ECE437: Sensors and Instrumentation

Lab 2: USB communication between FGPA and PC

Introduction

As part of this course, you will develop firmware and software programs that enable you to communicate with various sensors using OpalKelly data acquisition card. The data acquired from the sensors will be sent from the FPGA to the PC for further data analysis and display. This is a common requirement for many industrial- and research-based sensory applications: data from many different types of sensors has to be aggregated at the PC, where a high-level decision is executed. Hence, the skills that you will acquire in this lab are required for many industrial- and research-based jobs. Don't forget to put this experience on your resume next time you apply for a job!

OpalKelly makes the data transfer process easy by providing Verilog and Python code, which enables seamless data transfer between the FPGA and PC. In this lab, we will get acquainted with low bandwidth data transfer. Specifically, you will use OkWireIn and OkWireOut syntax to send/receive data at about 1KHz bandwidth. The framework that will be developed in this class will be used for future labs, so make sure you are comfortable with the code in this lab. Later in the semester, we will focus on high bandwidth data transfer (such as ~1GByte bandwidth), which is important when large amounts of data are transferred to and from the PC.

Opal Kelly website provides many useful examples and tutorials. You should look over some of these examples and pay attention how other people have implemented finite state machines. Additional information for XEM7310 can be retrieved here. Tutorials and examples for the XEM7310 board can be located here. The online documentation for OpalKelly's Front Panel software has lot of good documentation and examples.

We will also be using Python in this lab and throughout the semester. We will be providing examples codes in Python through the semester. There are many online tutorials for Python and you are encouraged to look up some of them. A useful website containing various examples and tutorials can be located here.

Relevant Documents for this Lab

Required reading material for this lab:

1. Information for the XEM7310-A75 board can be located here.
2. OpalKelly Front Panel online documentation .
3. Verilog example code on the course website: lab2_example.v
4. Python example code on the course website: lab2_example_python.py

Additional reference material:

1. Python tutorials can be located here.

The goals of this lab are:

1. Develop firmware (Verilog) and software (Python) for low bandwidth bidirectional communication between PC and FPGA.
2. Get familiar with basic Python syntax.

Prelab Questions (5 points per question):

1. Look over the FrontPanel online documentation. How many WireIn and WireOut registers do you have at your disposal?
2. What is the bit depth for each one of these registers?
3. What is the maximum data bandwidth when using Block-Throttled Pipes?
4. What is the memory address range for WireIn and WireOut?

Bidirectional Communication

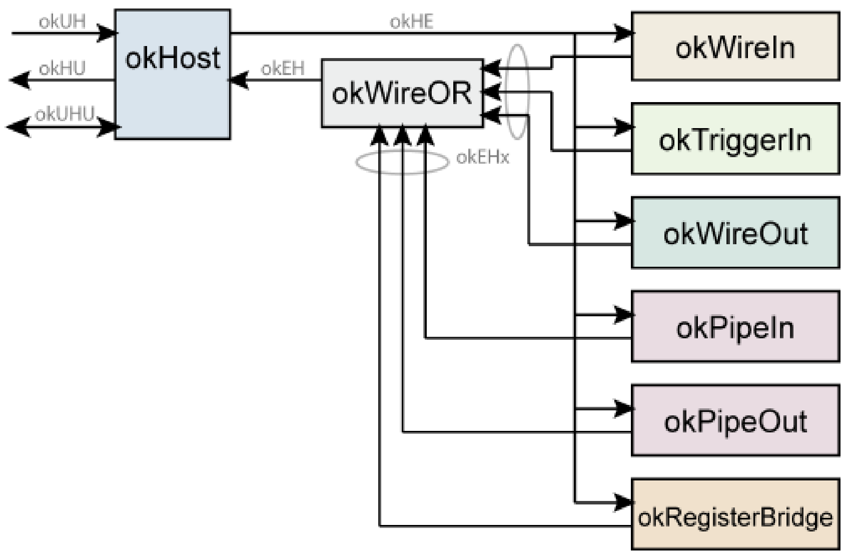
We will be developing code that will enable sending and receiving data between the host (PC) and the FPGA. This will require developing Verilog code for the FPGA and Python code that will run on the PC. Both Verilog and Python code must work together to have successful data transfer. For example, if you write a code in Verilog whose task is to send one byte of data to the PC, then the Python code on the PC side must be prepared to receive this data. OpalKelly provides very intuitive interface which makes this process easy to implement and understand.

Verilog

From the Verilog standpoint, Opal Kelly gives us some plug-and-play modules to work with. Some of these modules are presented in the figure bellow. The naming of these modules is FPGA-centric. For example, okWireIn is a module that enables inputting 32-bit data in the FPGA from the PC. Conversely, okWireOut send 32-bit data from the FPGA to the PC.

Each one of these modules send or receive data from a given memory address in the FPGA and PC. For example, okWireIn module enables a 32-bit register, located in any one of the memory addresses between 0x00 and 0x1F, to receive data from the PC. This means that your Verilog and Python code have to be synchronized. For example, if you like to receive data in a register located in memory 0x10 in the FPGA, your Python code will have to send data at this specific memory address (0x10). It is the programmer responsibility to make sure data gets received and delivered to the correct memory address. This will become more obvious with the next example.

Note that if you are using okWireIn, you can only use registers located in memory space between 0x00 and 0x1F. This means that you only have total of 32 registers to receive data in the FPGA. Each one of these registers is 32-bit wide. The OkWireOut module uses registers located in memory address 0x20 to 0x3F. The correct memory address for each one of the modules outlined in the figure below can be located in the online documentation on OpalKelly's website.



***G** is Gateway, **H** is Host, **E** is Endpoint*

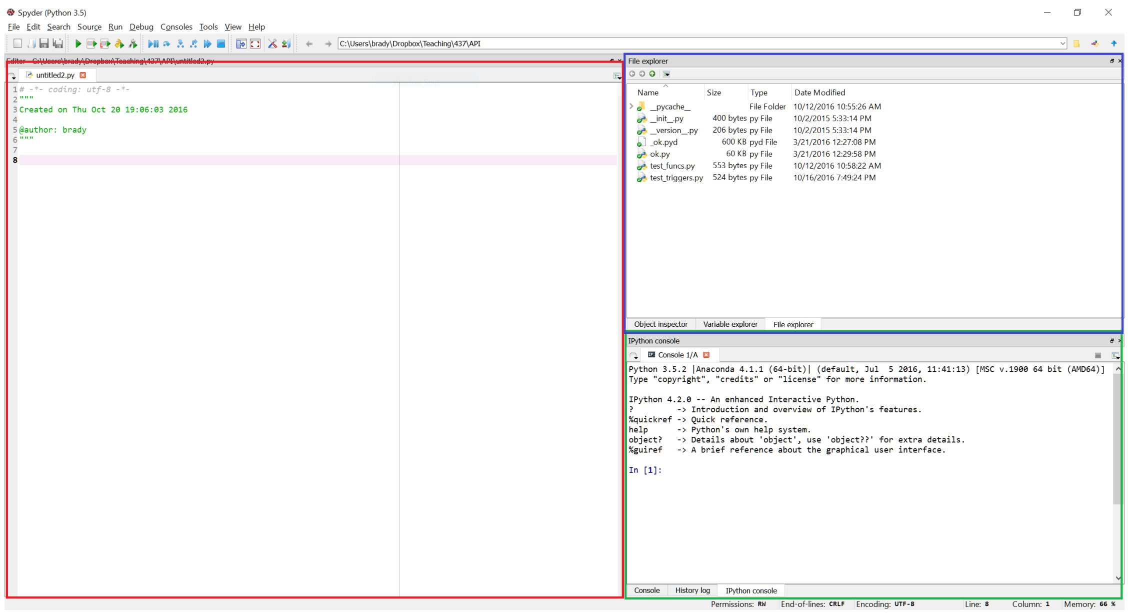
The notation used above is okXY, where we send some Opal Kelly module from X to Y. We'll represent everything "off" the FPGA as the **G**ateway to the computer. Opal Kelly has a virtual **H**ost (okHost) that sits in between the computer and our memory. After our **H**ost, we can directly write to various WireIn, TriggerIn, and PipeIn **E**ndpoints. These **E**ndpoints are equivalent to memory address in your virtual memory, and this is effectively a write operation to the virtual memory.

For a read operation, we need to get data from an **E**ndpoint back to the **H**ost and ultimately to the **G**ateway. However, rather than read out *N* parallel lines at once, we'll use a serializer to combine multiple short parallel words into one long serial word. This is the function of the okWireOR module. The output of the okWireOR module goes back to the okHost, which then returns to the gateway.

Now while numerous datatypes exist between the **G**ateway and the **H**ost, we'll only start with the simplest one for this lab. **Wires** are the simplest datatype and represent signal states or levels. They can be read asynchronously and can be thought of as a status indicator. The OpalKelly board that we are using in this class, XEM7310-A75, supports 32 **WireIn** and 32 **WireOut**.

Python

Similar to the Verilog side, the majority of the hard work is done by Opal Kelly already. A Python module is given by Opal Kelly to handle most of the device communication, as well as programming the FPGA directly. We'll be using a popular IDE for Python called **Spyder**, which will simplify much of the debugging process. Run Spyder from either the Desktop or from the Start Menu, under Spyder. You should be greeted with the screen below:



```python
# -*- coding: utf-8 -*-
"""
Created on Thu Oct 20 19:06:03 2016

@author: brady
"""

```

Name	Size	Type	Date Modified
__pycache__		File Folder	10/12/2016 10:55:26 AM
__init__.py	400 bytes	py File	10/2/2015 5:33:14 PM
__version__.py	206 bytes	py File	10/2/2015 5:33:14 PM
_ok.pyd	600 KB	pyd File	3/21/2016 12:27:08 PM
ok.py	60 KB	py File	3/21/2016 12:29:58 PM
test_funcs.py	553 bytes	py File	10/12/2016 10:58:22 AM
test_triggers.py	524 bytes	py File	10/16/2016 7:49:24 PM

```
Python 3.5.2 |Anaconda 4.1.1 (64-bit)| (default, Jul  5 2016, 11:41:13) [MSC v.1900 64 bit (AMD64)]
Type "copyright", "credits" or "license" for more information.

IPython 4.2.0 -- An enhanced Interactive Python.
?         -> Introduction and overview of IPython's features.
%quickref -> Quick reference.
help      -> Python's own help system.
object?   -> Details about 'object', use 'object??' for extra details.
%guiref   -> A brief reference about the graphical user interface.

In [1]:
```

Permissions: RW End-of-lines: CRLF Encoding: UTF-8 Line: 8 Column: 1 Memory: 66 %

If you've ever used MATLAB, Octave, or similar scientific programming IDEs, the layout should be quite familiar. The section highlighted in red is our code editor, the section in green is a console, and the section in blue keeps track of our variables and files. If you have never used Python or Spyder before, you can download the python example from the website to get an overview of the syntax and Spyder-specific commands. For other example code, you can see more tutorials here.

Example Code – Verilog part

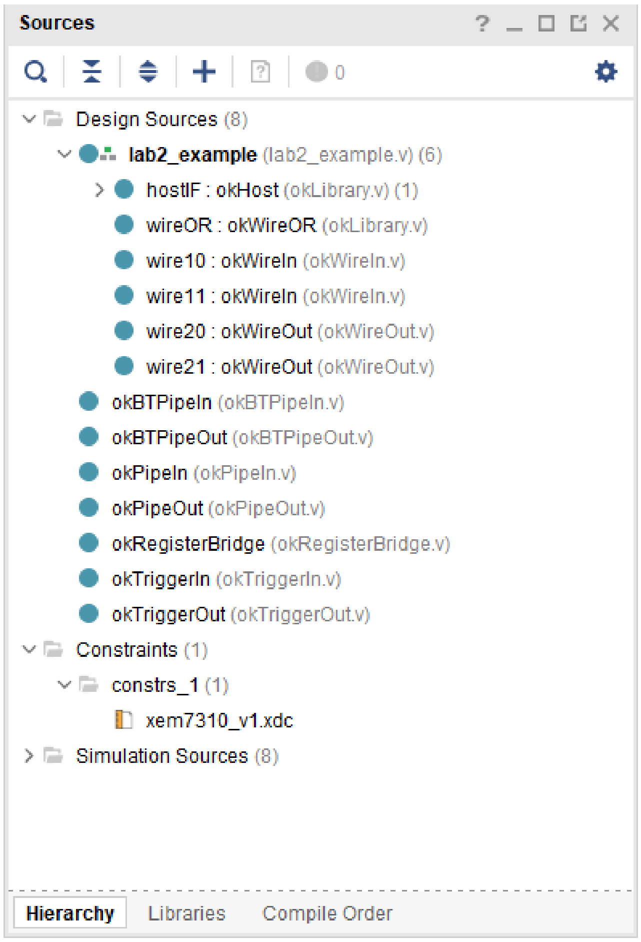
If the above was unclear, it will hopefully be clearer with an example. For simplicity, we'll do everything in one file, so you can see what's going on everywhere at once. For future labs, you are **highly encouraged** to break your code into modules, as it will become difficult to manage your code otherwise. To ensure you're comfortable with the first part of the lab, we'll make a new project in Vivado. Create a new project with the name Lab2, and then download the **lab2_example.v** and **lab2_example.py** from the course website. You can also download the **xem7310_v1.xdc** from lab 1. Most likely these files will be in your Download folder and you can keep them there for now. Vivado will copy them in particular directory structure as we add them to the project. Follow the next steps to add the Verilog code and constrains file to your project.

1. Select **Add Sources** from the *Flow Navigator* plane.
2. Select the *Add or Create Design Sources* option in the next window and click **Next**.
3. Click on *Add Files* button in the next window. Browse to the directory where you have downloaded the **lab2_example.v** file. Select the **Copy sources into project**. Once you select this option, your test bench file will be copied into a directory structure that Vivado likes to organize its files. This way, you don't have to worry where exactly the test bench file should be placed.
4. Click on the **Finish** button.
5. Next, we will add the **xem7310_v1.xdc** constrain file to the project. Select **Add Sources** from the *Flow Navigator*.

6. Select the *Add or Create Constrains* option and click **Next**.
7. Click on *Add Files* button in the next window. Browse to the directory where you have downloaded the **xem7310_v1.xdc** file. Select the **Copy sources into project**. Once you select this option, your test bench file will be copied into a directory structure that Vivado likes to organize its files. This way, you don't have to worry where exactly the test bench file should be placed.

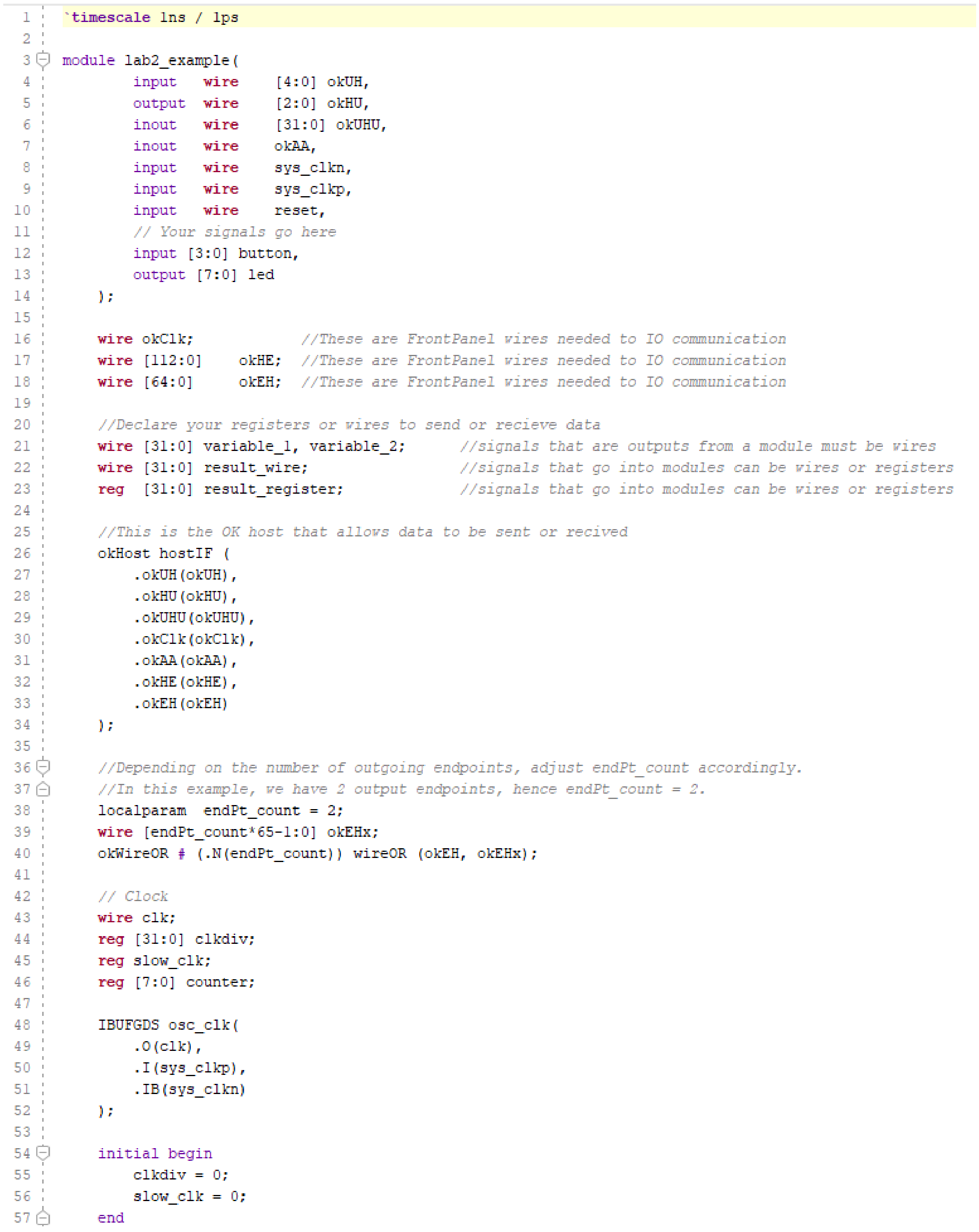
If you try to synthesize your project now, it will fail. Despite using all the various okXXX modules, we haven't defined them anywhere in the project. These modules are provided by OpalKelly and you can locate them at "C:\Program Files\Opal Kelly\FrontPanelUSB\FrontPanelHDL\XEM7310-A75".

Copy all Verilog files from the OpalKelly folder (i.e. all files with extension *.v*) in your project folder and add them to your project. You can leave the default file import options, and everything should import successfully. Now that's done, you can synthesize, implement, and generate the programming file (you may have a few warnings, that's fine). Your *Source* window in your project should look like this if all steps are completed successfully:

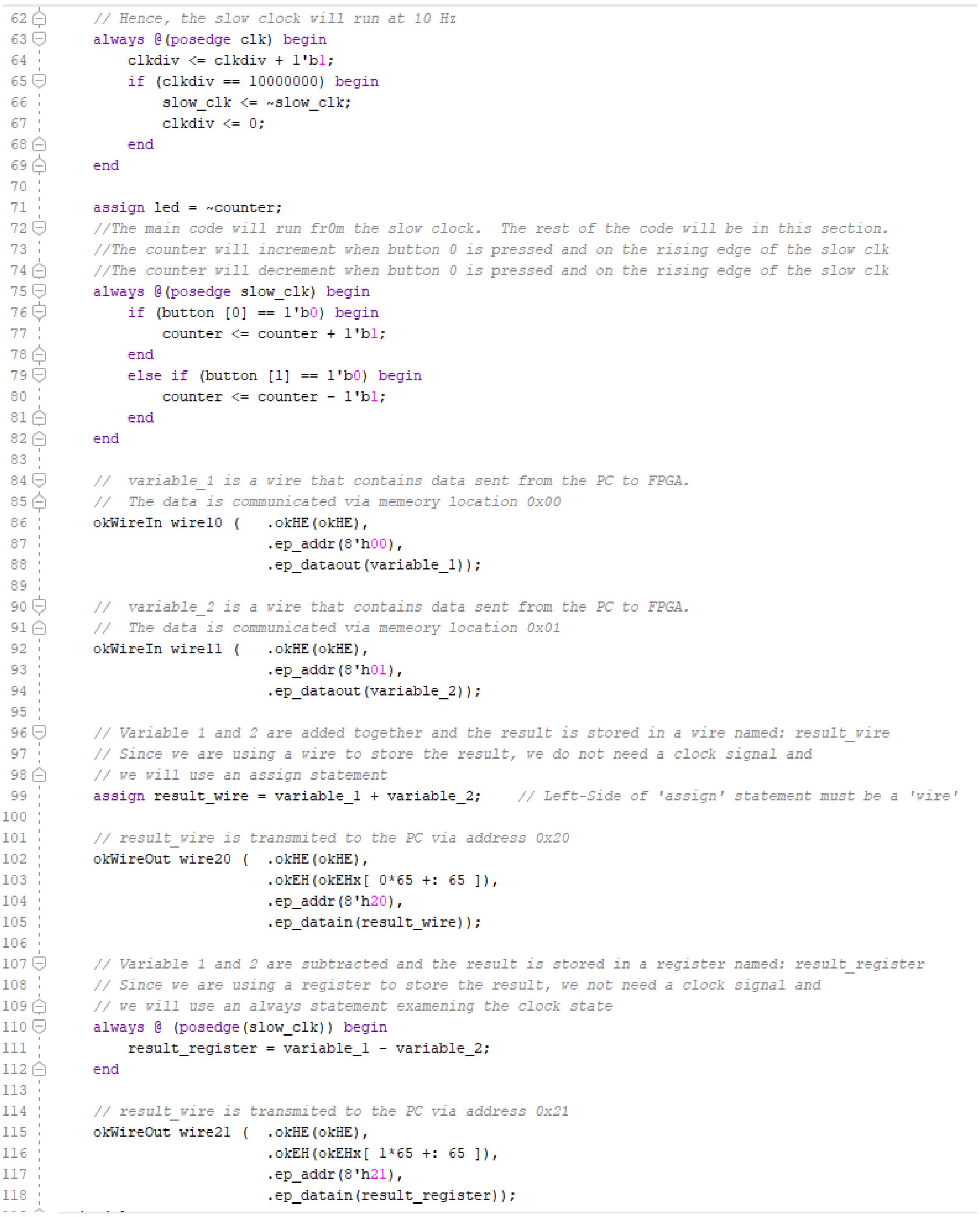


```
Design Sources (8)
  lab2_example (lab2_example.v) (6)
    hostIF : okHost (okLibrary.v) (1)
    wireOR : okWireOR (okLibrary.v)
    wire10 : okWireIn (okWireIn.v)
    wire11 : okWireIn (okWireIn.v)
    wire20 : okWireOut (okWireOut.v)
    wire21 : okWireOut (okWireOut.v)
  okBTPipeIn (okBTPipeIn.v)
  okBTPipeOut (okBTPipeOut.v)
  okPipeIn (okPipeIn.v)
  okPipeOut (okPipeOut.v)
  okRegisterBridge (okRegisterBridge.v)
  okTriggerIn (okTriggerIn.v)
  okTriggerOut (okTriggerOut.v)
Constraints (1)
  constrs_1 (1)
    xem7310_v1.xdc
Simulation Sources (8)
```

Open **lab2_example.v** file by double clicking on the file name. You should see the following code displayed on your computer:



```verilog
`timescale 1ns / 1ps

module lab2_example(
        input   wire    [4:0] okUH,
        output  wire    [2:0] okHU,
        inout   wire    [31:0] okUHU,
        inout   wire    okAA,
        input   wire    sys_clkn,
        input   wire    sys_clkp,
        input   wire    reset,
        // Your signals go here
        input [3:0] button,
        output [7:0] led
    );

    wire okClk;            //These are FrontPanel wires needed to IO communication
    wire [112:0]    okHE;  //These are FrontPanel wires needed to IO communication
    wire [64:0]     okEH;  //These are FrontPanel wires needed to IO communication

    //Declare your registers or wires to send or recieve data
    wire [31:0] variable_1, variable_2;      //signals that are outputs from a module must be wires
    wire [31:0] result_wire;                 //signals that go into modules can be wires or registers
    reg  [31:0] result_register;             //signals that go into modules can be wires or registers

    //This is the OK host that allows data to be sent or recived
    okHost hostIF (
        .okUH(okUH),
        .okHU(okHU),
        .okUHU(okUHU),
        .okClk(okClk),
        .okAA(okAA),
        .okHE(okHE),
        .okEH(okEH)
    );

    //Depending on the number of outgoing endpoints, adjust endPt_count accordingly.
    //In this example, we have 2 output endpoints, hence endPt_count = 2.
    localparam  endPt_count = 2;
    wire [endPt_count*65-1:0] okEHx;
    okWireOR # (.N(endPt_count)) wireOR (okEH, okEHx);

    // Clock
    wire clk;
    reg [31:0] clkdiv;
    reg slow_clk;
    reg [7:0] counter;

    IBUFGDS osc_clk(
        .O(clk),
        .I(sys_clkp),
        .IB(sys_clkn)
    );

    initial begin
        clkdiv = 0;
        slow_clk = 0;
    end
```



```
    // Hence, the slow clock will run at 10 Hz
    always @(posedge clk) begin
        clkdiv <= clkdiv + 1'b1;
        if (clkdiv == 10000000) begin
            slow_clk <= ~slow_clk;
            clkdiv <= 0;
        end
    end

    assign led = ~counter;
    //The main code will run fr0m the slow clock.  The rest of the code will be in this section.
    //The counter will increment when button 0 is pressed and on the rising edge of the slow clk
    //The counter will decrement when button 0 is pressed and on the rising edge of the slow clk
    always @(posedge slow_clk) begin
        if (button [0] == 1'b0) begin
            counter <= counter + 1'b1;
        end
        else if (button [1] == 1'b0) begin
            counter <= counter - 1'b1;
        end
    end

    //  variable_1 is a wire that contains data sent from the PC to FPGA.
    //  The data is communicated via memeory location 0x00
    okWireIn wire10 (   .okHE(okHE),
                        .ep_addr(8'h00),
                        .ep_dataout(variable_1));

    //  variable_2 is a wire that contains data sent from the PC to FPGA.
    //  The data is communicated via memeory location 0x01
    okWireIn wire11 (   .okHE(okHE),
                        .ep_addr(8'h01),
                        .ep_dataout(variable_2));

    // Variable 1 and 2 are added together and the result is stored in a wire named: result_wire
    // Since we are using a wire to store the result, we do not need a clock signal and
    // we will use an assign statement
    assign result_wire = variable_1 + variable_2;    // Left-Side of 'assign' statement must be a 'wire'

    // result_wire is transmited to the PC via address 0x20
    okWireOut wire20 (  .okHE(okHE),
                        .okEH(okEHx[ 0*65 +: 65 ]),
                        .ep_addr(8'h20),
                        .ep_datain(result_wire));

    // Variable 1 and 2 are subtracted and the result is stored in a register named: result_register
    // Since we are using a register to store the result, we not need a clock signal and
    // we will use an always statement examening the clock state
    always @ (posedge(slow_clk)) begin
        result_register = variable_1 - variable_2;
    end

    // result_wire is transmited to the PC via address 0x21
    okWireOut wire21 (  .okHE(okHE),
                        .okEH(okEHx[ 1*65 +: 65 ]),
                        .ep_addr(8'h21),
                        .ep_datain(result_register));
```

The main purpose of this code is to demonstrate how to send and receive data between the FPGA and PC. We are going to send two different variables, named variable_1 and variable_2, from the PC to the FPGA. These two values will be transmitted as wires. The two variables will be then added, and the result will be transmitted back to the PC as a **wire**. Also, these two variables will be subtracted, and the result will be transmitted back to the PC as a register. This will enable you to understand the difference between registers and wires. Pay close attention how we handled these two types of variables in the code.

Let's look at some of the different sections in the code. The module now has few more input/output wires declarations than in lab 1. Most of these variables must deal with the OpalKelly implementation of the USB interface. Some of your wires, such as led and button, are also located here.

The first thing to note is that you must use okWireOR when you are using okWireOut modules. This enables more efficient data transfer rates between the FPGA and PC. okWireOR specifies how many output endpoints are in your design. In our example code, we will use two output entities: one will be a wire, named result_wire, and the second one will be a register, named result_register. We will use a local parameter variable named endPt_count to specify the number of output endpoints. If you are going to use more output endpoints, make sure you update the value of endPt_count variable accordingly. For example, if you are going to use five different okWireOR modules, then endPt_count should be initialized to five.

Let's look at how data is received in the FPGA. We are using okWireIn module to receive data and we have declared two different modules, wire10 and wire 11. The name of these modules can be anything and you should use names that will make you debug easier. Both modules will receive data from the PC but at different memory addresses. The first one will receive data at memory address 0x00 and the second will receive data at 0x01 memory address. The data will be stored in variable_1 and variable_2 as wires, respectively. Since all these variables are wires, the result is computed instantaneously once the new data is received in the FPGA. These types of circuits are named asynchronous. Note, that you will always use wires to receive data in the FPGA using OK modules.

Let's look at how data is sent to the PC from the FPGA. We are using okWireOut for this task. The data that will be sent to the PC can be either a register or a wire. The first okWireOut module is called wire20 and sends the sum of the two variables that were previously received. The result is stored as a wire and this is the reason we are using the assign statement. The assign statement updates the values of the wires immediately i.e. without the use of a clock or asynchronously. The result from the addition is sent to the PC via memory address 0x20.

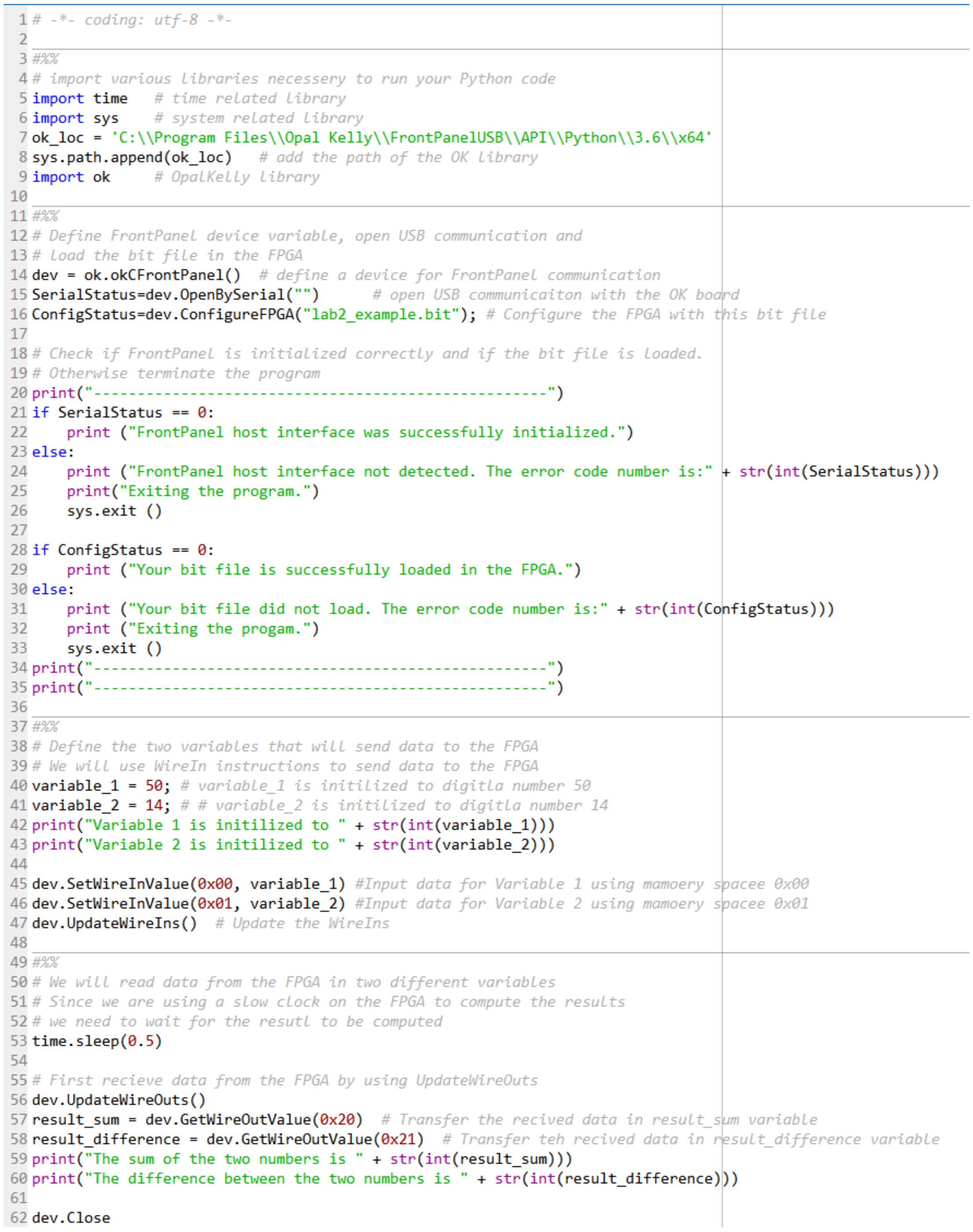
The second operation that is performed on the two variables is subtraction and the result is stored in a register named result_register. To compute this operation correctly, we need a clock because we are dealing with a register i.e. synchronous circuits. This is the reason that the subtraction is computed in the always block where we evaluate the code if the clock has changed its state. We are using the slow clock for this operation, but you can use any clock if you wish. The result from this operation is sent to the PC via memory address 0x21.

Example Code – Python part

Now that everything is finished on the Verilog side, we will focus on the Python side. Make sure the **FrontPanel** program is closed for this part of the exercise, or else the example code won't work. Once **FrontPanel** is opened, it keeps the communication port open between the PC and USB and it does not allow another program to connect to the FPGA. From now on, you should not use **FrontPanel** software to load your bit file. This will be done only through Python only. Make sure that **FrontPanel** software is closed. Otherwise your Python code will not be able to communicate with your board.

Download **lab2_example_python.py** file from the course website and store it in a new project directory. It would be good to create a directory within lab2 project which will be dedicated only to Python code.

Go to your new project directory and open **lab2_example_python.py** in Spyder. Here is a copy of the code you should see in Python (i.e. Spyder IDE).



```python
# -*- coding: utf-8 -*-

#%%
# import various libraries necessery to run your Python code
import time   # time related library
import sys    # system related library
ok_loc = 'C:\\Program Files\\Opal Kelly\\FrontPanelUSB\\API\\Python\\3.6\\x64'
sys.path.append(ok_loc)   # add the path of the OK library
import ok     # OpalKelly library

#%%
# Define FrontPanel device variable, open USB communication and
# load the bit file in the FPGA
dev = ok.okCFrontPanel()  # define a device for FrontPanel communication
SerialStatus=dev.OpenBySerial("")      # open USB communicaiton with the OK board
ConfigStatus=dev.ConfigureFPGA("lab2_example.bit"); # Configure the FPGA with this bit file

# Check if FrontPanel is initialized correctly and if the bit file is loaded.
# Otherwise terminate the program
print("----------------------------------------------------")
if SerialStatus == 0:
    print ("FrontPanel host interface was successfully initialized.")
else:
    print ("FrontPanel host interface not detected. The error code number is:" + str(int(SerialStatus)))
    print("Exiting the program.")
    sys.exit ()

if ConfigStatus == 0:
    print ("Your bit file is successfully loaded in the FPGA.")
else:
    print ("Your bit file did not load. The error code number is:" + str(int(ConfigStatus)))
    print ("Exiting the progam.")
    sys.exit ()
print("----------------------------------------------------")
print("----------------------------------------------------")

#%%
# Define the two variables that will send data to the FPGA
# We will use WireIn instructions to send data to the FPGA
variable_1 = 50; # variable_1 is initilized to digitla number 50
variable_2 = 14; # # variable_2 is initilized to digitla number 14
print("Variable 1 is initilized to " + str(int(variable_1)))
print("Variable 2 is initilized to " + str(int(variable_2)))

dev.SetWireInValue(0x00, variable_1) #Input data for Variable 1 using mamoery spacee 0x00
dev.SetWireInValue(0x01, variable_2) #Input data for Variable 2 using mamoery spacee 0x01
dev.UpdateWireIns()  # Update the WireIns

#%%
# We will read data from the FPGA in two different variables
# Since we are using a slow clock on the FPGA to compute the results
# we need to wait for the resutl to be computed
time.sleep(0.5)

# First recieve data from the FPGA by using UpdateWireOuts
dev.UpdateWireOuts()
result_sum = dev.GetWireOutValue(0x20)  # Transfer the recived data in result_sum variable
result_difference = dev.GetWireOutValue(0x21)  # Transfer teh recived data in result_difference variable
print("The sum of the two numbers is " + str(int(result_sum)))
print("The difference between the two numbers is " + str(int(result_difference)))

dev.Close
```

Let us look over the code and understand its functionality. First, we need to import two libraries that are necessary for our project and these are: *time and sys*. Next, we define where are the support files for the OpalKelly library and then we import them in the code. The OpalKelly libraries are located in C:\Program Files\Opal Kelly\FrontPanelUSB\API\Python\3.6\x64 directory. In future codes, you will be importing additional modules in this section of the code, such as plot libraries, instrumentation libraries and others.

Next, we need to open the USB communication interface between the PC and OpalKelly board. This is done by the following instruction: *dev.OpenBySerial("")*. Once the USB interface is established, we need to download the bit file that we generated from Vivado. You can do so by executing the following instruction: *dev.ConfigureFPGA("lab2_example.bit")*. In lab 1, you essentially did these two steps from the graphical user interface. Starting from this lab, you will use Python to download your bit file into the FPGA and proceed with testing your Verilog code.

The *lab2_example.bit* file is generated by Vivado, once you have successfully completed and synthetized your Verilog code. This file is in one of the subdirectories belonging to your lab2 Verilog project. Search for this file, copy it and place it in the Python project directory which contains *lab2_example_python.py* file.

In the future, you should not copy the bit file back and forth between these directories. Instead, you should specify the relative path for the location of the bit file. The reason for this is the following. During the debugging phase of your Verilog code, you will be resynthesizing your main code and flashing the FPGA many times. Sometimes you will forget to copy your new bit file to the python code directory and you will waste time debugging old code. To avoid this problem, use the relative path which will describe the location of your bit file in the Python code.

Before proceeding with your code, you should always check if indeed there is an OpalKelly board and that the bit file was successfully downloaded. Python will not let you know if these two instructions were not successfully executed. You need to explicitly check for error messages in your code and take adequate steps if they occur. For example, *if SerialStatus == 0* statement checks if there was an error encountered when the board serial interface was initialized. Never skips these errors checks in future codes. It will save you precious time by realizing your error is simply because your board is not working properly and not due to erroneous finite state machine.

Next, we specify two different variables, named variable_1 and variable_2. Thse two variables are initialized to some randomly chosen digital numbers. Feel free to change the value of these numbers. Using *dev.SetWireInValue(0x00, variable_1)* instruction, we are beginning to transmit this variable to the FPGA via memory address 0x00. Variable_2 will be transmitted via memory space 0x01. The data transmission for both variables is completed once *dev.UpdateWireIns()* is executed.

Next, we will receive the data from the FPGA and display the results on the screen. Remember that the Verilog code will sum and subtract both numbers that we had previously transmitted and send the result back via memory addresses 0x20 and 0x21, respectively. First, we have to receive the data from the FPGA by executing the *dev.UpdateWireOuts()* instruction. Next, we display the results on the screen. Since the Verilog code is using the slow clock to compute the result, which runs at 10 Hz, we put the Python code to sleep for half a second before receiving the data from the FPGA. Remove this statement and observe the result. It will be erroneous. You can also fix this by running the Vivado code out of the main, high frequency clock signal.

Checkpoint (140 points)

Your assignment is to develop Verilog and Python code for two separate exercises and demonstrate its functionality to the TA. You should use the Verilog and Python code from the first lab as a starting point for these exercises.

Exercise 1 (60 points): The first assignment is to write both Verilog and Python code that will perform the following task:

- You will define an 8-bit counter in Verilog and use the buttons to control the counter. The counter will be clocked at 5 Hz and the results will be displayed on the LEDs. Here are the rest of requirements for your Verilog implementation:
    - If button[0] is pressed, all LEDs should turn on.
    - If button[1] is pressed, all LEDs should turn off.
    - If button[2] is pressed, the LEDs should count up by 2.
    - If button[3] is pressed, the LEDs should count down by 2.
    - If more than one button is pressed, nothing should happen.
- The value from the counter will be also transmitted to the PC and displayed on the screen. On the PC side, you will write a python code that will receive the counter value and display it on the screen.

Exercise 2 (60 points): We will use the code that you develop in part 1 of this lab and add two additional requirements. The first requirement is that the counter clock frequency will be programmable and specified by the end user in Python. In other word, the clock divider variable will be specified and controlled from Python. Once the user inputs a new clock divider, this variable will be sent to the FPGA and the new clock speed will be updated.

The second requirement is that when the counter reaches 100, the counter will be rest to 0. The decision for resetting the counter will come from the PC. In other words, as the counter value is transmitted from the FPGA to the PC, the PC will decide if the counter needs to be reset or not. Once the PC determines that the counter's value has reached 100, the PC will send a reset signal to the FPGA (i.e. Verilog code) and the counter will be reset once this signal is received. The reset decision process should not be implemented only in Verilog.

Demonstrate both examples to the TA.

Postlab Questions (5 points each question):

8. How much data can be sent per second using okWireOut?
9. How much data can be sent per second using okWireIn?
10. Look at the Project Summary window. How many resources on the FPGA are used to implement your code? How does this compare to lab 1?
11. What is the maximum allowed values for both variable_1 and variable_2 such that the addition of these two number is correct using the example code provided in this lab? What happens if you go over these maximum values? Double check this in your code to make sure your answer is correct.
12. Include a printout of both codes with the final report.