

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
----------------------	--------------------------------	-----------------------------	--------------------	----------------------------	-----------------	--------------------------	-------------------

Using the RTL Kernel Wizard

Introduction

This lab guides you through the steps involved in using the Vitis RTL Kernel wizard. This allows RTL code to be used in a Vitis design.

Objectives

After completing this lab, you will be able to:

- Understand how to use the RTL Kernel wizard available in Vitis
- Create a new RTL based IP
- Add the new IP to an application
- Verify the functionality of the design in hardware

Steps

Create a Vitis Project

1. Start Vitis and select the default workspace (or continue with the workspace from the previous lab)
2. Create a new application project

Use `Create Application Project` from Welcome page, or use `File > New > Application Project` to create a new application

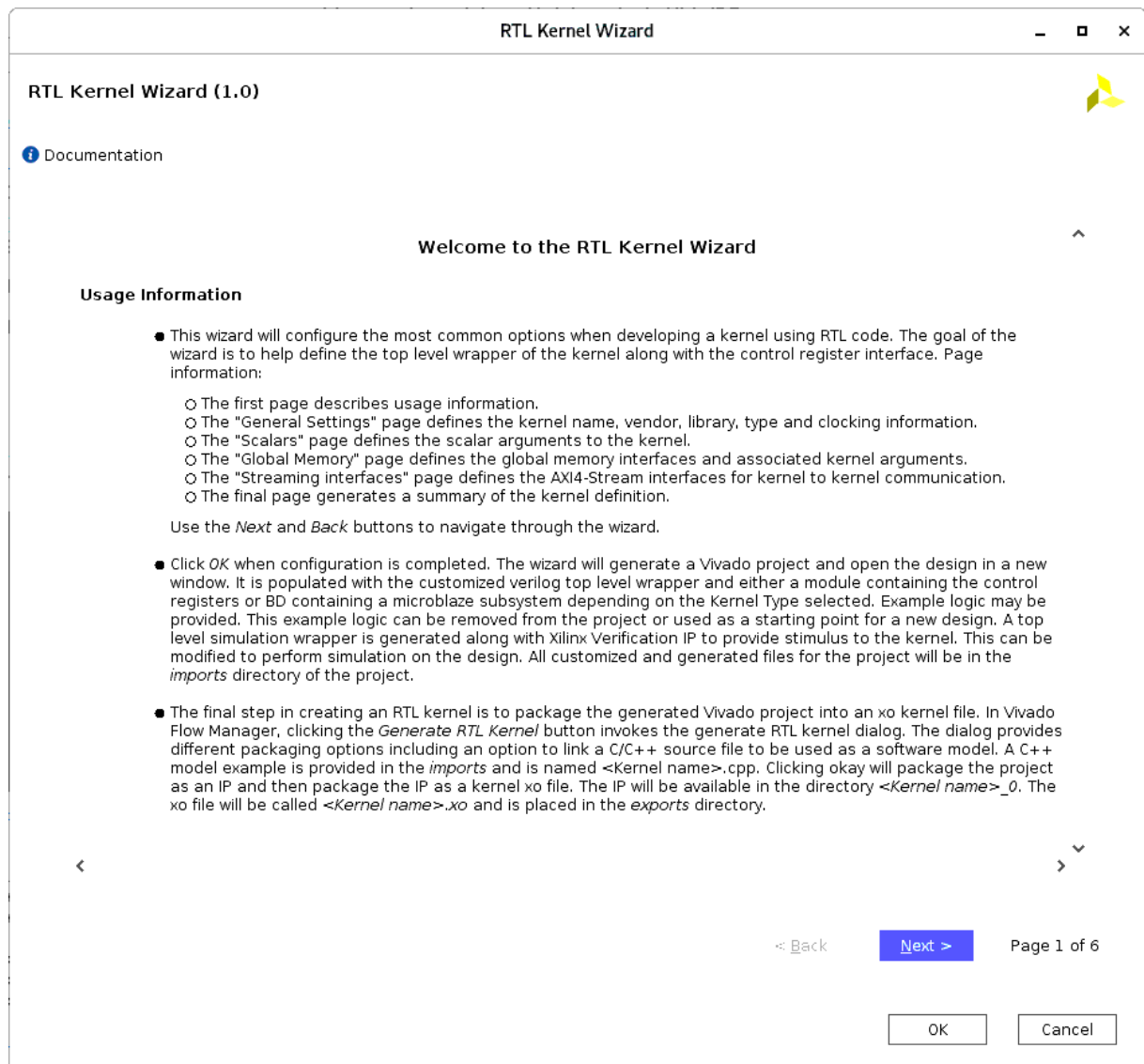
3. In the *New Project's* page enter `rtl_kernel` in the *Project name:* field and click **Next>**
4. Select your target platform and click **Next>**

If you don't see your target platform, then click on '+' button, browse to directory where platform is located and click **OK**

5. Select **Empty Application** and click **Finish**

Create RTL_Kernel Project using RTL Kernel Wizard

1. Make sure the `rtl_kernel.prj` under *rtl_kernel_example* in the *Explorer* view is selected
2. Select **Xilinx > RTL Kernel Wizard...**



3. Click **Next>**

4. Change **Kernel name** to **KVAdd**, (for Kernel Vector Addition), **Kernel vendor** to **xilinx.com** leaving the **Kernel library**, **Kernel type**, **Kernel control interface**, **Number of clocks** and **Has reset** to the default values



General Settings

Kernel identification

Kernel name

Kernel vendor

Kernel library

Kernel options

Kernel type

Kernel control interface

Clock and Reset options

Number of clocks

Has reset

Page 2 of 6

5. Click **Next>**

6. Leave **Number of scalar kernel input arguments** set to the default value of **1**, **Argument Name** set to the default value of **scalar00** and the **Argument type** as **uint**, and click **Next>**

Scalars

Number of scalar kernel input arguments

Scalar input argument definition

Argument name	Argument type
scalar00	<input type="text" value="uint"/>

7. We will have three arguments to the kernel (2 inputs and 1 output) which will be passed through Global Memory. Set **Number of AXI master interfaces** to be **3**

8. Keep the width of each AXI master data width to **64** (note this is specified in bytes so this will give a width of 512 bits for each interface), name **A** as the argument name for *m00_axi*, **B** for *m01_axi*, and **Res** for *m02_axi*

Global Memory

Number of AXI master interfaces 3 ▼

AXI master definition

Interface name		Width (bytes)		Number of arguments
m00_axi	✕	64	▼	1
m01_axi	✕	64	▼	1
m02_axi	✕	64	▼	1

Argument definition

Interface	Argument name
m00_axi	A
m01_axi	B
m02_axi	Res

9. Click **Next>** and Stream Interface page will be displayed. Notice: this example does not use AXI4-Stream interfaces. Therefore, make sure that `Number of AXI4-Stream interfaces` is set to 0

Streaming interfaces

Number of AXI4-Stream interfaces 0 ▼

Stream settings

Name	Mode	Width (bytes)
------	------	---------------

10. Click **Next>** and the summary page will be displayed showing a function prototype and register map for the kernel.

Note the control register and the scalar operand are accessed via the S_AXI_CONTROL interface. The control register is at offset 0x0 and the scalar operand is at offset 0x10

VLNV: Xilinx:kernel:KVAdd:1.0

Target platform: xilinx:aws-vu9p-f1:shell-v04261818:201920.1

Function prototype: `void KVAdd(const uint scalar00, global void *A, global void *B, global void *Res);`

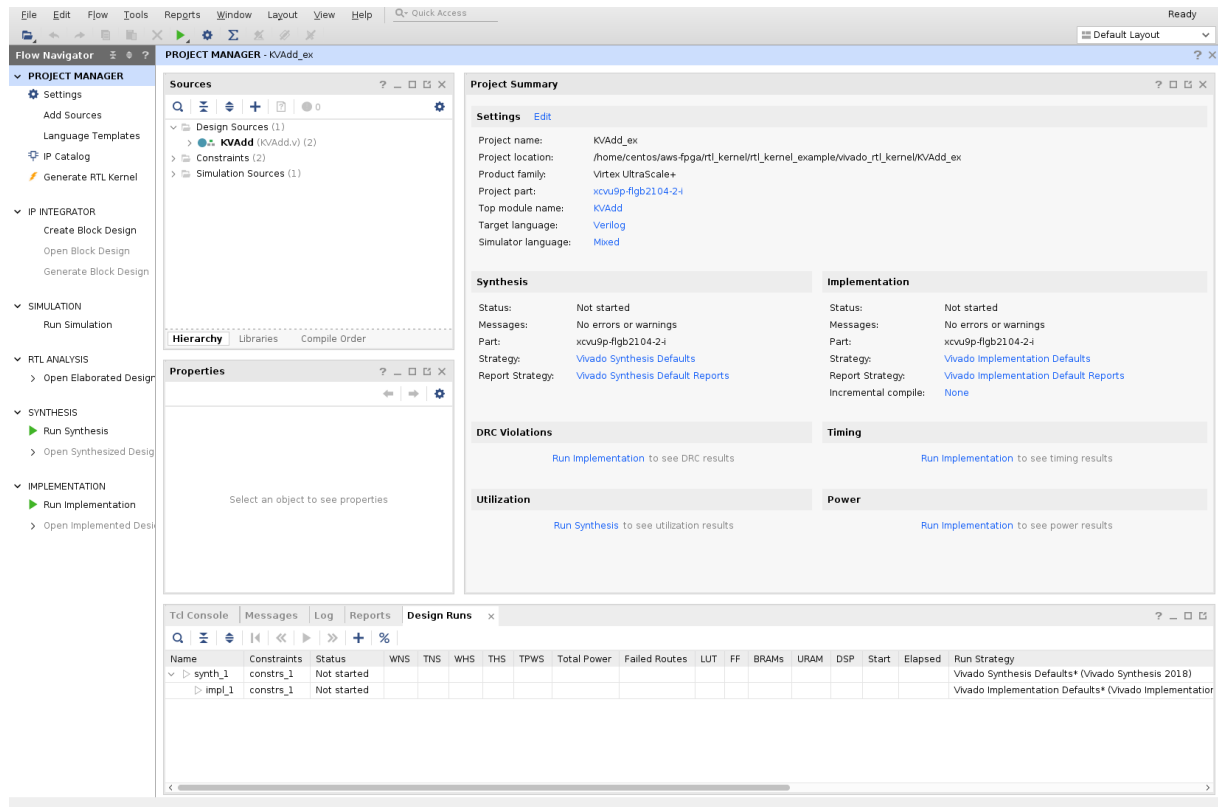
Register map:

ID	Name	Offset	Type (bits)	Interface
N/A	Control	0x000	N/A	S_AXI_CONTROL
0	scalar00	0x010	uint (32)	S_AXI_CONTROL
1	A	0x018	generic pointer (64)	m00_axi
2	B	0x024	generic pointer (64)	m01_axi
3	Res	0x030	generic pointer (64)	m02_axi

Notes: Please see example host code generated in `./exports/src/host_example.cpp` for methods used to set the kernel arguments and execute the kernel. The global memory pointers are generic and the kernel should be tuned to handle the data type of the vectors.

11. Click **OK** to close the wizard

Notice that a Vivado Project will be created and opened after few seconds



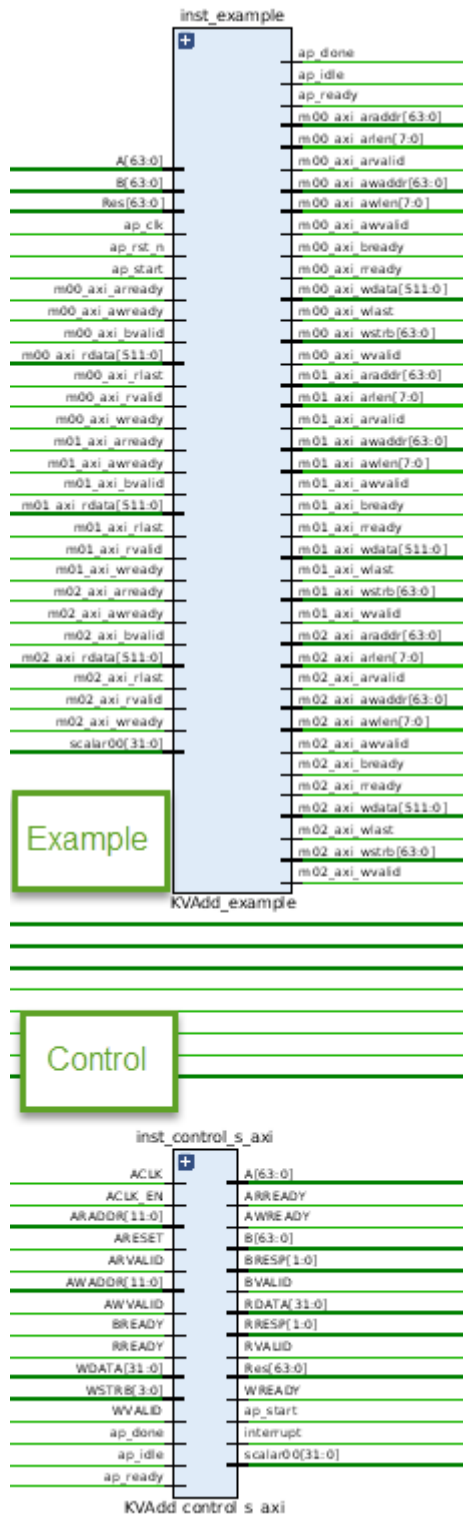
Analyze the design created by the RTL Kernel wizard

1. Expand the hierarchy of the *Design Sources* in the Sources window and notice all the design sources, constraint file, and the basic testbench generated by the wizard



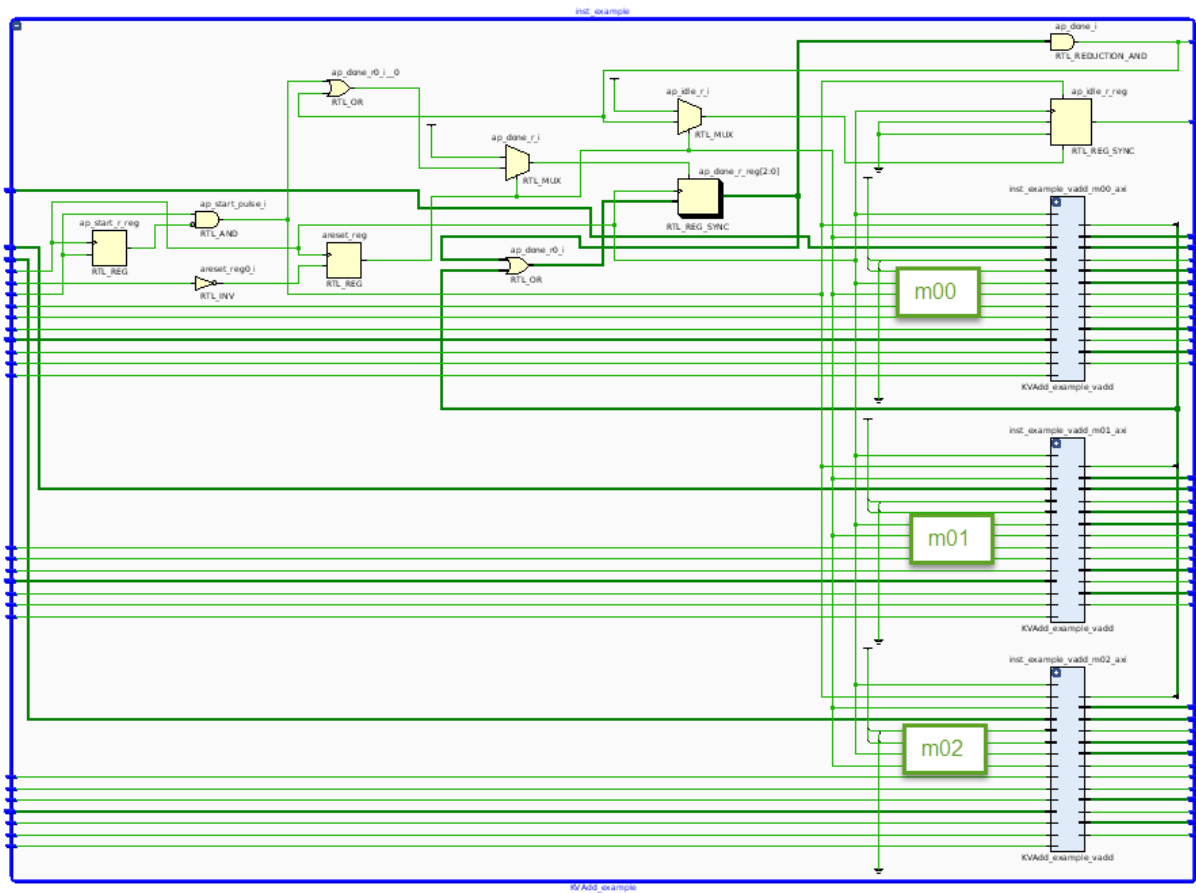
There is one module to handle the control signals (ap_start, ap_done, and ap_idle) and three master AXI channels to read source operands from, and write the result to DDR. The expanded m02_axi module shows *read*, *adder*, *write* instances

2. Select **Flow Navigator > RTL ANALYSIS > Open Elaborated Design** which will analyze the design and open a schematic view. Click **OK**
3. You should see two top-level blocks: example and control as seen below

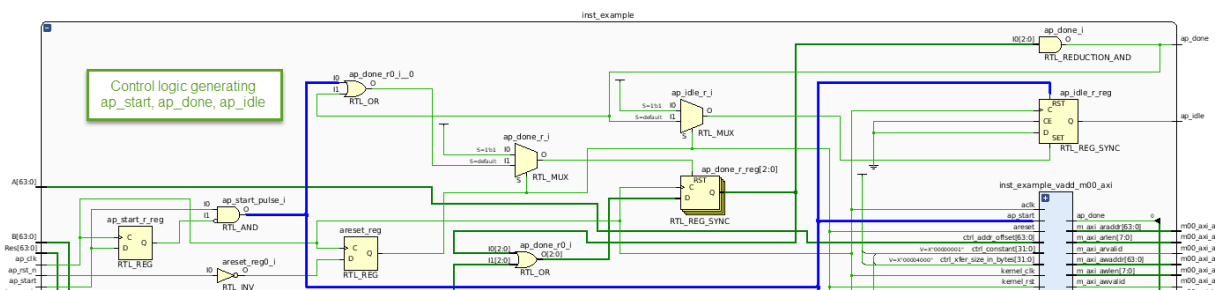


Notice the AXI Master interfaces are 64 bytes (or 512 bits) wide as specified earlier

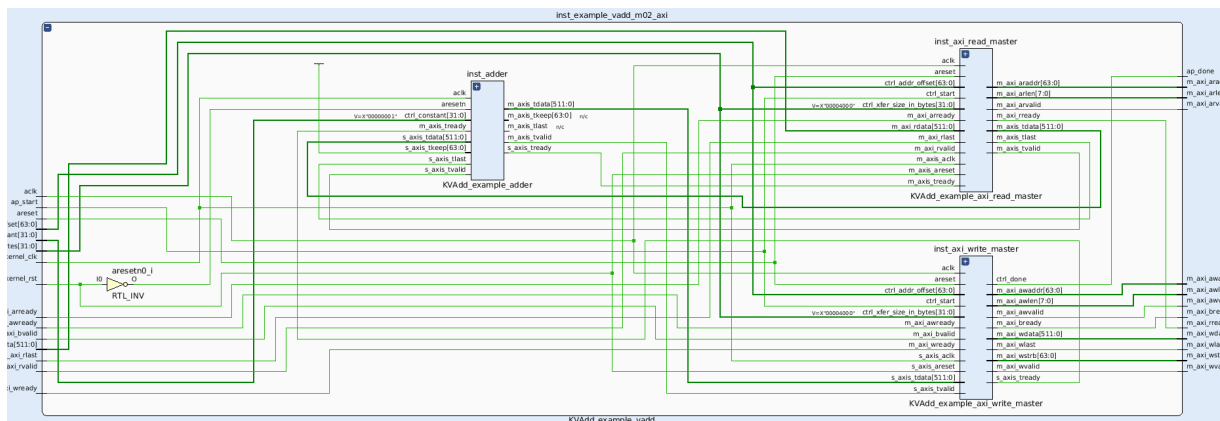
4. Double-click on the example block and observe the three hierarchical Master AXI blocks



5. Zoom in into the top section and see the control logic, the wizard has generated the ap_start, ap_idle, and ap_done control signals



6. Traverse through one of the AXI interface blocks (e.g. m02) and observe that the design consists of a Read Master, Write Master, and an Adder. (Click on the image to download an enlarged version if necessary)



7. Close the elaborated view by selecting **File > Close Elaborated Design**

8. Click **OK**

Generate the RTL Kernel

1. Select **Flow > Generate RTL Kernel...** or click **Generate RTL Kernel** in the left-hand side pane

2. Click **OK** using the default option (**Sources-only kernel**)

The packager will be run, generating the xo file which will be used in the design

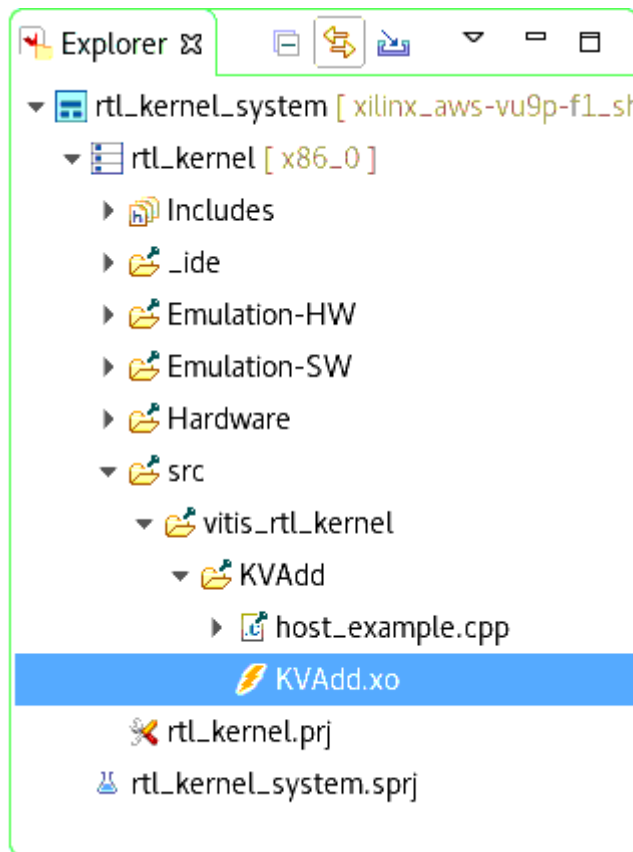
3. Click **OK**, then **Yes** to exit Vivado

4. Click **OK** on the message box indicating that the *RTL Kernel has been imported* and return to Vitis

Analyze the RTL kernel added to the Vitis project

1. Expand the `src` folder under the `rtl_kernel`

Notice that the `vitis_rtl_kernel` folder has been added to the project. Expanding this folder (`KVAdd`) shows the kernel (`.xo`) and a C++ file which have been automatically included




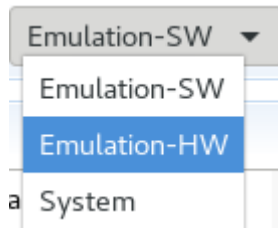
2. Double-click on the `host_example.cpp` to open it

- The `main` function is defined around line 62. The number of words it transfers is 4,096
- Notice around line 96 the source operands and expected results are initialized
- Around line 209 (from the `clCreateProgramWithBinary()` function) shows the loading of the xclbin and creating the OpenCL kernel (`clCreateKernel()`)
- The following lines (~250) show how the buffers are created in the device memory and enqueued (`clCreateBuffer()`, `clEnqueueWriteBuffer()`)

- Around lines 305, the arguments to the kernel are set (`c1SetKernelArg()`), and the kernel is enqueued to be executed (`c1EnqueueTask()`)
- Around line 338 results are read back (`c1EnqueueReadBuffer()`) and compared to the expected results.
- The *Shutdown and cleanup* section shows releasing of the memory, program, and kernel

Define a hardware kernel, and build the project

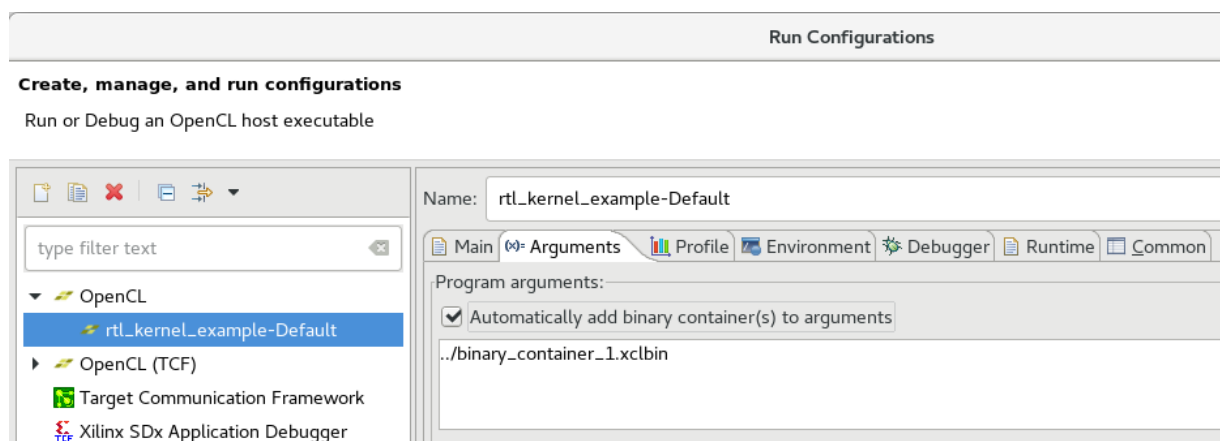
1. Select **rtl_kernel.prj** in the *Explorer* view to see the project settings page
2. Click on the **Add Hardware Function button** () and select *KVAdd*
3. Select **Emulation-HW** on the drop-down button of *Active build configuration*:



4. Select **Project > Build Project** or click on the () button

This will build the project including *rtl_kernel* executable file under the Emulation-HW directory. It may take about 10 minutes to build

5. Select **Run > Run Configurations...** to open the configurations window
6. Click on the **Arguments** tab, and then select **Automatically add binary container(s) to arguments**

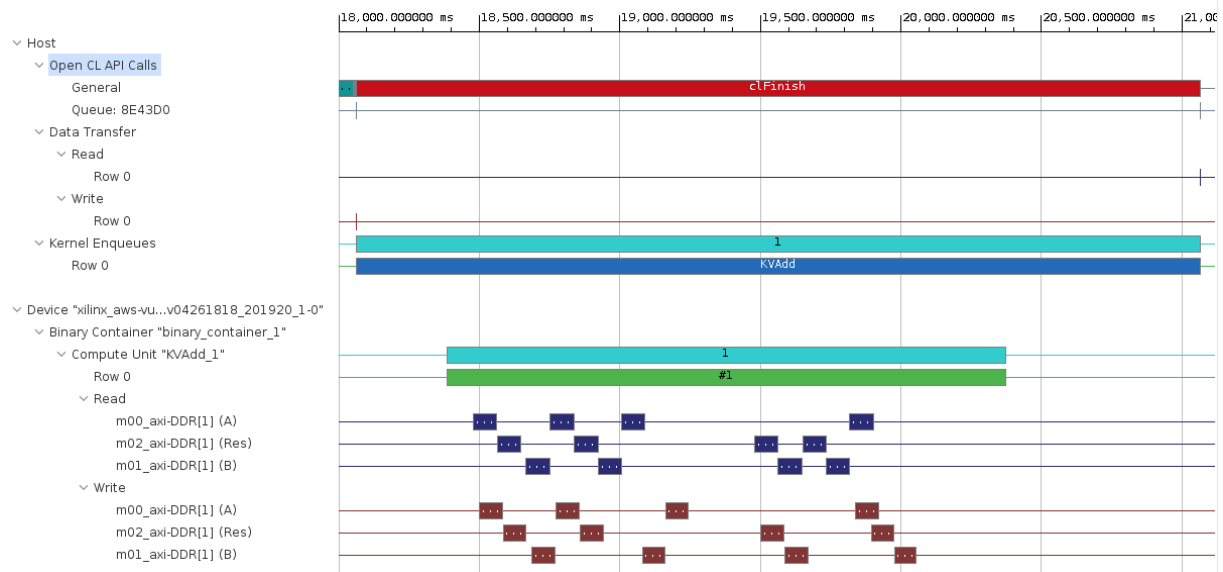


7. Click **Apply**, and then click **Run** to run the application
8. The Console tab shows that the test was completed successfully along with the data transfer rate

Note that three memory controller are used, all of which targeting to the same DDR

```
<terminated> (exit value: 0) rtl_kernel-Default [OpenCL] /home/centos/workspace/rtl_kernel/Emulation-HW/rtl_kernel (4/29/20, 2:15 AM)
[[Console output redirected to file:/home/centos/workspace/rtl_kernel/Emulation-HW/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_shell-v04261818_201920_1
Selected xilinx_aws-vu9p-f1_shell-v04261818_201920_1 as the target device
INFO: loading xclbin ../binary_container_1.xclbin
INFO: [HW-EM 01] Hardware emulation runs simulation underneath. Using a large data set will result in long simulation times.
INFO: Test completed successfully.
INFO: [ Vitis-EM 22 ] [Time elapsed: 0 minute(s) 23 seconds, Emulation time: 0.0593202 ms]
Data transfer between kernel(s) and global memory(s)
KVAdd_1:m00_axi-DDR[1]      RD = 16.000 KB      WR = 16.000 KB
KVAdd_1:m01_axi-DDR[1]      RD = 16.000 KB      WR = 16.000 KB
KVAdd_1:m02_axi-DDR[1]      RD = 16.000 KB      WR = 16.000 KB
```

9. In the *Assistant* view, expand **Emulation-HW > rtl_kernel_example-Default**, and double-click on **Run Summary (xclbin)**
10. The *Vitis Analyzer* window will open. Click on **xclbin (Hardware Emulation) > Application Timeline** entry, expand all entries in the timeline graph, zoom appropriately and observe the transactions



Run the Application in hardware

Since building the FPGA hardware takes some time, a precompiled solution is provided.

Note: There is a mismatch between the TARGET_DEVICE named in shell and what wizard based host code is expecting. So before generating the bitstream it is necessary to add one line in the host code and re-compile host application by clicking **Project > Build Project**. Add the following code after line 37 in `host_example.cpp`

```
#define TARGET_DEVICE "xilinx_aws-vu9p-f1_dynamic_5_0"

33 #if defined(VITIS_PLATFORM) && !defined(TARGET_DEVICE)
34 #define STR_VALUE(arg)      #arg
35 #define GET_STRING(name) STR_VALUE(name)
36 #define TARGET_DEVICE GET_STRING(VITIS_PLATFORM)
37 #endif
38 #define TARGET_DEVICE "xilinx_aws-vu9p-f1_dynamic_5_0"
```

1. Change directory to `~/compute_acceleration/solutions/rtlkernel_lab`
2. Execute the following command to run the application

```
./rtl_kernel ../binary_container_1.awsxclbin
```

The FPGA bitstream will be downloaded and the host application will be executed showing output similar to:

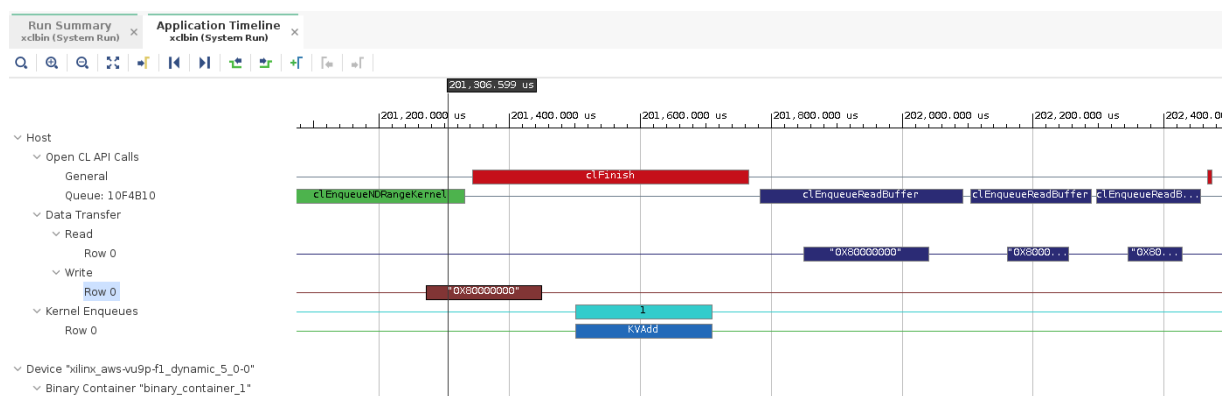
```
[Console output redirected to file:/home/centos/workspace/rtl_kernel/Hardware/rtl_kernel-Default.launch.log]
INFO: Found 1 platforms
INFO: Selected platform 0 from Xilinx
INFO: Found 1 devices
CL_DEVICE_NAME xilinx_aws-vu9p-f1_dynamic_5_0
Selected xilinx_aws-vu9p-f1_dynamic_5_0 as the target device
INFO: loading xclbin ../binary_container_1.awsxclbin
INFO: Test completed successfully.
```

Analyze hardware application timeline and profile summary

1. Execute the following command to see the application timeline

```
vitis_analyzer timeline_trace.csv
```

2. Zoom in into the tail end of the execution and see various activities



3. When finished, close the analyzer by clicking **File > Exit** and clicking **OK**
4. Execute the following command to see the profile summary

```
vitis_analyzer profile_summary.csv
```

- Top Operations

▼

Top Data Transfer: Kernels to Global Memory

No Data. Please use 'v++ -l --profile_kernel' to monitor and report kernel data

▼

Top Kernel Execution

Kernel Instance Address	Kernel	Context ID	Command Queue ID	Device	Start Time (ms)	Duration (ms)	Global Work Size	Local Work Size
0x1103e10	KVAdd	0	0	xilinx_aws-vu9p-f1_dynamic_5_0-0	201.501	0.208	1:1:1	1:1:1

▼

Top Memory Writes: Host to Global Memory

Buffer Address	Context ID	Command Queue ID	Start Time (ms)	Duration (ms)	Buffer Size (KB)	Writing Rate (MB/s)
0x800000000	0	0	201.273	0.176	49.152	278.647

▼

Top Memory Reads: Host to Global Memory

Buffer Address	Context ID	Command Queue ID	Start Time (ms)	Duration (ms)	Buffer Size (KB)	Reading Rate (MB/s)
0x800000000	0	0	201.850	0.191	16.384	85.760
0x800004000	0	0	202.161	0.094	16.384	174.970
0x800008000	0	0	202.345	0.083	16.384	198.243

- Kernels & Compute Units

▼

Kernel Execution

Kernel	Number Of Enqueues	Total Time (ms)	Minimum Time (ms)	Average Time (ms)	Maximum Time (ms)
KVAdd	1	0.208	0.208	0.208	0.208

- Data Transfers

▼

Data Transfer: Host to Global Memory

Context: Number of Devices	Transfer Type	Number Of Buffer Transfers	Transfer Rate (MB/s)	Average Bandwidth Utilization (%)	Average Buffer Size (KB)	Total Time (ms)	Average Time (ms)
context0:1	READ	3	133.809	1.394	16.384	0.367	0.122
context0:1	WRITE	1	278.647	2.903	49.152	0.176	0.176

When finished, close the analyzer by clicking `File > Exit` and clicking **OK**

Conclusion

In this lab, you used the RTL Kernel wizard to create an example RTL adder application. You configured the template and saw the example code that was automatically generated. You performed HW emulation and analyzed the application timeline.

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