ECE4810J System-on-Chip (SoC) Design

Fall 2021

Lab #2 Design with high level languages on Arty Z7 SoC development platform

Due: 11:59pm Oct. 14th, 2021

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.

1. Overview

In this lab, you will go through two exercises to create design with high level languages (instead of Verilog). 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 notebook with the board

2. Creating and using 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, difficult to debug and verify, difficult to update, etc. Moreover, it can not be done by software engineers without an intense training on 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.

Creating the Vivado HLS project

- Open Vivado HLS and proceed with Create New Project.
- Call the project hls_multiplier and locate it in your group's working folder.
- 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.
- Leave the Solution Name as default, the Period is 10 ns since the board frequency is 100 MHz, and for the Part please select xc7z020clg400-1, which is the chip on the PYNQ-Z1 board.

You will notice that Vivado 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 Vivado 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.

- In the *Explorer*, right click on *Source* and select *New File...*. Locate the file in the suggested directory and name it *hls_multiplier.cpp*. The code for this file is given below.

```
unsigned int hls_multiplier(unsigned short int a, unsigned short int b) {
#pragma HLS INTERFACe s_axilite port=return bundle=CRTLS
#pragma HLS INTERFACe s_axilite port=a bundle=CRTLS
#pragma HLS INTERFACe s_axilite port=b bundle=CRTLS
    unsigned int p;
    p = a * b;
    return p;
}
```

The code is really quite 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 * 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.

- Next, right-click on *Test Bench* and select *New File...*. Similarly, store the file in the suggested location and name it *test_hls_multiplier.cpp*. The code for this file is presented below.

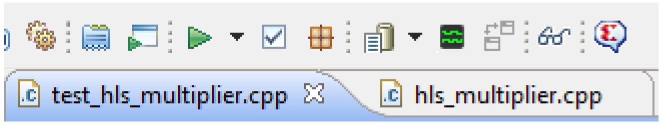
```
#include <stdio.h>
unsigned int hls_multiplier(unsigned short int a, unsigned short int b);
```

```
int main() {
    unsigned short int a, b;
    unsigned int p;
    a = 2;
    b = 3;
    p = 0;
    printf("initialized variables: a=%d, b=%d, p=%d \n", a, b, p);
    p = hls_multiplier(a, b);
    printf("testing hls_multiplier: %d * %d = %d \n", a, b, p);
    return 0;
}
```

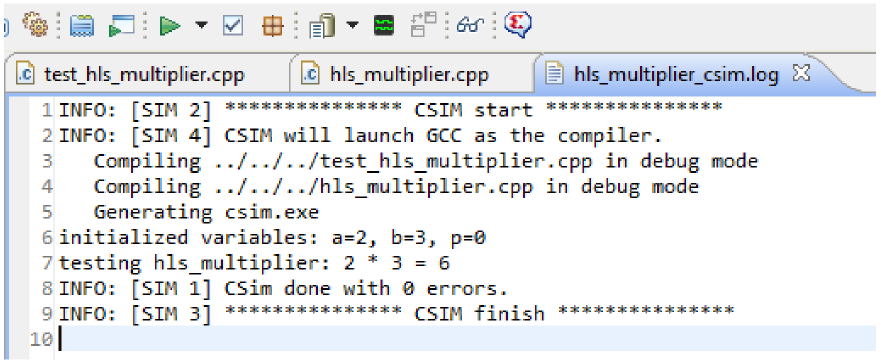
Again, the code is very straightforward - please read it and understand what is going on.

Testing the hls_multiplier and exporting the IP block

Vivado 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.

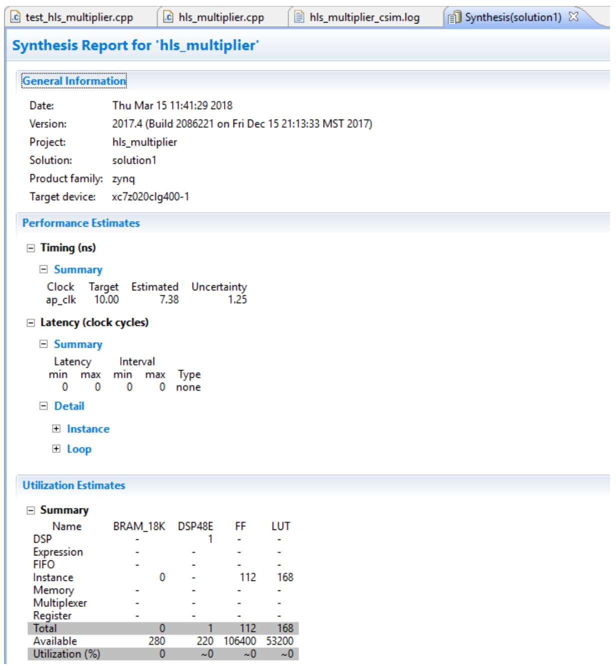


- First, click the Index C Source button (and watch as it doesn't really do anything visible).
- Next, click on the Run C Simulation button. In the C Simulation Dialog feel free to enable the Clean Build option. You should see the success message testing hls_multiplier: 2 * 3 = 6.

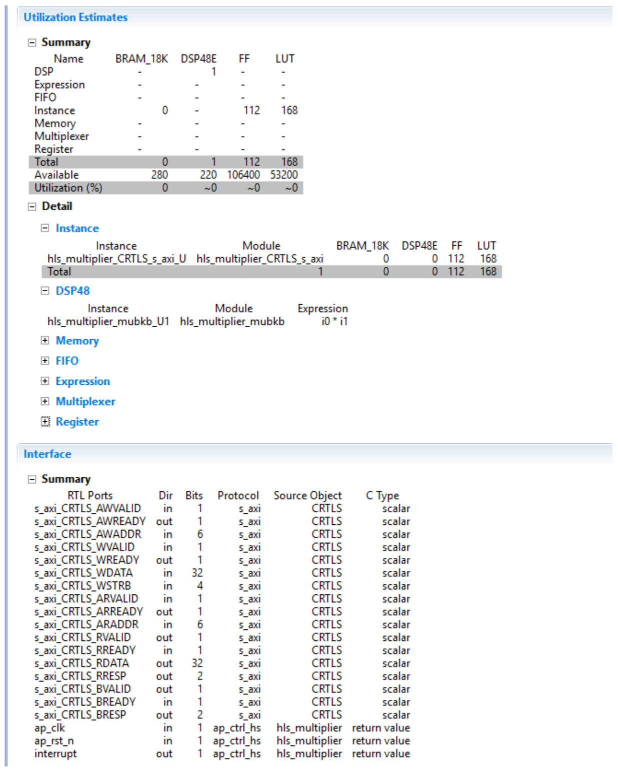


```
INFO: [SIM 2] *************** CSIM start ***************
INFO: [SIM 4] CSIM will launch GCC as the compiler.
   Compiling ../../../test_hls_multiplier.cpp in debug mode
   Compiling ../../../hls_multiplier.cpp in debug mode
   Generating csim.exe
initialized variables: a=2, b=3, p=0
testing hls_multiplier: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

- Next, click on the C Synthesis button. This button 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 containing timing information and even an Utilization Estimate. Do you observe anything interesting in the Utilization Estimate?



```
test_hls_multiplier.cpp   hls_multiplier.cpp   hls_multiplier_csim.log   Synthesis(solution1)

Synthesis Report for 'hls_multiplier'

General Information
  Date:            Thu Mar 15 11:41:29 2018
  Version:         2017.4 (Build 2086221 on Fri Dec 15 21:13:33 MST 2017)
  Project:         hls_multiplier
  Solution:        solution1
  Product family:  zynq
  Target device:   xc7z020clg400-1

Performance Estimates
  Timing (ns)
    Summary
      Clock   Target  Estimated  Uncertainty
      ap_clk   10.00       7.38         1.25
  Latency (clock cycles)
    Summary
       Latency    Interval
      min  max   min  max  Type
        0    0     0    0  none
    Detail
      Instance
      Loop

Utilization Estimates
  Summary
      Name        BRAM_18K  DSP48E      FF    LUT
  DSP                    -       1       -      -
  Expression             -       -       -      -
  FIFO                   -       -       -      -
  Instance               0       -     112    168
  Memory                 -       -       -      -
  Multiplexer            -       -       -      -
  Register               -       -       -      -
  Total                  0       1     112    168
  Available            280     220  106400  53200
  Utilization (%)        0      ~0      ~0     ~0
```



```
Utilization Estimates
  Summary
      Name        BRAM_18K  DSP48E      FF    LUT
  DSP                    -       1       -      -
  Expression             -       -       -      -
  FIFO                   -       -       -      -
  Instance               0       -     112    168
  Memory                 -       -       -      -
  Multiplexer            -       -       -      -
  Register               -       -       -      -
  Total                  0       1     112    168
  Available            280     220  106400  53200
  Utilization (%)        0      ~0      ~0     ~0
  Detail
    Instance
             Instance                    Module              BRAM_18K  DSP48E   FF  LUT
      hls_multiplier_CRTLS_s_axi_U  hls_multiplier_CRTLS_s_axi        0       0  112  168
      Total                                                1          0       0  112  168
    DSP48
            Instance                 Module            Expression
      hls_multiplier_mubkb_U1  hls_multiplier_mubkb      i0 * i1
    Memory
    FIFO
    Expression
    Multiplexer
    Register

Interface
  Summary
        RTL Ports          Dir  Bits  Protocol    Source Object   C Type
  s_axi_CRTLS_AWVALID      in     1      s_axi           CRTLS    scalar
  s_axi_CRTLS_AWREADY      out    1      s_axi           CRTLS    scalar
  s_axi_CRTLS_AWADDR       in     6      s_axi           CRTLS    scalar
  s_axi_CRTLS_WVALID       in     1      s_axi           CRTLS    scalar
  s_axi_CRTLS_WREADY       out    1      s_axi           CRTLS    scalar
  s_axi_CRTLS_WDATA        in    32      s_axi           CRTLS    scalar
  s_axi_CRTLS_WSTRB        in     4      s_axi           CRTLS    scalar
  s_axi_CRTLS_ARVALID      in     1      s_axi           CRTLS    scalar
  s_axi_CRTLS_ARREADY      out    1      s_axi           CRTLS    scalar
  s_axi_CRTLS_ARADDR       in     6      s_axi           CRTLS    scalar
  s_axi_CRTLS_RVALID       out    1      s_axi           CRTLS    scalar
  s_axi_CRTLS_RREADY       in     1      s_axi           CRTLS    scalar
  s_axi_CRTLS_RDATA        out   32      s_axi           CRTLS    scalar
  s_axi_CRTLS_RRESP        out    2      s_axi           CRTLS    scalar
  s_axi_CRTLS_BVALID       out    1      s_axi           CRTLS    scalar
  s_axi_CRTLS_BREADY       in     1      s_axi           CRTLS    scalar
  s_axi_CRTLS_BRESP        out    2      s_axi           CRTLS    scalar
  ap_clk                   in     1  ap_ctrl_hs  hls_multiplier  return value
  ap_rst_n                 in     1  ap_ctrl_hs  hls_multiplier  return value
  interrupt                out    1  ap_ctrl_hs  hls_multiplier  return value
```

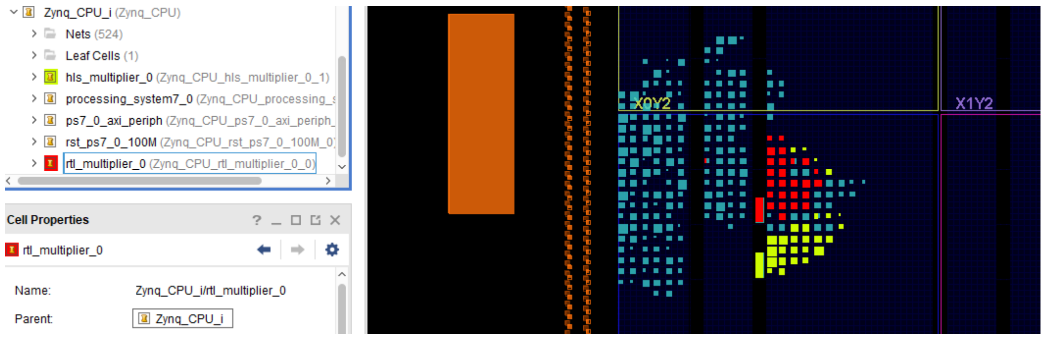
- Now that everything seems reasonable at the C Simulation level and at the C Synthesis level, click on the Run C/RTL Cosimulation button. Use the default settings in the Co-simulation

Dialog. This can take a long time since this is quite a powerful operation. It simulates both the C and the generated RTL and compares the final results and ensures that they match. In the generated report you are expecting the word Pass - which means the test was successful. Now that all these tests passed, it's time to export the design to an IP block.

- Click on the Export RTL button and go with the default options. You can now close Vivado HLS.

Adding the* hls_multiplier *AXI Lite Slave IP Block to your Vivado design

- In Vivado, go back to Tools -> Settings... and under IP -> Repository add the hls_multiplier/solution1/impl/ip directory. This will allow Vivado to find the IP we just created using Vivado HLS.
- Next, in the Block Design, click on Add IP and search for the hls_multiplier. Add it to your design and Run Connection Automation to automatically have it connected to the AXI Interconnect.
- 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. Example results are shown below. In the following image, the hls_multiplier area is highlighted yellow, while the rtl_multiplier area is highlighted red.


Zynq_CPU_i (Zynq_CPU)
Nets (524)
Leaf Cells (1)
hls_multiplier_0 (Zynq_CPU_hls_multiplier_0_1)
processing_system7_0 (Zynq_CPU_processing_s
ps7_0_axi_periph (Zynq_CPU_ps7_0_axi_periph_
rst_ps7_0_100M (Zynq_CPU_rst_ps7_0_100M_0
rtl_multiplier_0 (Zynq_CPU_rtl_multiplier_0_0)
Cell Properties
rtl_multiplier_0
Name: Zynq_CPU_i/rtl_multiplier_0
Parent: Zynq_CPU_i
X0Y2
X1Y2

Now let's interface the multipliers into Xilinx SDK.

Interfacing with the RTL and HLS blocks in Software

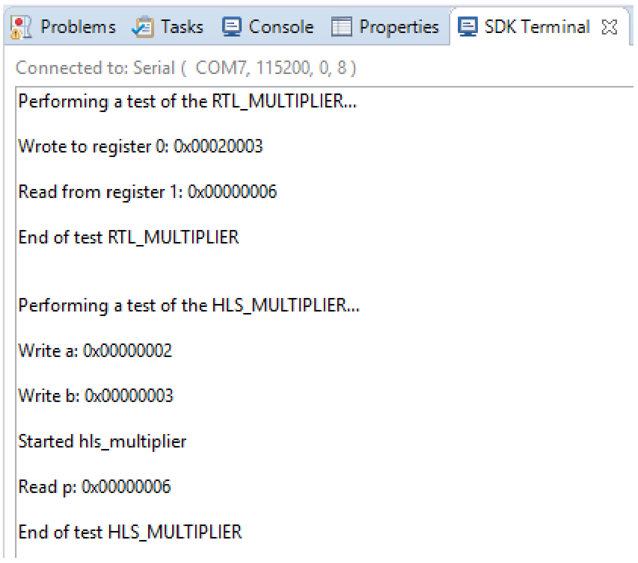
- As what has been practiced in Lab#1, Export Hardware... and Launch SDK.
- Close the previous hello_rtl_multiplier and hello_rtl_multiplier_bsp projects and create a new application called hello_hls_rtl_m starting with the Hello World template.
- In the helloworld.c file paste the following code.

```
/******************************************************************************
 *
 * Copyright (C) 2009 - 2014 Xilinx, Inc.  All rights reserved.
 *
 * Permission is hereby granted, free of charge, to any person obtaining a copy
 * of this software and associated documentation files (the "Software"), to deal
 * in the Software without restriction, including without limitation the rights
 * to use, copy, modify, merge, publish, distribute, sublicense, and/or sell
 * copies of the Software, and to permit persons to whom the Software is
 * furnished to do so, subject to the following conditions:
 *
 * The above copyright notice and this permission notice shall be included in
 * all copies or substantial portions of the Software.
 *
 * Use of the Software is limited solely to applications:
 * (a) running on a Xilinx device, or
 * (b) that interact with a Xilinx device through a bus or interconnect.
 *
 * THE SOFTWARE IS PROVIDED "AS IS", WITHOUT WARRANTY OF ANY KIND, EXPRESS OR
 * IMPLIED, INCLUDING BUT NOT LIMITED TO THE WARRANTIES OF MERCHANTABILITY,
 * FITNESS FOR A PARTICULAR PURPOSE AND NONINFRINGEMENT. IN NO EVENT SHALL
 * XILINX  BE LIABLE FOR ANY CLAIM, DAMAGES OR OTHER LIABILITY,
 * WHETHER IN AN ACTION OF CONTRACT, TORT OR OTHERWISE, ARISING FROM, OUT OF
 * OR IN CONNECTION WITH THE SOFTWARE OR THE USE OR OTHER DEALINGS IN THE
 * SOFTWARE.
 *
 * Except as contained in this notice, the name of the Xilinx shall not be used
 * in advertising or otherwise to promote the sale, use or other dealings in
 * this Software without prior written authorization from Xilinx.
 *
 ******************************************************************************/

/*
 * helloworld.c: simple test application
 *
 * This application configures UART 16550 to baud rate 9600.
 * PS7 UART (Zynq) is not initialized by this application, since
 * bootrom/bsp configures it to baud rate 115200
 *
 * ------------------------------------------------
 * | UART TYPE   BAUD RATE                        |
 * ------------------------------------------------
 *   uartns550   9600
 *   uartlite    Configurable only in HW design
 *   ps7_uart    115200 (configured by bootrom/bsp)
 */
```

```
#include "platform.h"
#include "xbasic_types.h"
#include "xparameters.h" // Contains definitions for all peripherals
#include "xhls_multiplier.h" // Contains hls_multiplier macros and functions

// we will use the Base Address of the RTL_MULTIPLIER
Xuint32 *baseaddr_rtl_multiplier =
                        (Xuint32 *) XPAR_RTL_MULTIPLIER_0_S00_AXI_BASEADDR;

int main() {
            init_platform();

            /////////////////////////////////////////////////////////////////////////////////////////
            // RTL MULTIPLIER TEST
            xil_printf("Performing a test of the RTL_MULTIPLIER... \n\r");

            // Write multiplier inputs to register 0
            *(baseaddr_rtl_multiplier + 0) = 0x00020003;
            xil_printf("Wrote to register 0: 0x%08x \n\r",
                                    *(baseaddr_rtl_multiplier + 0));

            // Read multiplier output from register 1
            xil_printf("Read from register 1: 0x%08x \n\r",
                                    *(baseaddr_rtl_multiplier + 1));

            xil_printf("End of test RTL_MULTIPLIER \n\n\r");

            /////////////////////////////////////////////////////////////////////////////////////////
            // HLS MULTIPLIER TEST
            xil_printf("Performing a test of the HLS_MULTIPLIER... \n\r");

            // Vivado HLS generates
            int status;
            // Create hls_multiplier pointer
            XHls_multiplier do_hls_multiplier;
            XHls_multiplier_Config *do_hls_multiplier_cfg;
            do_hls_multiplier_cfg = XHls_multiplier_LookupConfig(
            XPAR_HLS_MULTIPLIER_0_DEVICE_ID);

            if (!do_hls_multiplier_cfg) {
                        xil_printf(
                                                "Error loading configuration for do_hls_multiplier_cfg \n\r");
            }

            status = XHls_multiplier_CfgInitialize(&do_hls_multiplier,
                                     do_hls_multiplier_cfg);
            if (status != XST_SUCCESS) {
                        xil_printf("Error initializing for do_hls_multiplier \n\r");
            }

            XHls_multiplier_Initialize(&do_hls_multiplier,
            XPAR_HLS_MULTIPLIER_0_DEVICE_ID); // this is optional in this case

            unsigned short int a, b;
            unsigned int p;

            a = 2;
            b = 3;
            p = 0;

            // Write multiplier inputs to register 0
            XHls_multiplier_Set_a(&do_hls_multiplier, a);
            XHls_multiplier_Set_b(&do_hls_multiplier, b);
            xil_printf("Write a: 0x%08x \n\r", a);
            xil_printf("Write b: 0x%08x \n\r", b);

            // Start hls_multiplier
            XHls_multiplier_Start(&do_hls_multiplier);
            xil_printf("Started hls_multiplier \n\r");

            // Wait until it's done (optional here)
            while (!XHls_multiplier_IsDone(&do_hls_multiplier))
                        ;

            // Get hls_multiplier returned value
            p = XHls_multiplier_Get_return(&do_hls_multiplier);

            xil_printf("Read p: 0x%08x \n\r", p);

            xil_printf("End of test HLS_MULTIPLIER \n\n\r");

            cleanup_platform();
            return 0;
}
```

The first part of this code does the same as before, testing the rtl_multiplier in the lab #1. The second part of the code tests the hls_multiplier. Vivado HLS generates not only an HDL implementation of the IP block, but also functions that enable the user to use the IP block correctly. The Code might appear more involved, but try to understand it.

- **Build** and run the code on the Arty Z-7 board. The result from the SDK Terminal is shown below.



```
Connected to: Serial ( COM7, 115200, 0, 8 )
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
```

Follow-up Exercises:

- Change the HLS code from multiplication to division, and repeat the above steps. Please report the resource utilization and make comparisons with the RTL implementation in Lab #1. Summarize any differences you observed.
- Next, using Vivado HLS, implement a 32-bit ALU (Arithmetic Logic Unit), i.e. a block that can perform a number of arithmetic operations (at least multiplication, addition, subtraction, division, etc.). The ALU takes as arguments the number values and also the operation, and returns the result. Please verify each function and make sure it works correctly.

3. Programming the FPGA and Booting Software from the SD card

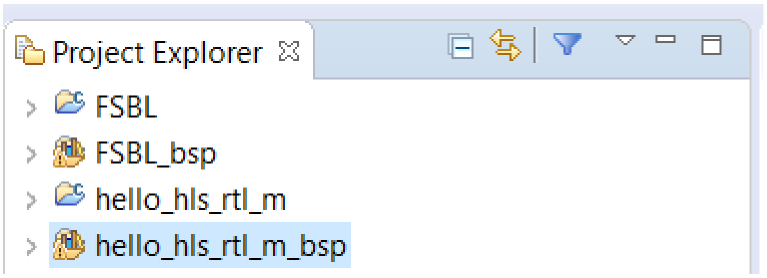
The following process outlines how you can take your developed C-code and boot it from the SD card inserted in the FPGA, which saves the effort of having to run the software from the SDK.

- First, modify the hello_world.c file of your hello_hls_rtl_m project with the following code. 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.

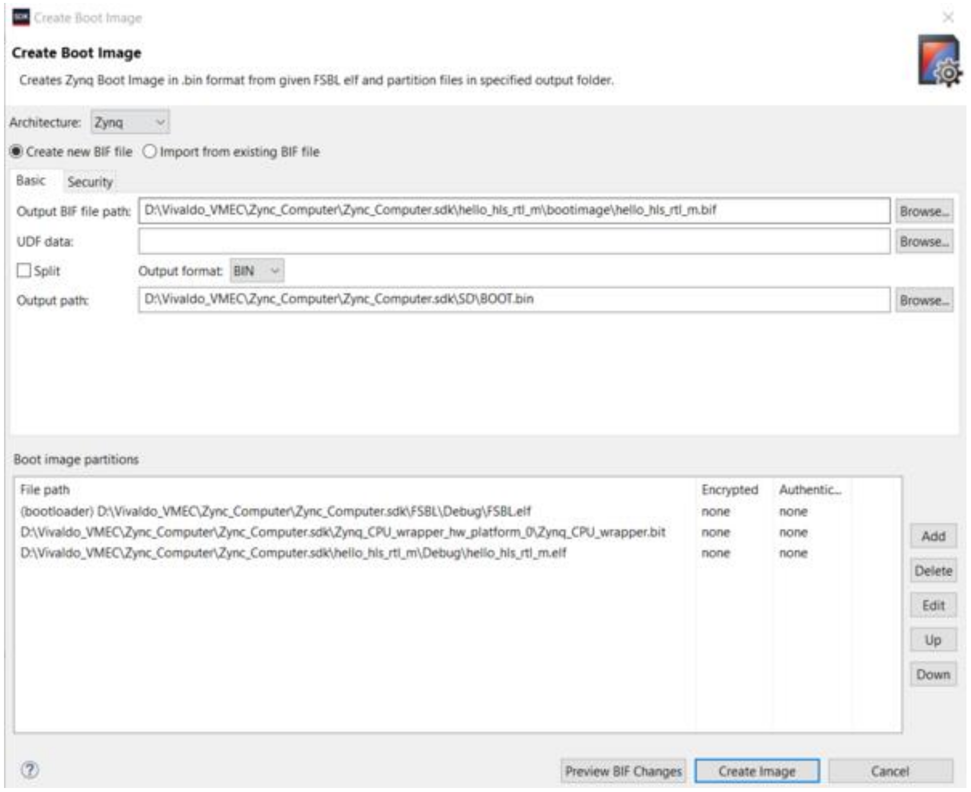
```
#include "platform.h"
#include "xbasic_types.h"
#include "xparameters.h" // Contains definitions for all peripherals
#include "xhls_multiplier.h" // Contains hls_multiplier macros and functions

// we will use the Base Address of the RTL_MULTIPLIER
Xuint32 *baseaddr_rtl_multiplier =
                        (Xuint32 *) XPAR_RTL_MULTIPLIER_0_S00_AXI_BASEADDR;

// global for HLS MULTIPLIER
XHls_multiplier do_hls_multiplier;
XHls_multiplier_Config *do_hls_multiplier_cfg;

// function that prompts user for 2 numbers (ints)
void get_inputs(int *c, int*d) {
            int a, b;
            // get first operand
            xil_printf("Enter operand A: ");
            scanf("%d", &a);
            xil_printf("%d\n\r", a);
            // get second operand
            xil_printf("Enter operand B: ");
            scanf("%d", &b);
            xil_printf("%d\n\r", b);
            // return the two numbers
            *c = a;
            *d = b;
            return;
}
//function that multiplies with RTL multiplier
void multiply_rtl(int a, int b) {
            // concatenate the two integers to a 32-bit value to be passed to RTL multiplier
            int passing_int = (a << 16) + b;
            // Write multiplier inputs to register 0

            *(baseaddr_rtl_multiplier + 0) = passing_int;
            xil_printf("Wrote to register 0: 0x%08x \n\r",
                                *(baseaddr_rtl_multiplier + 0));

            // Read multiplier output from register 1
            xil_printf("Read from register 1: 0x%08x \n\r",
                                *(baseaddr_rtl_multiplier + 1));

            xil_printf("End of test RTL_MULTIPLIER \n\n\r");

}
// initialize the HLS multiplier
void init_HLS_multiplier(){
                        // Vivado HLS generates
                        int status;
                        // Create hls_multiplier pointer
                        do_hls_multiplier_cfg = XHls_multiplier_LookupConfig(
                        XPAR_HLS_MULTIPLIER_0_DEVICE_ID);

                        if (!do_hls_multiplier_cfg) {
                                    xil_printf(
                                                        "Error loading configuration for do_hls_multiplier_cfg
\n\r");
                        }

                        status = XHls_multiplier_CfgInitialize(&do_hls_multiplier,
                                                do_hls_multiplier_cfg);
                        if (status != XST_SUCCESS) {
                                    xil_printf("Error initializing for do_hls_multiplier \n\r");
                        }

                        XHls_multiplier_Initialize(&do_hls_multiplier,
                        XPAR_HLS_MULTIPLIER_0_DEVICE_ID); // this is optional in this case
}
// function that multiplies with HLS multiplier
void multiply_hls(int a, int b) {
            unsigned int p;
            p = 0;

            // Write multiplier inputs to register 0
            XHls_multiplier_Set_a(&do_hls_multiplier, a);
            XHls_multiplier_Set_b(&do_hls_multiplier, b);
            xil_printf("Write a: 0x%08x \n\r", a);
            xil_printf("Write b: 0x%08x \n\r", b);
            // Start hls_multiplier
            XHls_multiplier_Start(&do_hls_multiplier);
            xil_printf("Started hls_multiplier \n\r");

            // Wait until it's done (optional here)
            while (!XHls_multiplier_IsDone(&do_hls_multiplier))
                        ;

            // Get hls_multiplier returned value
            p = XHls_multiplier_Get_return(&do_hls_multiplier);

            xil_printf("Read p: 0x%08x \n\r", p);

            xil_printf("End of test HLS_MULTIPLIER \n\n\r");
}

int main() {
            // setup
            init_platform();
            init_HLS_multiplier();
            int *c, *d;
            // Infinite loop steps: (1) test RTL multiplier with user inputs and (2) test HLS multiplier with user
inputs
            while (1) {
                        ///////////////////////////////////////////////////////////////////////////////////////////
                        // RTL MULTIPLIER TEST
                        xil_printf("Performing a test of the RTL_MULTIPLIER... \n\r");
                        // prompt user for 2 numbers (to be used for RTL multiplication)
                        get_inputs(c, d);
                        // perform RTL multiplication
                        multiply_rtl(*c, *d);

                        ///////////////////////////////////////////////////////////////////////////////////////////
                        // HLS MULTIPLIER TEST
                        xil_printf("Performing a test of the HLS_MULTIPLIER... \n\r");
                        // prompt user for 2 numbers (to be used for HLS multiplication)
                        get_inputs(c, d);
                        // perform HLS multiplication
                        multiply_hls(*c, *d);
            }

            cleanup_platform();
            return 0;
```

```
}
```

- Open the SDK/Vitis. Create a New Project by going to File -> New -> Application Project. 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 how your project set-up should look like.
  - Make sure the FSBL project uses the same Hardware Platform
  - Feel free to select an existing BSP from the workspace or simply let the wizard generate a new one
  - FSBL stands for **F**irst **S**tage **B**oot **L**oader



- Right Click on the hello_hls_rtl_m project folder and select "Create Boot Image", which should open a new window. The Following Window, has the set-up required to successfully create a boot-able SD card Image of your hardware design with the chosen software to run on it. For convenience, change the "Output path" by clicking on Browse and selecting a location where the generated file that is to be uploaded to the SD card is to be saved (I create a folder called SD for this step for convenience).
- Now, you are all set to create the Image to be loaded onto the SD card. The below figure shows a simple set-up that should look like yours.



Create Boot 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

Basic | Security

Output BIF file path: D:\Vivaldo_VMEC\Zync_Computer\Zync_Computer.sdk\hello_his_rtl_m\bootimage\hello_his_rtl_m.bif

UDF data:

Split [ ] Output format: BIN

Output path: D:\Vivaldo_VMEC\Zync_Computer\Zync_Computer.sdk\SD\BOOT.bin

Boot image partitions

File path	Encrypted	Authentic...
(bootloader) D:\Vivaldo_VMEC\Zync_Computer\Zync_Computer.sdk\FSBL\Debug\FSBL.elf	none	none
D:\Vivaldo_VMEC\Zync_Computer\Zync_Computer.sdk\Zynq_CPU_wrapper_hw_platform_0\Zynq_CPU_wrapper.bit	none	none
D:\Vivaldo_VMEC\Zync_Computer\Zync_Computer.sdk\hello_his_rtl_m\Debug\hello_his_rtl_m.elf	none	none

- Click on "Create Image". Go to the "Output Path" directory you selected in the previous step and make sure that a BOOT.bin file was created.
- Insert your FPGA’s SD card in your computer with an SD card reader / adapter. Paste the BOOT.bin file in the SD card.
- Insert the SD card back in the FPGA board. The Arty Z-7 board has several booting options: (i) SD (ii) QSPI and (iii) JTAG. Select booting from SD card, by physically moving the jumper selector over the two pins next to the "SD" indication. This will force the FPGA to boot from the SD card.

Now you are ready to observe that the FPGA indeed boots software from the SD card. Connect the FPGA to your PC and turn it ON. Make sure a connection via the serial port is established either using the SDK Terminal or PUTTY (Go to 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 (which is observed either on the SDK Terminal in the Xilinx SDK/Vitis or through PUTTY), 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 SD card! Note: The same BOOT.bin file can be used to boot from the SPI Flash memory.

4. Programming Python on Arty Z7-20

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**, **Py**thon Productivity for Zy**nq**, 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 application programming interface (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 uses 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 here https://pynq.readthedocs.io/en/v2.5.1/#

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 gig if possible).

- Here is a list of PYNQ images that correspond to different boards, please choose the PYNQ-Z1 image. http://www.pynq.io/board.html
- To burn the ISO image to the SD card, you can use win32 Disk Imager.

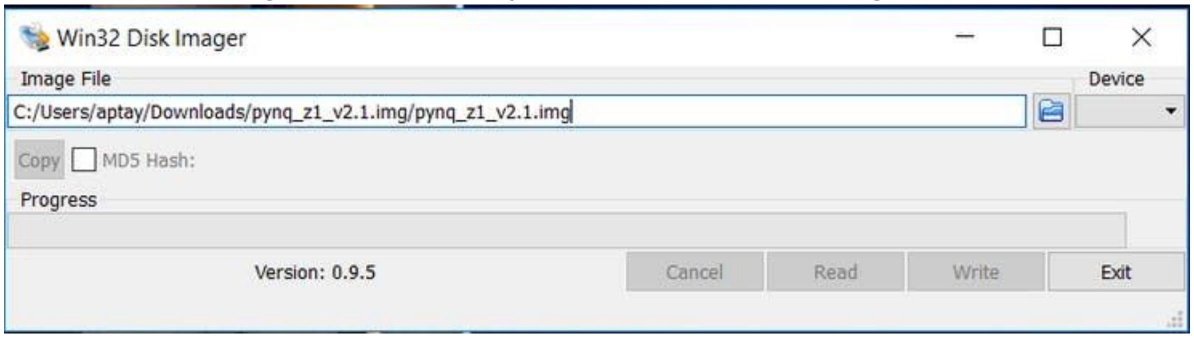
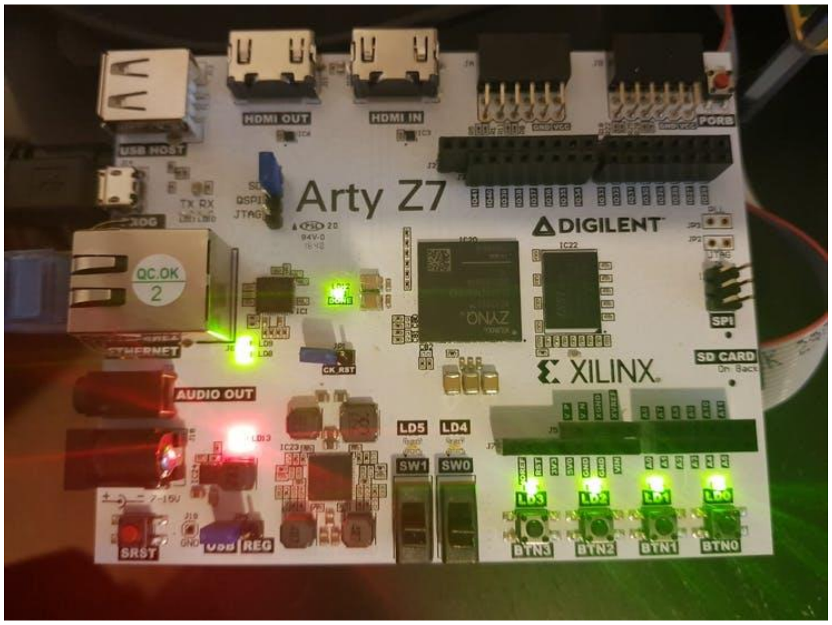
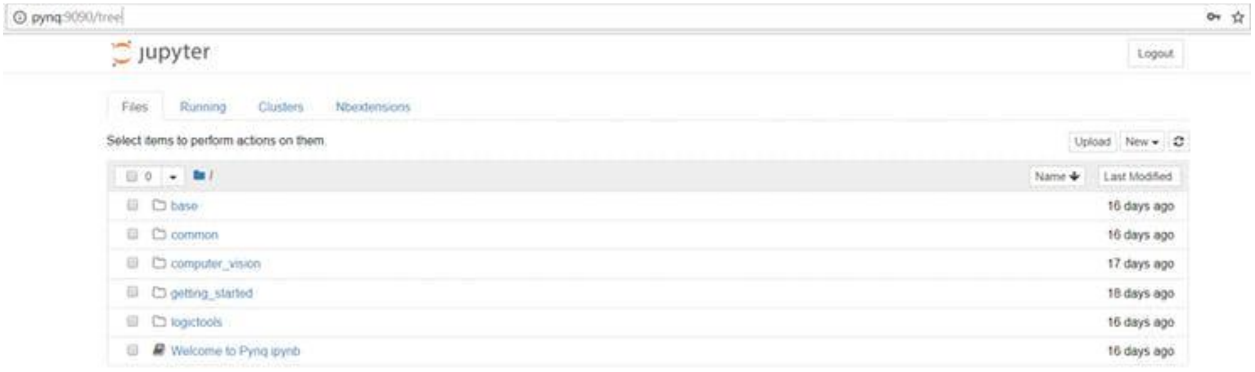


Image File: C:/Users/aptay/Downloads/pynq_z1_v2.1.img/pynq_z1_v2.1.img

- Once the image is written to the SD Card, it is time to insert the card into the Arty Z7-20 ensure the configuration jumper is set to boot from SD and connect an Ethernet cable (to computer) before powering it on.
- Power on the board, and after a few seconds you will see the Done LED illuminate, 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.



- 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.


pynq:9090/tree

jupyter — Logout

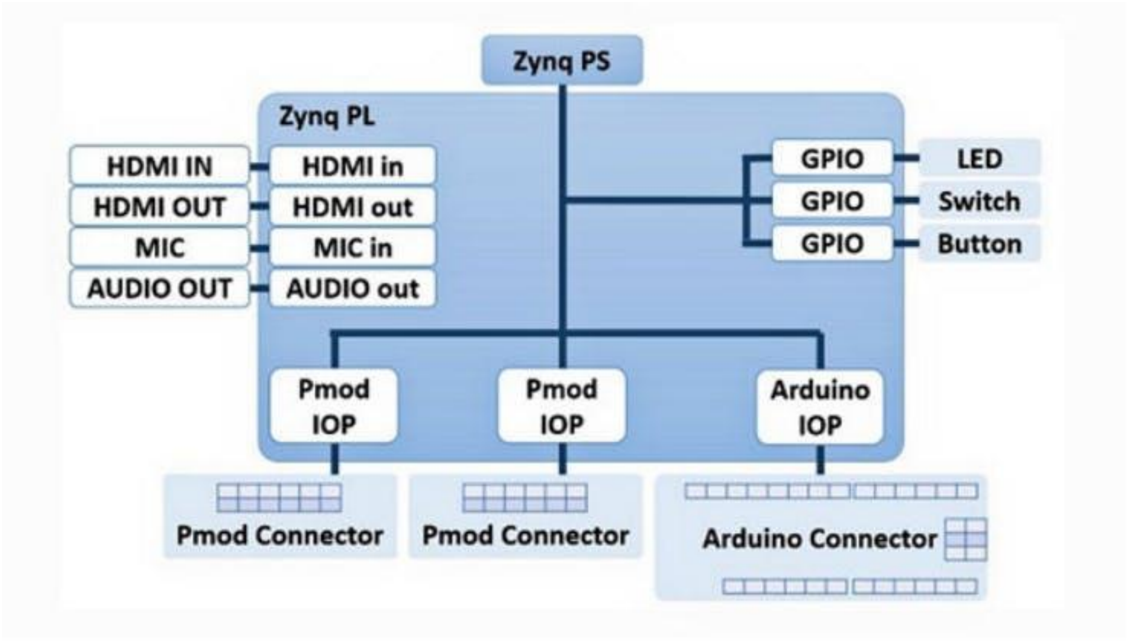
Files | Running | Clusters | Nbextensions

Select items to perform actions on them.

```
0  /                                   Name ↓   Last Modified
   base                                          16 days ago
   common                                        16 days ago
   computer_vision                               17 days ago
   getting_started                               18 days ago
   logictools                                    16 days ago
   Welcome to Pynq.ipynb                         16 days ago
```

- 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 on how to get started using the development environment.
- 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 which supports all the inputs and outputs (IO) on Arty Z7-20.



- 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.

References:

1. Arty Z7 Reference Manual https://digilent.com/reference/programmable-logic/arty-z7/reference-manual
2. Getting Started with the Vivado IP Integrator https://digilent.com/reference/vivado/getting-started-with-ipi/2018.2
3. https://pynq.readthedocs.io/en/v2.5.1/
4. https://xilinx-wiki.atlassian.net/wiki/spaces/A/overview
5. https://www.dataquest.io/blog/jupyter-notebook-tutorial/
6. https://www.hackster.io/90432/programming-python-on-zynq-fpga-ec4712

Acknowledgement:

- Sergiu Masanu, Mircea Stan (U of Virginia)
- Andreas Gerstlauer (U of Austin)
- Digilent
- Xilinx PYNQ project
- Xilinx University Program

Deliverables:

- Group Deliverables (Compile everything as a single pdf report besides the code):
  - Based on **part 2**, please submit
    - The resource utilization of the HLS multiplier (screenshot)
    - The resource utilization of the HLS divider (screenshot) and the source code, together with the SDK terminal printout
    - ALU source code and resource utilization, SDK terminal printout (screenshot)
  - Based on **part 3**, please submit the following
    - The screenshot from the SDK terminal or PUTTY terminal
  - Based on **part 4**, please submit the following
    - The screenshot of the Jupyter notebook
  - Answer the following questions (based on your understanding, feel free to use the internet):
    - What was the resource utilization of the rtl implementation vs. HLS implementation? If they are different, can you guess why there are differences?
    - Imagine that you have to create an IP block implementing 10 different sorting algorithms. Would you rather use a hardware description language or High-level Synthesis? Why?
    - 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?
- Individual Deliverables
  - Complete the peer-evaluation form appended in the end of this document

Grading Policy

Factor	**Percentage**
Part 2	40%
Part 3	20%
Part 4	20%
Questions	20%

ECE4810J System-on-Chip (SoC) Design

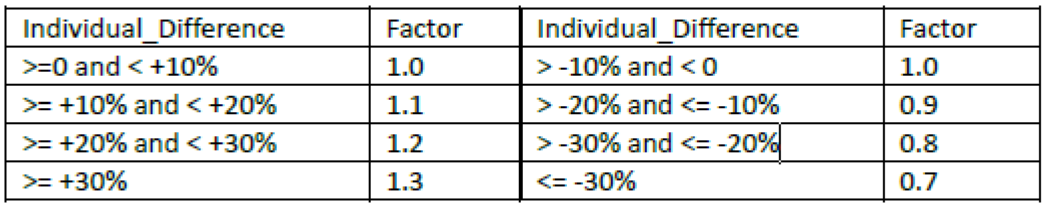
Fall 2021

Lab #2 Peer Evaluation Form

Each team member is required to provide a peer evaluation for the team effort of the lab. The score of the peer evaluation should be integers ranging between 0 to 5, inclusively, with 5 indicating the biggest contribution. A score should be given to each team member including yourself according the team member's contribution based on your observation. A brief description of specific contribution of each team member should also be provided. Note this form needs to be finished without discussing with your teammate, if all forms are found the same, then you have to justify how each member makes exactly the same contribution (through demonstration to the instructor).

Name	Level of contributions (0 – 5)	Description of contributions
(yourself)		
Team member 1		
Team member 2		
Team member 3		

Your lab grade is calculated based on the following:
Individual_Average = (sum of all team member's marks) / (size of the student team)
Group_Average = sum of Individual_Average of all team members / size of student team
Individual_Difference = Individual_Average / Group_Average – 1.0
Using the calculated Individual_Difference, we find a factor from the following lookup table.



Individual_Difference	Factor	Individual_Difference	Factor
>=0 and < +10%	1.0	> -10% and < 0	1.0
>= +10% and < +20%	1.1	> -20% and <= -10%	0.9
>= +20% and < +30%	1.2	> -30% and <= -20%	0.8
>= +30%	1.3	<= -30%	0.7

Your final lab grade is calculated by :
Final grade of lab = points for team effort * factor

Note that the final grade of the lab will not exceed the maximum points assigned to the project. If the peer-evaluation form is not submitted, then your individual_difference will be based on the available data (without yours) only, and your final score will be:
Final grade of lab = points for team effort * factor * 95%