Lab 3: Programming the Basys3 Board

EE 316: Digital Logic Design

1 Overview

This lab is intended to introduce you to the board programming process of Vivado. By the end of this lab, you should be able to:

- describe (at a high level) what happens in each stage of the board programming process: synthesis, implementation, bitstream generation.
- describe the role of the constraints file in the board programming process.
- explain Verilog data types (`wire` vs `reg`) and their use, in addition to previously learned syntax.
- describe how the seven-segment display on the Basys3 board works.

2 Background

2.1 FPGA Design Workflow

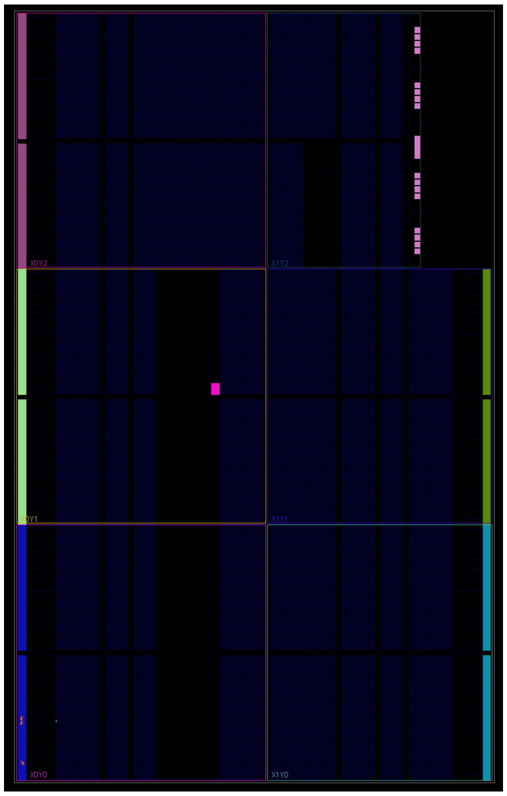
In the FPGA industry, there is a standard workflow that most FPGA designers (including Vivado) follow: simulation, synthesis, implementation, and programming. This section will break each of these processes down in more detail.

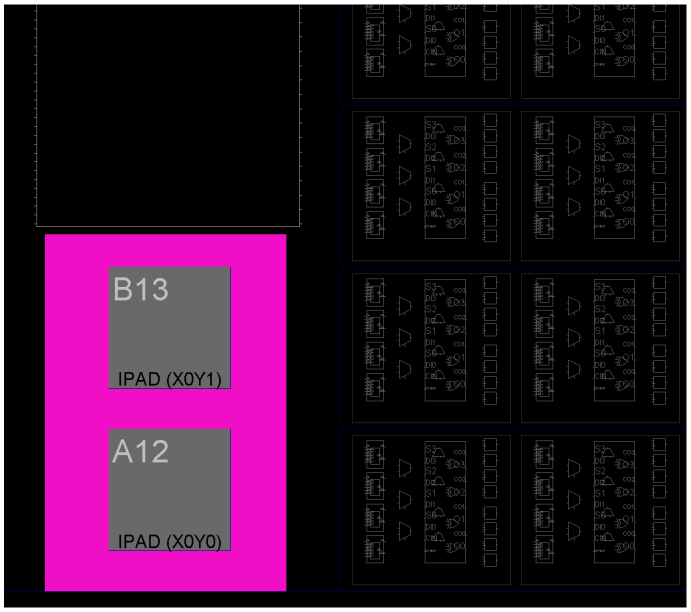
2.1.1 Simulation

During the simulation stage, Vivado acts similar to LogiSim/ModelSim/MultiSim. It compiles your code and generates a simulated circuit. Using your simulation source (your testbench), Vivado toggles the inputs to your module and observe the outputs and compiles the data into a nice waveform for you to view.

2.1.2 Synthesis

During the synthesis stage, Vivado compiles your code and generates a simulated circuit called a netlist. This netlist is similar to SPICE and dictates every node, component, module, and register in the resulting logic circuit with its name and what it is connected to on the board. This is also where Vivado optimizes your design, so your on-board circuit may look (and perform) slightly differently than in simulation.


(a) Implemented design



(b) FPGA internals

Figure 1: Inside of an FPGA

2.1.3 Implementation

Once the synthesis is complete, Vivado begins the implementation process. The implementation stage takes the netlist created in the synthesis stage and generates a board-level design. The implementation process is comprised of four steps: optimizing your design, placing your design, routing the components, and analyzing the timing of your design. Placing and routing are the key steps of the implementation stage, as this is where Vivado determines which lookup tables to use and which lookup tables need to be connected to each other to implement your code's functionality.

You can see the results of the implemented design (Figure 1) by opening it and zooming in. As you keep zooming in, Vivado will begin to show you the individual combinational logic blocks (CLBs) and circuitry inside the FPGA. This will take a bit, as Vivado's scale accurately represents the size of each block on the board.

2.1.4 Programming

After the implementation stage, Vivado generates a bitstream that boils all the information of your implemented design into a series of 0s and 1s that will eventually be stored in the lookup tables. This file can then be transmitted (via USB, SPI, UART or your favorite serial communication protocol of choice) to the FPGA, where the information will be used to set the lookup tables (LUTs) in the FPGA.

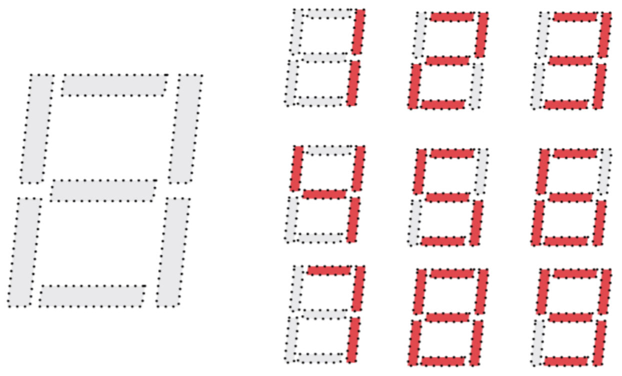


Figure 2: Displaying digits on the seven-segment display

2.2 Constraints Files

Constraints (`.xdc`) files are used in the synthesis and implementation steps of the FPGA design process to create the netlist and place your components. Most importantly, it tells Vivado which on-board (hardware) components correlate to the inputs and outputs of your (software) top module. These specifications drive the placement and routing of your design.

Constraints files have a specific syntax that Vivado uses to generate the settings for each pin on the Basys3 board. Consider a snippet of a constraints file below:

```
set_property PACKAGE_PIN V17 [get_ports a]
set_property IOSTANDARD LVCMOS33 [get_ports a]
```

In the snippet above, pin `V17` refers to switch `SW0` on the board. During the synthesis stage, Vivado will connect/map pin `V17` to input `a` in the top module in the netlist, and during the implementation stage, pin `V17` will be routed to `a`. Also, `LVCMOS33` is a keyword that sets pin `V17` to use 3.3V logic and defines a '1' as 3.3V. The board can be set to other logic standards such as TTL (5V logic).

Constraints files often come pre-packaged with the specific board you use. You can find the template for the constraints file for the Basys3 board under Files > Reference > Basys3_Master.xdc. To use, add the file as a constraints file via the "Add Sources..." dialog (the same as adding a design file, except choose "constraints file") and uncomment the necessary lines by removing the preceding hashtag on each line.

2.3 Seven-Segment Display

2.3.1 Operation

The four-digit seven-segment display on the Basys3 board allows us to display a variety of numbers, letters, and even words (Figure 2). This section will describe how the seven-segment display works and what you need to do to display a single digit/character.

Figure 3 displays the inputs and outputs of the seven-seg. Each segment is labeled a-g starting from the top and going clockwise, with g the line in the middle and dp the decimal point. The segment a's across all four digits are shorted together, as are all of segment b's, c's, etc. Furthermore, each digit has a common anode (AN0-AN3), which acts as an enable for all segments of that digit.

The segments and the anodes are both active-low devices, meaning that they must pass/be passed a '0' to be enabled or turned on. Therefore, if we wanted to display the number '3' in digit

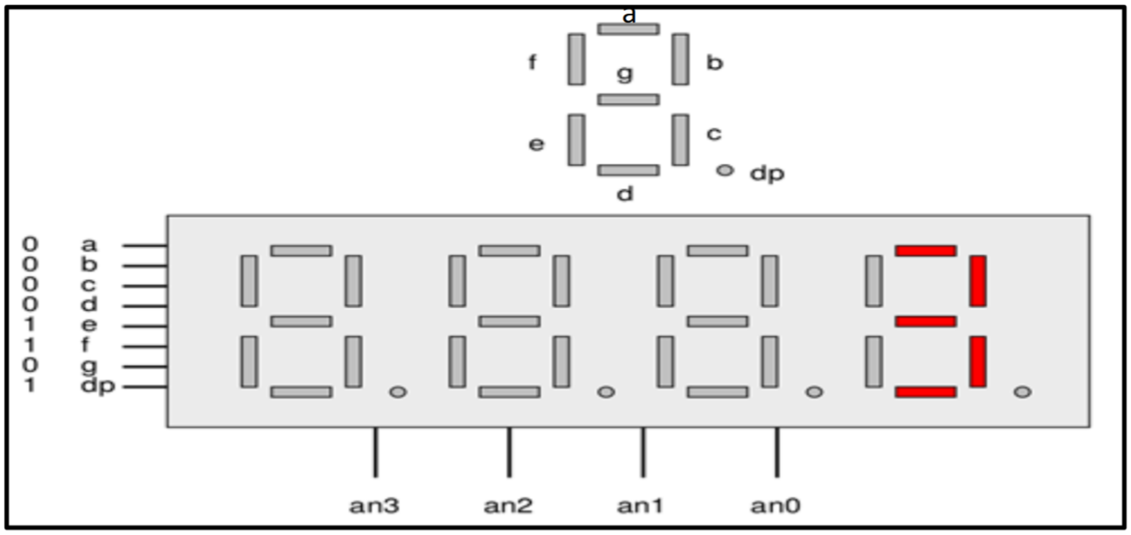


Figure 3: Inputs and outputs of the seven-segment display

0, we would set segments a-g and dp as 00001101 (to turn on segments a, b, c, d, and g) and set the anodes to 1110 (to enable digit 0 and disable the other anodes). Similarly, active-high devices - like the LEDs on the board - must be passed a '1' to be turned on. Switches are an active-high input since they input a '1' when flipped on and '0' when flipped off.

2.3.2 Circuitry

This section will delve into the circuitry interfacing the seven-segment display to the FPGA. You will not be held responsible for this information, but it is an interesting example of how a hardware circuit impacts software implementation.

Refer to Figure 4. Note that the anodes are connected to the seven-seg by a component represented as a triangle-like symbol with an arrow pointing into the flat side. This component is a pnp bipolar junction transistor (BJT) and effectively acts as an active-low switch in the circuit. When AN0=1, the BJT acts as an open circuit and does not allow current to flow from the 3.3V power bus to the corresponding digit on the seven-seg, effectively turning the entire digit off. When AN0=0, the BJT acts as a short circuit, allowing current to flow and driving the common anode of the segments to 3.3V. The other anodes operate in the same way. This behavior causes the anodes to be active-low: it is active only when a 0 is passed to it.

Each segment is controlled by a light-emitting diode (LED) that turns on when a positive voltage is applied across its terminals. This occurs when the common anode is at 3.3V and the cathode is at 0V, or grounded. When the common anode is at 3.3V (AN0=0) and the cathode is at 3.3V (seg=1), no voltage is applied across the diode (hence no current flows), so it does not turn on. Therefore, a segment is only lit when AN is passed a 0 and the segment is passed a 0. This makes the segments active-low devices.

2.4 Verilog Data Types: `wire` vs `reg`

Oftentimes, many students get confused about the difference between a `wire` and a `reg` in Verilog because they often think they are just datatypes like in software. However, this is ***NOT*** true. Both `wire` and `reg` represent hardware components on the FPGA that force restrictions on their usage in Verilog.

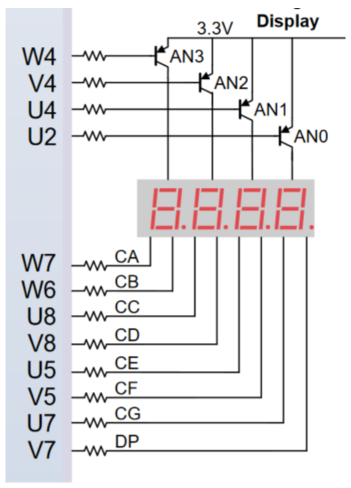


Figure 4: Interfacing circuitry of the seven-segment display

`wires` are connections made between two components, e.g. gates, components, modules, and registers. `wires` are used to model combinational logic and connect different modules. Inputs to modules must always be `wires`.

`reg`, short for register, is what you consider a "normal variable". In hardware, `regs` are initialized as D-flip-flops (D-FF), and the output of the D-FF is the value of the `reg` variable. Oftentimes, `regs` are used in `always` blocks as sequential elements or used to initialize values.

For more information and clarification on the difference between `wire` and `reg`, see http://www.asic-world.com/tidbits/wire_reg.html. You can also reference the "wire vs reg.pdf" file in Canvas under Files > Reference.

2.5 Bit Vectors in Verilog

Bit vectors (aka multi-bit variables, multi-bit buses) are variables with multiple bits. In the case of the seven-seg, it will be convenient to declare a 4-bit vector for the anodes and a 7- or 8-bit vector for the segments. This can be easily done as follows:

```
output [3:0] an;      //4-bit vector (of type wire) for anodes
reg [7:0] seg = 0;   //8-bit vector (of type reg) for segs
```

To access a specific bit in a vector, use the syntax "a[0]". For example,

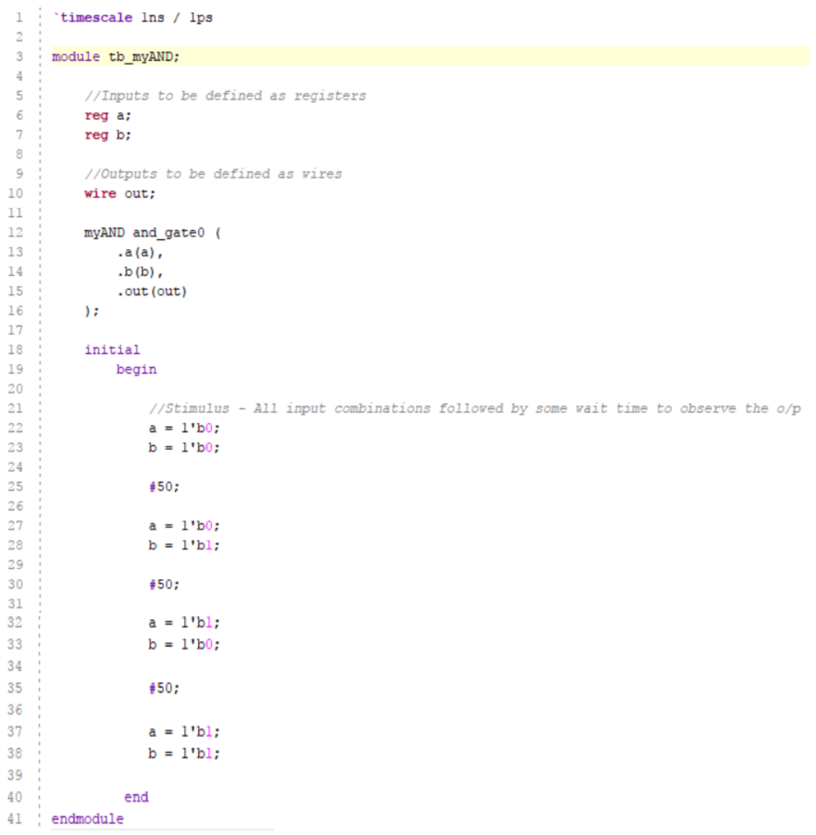
```
an[0] = 0;           //Clears LSB of an to 0
seg[7:5] = 3'b111;   //Sets 3 MSBs of seg to 1
assign wire c = an[0];
```

3 Procedure

3.1 Part A: Guided Design

For this part, we will walk you through how to implement your first FPGA program from scratch: an AND gate.

1. Create a new project with part number: XC7A35TCPG236-1. This is the Basys3 board.



```
`timescale 1ns / 1ps

module tb_myAND;

    //Inputs to be defined as registers
    reg a;
    reg b;

    //Outputs to be defined as wires
    wire out;

    myAND and_gate0 (
        .a(a),
        .b(b),
        .out(out)
    );

    initial
        begin

            //Stimulus - All input combinations followed by some wait time to observe the o/p
            a = 1'b0;
            b = 1'b0;

            #50;

            a = 1'b0;
            b = 1'b1;

            #50;

            a = 1'b1;
            b = 1'b0;

            #50;

            a = 1'b1;
            b = 1'b1;

         end
endmodule
```

Figure 5: AND testbench

2. Add a source file and name it "myAND". Define three I/O ports: `a`, `b`, and `out`. `a` and `b` are inputs, and `out` is an output. Once you click "OK", your module should look like:

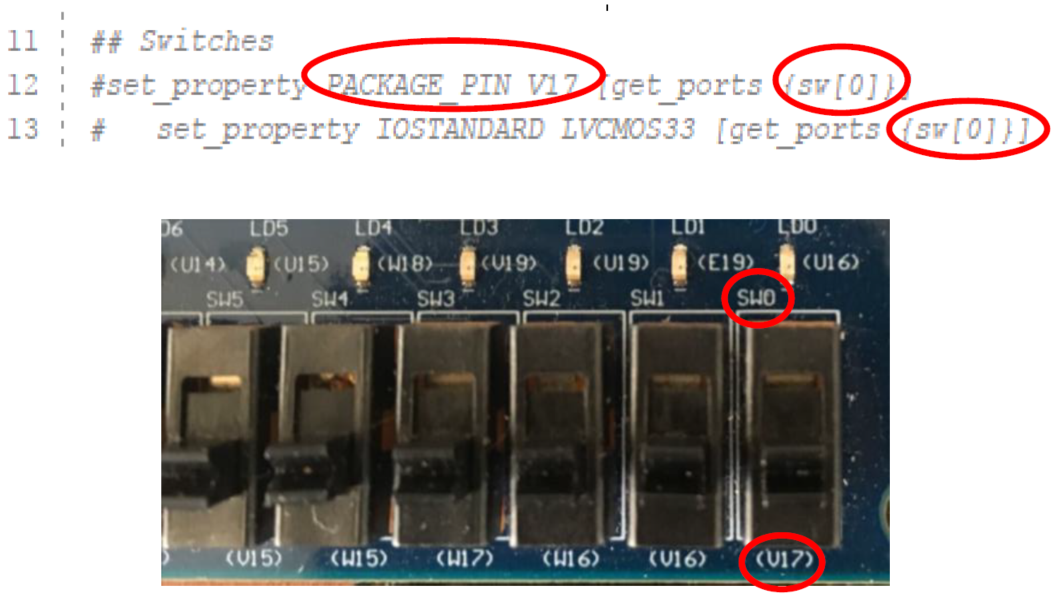
```
module gate(
input a,
input b,
output out
);
endmodule
```

3. Add the line `and g1(out,a,b);` to your module. Your resulting module should look like:

```
module gate(
input a,
input b,
output out
);
and g1(out,a,b);
endmodule
```

4. Create a testbench as shown in Figure 5.

5. Run the simulation and ensure that it works.



```
11  ## Switches
12  #set_property PACKAGE_PIN V17 [get_ports {sw[0]}]
13  #   set_property IOSTANDARD LVCMOS33 [get_ports {sw[0]}]
```

Figure 6: Constraints file for AND gate

6. Add the constraints file. Map `SW0` to port `a`, `SW1` to port `b`, and `LED0` to port `out` as shown in Figure 6. Don't forget to uncomment those lines by removing the hashtag in front of the line. After you edit the constraints file, lines 12-13 in Figure 6 should look like:
   `set_property PACKAGE_PIN V17 [get_ports a]`
   `set_property IOSTANDARD LVCMOS33 [get_ports a]`
   Be sure to change the names of the other ports.

7. Click "Run Synthesis" in the left sidebar and hit OK in the pop-up window. It should take 30s-1min.

8. Once synthesis finishes, click "Run Implementation" from the pop-up box and hit OK. It should take 1-2min.

9. Once implementation finishes, click "Generate Bitstream" from the pop-up box and hit OK. It should take 1-2min.

10. Connect the FPGA board to your computer with the USB cable. Click on "Open Hardware Manager" > "Open Target" > "Auto-connect".

11. Click "Program" > "Program Device". This should program the device.

12. Toggle SW1 and SW0 to make sure LED0 lights up in accordance to an AND gate.

3.2 Part B: Sprinkler Controller

In this part, you will implement the sprinkler controller system from Lab 2 on the board. You can reuse your code from Lab 2. Map each sprinkler valve to its own LED: `LED0` should map to `d0`, `LED1` should map to `d1`, and so on. Map each control to a switch: `SW2=a, SW1=b, SW0=c,` and `SW7=e`.

You will need to copy the uncommented sections of the constraints file for this part into your lab report.

3.3 Part C: Seven-Segment Display

In this part, you will be displaying a binary-coded-decimal number on the seven-segment display. A binary-coded-decimal (BCD) number represents a decimal number in binary.

1. Fill in the truth table in Table 1 to display the numbers 0-9 and letters A-F on the seven-segment display as specified in the Background section. The first row has been done for you.

Table 1: Seven-Segment Display Truth Table

Inputs (`SW[3:0]`)	Display (`an[0]`)	a	b	c	d	e	f	g
0000	0	0	0	0	0	0	0	1
0001	1							
0010	2							
0011	3							
0100	4							
0101	5							
0110	6							
0111	7							
1000	8							
1001	9							
1010	A							
1011	b							
1100	C							
1101	d							
1110	E							
1111	F							

2. Generate a minimized sum-of-products logic expression for each segment `a` through `g`. Show your work with K-maps.

   Again, these must be minimized and in sum-of-products form (i.e. if you accidentally write your truth table using active-high logic, you must redo the truth table and K-maps).

3. Use the logic expressions you generated to model the seven-seg behavior in Verilog with structural (gate-level) modeling. Use `SW[3:0]` to control `an[0]`, the right-most digit, on the board. The other digits should be turned off.

   Note that you will need to set both the segments and the anodes as outputs in your design file.

4. Write a testbench and run a simulation to ensure your model works. Take a screenshot of the waveform. Make sure your screenshot includes key features (e.g. the waveform for each individual bit).

5. Add the Basys3_Master.xdc constraints file from Canvas, and edit it to your module specifications. Ensure that it works; debug if it doesn't.

For this part, you will need to submit your truth table, minimized equations, a screenshot of your design file, a screenshot of your uncommented constraint file, and a screenshot of your simulation waveform.

4 Submission

There are two items to submit:

1. A .zip file containing your design/testbench files (.v), constraints file (.xdc), and bitstream (.bit) files for each component of the lab. If you need assistance locating these files in your directory, please visit a TA in office hours.
2. A PDF with the items listed on the Cover Sheet in Canvas.

Submit these two files as one single submission on Canvas. Be sure to schedule a checkout slot via the Canvas appointment scheduler.

4.1 Checkout Process

The checkout process involves a 10min one-on-one meeting with a TA the week after you submit your lab. You will schedule your checkout the week the lab is due via the Canvas appointment scheduler. The checkouts will be held in EER 0.716, the lab room. During the checkout, the TA will ask you to demonstrate the functionality of your code on the FPGA board, look at your source files, and ask you questions regarding the lab. The questions may range from syntax questions (e.g. "what does .a(a) mean?") to high-level design questions, such as "why did you choose these testcases?" You should be able to answer the questions in enough detail to show the TA that you understand the main concepts in the lab. Some sample checkout questions are provided below.

When it is time for you to check out, have your lab report (PDF file) and Vivado project open, and flash the board before the TA comes to meet with you. If you aren't prepared at your scheduled time, the TA will move on to the next person. If you miss your checkout time, you will need to re-schedule with the head TA.

4.2 Grading

Your grade on the lab will be based on a combination of your code's functionality on the board, your documentation (PDF file), and your oral responses to the TA during checkout. Late submissions will incur a 5% penalty per day up to a week, after which your grade will be a zero.

5 Sample Checkout Questions

Sample checkout questions may include, but are not limited to (in no particular order):

1. What does Vivado do in the synthesis and implementation phases of FPGA programming?
2. What is the difference between `wire` and `reg`? Give an example of the usage of each.
3. What is the role of the constraints file in the FPGA programming process?
4. What does structural Verilog mean? Why is it useful?
5. What inherent design "flaw" does the seven-segment display have on the board (aside from not being able to access the colon)? How do we work around this flaw?

6. Why does the synthesis fail if the constraints file variables do not match the variables you declare in your top module?
7. Aside from what connections are made between the board and the modules, what other information does the constraints file tell you?
8. Explain how your code works.
9. What is a Binary-Coded Decimal number? Why is this representation useful?
10. Describe how the seven-segment display works.
11. What do active-high and active-low mean? Give an example of an active-high device and an active-low device on the board.
12. How dothe anodes and segments on the seven-segment display work?

Note that all of the answers to these questions are given somewhere in the lab document or in previous labs; you do not need to do any external reading/searching for knowledge unless the lab document says so. For some of the questions, you may need to infer from or synthesize multiple pieces of information from the lab document to come up with the correct answer. As always, you are free to (and should) discuss these questions with a TA before checkout.