



Bridge of Life Education

AAHLS Special Project Lab A Presentation - RTL kernel workflow and Mixing C++ and RTL Kernels

61275047H 黃聖歲

https://github.com/kevin33713371/2024_Fall_NTU_AAHLSP

Outline

- Download & Insert the Board File of Alveo U50
- Before the Experiment – Clone the repository from Vitis tutorials
- RTL Kernel Workflow
 - Packaging an existing RTL design into an RTL kernel
 - Host application interacts with the kernel
 - Using the RTL kernel in a Vitis IDE project
- Mixing C++ & RTL Kernels
 - Building an Application with C++ Based Kernel
 - Building an Application with C++ & RTL Based Kernel
- Reference

Download & Insert the Board File of Alveo U50

Since the Alveo U50 board is not included by default in Vivado 2022.1, you need to download the required board file for the U50 from Xilinx's official GitHub repository, XilinxBoardStore.

At first, clone the overall repository of XilinxBoardStore by following command.

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

Since the Alveo U50 board file is only available in the “2020.1.1” branch of the XilinxBoardStore repository, switch the downloaded XilinxBoardStore folder to branch “2020.1.1” using the following command.

```
cd XilinxBoardStore  
git checkout 2020.1.1
```

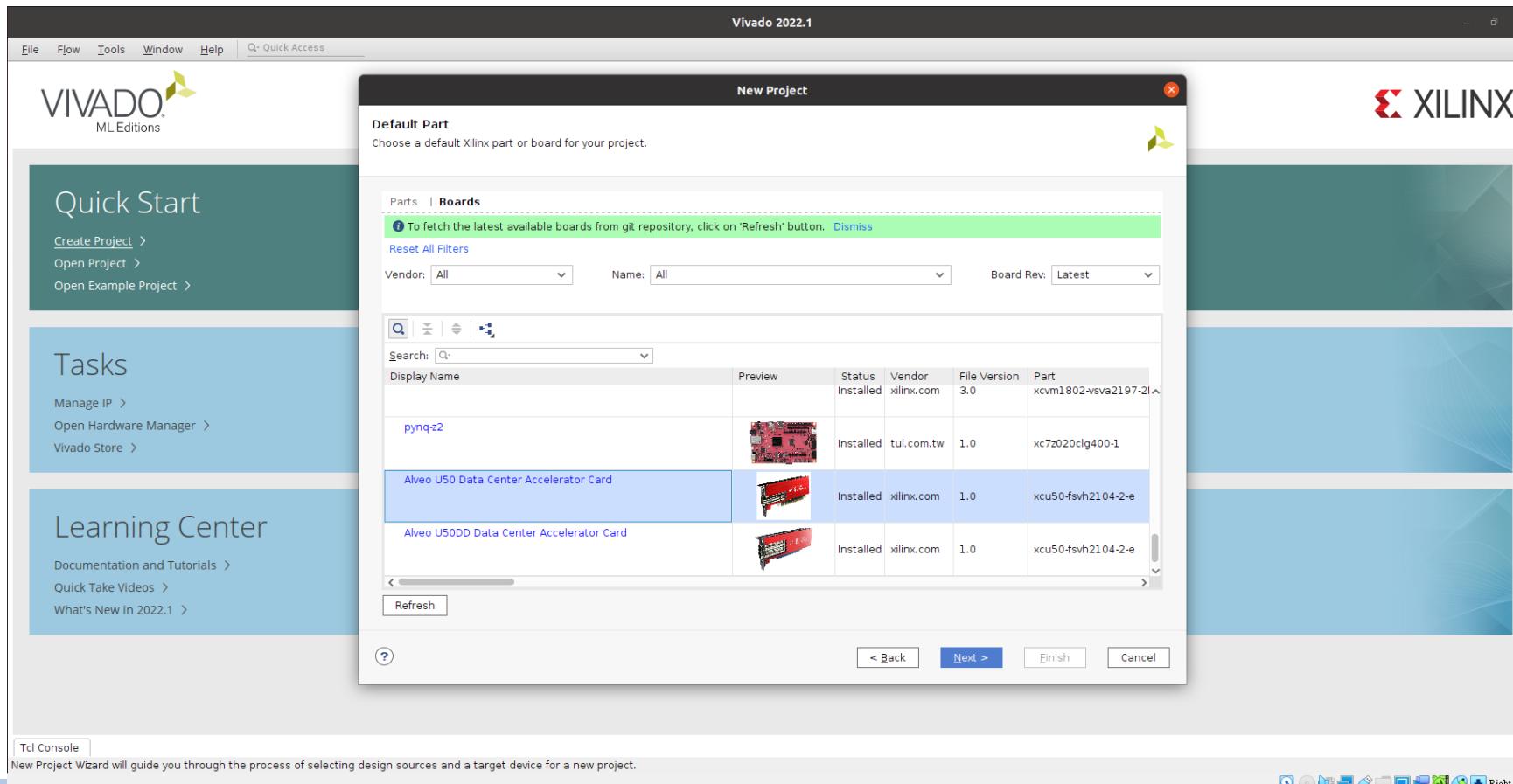
Download & Insert the Board File of Alveo U50

Next, change the current directory to the folder containing au50 (Alveo U50), and copy the au50 folder to the corresponding board file directory in Vivado, my directory is /tools/Xilinx/Vivado/2022.1/data/boards/board_files

```
cd XilinxBoardStore/boards/board_files  
sudo cp -r au50 /tools/Xilinx/Vivado/2022.1/data/boards/board_files
```

Download & Insert the Board File of Alveo U50

Finally, open Vivado and create a new project. Ensure that the Alveo U50 is available in the selectable boards list.



Before the Experiment – Clone the repository from vitis tutorials

Before the experiment, we need to access the tutorial reference files for Xilinx Vitis Tutorial GitHub. By following command, we can clone the repository by git. At first, clone the overall repository by following command.

```
git clone https://github.com/Xilinx/Vitis-Tutorials.git
```

Next, enter the directory of Vitis-Tutorials and switch the branch to 2022.1

```
cd Vitis_Tutorials  
git checkout 2022.1
```

RTL Kernel Flow

- Packaging an existing RTL design into an RTL kernel
- Host applications interacts with the kernel
- Using the RTL kernel in a Vitis IDE project

Package IP/ Package XO Flow

1. Create a New Project
2. Add Kernels Sources
3. Open the IP Packager
4. Specify the Control Protocol
5. Edit Ports and Interfaces
6. Add Control Registers and Address Offsets
7. Check Integrity, Assign Properties and Package IP

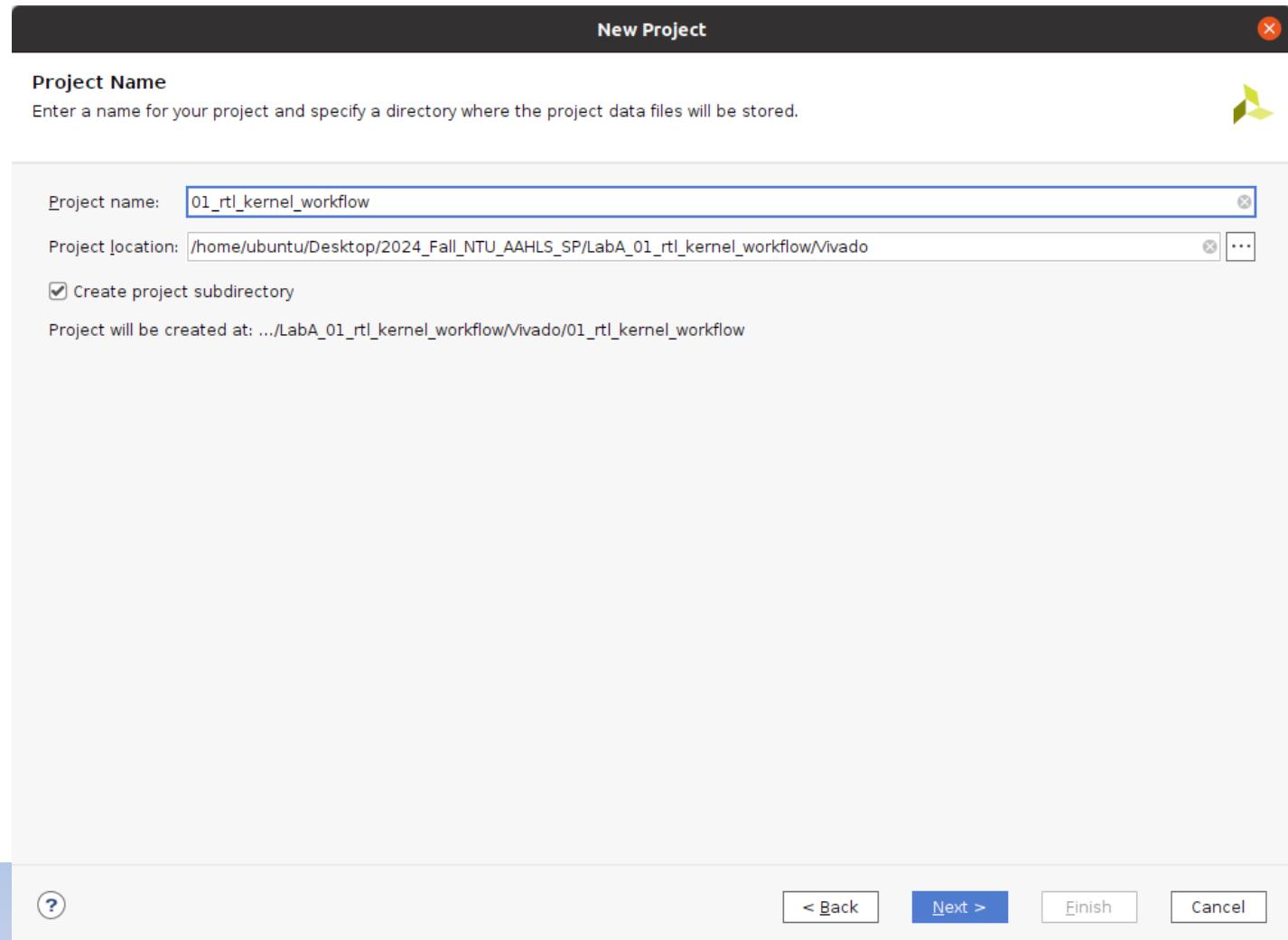
Create a New Project

Open the terminal window and enter the command “**vivado**” to launch the Vivado IDE.
Click the “**Create Project**” to create a vivado project.



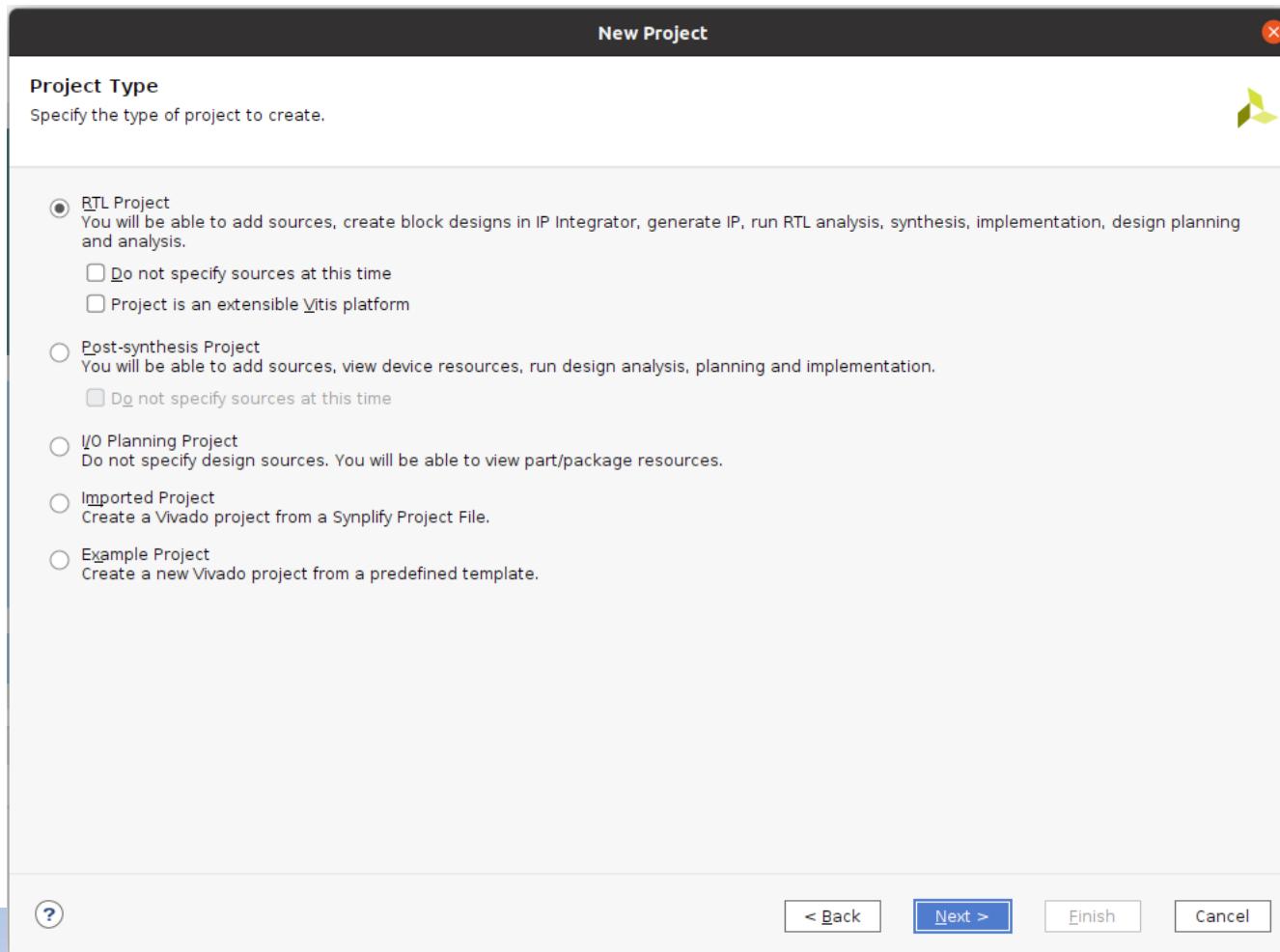
Create a New Project

Enter the project name and click the **Next** button. (The project name here is **01 rtl_kernel_workflow**)



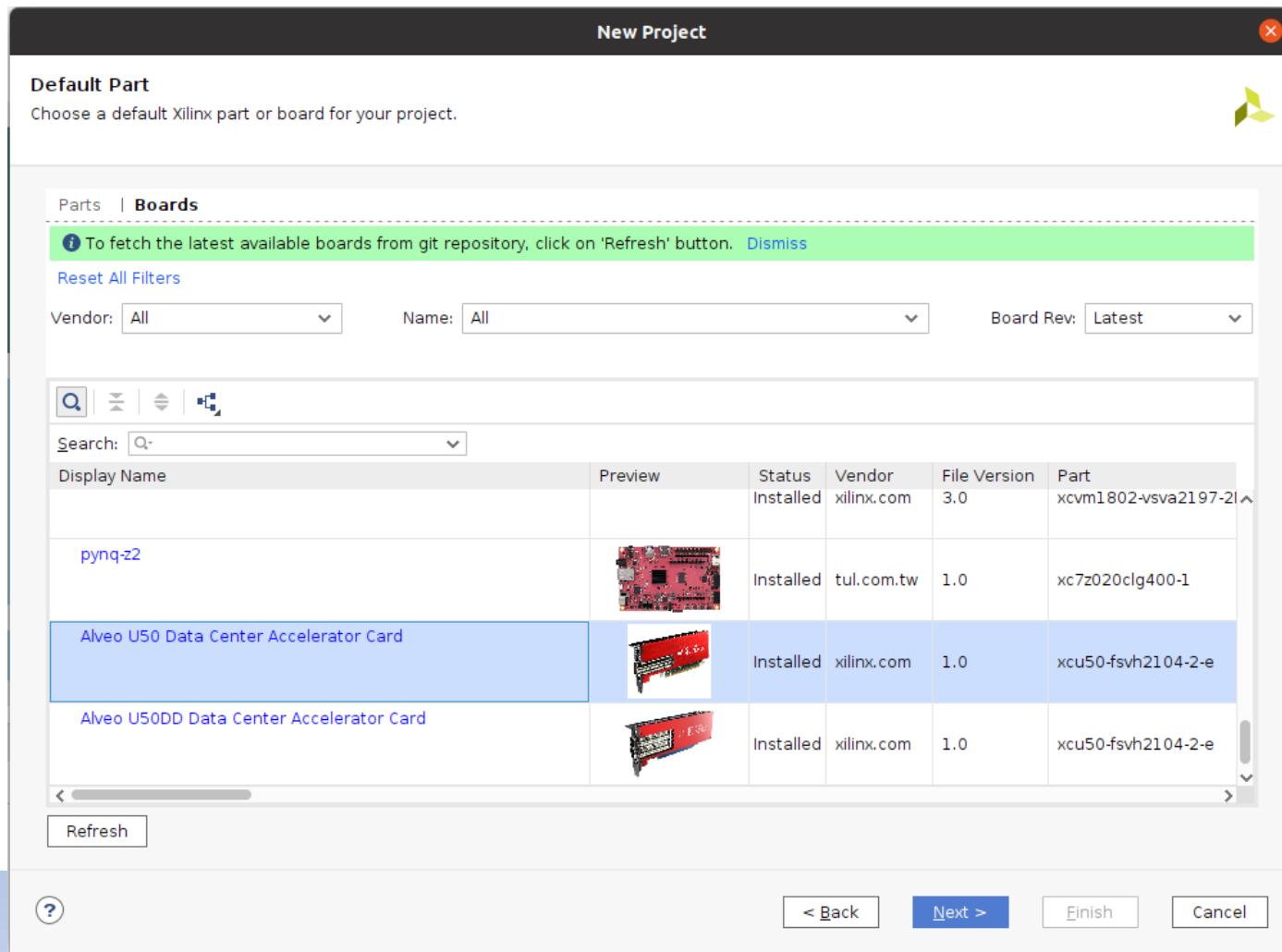
Create a New Project

Choose the “RTL Project” option and click the “**Next**” button.



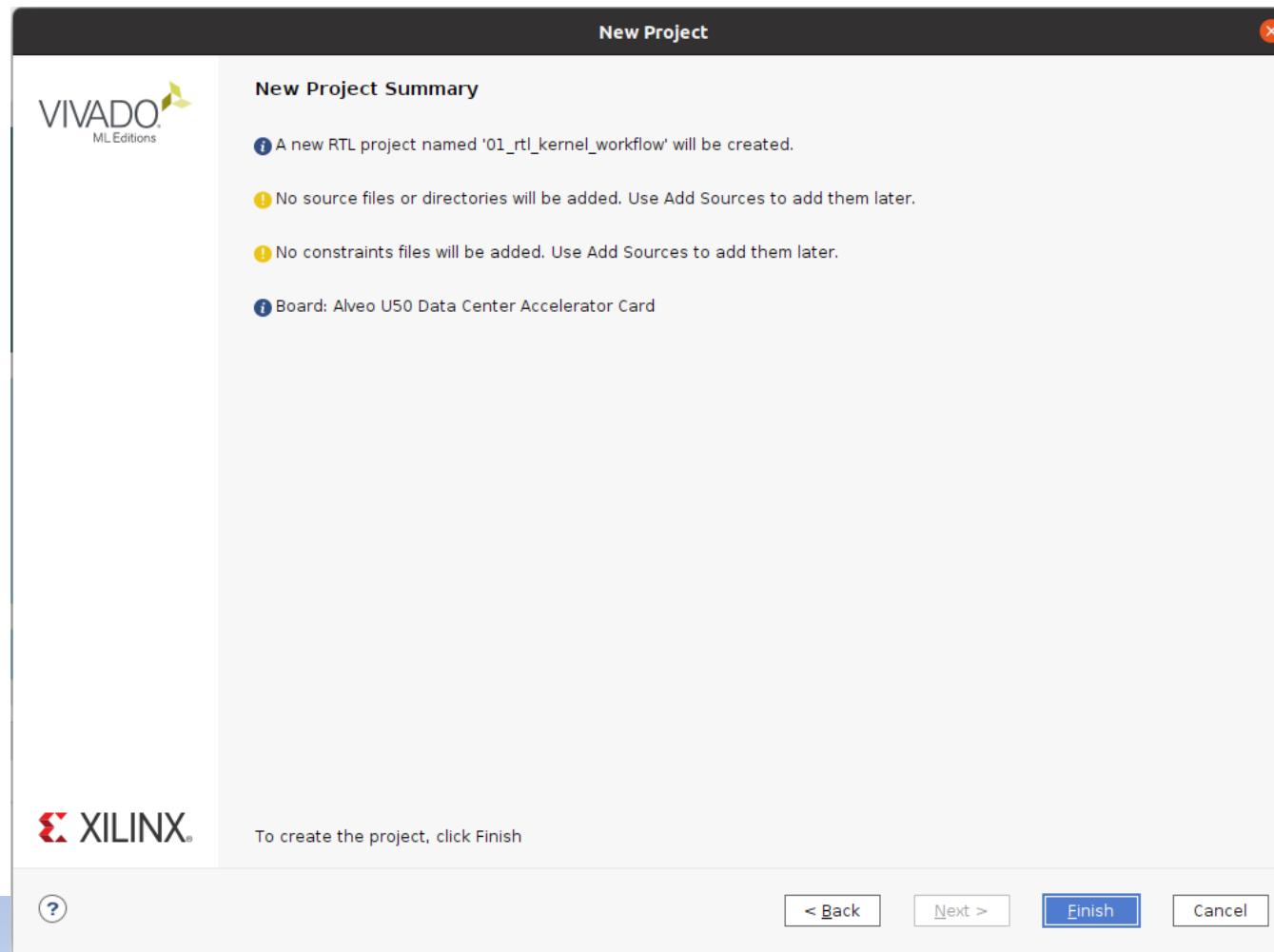
Create a New Project

Select the “Boards” option and find the “Alveo U50 Data Center Accelerator Card” that had been installed.



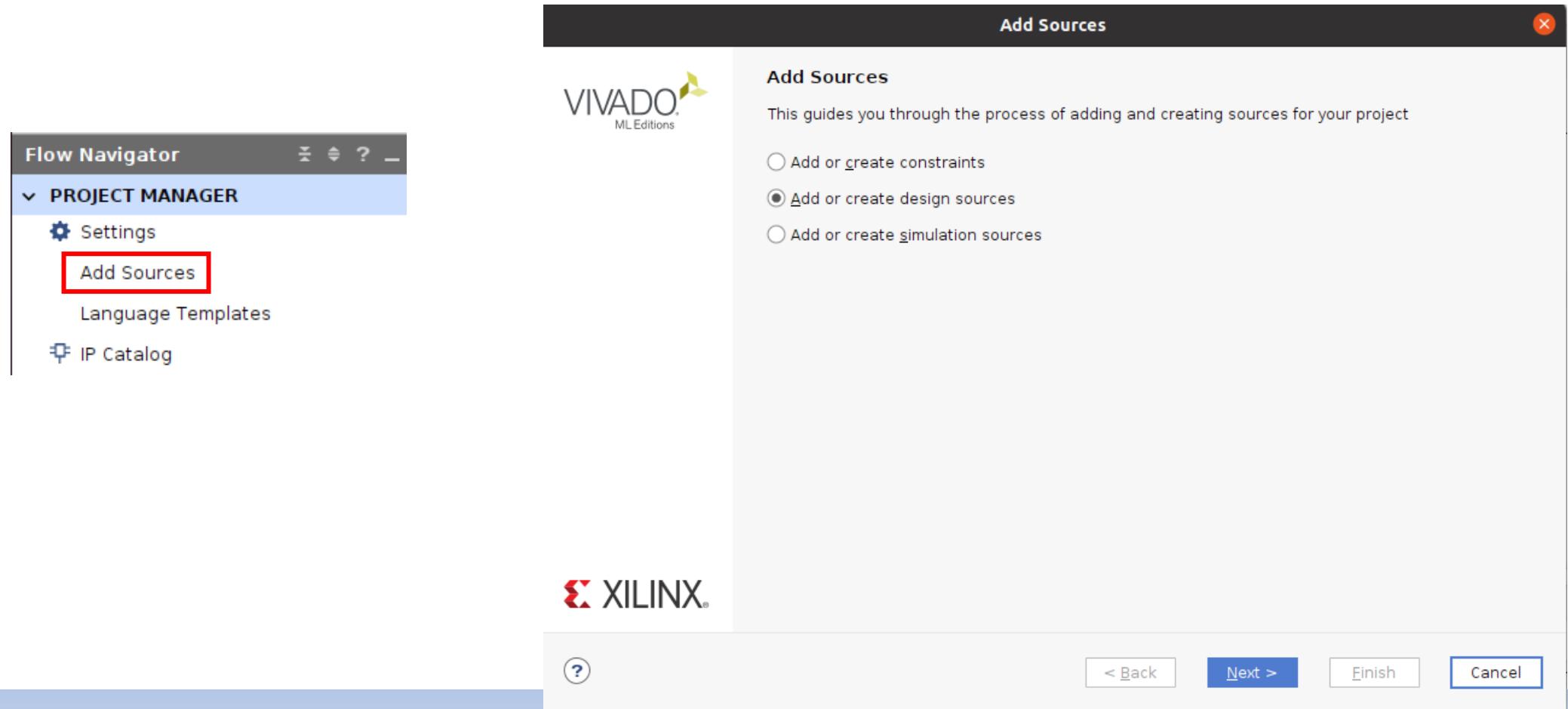
Create a New Project

Click the “Finish” button and the project will be created.



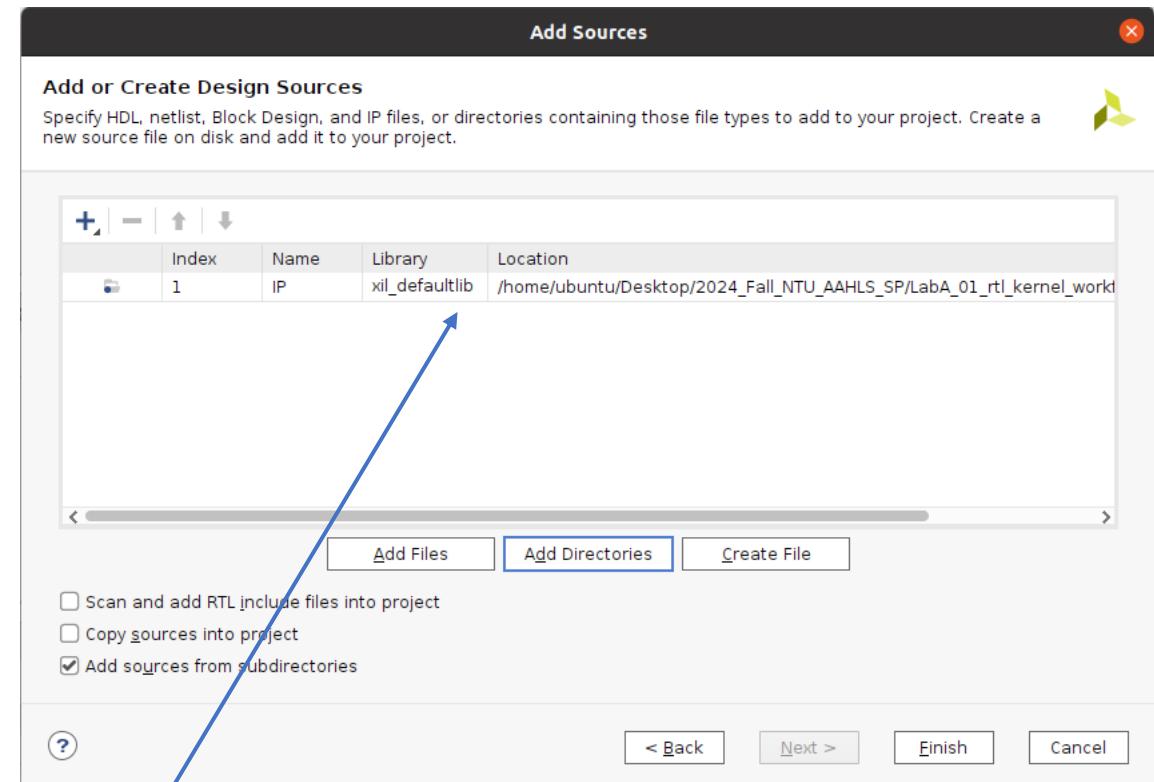
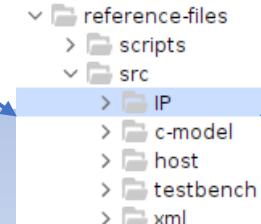
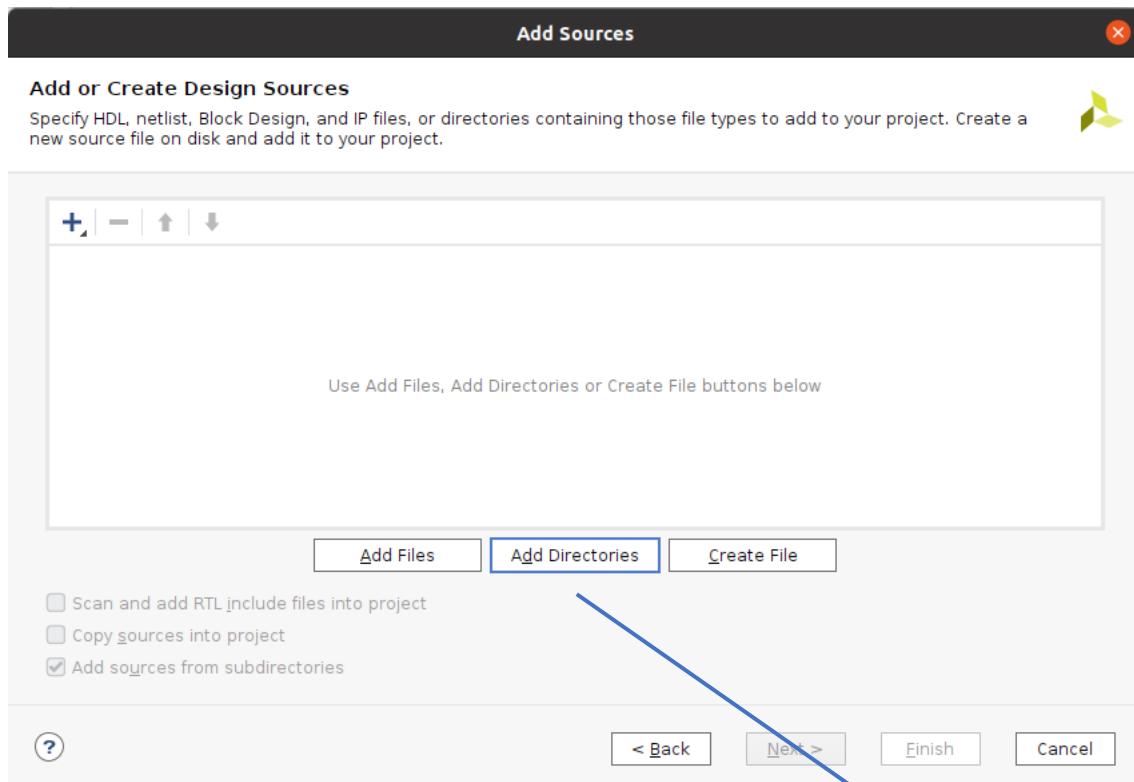
Add Kernels Sources

Find out "**PROJECT MANAGER**" at the top left corner of the window, and select the "**Add Source**" option.
A window like the one shown on the right side will pop up.



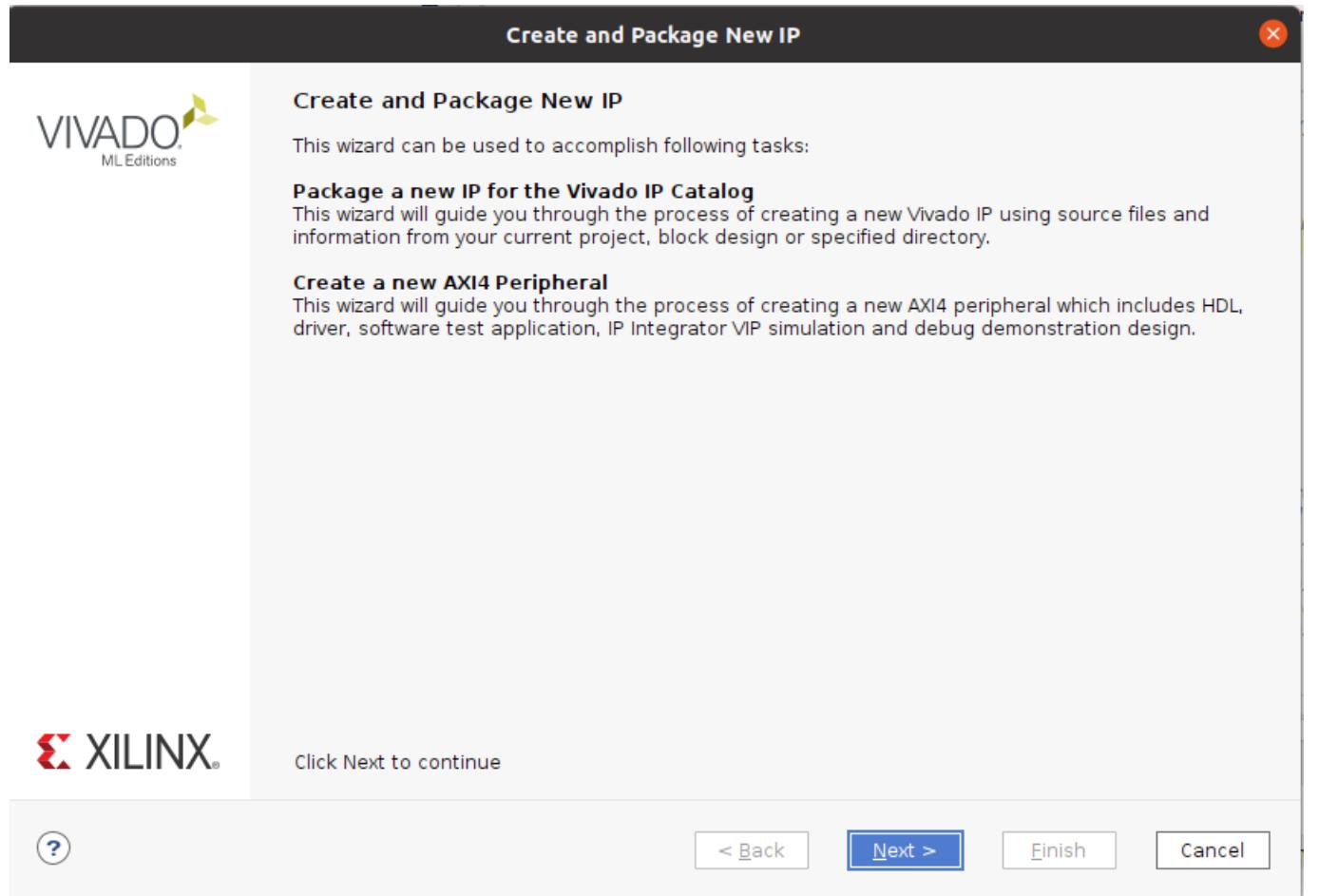
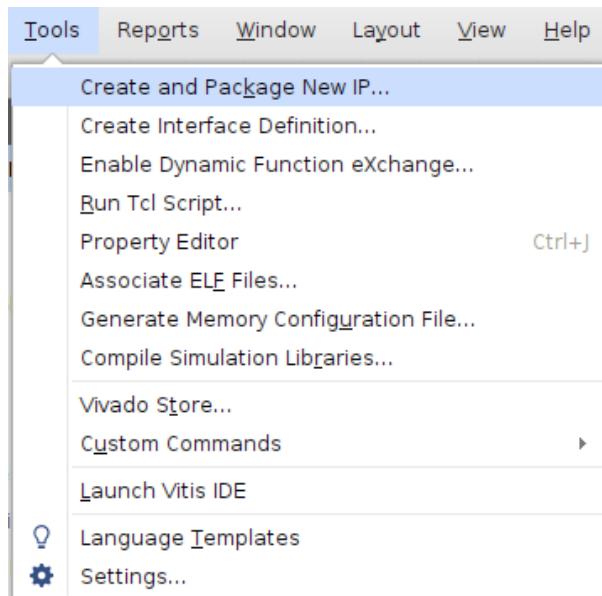
Add Kernels Sources

Click the “**Add Directories**” and set the folder that provided in the tutorial,
the IP folder path is “01_rtl_kernel_flow/reference-files/src/IP”.



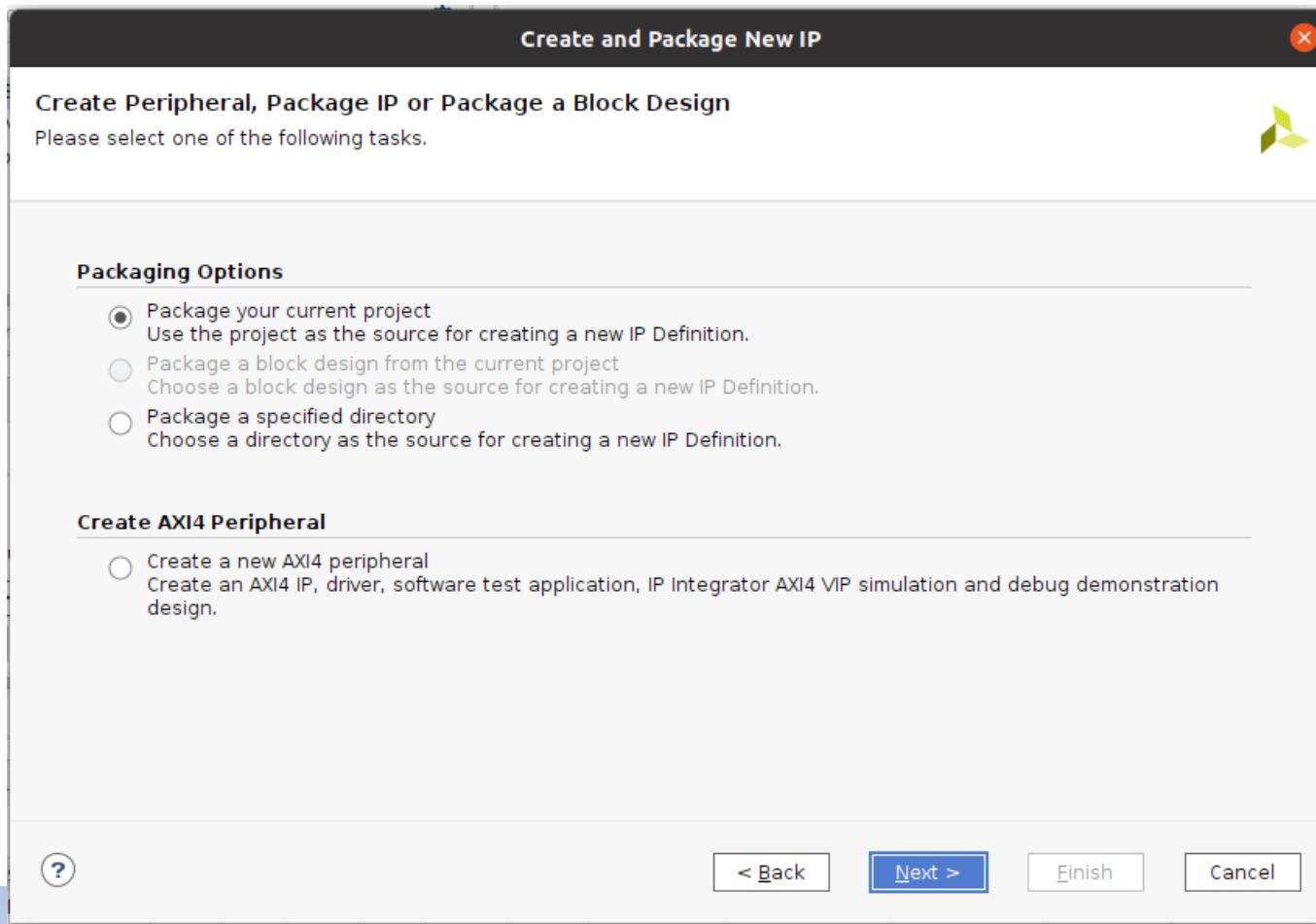
Open the IP Packager

Find out "**Tools**" in the top menu and select the "**Create and Package New IP...**" option.
A window similar to the one shown on the right side will pop up.



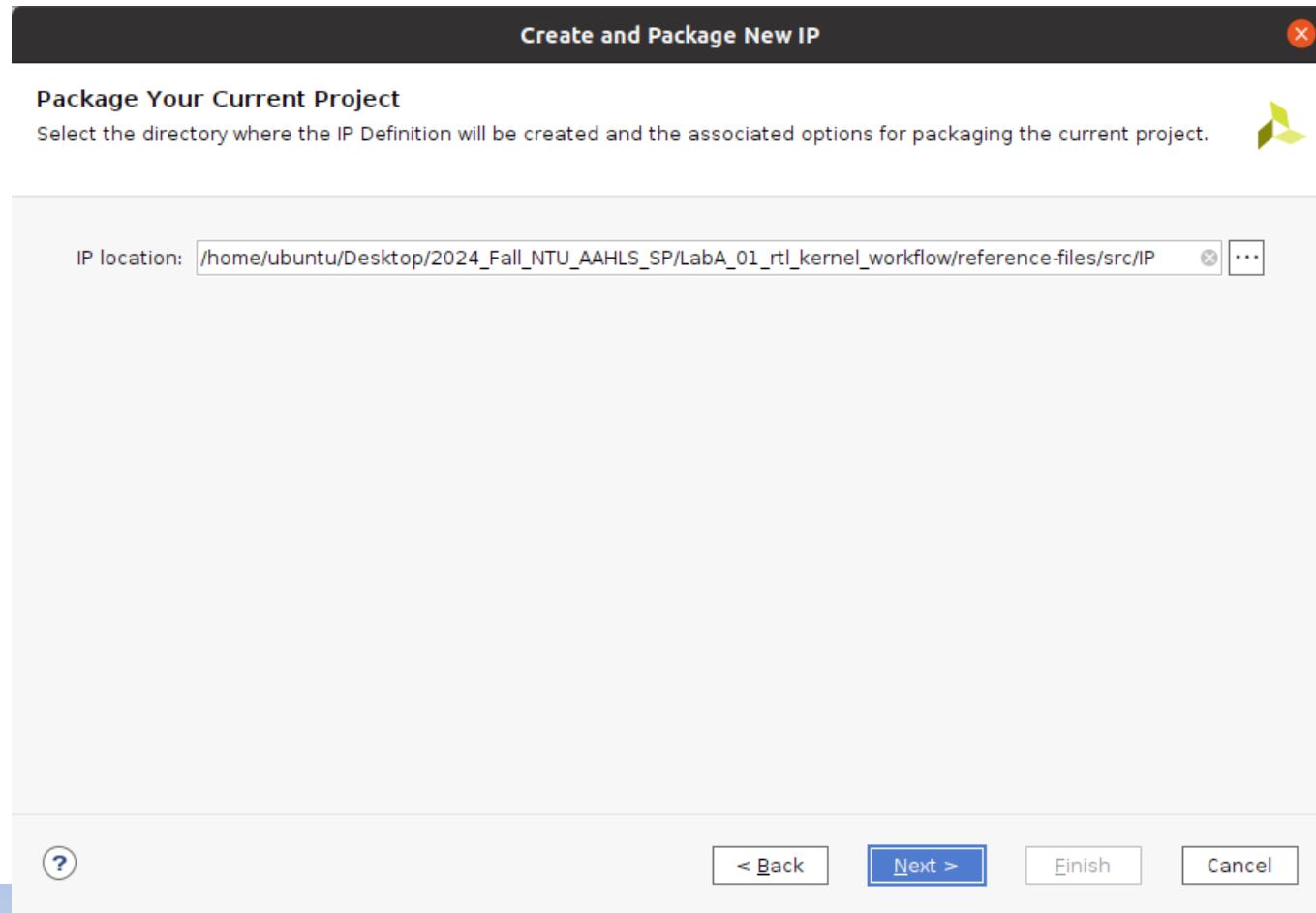
Open the IP Packager

Choose “**Package your current project**” and click the “**Next**” button.



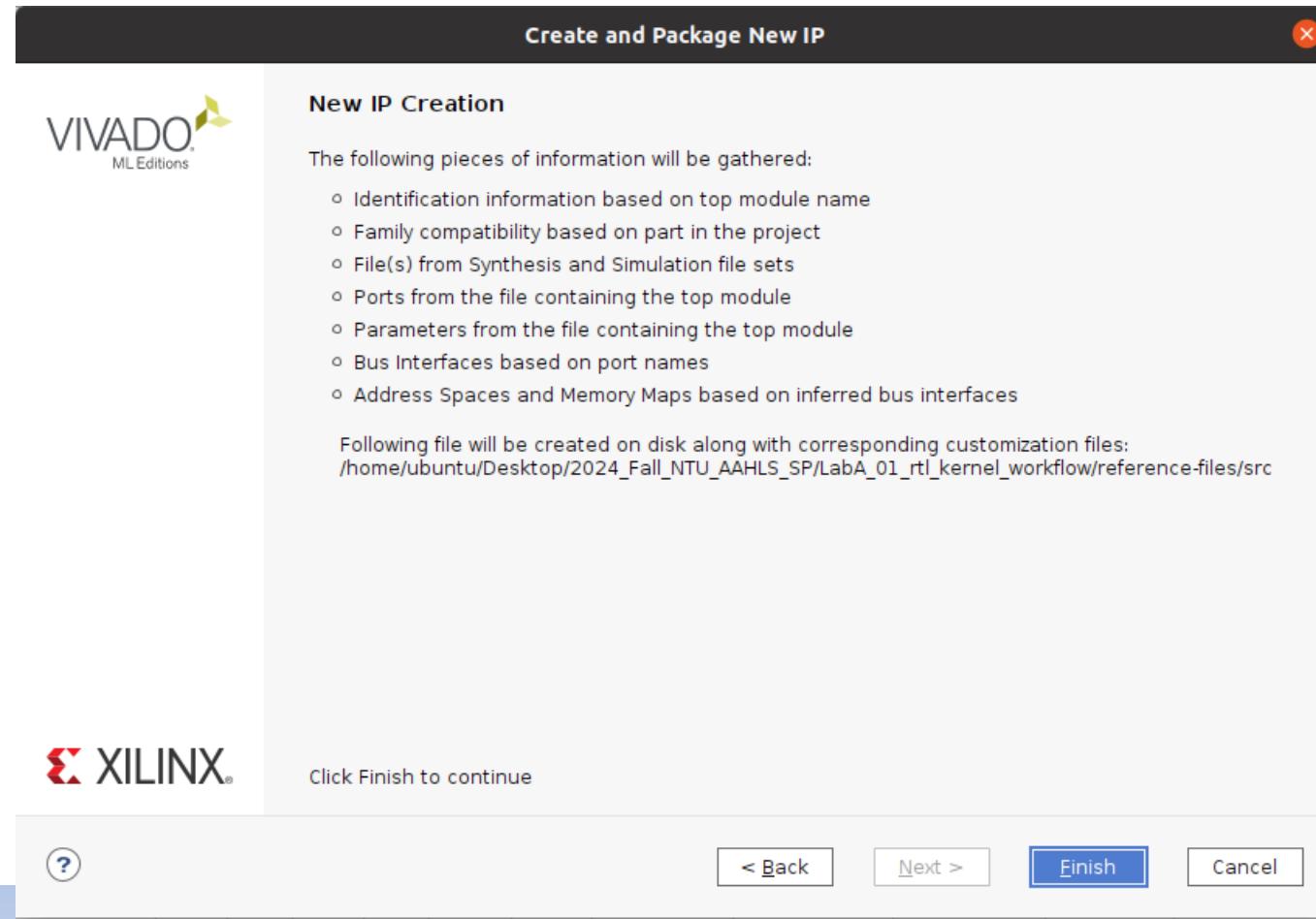
Open the IP Packager

The system will automatically set the IP location for us. Simply click the "Next" button.



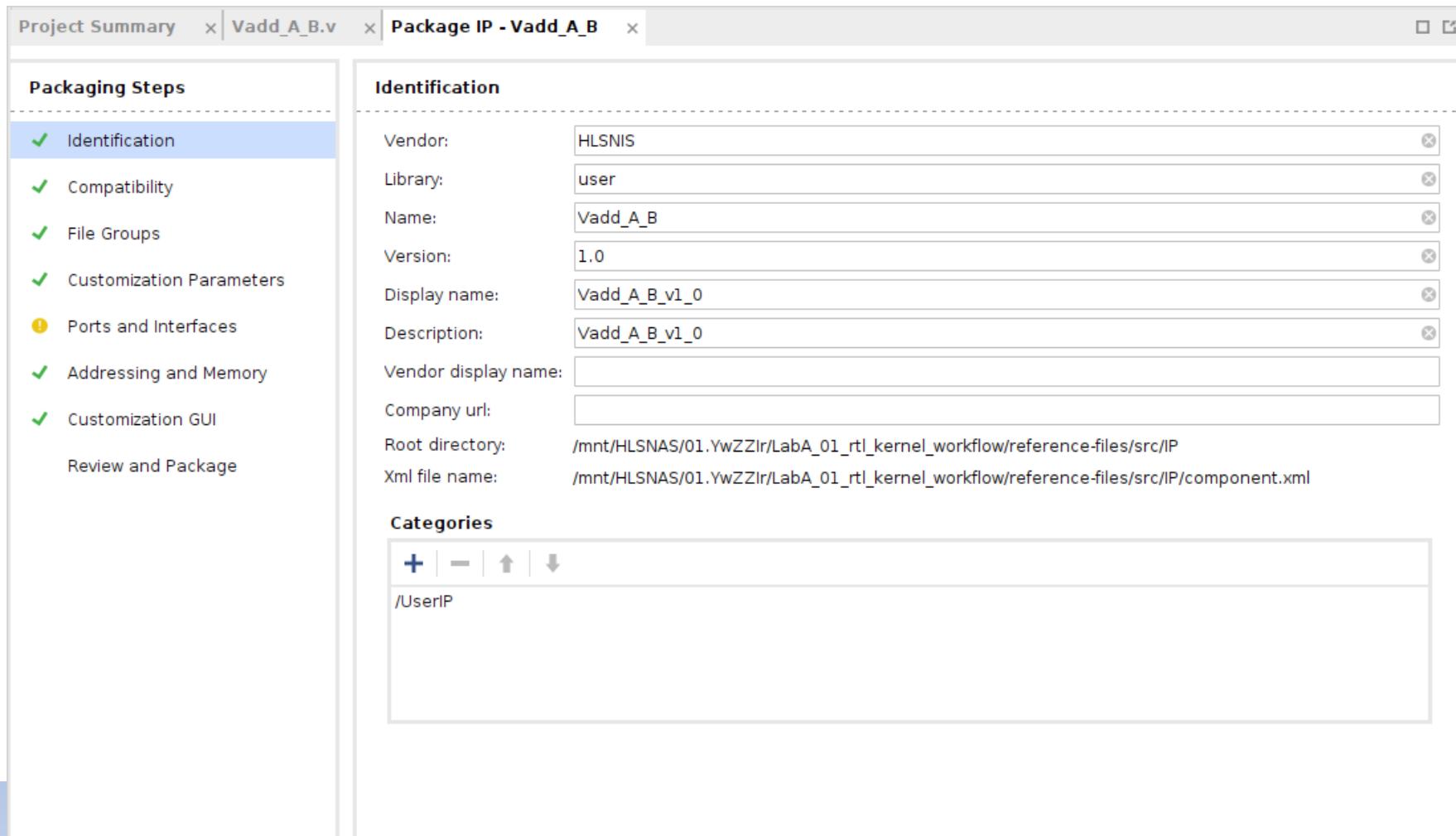
Open the IP Packager

Click the “Finish” button and the IP Packager window will be opened.



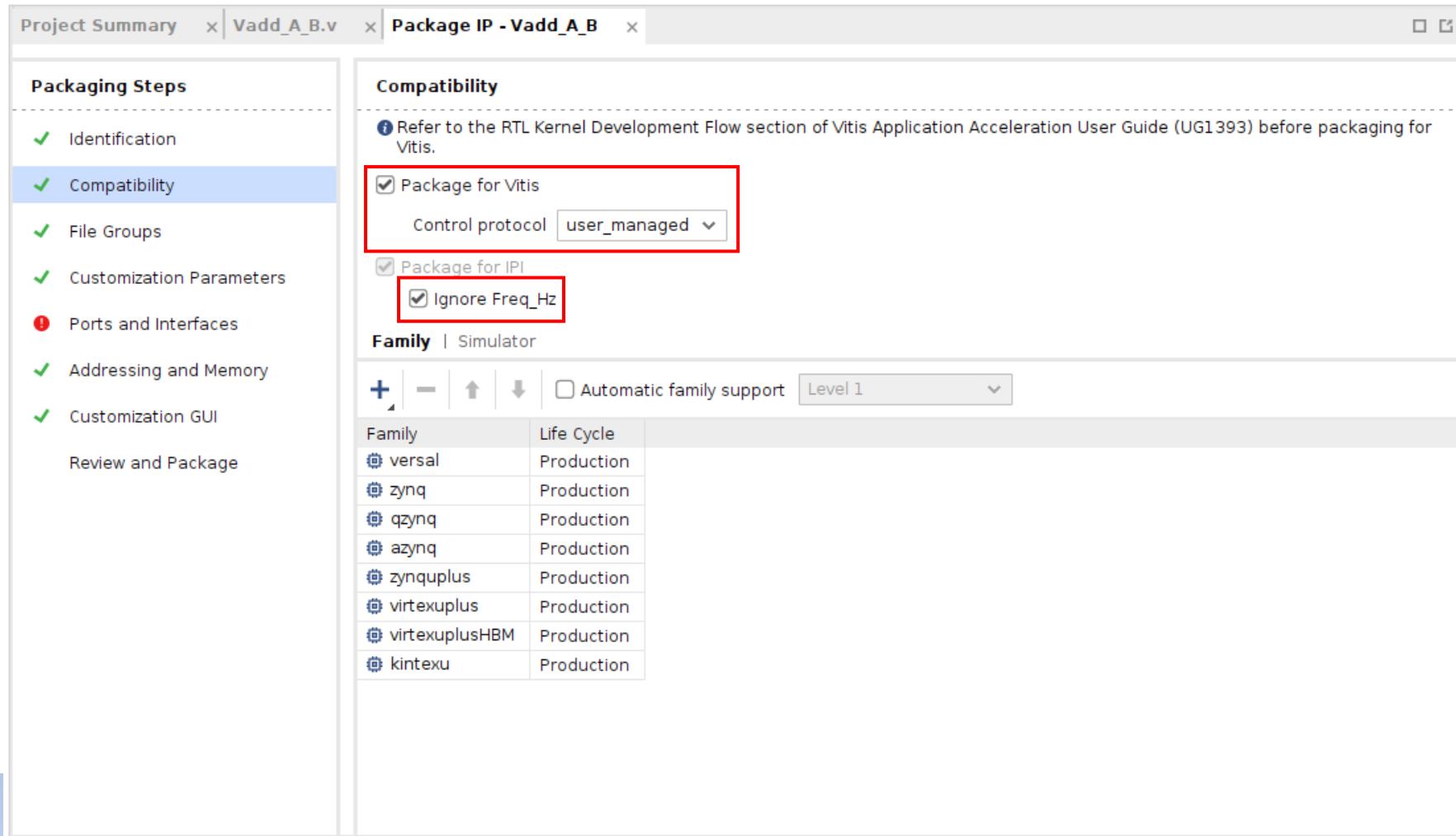
Open the IP Packager

The IP Packager window is shown as in the figure below.



Specify the Control Protocol

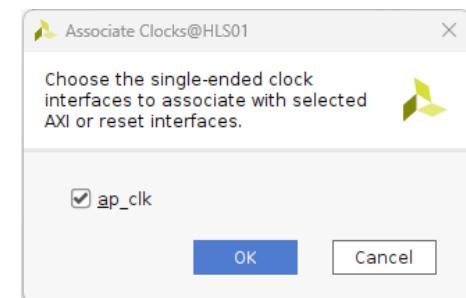
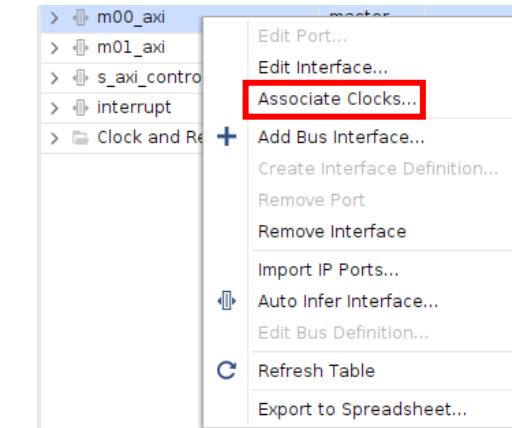
Click "**Compatibility**" on the left side of the window, check the "**Package for Vitis**" option, set the **control protocol** to "**user_managed**," and finally check the "**Ignore Freq_Hz**" option.



Edit Ports and Interfaces

Click “Ports and Interfaces” on the left side of the window, add “Associate Clock” to “m00_axi”, “m01_axi” and “s_axi_control”.

The screenshot shows the Vivado IP Packager interface. On the left, there's a sidebar with "Packaging Steps" including Identification, Compatibility, File Groups, Customization Parameters, Ports and Interfaces (which is selected and highlighted in blue), Addressing and Memory, Customization GUI, and Review and Package. The main area is titled "Ports and Interfaces" and contains a table with columns: Name, Interface Mode, Enablement Dependency, Direction, Driver Value, Size Left, Size Right, Size Left Dependency, Size Right Dependency, and Type Name. The table lists four entries: m00_axi (master), m01_axi (master), s_axi_control (slave), and interrupt (master). Below the table is a section for "Clock and Reset Signals".



Add Control Registers and Address Offsets

Click “Addressing and Memory” on the left side of the window and add four “Register”. (USER_CTRL, scalar00, A, B).

The screenshot shows the Xilinx IP Configurator interface with the following details:

- Project Summary:** Vadd_A_B.v
- Packaging Steps:** Identification, Compatibility, File Groups, Customization Parameters, Ports and Interfaces, **Addressing and Memory** (selected), Customization GUI, Review and Package.
- Addressing and Memory Tab:** Shows an **Address Blocks** table with one entry: **s_axi_control**.

Name	Display Name	Description	Base Address	Range	Range Dependency	Width
s_axi_control	s_axi_control			0x1000	pow(2,(C_S_AXI_CONTROL_ADDR_WIDTH - 1) - 0 + 1)	32
- Address Blocks Table:** Shows the **reg0** entry.

Name	Display Name	Description	Base Address	Range	Range Dependency	Width	Usage
reg0	reg0		0x0	0x1000	pow(2,(C_S_AXI_CONTROL_ADDR_WIDTH - 1) - 0 + 1)	32	register
- Registers Table:** Shows four entries: USER_CTRL, scalar00, A, and B.

Name	Display Name	Description	Address Offset	Size
USER_CTRL			0	1
scalar00			0	1
A			0	1
B			0	1
- Context Menu:** A context menu is open over the **reg0** row in the Address Blocks table, with the **Add Register...** option highlighted.

Add Control Registers and Address Offsets

Set the “**Description**”, “**Address Offset**” and “**Size**” of the register,
the Description is optional, the Address Offset and Size are required.

The screenshot shows the Xilinx IP Configurator interface for a Vadd_A_B IP core. The left sidebar lists packaging steps: Identification, Compatibility, File Groups, Customization Parameters, Ports and Interfaces, Addressing and Memory (which is selected), and Customization GUI. The main area is titled "Addressing and Memory". It contains three tables: "Addressing and Memory", "Address Blocks", and "Registers". The "Registers" table is highlighted with a red border and contains the following data:

Name	Display Name	Description	Address Offset	Size
USER_CTRL		Control Signals	0x000	32
scalar00		Scalar Values	0x010	32
A		pointer arguments	0x018	64
B		pointer arguments	0x024	64

The "Description" column is labeled "optional" in red at the bottom, while the "Address Offset" and "Size" columns are labeled "required" in blue at the bottom.

Add Control Registers and Address Offsets

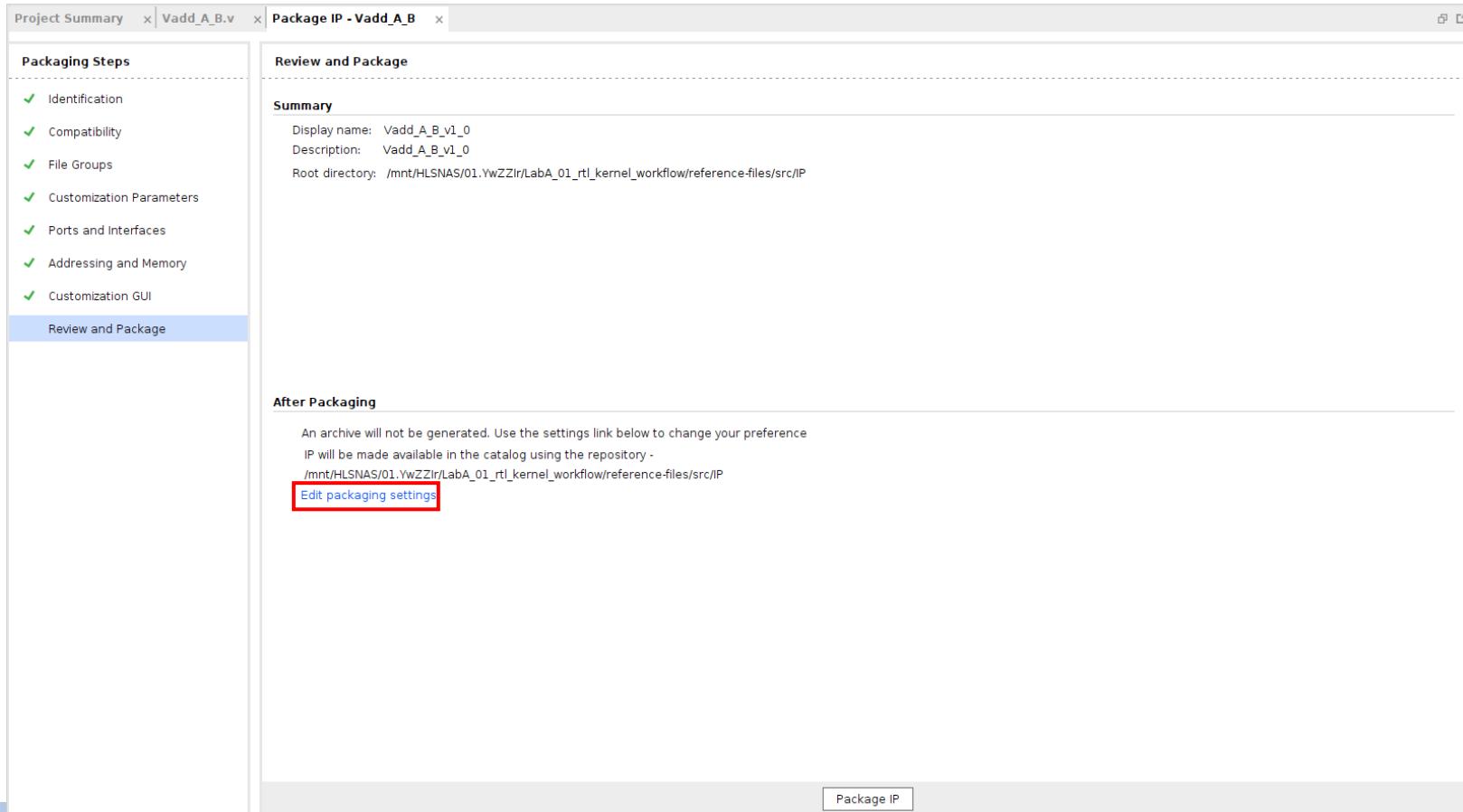
Add **Register Parameter** for A and B register, the parameter's name are both “**ASSOCIATE_BUSIF**”.

Add Value “**m00_axi**” for register A’s parameter and Value “**m01_axi**” for register B’s parameter.

The screenshot shows a software interface for managing memory maps and registers. On the left, a sidebar lists items A and B under "Memory Maps". A context menu is open over item A, with the "Add Register Parameter..." option highlighted by a red box. Below this, a modal dialog titled "Add Register Parameter" also has "ASSOCIATED_BUSIF" highlighted by a red box. The main window displays the "Addressing and Memory" configuration for register A. It includes sections for "Address Blocks" (containing reg0) and "Registers" (containing USER_CTRL, scalar00, and A). In the "Registers" section, the "A" row has "Value" set to "m00_axi". Another "Addressing and Memory" window is shown for register B, where the "Value" for register A is listed as "pointer arguments" with a value of "0x024". Both windows have "ASSOCIATED_BUSIF" highlighted by red boxes in their "Register Parameters" tables.

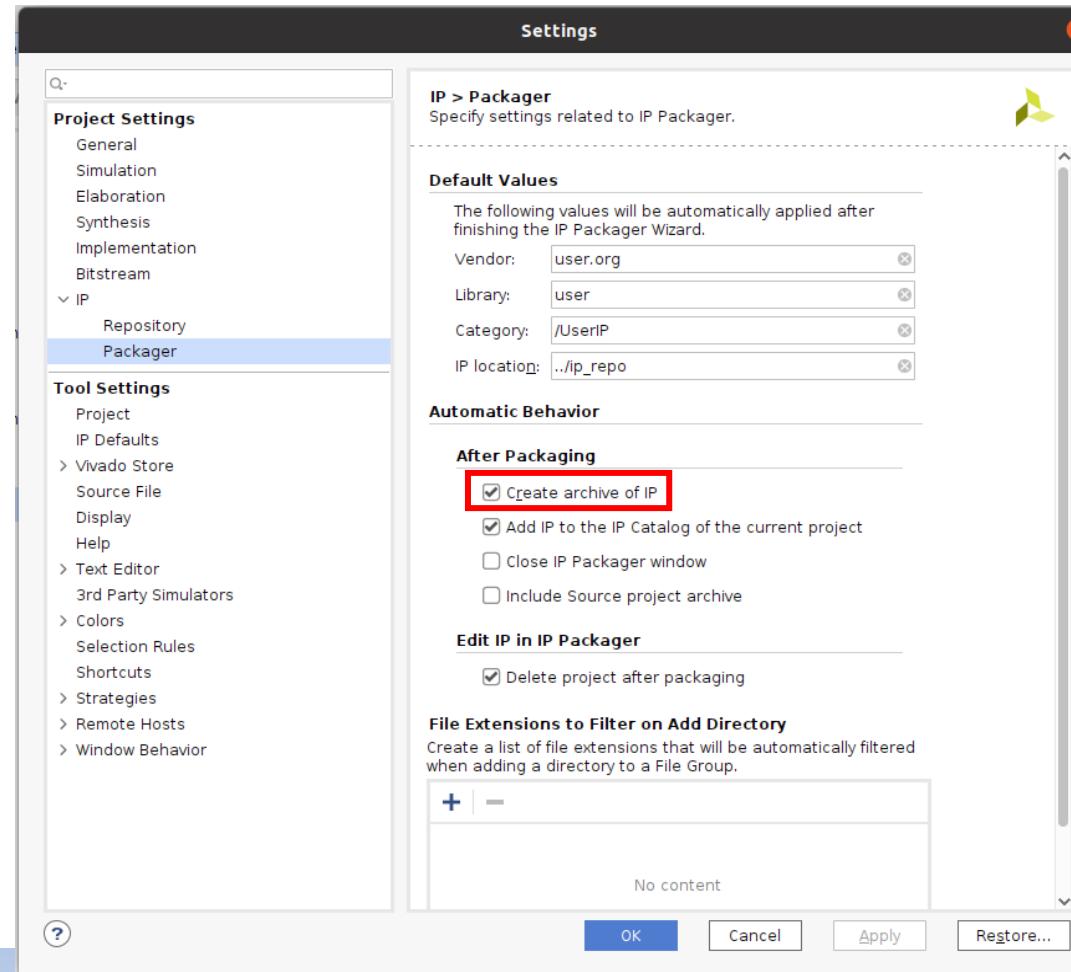
Check Integrity, Assign Properties and Package IP

Click “Review and Package” on the left side of the window and click the “Edit packaging settings”.



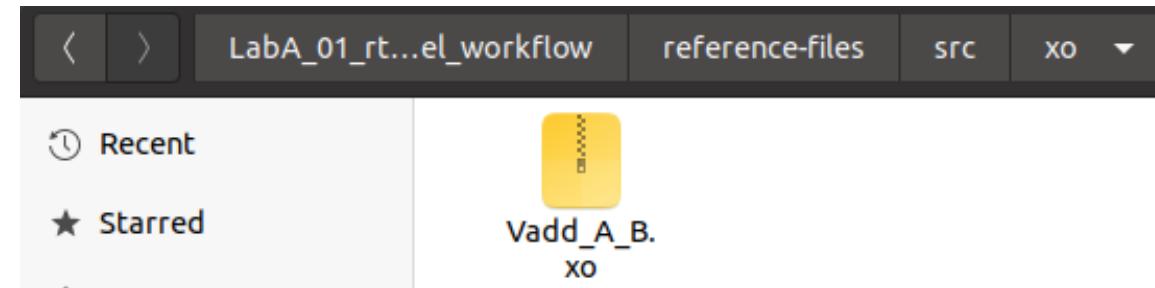
Check Integrity, Assign Properties and Package IP

Check the “**Create archive of IP**” option is select, click the “**OK**” button.
Click the “**Package IP**” button in the previous page to create the IP.



Check Integrity, Assign Properties and Package IP

The IP's (.xo) file path is “./01_rtl_kernel_flow/reference-files/src/xo”.



Host applications interacts with the kernel

1. Setting Up the XRT Native API
2. Specifying the Device ID and Loading the .xclbin file
3. Setting up the kernel and kernel arguments
4. Transferring Data
5. Running the kernel and returning results

Setting Up the XRT Native API

```
// XRT includes
#include "xrt/xrt_bo.h"
#include <experimental/xrt_xclbin.h>
#include "xrt/xrt_device.h"
#include "xrt/xrt_kernel.h"
```

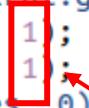
Specifying the Device ID and Loading the .xclbin file

```
// Read settings
    std::string binaryFile = argv[1];
auto xclbin = xrt::xclbin(binaryFile); ←
int device_index = 0;

std::cout << "Open the device " << device_index << std::endl;
auto device = xrt::device(device_index);
std::cout << "Load the xclbin " << binaryFile << std::endl;
auto uuid = device.load_xclbin(binaryFile);
```

Setting up the kernel and kernel arguments

```
std::cout << "Allocate Buffer in Global Memory\n";
//auto boA = xrt::bo(device, vector_size_bytes, krnl.group_id(1)); //Match kernel arguments to RTL kernel
//auto boB = xrt::bo(device, vector_size_bytes, krnl.group_id(2));
auto ip1_boA = xrt::bo(device, vector_size_bytes, 1);
auto ip1_boB = xrt::bo(device, vector_size_bytes, 1);
//auto ip2_boA = xrt::bo(device, vector_size_bytes, 0);
//auto ip2_boB = xrt::bo(device, vector_size_bytes, 1);
//auto ip3_boA = xrt::bo(device, vector_size_bytes, 0);
//auto ip3_boB = xrt::bo(device, vector_size_bytes, 1);
```



Setting the **memory bank index** to **1** here can cause the simulation to hang.
An explanation will follow after the configuration results are completed.

Transferring Data

```
// Map the contents of the buffer object into host memory
auto bo0_map = ip1_boA.map<int*>();
auto bo1_map = ip1_boB.map<int*>();

std::fill(bo0_map, bo0_map + DATA_SIZE, 0);
std::fill(bo1_map, bo1_map + DATA_SIZE, 0);

// Create the test data
int bufReference[DATA_SIZE];
for (int i = 0; i < DATA_SIZE; ++i) {
    bo0_map[i] = i;
    bo1_map[i] = i;
    bufReference[i] = bo0_map[i] + bo1_map[i]; //Generate check data for validation
}

std::cout << "loaded the data" << std::endl;
uint64_t buf_addr[2];
// Get the buffer physical address
buf_addr[0] = ip1_boA.address();
buf_addr[1] = ip1_boB.address();

// Synchronize buffer content with device side
std::cout << "synchronize input buffer data to device global memory\n";
ip1_boA.sync(XCL_BO_SYNC_BO_TO_DEVICE);
ip1_boB.sync(XCL_BO_SYNC_BO_TO_DEVICE);
```

Running the kernel and returning results

```
//std::cout << "Execution of the kernel\n";
//auto run = krnl(DATA_SIZE, boA, boB); //DATA_SIZE=size
//run.wait();

std::cout << "INFO: Setting IP Data" << std::endl;
std::cout << "Setting Register \"A\" (Input Address)" << std::endl;
ip1.write_register(A_OFFSET, buf_addr[0]);
ip1.write_register(A_OFFSET + 4, buf_addr[0] >> 32);

std::cout << "Setting Register \"B\" (Input Address)" << std::endl;
ip1.write_register(B_OFFSET, buf_addr[1]);
ip1.write_register(B_OFFSET + 4, buf_addr[1] >> 32);

uint32_t axi_ctrl = IP_START;
std::cout << "INFO: IP Start" << std::endl;
//axi_ctrl = IP_START;
ip1.write_register(USER_OFFSET, axi_ctrl);

// Wait until the IP is DONE
int i = 0;
//axi_ctrl = 0;
while (axi_ctrl != IP_IDLE) {
//while ((axi_ctrl & IP_IDLE) != IP_IDLE) {
    axi_ctrl = ip1.read_register(USER_OFFSET);
    i = i + 1;
    std::cout << "Read Loop iteration: " << i << " and Axi Control = " << axi_ctrl << "\n";
    if (i > 100000) {
        axi_ctrl = IP_IDLE;
        ip1.write_register(USER_OFFSET, axi_ctrl);
    }
}

std::cout << "INFO: IP Done" << std::endl;
// Get the output;
std::cout << "Get the output data from the device" << std::endl;
ip1_boB.sync(XCL_BO_SYNC_BO_FROM_DEVICE);

// Validate results
if (std::memcmp(bo1_map, bufReference, DATA_SIZE))
    throw std::runtime_error("Value read back does not match reference");
```

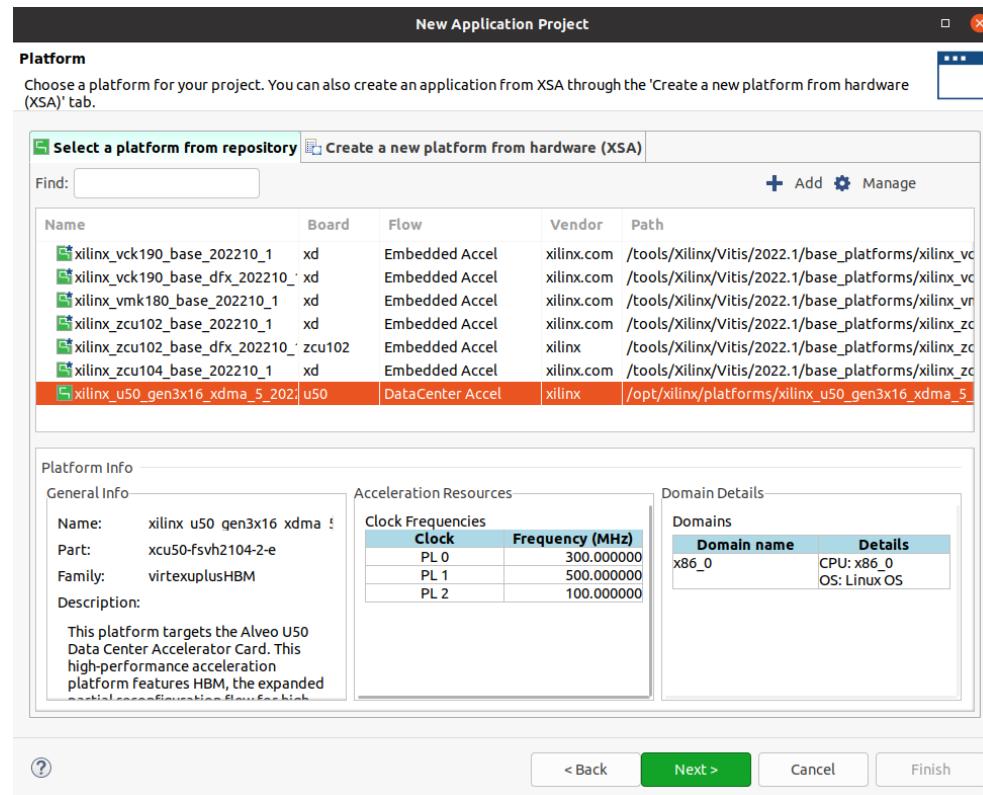
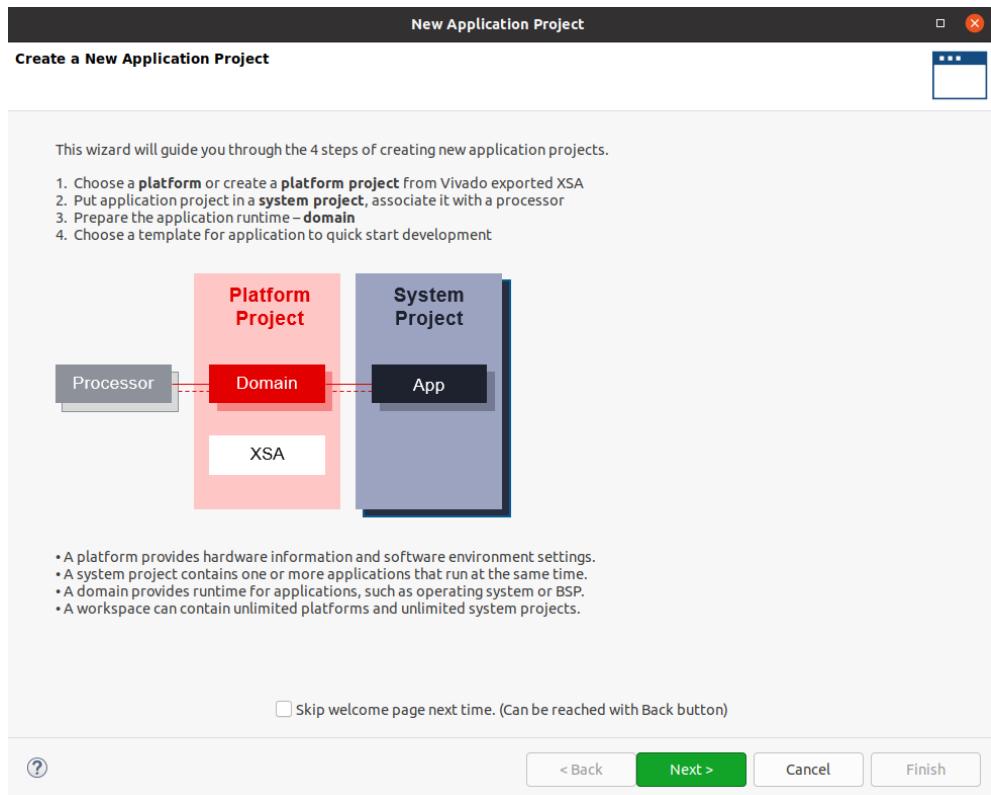
Using the RTL kernel in a Vitis IDE project

1. Create a new Vitis project
2. Add the Hardware Kernel (.xo)
3. Build the Project

Create a new Vitis project

Open the terminal window and enter the command “**vitis**” to launch the Vitis IDE.

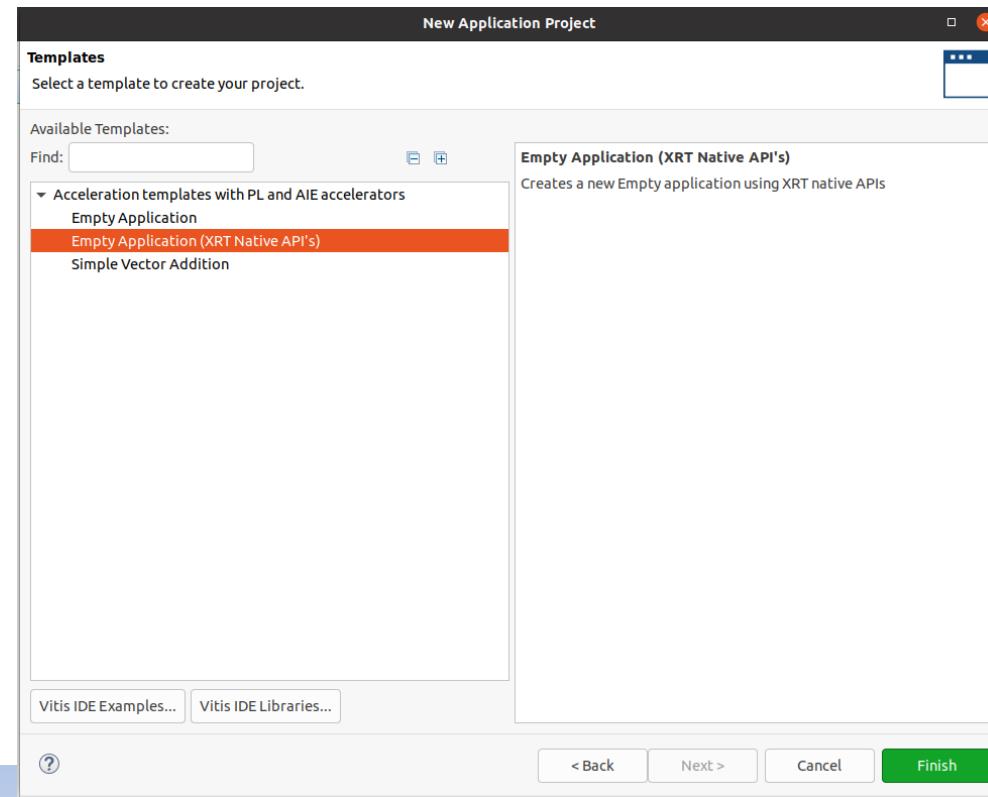
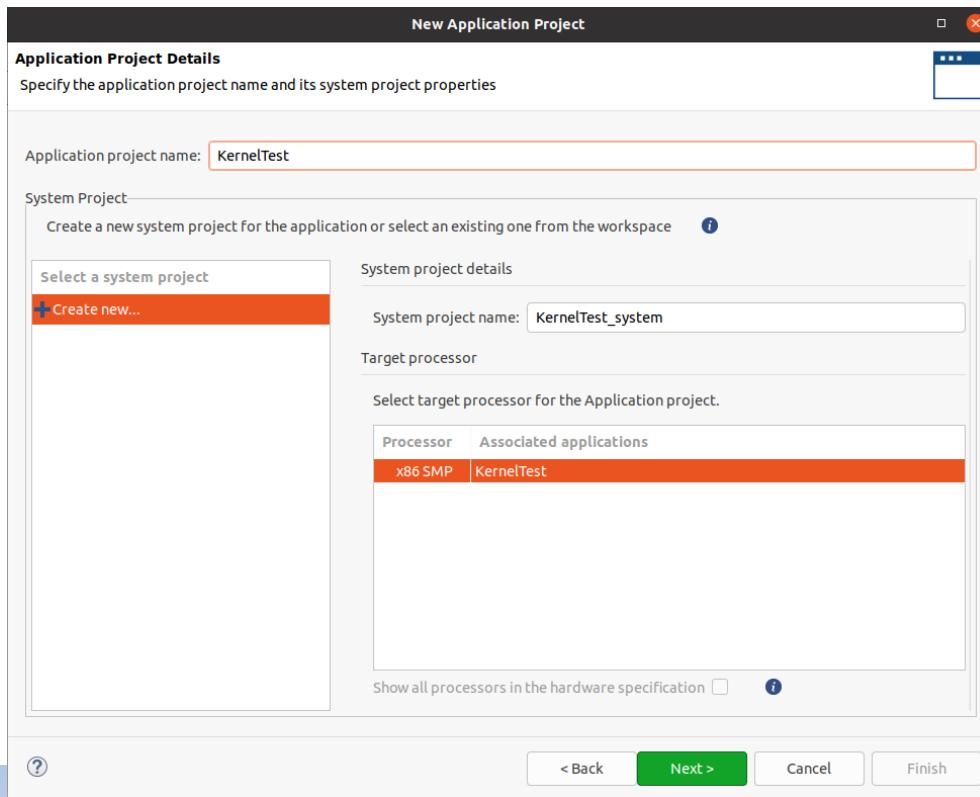
Click the “**Next**” to create a application project and select “**xilinx_u50_gen3x16_xdma_5_202210_1**”.



Create a new Vitis project

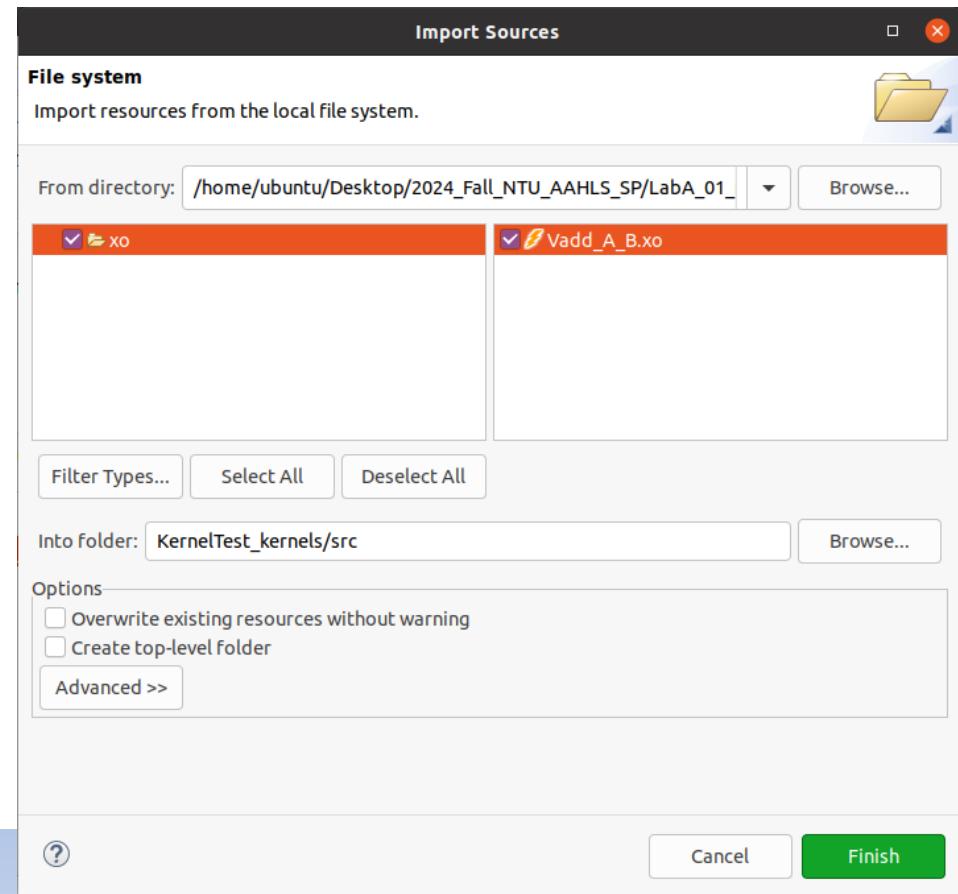
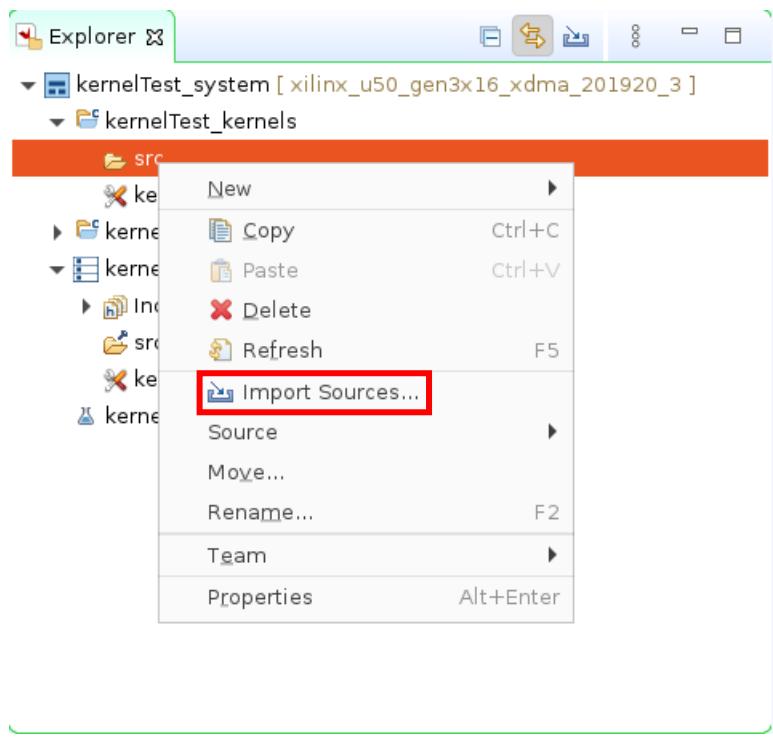
Enter the "**Application project name**" and select the option "**Empty Application (XPT Native API's)**".

Notice: If the option "**Empty Application**" is selected, the **g++ compiler** will be unable to recognize the **XRT APIs** during the hardware emulation host phase.



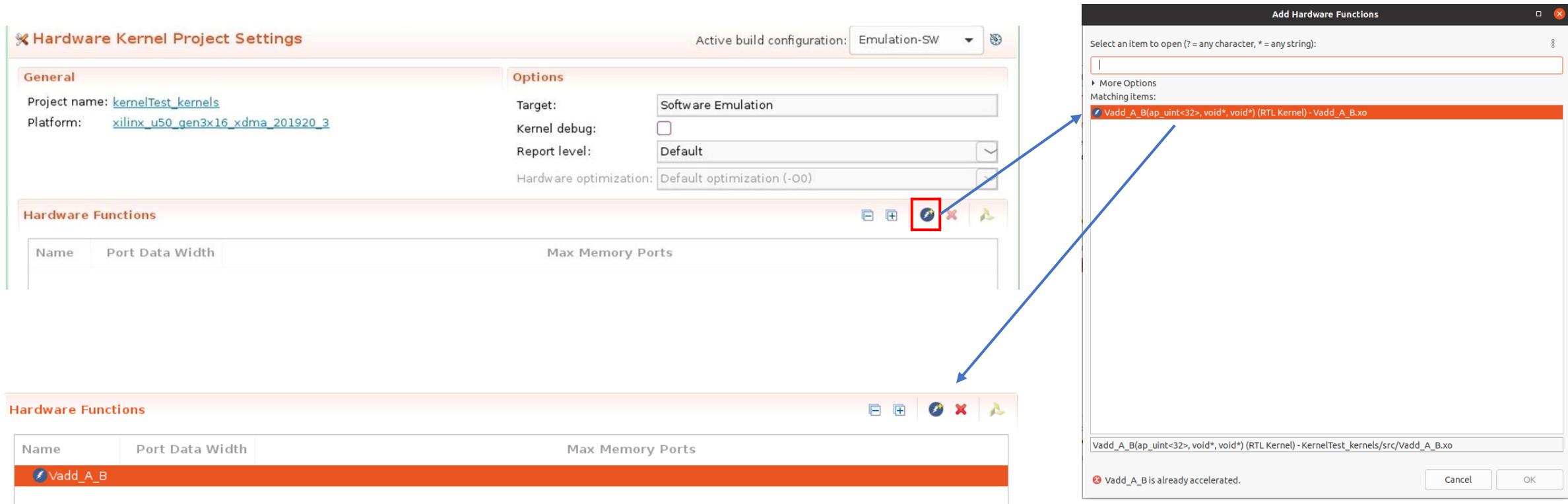
Add the Hardware Kernel

Find out "Explorer" in the top left corner of the window, right-click on **src**, and select "**Import Source...**". Navigate to the corresponding path where the Vivado packaged IP (xo file) is located, and click the "**Finish**" button.



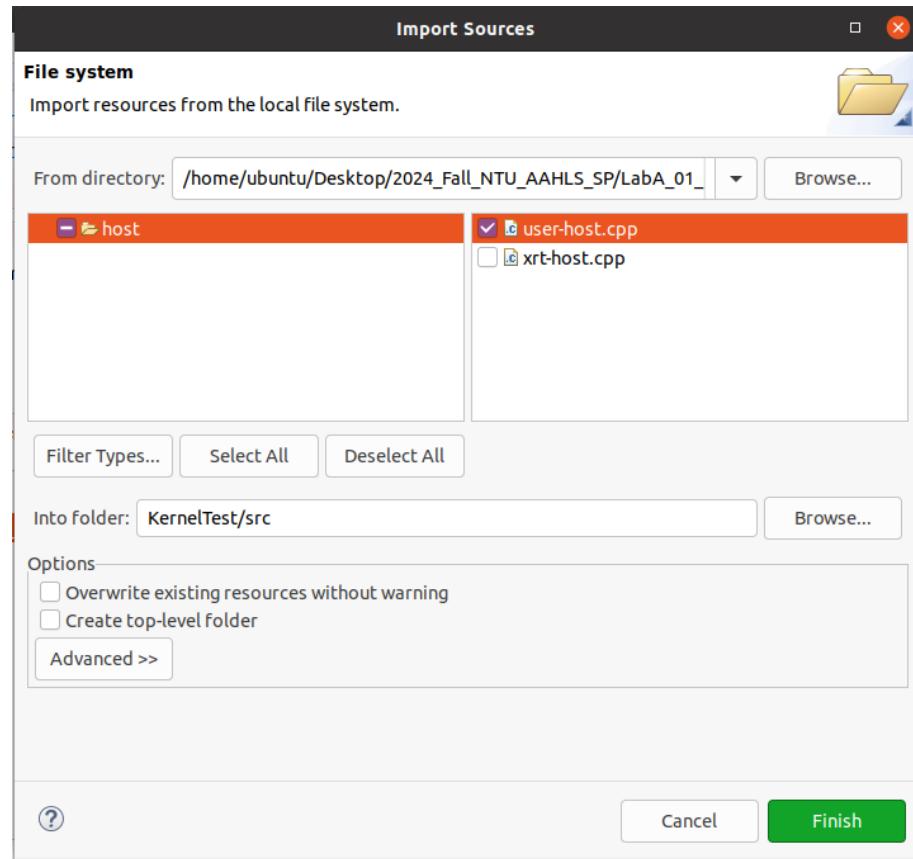
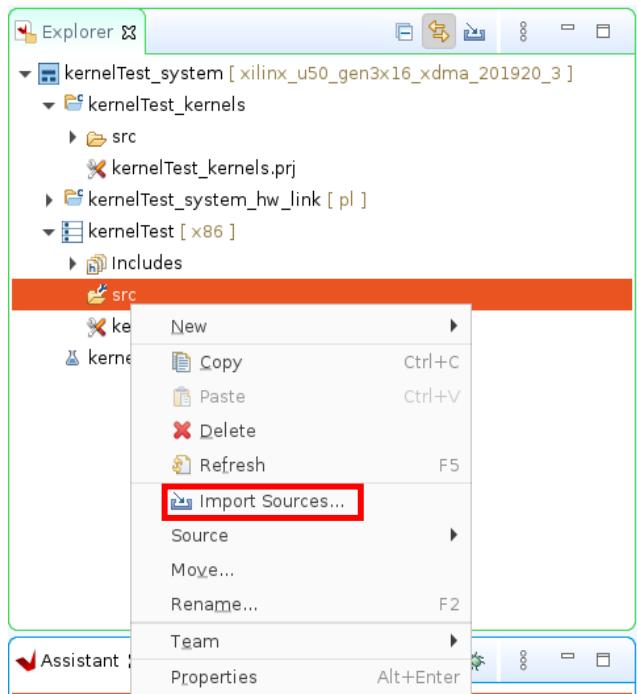
Add the Hardware Kernel

Add the “Hardware Functions” to the project.



Add the Hardware Kernel

Find out "Explorer" in the top left corner of the window, right-click on **src**, and select "**Import Source...**".
Navigate to the corresponding path where the host code is located, and click the "**Finish**" button.



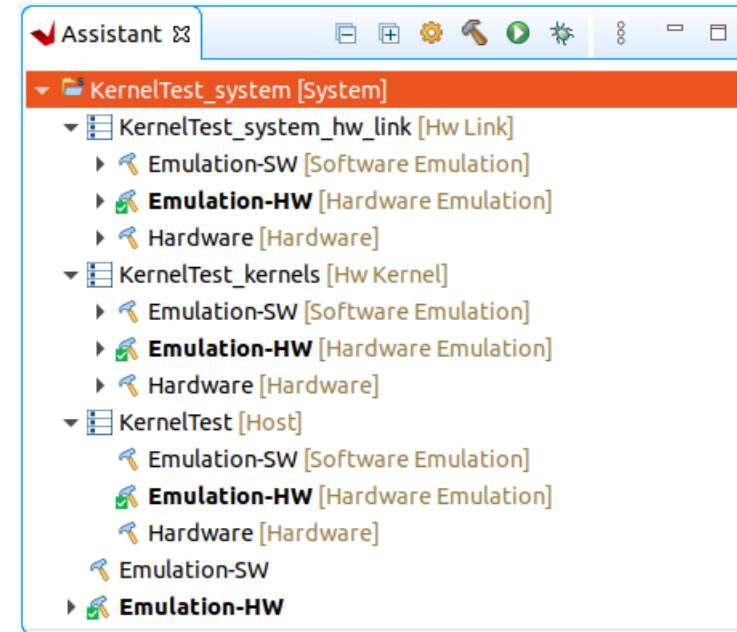
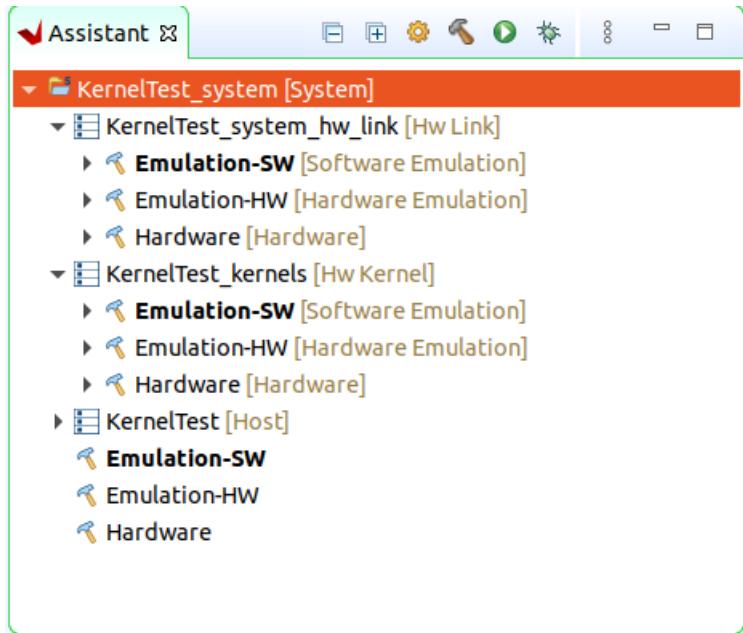
Build the Project

Since the RTL kernel does not support software emulation, we just run the hardware emulation.

The screenshot shows the 'Hardware Kernel Project Settings' window. At the top right, the 'Active build configuration:' dropdown is set to 'Emulation-SW'. A red box highlights the 'Emulation-HW' option in the dropdown menu. The 'General' tab shows 'Project name: rtl_kernel_workflow_kernels' and 'Platform: xilinx u50_gen3x16_xdma_5_202210_1'. The 'Options' tab includes settings for 'Target' (Software Emulation), 'Kernel debug' (unchecked), 'Report level' (Default), and 'Hardware optimization' (Default optimization (-O0)). The 'Hardware Functions' tab lists a single entry: 'Name' Vadd_A_B, 'Port Data Width' (indicated by a lightning bolt icon), and 'Max Memory Ports' (indicated by a green leaf icon). A toolbar at the bottom right contains icons for file operations.

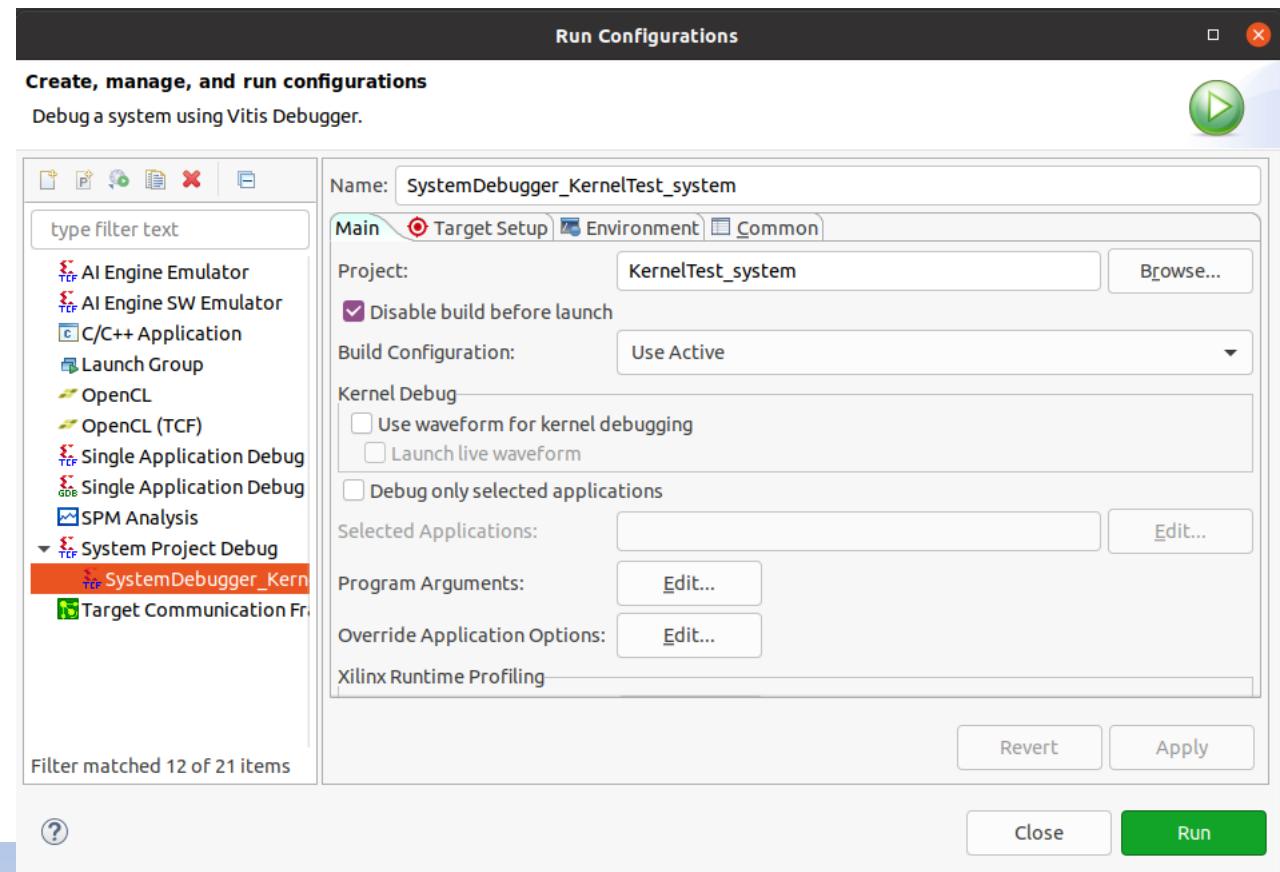
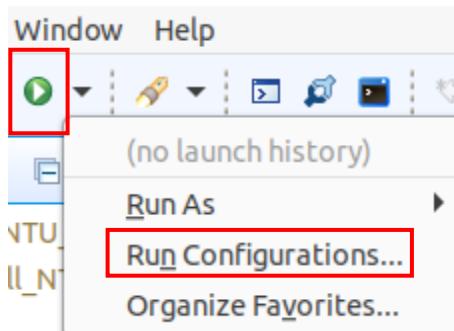
Build the Project

Run the hardware emulation in **kernel -> hw_link -> host -> system** sequence.
After each process done, the “**Assistant**” window will be like the figure shown in right side.



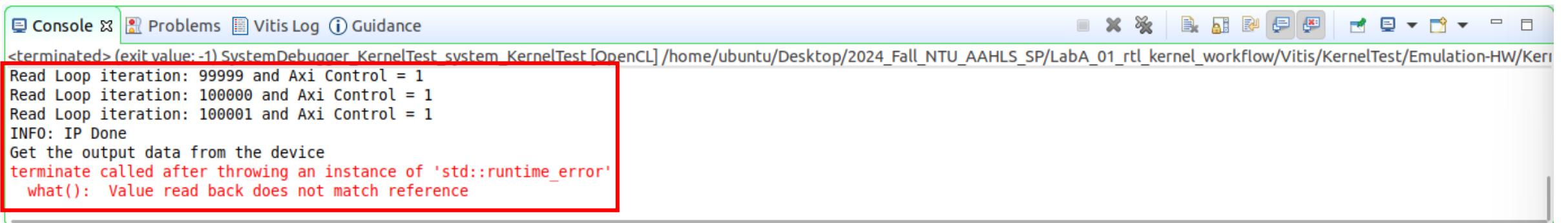
Build the Project

Select the “**Run Configurations...**”, a window like the one shown on the right side will pop up.
In the Program Argument part, the Vitis IDE can automatically search and include the xclbin file when the
Automatically update arguments is enabled.



Build the Project

If the **memory bank index** is set to **1**, the following phenomenon can be observed:
the configuration process will hang until it exceeds **100,000 iterations**.



```
Console ✘ Problems Vitis Log Guidance
<terminated> (exit value: -1) SystemDebugger_KernelTest_system_KernelTest [OpenCL] /home/ubuntu/Desktop/2024_Fall_NTU_AAHLSP/LabA_01 rtl_kernel_workflow/Vitis/KernelTest/Emulation-HW/Ker
Read Loop iteration: 99999 and Axi Control = 1
Read Loop iteration: 100000 and Axi Control = 1
Read Loop iteration: 100001 and Axi Control = 1
INFO: IP Done
Get the output data from the device
terminate called after throwing an instance of 'std::runtime_error'
what(): Value read back does not match reference
```

Build the Project

Solution: Change the memory bank index in the host code for allocating the buffer in Global Memory to 0.

This is based on the information provided by **xclbin.info**,
which indicates the HBM index we are using and which memory the Kernel's registers are utilizing.

```
std::cout << "Allocate Buffer in Global Memory\n";
//auto boA = xrt::bo(device, vector_size_bytes, krnl.group_id(1)); //Match kernel arguments to RTL kernel
//auto boB = xrt::bo(device, vector_size_bytes, krnl.group_id(2));
auto ip1_boA = xrt::bo(device, vector_size_bytes, 0);
auto ip1_boB = xrt::bo(device, vector_size_bytes, 0);
//auto ip2_boA = xrt::bo(device, vector_size_bytes, 0);
//auto ip2_boB = xrt::bo(device, vector_size_bytes, 1);
//auto ip3_boA = xrt::bo(device, vector_size_bytes, 0);
//auto ip3_boB = xrt::bo(device, vector_size_bytes, 1);
```

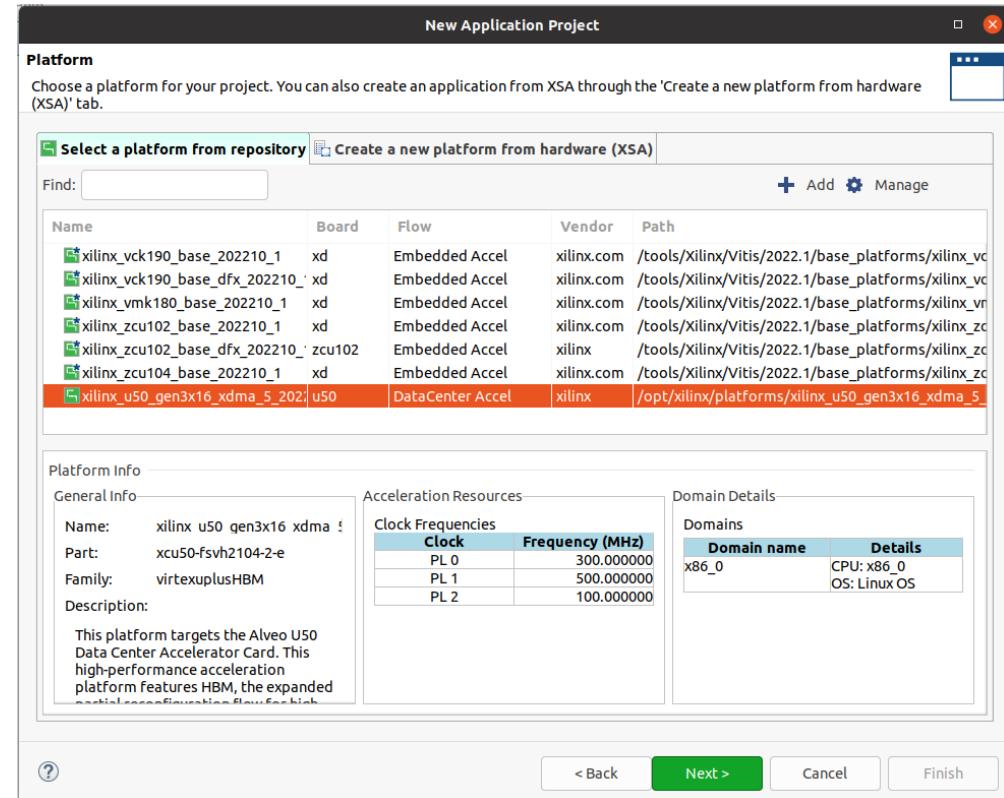
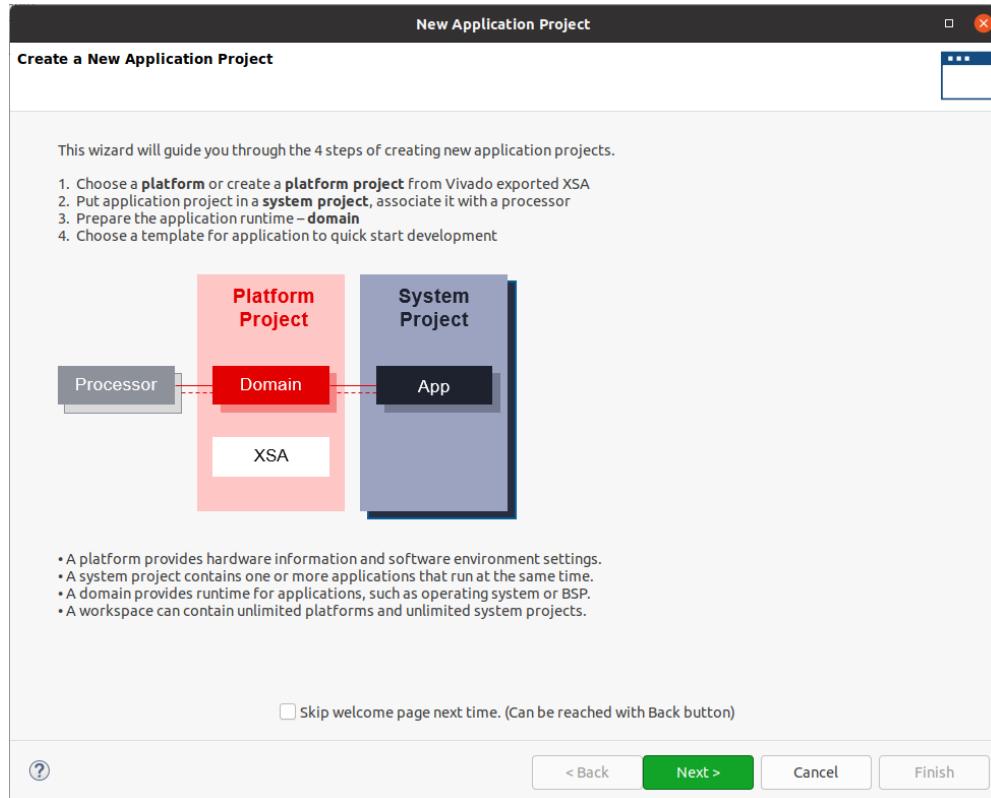
66	Memory Configuration	353	-----
67	-----	354	Instance: Vadd_A_B_1
68	Name: HBM[0]	355	Base Address: 0x1c001000
69	Index: 0	356	Argument: scalar00
70	Type: MEM_DRAM	357	Register Offset: 0x10
71	Base Address: 0x0	358	Port: s_axi_control
72	Address Size: 0x10000000	359	Memory: <not applicable>
73	Bank Used: Yes	360	
74		361	Argument: A
75	Name: HBM[1]	362	Register Offset: 0x18
76	Index: 1	363	Port: m00_axi
77	Type: MEM_DRAM	364	Memory: HBM[0] (MEM_DRAM)
78	Base Address: 0x10000000	365	
79	Address Size: 0x10000000	366	Argument: B
80	Bank Used: No	367	Register Offset: 0x24
		368	Port: m01_axi
		369	Memory: HBM[0] (MEM_DRAM)
		370	

```
Console Problems Vitis Log Guidance
<terminated> (exit value: 0) SystemDebugger_KernelTest_system_KernelTest [OpenCL] /home/ubuntu/Desktop/2024_Fall_NTU_AAHLSP/LabA_01 rtl_kernel_workflow/Vitis/KernelTest/Emulation-HW/KernelTest (10/23/24, 1:09 PM)
[Console output redirected to file:/home/ubuntu/Desktop/2024_Fall_NTU_AAHLSP/LabA_01 rtl_kernel_workflow/Vitis/KernelTest/Emulation-HW/SystemDebugger_KernelTest_system_KernelTest.launch.log]
argc = 2
argv[0] = /home/ubuntu/Desktop/2024_Fall_NTU_AAHLSP/LabA_01 rtl_kernel_workflow/Vitis/KernelTest/Emulation-HW/KernelTest
argv[1] = /home/ubuntu/Desktop/2024_Fall_NTU_AAHLSP/LabA_01 rtl_kernel_workflow/Vitis/KernelTest/system/Emulation-HW/binary_container_1.xclbin
Open the device 0
Load the xclbin /home/ubuntu/Desktop/2024 Fall NTU AAHLSP/LabA_01 rtl_kernel_workflow/Vitis/KernelTest system/Emulation-HW/binary_container_1.xclbin
INFO: [HW-EMU 01] Hardware emulation runs simulation underneath. Using a large data set will result in long simulation times. It is recommended that a small dataset is used for faster execution. The flow uses approximate
configuring embedded scheduler mode
scheduler config ert(1), dataflow(0), slots(16), cudma(0), cuisr(0), cdma(0), cus(1)
INFO: [HW-EMU 07-0] Please refer the path "/home/ubuntu/Desktop/2024_Fall_NTU_AAHLSP/LabA_01 rtl_kernel_workflow/Vitis/KernelTest/Emulation-HW/.run/20900/hw_em/device0/binary_0/beav_waveform/xsim/simulate.log" for more
Allocate Buffer in Global Memory
loaded the data
synchronize input buffer data to device global memory
INFO: Setting IP Data
Setting Register "A" (Input Address)
Setting Register "B" (Input Address)
INFO: IP Start
Read Loop iteration: 1 and Axi Control = 6
Read Loop iteration: 2 and Axi Control = 4
INFO: IP Done
Get the output data from the device
TEST PASSED
INFO: [HW-EMU 06-0] Waiting for the simulator process to exit
INFO: [HW-EMU 06-1] All the simulator processes exited successfully
INFO: [HW-EMU 07-0] Please refer the path "/home/ubuntu/Desktop/2024_Fall_NTU_AAHLSP/LabA_01 rtl_kernel_workflow/Vitis/KernelTest/Emulation-HW/.run/20900/hw_em/device0/binary_0/beav_waveform/xsim/simulate.log" for more
```

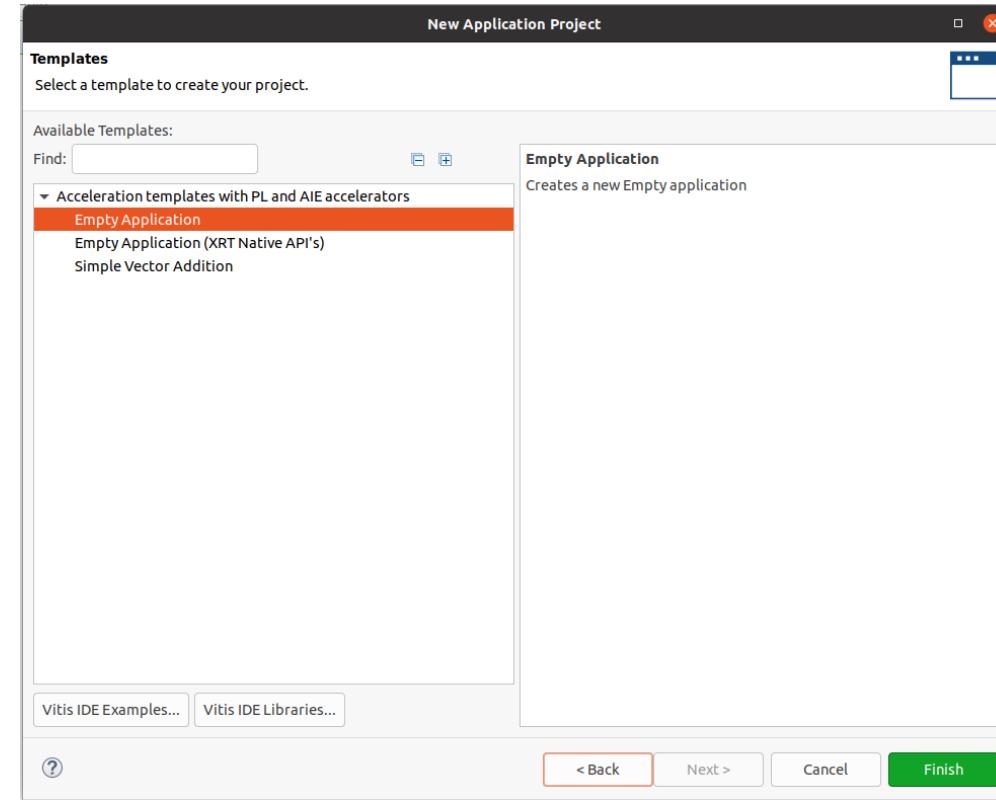
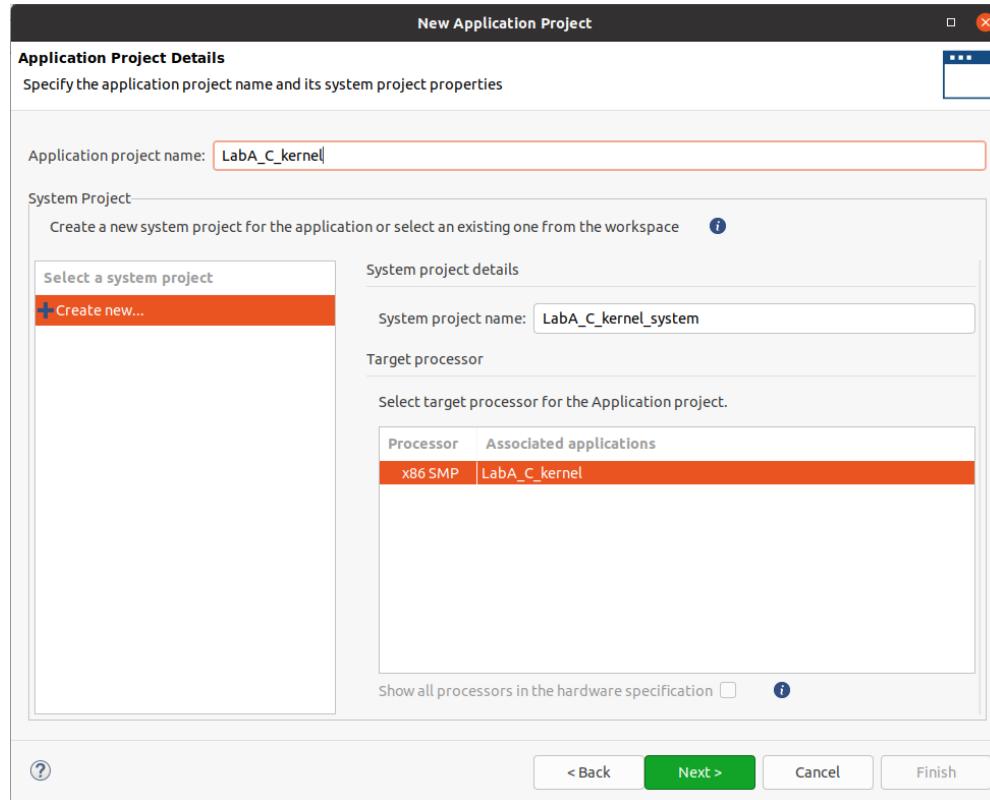
Mixing C++ & RTL Kernels

- Building an Application with C++ Based Kernel
- Building an Application with C++ & RTL Based Kernel

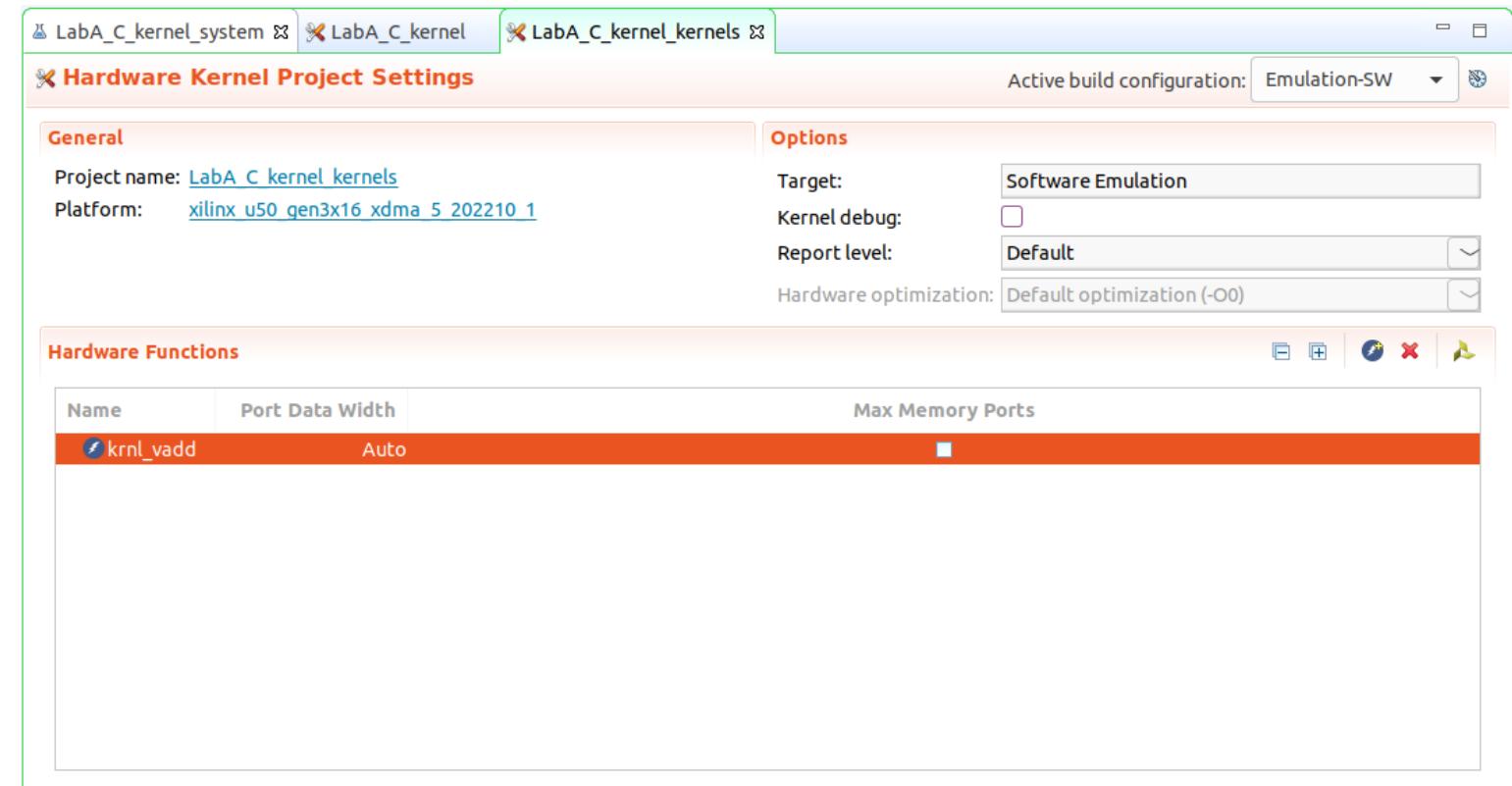
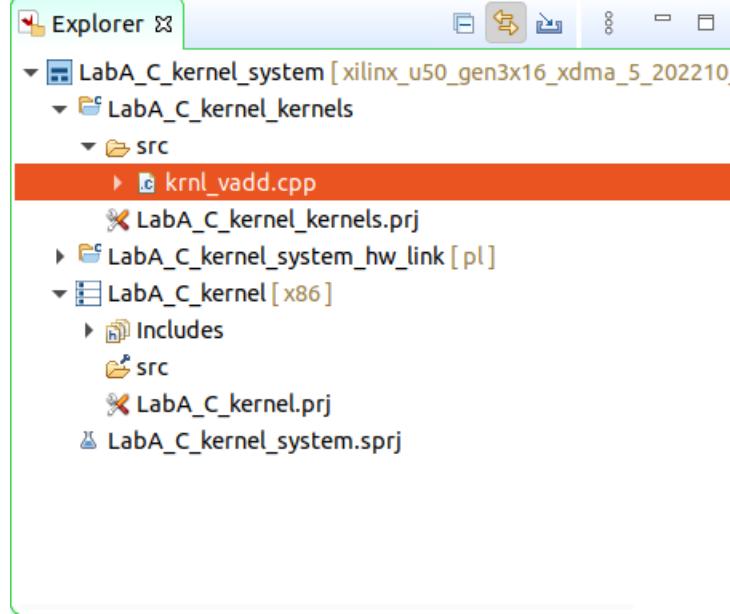
Building an Application with C++ Based Kernel



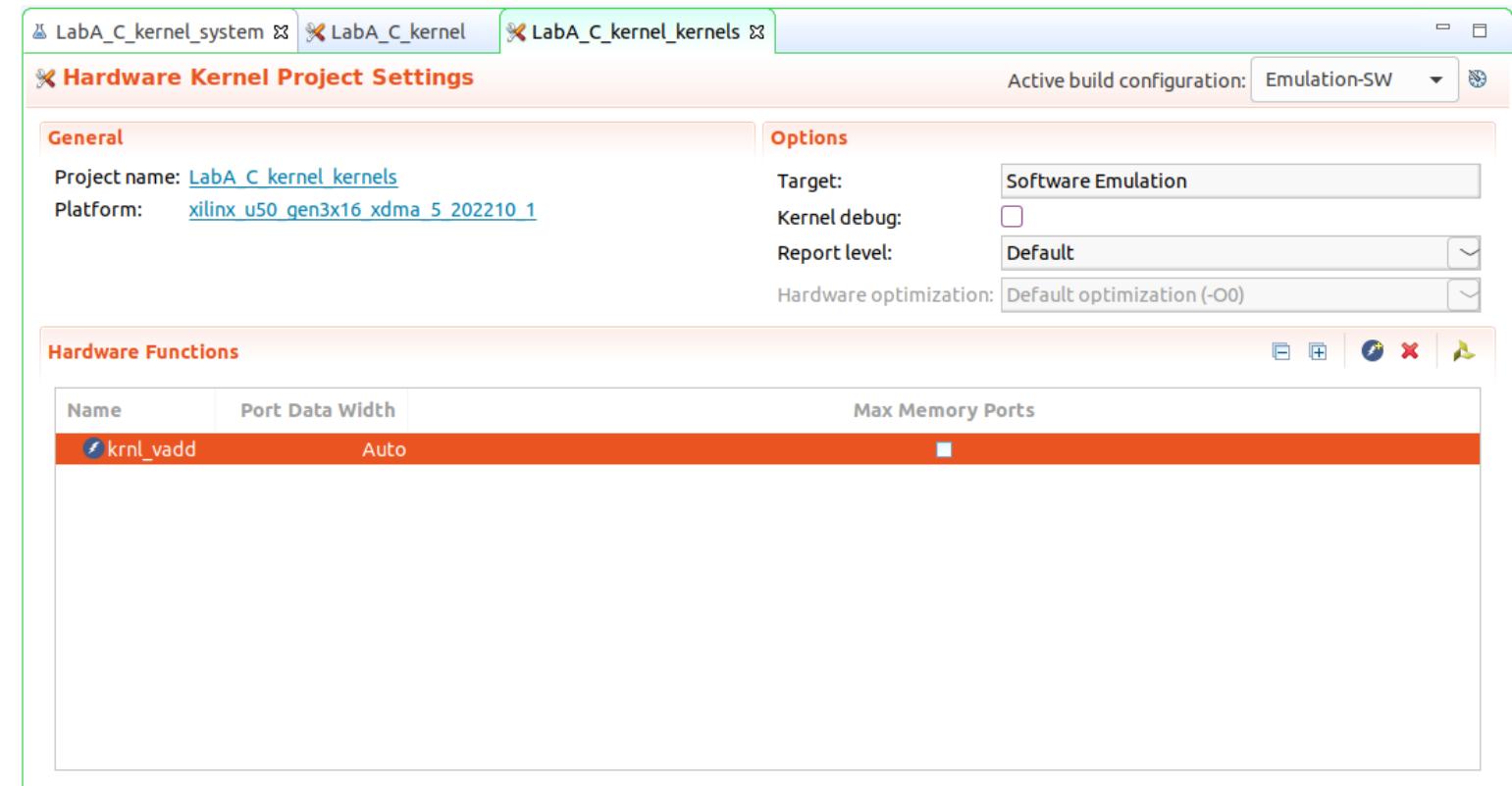
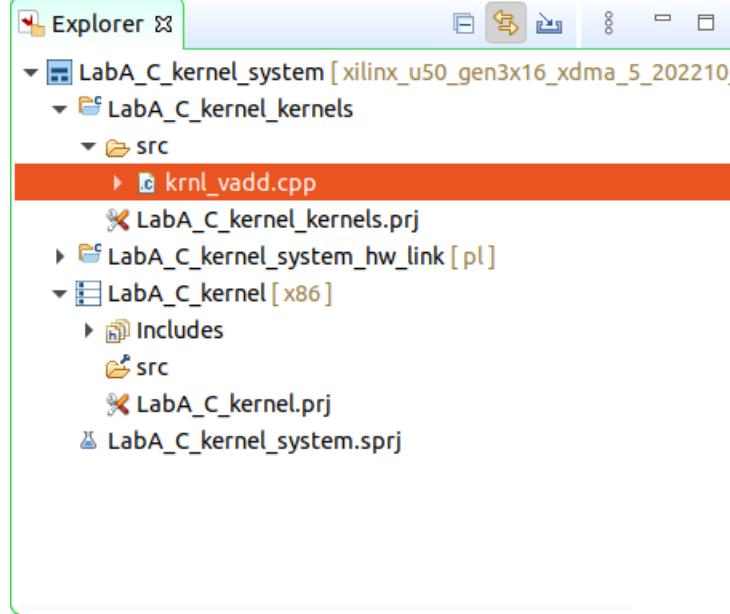
Building an Application with C++ Based Kernel



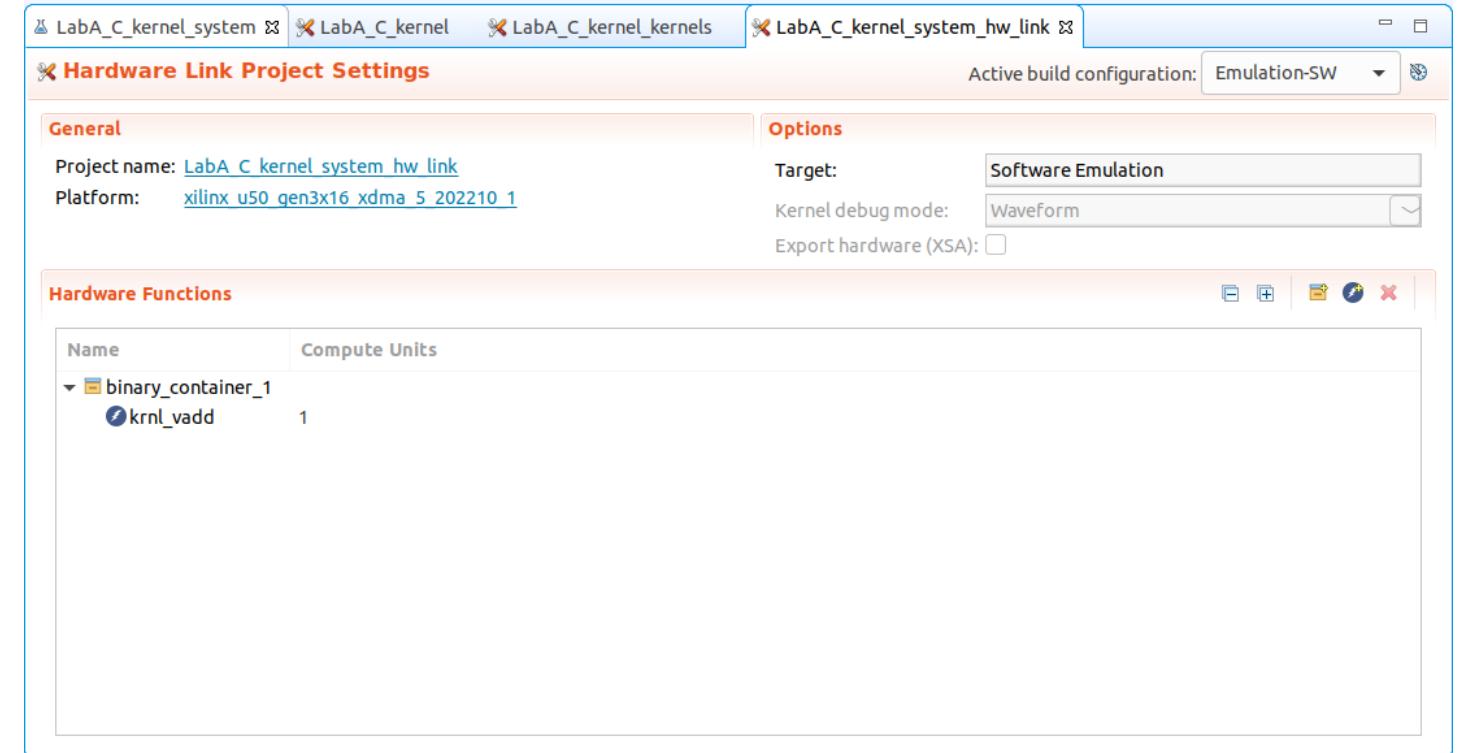
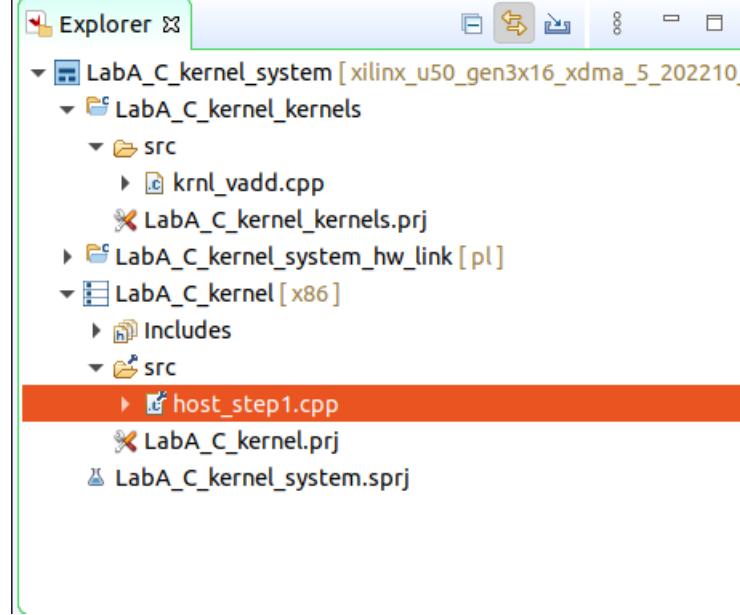
Building an Application with C++ Based Kernel



Building an Application with C++ Based Kernel



Building an Application with C++ Based Kernel



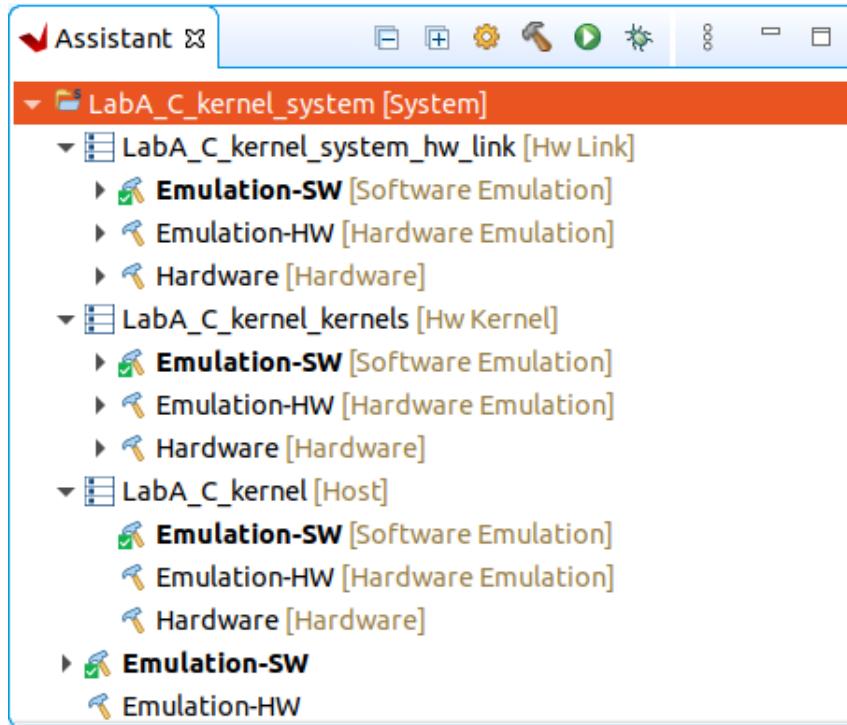
Building an Application with C++ Based Kernel

```
// Creating Program from Binary File
cl::Program::Binaries bins;
bins.push_back({buf,nb});
cl::Program program(context, devices, bins);

// 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");
```

Building an Application with C++ Based Kernel

Building an Application with C++ Based Kernel



Building an Application with C++ Based Kernel

The image shows two dialog boxes from the Vitis IDE:

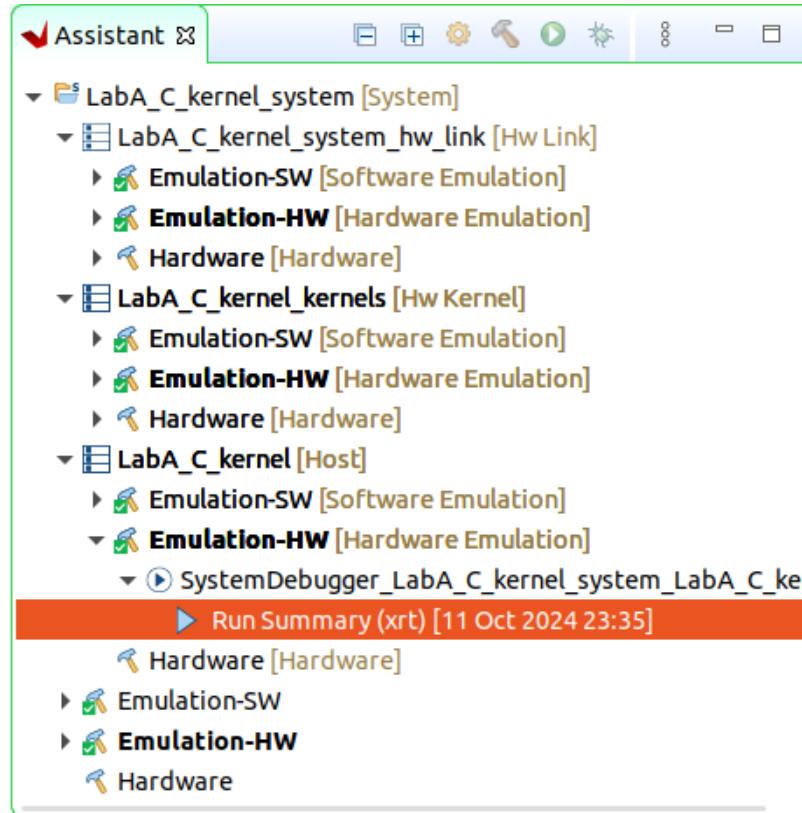
- Run Configurations Dialog:** This dialog is used to create, manage, and run configurations for a system using the Vitis Debugger. It includes fields for Name (SystemDebugger_LabA_C_kernel_system), Project (LabA_C_kernel_system), and Build Configuration (Use Active). It also has sections for Kernel Debug and Selected Applications. A blue arrow points from the "Edit..." button in the Program Arguments section to the "Edit..." button in the "Program Arguments" tab of the "Edit Program Arguments" dialog.
- Edit Program Arguments Dialog:** This dialog allows editing program arguments for the selected application. It shows the Application as "LabA_C_kernel" and the Program Arguments as "\${project_loc:LabA_C_kernel_system}/Emulation-SW/binary_container_1.xclbin". It includes a checkbox for "Automatically add binary container(s) to arguments". A blue arrow points from the "Edit..." button in the "Edit Program Arguments" dialog to the "Program arguments for 'LabA_C_kernel'" dialog.
- Program arguments for 'LabA_C_kernel' Dialog:** This dialog shows the resolved arguments for the selected application. It lists "Xilinx xilinx_u50_gen3x16_xdma_5_202210_1" under Arguments and "Xilinx xilinx_u50_gen3x16_xdma_5_202210_1 \${project_loc:LabA_C_kernel_system}/Emulation-SW/binary_container_1.xclbin" under Resolved Arguments.

Building an Application with C++ Based Kernel

The screenshot shows the Vitis IDE interface. At the top, there is a toolbar with icons for Console, Problems, Vitis Log, and Guidance. Below the toolbar is a terminal window displaying the output of a system debugger command. The output includes information about the platform (Xilinx), kernel loading, and a successful test. At the bottom of the terminal window, it says "device process sw_emu_device done".

Below the terminal window is a timeline trace viewer titled "xrt (Software Emulation)". The timeline shows various API calls and kernel executions. The timeline scale ranges from 240.000000 ms to 255.000000 ms. The API calls shown include "clCreateProgramWithBinary", "cl...clCreate...", "cl...clFinish", and "clEnqueue...". The kernel enqueue "xilinx_u50_gen3x1_1:knl_vadd" is also visible. The left sidebar lists categories such as OpenCL API Calls, Data Transfer (Read, Write, Copy), and Kernel Enqueues.

Building an Application with C++ Based Kernel



Building an Application with C++ Based Kernel

Console Problems Vitis Log Guidance

```
<terminated> (exit value: 0) SystemDebugger_LabA_C_kernel_system_LabA_C_kernel[OpenCL]/home/ubuntu/Desktop/02-mixing-c-rtl-kernels/LabA_C_kernel/LabA_C_kernel/Emulation-HW/LabA_C_kernel(1
[Console output redirected to file:/home/ubuntu/Desktop/02-mixing-c-rtl-kernels/LabA_C_kernel/LabA_C_kernel/Emulation-HW/SystemDebugger_LabA_C_kernel_system_LabA_C_kernel.
No FPGA binary file specified through the command line, using.../binary_container_1.xclbin
Found Platform
Platform Name: Xilinx
Loading: '../binary_container_1.xclbin'
INFO: [HW-EMU 01] Hardware emulation runs simulation underneath. Using a large data set will result in long simulation times. It is recommended that a small dataset is used
configuring dataflow mode with ert polling
scheduler config ert(1), dataflow(1), slots(16), cudma(0), cuisr(0), cdma(0), cus(1)
INFO: [HW-EMU 07-0] Please refer the path "/home/ubuntu/Desktop/02-mixing-c-rtl-kernels/LabA_C_kernel/LabA_C_kernel/Emulation-HW/.run/16410/hw_em/device0/binary_0/behave
TEST WITH ONE KERNEL PASSED
INFO::[ Vitis-EM 22 ] [Time elapsed: 0 minute(s) 22 seconds, Emulation time: 0.333585 ms]
Data transfer between kernel(s) and global memory(s)
krnl_vadd_1:m_axi_gmem-HBM[0] RD = 32.000 KB WR = 0.000 KB
krnl_vadd_1:m_axi_gmem1-HBM[0] RD = 0.000 KB WR = 16.000 KB

INFO: [HW-EMU 06-0] Waiting for the simulator process to exit
INFO: [HW-EMU 06-1] All the simulator processes exited successfully
INFO: [HW-EMU 07-0] Please refer the path "/home/ubuntu/Desktop/02-mixing-c-rtl-kernels/LabA_C_kernel/LabA_C_kernel/Emulation-HW/.run/16410/hw_em/device0/binary_0/behave

```

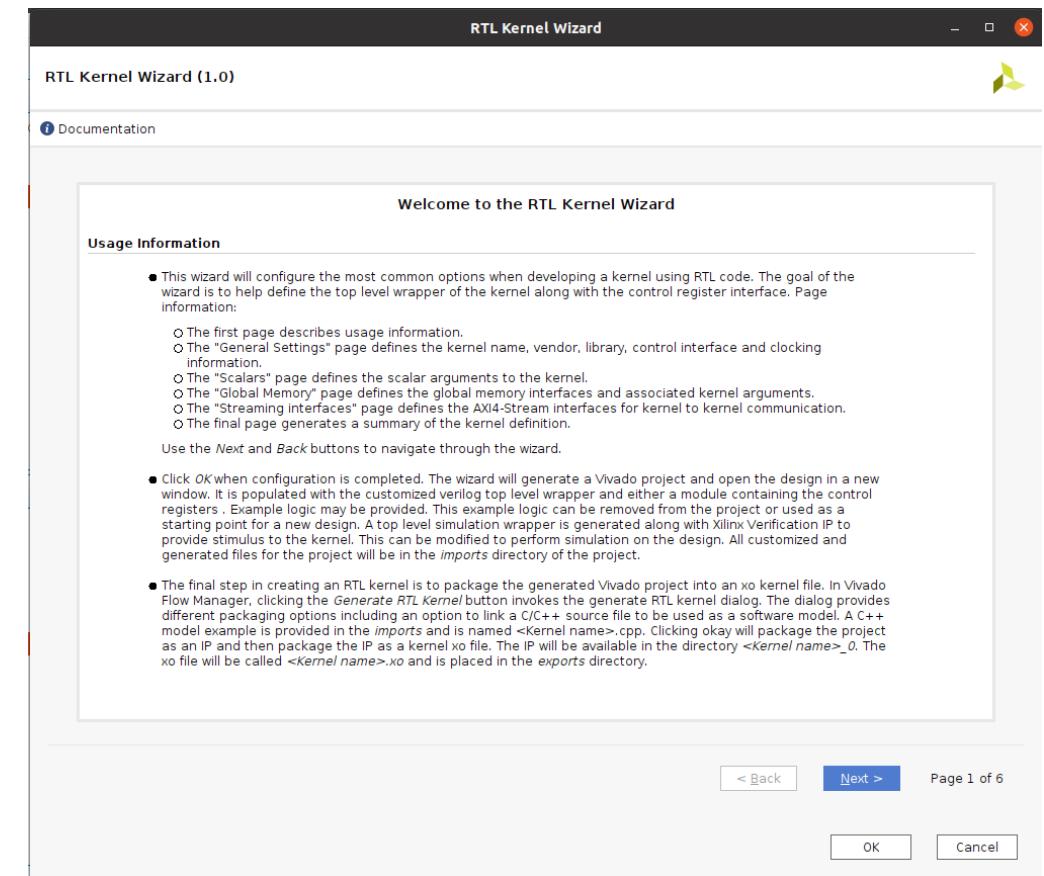
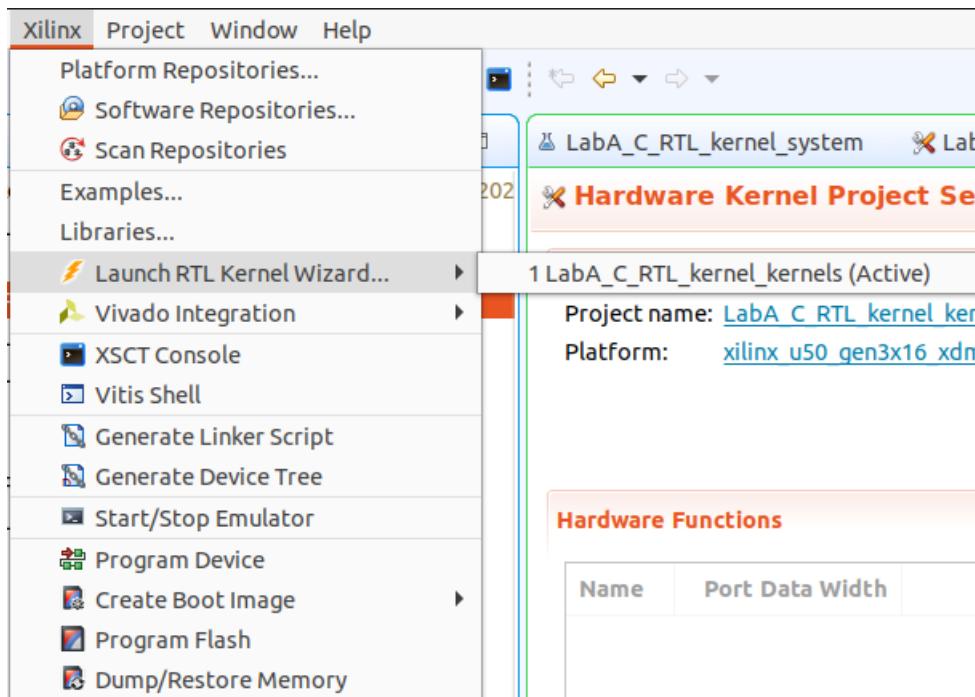
Right Ctrl

Name	Value
OpenCL Host Trace	
OpenCL API Calls	
General	
Queue 0x556b0a03d430	
Data Transfer	
Read	
Parallel Read 1	
Write	
Parallel Write 1	
Copy	00
Kernel Enqueues	
xilinx_u50_gen3x1...iner_1:kml_vadd_1	
Kernel Enqueue 1	
xilinx_u50_gen3x16_xdma_5_202210_1	
binary_container_1.xclbin	
Compute Unit krnl_vadd_1	
Executions	
Parallel Executions 1	
m_axi_gmem-HBM[0] (a b)	
Read Channel	
Write Channel	00
m_axi_gmem1-HBM[0] (a b c)	
Read Channel	
Write Channel	00

The timeline visualization shows the execution of OpenCL API calls and kernel executions over a period of 30.244 seconds. Key events include 'clCreateProgramWithBinary' at ~5s, 'clFinish' at ~25s, and multiple kernel executions for 'krnl_vadd_1' starting around 15s. Memory transfers are shown as horizontal bars for data reads and writes between host memory and device memory (HBM).

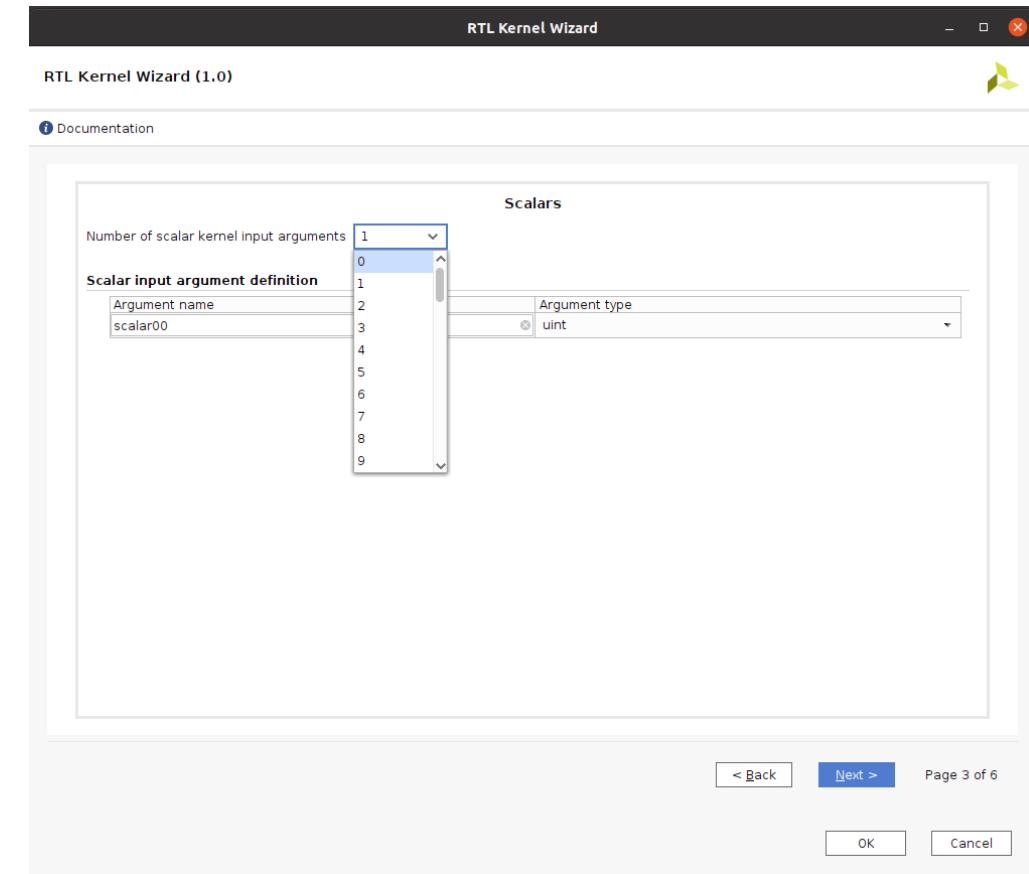
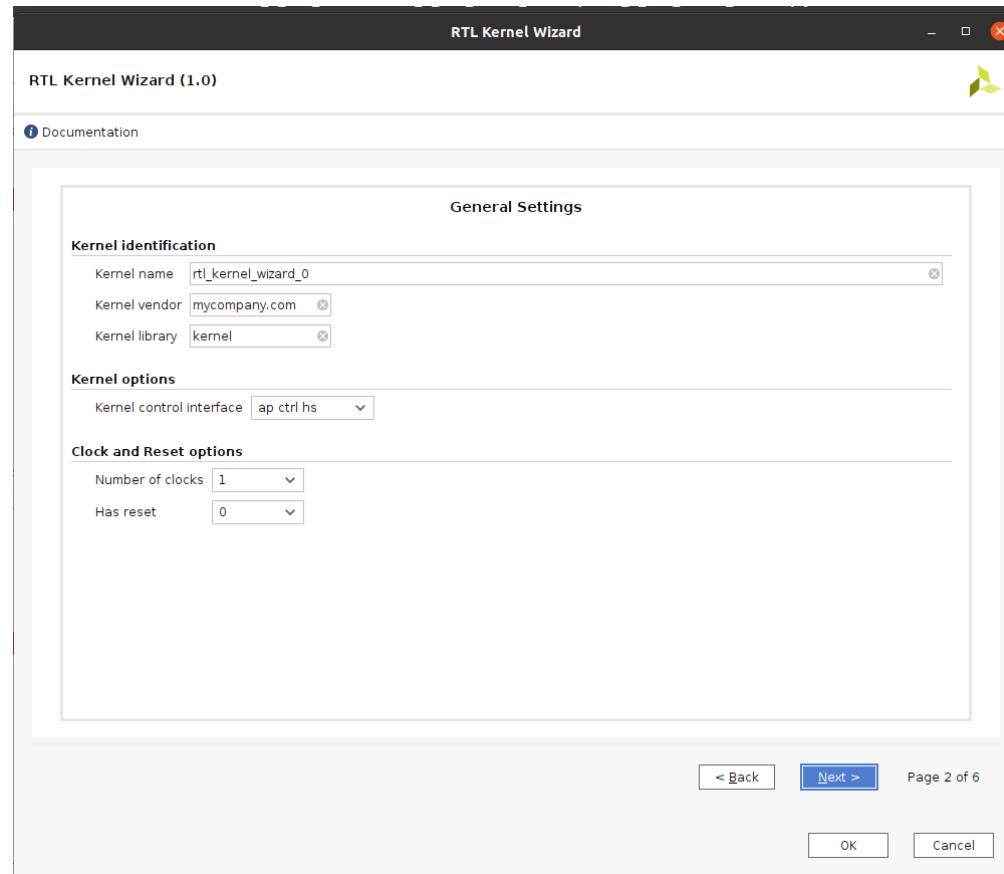
Building an Application with C++ & RTL Based Kernel

Find out "Xilinx" in the top menu and select the "**Launch RTL Kernel Wizard...**" option.
A window similar to the one shown on the right side will pop up.



Building an Application with C++ & RTL Based Kernel

Set the number of kernel input arguments to 0, other option use default settings.



Building an Application with C++ & RTL Based Kernel

The image displays three sequential screens from the RTL Kernel Wizard (1.0) application.

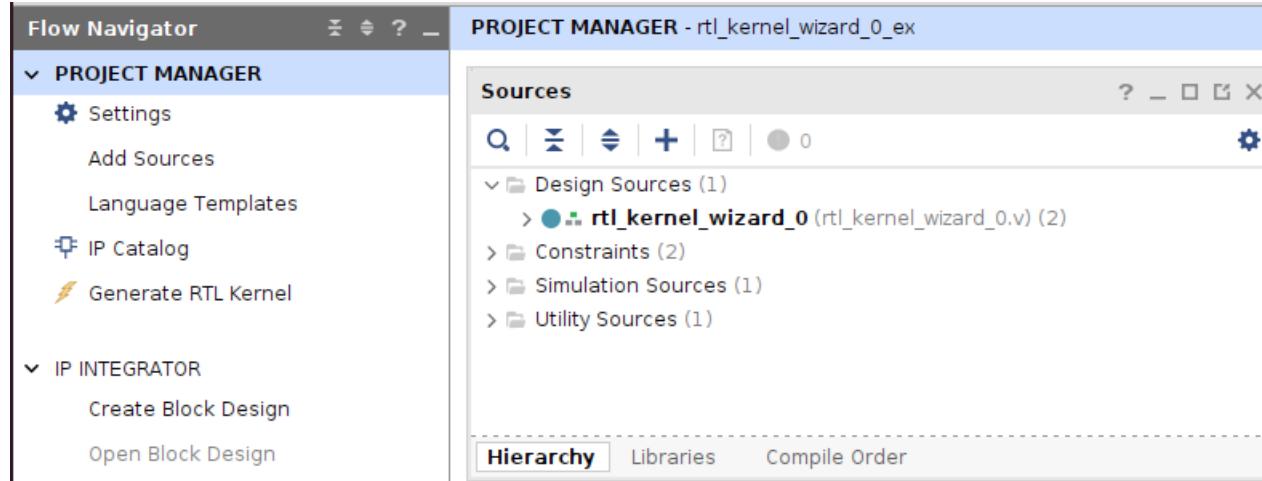
Global Memory (Page 4 of 6): This screen shows the configuration for a single AXI master interface. It includes fields for the number of AXI master interfaces (1), the AXI master definition (m00_axi, width 64 bytes, 1 argument), and the argument definition (axi00_ptr0). Buttons at the bottom include < Back, Next >, OK, and Cancel.

Streaming interfaces for Kernel to Kernel Communication (Page 5 of 6): This screen shows the configuration for streaming interfaces. It includes a table for stream settings and a note that 0 AXI4-Stream interfaces are defined. Buttons at the bottom include < Back, Next >, OK, and Cancel.

Summary (Page 6 of 6): This screen provides a summary of the kernel configuration. It lists the VLNV (mycompany.com:kernel:rtl_kernel_wizard_0:1.0), target platform (Unknown), function prototype (void rtl_kernel_wizard_0(global void *axi00_ptr0);), and a register map table. Notes at the bottom advise on setting kernel arguments and executing the kernel. Buttons at the bottom include < Back, Next >, OK, and Cancel.

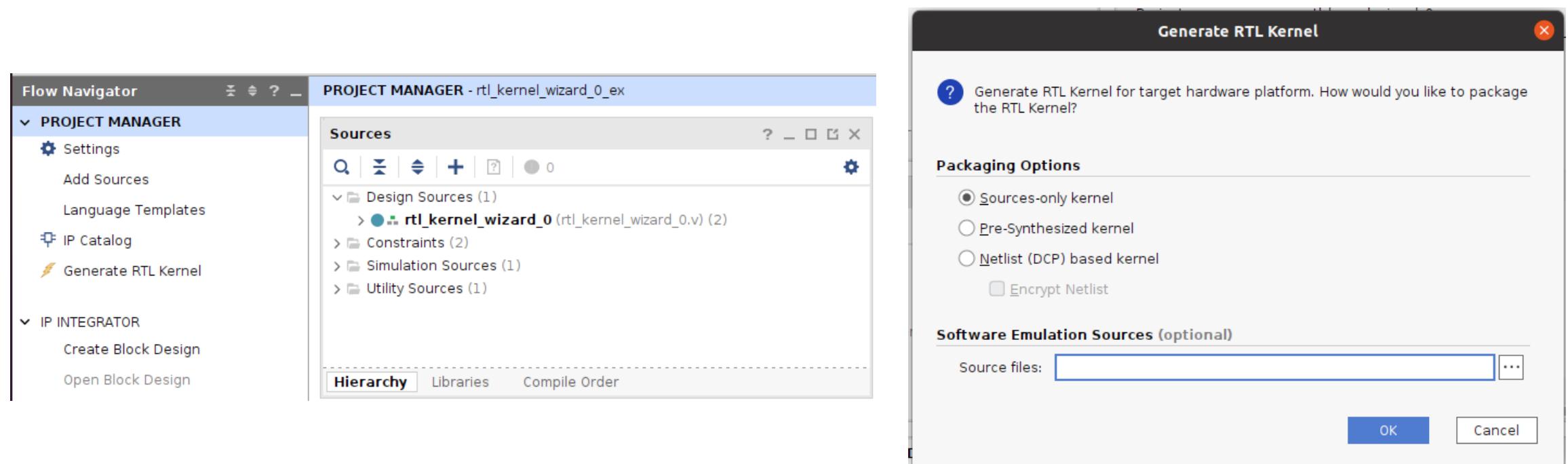
Building an Application with C++ & RTL Based Kernel

The Vivado IDE window will automatically pop up as shown in the figure below.



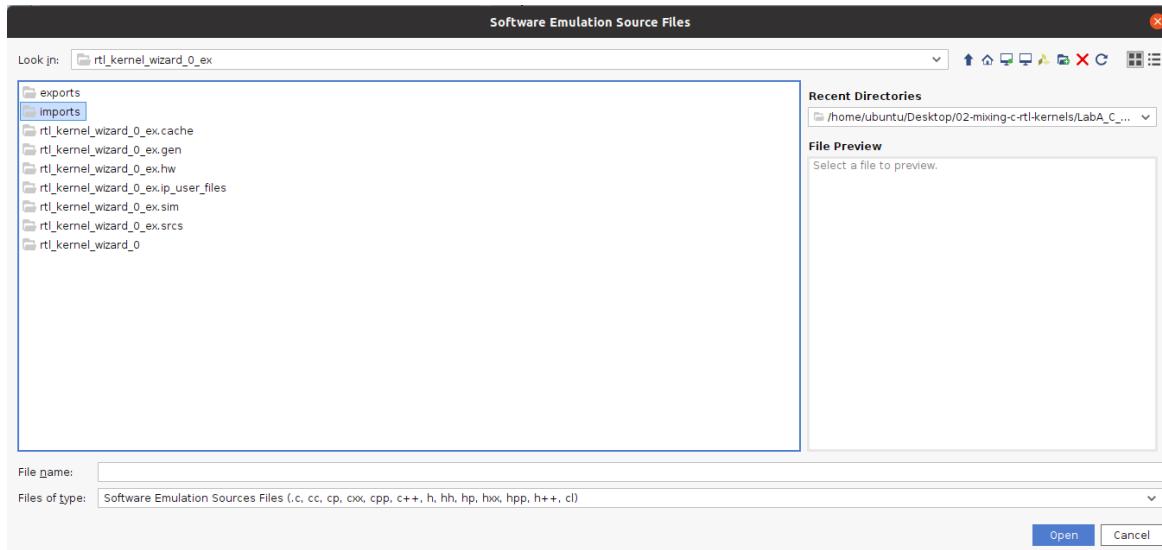
Building an Application with C++ & RTL Based Kernel

The Vivado IDE window will automatically pop up as shown in the figure below.
Click the “Generate RTL Kernel”, the window will pop up as the figure in the right side.



Building an Application with C++ & RTL Based Kernel

Add the file generated by the RTL Kernel Wizard to the software emulation and select the location of the file.



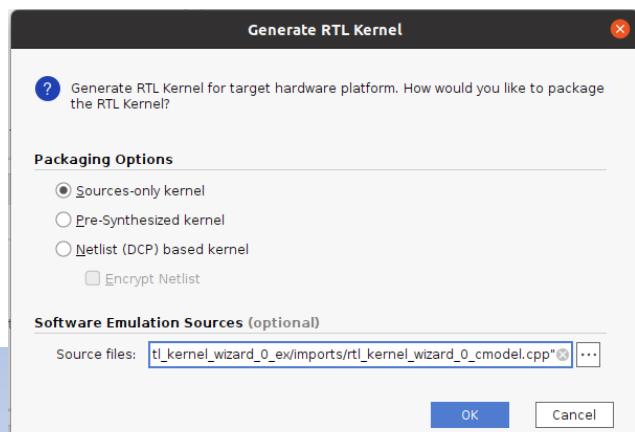
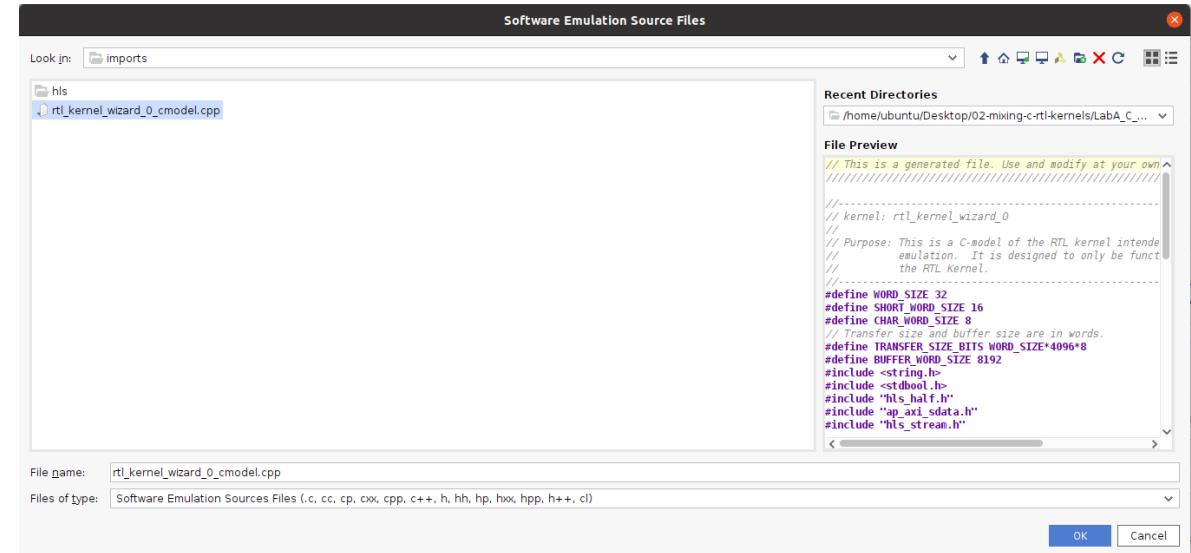
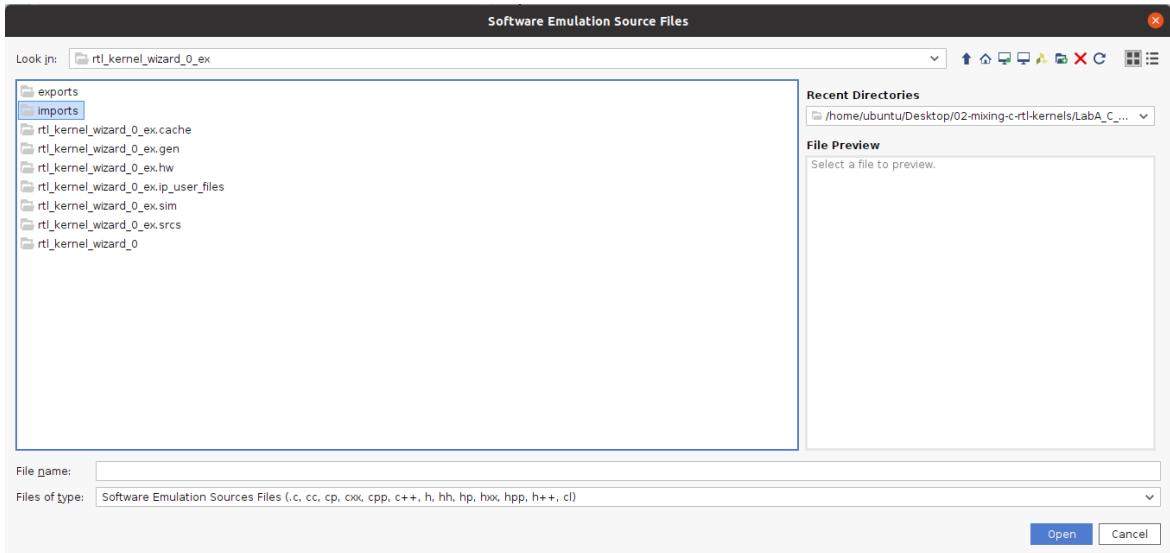
The screenshot shows the same 'Software Emulation Source Files' dialog, but now the 'rtl_kernel_wizard_0_cmodel.cpp' file is selected in the tree view. The right pane displays the contents of this file. The code includes comments explaining it is a C-model of the RTL kernel for emulation. It defines WORD_SIZE, SHORT_WORD_SIZE, CHAR_WORD_SIZE, and includes various headers like string.h, dbbool.h, half.h, ap_axi_sdata.h, and ap_stream.h. At the bottom, there are fields for 'File name:' and 'Files of type:', both set to 'Software Emulation Sources Files (.c, .cc, .cp, .cxx, .cpp, .c++, .h, .hh, .hp, .hxx, .hpp, .h++, .cl)'. There are 'OK' and 'Cancel' buttons at the bottom right.

```
// This is a generated file. Use and modify at your own risk.
// kernel: rtl_kernel_wizard_0
// Purpose: This is a C-model of the RTL kernel intended for
// emulation. It is designed to only be functional
// with the RTL Kernel.

#define WORD_SIZE 32
#define SHORT_WORD_SIZE 16
#define CHAR_WORD_SIZE 8
// Transfer size and buffer size are in words.
#define TRANSFER_SIZE_BITS WORD_SIZE*4096*8
#define BUFFER_WORD_SIZE 8192
#include <string.h>
#include <dbbool.h>
#include "hls_half.h"
#include "ap_axi_sdata.h"
#include "hls_stream.h"
```

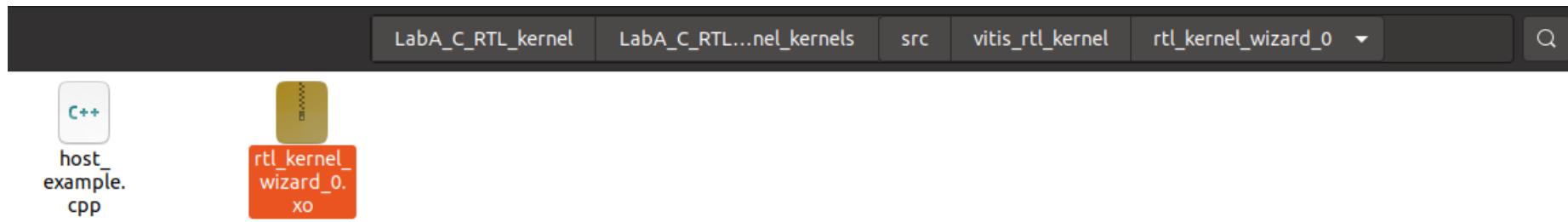
Building an Application with C++ & RTL Based Kernel

Add the file generated by the RTL Kernel Wizard to the software emulation and select the location of the file.



Building an Application with C++ & RTL Based Kernel

The IP file is located in `./LabA_C_RTL_kernel_kernels/src/vitis rtl kernel/rtl_kernel_wizard_0`.



Building an Application with C++ & RTL Based Kernel

```
// Creating Program from Binary File
cl::Program::Binaries bins;
bins.push_back({buf,nb});
cl::Program program(context, devices, bins);

// 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");

cl::Kernel krnl_const_add(program,"rtl_kernel_wizard_0");
```

Building an Application with C++ & RTL Based Kernel

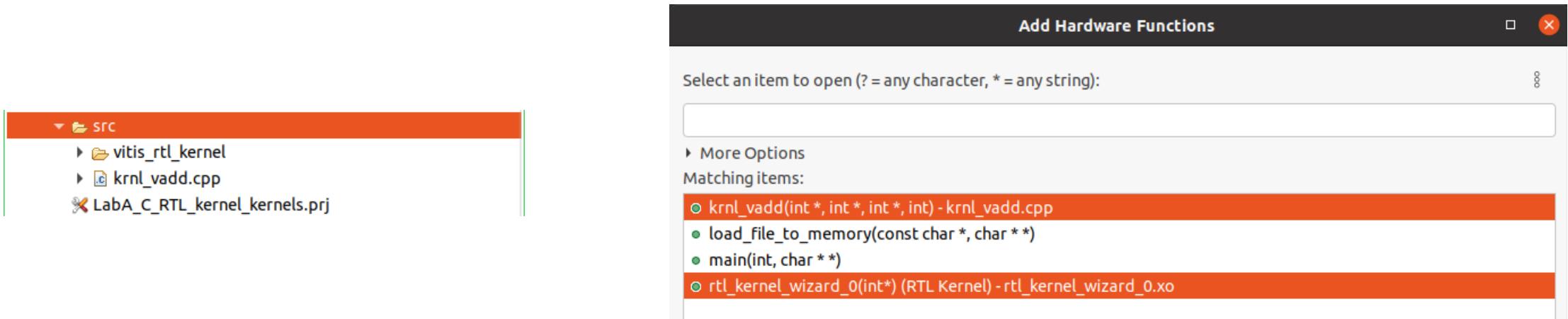
```
//set the kernel Arguments

krnl_vector_add.setArg(0,buffer_a);
krnl_vector_add.setArg(1,buffer_b);
krnl_vector_add.setArg(2,buffer_result);
krnl_vector_add.setArg(3,DATA_SIZE);

krnl_const_add.setArg(0,buffer_result);
//Launch the Kernel
q.enqueueTask(krnl_vector_add);
q.enqueueTask(krnl_const_add);
```

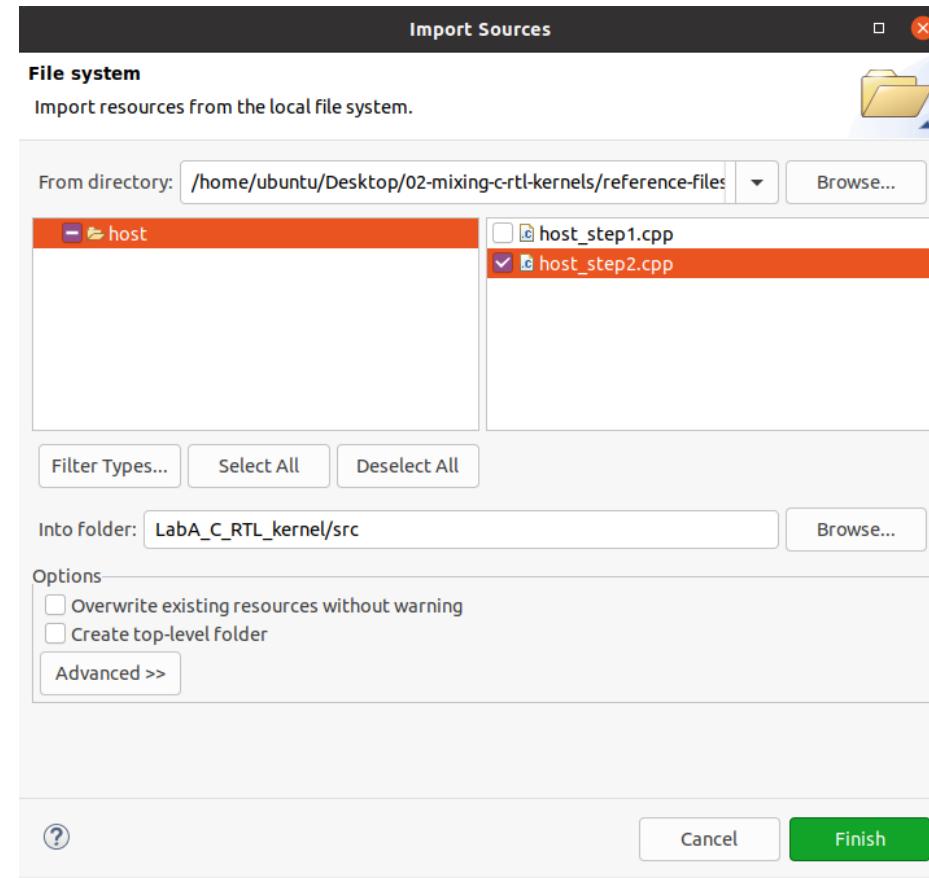
Building an Application with C++ & RTL Based Kernel

Add C++ & RTL kernel to src, select “krnl_vadd” and “rtl_kernel_wizard_0”.

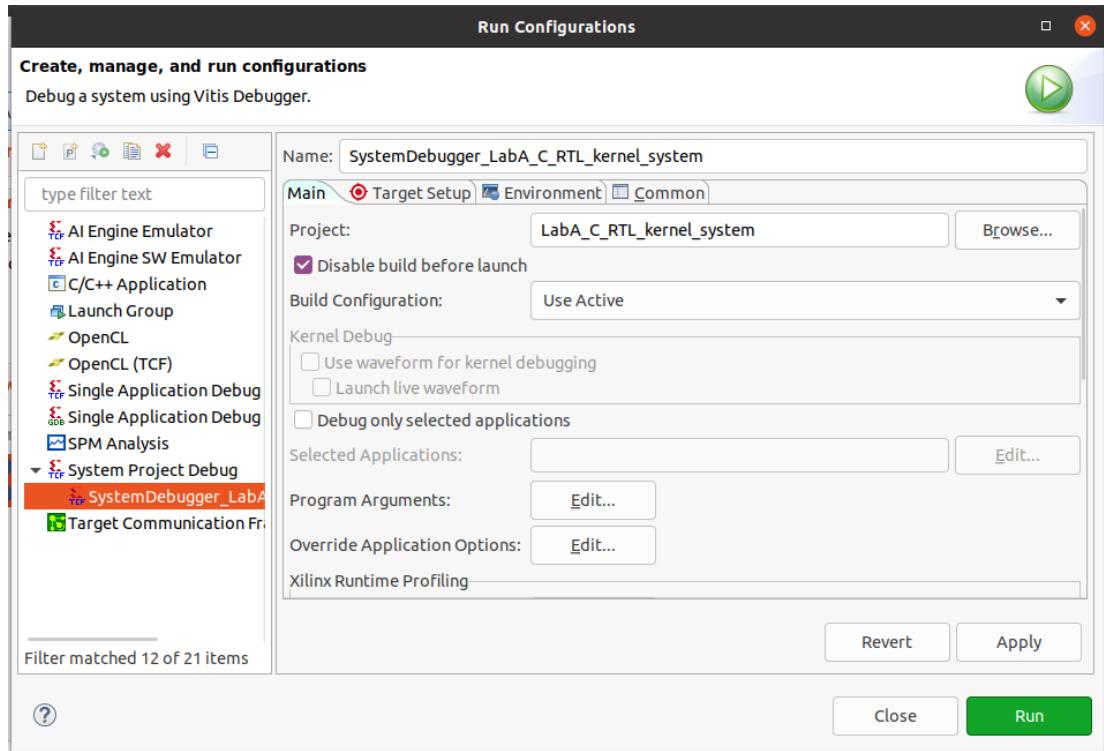
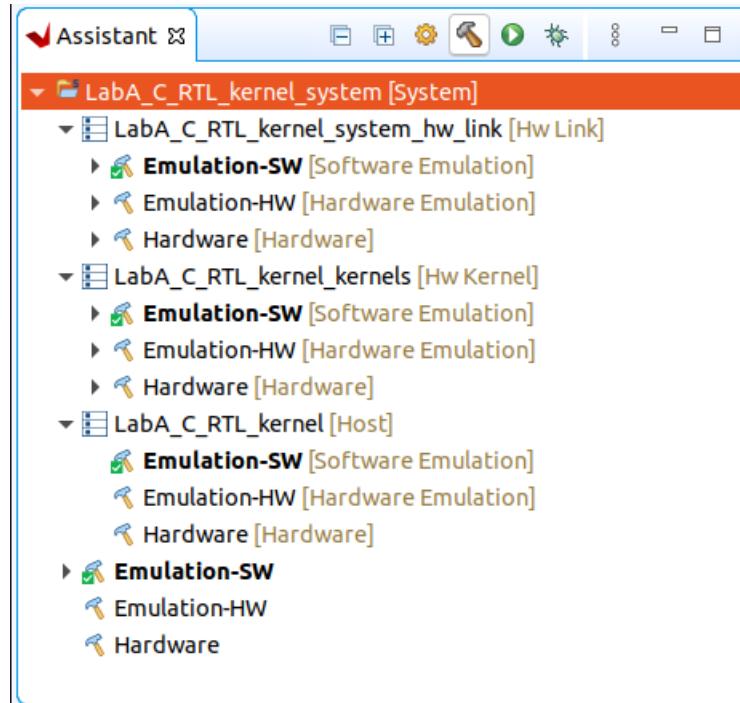


Building an Application with C++ & RTL Based Kernel

Add host code to src, select “**host_step2.cpp**”.

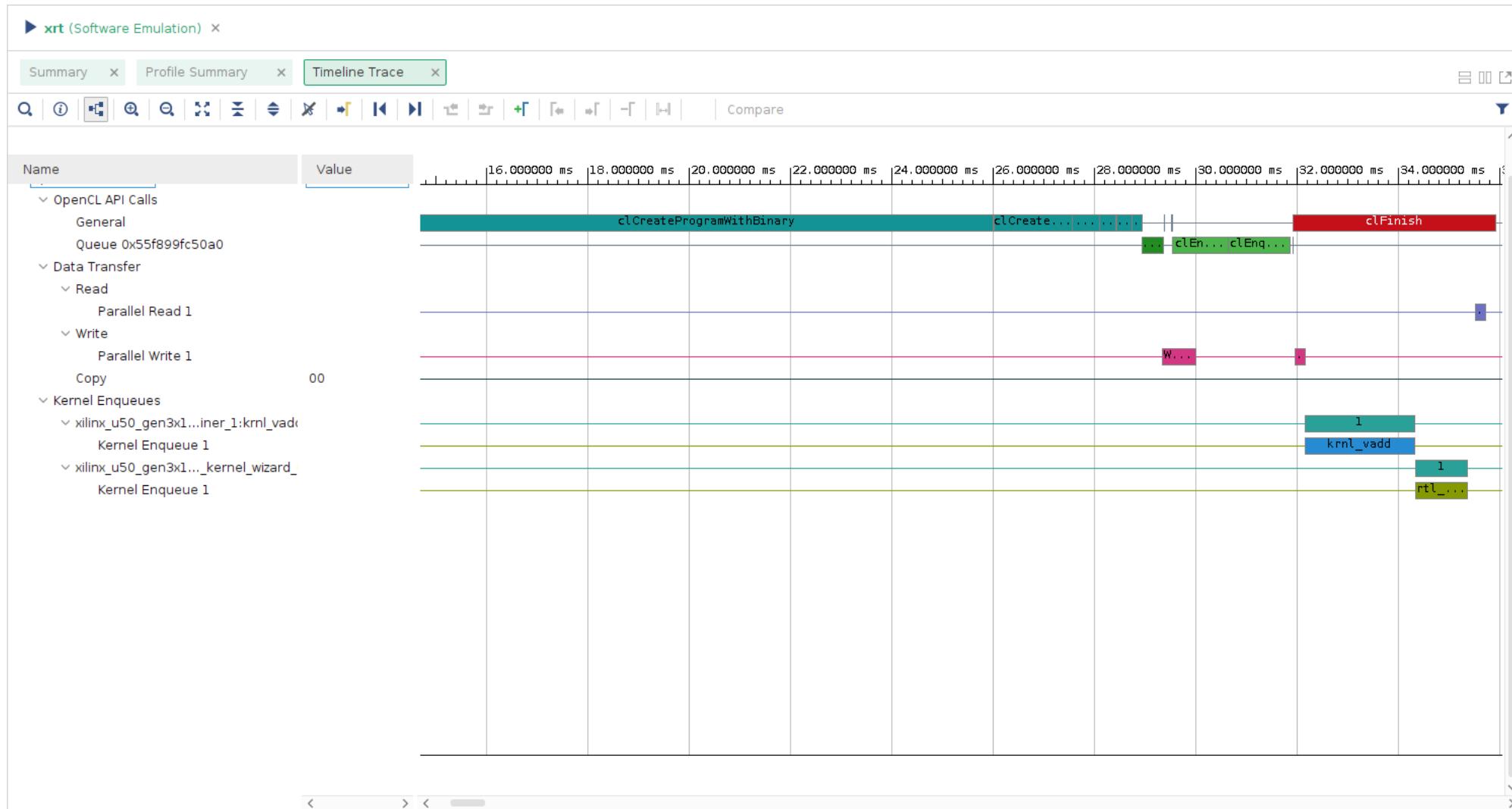


Building an Application with C++ & RTL Based Kernel



```
Console Problems Vitis Log Guidance
<terminated> (exit value: 0) SystemDebugger_LabA_CRTL_kernel_system_LabA_CRTL_kernel [OpenCL] /home/ubuntu/Desktop/02-mixing-c-rtl-kernels/LabA_CRTL_kernel/LabA_CRTL_kernel/Emulation-SW
[Console output redirected to file:/home/ubuntu/Desktop/02-mixing-c-rtl-kernels/LabA_CRTL_kernel/LabA_CRTL_kernel/Emulation-SW/SystemDebugger_LabA_CRTL_kernel_system_LabA_CRTL_kernel]
No FPGA binary file specified through the command line, using:./binary_container_1.xclbin
Found Platform
Platform Name: Xilinx
Loading: './binary_container_1.xclbin'
Kernel Name: rtl_kernel_wizard_0_1, CU Number: 0, Thread creation status: success
Kernel Name: krl_vadd_1, CU Number: 1, Thread creation status: success
TEST WITH TWO KERNELS PASSED
device process sw_emu_device done
```

Building an Application with C++ & RTL Based Kernel



Building an Application with C++ & RTL Based Kernel

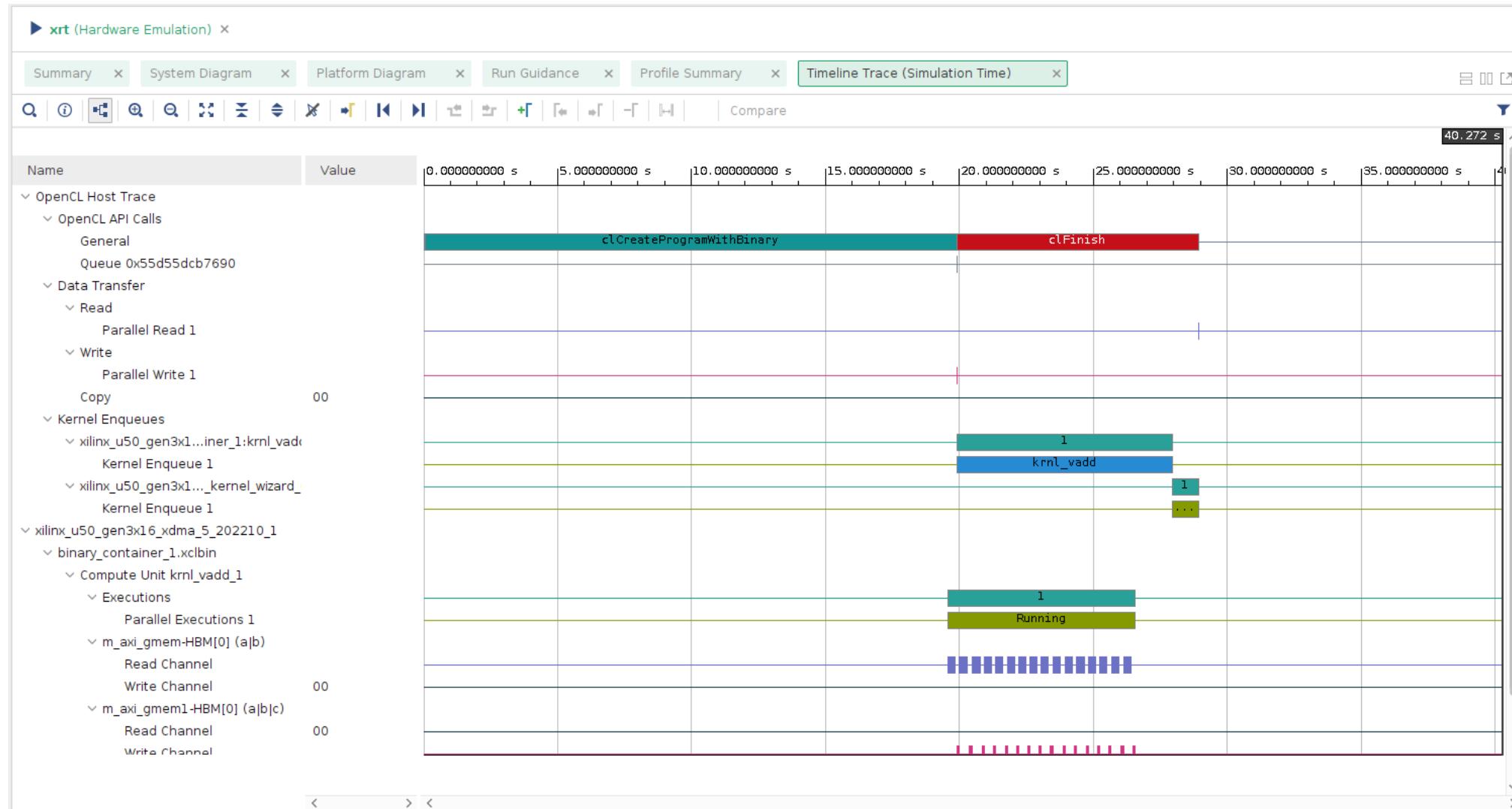
The screenshot shows the Vitis IDE interface. On the left, the Assistant view displays the project structure:

- LabA_CRTL_kernel_system [System]
 - LabA_CRTL_kernel_system_hw_link [Hw Link]
 - Emulation-SW [Software Emulation]
 - Emulation-HW [Hardware Emulation]
 - Hardware [Hardware]
 - LabA_CRTL_kernel_kernels [Hw Kernel]
 - Emulation-SW [Software Emulation]
 - Emulation-HW [Hardware Emulation]
 - Hardware [Hardware]
 - LabA_CRTL_kernel [Host]
 - Emulation-SW [Software Emulation]
 - SystemDebugger_LabA_CRTL_kernel_system_LabA_CRTL_kernel [xrt] [12 Oct 2024 01:07]
 - Run Summary (xrt) [12 Oct 2024 01:07]
 - Emulation-HW [Hardware Emulation]
 - SystemDebugger_LabA_CRTL_kernel_system_LabA_CRTL_kernel [xrt] [12 Oct 2024 01:07]
 - Run Summary (xrt) [12 Oct 2024 01:07]
 - Hardware [Hardware]
 - Emulation-SW
 - Emulation-HW
 - Hardware

On the right, the Console tab shows the build log output:

```
<terminated> (exit value: 0) SystemDebugger_LabA_CRTL_kernel_system_LabA_CRTL_kernel[OpenCL] /home/ubuntu/Desktop/02-mixing-c-rtl-kernels/LabA_CRTL_kernel/LabA_CRTL_kernel/Emulation-HW [Console output redirected to file: /home/ubuntu/Desktop/02-mixing-c-rtl-kernels/LabA_CRTL_kernel/LabA_CRTL_kernel/Emulation-HW/SystemDebugger_LabA_CRTL_kernel_system_LabA_CRTL_kernel [No FPGA binary file specified through the command line, using:../binary_container_1.xclbin] Found Platform Platform Name: Xilinx Loading: '../binary_container_1.xclbin' INFO: [HW-EMU 01] Hardware emulation runs simulation underneath. Using a large data set will result in long simulation times. It is recommended that a small dataset is used for faster simulation times. INFO: [HW-EMU 07-0] Please refer the path "/home/ubuntu/Desktop/02-mixing-c-rtl-kernels/LabA_CRTL_kernel/LabA_CRTL_kernel/Emulation-HW/.run/32566/hw_em/device0/binary_0" TEST WITH TWO KERNELS PASSED INFO:[ Vitis-EM 22 ] [Time elapsed: 0 minute(s) 27 seconds, Emulation time: 0.266505 ms] Data transfer between kernel(s) and global memory(s) krnl_vadd_1:m_axi_gmem-HBM[0] RD = 32.000 KB WR = 0.000 KB krnl_vadd_1:m_axi_gmem1-HBM[0] RD = 0.000 KB WR = 16.000 KB rtl_kernel_wizard_0_1:m00_axi-HBM[0] RD = 16.000 KB WR = 16.000 KB INFO: [HW-EMU 06-0] Waiting for the simulator process to exit INFO: [HW-EMU 06-1] All the simulator processes exited successfully INFO: [HW-EMU 07-0] Please refer the path "/home/ubuntu/Desktop/02-mixing-c-rtl-kernels/LabA_CRTL_kernel/LabA_CRTL_kernel/Emulation-HW/.run/32566/hw_em/device0/binary_0"
```

Building an Application with C++ & RTL Based Kernel



Reference

- https://github.com/Xilinx/Vitis-Tutorials/tree/2022.1/Hardware_Acceleration/Feature_Tutorials/01-rtl_kernel_workflow
- https://github.com/Xilinx/Vitis-Tutorials/tree/2022.1/Hardware_Acceleration/Feature_Tutorials/02-mixing-c-rtl-kernels
- <https://github.com/Xilinx/Vitis-Tutorials/issues/302>
- [https://github.com/bol-edu/2022-fall-ntu/blob/main/LabA/RTL_kernel_workflow_and_Mixing_C%2B%2B_and_RTL_Kernels/r11921061%23_%E9%83%AD%E9%9C%96%E7%92%9F%20\(KUO%2C%20LIN-JING\)_139712_2908799_LabA_r11921061.pdf](https://github.com/bol-edu/2022-fall-ntu/blob/main/LabA/RTL_kernel_workflow_and_Mixing_C%2B%2B_and_RTL_Kernels/r11921061%23_%E9%83%AD%E9%9C%96%E7%92%9F%20(KUO%2C%20LIN-JING)_139712_2908799_LabA_r11921061.pdf)