



## ECE4810J SoC Design

Fall 2024

### Lab #2 Design with High Level Synthesis on Arty Z7 SoC Development Platform

Due: 11:59pm Oct. 23rd, 2024

#### Logistics:

- This lab is a team exercise.
- Please use the discussion board on Piazza for Q&A.
- All reports and code (if available) MUST be submitted to the assignment of Canvas.
- Internet usage is allowed and encouraged.
- No late submission is allowed for this lab.

#### Contents

1 Overview	2
2 Create and Use Custom IP Blocks Using High-Level Synthesis	2
2.1 Creating the Vitis HLS project	2
2.2 Testing the hls_multiplier and exporting the IP block	3
2.3 Adding the hls_multiplier AXI Lite Slave IP Block to your Vivado design	5
2.4 Interfacing with the RTL and HLS blocks in Software	6
2.5 Follow-up Exercises	9
3 Program the FPGA and Boot Software from the SD Card	10
4 Program Python on Arty Z7	12
5 (Optional) Optimize Your RTL ALU by Pipelining	16
6 Post-lab Questions	16
7 Deliverables	17
8 Grading policy	18
9 Acknowledgment	18
A Peer Evaluation Form	19

---

<b>B Troubleshooting</b>	<b>19</b>
B.1 PYNQ HTTP Connection . . . . .	19

## 1 Overview

In this lab, you will go through two exercises to create designs with high-level synthesis languages (instead of strict hardware description languages like Verilog/VHDL/SystemVerilog) and PYNQ. The goals of this lab are to

- Get familiar with the concept of High-level Synthesis (HLS)
- Learn to use HLS to create an IP block in C/C++
- Learn about differences between programming with HLS vs. Verilog in terms of design quality
- Get started with the Arty Z-7 board as a Python accelerator and be able to run example Jupyter notebooks with the board
- Load PYNQ overlays, use overlays and create overlays

## 2 Create and Use Custom IP Blocks Using High-Level Synthesis

There are a number of reasons why High-Level Synthesis (HLS) tools have been developed. Most of all, it has to do with developer productivity and code reuse, as well as other business-related reasons. RTL development in hardware description languages such as Verilog and VHDL is slow and difficult to debug, verify, and update. Moreover, it can not be done by software engineers without intense training in hardware development. These translate into a high expense for businesses. High-level synthesis attempts to partially solve this problem, by generating HDL code from a higher-level language such as C/C++. Further hardware control is enabled by the use of HLS PRAGMAS. Designs made using HLS are typically not the most optimal possible solutions, even after several optimizations, but are often considered acceptable considering the low engineering cost (NRE) and the throughput gains.

In this part of the lab, we will only get started with HLS, by reimplementing the 16-bit multiplier using C++ and HLS to generate the AXI-Lite slave IP block.

### 2.1 Creating the Vitis HLS project

1. Open Vitis HLS->PROJECT->Create Project.
2. Call the project hls\_multiplier and locate it in your group's working folder.
3. Similarly, in the Top Function field, write hls\_multiplier. The Top Function is the C/C++ function that will be translated to HDL by the HLS algorithm. If it calls

other functions, they will also be translated to HDL. Skip Add/Remove Testbench Files.

4. Leave the Solution Name as default, the Period is 10 ns since the board frequency is 100 MHz, and for the Part Selection, please select xc7z020clg400-1 of Family Zynq-700, Speed = -1, Flow Target set to be Vivado IP Flow Target. You will notice that Vitis HLS looks less “busy” compared to Vivado. This is because Vivado HLS only simulates, synthesizes, and packages the IP block, but does not interface with the hardware directly. The output from Vitis HLS is to be later imported into a Vivado project. Let’s add the necessary code to implement and test the HLS multiplier AXI Lite slave IP block.
5. In the Explorer, right-click on Source and click Add Source File... Add lab2\_starter/hls\_multiplier.cpp. The code is really simple. The hls\_multiplier C++ function takes in two unsigned short int arguments  $a$  and  $b$ , 16-bits each, and returns an unsigned int which is 32-bit. Inside, the product  $p$  is declared as a 32-bit unsigned int, is given the value  $a \times b$  and is returned. The `#pragma HLS INTERFACE` lines specify that the interface to be used is a slave AXI Lite and are all into the same bundle.
6. In the Explorer, right-click on Test Bench and click Add Test Bench File... Add lab2\_starter/test\_hls\_multiplier.cpp. Again, the code is very straightforward - please read it and understand what is going on.

## 2.2 Testing the hls\_multiplier and exporting the IP block

Vitis HLS offers a few ways to test your design, both as C Simulation and as C/RTL Cosimulation. To start a simulation - use the comfortably ordered buttons on the top area.

1. Project menu->Index C source.
2. Project menu->Run C Simulation or click the black triangle button next to the green triangle button, click C Simulation. In the pop-up C Simulation Dialog, you can check Clean Build in the Options frame. Click OK. Observe the Vitis HLS Console. After the C simulation is finished, hls\_multiplier\_csim.log will pop up, showing:

```
INFO: [SIM 2] ***** CSIM start *****
INFO: [SIM 4] CSIM will launch GCC as the compiler.
Compiling ../../../../lab2_starter/test_hls_multiplier.cpp in
  → debug mode
Compiling ../../../../lab2_starter/hls_multiplier.cpp in debug
  → mode
Generating csim.exe
initialized variables: a=2, b=3, p=0
testing hls_multiplier: 2 * 3 = 6
INFO: [SIM 1] CSim done with 0 errors.
INFO: [SIM 3] ***** CSIM finish *****
```

3. Solution menu->Run C Synthesis->Active Solution or click the black triangle button next to the green triangle button, click C Synthesis. You can keep default settings in the pop-up dialog C Synthesis - Active Solution. This step starts the translation of the C/C++ code to HDL (both Verilog and VHDL), synthesizes it for the targeted chip and the targeted frequency, and generates a report. After the synthesis is finished, you can observe "Finished C synthesis" in Vitis HLS Console and view the Synthesis Summary Report of 'hls\_multiplier' in the pop-up tab Synthesis Summary(solution1). Read the Timing Estimate and Performance & Resource Estimates section especially. Note that the organization of the report is very different from those generated by previous versions of Vitis HLS (or originally Vivado HLS).

Synthesis Summary Report of 'hls\_multiplier'

General Information

Date:Mon Sep 19 14:46:19 2022

Version: 2022.1.2 (Build 3605665 on Fri Aug 5 22:53:37 MDT 2022)

Project:hls\_multiplier

Solution:solution1 (Vivado IP Flow Target)

Product family:zynq

Target device:xc7z020-clg400-1

Timing Estimate

Target	Estimated	Uncertainty	
10.00 ns	3.150 ns	2.70 ns	

Performance & Resource Estimates ⓘ

Modules

Loops

Modules & Loops	Issue Type	Violation Type	Distance	Slack	Latency(cycles)	Latency(ns)	Iteration Latency	Interval	Trip Count	Pipelined	BRAM	DSP	FF	LUT	URAM	
<div><div></div>hls_multiplier</div>				-	3	30.000		-	4	-	no	0	1	116	193	0

HW Interfaces

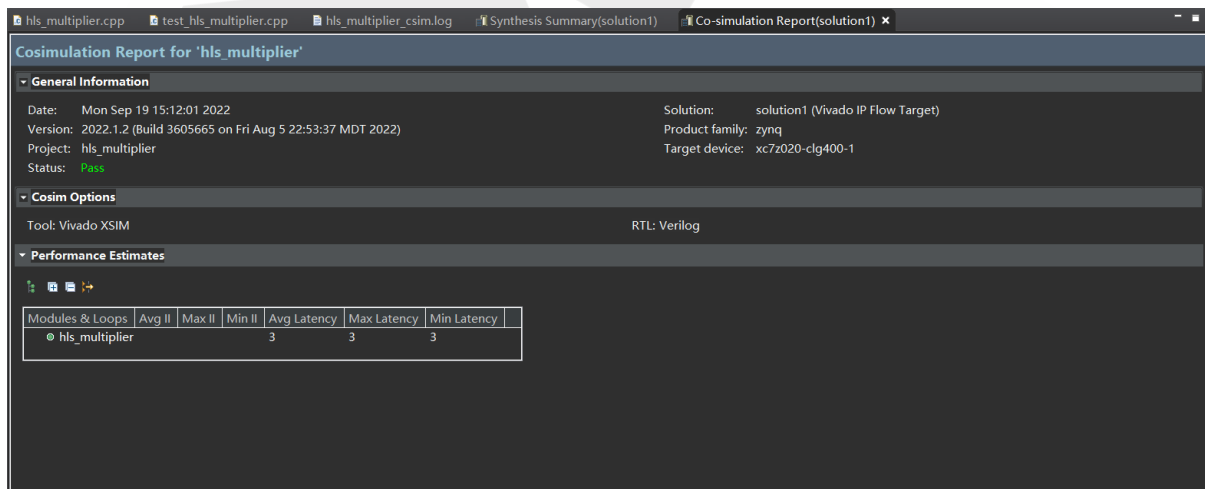
4. Now that everything seems reasonable at the C Simulation level and at the C Synthesis level, click the black triangle button next to the green triangle button, click Cosimulation or run Flow Navigator->C/RTL COSIMULATION->Run Cosimulation. Use the default settings in the Co-simulation Dialog (Vivado XSIM, Verilog). It does the simulation of both the C and the generated RTL and compares the final results, and ensures that they match. After the cosimulation finished, Vitis HLS Console outputs:

```
INFO: [Common 17-206] Exiting xsim at Mon Sep 19 15:12:00 2023...
INFO: [COSIM 212-316] Starting C post checking ...
initialized variables: a=2, b=3, p=0
testing hls_multiplier: 2 * 3 = 6
INFO: [COSIM 212-1000] *** C/RTL co-simulation finished: PASS ***
INFO: [COSIM 212-211] II is measurable only when transaction number
→ is greater than 1 in RTL simulation. Otherwise, they will be
→ marked as all NA. If user wants to calculate them, please make
→ sure there are at least 2 transactions in RTL simulation.
INFO: [HLS 200-111] Finished Command cosim_design CPU user time: 1
→ seconds. CPU system time: 0 seconds. Elapsed time: 18.82
→ seconds; current allocated memory: 0.000 MB.
```

INFO: [HLS 200-112] Total CPU user time: 7 seconds. Total CPU system  
 ↳ time: 1 seconds. Total elapsed time: 24.436 seconds; peak  
 ↳ allocated memory: 1.537 GB.

Finished C/RTL cosimulation.

In the generated Cosimulation Report for 'hls\_multiplier' in the pop-up tab Co-Simulation Report(solution1), you are expected to see Status: **Pass** - which means the test was successful. Now that all these tests passed, it's time to export the design to an IP block.

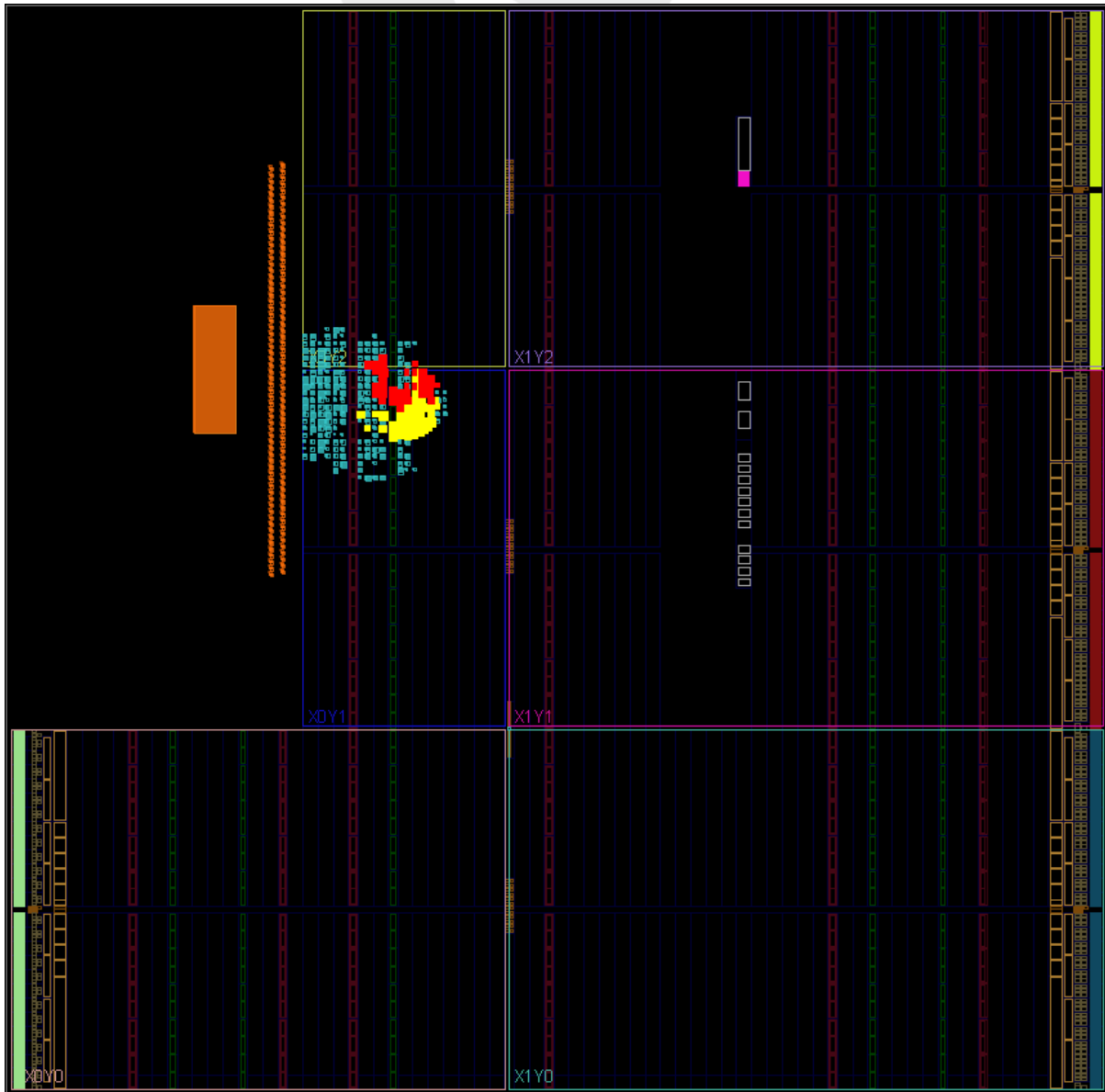


5. Solution menu->Export RTL or click the black triangle button next to the green triangle button, click Export RTL. In the pop-up dialog Export RTL, choose the Output Location, then click OK. Wait until Vitis HLS Console shows "Finished Export RTL/Implementation". You can now close Vivado HLS.

## 2.3 Adding the hls\_multiplier AXI Lite Slave IP Block to your Vivado design

1. Open your Vivado project for Lab 1 or create a new Vivado project and create the Zynq\_CPU block design with a ZYNQ7 Processing System and its HDL wrapper. In Tools->Settings->Project Settings->IP->Repository. Add the hls\_multiplier/solution1/impl/ip directory. This will allow Vivado to find the IP we just created using Vitis HLS. If you have not add the rtl\_multiplier\_v1.0 IP, also add the ip\_repo/rtl\_multiplier\_1.0 directory.
2. Next, in the Block Design, click on Add IP and search for the Hls\_multiplier. Add it to your design. If you have not add rtl\_multiplier\_v1.0 IP, also add rtl\_multiplier\_v1.0. Run Connection Automation to automatically have it connected to the AXI Interconnect.
3. Save all files and click on Generate Bitstream to (re-)run Synthesis and Implementation over the (updated) design. Check the resource utilization and look over the implementation. In the following image, the hls\_multiplier area is highlighted yellow,

while the rtl\_multiplier area is highlighted red. To highlight hls\_multiplier\_area and rtl\_multiplier area, Open Implemented Design, in the Netlist tab, choose Zynq\_CPU\_wrapper->Zynq\_CPU\_i (Zynq\_CPU), right click hls\_multiplier\_0 (Zynq\_CPU\_hls\_multiplier\_0\_0) and rtl\_multiplier\_0 (Zynq\_CPU\_rtl\_multiplier\_0\_0), select "Highlight Leaf Cells"->Default/Color 2 and Color 7 respectively. Note that the resource utilization images in the Device window generated by different people are different. Example results are shown below.



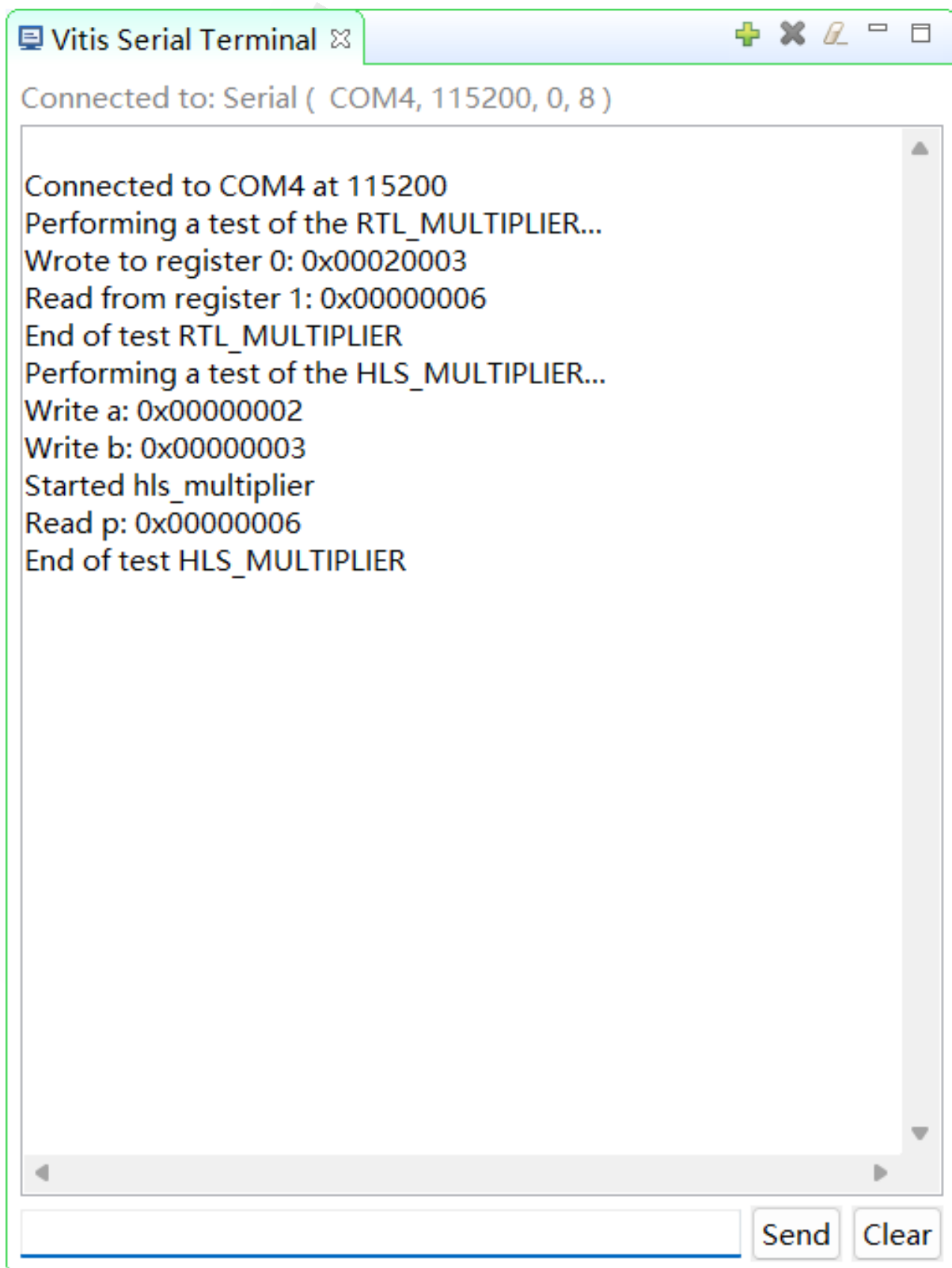
## 2.4 Interfacing with the RTL and HLS blocks in Software

1. As what has been practiced in Lab#1, Export Hardware... as Zynq\_CPU\_wrapper\_hw\_platform\_4.xsa and Launch Vitis.
2. Create a new application called hello\_hls\_rtl\_multiplier starting with the Hello World template.



- 
3. Right-click Explorer->hello\_hls\_rtl\_multiplier\_system->hello\_hls\_rtl\_multiplier->src, click Import Sources to replace helloworld.c with lab2\_starter/hls\_rtl\_multiplier.c through right. The first part of this code does the same as before, testing the rtl\_multiplier in Lab #1. The second part of the code tests the hls\_multiplier. Vitis HLS generates not only an HDL implementation of the IP block but also functions that enable the user to use the IP block correctly. Make sure you understand the code.

4. Build and run the code on the Arty Z-7 board. The result from the Vitis Serial Terminal is shown below:

A screenshot of a Vitis Serial Terminal window. The window title is 'Vitis Serial Terminal'. The status bar at the top indicates 'Connected to: Serial ( COM4, 115200, 0, 8 )'. The main text area shows the following output:

```
Connected to COM4 at 115200
Performing a test of the RTL_MULTIPLIER...
Wrote to register 0: 0x00020003
Read from register 1: 0x00000006
End of test RTL_MULTIPLIER
Performing a test of the HLS_MULTIPLIER...
Write a: 0x00000002
Write b: 0x00000003
Started hls_multiplier
Read p: 0x00000006
End of test HLS_MULTIPLIER
```

At the bottom of the window, there is a text input field and two buttons labeled 'Send' and 'Clear'.



## 2.5 Follow-up Exercises

In this task, you are required to implement a 32-bit ALU (Arithmetic Logic Unit), i.e. a block that can perform operations defined in RV32IM (RISC-V 32-bit Base Integer Instruction Set Version 2.1 + “M” Standard Extension for Integer Multiplication and Division Version 2.0) [5]:

Operation	Code	Selector ID
Add	ALU_ADD	0
Subtract	ALU_SUB	1
Set less than	ALU_SLT	2
Set less than (unsigned)	ALU_SLTU	3
And	ALU_AND	4
Or	ALU_OR	5
Exclusive or	ALU_XOR	6
Shift left (logical)	ALU_SLL	7
Shift right (logical)	ALU_SRL	8
Shift right (arithmetic)	ALU_SRA	9
Multiply	ALU_MUL	10
Multiply (upper XLEN bits, signed $\times$ signed)	ALU_MULH	11
Multiply (upper XLEN bits, signed $\times$ unsigned)	ALU_MULHSU	12
Multiply (upper XLEN bits, unsigned $\times$ unsigned)	ALU_MULHU	13
Divide	ALU_DIV	14
Divide (unsigned)	ALU_DIVU	15
Remainder	ALU_REM	16
Remainder (unsigned)	ALU_REMU	17

**Table 1:** RV32IM ALU Operations.

Based on lab2\_starter/rtl\_alu.sv, create and package a new IP `rtl_alu` in Vivado. In this part, you only need to complete the `rtl_alu` module. Suppose you have two 32-bit inputs, one 5-bit input, and one 32-bit output, you will need four registers, which are specified when you add AXI interfaces during the "Create and Package New IP" process. In Vitis HLS, create a new project `hls_alu` with source `hls_alu.cpp` and test bench `test_hls_alu.cpp`. In the test bench, test the functional correctness of all 18 operations. Run C simulation and take a screenshot of `hls_alu_csim.log`; run C synthesis and take a screenshot of the Timing Estimate and Performance & Resource Estimates in the Synthesis Summary Report; run cosimulation and take a screenshot of the Cosimulation Report.

Add `rtl_alu` and `hls_alu` to the block design. Create an HDL wrapper and generate the bitstream. Take a screenshot of the Device highlighting `rtl_alu` and `hls_alu` leaf cells and screenshots of the Summary in the Utilization report and the Design Timing Summary in the Timing report in your Implemented Design. Pay attention to whether there are latches. Compare the `rtl_alu` and `hls_alu` and summarize any differences you observed. Then, export hardware as `Zynq_CPU_wrapper_hw_platform_5.xsa`.

Create a Vitis application project named `hello_hlr_rtl_alu` from the Hello World template and import the hardware. Replace `helloworld.c` with `hls_rtl_alu.c`. In the project, test all 18 operations of both `rtl_alu` and `hls_alu`. Take a screenshot of your Vitis IDE window at the build-finished state and a screenshot of the output in the Vitis Serial Terminal.

### 3 Program the FPGA and Boot Software from the SD Card

The following process outlines how you can take your developed C-code and boot it from the SD card inserted in the FPGA, which saves the effort of having to run the software from Vitis.

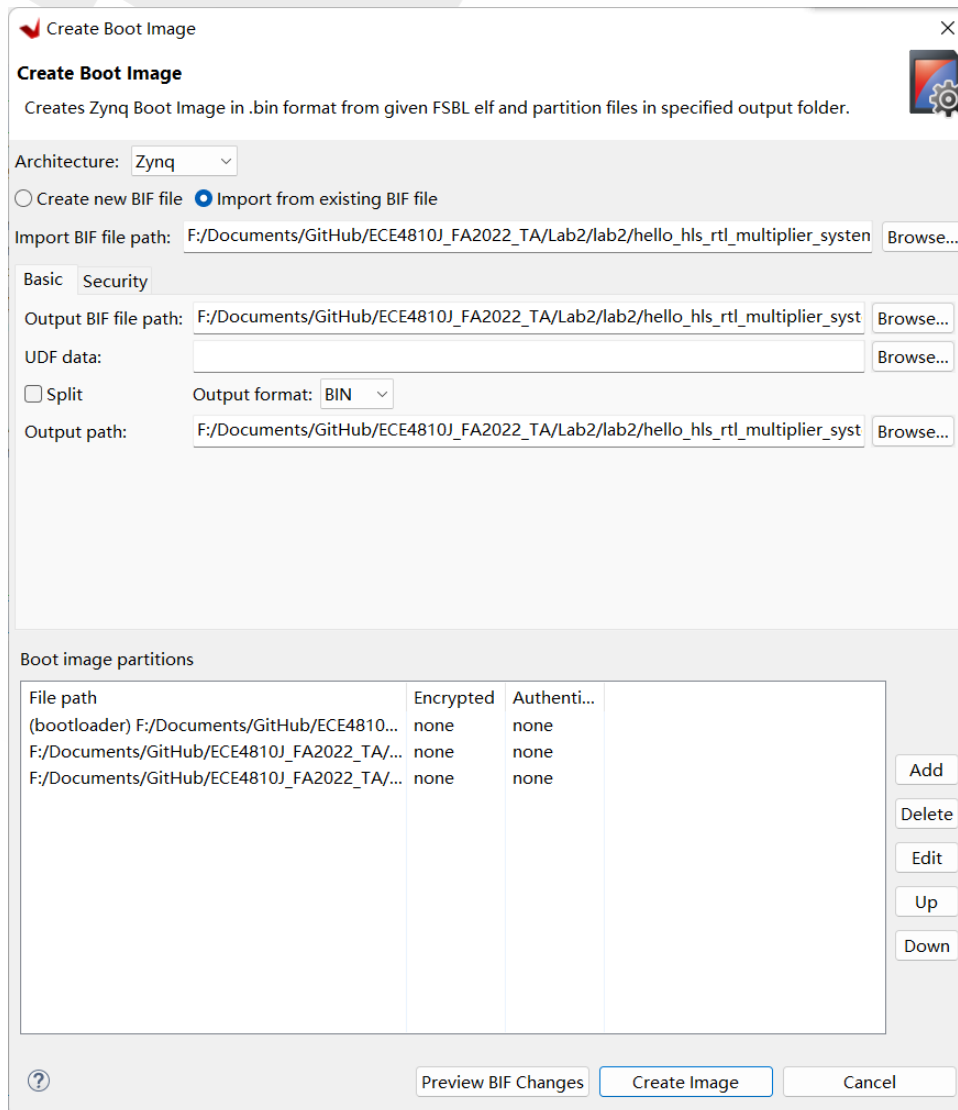
1. Open Vitis in your previous workspace. First, replace the `hls_rtl_multiplier.c` file of your `hello_hls_rtl_multiplier` project with `lab2_starter/hls_rtl_m.c`. The modified version of the code creates an infinite loop where the user inputs the operands for the RTL and HLS multipliers, respectively, which is a necessary modification when the code will continually have to execute from the SD card.
2. In the Assistant window, right-click `Zynq_CPU_wrapper_hw_platform_4` [Platform], click "Open Platform Editor". In the platform editor, click `Zynq_CPU_wrapper_hw_platform_4` -> `ps7_cortexa9_0` -> `standalone_ps7_cortexa9_0` -> Board Support Package. Click "Modify BSP Settings..."; in the pop-up dialog Board Support Package Settings, in the Supported Libraries frame of the Overview page, check `xilffs 4.7 Generic Fat File System Library`.

```
platform read {F:\Documents\GitHub\ECE4810J_FA2024_TA\Lab2\lab2\Zynq_
  ↳ q_CPU_wrapper_hw_platform_4\platform.spr}
platform active {Zynq_CPU_wrapper_hw_platform_4}
::scw::get_hw_path
::scw::regenerate_psinit
  ↳ F:/Documents/GitHub/ECE4810J_FA2024_TA/Lab2/lab2/Zynq_CPU_wrapp
  ↳ er_hw_platform_4/hw/Zynq_CPU_wrapper_hw_platform_4.xsa
::scw::get_mss_path
::scw::get_target
bsp setlib -name xilffs -ver 4.7
bsp write
bsp reload
bsp regenerate
```

3. In the Assistant window, right-click `Zynq_CPU_wrapper_hw_platform_4` [Platform], click "Clean" then click "Build". Also, re-build your `hello_hls_rtl_multiplier` and fix Makefile and `qemu_args.txt` bugs if needed.
4. Create a New Project by going to File -> New -> Application Project. For Platform, in the "Select a platform from repository" tab, select `Zynq_CPU_wrapper_hw_platform_4` [custom]. Make sure the top bar shows "Choose a platform for your project. You

can also create an application from XSA through the 'Create a new platform from hardware (XSA)' tab." The top bar will show a warning or an error if your hardware platform is not prepared to be used, for example, your hardware platform will have a suffix "[Out-of-Date]". If so, clean and build your hardware platform and your hello\_hls\_rtl\_multiplier. Make sure you are using the same hardware platform as your hello\_hls\_rtl\_multiplier. For Project Name, enter "FSBL". Click Next. In the Available Templates Menu, select "Zynq FSBL". Click Finish. You have now created the FSBL project. The below figure shows what your project set-up should look like.

- Right-click on the hello\_hls\_rtl\_multiplier\_system project in the Explorer window and click "Create Boot Image". After generating bif, the pop-up dialog "Create Boot Image", has the set-up required to successfully create a bootable SD card Image of your hardware design with the chosen software to run on it. Click "Create Image".



**Create Boot Image**

Creates Zynq Boot Image in .bin format from given FSBL elf and partition files in specified output folder.

Architecture: Zynq

☐ Create new BIF file ☒ Import from existing BIF file

Import BIF file path: F:/Documents/GitHub/ECE4810J\_FA2022\_TA/Lab2/lab2/hello\_hls\_rtl\_multiplier\_system Browse...

Basic Security

Output BIF file path: F:/Documents/GitHub/ECE4810J\_FA2022\_TA/Lab2/lab2/hello\_hls\_rtl\_multiplier\_syst Browse...

UDF data: Browse...

☐ Split Output format: BIN

Output path: F:/Documents/GitHub/ECE4810J\_FA2022\_TA/Lab2/lab2/hello\_hls\_rtl\_multiplier\_syst Browse...

Boot image partitions

File path	Encrypted	Authenti...
(bootloader) F:/Documents/GitHub/ECE4810...	none	none
F:/Documents/GitHub/ECE4810J_FA2022_TA/...	none	none
F:/Documents/GitHub/ECE4810J_FA2022_TA/...	none	none

Add  
Delete  
Edit  
Up  
Down

Preview BIF Changes Create Image Cancel

The Bootgen console shows:

```
cmd /C bootgen -image hello_hls_rtl_multiplier_system.bif -arch zynq  
↪ -o \  
F:/Documents/GitHub/ECE4810J_FA2024_TA/Lab2/lab2/hello_hls_rtl_mult_  
↪ iplier_system/_ide/bootimage/BOOT.bin
```

```
***** Xilinx Bootgen v2023.1.2  
**** Build date : Aug  5 2023-23:12:34  
** Copyright 1986-2023 Xilinx, Inc. All Rights Reserved.
```

```
[WARNING]: Partition fsbl.elf.0 range is overlapped with partition  
↪ Zynq_CPU_wrapper_hw_platform_4.bit.0 memory range  
[WARNING]: Partition Zynq_CPU_wrapper_hw_platform_4.bit.0 range is  
↪ overlapped with partition hello_hls_rtl_multiplier.elf.0 memory  
↪ range
```

```
[INFO] : Bootimage generated successfully
```

6. Plug in your MicroSD card for FPGA in your computer with a MicroSD card reader/adaptor. Paste the `hello_hls_rtl_multiplier_system/_ide/bootimage/BOOT.bin` file into the MicroSD card.
7. Plug your MicroSD card back to the FPGA board. Change JP4 from JTAG to SD on Arty Z7.

Now you are ready to observe that the FPGA indeed boots software from the MicroSD card. Connect the FPGA to your PC. If the Vitis Serial Terminal has no response, or does not show "Enter operand A: Performing a test of the RTL\_MULTIPLIER..." and directly show your input number and "Enter operand B:" after your send a number, press PORB (Power-On Reset B) or SRST (external System Reset) used to reset the Zynq without disconnecting and reconnecting the power supply first, because the Arty Z7 does not have a power switch, so when a power source is connected and selected with JP5 it will always be powered on [1] [4] [3] [6]. Make sure a connection via the serial port is established either using the Vitis Serial Terminal or other serial terminals like PuTTY (in PuTTY, change Connection Type to Serial, change Serial Line to your COM port, and change Speed to 115200). Once the FPGA has been connected to your PC via the serial COM port, enter any number through the terminal. The program should start execution, and you should observe that it continually asks the user to enter operands for the RTL/HLS multipliers, respectively, and that it outputs the correct multiplication results. Congrats! Now you know how to boot software from the FPGA's MicroSD card! Take a screenshot of the Vitis Serial terminal or PuTTY terminal. Note: The same BOOT.bin file can be used to boot from the SPI Flash memory.

## 4 Program Python on Arty Z7

Python is a very powerful and flexible programming language, enabling engineers to perform complex mathematics analysis, implement Artificial Intelligence solutions, and

develop a range of other complex engineering solutions. The ability to use Python within the Field Programmable Gate Array (FPGA) space has however previously been limited. With the release of the PYNQ framework, Python developers for the first-time were able to exploit the capabilities and performance provided by programmable logic. While FPGA developers also benefited as they were able to integrate the high-level capabilities of Python with their FPGA designs.

In this lab, we will introduce the PYNQ project. The main goal of PYNQ, Python Productivity for Zynq, is to make it easier for designers of embedded systems to exploit the unique benefits of Xilinx devices in their applications. Specifically, PYNQ enables architects, engineers and programmers who design embedded systems to use Zynq devices, without having to use ASIC-style design tools to design programmable logic circuits. PYNQ achieves this goal in three ways:

- Programmable logic circuits are presented as hardware libraries called overlays. These overlays are analogous to software libraries. A software engineer can select the overlay that best matches their application. The overlay can be accessed through an Python API. Creating a new overlay still requires engineers with expertise in designing programmable logic circuits. The key difference however, is the build once, re-use many times paradigm. Overlays, like software libraries, are designed to be configurable and re-used as often as possible in many different applications.
- PYNQ supports Python for programming both the embedded processors and the overlays. Python is a “productivity-level” language. To date, C or C++ are the most common, embedded programming languages. In contrast, Python raises the level of programming abstraction and programmer productivity. These are not mutually exclusive choices, however. PYNQ uses CPython which is written in C, and integrates thousands of C libraries and can be extended with optimized code written in C. Wherever practical, the more productive Python environment should be used, and whenever efficiency dictates, lower-level C code can be used.
- PYNQ is an open-source project that aims to work on any computing platform and operating system. This goal is achieved by adopting a web-based architecture, which is also browser agnostic. We incorporate the open-source Jupyter notebook infrastructure to run an Interactive Python (IPython) kernel and a web server directly on the ARM processor of the Zynq device. The web server brokers access to the kernel via a suite of browser-based tools that provide a dashboard, bash terminal, code editors and Jupyter notebooks. The browser tools are implemented with a combination of JavaScript, HTML and CSS and run on any modern browser.

Please read more details about PYNQ from [PYNQ Introduction — Python productivity for Zynq \(Pynq\)](#).

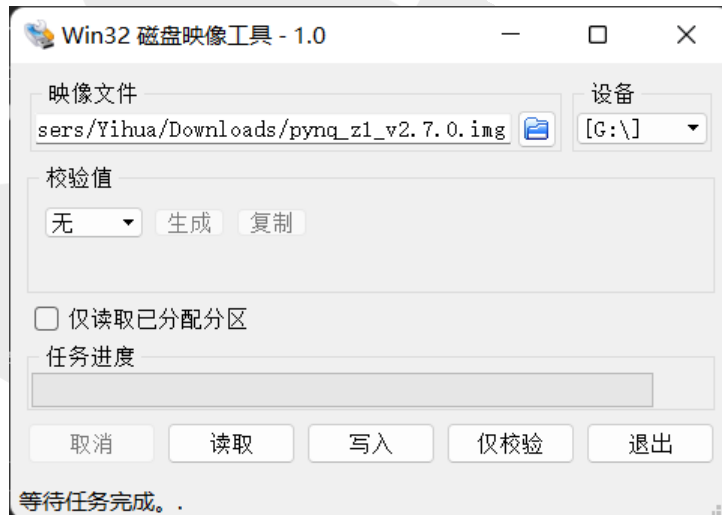
To get started with the tutorial, one must first pick up an Arty Z7-20 as well as a MicroSD card. The first steps involve getting the PYNQ image up and running on the Arty. Once this is complete, the ISO File must be burned to the MicroSD card (which should be at least 8 gigs if possible).

1. Here is a list of PYNQ images that correspond to different boards, please choose

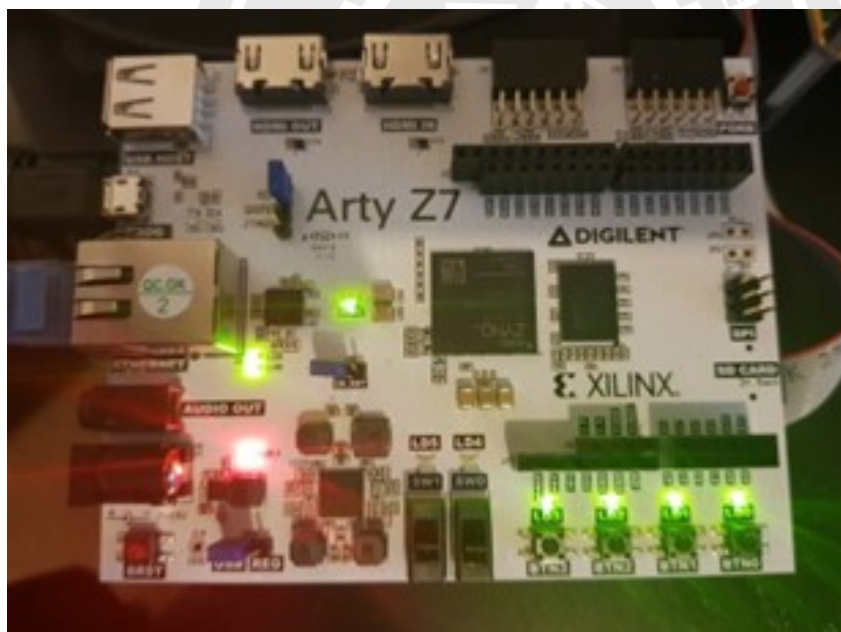


the PYNQ-Z1 image. <http://www.pynq.io/board.html>.

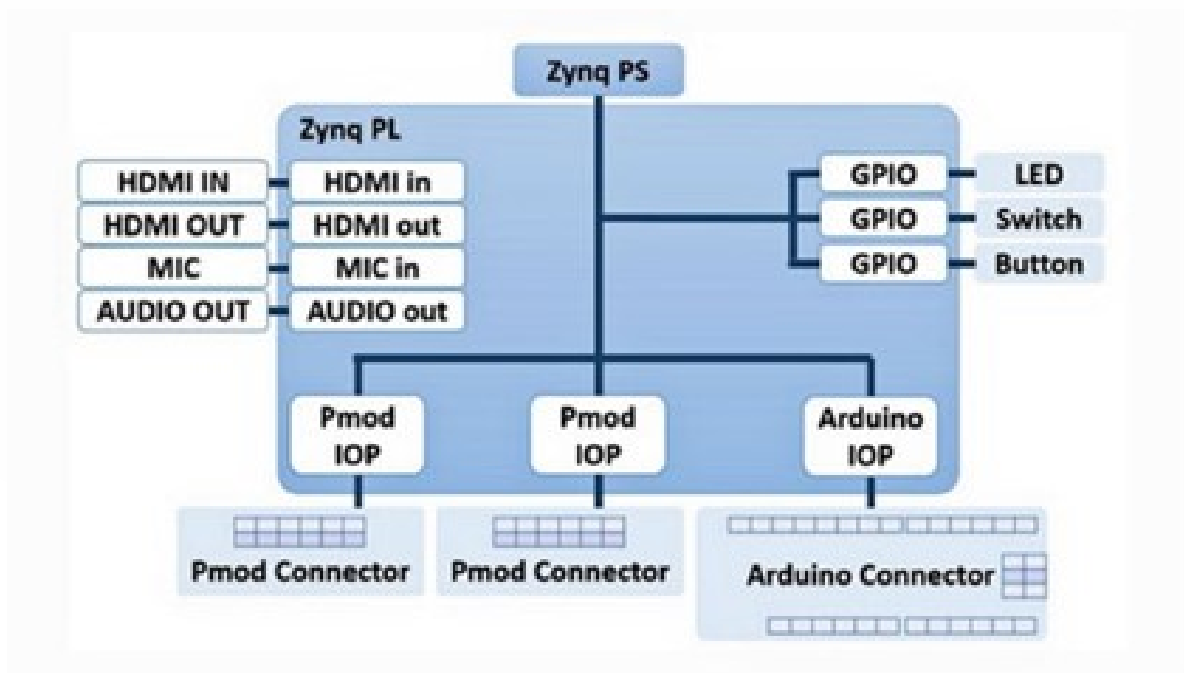
2. To burn the ISO image to the SD card, you can use imager tools like [Win32 Disk Imager](#).



3. Once the image is written to the MicroSD Card, it is time to plug the card into the Arty Z7-20, ensure the configuration jumper is set to boot from SD, and connect an Ethernet cable to your computer before powering it on.
4. Power on the board, and after tens of seconds the yellow LD10, the green LD8 and L9 blinking, you will see the Done LD12 illuminates, indicating the Xilinx Zynq has been configured, followed by activity on the Ethernet LEDs. Once the boot sequence is complete, and the PYNQ framework is ready. The four LEDs and two RGB LEDs will flash several times before leaving the four LEDs illuminated.



5. We are now ready to leverage Python on Arty Z7-20, we do this by using Jupyter notebooks running on the Arty Z7-20. To connect to the Arty Z7-20 and its Jupyter notebooks, we need to open a browser on the same network as the Arty Z7-20, entering the address `pynq:9090` this will open the initial page as shown below. It is from here that we can develop our Python-based applications. If you cannot open `pynq:9090` because of HTTP errors like 500 or 503, see Appendix B.
6. Take a screenshot of the Jupyter Notebook including the address bar of your browser. Once on the landing page, open the `getting_started` directory and read the information on the Jupyter notebooks, Python environments, and advanced Python features. These will provide a great introduction to how to get started using the development environment.
7. While the Linux image with its Jupyter notebooks and Python run on the processor cores of the Zynq, the programmable logic is used to provide several interfaces and programmable logic overlays. These overlays can be loaded and used from within the Jupyter environment as your Python script executes. The PYNQ image comes with a base overlay that supports all the inputs and outputs (IO) on Arty Z7-20.



8. Of course, it is possible to develop custom overlays. These custom overlays are developed using either a traditional flow of Vivado Design Suite including SDK and High-Level Synthesis using Vivado HLS or SDSoc Development Environment. To enable integration within the Python environment, the software (SW) drivers for the programmable logic (PL) are exploited using Python's C Foreign Function Interface.



---

## 5 (Optional) Optimize Your RTL ALU by Pipelining

In Section 2, you probably simply implement multipliers and dividers in a single clock cycle, the timing violation will be serious, and a huge amount of programmable logic will cost, thus, you will need to pipeline your multiplier and divider in order to derive more acceptable performance, but you do not need to achieve the optimal pipelined performance.

In this section, you are expected to create and package a new IP `rtl_pipe_alu` that combines the pipelined multiplier and divider with the RTL ALU you wrote in Section 2. Add it into a block design together with your previously created and packaged IP `hls_alu` and the ZYNQ7 Processing System. Make an HDL wrapper, generate the bitstream, take a screenshot of the Implemented Design highlighting RTL ALU and HLS ALU leaf cells, take a screenshot of the Utilization Summary and the Design Timing Summary of the Implemented Design, compare them with the implemented design of the RTL ALU in Section 2, export the hardware XSA, create a Vitis application project using the XSA, write an application for testing multiplication and division, build the project, run the project, take a screenshot of the Vitis Serial Terminal to verify the functional correctness of your pipelined multiplier and divider.

If you feel difficult about writing a pipelined divider on your own, you can try to alternatively replace your current design of dividers with the Divider Generator IP of Xilinx LogiCORE IP Product.

## 6 Post-lab Questions

1. What was the resource utilization of the RTL implementation vs. HLS implementation? If they are different, can you guess why there are differences?
2. Imagine that you have to create an IP block implementing ten different sorting algorithms. Would you rather use a hardware description language or High-level Synthesis? Why?
3. Read more about PYNQ, and answer the following questions:
  - What is a Jupyter notebook? Why do we use it?
  - What are overlays? What are the benefits of using overlays?
  - Please list available overlays on the Arty Z7 board.
  - Comparing part 2 where you design with C/C++ vs. part 4 (PYNQ framework with python), what are the major differences?

---

## 7 Deliverables

Group Deliverables (Compile everything as a single pdf report besides the code):

- Section 2:
  - the screenshot of the Implemented Design highlighting RTL multiplier and HLS multiplier leaf cells
  - the screenshot of the Utilization Summary and the Design Timing Summary of the Implemented Design
  - the screenshot of the Vitis Serial Terminal of hello\_hls\_rtl\_multiplier Vitis project
  - the screenshot of the Synthesis Summary Report of hls\_alu containing Timing Estimate and Performance & Resource Estimates
  - the screenshot of the generated Cosimulation Report for hls\_alu
  - the screenshot of the resource utilization of the Implemented Design highlighting RTL ALU and HLS ALU leaf cells
  - the screenshot of your Vitis IDE window at the build-finished state of hello\_hls\_rtl\_alu Vitis project
  - the screenshot of the Vitis Serial Terminal of hello\_hls\_rtl\_alu Vitis project
  - the code rtl\_alu.sv for Vivado Design Source
  - the code hls\_alu.cpp for Vitis HLS Source
  - the code test\_hls\_alu.cpp for Vitis HLS Test Bench
  - the code hls\_rtl\_alu.c for Vitis Application Project hello\_hls\_rtl\_alu src
- Section 3: the screenshot of the Vitis Serial terminal or PuTTY terminal
- Section 4: the screenshot of the Jupyter notebook including the address bar of your browser
- Section 5:
  - the screenshot of the Implemented Design highlighting RTL ALU and HLS ALU leaf cells
  - the screenshot of the Utilization Summary and the Design Timing Summary of the Implemented Design
  - the screenshots of the Vitis Serial Terminal of pipe\_hls\_rtl\_alu Vitis project
  - the code deliverables are the same as Section 2; if you finish this part, submit your code used for this section instead of the code for Section 2
- Section 6: answers

## 8 Grading policy

Factors	Percentage
Section 2	45%
Section 3	20%
Section 4	20%
Section 5	20% (bonus)
Section 6	15%

## 9 Acknowledgment

We would like to acknowledge previous TA Mr. Yihua Liu (FA 2022) for helping develop this lab and laying solid foundation for this course. Special thanks to the AMD-Xilinx University Program for providing the starters' files and ideas for this lab.



## A Peer Evaluation Form

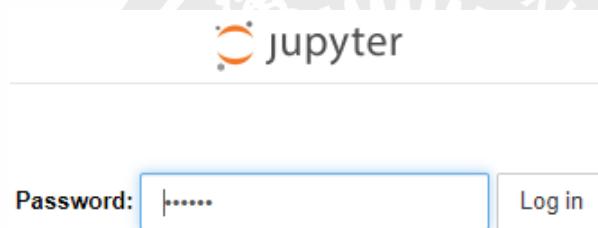
Part	Your work	Your partner's work	Your score	Your partner's score
Section 2				
Section 3				
Section 4				
Section 5				
Section 6				

## B Troubleshooting

### B.1 PYNQ HTTP Connection

If you could not connect to the pynq after you have set up your board and LEDs illuminate properly, you should do the following configuration:

Quit any VPN connections and proxy connections. For Windows 10/11, go Control Panel->Network and Internet->Network and Sharing Center->View your active networks->Access type: Internet, Connections: WLAN (or the network you are currently connecting, since you may connect the board and your computer by a cable, you are probably using WLAN)->Properties [2]. In the pop-up window <network\_name> Properties, go tab Sharing, in the Internet Connection Sharing frame, check "Allow other network users to connect through this computer's Internet connection", and for Home networking connection: Select a private network connection, select the proper connection (usually it is ethernet). Refresh pynq:9090, you will be able to connect. The password is xilinx by default.



## References

- [1] Sam Bobrowicz. *Arty Z7-20 Petalinux BSP Project Quick-Start Guide. Run the pre-built image from SD.* Oct. 11, 2017. URL: <https://github.com/sbobrowicz/Petalinux-Arty-Z7-20#run-the-pre-built-image-from-sd>.
- [2] demi. *PYNQ classic project sharing - Hello World.* Dec. 2, 2019. URL: <http://xilinx.eetrend.com/content/2019/100046325.html>.
- [3] Digilent. *Arty Z7 Reference Manual.* 2022. URL: <https://digilent.com/reference/programmable-logic/arty-z7/reference-manual>.
- [4] jpeyron. *Arty Z7 Zynq setup step-by-step.* Sept. 6, 2017. URL: <https://forum.digilent.com/topic/4760-arty-z7-zynq-setup-step-by-step/>.

- 
- [5] Andrew Waterman and Krste Asanović. *The RISC-V Instruction Set Manual. Unprivileged ISA*. Tech. rep. Version 20191213. RISC-V International, Dec. 2019.
  - [6] Xilinx. *UG585 Zynq-7000 SoC Technical Reference Manual*. Version v1.13. Apr. 2, 2021. URL: <https://docs.xilinx.com/v/u/en-US/ug585-Zynq-7000-TRM>.

