Introduction:

In this lab, you will be learning how to create a custom peripheral in the programmable logic on the ZedBoard and how to communicate with that peripheral from software running on the ARM processor.

UPDATE: You will be working in groups of 2 on this project. Unless you are an EDGE student or have special permission from the instructor, you must work in a group of 2. Note that those in any overflow sections are *not* EDGE students, and must work in a group of 2. Please only submit once per group.

Part 1 – Base project and peripheral tutorial

1) Watch the lecture video and the tutorial video linked from the lab website to learn how to create a base Zynq project, and how to create a peripheral.

   IMPORTANT: Make sure to select the ZedBoard when creating the project.

   In this example, you will be creating a multiplier peripheral. Vivado provides you with a basic peripheral template that has 4 32-bit memory-mapped registers. We are going to add a multiplier to that takes two 16-inputs and generates a 32-bit output. The two 16-bit puts will come from register 0, where one input gets bits 31 downto 16 and the other input gets bits 15 downto 0. Alternatively, we could have designed the lab to use two separate registers, but the software expects the both inputs to be provided in register 0. The output of the multiplier will essentially replace register 1, so when software tries to register from register 1, it is actually accessing your multiplier output. Similarly, when software writes to register 0, it is writing two 16-bit inputs to the 32-bit register.

   The provided peripheral template has other registers that we are going to ignore for this lab.

2) Create a multiplier entity that uses a generic width, where the inputs are WIDTH bits, and the output is 2*WIDTH bits. The multiplier can be purely combinational, or you can add a clock and reset if you like.
3) Instantiate the multiplier in the Vivado-provided peripheral template as shown in lecture and the tutorial video. Set the generic WIDTH to 16 in the generic map. In the port map, connect the inputs to slv_reg0(31 downto 16) and slv_reg0(15 downto 0). Connect the output to a new 32-bit signal (e.g., multiplier_out). Make sure to declare the signal. If you used a clock and reset on your multiplier, make sure to connect them in the port map.
4) Update the process that assigns reg_data_out. Replace slv_reg1 with your multiplier output signal. Make sure to change it in the process and in the sensitivity list.
5) Merge your changes to the files (see videos)
6) Re-package the IP for the peripheral and return to the base project.
7) In the sources tab, under design sources, right click the block diagram (.bd) file and select “Create HDL Wrapper.”
8) Once in the base project, run the entire synthesis flow through implementation.

9) Find the generated bitfile in:

   *project_path**project_name*.runs\impl_1

   Note that project_path is the path of your Vivado project. Project_name is the name of that project. The bitfile will be called *block_diagram_name*_wrapper.bit, which is likely design_1_wrapper.bit if you accepted the defaults values when creating your block diagram.

   Rename the bitfile to part1.bit. Place it in the same directory as the provided C++ code for part1.

10) Log onto ece-b312-recon2.ad.ufl.edu using an ssh client (e.g. putty) using your Gatorlink account information.
11) From the provided shell, run the following command: /usr/bin/zedboard/setup/setup_account.sh
12) **Log off the server and then log back on.**
13) Use an scp program (e.g. WinSCP) to transfer the provided code with the bitfile into your account on ece-b312-recon2.ad.ufl.edu.
14) From ece-b312-recon2.ad.ufl.edu, compile the C++ code using “make”. Make sure there are no errors or warnings.
15) Run your code on a ZedBoard using:

    zed_schedule.py ./zed_app part1.bit

    zed_schedule is a scheduling script that connects to the board server and finds an available board. If a board is available, the script copies your current directory to the selected board and then runs the specified executable (e.g., ./zed_app). This application takes the name of the bitfile as a command line input, so make sure to include part1.bit.
16) Verify there are no errors. The output should look similar to this:

```
Starting job "./zed_app part1.bit" on board 192.168.1.102:
Application completed successfully!
```

17) Take a screenshot of the software output.
18) Find and save the IP directory for your peripheral. This directory should have a component.xml file, in addition to various other directories (e.g., src, hdl, drivers, example_designs, etc.). Its location depends on the path of the IP repository that was specified when you created the peripheral.
19) Submission requirements: IP directory, screenshot of software output, and part1.bit. See submission instructions at the end of the document for details.

Part 2 – Create your own custom peripheral

In part 2, you will be using what you learned in part 1 to create a new peripheral.

1) Start a new project for the ZedBoard, following the same steps you did in part 1.
2) Create a new AXI peripheral called part2. On the Add Interfaces dialog box, **select 8 registers instead of the 4 registers you used in part 1.**
3) Download the provided entity peripheral_test.vhd. Instantiate this entity within the AXI peripheral code (part2_v1_0_S00_AXI.vhd) in a similar way as the multiplier in part 1. Registers 0-3 should connect to inputs 0-3. Registers 4-7 should connect to outputs 0-3. For the generic width, use a value of 32 bits to match the width of the registers.
4) Modify the AXI peripheral code to prevent registers 4-7 from having multiple sources by commenting out every assignment to those registers. This modification is required because in the generated code, the AXI bus writes to all the registers. We only want peripheral_test to write to registers 4-7, so we have to remove the generated code that writes to these registers. After these modifications, you are actually removing registers 4-7 and are just reusing those signals to act as wires. If you want, you could rename the signals to be more accurate, but this is not required.
5) Fill in the default architecture for peripheral_test.vhd with the following functionality. You can use either **signed** or **unsigned** for the arithmetic; the output is identical for these operations:

   out0 = in0 * in1 (ignore overflow by only using the lower width bits)

   out1 = in0 + in1

   out2 = in2 - in3

   out3 = in2 xor in3

6) Create a testbench peripheral_test_tb.vhd that demonstrates the correctness of peripheral_test. You do **not** need a simulation screenshot. It is up to you do determine the thoroughness of the testbench.
7) Synthesize the IP to make sure there are no errors.
8) Package your peripheral IP and connect it the Zynq in the block diagram (see tutorials in part 1).
9) Generate a bitfile.
10) Find the bitfile and rename it to part2.bit. Copy it into the directory with the provided C++ code for part2.
11) Transfer the directory with the C++ code and the bitfile to ece-b312-recon2.ad.ufl.edu.
12) From ece-b312-recon2.ad.ufl.edu, compile the C++ code using “make”.
13) Run your code on a ZedBoard using:

```
zed_schedule.py ./zed_app part2.bit
```

At this point, there should be no errors for out0 and out1. If there are errors, your peripheral is not working. IMPORTANT: out2 and out3 should be ***incorrect*** at this step.

14) Open main.cpp and find the code that transfers in0 and in1 to the ZedBoard. Extend the code to write in2 and in3 to the board. Make sure to use the correct AXI address (see the enum at the top of the file).
15) Find the code that reads results from out0 and out1. Add the code that replicates this process for out2 and out3.
16) Recompile using “make”.
17) Rerun the code on the ZedBoard. If there are still errors, there is either a problem with your peripheral or with the C++ code you added. Debug until there no errors.

18) Take a screenshot of the working execution (or of the errors that you received if you don't have it working). e.g.:

```
Out0 Errors: 0
Out1 Errors: 0
Out2 Errors: 0
Out3 Errors: 0
Total Errors: 0
Application completed successfully!
```

19) Find and save the IP directory for part2_peripheral.

20) Submission requirements: IP directory, screenshot of software output, C++ code, part2.bit, peripheral_test.vhd, peripheral_test_tb.vhd, and part2_v1_0_S00_AXI.vhd. See submission instructions at the end of the document for details.

SUBMISSION INSTRUCTIONS (One submission per group)

Create a directory called lab2. Inside this directory, create a part1 and part 2 directory. Include a brief readme.txt file in the lab2 directory that specifies the members of the group, any other relevant information that the grader might need (e.g., if one of the parts doesn't work, crashes, doesn't compile, etc.). The submitted directory structure should look like this:

```
lab2/
    readme.txt
    part1/
        screenshot
        part1_1.0/          // IP core from repository
        part1.bit           // DON'T FORGET
    part2/
        screenshot
        part2_1.0/          // IP core from repository
        sw/
            main.cpp
            Board.h
            Board.cpp
            Makefile
            part2.bit           // DON'T FORGET
        hdl/
            peripheral_test.vhd
            peripheral_test_tb.vhd
            part2_v1_0_S00_AXI.vhd
```

Zip the entire lab2 directory and submit the lab2.zip file.

Extra Credit:
In peripheral_test_tb.vhd, create a testbench that exhaustively tests all possible input combinations for an instance of peripheral_test with a width of 4 bits. Because there are 4 inputs, a width of 4 corresponds to 2^16 = 64k possible combinations. NOTE: Larger widths will take an extremely long time, so do not try this.

Common Problems:

- Do not create a bitfile for the base project tutorial. Only do it after adding a peripheral. Running a bitfile on the boards without the peripheral will lock them up and require a reboot.
- If you make a mistake in the IP core, you can't directly edit the code from within your project. This occurs because Vivado copies a read-only version of the IP core into your project. If you need to edit it, right click the core in the block diagram and select "Edit in IP Packager". This will open a separate project for the IP core. From the IP project, modify the code, re-package the IP after any changes, and then refresh the IP within your original project. This process will get very annoying, so make sure you have thoroughly tested your IP core code in simulation before packing it to use with the Zynq.

  There actually is a way to edit the read-only copy of the core, but I don't recommend that yet. I'll explain how for later labs.
- Unless you make modifications to the code that are not explained in the instructions, you will receive a few synthesis warnings for your peripheral. E.g.:

```
[Synth 8-3331] design part2_v1_0 has unconnected port
s00_axi_awprot[1]
[Synth 8-3332] Sequential element
(\part2_v1_0_S00_AXI_inst/axi_awaddr_reg[0] ) is unused and
will be removed from module part2_v1_0.
```

  Although for future labs I would encourage you to get rid of all wanings, these warnings will not cause any problems for this lab.