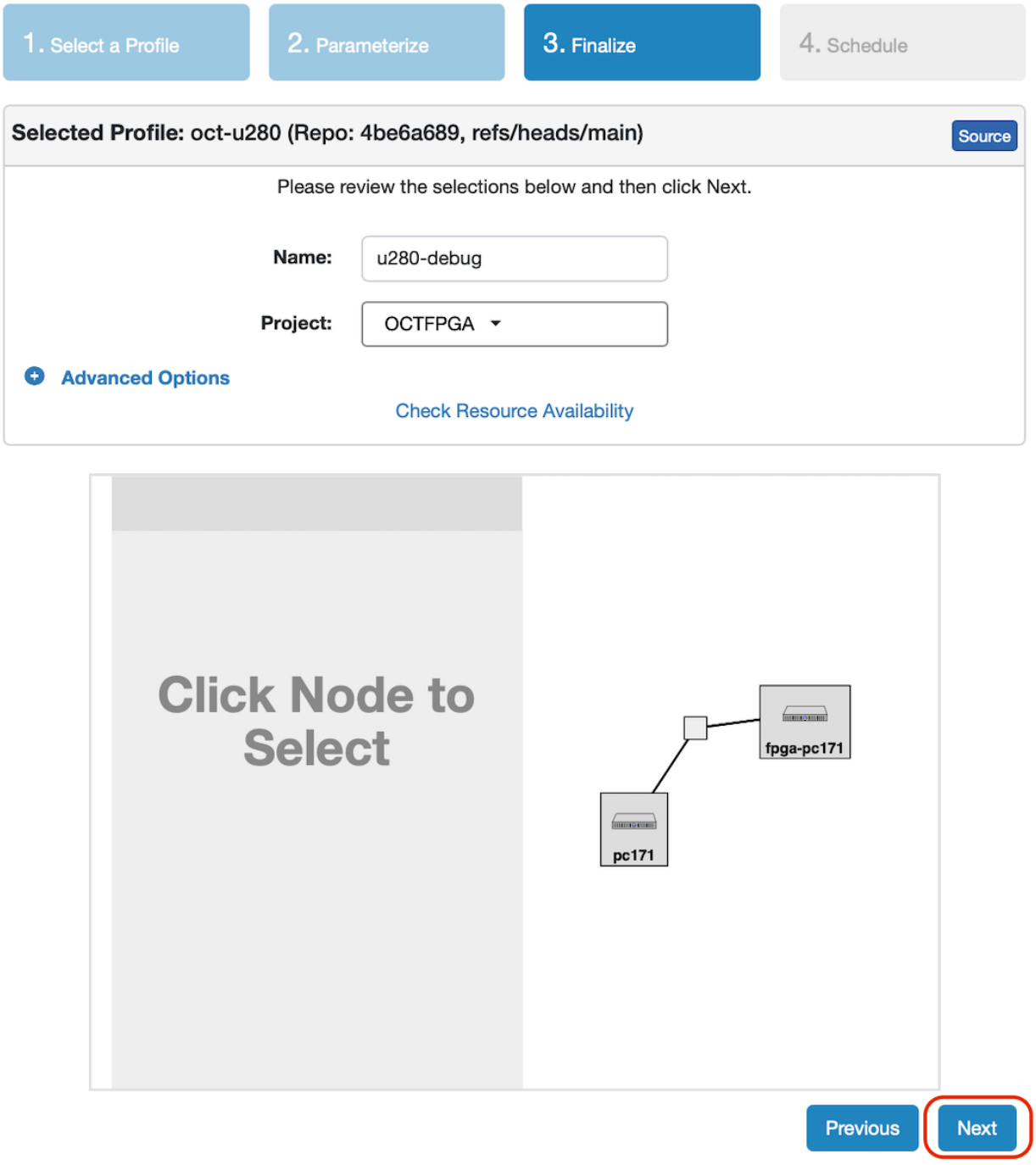
Select the parameters as shown in the screenshot below. You also need to choose the **Remote Desktop Access** option to use the VNC server for this experiment.



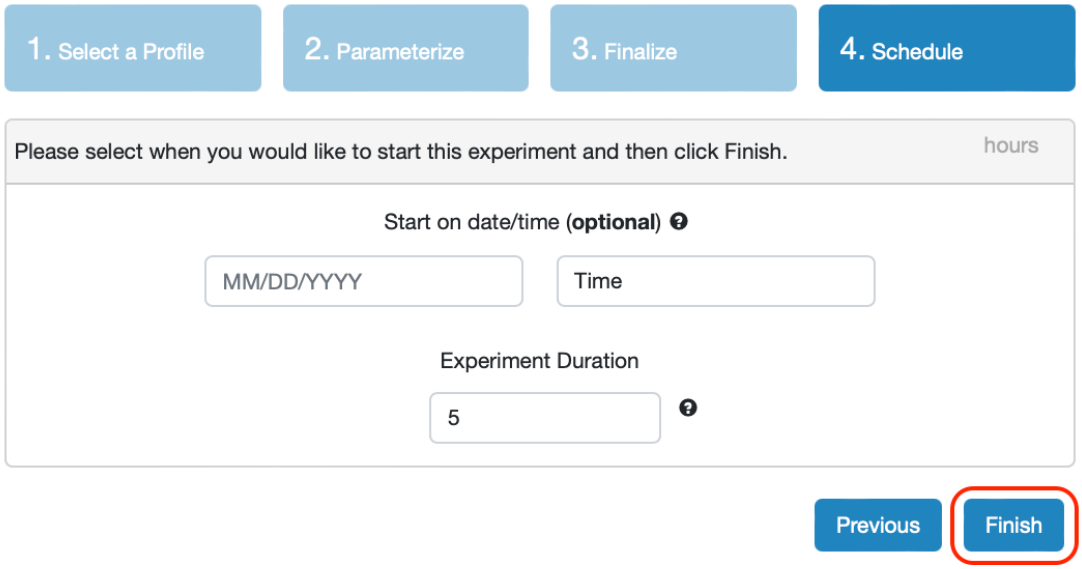
For this experiment, you can use any available OCT node. You can check node availability at the following link:

https://www.cloudlab.us/cluster-status.php

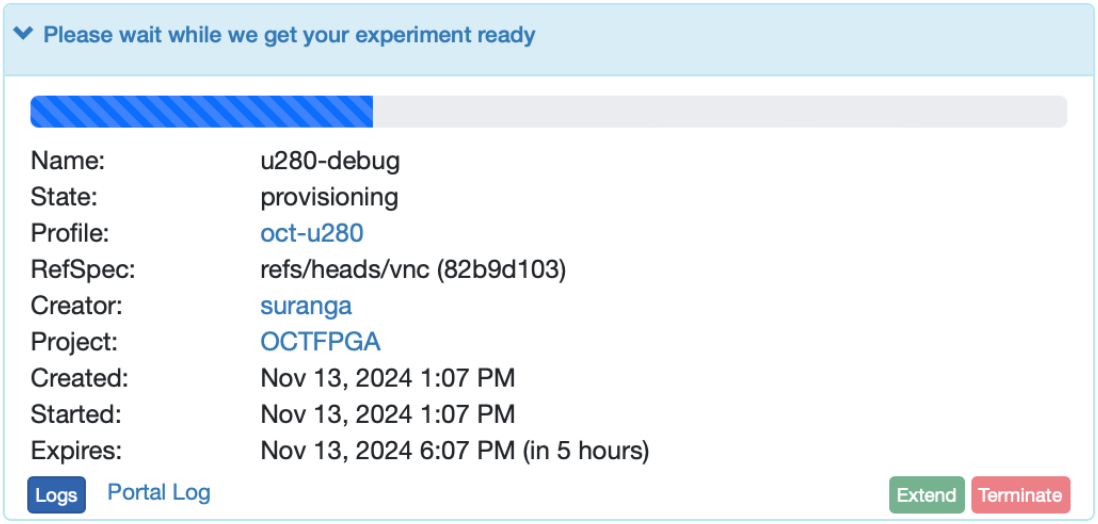
Enter a name for the experiment, and click **Next**.

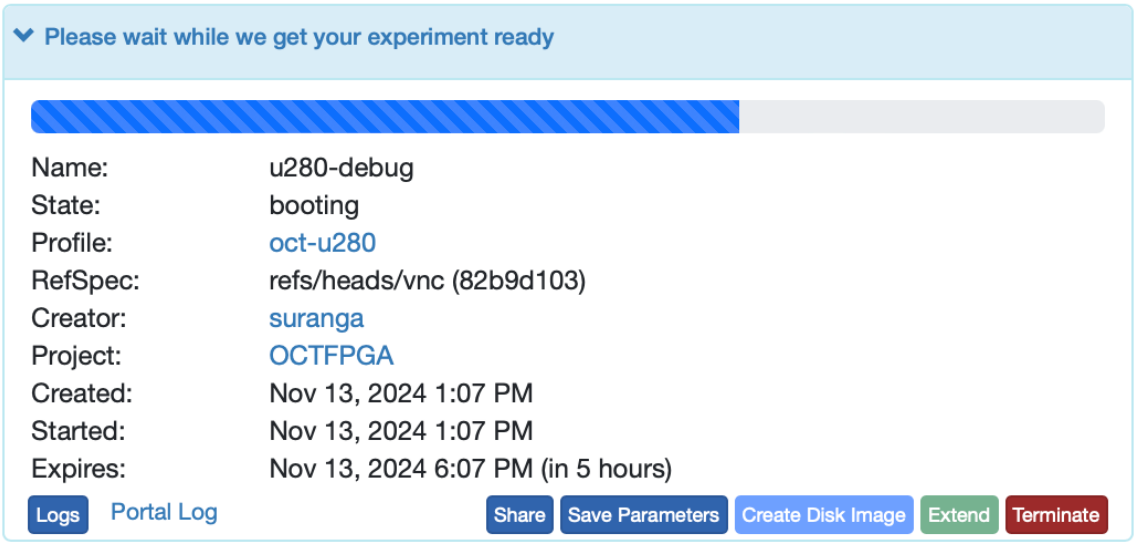


Set the experiment duration and click **Finish**.

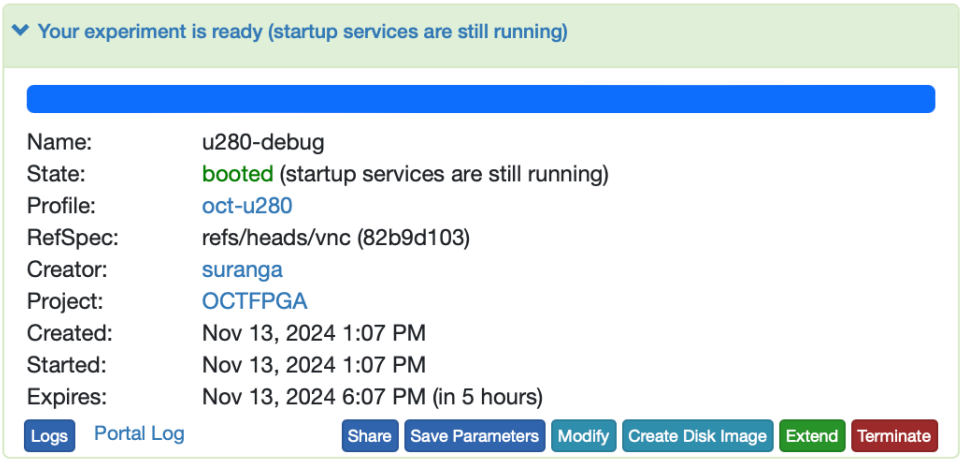


The experiment will start now.

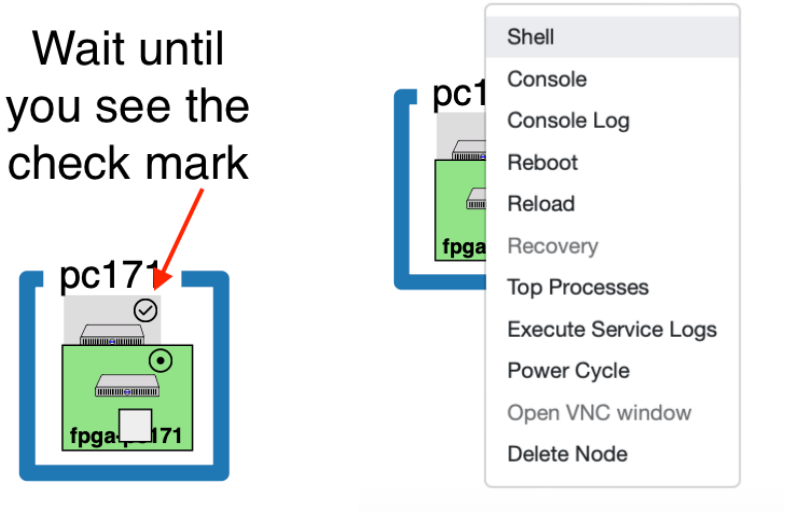




Once the node has booted up, startup services start running. These include installing Xilinx runtime tools, graphical desktop tools, VNC, and more.



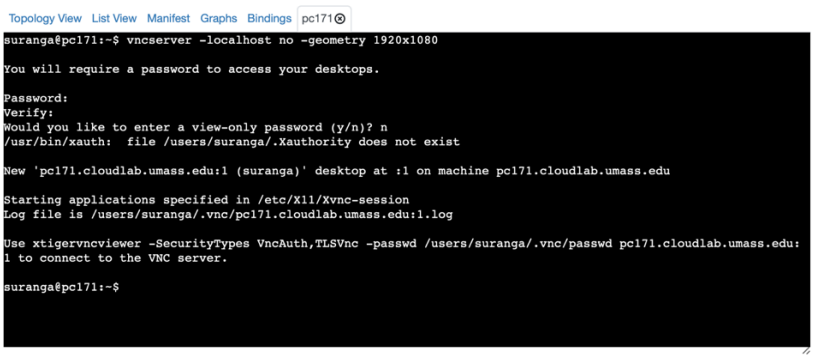
After the startup services finish running (indicated by the checkmark on the node icon), open the shell or log into the node via SSH.



In this tutorial, you will use Vitis IDE, which requires a graphical desktop. First, we will start a VNC server on the OCT node.

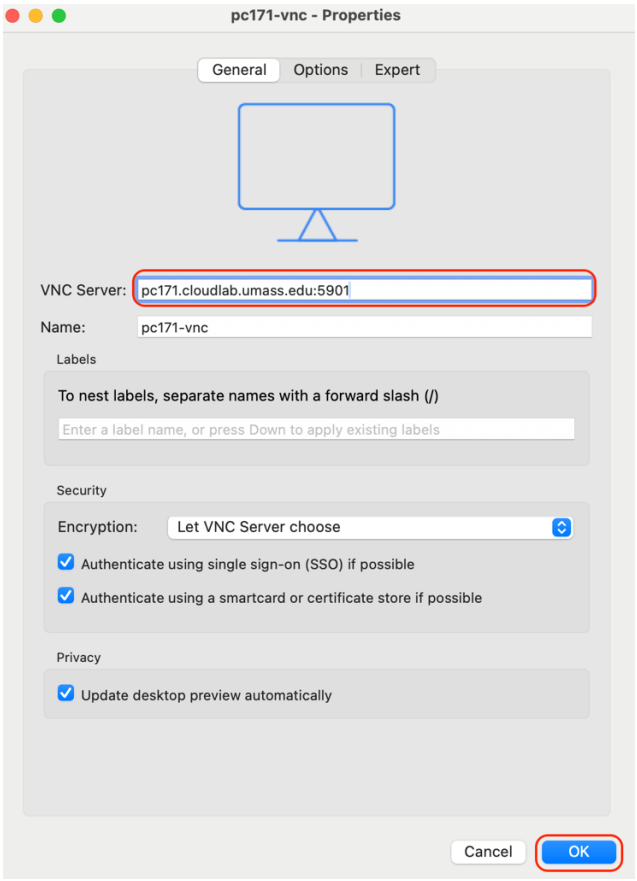
Run the following command on the OCT node:

```
vncserver -localhost no -geometry 1920x1080
```

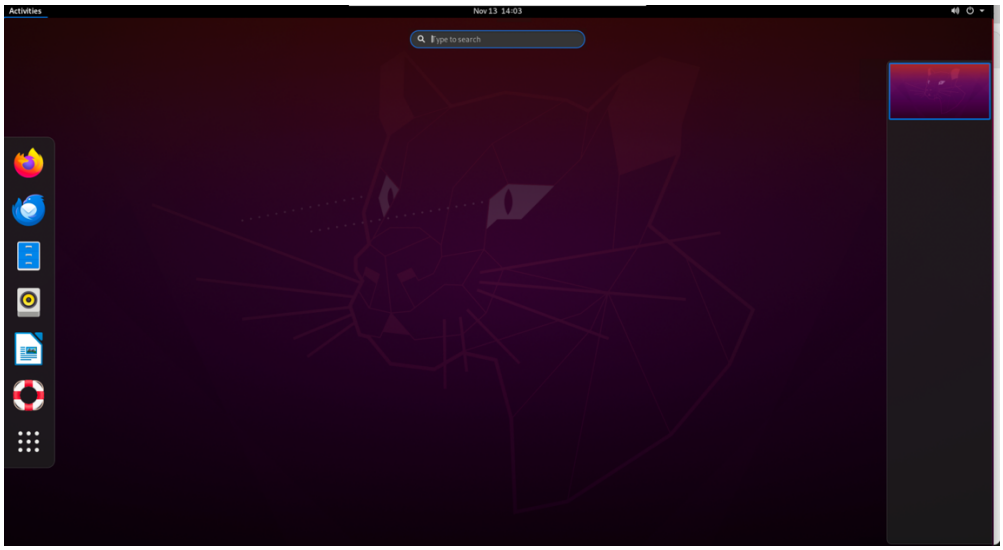


Use any VNC client to access the GUI of the OCT node. Here we are using Real VNC which you can download at https://www.realvnc.com/en/connect/download/viewer/

Since we didn't specify a VNC port number, the default is 5901. Use this port number in the client. For example, if the node assigned to you is pc171, set the VNC server to `pc171.cloudlab.umass.edu:5901`.



After setting up the VNC client, click **OK**. Then, access the graphical desktop by double-clicking the node's icon.



Part 2: Building and Debugging the Vector Addition Example

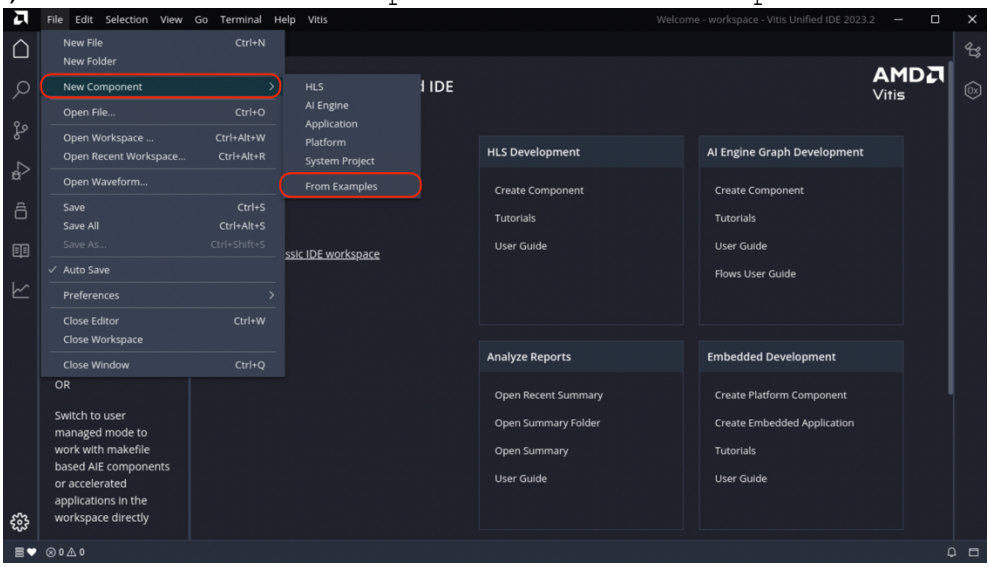
After you logged into VNC desktop, open a terminal. Make sure you are using **Vitis 2023.2** by running `which vitis`.

```
suranga@pc171:~$ which vitis
/share/Xilinx/Vitis/2023.2/bin/vitis
```

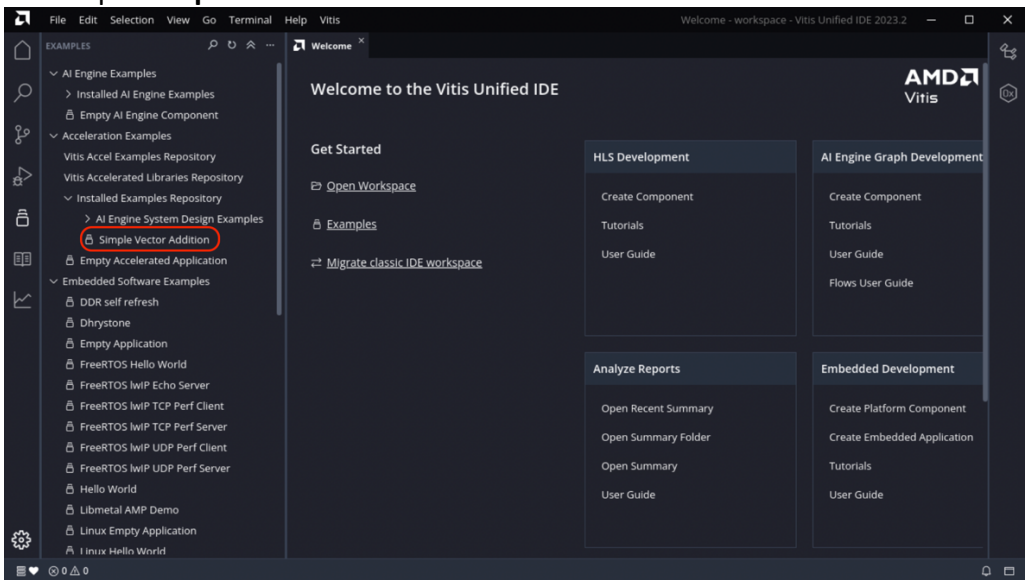
Launch Vitis with the following command:

```
vitis -w ~/workspace
```

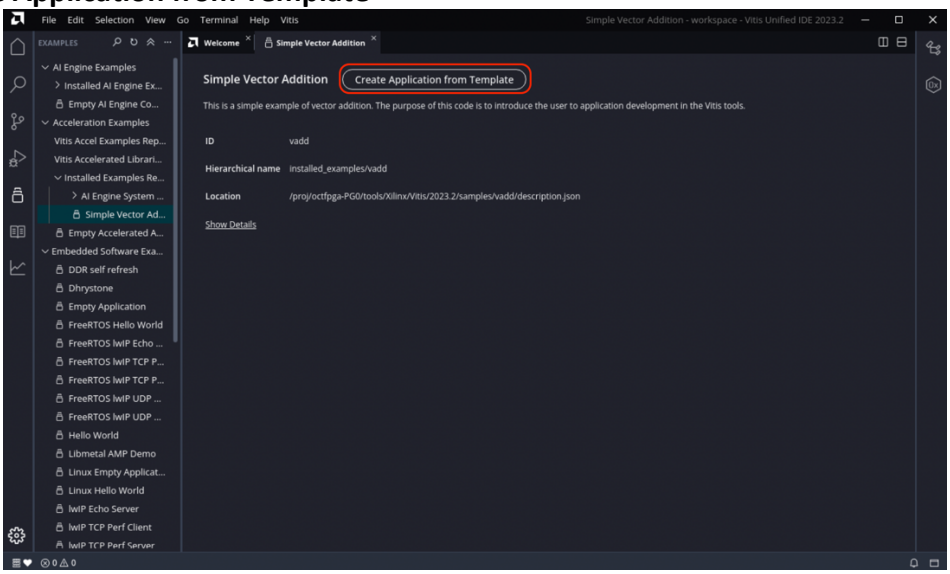
In Vitis IDE, click `File -> New Component -> From Examples`



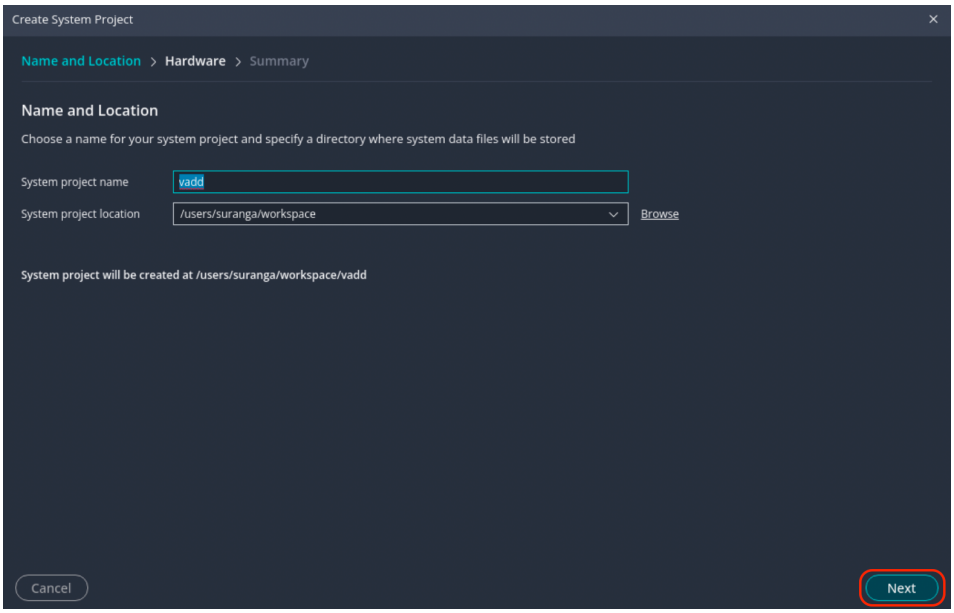
Select the example **Simple Vector Addition**.



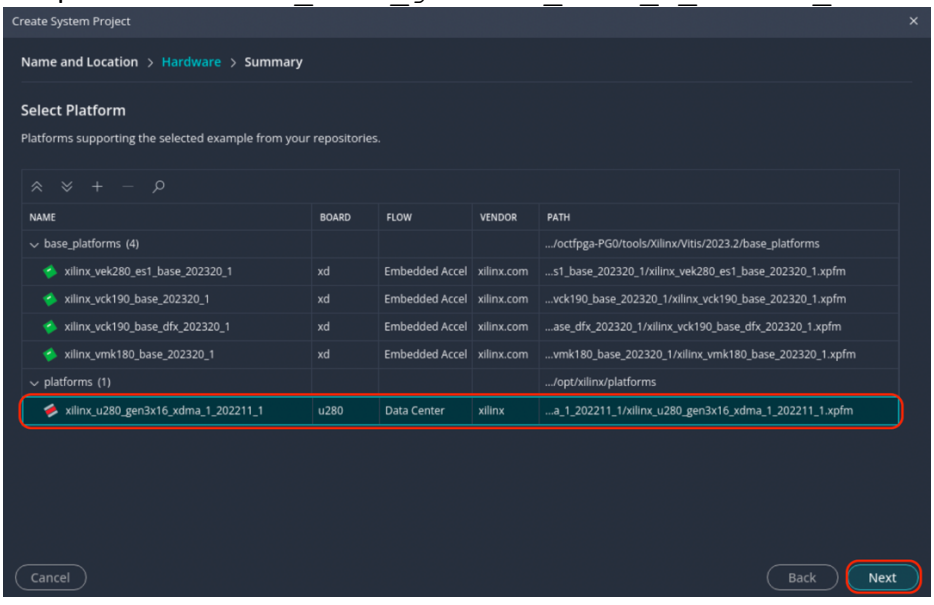
Click **Create Application from Template**



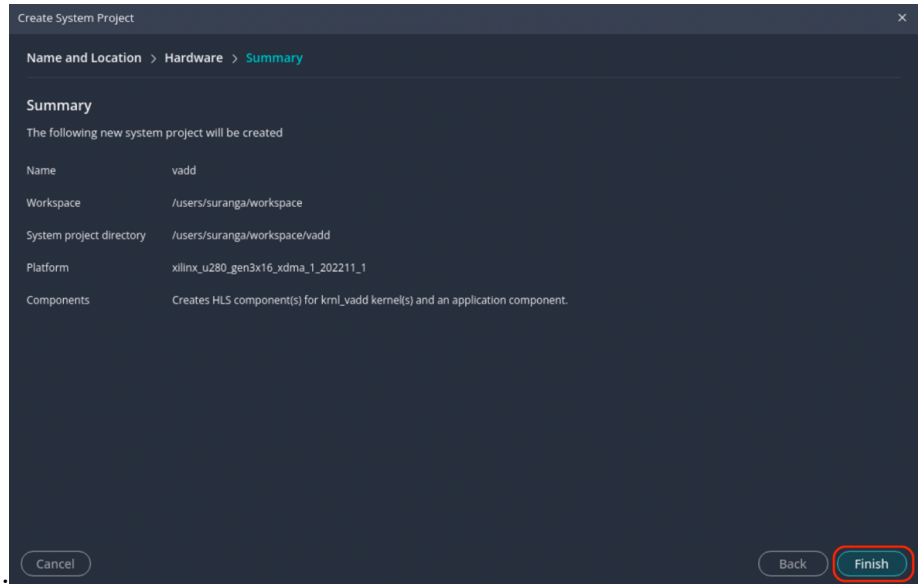
Click **Next**.



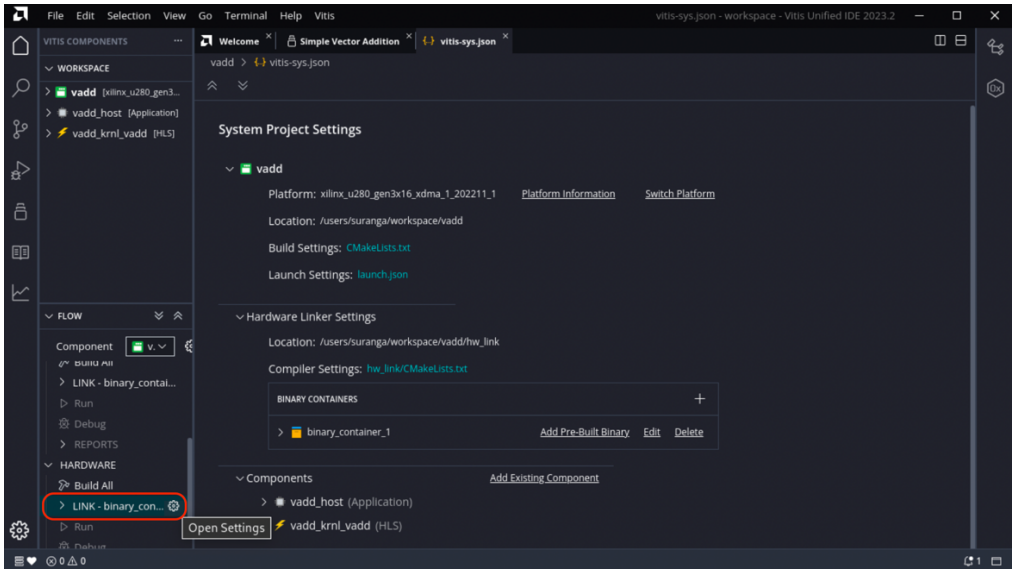
Then, select the platform `xilinx_u280_gen3x16_xdma_1_202211_1` and click **Next**.



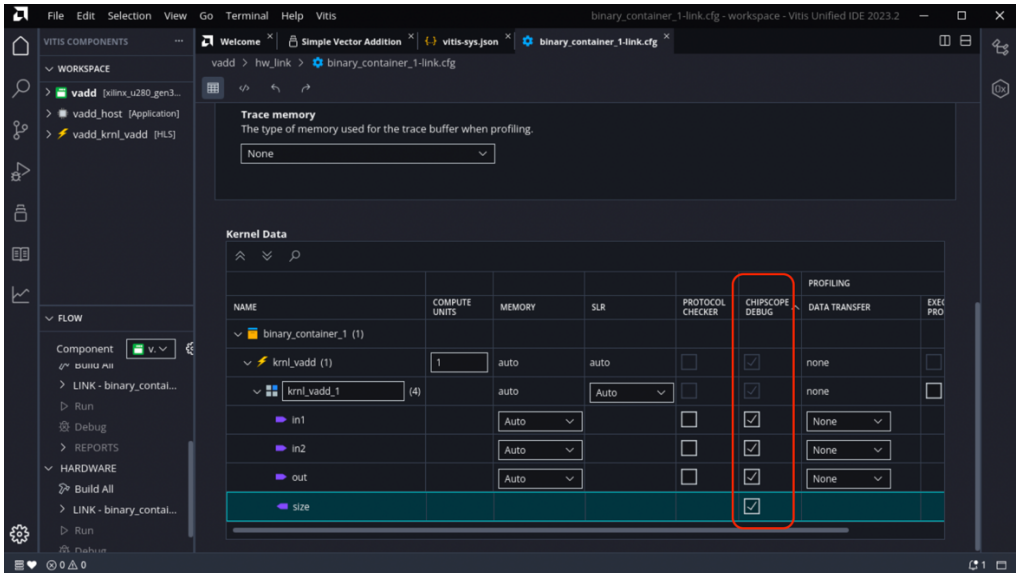
Click **Finish**.



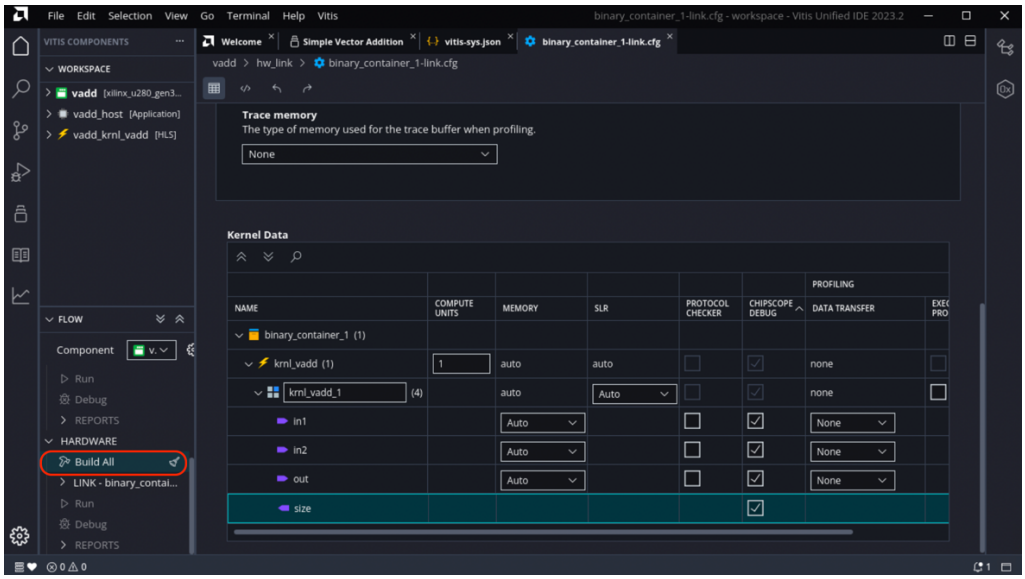
After the project is open, in **Flow** -> **Hardware**, open the settings under `LINK-binary_container_1` (click on the gear icon ⚙).



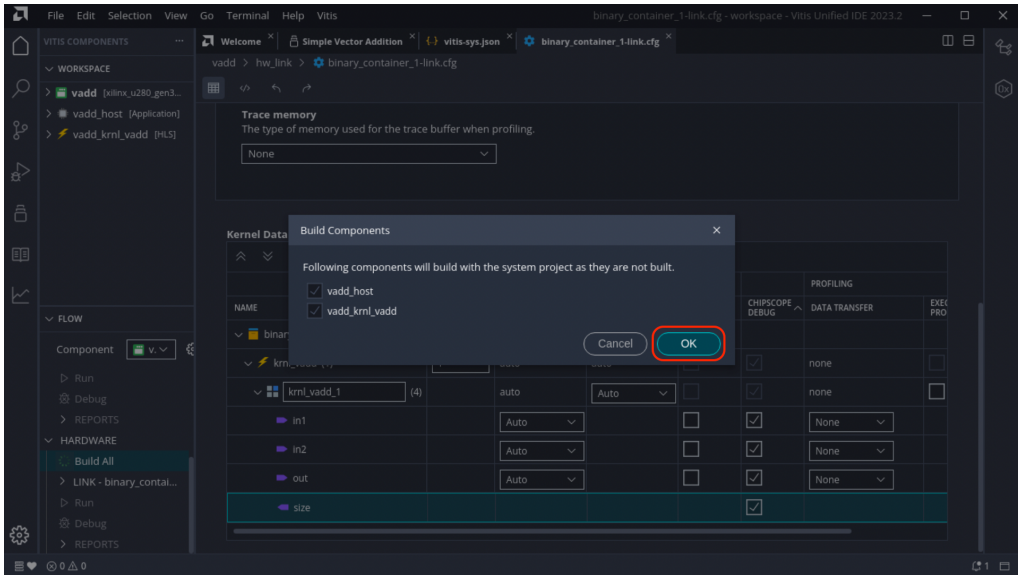
Here, you have the option to select the specific ports you want to debug within your design. In this case, we have chosen **in1**, **in2**, **out**, and **size**. These selected ports will be connected to a **ChipScope debug core** in Vivado. The ChipScope core is used for monitoring and analyzing the internal signals of your FPGA design in real-time.



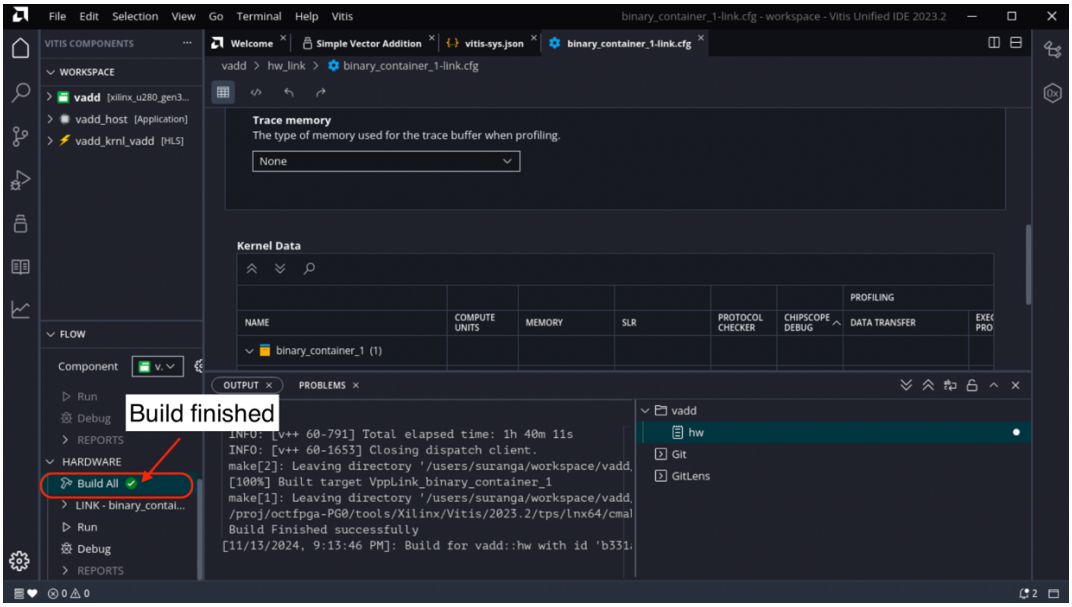
After doing this, you can build the project. Click **Build All**.



Here, we are going to build both the bitstream and the host executable. Click **OK.**



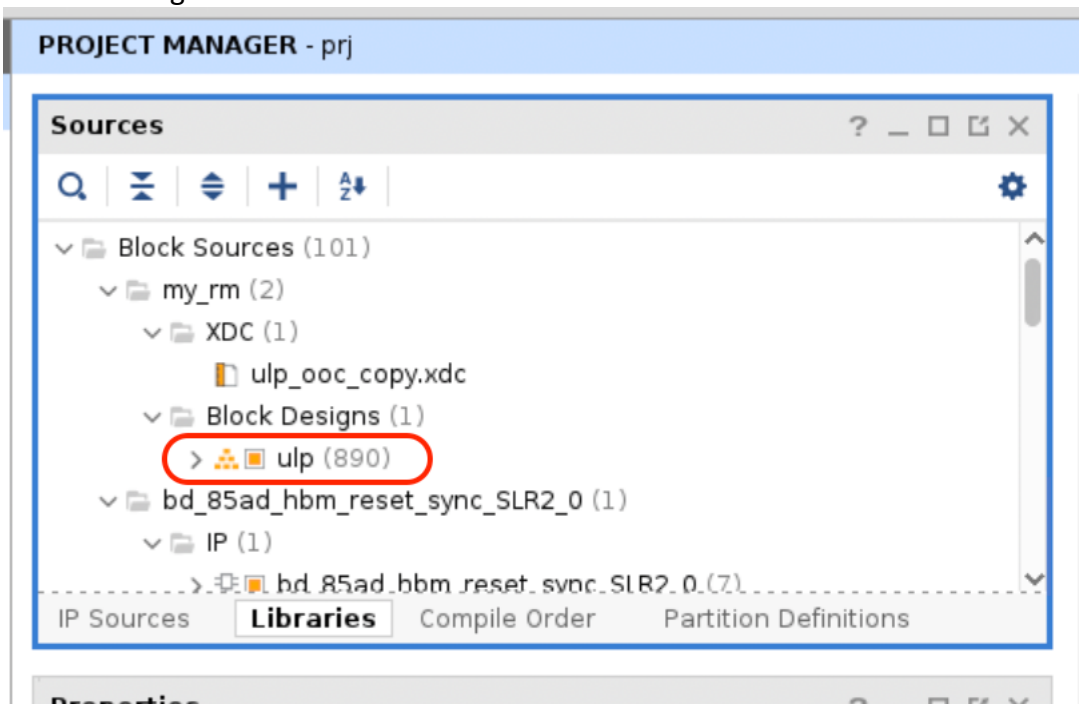
Once the project has finished building, you will see a green checkmark, as shown.



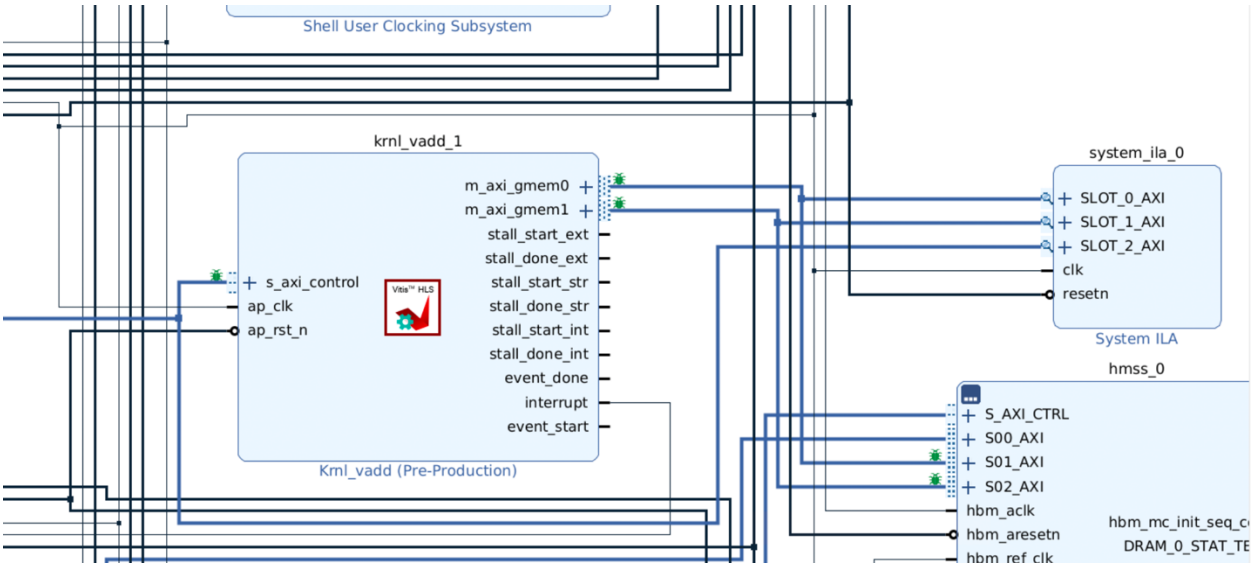
Launch the Vivado project by entering the command below in a separate terminal window.

```
vivado
~/workspace/vadd/build/hw/hw_link/binary_container_1/binary_cont
ainer_1/vivado/vpl/prj/prj.xpr
```

Once the Vivado project is open, in the Project Manager, under Sources, double-click on **ulp** to open the block design.



Note that three ports of the `vadd` kernel have been connected to the ILA for debugging.

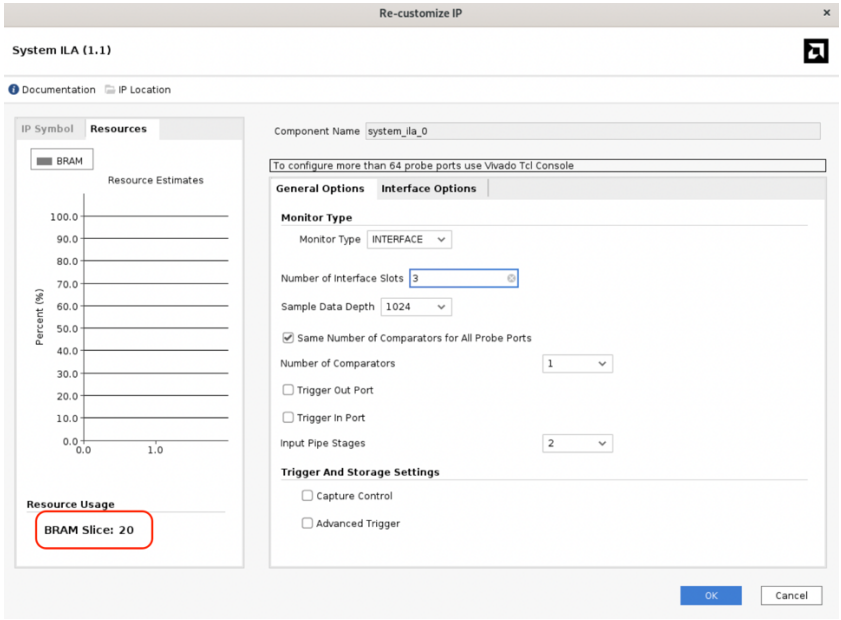


In the kernel, `s_axi_control` is the AXI-Lite control interface that allows the processor to manage the behavior of the kernel by writing control registers, reading status registers, etc.

`m_axi_gmem0` and `m_axi_gmem1` are the AXI memory-mapped ports used for transferring data between the kernel and host memory. In the kernel code found in `~/workspace/vadd_krnl_vadd/krnl_vadd.cpp`, you'll see that the two inputs to the kernel, `in1` and `in2`, are read from `gmem0` and `gmem1`, respectively.

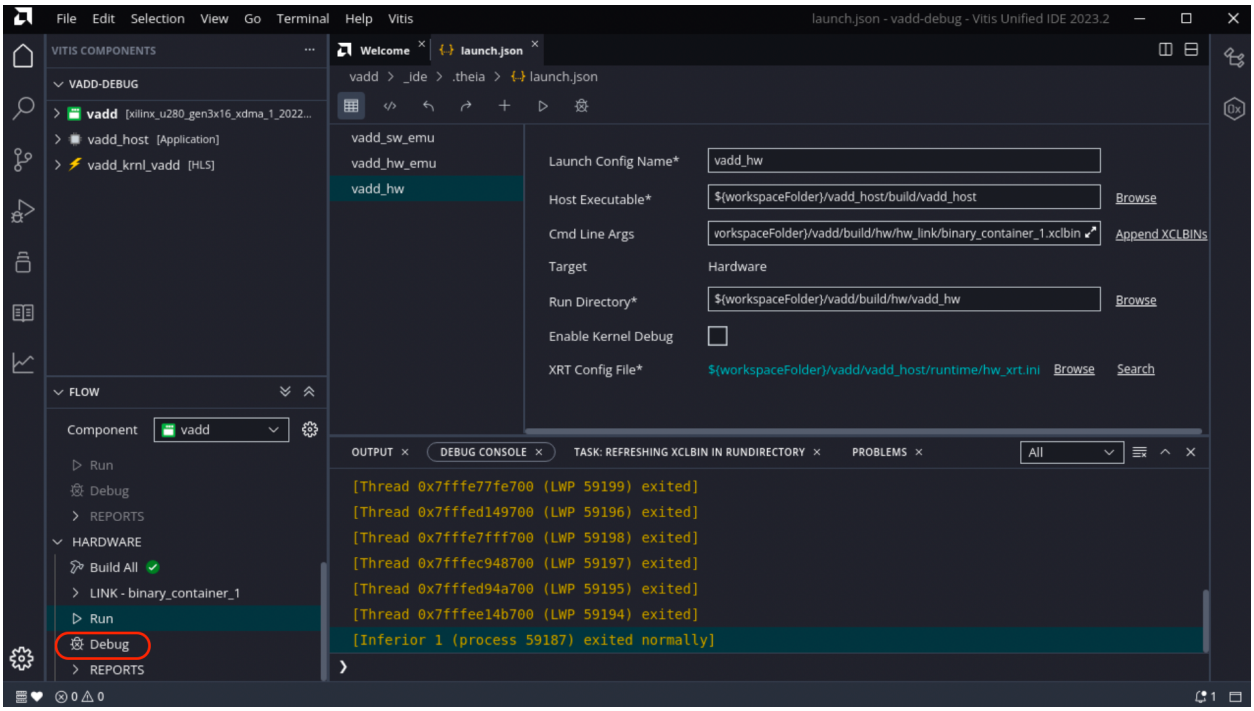
```
void krnl_vadd(uint32_t* in1, uint32_t* in2, uint32_t* out, int size) {
#pragma HLS INTERFACE m_axi port = in1 bundle = gmem0
#pragma HLS INTERFACE m_axi port = in2 bundle = gmem1
#pragma HLS INTERFACE m_axi port = out bundle = gmem0
```

Double-click on the System ILA. You'll see that it uses 20 BRAM slices. With a total of 2016 BRAM slices on the Alveo U280, this accounts for about 1% of BRAM utilization.

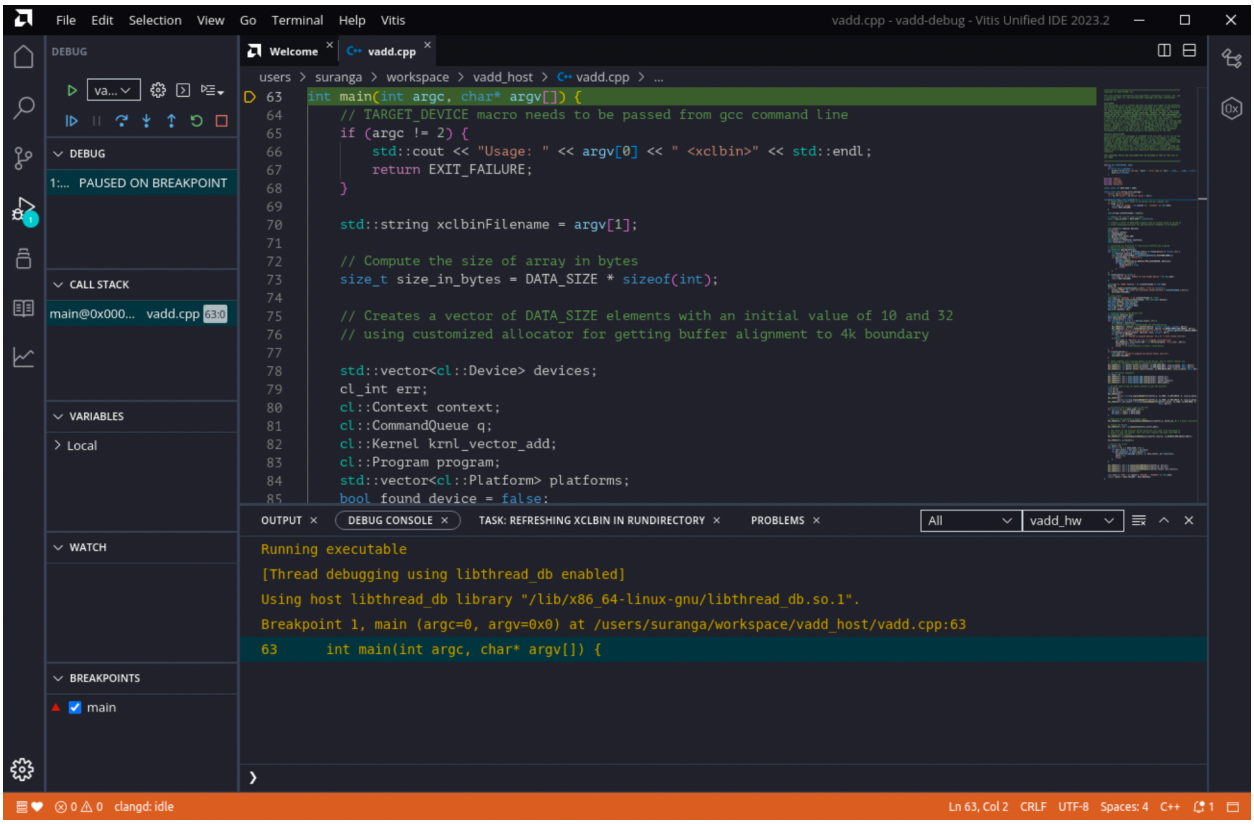


Close this Vivado project.

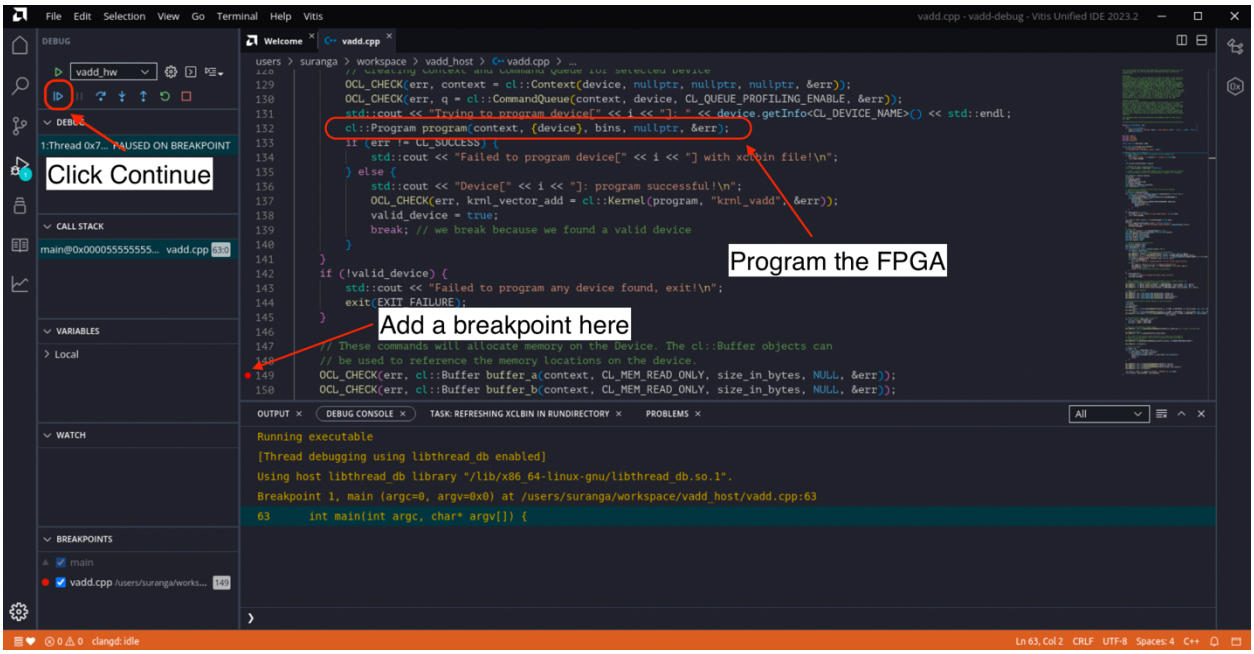
Now, go back to the Vitis IDE. In Flow -> Hardware, click **Debug**.



After this, the host code (`vadd.cpp`) will open, allowing you to step through it. Note that the execution pointer is now at the start of the `main` program.



Next, we add a breakpoint in the code right after the section where the FPGA is programmed. Then, click **Continue**.



After the program hits this breakpoint, we can launch the XVC server. The XVC server cannot be launched until the bitstream is programmed, because the XVC server endpoint of the FPGA is only exposed after programming the bitstream.

Now, we are going to launch the XVC server on the host node. It acts as a bridge between the FPGA hardware and Vivado Hardware Manager, converting JTAG commands into actions on the FPGA.

Open a new tab in the terminal and run the following command.

```
debug_hw --xvc_pcie /dev/xvc_pub.u0 --hw_server
```

Here's a breakdown of what each part of the command does:

`debug_hw`: This is the command used to initiate a hardware debugging session to interact with the FPGA.

`--xvc_pcie`: This flag specifies that the XVC server will communicate with the FPGA over a PCIe interface. It tells the tool to use PCIe to establish a connection with the FPGA's XVC server rather than using traditional USB or Ethernet connections.

`/dev/xvc_pub.u0`: This refers to the device file that represents the XVC server endpoint. The path `/dev/xvc_pub.u0` refers to the device interface exposed by the system that allows the debugger to access the FPGA through the XVC protocol.

`--hw_server`: This flag tells the debugging tool to connect to the XVC server. It enables the debugger to interact with the FPGA's hardware configuration and state.

After launching the XVC server, you will see an output similar to this:

```
suranga@pc171:~$ debug_hw --xvc_pcie /dev/xvc_pub.u0 --hw_server
launching xvc_pcie...
/proj/octfpga-PG0/tools/Xilinx/Vivado/2023.2/bin/xvc_pcie -d /dev/xvc_pub.u0 -s TCP::10200
launching hw_server...
/proj/octfpga-PG0/tools/Xilinx/Vivado/2023.2/bin/hw_server -sTCP::3121

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

Open another tab in the terminal and run the following command to connect to the XVC server.

```
debug_hw --vivado --host localhost --ltx_file <filename.ltx>
```

The `<filename.ltx>` file in this example is located at `~/workspace/vadd/build/hw/hw_link/binary_container_1.ltx`

Here's a breakdown of what each part of the command does:

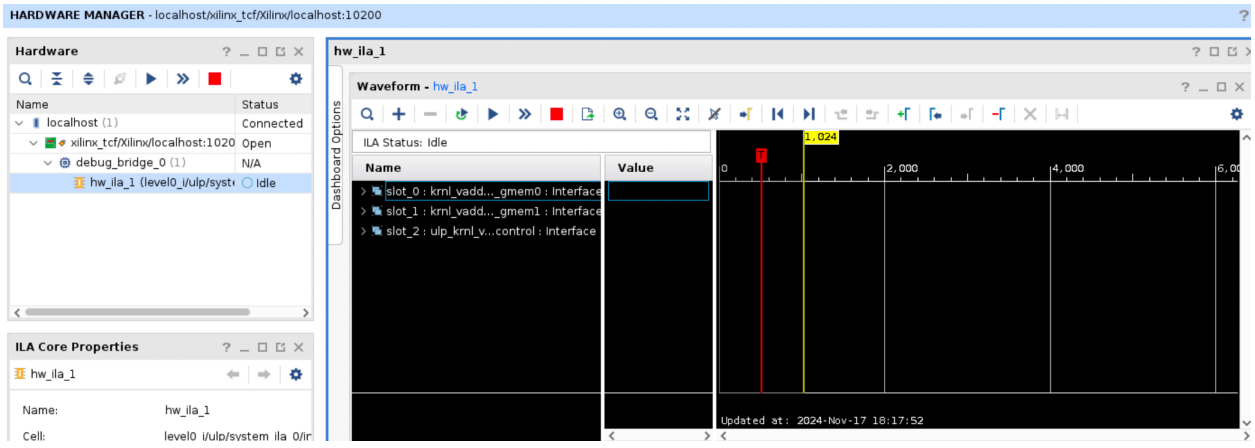
`debug_hw`: Initiates the debug session.

`--vivado`: This option specifies that the debugging should be done using Vivado's debugging capabilities.

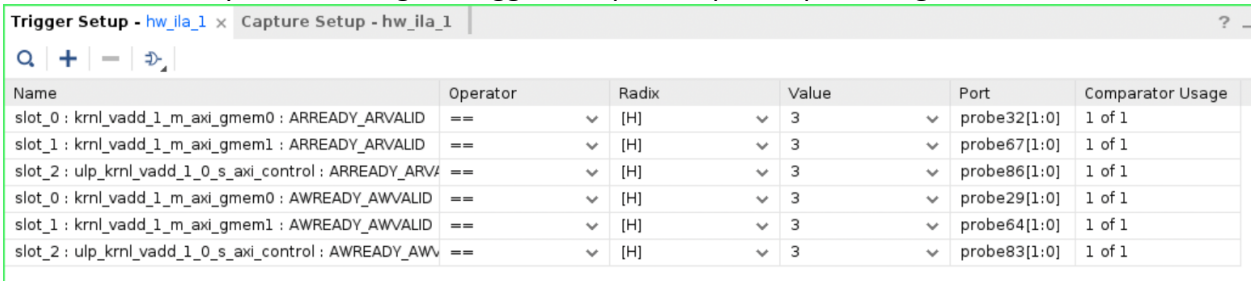
`--hostname <host name>`: Specifies the hostname or IP address of the system where the FPGA hardware is located. In our case, since the FPGA is on the same machine as the debugger, the `hostname` is set to `localhost`.

`--ltx_file <filename.ltx>`: Specifies the path to the `.ltx` (Logic Trace Exchange) file. It is a configuration file created during FPGA design and synthesis which includes information about signals, nets, and probes you want to monitor or analyze using Vivado's debugging tools.

After running this command, Vivado automatically launches the **Hardware Manager** and **Waveform Viewer**.

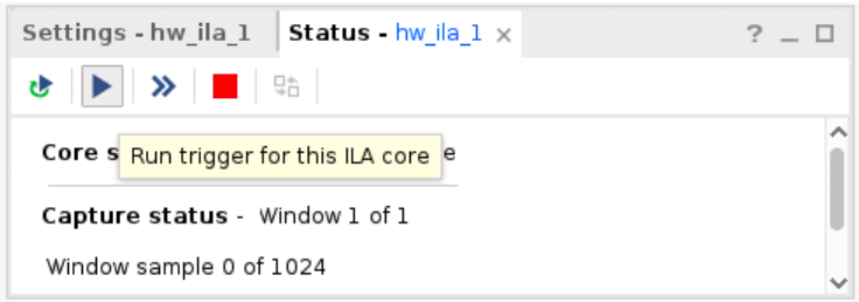


The Hardware Manager connects to the FPGA hardware (in this case, on `localhost`) and initializes the debugging session using the provided `.ltx` file. The **ILA** interface becomes available, where you can configure trigger setups to capture specific signal conditions.

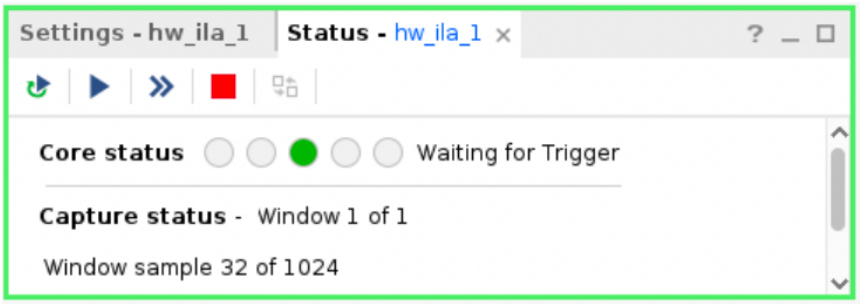


We'll leave the default trigger setup as it is.

Once the trigger is set, click on the **Run** button to start the monitoring process.

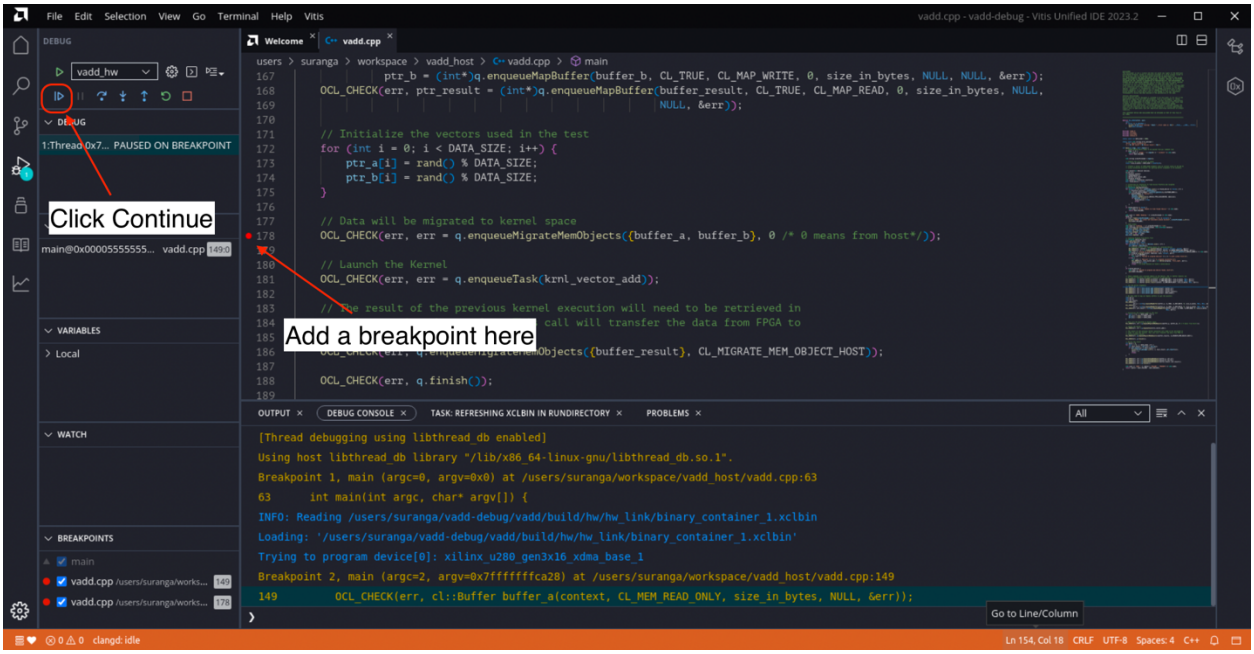


Note that the system waits for the trigger condition to be met.



The ILA is now ready to capture the relevant waveform data once the trigger condition is met. The captured data will be displayed in the Waveform Viewer.

Set a breakpoint right before transferring data from memory to the FPGA, then click Continue.



At the time the program hits this breakpoint, no data have yet been transferred to the kernel space. To verify this, run:

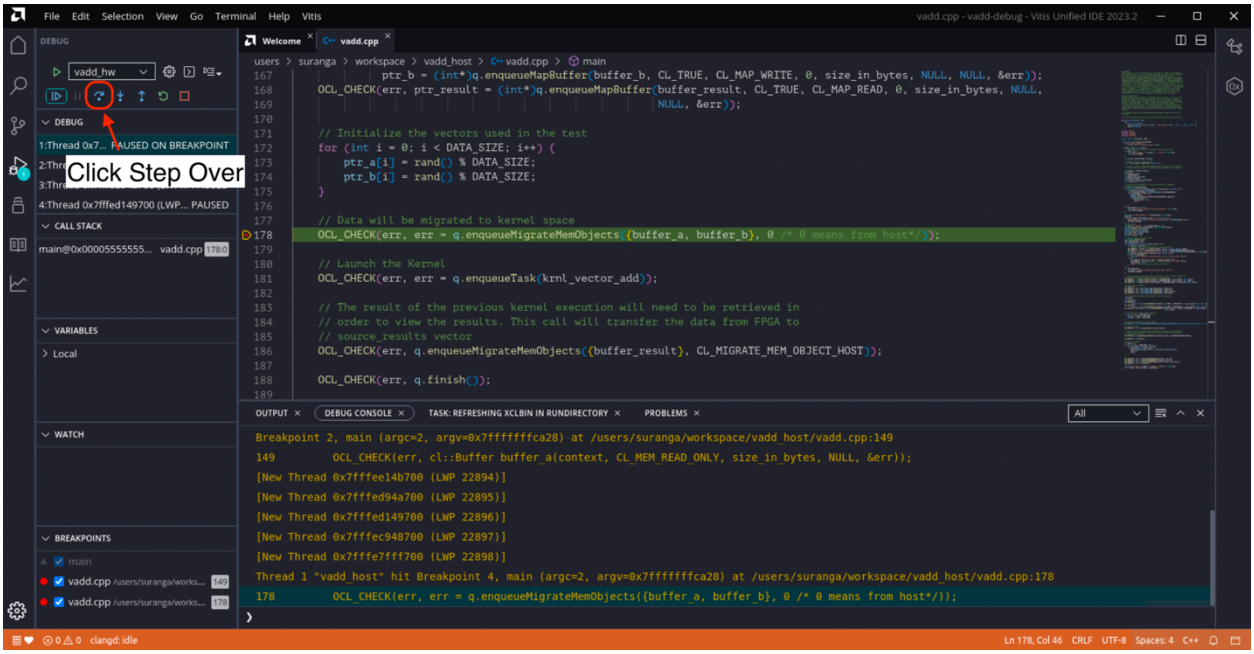
```
xbutil examine -d <device ID> --report memory
```

For example: `xbutil examine -d 37:00.1 --report memory`

Notice that all DMA transfer metrics show 0 bytes have been transferred from/to kernel space.

```
DMA Transfer Metrics
  Chan[ 0].h2c:   0 Byte
  Chan[ 0].c2h:   0 Byte
  Chan[ 1].h2c:   0 Byte
  Chan[ 1].c2h:   0 Byte
```

Now, step over the breakpoint to transfer input date to kernel space.



Check DMA transfer metrics again.

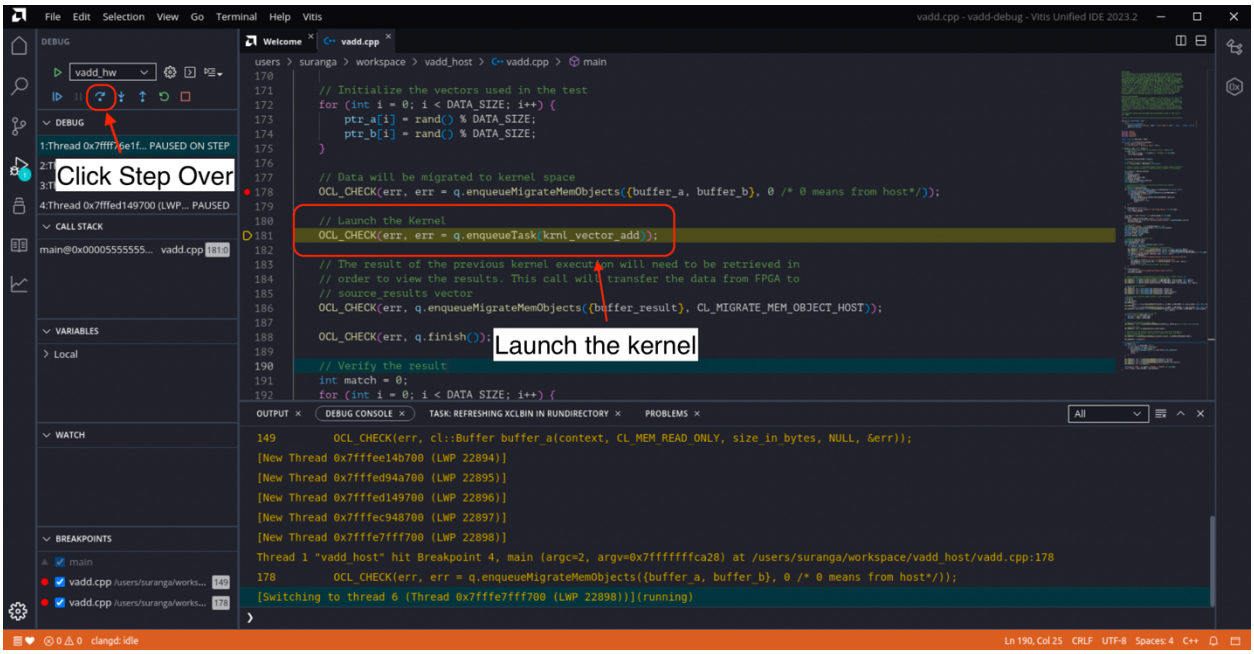
```
DMA Transfer Metrics
  Chan[ 0].h2c:   16 KB
  Chan[ 0].c2h:   0 Byte
  Chan[ 1].h2c:   16 KB
  Chan[ 1].c2h:   0 Byte
```

Note that h2c (host to card) channels are used to transmit data from the host system to the FPGA. Each of these has transferred 16 KB of data. These data represent the two input vectors that are to be added.

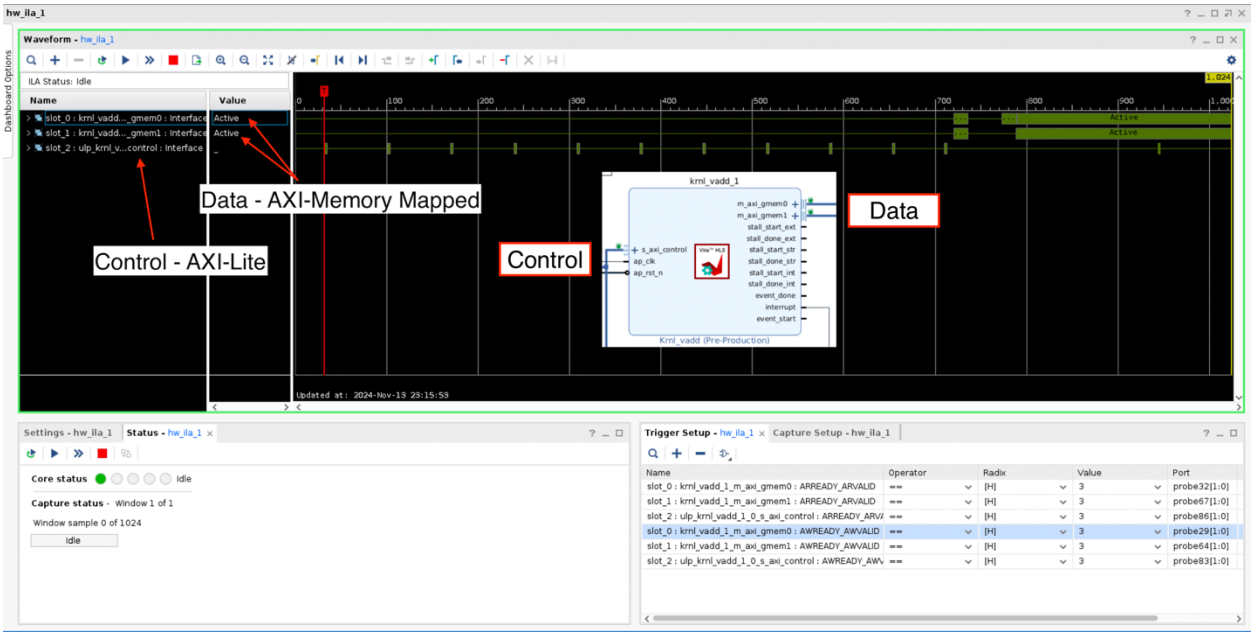
$$Total\ data\ size = Size\ of\ a\ vector\ element \times Number\ of\ vector\ elements$$

Each input vector has 32-bit elements, and there are 4,096 elements in total. That makes each vector 16 KB, which matches the DMA transfer numbers shown when running the `xbutil examine --report memory` command.

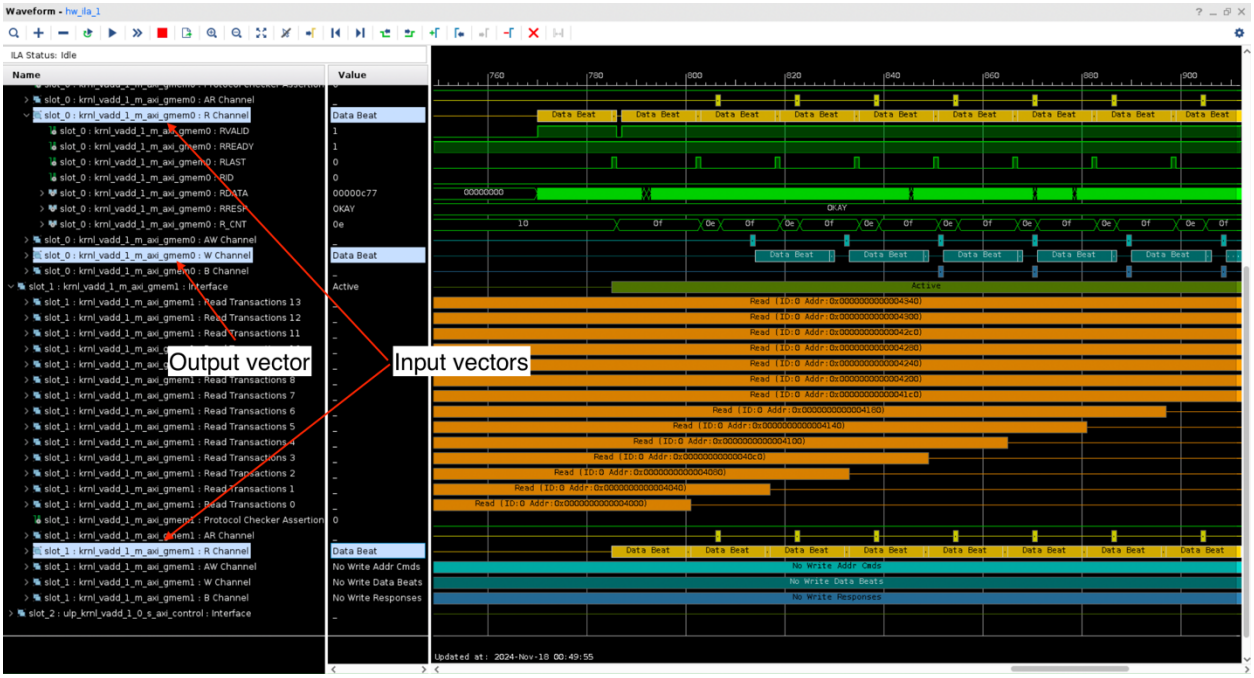
Click **Step Over** to launch the kernel.



Once you launch the kernel, you'll see that the trigger condition is met, allowing you to visualize the waveforms. Identify the signals of AXI memory-mapped channels and AXI-Lite control channel.

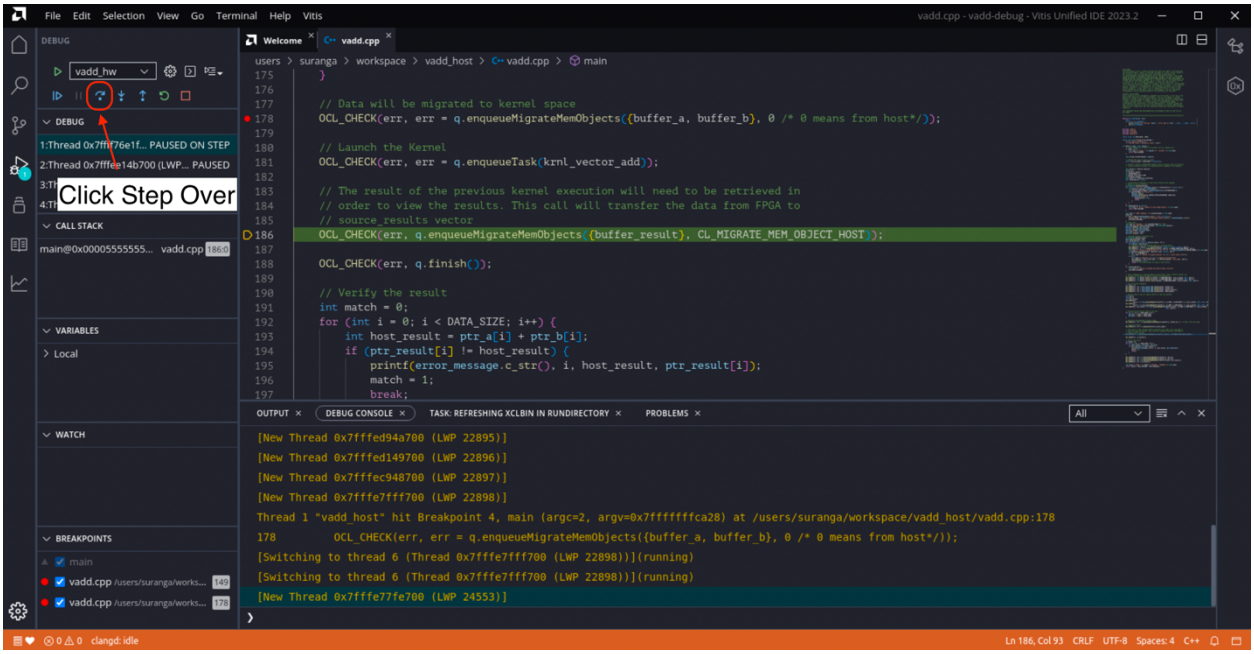


In the waveform viewer, you can identify the channels corresponding to the two input vectors and the output vector.



Now, review the first few values of the input vectors and verify if the output vector matches the sum of the two input vectors.

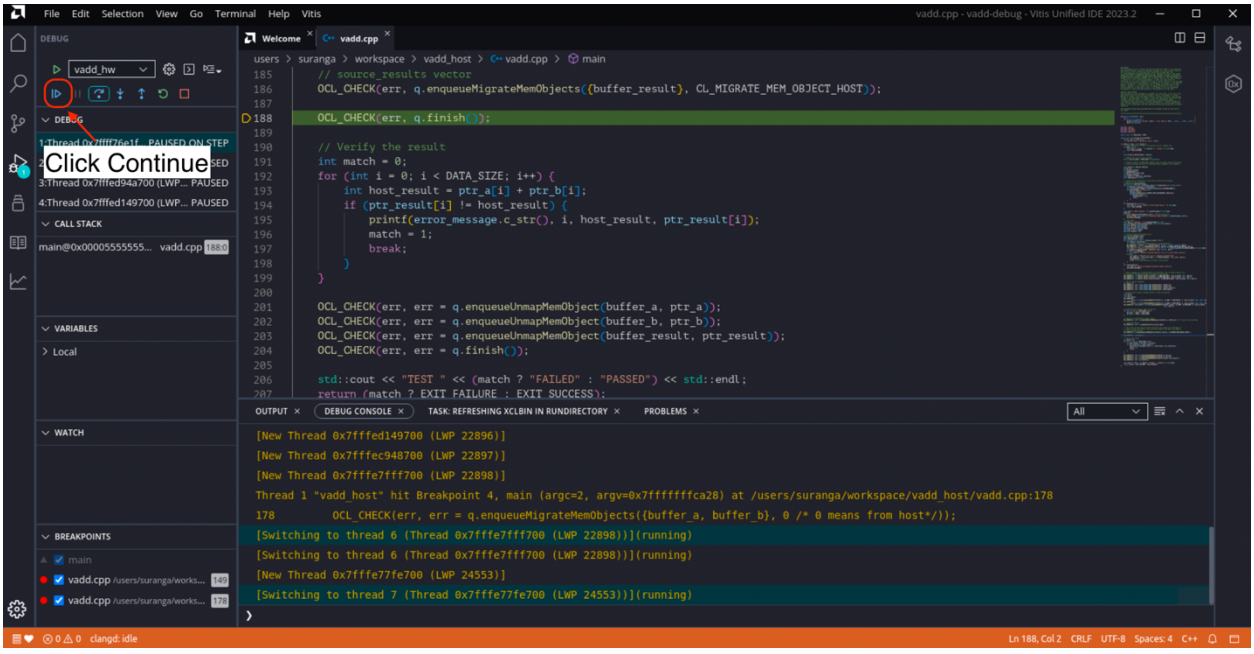
After the kernel execution is completed, you may transfer the output data to host. Click on **Step Over** button to do this.

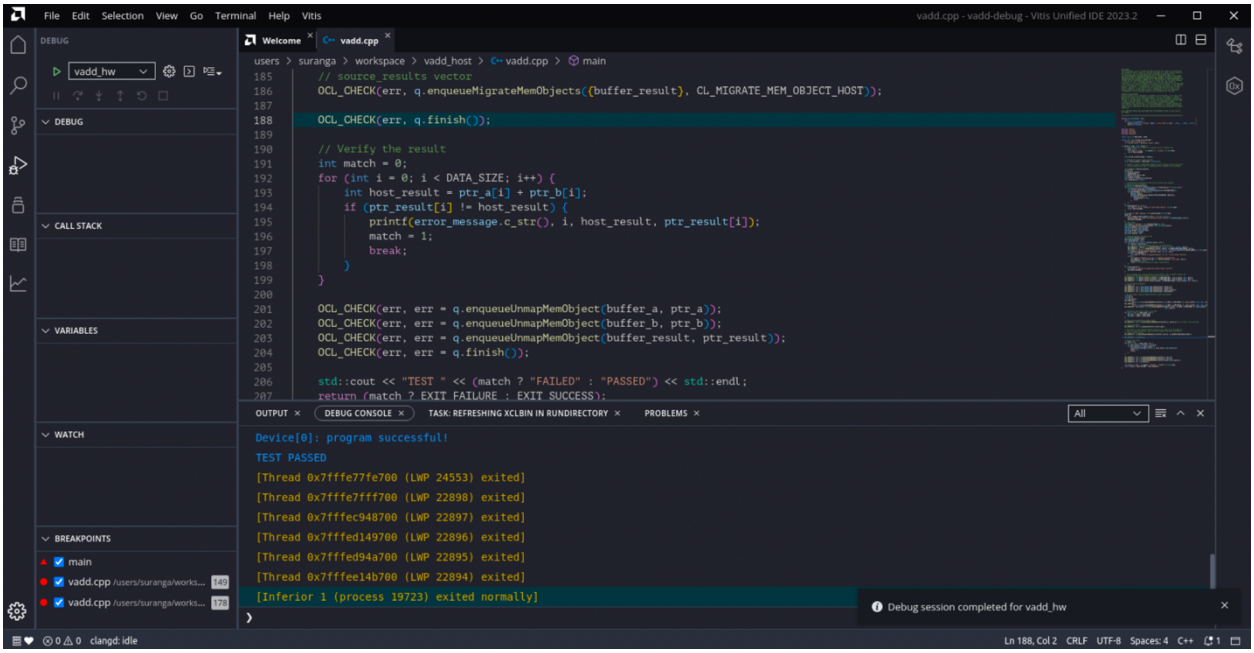


Now, check the DMA transfer metrics again. You'll see that 16 KB of data has been transferred on the c2h (card to host) channel. This corresponds to the output vector written from the kernel to memory.

```
DMA Transfer Metrics
  Chan[ 0].h2c:   16 KB
  Chan[ 0].c2h:   16 KB
  Chan[ 1].h2c:   16 KB
  Chan[ 1].c2h:   0 Byte
```

Click **Continue** to finish the execution of the program.





Summary

In this tutorial, we performed hardware debugging on an FPGA in the Open Cloud Testbed. Here's what we did:

- Allocated an Alveo U280 node in OCT and accessed it via VNC.
- Launched Vitis IDE and created a vector addition example project.
- Inserted ChipScope debug cores into the design for real-time signal monitoring.
- Built the bitstream and host executables.
- Launched the debugger and stepped through the host code.
- Captured FPGA signals using the Vivado hardware server and waveform viewer.