**Lab 4: Sequential Logic Design**

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 and implement a clock divider in Verilog.
* Implement finite state machines in Verilog.
* Explain the utility of, and implement, time multiplexing.
* Explain the difference between blocking and non-blocking.

Be sure to attend your assigned lab section each week; 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.

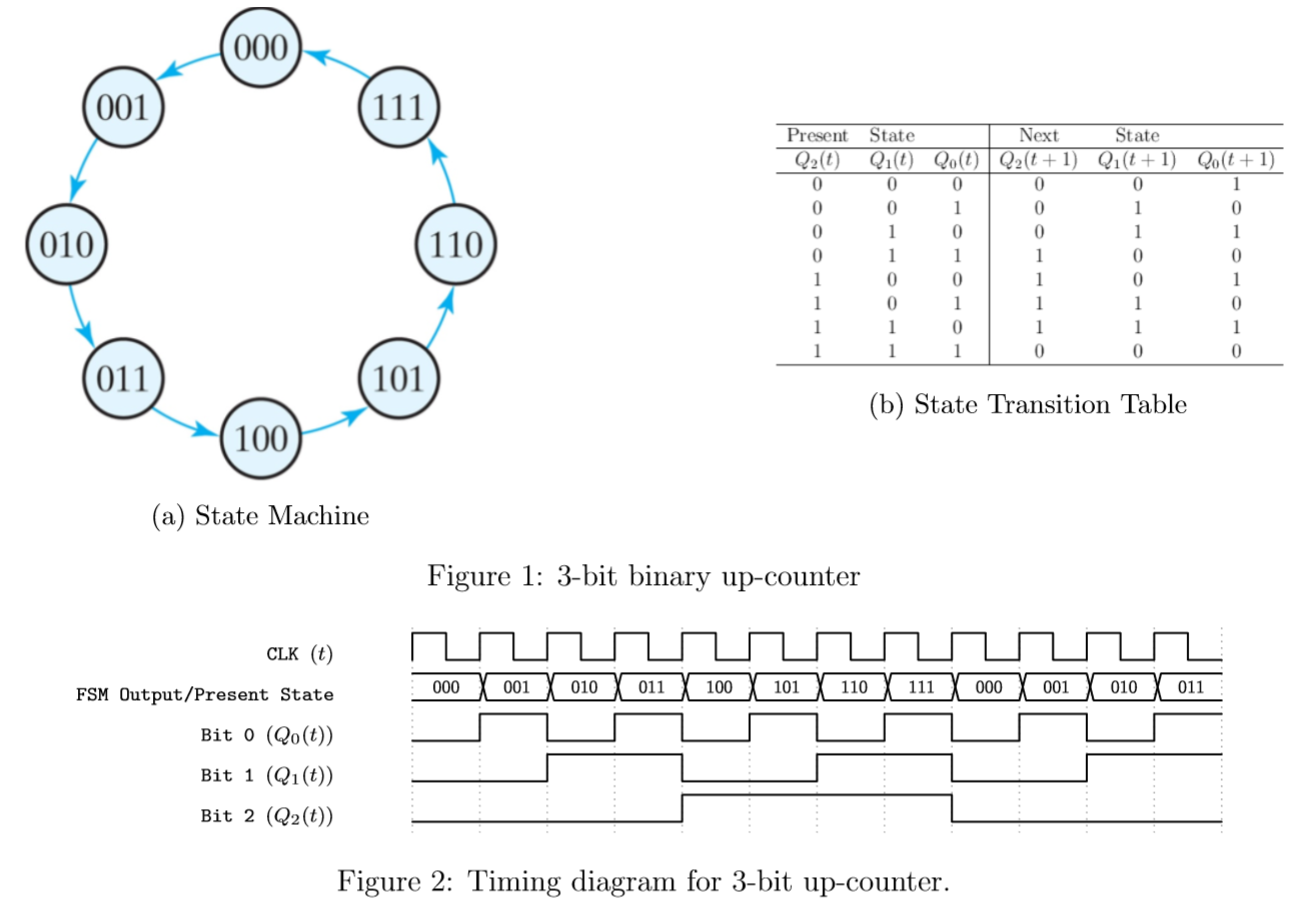
*NB:* In the past, many students have found Lab 4 to be significantly more challenging than the first three labs. We recommend that you at least start on a portion of it (ideally complete Parts A and B) before Spring Break.

**Background**

***Clock Division***

A clock is a periodic signal that is used as a timing mechanism for sequential circuits such as FSMs. In an FSM, each state lasts one clock cycle. Because an FSM changes state in response to the clock, FSMs are called synchronous. The Basys3 board has a 100MHz internal clock that we can use; however, we usually want to run FSMs at much slower speeds. Clock division is the process by which we generate slower clocks from fast clocks, and the clocks we make are called clock dividers. Clock dividers are usually made with a binary up-counter that increments at the speed of the main clock. By masking for certain bits in the up-counter, we can get periodic signals that can serve as new, slower clocks.

An FSM and state transition table for a 3-bit binary up-counter are shown in Figure 1a. In an up-counter, the states are usually named with the outputs, so state 0 outputs 000, state 1 outputs 001, etc. so that the present state number is what the system outputs. Unconditional transitions connect state 0 to state 1, 1 to 2, 2 to 3, etc. so the counter runs indefinitely.



If we draw a timing diagram showing when each bit in the counter is high in relation to the FSM clock (Figure 2), we can see that each bit in the counter actually transitions periodically - but at a slower rate than the FSM clock. Bit 0 transitions on every rising edge of the main clock; bit 1 transitions on every other rising edge; and bit 2 transitions on every 4 rising edges. Therefore, the period of bit 0 is 2 clock cycles, and the frequency of the signal on bit 0 is 1/2 the frequency of the original clock. Similarly, the period of bit 1 is 4 clock cycles, which means its frequency is 1/4 the original clock. The period of bit 2 is 8 clock cycles, which means its frequency is 1/8 of the original clock. These clock dividers are also called “divide-by-2”, “divide-by-4”, and “divide-by-8” clocks, respectively. In this way, we can generate clocks that divide the original clock frequency by a power of 2.

“Divide-by-” clock dividers are easy to generate with bit-accessing in Verilog, as shown in the code snippets below. Note that the sensitivity list of the always block (inside parentheses) now contains “posedge clk” instead of an asterisk. This indicates that the always block will now only run on the positive (rising) edge of the clock; therefore, the counter will only increment on the rising edges of the input clock.

wire div2, div4, div8;

reg [2:0] count = 0; //Declare and initialize up-counter register

assign div2 = count[0]; //Divide-by-2 clock (50MHz on Basys3)

assign div4 = count[1]; //Divide-by-4 clock (25MHz on Basys3)

assign div8 = count[2]; //Divide-by-8 clock (12.5MHz on Basys3)

always @(posedge clk) count = count + 1;

We can also divide the clock by a non-power of 2 by adjusting the maximum value of the up-counter. Suppose we wanted to create a divide-by-3 clock. While there are several ways to go about this, one simple way to do it is to adjust the maximum value of the counter. In Verilog:

wire div3;

reg [1:0] count = 0; //Declare and initialize up-counter register

assign div3 = count[1]; //Divide-by-3 clock (33.3MHz on Basys3)

always @(posedge clk) begin

if(count == 2) count = 0; //Modify max value of counter (set to 2)

else count = count + 1;

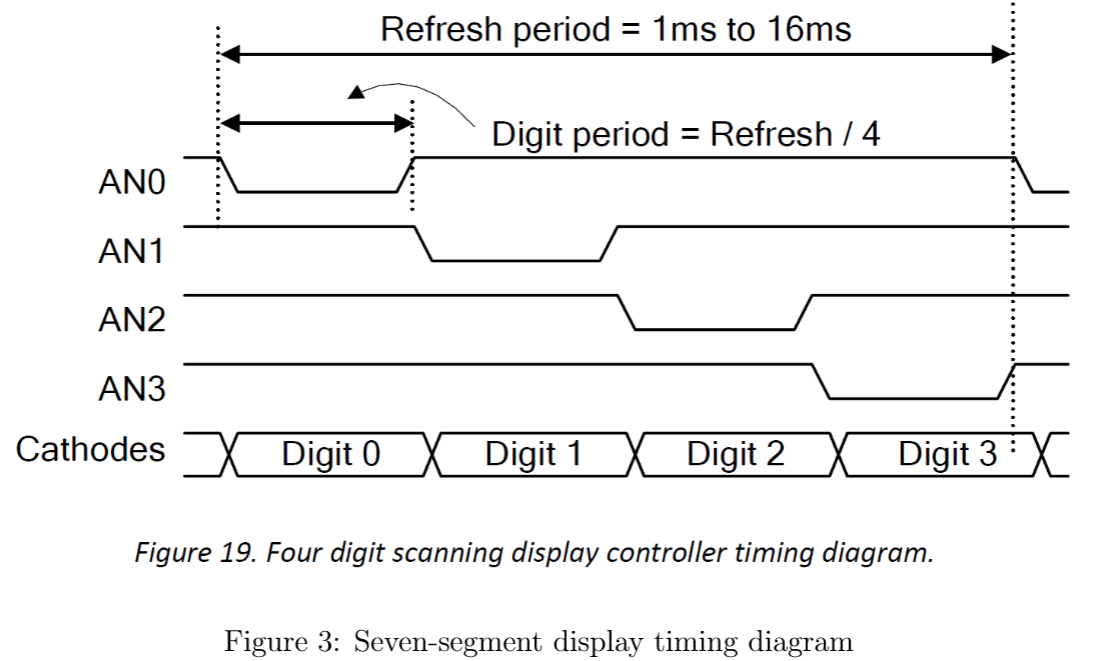
end

Note that this solution only works if we do not care about the duty cycle of our divide-by-3 clock (how long the clock is high in relation to its total period). If we wanted a divide-by-3 clock with a 50% duty cycle (a very real and interesting problem in digital logic design), we would need to use some logic to generate the signal. (We would have to feed the output of the up-counter into a falling-edge flip-flop and OR the output of the up-counter and the falling-edge flip-flop to generate the 50% duty cycle.)

***Seven-Segment Display: Time Multiplexing***

One key design problem raised in Lab 3 with the seven-segment display is that it cannot display four different numbers at a time because all the segments in the same position across the display are effectively shorted together when they are enabled. To solve this problem, instead of turning all four digits on at the same time, we can cycle through the digits, turning each one on individually. This design is called time-multiplexing – we use the clock to cycle through the digits, showing one at a time. If we speed up the clock to about 2-5kHz, the display will appear that all four digits are on simultaneously.

To implement this on the board, consider the timing diagram on page 17 of the Basys3 reference manual (on Canvas, reproduced in Figure 3). The anodes are enabled one at a time (remember: they are active-low devices), and the segments (cathodes) corresponding to the desired digit are displayed in the same window of time as the anode. This system behavior can be captured in an FSM, as you will do in this lab.



***Verilog: Blocking vs Non-blocking***

Since Verilog is a hardware description language (HDL), timing plays a key role in assignment. Consider the following always block:

always @(posedge clk) begin

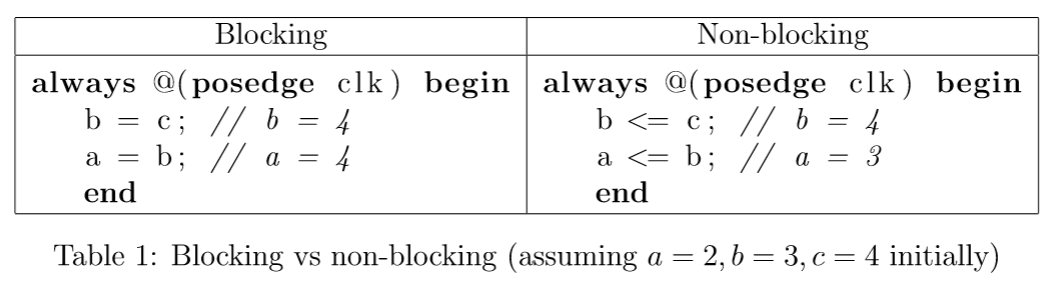
b = c; //a, b, c are regs

a = b;

end

If we didn't know any better, we would logically say that since a = b implies that a is wired to b and similarly for b=c, if the always block runs, a=c. This is problematic because the value of b is always destroyed, rendering the variable meaningless, and a will always hold the same value as c unless it is updated elsewhere.

To fix this, Verilog has defined two kinds of assignment: blocking and non-blocking (see Table 1). Blocking (denoted by `=') executes the statements sequentially, *blocking* execution of the next statement until the current statement completes. In the above example, a = 2, b = 3, c = 4, then after execution, a = 4, b = 4, c = 4. Non-blocking (denoted by `<=') executes the statements simultaneously: if in the above example, a = 2, b = 3, c = 4, then after execution, a = 3, b = 4, c = 4. Note that a was assigned the old value of b instead of the new value because the assignment of c to b happens at the same time: when a=b is executed, b holds the value 3 until the end of the assignment. (Another way to think about it is that these are non-ideal flip-flops in series.)



**Procedure**

In this lab, you will design a time multiplexing state machine to operate the seven-segment display. Parts A and B will have you design smaller modules that you will modify in the final system, and Part C is the final system design.

***Part A: Clock Divider Bank***

In this part, you will create 16 clock dividers from the 100MHz Basys3 clock and display the new clocks by flashing LEDs on the board. The frequencies at which each LED should flash are listed in Table 2. For LEDs 0-11, you do not have to get the exact frequency; an error of 1% is allowed (hint: start dividing 100MHz by powers of 2). For LEDs 12-15, you need to generate the exact frequency listed in the table; you should be able to explain how your design creates the exact frequency. Note: frequencies have changed to make sure Vivado doesn’t crash! Your module declaration should look something like:

module clks( // Name your module clks.v

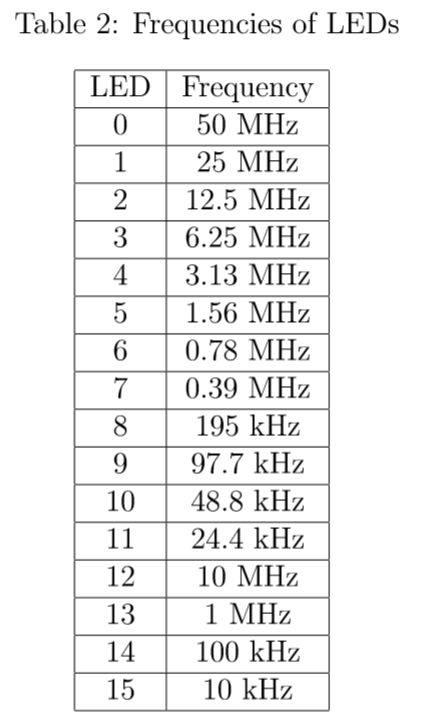
input clk,

output [15:0] led

);

//Your code here

endmodule



Create a testbench named tb\_clks.v for the clock divider bank. Your testbench should look something like this:

module tb\_clks;

reg clk = 0;

wire [15:0] led;

clks c(clk, led);

always #5 clk = ~clk;

endmodule

When you run the simulation, it will run for 1 us (microsecond) by default. To see all the clocks toggle, we want to run the simulation for at least 200 us since the slowest clock has a period of 100 us. Once the simulation finishes and you see the waveform, go to the Tcl console and type two commands:

restart

run 200 us

This should reset and run the simulation for 200 us. Depending on your processor speed, the simulation may take a while to complete. On the Linux servers, it should take about 2-5 seconds to complete. When you take a screenshot of your waveform for your PDF, make sure that ALL VALUES, especially those of bit vectors, are visible in the waveform. You may need to expand the bit vector in the waveform window to see each individual bit, and you may need multiple screenshots to capture all the features of the waveform in the 200 us time window.

Once you verify that the simulation works, create the constraints file that you would use to program the board. Be sure to uncomment the clock signal and all the LEDs in the constraints file (refer to the example project “onoff”). Note that there are no “interactive” inputs (i.e. switches, buttons): the LEDs should flash continuously. Name your constraints file clks\_constrs.xdc. Delete all unnecessary lines (i.e. lines that would remain commented); this will help with readability when grading. Generate the bitstream to make sure there are no errors.

***Part B: Rising-Edge Detector***

In this lab, you will build a rising-edge detector. A rising-edge detector is a finite state machine that detects when a signal goes from low to high or 0 to 1. Your rising-edge detector will be tied to SW0. When SW0 transitions from low to high, LED0 should flash (turn on then off) for 50 ns, independent of the switch behavior.

1. Download the example project labeled “onoff” from Canvas.
2. Read through the project and run it on the board. The example project models how FSMs are generally modeled in Verilog and will be a resource for you in this section.
3. Based on the problem description, draw a Moore FSM state graph that models the rising-edge detector.
4. Calculate the frequency that your FSM should operate at and the number of board clock cycles that must elapse before your slow clock toggles.
5. Create a module named clkdiv.v that will be your clock divider. Model it off of the provided example code. Do not use the clks.v module from part A.
6. Use behavioral modeling and the example code to model the rising-edge detector in Verilog. Name your top module edge\_detector.v.
7. Open the schematic in Vivado and take a screenshot of it.
8. Create a testbench named tb\_edge.v and simulate your design. Be sure to screenshot your waveform. Make sure that ALL VALUES, especially those of bit vectors, are visible in the waveform. Note that the total simulation time must be long enough to showcase the functionality of the entire FSM, but do not exceed more than 200 us.
9. Add your constraints file and edit it to the module specifications. Name your constraints edge\_constrs.xdc. Delete all unnecessary lines (i.e. lines that would remain commented); this will help with readability when grading.
10. Generate your bitstream as you would normally to ensure that it is theoretically possible to program the board with your code.

***Part C: Seven-Segment Display Controller***

For this part of the lab, you will design a seven-segment display controller, as described in the Background section. Be sure to review time-multiplexing before beginning this part.

Your seven-segment display controller should take in four four-bit numbers, represented by SW[15:12], SW[11:8], SW[7:4], SW[3:0]. The hex number represented by SW[15:12] should be displayed on AN[3], SW[11:8] on AN[2], and so forth. You should also have a reset button to re-initialize the controller on btnC. Make sure you enable the board clock in your constraints file.

1. Draw a Moore FSM that models this system. You should have four states that cycle unconditionally when reset is not asserted, with each state outputting an anode value and a switch combination. For example, your initial state will have an = 4’1110 and SW[3:0] as outputs. When reset is asserted, you should send your FSM to the initial state.
2. Create a module named sevenseg.v that will serve as the top module. Your module declaration should look something like this:

module sevenseg(

input clk,

input [15:0] sw,

input reset,

output reg [3:0] an,

output reg [6:0] seg

);

endmodule

1. Because you will be converting each of the four-input switch combos into hex, you will need four instances of the BCD you designed in Lab 3. Add the bcd.v file from Lab 3 and instantiate four instances of it in your top module, one for each set of four switches. You will need to modify the file slightly and add wires/regs to hold its output values.
2. Instantiate a clock divider module and set it to run at 40 MHz. Provide calculations for the number of board clock cycles that need to elapse before toggling your clock divider output.
3. Using your FSM from above and the example code provided with the lab, write the Verilog for the state machine in your top module.
4. Open the schematic in Vivado and take a screenshot of it.
5. Create a testbench named tb\_sevenseg.v to simulate your design. Screenshot your waveform. Make sure that ALL VALUES, especially those of bit vectors, are visible in the waveform. Note that the total simulation time must be long enough to showcase the functionality of your FSM, but do not exceed more than 200 us.
6. Add and edit the constraints file to meet the specifications of the module. Name it sevenseg\_constrs.xdc. Delete all unnecessary lines (i.e. lines that would remain commented); this will help with readability when grading.
7. Generate your bitstream as you would normally to ensure that it is theoretically possible to program the board with your code.

**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 3 design files (clks.v, edge\_detector.v,sevenseg.v), 3 testbench files (tb\_clks.v, tb\_edge.v, tb\_sevenseg.v), 3 constraint files (clks\_constrs.xdc, edge\_constrs.xdc, sevenseg\_constrs.xdc), and 3 bitstreams (clks.bit, edge\_detector.bit,sevenseg.bit), for a total of eleven files.
2. Create your PDF. Include the following, in order:
   1. A cover page with your name, EID, section unique number, and professor.
   2. Clock divider bank schematic.
   3. Clock divider bank waveform(s).
   4. Rising-edge detector Moore FSM.
   5. Rising edge detector frequency calculations.
   6. Rising edge detector schematic.
   7. Rising edge detector waveform screenshot.
   8. Seven-segment display Moore FSM.
   9. Seven-segment display frequency calculations.
   10. Seven-segment display schematic.
   11. Seven-segment display waveform screenshot.
   12. Answers to the 8 checkout questions listed below.
3. Submit your zip and PDF as one submission on Canvas.

**Checkouts/Grading**

In lieu of in-person checkouts, you will now be required to answer all of the checkout questions (listed below). We will evaluate your code’s functionality by looking at your design source files, testbenches, and constraints files according to the rubric below. The rubric has been slightly adjusted to accommodate the lack of in-person checkouts.

**Checkout Questions**

In lieu of in-person checkouts, please answer the following questions in your PDF. Try to be as concise as possible; full sentences are not required, as long as you communicate your ideas clearly.

1. In less than 20 words, how would you design a divide-by-5 clock? Draw a timing diagram of your divide-by-5 clock.
2. What is a sensitivity list? Why is it important?
3. Define blocking and non-blocking. When is each used?
4. Why is it impossible to get a clock of exactly 10Hz by using bit-masking in Verilog?
5. Define synchronous and asynchronous. Is an FSM synchronous or asynchronous?
6. How do we get around the “problem” of not being able to display four different numbers on the seven-seg at the same time?
7. What do the keywords posedge and negedge mean, and where in your code do you use them?
8. How would you change your edge detector state machine to create an edge detector that senses both rising and falling edges?

**Rubric**

Below is the grading rubric for this lab. Please review it before submitting your lab to ensure you meet all the requirements. We will strive to stick to the rubric as much as we can; however, we reserve the right to assign point values that are not stated in the rubric (e.g. 4 points or 7 points) based on your submission. If we deviate from the rubric, we will provide an explanation in the rubric for our reasoning.

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
|  | Missing | Attempted | Half-correct | Almost correct | Fully correct |
| ***Assessed from zip file submitted on Canvas*** | | | | | |
| Clock dividers | 0 | 5 | 8 | 9 | 10 |
| Rising-edge detector | 0 | 5 | 8 | 9 | 10 |
| Seven-segment display | 0 | 5 | 8 | 9 | 10 |
| Testbenches and constraints | 0 | 5 | 8 | 9 | 10 |
| ***Assessed from deliverables PDF*** | | | | | |
| FSMs | 0 | 5 | 8 | 9 | 10 |
| Frequency calculations | 0 | 5 | 8 | 9 | 10 |
| Schematics | 0 | 5 | 8 | 9 | 10 |
| Waveforms/Pictures | 0 | 5 | 8 | 9 | 10 |
| Checkout questions | 0 | 5 | 8 | 9 | 10 |
| On-time submission | 0 | 5 | 8 | 9 | 10 |