ECE437: Sensors and Instrumentation

Lab 1: Introduction to Opal Kelly FPGA and Digital I/O Lab

Introduction

Welcome to ECE437! This class focuses on developing communication interfaces with variety of sensors, such as temperature, pressure, capacitive, image sensors and others using Verilog and Python programming languages. As you develop the necessary firmware using Verilog and software in Python to acquire data from various sensors, you will also gain important understanding how these sensors operate. By the end of the course, you will gain knowledge how to develop large scale projects that will enable you to simultaneously and in real-time communicate with collection of different sensors. The data acquired from these sensors will be transferred to the PC using USB 3.0 interface and displayed in real-time on the computer screen. Although we don't focus on any particular sensory application in this class, the sensor platform that is available for you in this class can be used for other courses, such as senior design and others. You are strongly encouraged to leverage the resources available to you in this class and used them to solve real life problems outside this class.

We have developed a custom printed circuit board (PCB) which houses several different sensors: temperature, capacitive, humidity, imaging sensor and others. The board also has several different LEDs and push buttons that will be used to output and input data. This custom PCB interfaces with an FPGA board designed by OpalKelly Inc. OpalKelly provides versatile data acquisition cards based on Xilinx FPGA chips and today they are one of the leading companies in this space. The company was started few years ago by two undergraduate students in the Electrical Engineering department at the University of Illinois at Urbana-Champaign, while they were working on their senior design project. Having struggled with designing an interface between sensors and computers for their senior design project, they created a versatile digital input/output board that can be used for various projects and applications. Today, their cards are used in many industrial applications, such as automotive and airline industry, as well as for low-budget prototyping.

The OpalKelly board that we will be using in this class is XEM7310-A75. Additional information about this board can be located on the OpalKelly's website, located here. It is important to understand the different resources you will be using in this class which will enable you to communicate and acquire data from various sensors. The FPGA that is housed on the OpalKelly XEM7310-A75 board is physically connected to the input/output pins of the various sensors housed on our custom PCB. The FPGA also has capabilities to communicate to the PC via USB 3.0 interface using OpalKelly's proprietary firmware and software. This USB interface capabilities enable fast and easy communication between the sensors and PC using the FPGA as the command central.

You will be developing firmware code for the FPGA in Verilog using Xilinx's Vivado software – the leading development platform for hardware description language such as Verilog and VHDL. To receive data on the PC from the FPGA, you will be developing Python code. Note, OpalKelly also provides support for Matlab, C, C# and other programming languages. OpalKelly's website

provide numerous firmware and software examples, tutorial and other advices. They provide discussion forums where you can find answers to commonly encountered problems. You are strongly encouraged to explore OpalKelly's website and get familiar with their support. Here is a link where you can locate some of the tutorials and examples: link.

In this class, we will be developing primarily finite state machines (FSM) in Verilog hardware description language. Throughout the semester and especially in the next two labs, you will learn about the basic Verilog syntax that will enable you to write successful FSMs. In this first lab, you will learn about declaring input/output variables and communicating with the outside world using buttons and LED display. You will write a simple FSM that will enable you to control the clock speed of your FSM. There are lot of resources on the web that can help you master Verilog code. Here is one place where you will find many useful examples: link.

Relevant Documents for this Lab

Required reading material for this lab:

1. Information for the XEM7310-A75 board can be located here.
2. OpalKelly Verilog and Python tutorial can be located here.
3. Verilog example code on the course website: intro.v and xem7310_v1.xdc.

Additional reference material:

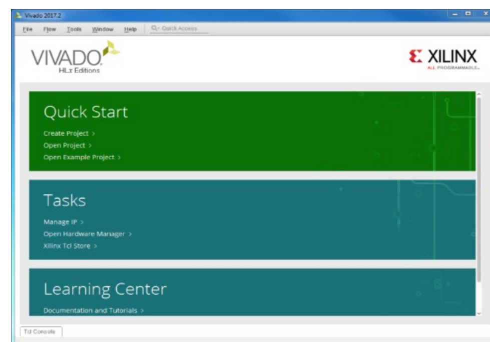
1. Verilog tutorials and example can be located here.
2. The sensor board layout and schematic are available on the course website.

The goals of this lab are:

1. Guide you to the process of setting up a project in Xilinx.
2. Write an introductory FPGA programs from Xilinx Vivado.
3. Successfully synthesize the code and program the FPGA using OpalKelly FrontPanel.

LEDs and Buttons

We'll begin with the Xilinx Vivado. Start **Vivado** from the Desktop or the Start Menu. You'll see the following screen:



Select **Create Project** to start the wizard. Give your project a simple name such as *Lab1*. Make sure to choose an easily accessible location to save your project, as we will need to get access to

some of the files in the project directory. Also assure that the "Create project subdirectory" box is checked. This will create your project as a folder, which is necessary to be able to store the project files you will be creating later.

Select **RTL Project** as your project type and check the "Do not specify sources at this time" box. On the **Default Part** screen, use the **Family** and **Package** filters with "Artix-7" and "fgg484" respectively. Choose **xc7a75tfgg484-1,** click **Next**, and then click **Finish**.

Warning: Make sure you select the aforementioned FPGA device (family and package). If you make the wrong selection, you will get strange errors later in your design and it will not be obvious why your design is not synthesizing. Please double check that you have selected the correct settings on this window. This oversight has been the number one issue preventing students from completing this and future labs in timely manner.

New Project

Default Part

Choose a default Xilinx part or board for your project. This can be changed later.

Parts | Boards

Reset All Filters

Category: All   Package: fgg484   Temperature: All Remaining

Family: Artix-7   Speed: All Remaining

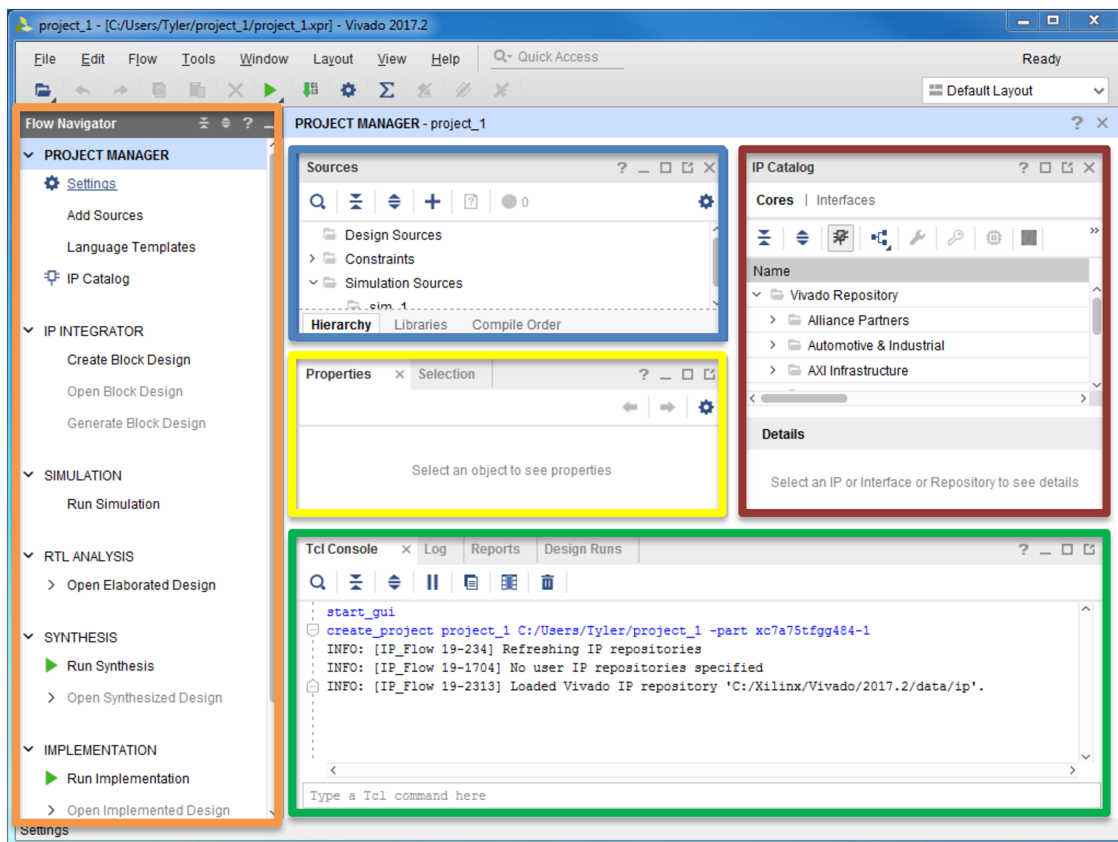
Search:

Part	I/O Pin Count	Available IOBs	LUT Elements	FlipFlops	Block RAMs	Ultra RAMs	DSPs
xc7a75tfgg484-3	484	285	47200	94400	105	0	180
xc7a75tfgg484-2	484	285	47200	94400	105	0	180
xc7a75tfgg484-2L	484	285	47200	94400	105	0	180
xc7a75tfgg484-1	484	285	47200	94400	105	0	180
xc7a75tifgg484-1L	484	285	47200	94400	105	0	180
xc7a100tfgg484-3	484	285	63400	126800	135	0	240
xc7a100tfgg484-2	484	285	63400	126800	135	0	240
xc7a100tfgg484-2L	484	285	63400	126800	135	0	240
xc7a100tfgg484-1	484	285	63400	126800	135	0	240
xc7a100tifgg484-1L	484	285	63400	126800	135	0	240

? < Back   Next >   Finish   Cancel

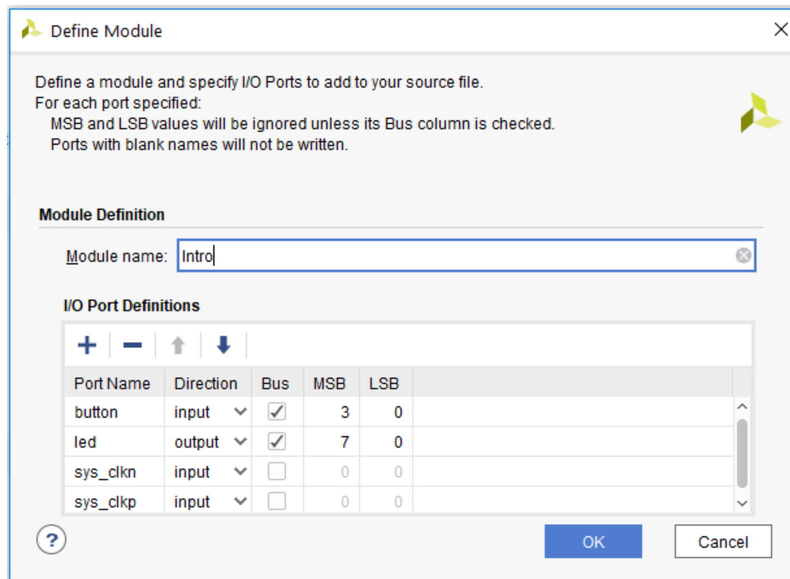
XEM 7310 Default Part window

Now that we have a project initialized, let's go into the detail of the various panes in Vivado. The left pane, highlighted in orange, is the *Flow Navigator*. From here, you can create diagrams, add sources to your program, run simulations, implementations, and create bitstreams. The box highlighted in blue near the top left is the *Sources Window* that allows you to view and manage your sources and constraints. Below that is a box highlighted with a yellow rectangle. This pane will allow you to see some of the properties of your simulation, if you wish to do so. On the right is the red-highlighted "IP Catalog" pane, which contains many wizards provided by Xilinx, along with many of the functions supported on the board. Lastly, the bottom pane, highlighted in green, is the *Console*. All the debugging warnings and errors show up here, as well as interaction with the simulator.



Default Project Manager Window Panes

Get started by adding a new source file from the *Flow Navigator* using the "Add Sources" button. Choose **Add or create design sources** in the *Add Sources* dialogue box. In the next window, click **Create File** and choose **Verilog** as the file type, and give it a simple file-name like "Intro" Make sure the file location says "<Local to Project>". Once the file is added, click **Finish**. On the next screen, we'll add our input and output variables, which you can copy from the example below.



In this example, *button* is a 4-bit input signal; *led* is an 8-bit output signal; *sys_clkn* and *sys_clkp* are single bit input signals. These four signals are the input and output signals from the module that we will create shortly. They are also input/output signals from the FPGA to the outside world in this example. You will shorty create a constrains file which will define how these signals are mapped to physical pins on the FPGA.

After pressing **OK**, we're finally ready to write some Verilog. Locate the new source file in the *Sources* panel and double clicking on the filename "*Intro.v*". There will be few lines of code already inserted for you by Vivado in this file. This will be the input/output variables that this module will use to communicate with the rest of project. In our example, these four variables will communicate between the FPGA and the outside world, such as LEDs, buttons and clock signals.

Our first example code will be a counter. We will define an 8-bit counter whose contents will be displayed on the LEDs. We will use two different buttons to determine if the counter will increment or decrement. When either one of these buttons is pressed, the counter will count up or down, respectively. Note that the buttons and LEDS are **active low** by default. This was a design decision when we created this custom board. When the buttons are pressed, they are in state 0. Otherwise, the buttons are in state 1. Similarly, the LEDs turn on when a low signal (i.e. state 0) is presented and they are off when a high signal is applied.

Finite state machines require a clock signals which controls the current state of your code and computes the next state. The counter is a simple example of FSM and requires a clock signal which will dictate when the counter will change its current state. The two clock signals, *sys_clkn* and *sys_clkp*, are low voltage differential clock signals, also known as LVDS signals. These two signals cannot be directly used to control the various finite state machines that you will develop. Instead, we will generate a main clock signal from these two LVDS clock signals using the IBUFGDS module. This new clock signal, which we will define as *clk* in our code, operates at 200MHz.

Since the new clock signal (i.e. *clk*) operates at 200MHz, it will be impossible for us to observe the counter changes on the LEDs when we press various buttons. These high frequency signals

can only be observed on the oscilloscope, which will be explored in a later lab. To be able to observe the various states of the counter on the LEDs, we will step down the main clock by writing your first FSM code. Once we generate a slow clock, we will write a second FSM that will control the counter state depending on which buttons is pressed. The second FSM will run from the slow clock and will control the counter current and next state depending on which button is pressed.

The counter code is shown next and can be downloaded from the course website. Can you concisely explain what this module does? If you are confused, the answer is below.

```
`timescale 1ns / 1ps

module intro(
    input [3:0] button,
    output [7:0] led,
    input sys_clkn,
    input sys_clkp
    );

    reg [23:0] clkdiv;
    reg [7:0] counter;
    reg slow_clk;

    // This section defines the main system clock from two
    //differential clock signals: sys_clkn and sys_clkp
    // Clk is a high speed clock signal running at ~200MHz
    wire clk;
    IBUFGDS osc_clk(
        .O(clk),
        .I(sys_clkp),
        .IB(sys_clkn)
    );

    initial begin
        clkdiv = 0;
        counter = 8'h00;
    end

    assign led = ~counter;

    // This code creates a slow clock from the high speed Clk signal
    // You will use the slow clock to run your finite state machine
    // The slow clock is derived from the fast 20 MHz clock by dividing it 10,000,000 time
    // Hence, the slow clock will run at 2 Hz
    always @(posedge clk) begin
        clkdiv <= clkdiv + 1'b1;
        if (clkdiv == 10000000) begin
            slow_clk <= ~slow_clk;
            clkdiv <= 0;
        end
    end

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

Here is a description of the code above. First, you should note how we have initialized the various registers (i.e. variables) in the code. All registers must be first declared using **reg** data type with the correct bit length. For example, *slow_clk* is a single bit register because clock signals require a single bit. *Counter* is an 8-bit register which is mapped directly to the LEDs.

Second, you should observe the various ways we have initialized the registers in the code. You can initialize registers using binary, hexadecimal or digital numbers. For example, counter is an 8-bit register. It is initialized to value 0 using hexadecimal notation 8h'00. The 8h means this is an 8-bit number initialized to hexadecimal value 0. When we increment the counter, we use binary notation 1'b1. Register *clk_div* is compared to decimal number 10,000,000 because we like to slow down the main clock 10 million times. Most of the time you will use digital values to initialize or compare the values of your registers. However, sometimes using binary or hex notation will make your problem easier to understand.

In the first FSM, we like to slow down the main clock, which runs at 200MHz, and create a slow clock that runs at 20 Hz. This is done by using an intermediate register (or variable) named *clkdiv*. The register *clkdiv* is first initialized to value 0. This is a very important step and always remember to initialize your registers in your code. Otherwise, you will not know what state they are initialized, and you might observe erroneous results or even your code might not run at all. In this example, at every rising edge of the high-speed clock signal, we will increment *clkdiv* and check if it exceeds 10,000,000. If it does, we will toggle the state of the *slow_clk* register. Hence, we toggle the *slow_clk* every 10 million cycles from the fast clock.

The second FSM controls the counter state. Note that the code that is in the *always* block runs out of the *slow_clk*. In fact, every time the *slow_clk* goes from state 0 to state 1, that code will be evaluated. In this code, we will decrement the counter register if button 0 is pressed. Otherwise, the counter is incremented.

Note that we have used the *assign* statement to map the counter register to the external LEDs via the variable *led*. The variable *led* is a wire (not a register) and this is why we use assign statement. If *led* was a register it will have to be in the *always* statement because register need clock signals to be updated. Since the external LEDs are active low, we have mapped the complement of the counter to the LEDs.

When you save a Verilog file in Vivado (Ctrl+S, File → Save, or click the save icon), it automatically parses and checks for correct Verilog syntax. The next step is to synthesize your code which is equivalent to compiling code in many programming language, such as C or C++. Since the synthesize process is slightly long, you are highly encouraged to save frequently to avoid wasting time on simple errors.

The next step is to add a constrain file in your design. The constrain file maps the input/output variables in your top module (i.e. *Intro.v* in this example) to physical pins on the FPGA. For example, the 4 buttons that are on the custom board are physically connected to 4 pins on the FPGA. The constrains file keeps track of all these physical connections to the proposer variable names. Since we have designed this board, we have created the necessary constrains file and you can download it on the course website.

Next, you will need to add the constrains file in your project. Go to the *Flow Navigator* pane again and add a new source, but this time choose **Add or create constraints.** Add the constrains file

that you downloaded from the course website by selection *Add Files* in the **Add or create constrains** window. The name of the constrain file is *xem7310_v1.xdc*.

The *Sources Pane* should now show the added constraints file. Open the constrain file by double click on the file name. Scroll through the file and locate the "led" variable. The led[0] variable is connected to pin A13 on the FPGA. This pin from the FPGA is directly connected to an LED on the OpalKelly board. You will refer to this constrains file later in the class when you will need to find which pins from the FPGA are connected to the various pins from the sensors.

Now that we have our top-level module and our pin-mapping, we are ready to synthesize and program the FPGA (or flash) with our first design. In the *Flow Navigator*, press **Run Synthesis** to start the build process. The *Console* will print a series of messages describing the synthesis. Once the synthesis is complete, you will be prompted to begin implementation. Go ahead and begin implementation as well. Once *that* is finished, you will have the option to **Generate Bitstream**. Select this option and click **OK** in the dialogue box. This command should go quite quickly and will generate a *.bit file used by the FPGA to run the program. If there are any error messages, they will be displayed in this window and you will need to correct them so that you can generate a bit file. Also pay attention to the warning message. Sometimes the reason for problems in your design are in the warning messages.

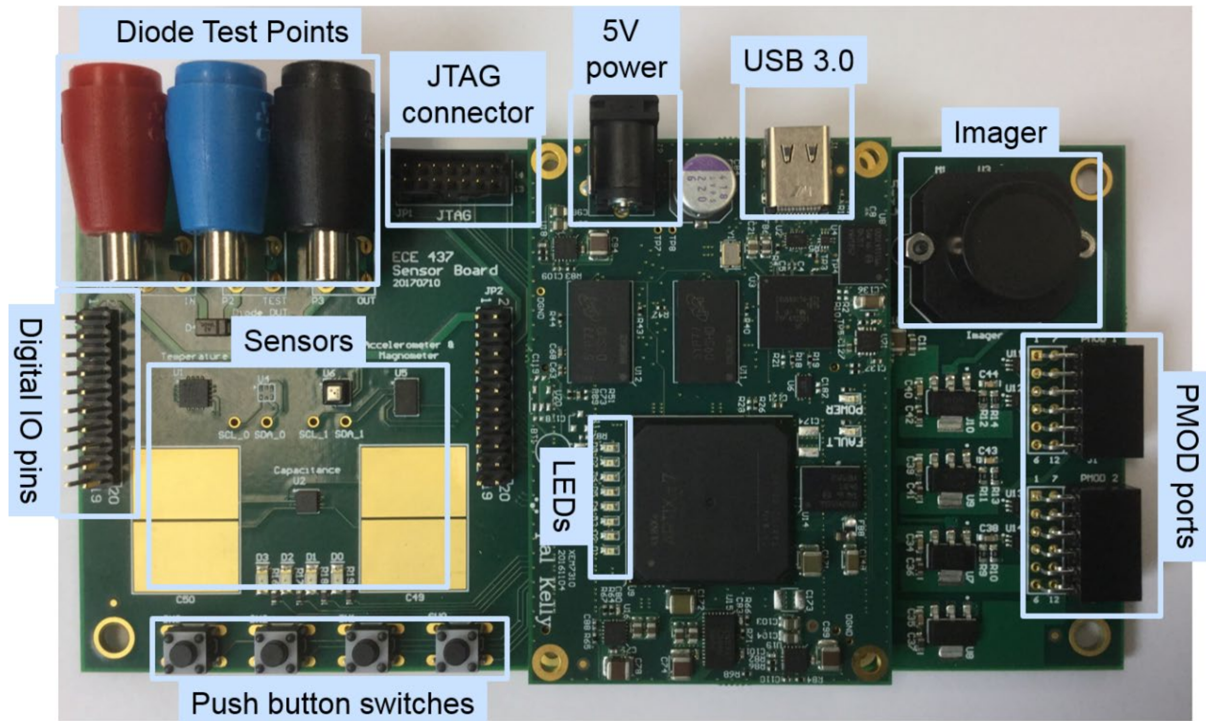
Custom Sensor Board Overview

Before you can flash your FPGA with the newly compiled bit file, you should get familiar with the sensor's board. The test board is composed of two PCB boards: the FPGA board designed by OpalKelly and our custom sensors board which houses various sensors. The image bellow outlines all the components on both boards. There are four switch buttons on the bottom left corner of the custom board. There are also eight LEDs on the OpalKelly board which are relevant for this exercise. Take the time to get familiar with all components.

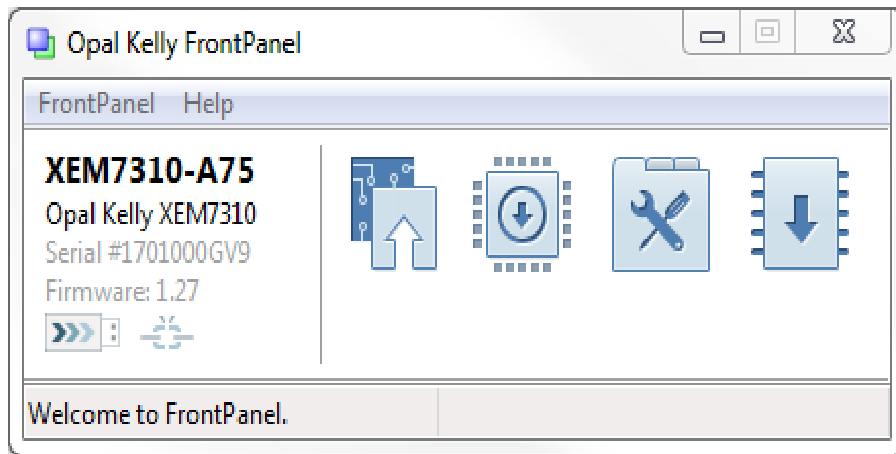
A schematic representation of our custom board can be located on the course website. In the schematic view, you can trace how the various sensors and other external components are connected to the FPGA.

Do not touch any of the component on the board with your hands because there is a high chance you will damage the board. As you walk around, you accumulate lot of electrical charges. When you touch the PCB, you will most likely discharge these charges on the electrical components and cause damage.

Next, you will need to power up your board and connected to the PC. Only use the power adapters that are provided with the PCB board. Connect the power cable to the "5V power" port and connect the board to the PC via the "USB 3.0" port. Note, that your computer has USB 2.0 and USB 3.0 ports. Although for these first few labs it will not make much difference which port on the computer you have connected the board, this will be an issue later in the class when we start using high speed data transfers. It is a good practice to start connecting the board to the correct USB port on the PC.



Now that you have created a bit file of your design, you are ready to program (or flash) the FPGA. Open the FrontPanel program from either the Desktop or Start Menu (under Opal Kelly -> FrontPanel). If you cannot locate the program in the start menu, open up Windows Explorer and go to: C:\Program Files\Opal Kelly\FrontPanelUSB\FrontPanel.exe. You'll be greeted with the following program:



Click the middle-left button (IC with down arrow) and go to your corresponding project directory. Select the <FileName>.bit file, where <FileName> is your top-level filename. The FPGA should automatically flash and start running your program. Test it out by pressing button[0] on the board and see how the counter increments or decrements.

Checkpoint 1 – (100 points)

The test code that you just compiled and tested it on the system ensures that your board is working, and you have written a correct HDL code. You will use this sample as a starting point for the code that you will need to develop.

You will need to write Verilog code that will do the following task:

- Count from 0 to 100 in steps of 10 and display the results on the LEDs. Stop counting once the counter reaches 100. (50 points).
- Count from 0 to 100 in steps of 10. Once the counter reaches 100, it will count back 0 in steps of 10. The counting up and down will be repeated indefinitely (50 pts).

Once you complete your code, please demonstrate it to the TA.

Post lab Questions (5 points each question):

1. How many total digital input/output pins does the XEM 7310-A75 board have?
2. What is the maximum clocking speed of the XEM 7310-A75 board?
3. How does the XEM 6002 board compare to XEM 7310-A75 in terms of logic gate count, transfer speeds between the board and PC, external memory, and clocking speed of digital logic?
4. Why is the *clkdiv* register 24-bit long?
5. If *clkdiv* is declared as an 8-bit register, what is the minimum frequency you can achieve for the *slow_clk* signal using these FSMs?
6. Look at the Project Summary window. In the Utilization section, you will find out the number of look up tables (LUT), flip flops (FF), input/output pins (IO) and buffers (BUFG) are used for your design. How many resources on the FPGA are used to implement your code?
7. Include a printout of your Verilog code with your report.