Introduction:

In this lab, you will be using the Perfect Square calculator from an earlier lab as a custom peripheral to the ARM processor on the ZedBoard. To enable communication with this peripheral, you will be implementing a custom memory map that associates different FPGA resources with a specific address. Although the AXI interface could provide this same functionality, in this lab we use a specialized abstract interface (i.e., a virtual board) with a memory-map interface that is built on top of the AXI interface. This virtual board enables us to potentially use the exact same code on different boards, instead of porting the memory-map code for different board interfaces.

You will again 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 section are *not* EDGE students, and must work in a group of 2. Please only submit once per group.

EDGE INSTRUCTIONS (All non-EDGE students must do the entire lab):

EDGE students must complete part 1, and can do part 2 for extra credit. If you decide to not do part 2, you will need to email me for the solution. I would recommend trying part 2 first and then emailing me for the solution if you get stuck. After I send you the solution there is no longer an option for extra credit.

Part 1 – VHDL

1) Download the provided code from the lab website. This code contains a solution from lab 1 in case you didn't get that lab fully working. However, I would encourage you to use your own code after testing everything with the provided code. Instead of providing the source for my solution, I provided a pre-synthesized version (in is_perfect_square.edn) and a simulation model of that synthesized solution (is_perfect_square_sim.vhd). In case Vivado messes up the IP core settings, is_perfect_square.edn needs to be a design source (and only a design source), and is_perfect_square_sim.vhd must be a simulation source (and only a simulation source).
2) For this lab, I have provided an existing IP core that you will extend. This core is located in the is_perfect_square_1_0/ directory. To instantiate this core in your Vivado project, move the folder into your IP repository that you used in lab2, or alternatively create a new repository. To use multiple repositories, right click in the block diagram (over several other places), and select "IP Settings." This will allow you to set the path to multiple repositories. As long as you put the is_perfect_square_1_0/ folder into one of these paths, you should be able to instantiate it with the "Add IP" option.
3) Look over the provided code in the is_perfect_square_1_0/src directory.
    - The top-level entity in *this directory* is called wrapper. **You do not need to change your top-level entity in Vivado. It should be the auto-generated AXI entity.** The wrapper entity doesn't do much yet, but will be expanded upon in later labs to provide access to various resources (e.g., external memories). This is the entity that you would instantiate from

within the auto-generated AXI peripheral code. However, I have already provided that code for you in the is_perfect_square_1_0/hdl/ directory, which you do not need to change.

- The wrapper entity instantiates the user_app entity, which is the abstraction you will be using for all future labs. Currently, user_app provides a clock, a reset, and a simplified memory-map interface that is easier to work with (and can be made portable across different boards) compared to the AXI interface.
- config_pkg.vhd contains constants and types that are used for configuring the virtual board interface.
- user_pkg.vhd provides constants that are used throughout the application. Think of this file as a custom header file in C code. You should use the constants that I have already provided when creating your memory-map implementation. For future labs, you should use this file for your own custom constants and types.
- user_app_tb.vhd is a provided testbench that demonstrates how to send inputs and read outputs to/from the memory map. You do not have to change the testbench for this lab. However, I would recommend using this testbench as a template for all future labs. Note that the testbench outputs a score, which gives you an idea of your grade. The graders will use a similar testbench, so I can't promise your grade will be identical, but this at least gives a good idea of what to expect.
- memory_map.vhd implements the memory-map entity that enables application-specific communication with FPGA resources
- The is_perfect_square/ directory contains the solution for the Perfect Square calculator. The top-level entity is called is_perfect_square.

4) Modify the memory map entity to enable communication with the two application inputs: *go* and *n*. Do the same for the two outputs: *done* and *output*. This will be explained in more detail in class.

   The memory-map interface works as follows. When *wr_en* is asserted (active high), this means that the processor is currently providing valid data on *wr_data*. This data will only be valid for one cycle, so in the same cycle you must check *wr_addr* and then store *wr_data* into the corresponding resource, which for this assignment is just a register. For example, you will have separate registers for inputs *go* and *n*.

   When *rd_en* is asserted (active high), this means that the processor is requesting data from the address on *rd_addr*. The interface requires the data to be provided on the *rd_data* port **one cycle after** *rd_en* is asserted. If you provide the data at any other time, your code will not work.

5) Simulate user_app_tb to test the correctness of your design. Note that even if the testbench reports no errors, there could still be bugs in your code. In fact, I purposely created the testbench in such a way that it is unlikely to expose certain

types of bugs. I leave it to you as an exercise to make the testbench better, but you do not have to submit the modified testbench.

6) Repackage the accelerator IP with your modified files, add it as a peripheral for the Zynq, and generate the corresponding bitfile (see lab 2 for the exact steps).
7) Rename the bitfile to lab3.bit and upload it to your project directory on ece-b312-recon2.ad.ufl.edu.

Part 2 – C++

In part 2, you will be creating a software program to communicate with the peripheral. Use the code in the sw/ directory of the provided code as a basic template, and refer to lab 2 for how to use the Board class' read and write functions. Make sure to define constants for the memory-map addresses that match those in user_pkg.vhd.

The C++ code should determine if integers 0 through 49 are perfect squares using the FPGA circuit and output them to the screen. To verify correctness, you should also calculate each output using software using the same algorithm that was specified in lab 1. The provided software template includes a is_perfect_square function that you can use. The output of the program should look **exactly** like this:

```
0: HW = 1, SW = 1
1: HW = 1, SW = 1
2: HW = 0, SW = 0
3: HW = 0, SW = 0
4: HW = 1, SW = 1
5: HW = 0, SW = 0
6: HW = 0, SW = 0
7: HW = 0, SW = 0
8: HW = 0, SW = 0
9: HW = 1, SW = 1
10: HW = 0, SW = 0
11: HW = 0, SW = 0
12: HW = 0, SW = 0
13: HW = 0, SW = 0
14: HW = 0, SW = 0
15: HW = 0, SW = 0
16: HW = 1, SW = 1
17: HW = 0, SW = 0
18: HW = 0, SW = 0
19: HW = 0, SW = 0
20: HW = 0, SW = 0
21: HW = 0, SW = 0
22: HW = 0, SW = 0
23: HW = 0, SW = 0
24: HW = 0, SW = 0
25: HW = 1, SW = 1
26: HW = 0, SW = 0
27: HW = 0, SW = 0
28: HW = 0, SW = 0
29: HW = 0, SW = 0
30: HW = 0, SW = 0
```

```
31: HW = 0, SW = 0
32: HW = 0, SW = 0
33: HW = 0, SW = 0
34: HW = 0, SW = 0
35: HW = 0, SW = 0
36: HW = 1, SW = 1
37: HW = 0, SW = 0
38: HW = 0, SW = 0
39: HW = 0, SW = 0
40: HW = 0, SW = 0
41: HW = 0, SW = 0
42: HW = 0, SW = 0
43: HW = 0, SW = 0
44: HW = 0, SW = 0
45: HW = 0, SW = 0
46: HW = 0, SW = 0
47: HW = 0, SW = 0
48: HW = 0, SW = 0
49: HW = 1, SW = 1
```

IMPORTANT: Make sure to test your output with the provided grade.py script. Do the following when you run your application:

zed_schedule.py ./zed_app lab3.bit > test.txt

python3 grade.py test.txt

Note that the reported score only applies to part 2, so a perfect score is 50%. Again, the grader will use a similar script to grade the projects, so this will give you an idea of your grade.

If you think your code works, but are getting errors reported, check the contents of the test.txt file. It is possible that some misc issue occurred. For example, if the you get the message about the boards rebooting, you just need to run your code again.

SUBMISSION INSTRUCTIONS (One submission per group)

Make sure the names of all group members are at the top of every file.

Create a directory with your UFID. Use the following structure:

```
ufid/
    readme.txt                  // Group members, anything that the grader needs to
                                // be aware of
    lab3.bit
    is_perfect_square_1_0/      // IP core from repository with this exact name
                                // Make sure all VHDL is included
    sw/
        All C++ code            // All .cpp, .h files, including the Board files
        Makefile                // Either the provided one, or a version with any of
                                // your modifications
```

Zip the entire directory and submit the zip file.

EXTRA CREDIT OPPORTUNITIES

Instead of having the software perform two writes for go=1 and go=0, there is a trick you can do to create assert and de-assert go with only one write from software. If you can figure this out, you will receive extra credit. Make sure to specify in your readme.txt file if you do this.