**Lab 3: Seven-Segment Display**

EE 316: Digital Logic Design

**Overview**

This lab is intended for you to become familiar with how the on-board seven-segment display works. By the end of this lab, you should be able to:

* Describe (at a high level) what happens in each stage of the Vivado workflow/board programming process.
* Describe how the seven-segment display works.
* Explain Verilog datatypes (wire vs reg) and when they are used.
* Use bit vectors and represent numbers to write Verilog.

Be sure to attend your assigned lab section each week. There is a fair amount of background information for this lab, and we will discuss the key concepts/ideas you need to know in-person. However, you are expected to know all of the material covered in the document for checkouts and/or exams. We will also field questions about the lab exercises during the lab sections.

Before starting on this lab, please review the “module instantiation” section of the background information of Lab 2.

**Background**

***FPGA Design Workflow***

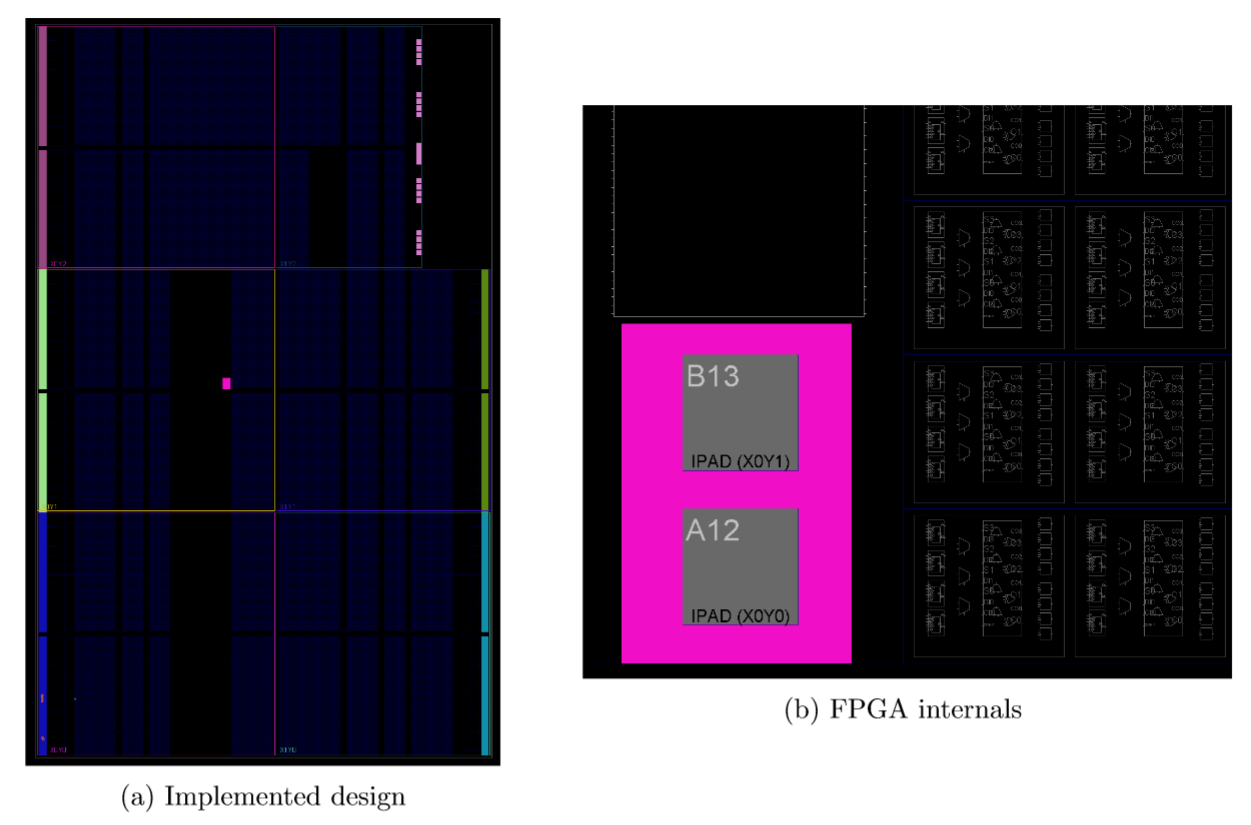
In the FPGA industry, there is a standard workflow that most FPGA designers (including Vivado) follow: simulation, synthesis, implementation, and programming. This section will break each of these processes down in more detail.

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.

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.

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 by opening it and zooming in. (The quality of the images below isn’t the best, but you can see it in Vivado.) 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.

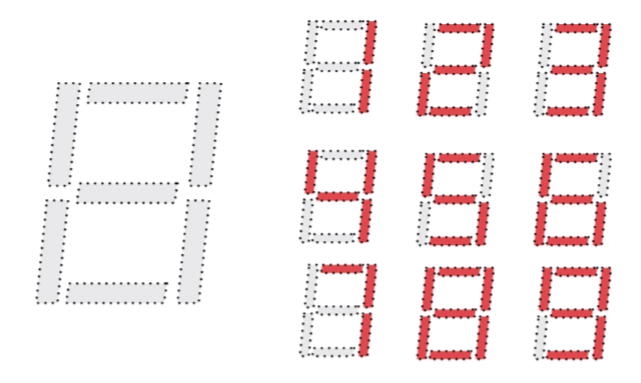


After the implementation stage, Vivado generates a bitstream that boils all the information of your implemented design into a series of 0s and 1s that will eventually be stored in the lookup tables. This file can then be transmitted (via USB, SPI, UART or your favorite serial communication protocol of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

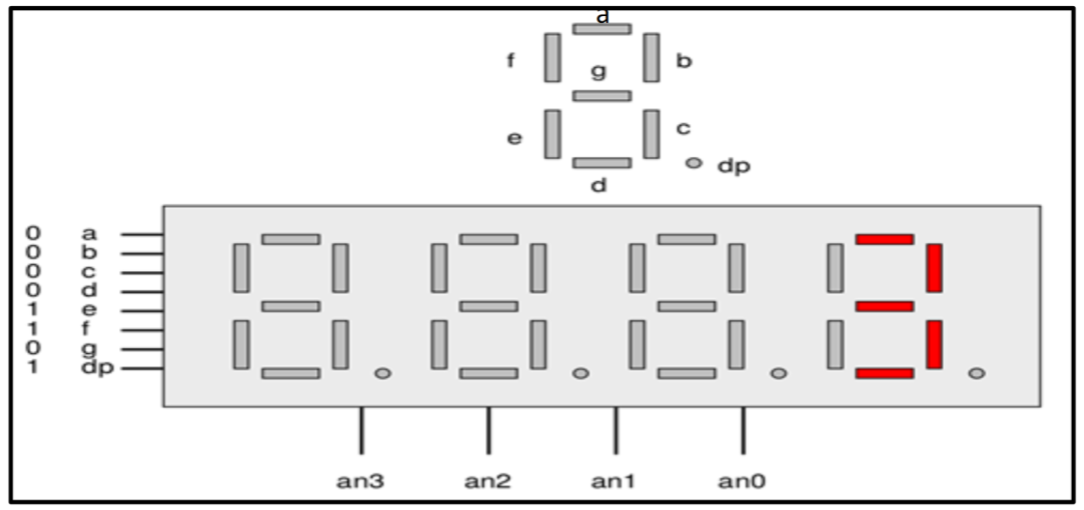
***Seven-Segment Display***

*Operation*

The four-digit seven-segment display on the Basys3 board allows us to display a variety of numbers, letters, and even words. We display characters on the seven-segment display by lighting up a pattern of LEDs inside the device (see figure below). This section will describe how the seven-segment display works and what you need to do to display a single digit/character.



The figure below 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.

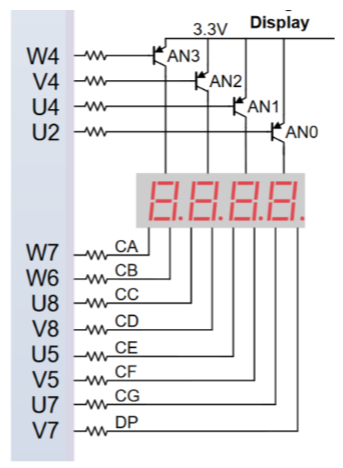


The segments and the anodes are both active-low devices, meaning that they must pass/be passed a '0' to be enabled or turned on. Therefore, if we wanted to display the number '3' in digit 0, we would set segments a-g and dp as 00001101 (to turn on segments a, b, c, d, and g) and set the anodes to 1110 (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.

*Internal 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.

The schematic below shows the internals of the seven-seg. 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.

***Verilog: 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. wire and reg are specific types of the two main groups of datatypes Verilog: net datatypes and variable datatypes.

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 and must always be *driven*, i.e. must always have a single defined value or expression. This characteristic places it in the group of datatypes called *net datatypes.* All net datatypes must be driven, and when the driver (a combinational circuit or other logic) changes, the net automatically updates. Inputs to modules must always be of datatype wire.

reg is what you consider a “normal variable”. regs are part of the *variable datatype* group, as they hold their value until they are assigned a new value. Unlike wires, regs do not have to be driven, can be driven by multiple values at different points in time, and/or can be initialized and then changed. 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 > Labs > Reference.

***Verilog: (Integer) Number Representation***

In Verilog, numbers are used both as series of bits (e.g. in case statements) and as numerical values. The basis of Verilog numerical representation is the *bit.* A bit in Verilog can have four distinct values: 0, 1, X, and Z. X stands for an unknown value, and Z stands for a high impedance. In general, we won’t use X and Z in this class, but you may see them in failed simulations: if you see an X, it means that you have an unknown value, and if you see a Z, it means you didn’t properly instantiate a module or connect ports correctly. We also won’t use non-integer numbers in this class, so the information below only applies to integer numbers.

The general syntax for number representation is:

<# bits>’<number system><value>.

The first number specifies the number of bits that will be used to store the value. For example, in the number 3’b1, three bits will be used to store the value 1, so the stored value would be 001. The default number of bits is 32 bits e.g. ‘b10 would be stored as a 32-bit value. The second field specifies the number system i.e. binary (b), octal (o), decimal (d), or hexadecimal (h). For example, 4’b1010 = 4’d10 = 4’hA. The default number system is decimal e.g. 302 is stored as 32’d302, aka a 32-bit decimal number of value 302. The number system field is case-insensitive, so you can either use b or B to denote binary numbers. The third field is the value. Hex digits in the value are case-insensitive. If a value is very long, you can use an underscore to delimit the digits without changing the value e.g. 16’hABCDEF = 16’hABC\_DEF. If the number of bits cannot represent the value, the most significant bits are truncated, and if the number of bits is larger than the value, the most significant bits are padded with zeroes. To specify negative numbers, place a minus sign in front of the number e.g. -4’d10. It will be stored in its twos-complement form.

***Verilog: Bit Vectors***

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 (output) for anodes

reg [7:0] seg = 0; //8-bit vector (of datatype reg) for segments

Below is the code for the decoder and multiplexer, rewritten using bit vectors. Note that you can also declare modules to have bit vectors as inputs and outputs, and you can access each bit (or range of bits) in a bit vector as well.

|  |  |
| --- | --- |
| module decoder(  input enable,  input [2:0] btn,  output [7:0] led);    assign led[0] = enable & ~btn[2] & ~btn[1] & ~btn[0];  assign led[1] = enable & ~btn[2] & ~btn[1] & btn[0];  //et cetera …  endmodule | module mux(  input [3:0] btn,  input [1:0] sel,  output led  );  reg led\_buf = 0;  assign led = led\_buf;  always @(\*) begin  case(sel) // Note no curly braces here  0: led\_buf = btn[0];  1: led\_buf = btn[1];  2: led\_buf = btn[2];  3: led\_buf = btn[3];  endcase  end  endmodule |

You can also select a range of bits in a bit vector or concatenate two bit vectors:

reg [3:0] foo = 4’b1010;

reg [3:0] bar = 4’b0101;

reg [7:0] foobar, barfoo;

reg [3:0] nine;

always @(\*) begin

foobar = {foo, bar}; // Equals 8’b10100101

barfoo = {bar, foo}; // Equals 8’b01011010

nine = {foo[3:2], bar[1:0]}; // Equals 4’b1001

end

These bit vector manipulations become extremely powerful when working with datapath components, as you will see later.

**Procedure**

In this lab, you will design a binary-coded decimal (BCD) converter and a 2-bit multiplier. Before starting this lab, please also review module instantiation from Lab 2.

***Part A: Binary-Coded Decimal Converter***

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. For example, the number 4’b1001 represents 9 in decimal.

Switches SW0-SW3 will represent the binary input to your system, and the seven-segment display, which has four anodes, seven segments, and a decimal point, will be the output from your system. For this lab, we will use the rightmost anode only.

1. Fill in the truth table below 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.

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

1. Generate minimized sum-of-products logic expressions 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).

1. Create a module named bcd.v in Vivado. (Make a new project and add a design file with the module name.) You should declare the module as follows:

module bcd(

input [3:0] sw,

output [3:0] an,

output [6:0] seg

output dp);

assign an = 4’b1110; // Enable rightmost anode, an0

assign dp = 1; // Disable decimal point since we aren’t using it in this lab

// Structural modeling

// Dataflow modeling

// Behavioral modeling

endmodule

1. Fill in the module above with a structural, dataflow, and behavioral model of the BCD. (This is somewhat tedious, but we won’t ask you to do this again.)
2. Open the elaborated design of the structural model and view the circuit schematic. Take a screenshot of the circuit.
3. Write a testbench (remember: this is a simulation source) and name it tb\_bcd.v. Use the testbench from Lab 2 to guide you in writing the testbench.
4. 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). You can choose which type of modeling to screenshot, but make sure to verify that all three work.
5. Add the Basys3\_Master.xdc constraints file from Canvas, rename it to bcd\_constrs.xdc and edit it to your module specifications. Ensure that it works; debug if it doesn't.

***Part B: 2x2-bit Multiplier***

In this part and the next part, you will create a 2-bit-by-2-bit multiplier. Similar to the BCD, your inputs to this system are the four switches SW[3:0], where SW[3:2] represents one multiplicand, and SW[1:0] represents the other multiplicand. Your output is the seven-segment display.

1. Fill in the truth table below for a 2-bit by 2-bit multiplier.

|  |  |  |  |
| --- | --- | --- | --- |
| SW[3:2] | SW[1:0] | Product (decimal) | Product (4-bit binary) |
| 00 | 00 | 0 | 0000 |
| 00 | 01 |  |  |
| 00 | 10 |  |  |
| 00 | 11 |  |  |
| 01 | 00 |  |  |
| 01 | 01 |  |  |
| 01 | 10 |  |  |
| 01 | 11 |  |  |
| 10 | 00 |  |  |
| 10 | 01 |  |  |
| 10 | 10 |  |  |
| 10 | 11 |  |  |
| 11 | 00 |  |  |
| 11 | 01 |  |  |
| 11 | 10 |  |  |
| 11 | 11 |  |  |

1. From your truth table above, generate sum-of-products logic expressions for each bit of the 4-bit binary number that represents the product of SW3SW2 and SW1SW0 (minimization not required but strongly encouraged).
2. Create a new module in Vivado and name it mult.v. Your top module declaration should be the same as that of the BCD in Part A; **do not modify the module declaration**.
3. From your sum-of-products logic equations, use a dataflow model to implement your system. In your model, you must instantiate and use the bcd.v module (see Lab 2 for more info on how to do this).
4. Open the elaborated design of the model and view the circuit schematic. Take a screenshot of the circuit.
5. Write a testbench (remember: this is a simulation source) and name it tb\_mult.v. Use the testbench from Lab 2 to guide you in writing the testbench.
6. 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).
7. Note that the constraints file is the same as Part A. You can reuse the same constraints file to program the board. Ensure that it works; debug if it doesn’t.

**Submission**

When you submit labs, you will need to submit two things: your zipped source code and a PDF answering questions given in the lab document. Below are the instructions for creating your zip file. Your PDF should be submitted separately, NOT included inside your zip file.

1. Create the zip file with your design files, testbenches, constraints files, and bitstreams. You should have a total of 2 design files (bcd.v, mult.v), 2 testbench files (tb\_bcd.v, tb\_mult.v), 1 constraint file (bcd\_constrs.xdc), and 2 bitstreams (bcd.bit, mult.bit), for a total of seven files. You can choose the type of modeling for your bitstreams.
2. Create your PDF. Include the following, in order:
   1. A cover page with your name, EID, section unique number, and professor.
   2. Completed truth table and seven K-maps for the BCD.
   3. BCD circuit schematic screenshot.
   4. BCD simulation waveform screenshot.
   5. Completed truth table and logic equations for mult.v.
   6. mult.v module schematic screenshot.
   7. mult.v module simulation waveform screenshot.
3. Submit your zip and PDF as one submission on Canvas.

**Checkout Process/Grading**

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 PDF file and Vivado project open, and flash the board before the TA comes to meet with you. **You MUST check out a Basys3 board from the lab checkout desk (EER 1.834) BEFORE you come to the lab.** 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.

Your grade will be determined during your checkout by the TA who is checking you out. The rubric is given below and posted on Canvas. Your TA will use the Canvas rubric to assess you based on a combination of your code’s functionality, your documentation (PDF file), and your oral responses to the TA during checkout. You should know your grade when you walk out of your checkout. Late submissions will incur a 5% penalty per day up to a week, after which your grade will be a zero. In addition, failure to check out within a week of submitting your lab will also result in a zero.

**Sample Checkout Questions**

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 do the anodes and segments on the seven-segment display work?
13. What is the default size of an integer number in Verilog?
14. A common error when using bit vectors as inputs and outputs is not ensuring that the size of bit vectors is the same in a module instantiation. If a byte-wide bit vector is passed into a module whose input port is declared a nibble-wide bit vector, what is actually passed into the module?

Note that because digital logic design and Verilog (and all pursuits of life in general) builds on itself, content from previous labs is also fair game for checkout questions. We will usually ask them in the context of the current lab, however, so you won’t need to know e.g. how a decoder works but you do need to know how module instantiation works (cf. Lab 2).

**Rubric**

Below is the grading rubric for this lab. Please review it before submitting your lab to ensure you meet all the requirements.

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
|  | Missing | Attempted | Half-correct | Almost correct | Fully correct |
| ***Assessed from zip file submitted on Canvas*** | | | | | |
| BCD: Models | 0 | 2.5 | 5 | 7.5 | 10 |
| Multiplier model | 0 | 2.5 | 5 | 7.5 | 10 |
| ***Assessed from deliverables PDF*** | | | | | |
| Truth tables | 0 | 2.5 | 5 | 7.5 | 10 |
| Logic equations/K-maps | 0 | 2.5 | 5 | 7.5 | 10 |
| Schematics | 0 | 2.5 | 5 | 7.5 | 10 |
| Waveforms | 0 | 2.5 | 5 | 7.5 | 10 |
| ***Assessed from in-person checkout*** | | | | | |
|  | No-show | Needs work | Fair | Very good | Excellent |
| Board functionality: BCD | 0 | 5 | 8 | 9 | 10 |
| Board functionality: Multiplier | 0 | 5 | 8 | 9 | 10 |
| Checkout questions | 0 | 5 | 8 | 9 | 10 |
| On-time submission and checkout | 0 | 5 | 8 | 9 | 10 |