

ECE437: Sensors and Instrumentation

Lab 3: Finite State Machines and Vivado Simulator Lab

Introduction

One of the main skills you will develop in the class is the ability to write communication interfaces with various sensors and transport this data to the PC for further analysis. To be able to communicate and acquire data from sensors on the custom PCB, you will develop several different finite state machines (FSMs) in Verilog programming language. The complexity of these FSMs will progressively become more complex and will require diligent debugging of the code you will develop. Synthetizing, implementing and generating a bit-file is a lengthy process that can take anywhere from a minute to a half an hour to complete. Even simple examples, such as the counter example in the first lab, can take up to 2 minutes to generate a bit-file! Verifying and debugging the functionality of your code by going through all these steps, i.e. synthetizing, implementing and generating a bit-file, can be a lengthy process and we would like to avoid it as much as possible during the debugging phase to increase our productivity.

In this lab, you will learn how to simulate and debug your Verilog code **without** synthetizing, implementing and generating a bit-file. Vivado provides a simulator that enables you to examine the functionality of your code in matter of seconds. Remember, you don't need to synthetize your code to simulate your code. You can make changes in your Verilog code, immediately run the simulator and observe your results in Vivado. This entire process can take few seconds and will enable you to rapidly change your code, debug it and make progress with your labs and projects.

In this lab, we will also introduce a simple FSM and simulate its functionality in Vivado. The example FSM will provide a good starting point for your future FSM implementations. **You will first need to go diligently through the tutorial described in this document.** There are no shortcuts around this process and if you skip any part of this tutorial, you will encounter problems in the second part of the lab. Once you complete the tutorial, demonstrate to the TA that you have completed this task and answer the necessary questions at the end of each section. The second part of the lab, focuses on implementing an FSM. You should extensively use the Vivado simulator to debug your FSM before generating the final bit file.

Relevant Documents for this Lab

Required reading material for this lab:

1. Verilog example code on the course website: lab3_example.v
2. Python example code on the course website: lab3_TestBench.v

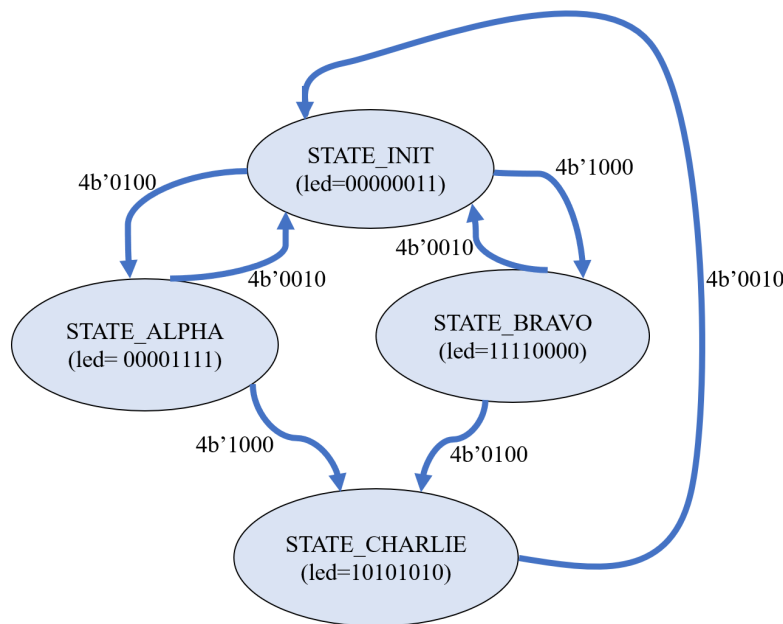
Prelab Questions (20 points):

1. What is the difference between **locaparam** and **parameter** in Verilog? When would you use either one of these declarations? (5 pts)

2. What is the difference between **wires** and **registers** in Verilog? List several syntax rules for both wires and registers? (5 pts)
3. When designing a finite state machine module, when would you use a **wire** and when would you use **register**? (5 pts)
4. What is the difference between blocking and non-blocking assignments in Verilog? (5 pts)

Finite State Machine Example

We will first start by describing the FSM we like to implement in Verilog. Here is a block diagram of the FSM.



The finite state machine has four different states, STATE_INIT, STATE_ALPHA, STATE_BRAVO and STATE_CHARLIE. The LEDs are assigned to different values for each one of the states. The FSM starts in the STATE_INIT state, where the LED output are set to 8'b00000011. The FSM transitions to different states depending on the 4-bit input from the external buttons. For example, if the FSM is in STATE_INIT state and the button input vector is 4b'0100, i.e. the 3rd button (switch SW2 on the board) is pressed, the FSM will transition to STATE_ALPHA. The LEDs in the new state will be set to 8'b00001111. You can follow the rest of the FSM from the block diagram.

The Verilog code for this FSM is provided on the course website under the file name **lab3_example.v**. Download the example code on your computer. Start a new project in Vivado following the same steps as outlined in lab 1. You should name the new project **lab3_example**. Make sure you select the correct FPGA when you declare your new project. Next add both the Verilog code for the FSM, i.e. **lab3_example.v** file, and the constraints file, i.e. **xem7310_v1.xdc** file. Open the Verilog code for **lab3_example.v** by double clicking on the file name in the Vivado IDE. You should see the following code displayed on the screen:

```

1  `timescale 1ns / 1ps
2  module lab3_example(
3      input [3:0] button,
4      output [7:0] led,
5      input sys_clk,
6      input sys_clkp
7  );
8
9      reg [1:0] state = 0;
10     reg [7:0] led_register = 0;
11     reg [3:0] button_reg;
12
13     wire clk;
14     IBUFGDS osc_clk(
15         .O(clk),
16         .I(sys_clkp),
17         .IB(sys_clk)
18     );
19
20     assign led = ~led_register; //map led wire to led_register
21     localparam STATE_INIT      = 2'd0;
22     localparam STATE_ALPHA     = 2'd1;
23     localparam STATE_BRAVO     = 2'd2;
24     localparam STATE_CHARLIE   = 2'd3;
25
26     always @(posedge clk)
27     begin
28         button_reg = ~button;
29         if (button_reg [3:0] == 4'b1110) state <= STATE_INIT;
30         else
31         begin
32             case (state)
33             STATE_INIT : begin
34                 if (button_reg == (4'b0100)) state <= STATE_ALPHA;
35                 else if (button_reg == 4'b1000) state <= STATE_BRAVO;
36                 else led_register <= 8'b00000011;
37             end
38
39             STATE_ALPHA : begin
40                 if (button_reg == 4'b1000) state <= STATE_CHARLIE;
41                 else if (button_reg == 4'b0010) state <= STATE_INIT;
42                 else led_register <= 8'b00001111;
43             end
44
45             STATE_BRAVO : begin
46                 if (button_reg == 4'b0100) state <= STATE_CHARLIE;
47                 else if (button_reg == 4'b0010) state <= STATE_INIT;
48                 else led_register <= 8'b11110000;
49             end
50
51             STATE_CHARLIE : begin
52                 if (button_reg == 4'b0010) state <= STATE_INIT;
53                 else led_register <= 8'b10101010;
54             end
55
56             default: state <= STATE_INIT;
57
58         endcase
59     end
60 end
61 endmodule
62

```

Let's look over the example code. The variables declared in the module are the same as in lab 1 and they provide input/output interface between the FPGA and the outside world, namely with LEDs, buttons and clock signals. Next, we declare three registers that are important for implementing the FSM. Because the buttons are active low, we have declared a register named *button_reg* which is equal to the complement of the button signal. This way, when we press a button, we will register this signal as high and if the button is not pressed the signal is low in *button_reg* variable. The rest of the code will examine the value in *button_reg* variable.

Since there are 4 states in the FSM, we have declared a 2-bit variable named *state*. This variable is initialized to zero and it will keep track of the different states in the FSM. Next, we define various numbers to the four states. For example, *STATE_INIT* is assigned to 0, *STATE_ALPHA* is set to 1 and so on. These are randomly assigned, and you can assign different numbers without any effects on the FSM accuracy. Since we are using a 2-bit register to keep track of the different states, the digital value assigned to each state cannot be greater than the number 3 and there cannot be duplicate numbers for the different states.

The core of the FSM is the case statement. Since we are using localparam to define what number is associated with each state, the case statement code is very easy to read. If the FSM state is set to *STATE_INIT*, then we examine the buttons state to determine if the next state will be either *STATE_ALPHA* or *STATE_BRAVO*. At the same time, we set the output of the LED to 8'b00000011. If we press the third button, i.e. button vector is 4b'0100, then the next state is *STATE_ALPHA*. If we press the fourth button, i.e. button vector is 4b'1000, then the next state is *STATE_BRAVO*.

The rest of the FSM state machine can be easily mapped to the block diagram presented above. A good practice is to have a *default* statement in the case code. This statement gets executed if the FSM is in a state that is not described by any of the case statements. Although this will never happen in this code, it is a good practice to include it when you write more complex FSM. Sometimes your FSM can end up in an unknown state and having a default state can catch these erroneous states. It will be up to you how you handle those cases which will help you debug your code.

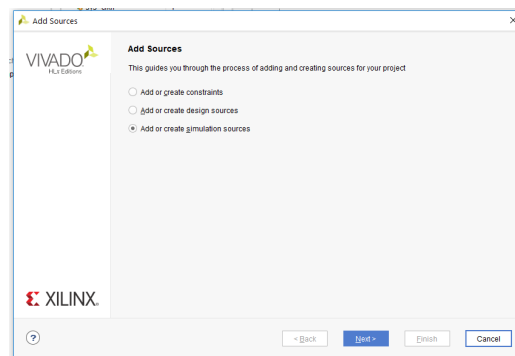
Next, you will synthesize, implement and generate a bit-file for your Verilog code. Open FrontPanel software on the computer and download the bit file from this code to the FPGA. Carefully evaluate the LEDs state as you press different button to validate the correctness of this FSM.

Simulating Verilog Code in Vivado

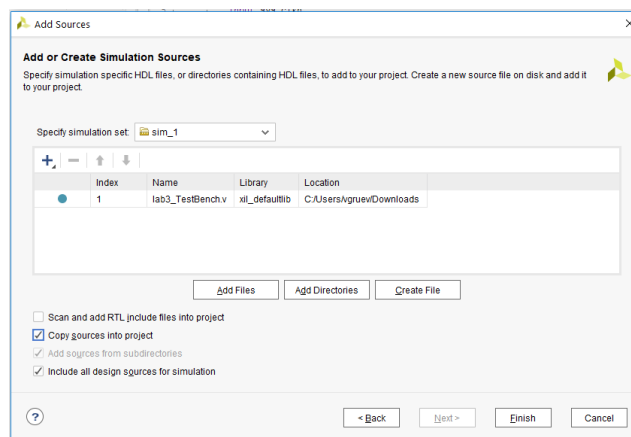
The next step is to simulate the behavior (i.e. the functionality) of the Verilog code from the first part of this lab. Since the FSM requires clock and button inputs to determine the next state, we will provide a test bench for these signals. This will enable us to examine the correctness of the FSM code by stepping through different states as different inputs are presented.

The first thing we will do is add the test bench file in our Vivado project. The test bench file will provide various inputs at different time periods to test your code. You will need to download the test bench file from the course website. The name of the file is: **lab3_TestBench.m**. Here is how to add the test bench file:

1. Click on **Add Sources** under the *Project Manager* tasks of the *Flow Navigator* pane.
2. Select the *Add or Create Simulation Sources* option and click **Next**. Here is an example of this window.

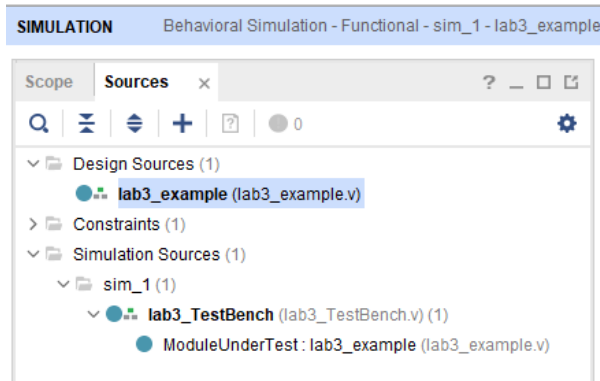


3. Click on *Add Files* button in the next window. Browse to the directory where you have downloaded the **lab3_TestBench.v** file. Select the **Copy sources into project**. Once you select this option, your test bench file will be copied into a directory structure that Vivado likes to organize its files. This way, you don't have to worry where exactly the test bench file should be placed.
4. The *Add Sources* should look like this. In our case, the test bench file was in the `c:/users/vgruev/downloads` directory.



5. Click on the **Finish** button.

6. Select the *Sources* tab and expand the *Simulation Sources* group.
7. The **lab3_TestBench.v** file is added under the *Simulation Sources* group, and **lab3_example.v** is automatically placed in its hierarchy. This is how your Source panel should look like:



8. Double-click on the **lab3_TestBench.v** in the *Sources* pane to view its contents. This is the code you should see on your screen:

```

1      `timescale 1ns / 1ps
2
3      module lab3_TestBench();
4          //Declare wires and registers that will interface with the module under test
5          //Registers are initialized to known states. Wires cannot be initialized.
6          reg sys_clk=1;
7          wire sys_clkp;
8          wire [7:0] led;
9          reg [3:0] button;
10
11         //Invoke the module that we like to test
12         lab3_example ModuleUnderTest (.button(button),.led(led),.sys_clkp(sys_clkp),.sys_clk(sys_clk));
13
14         // Generate a clock signal. The clock will change its state every 5ns.
15         //Remember that the test module takes sys_clkp and sys_clk as input clock signals.
16         //From these two signals a clock signal, clk, is derived.
17         //The LVDS clock signal, sys_clkp, is always in the opposite state than sys_clk.
18         assign sys_clkp = ~sys_clk;
19         always begin
20             #5 sys_clk = ~sys_clk;
21         end
22
23         initial begin
24             #0 button <= 4'b1111;
25             #200 button <= 4'b1111;
26             #20 button <= 4'b1011;
27             #20 button <= 4'b1101;
28             #20 button <= 4'b1011;
29             #20 button <= 4'b0111;
30             #20 button <= 4'b1101;
31             #20 button <= 4'b0111;
32             #20 button <= 4'b1101;
33             #20 button <= 4'b0111;
34             #20 button <= 4'b1011;
35             #20 button <= 4'b1101;
36             #20 button <= 4'b1011;
37             #20 button <= 4'b0001;
38         end
39
40     endmodule

```

Let's look over the test bench Verilog code. The test bench defines the simulation step size and the resolution in line 1. In the module, we have defined two registers and two wire variables. These four variables are going to interface with the *lab3_example* module that was discussed in the first part of this lab. We instantiate *lab3_example* module in line number 12. Note that the name of the four variables in our top module, **lab3_TestBench**, are the same as the names of the variables in the *lab3_example* module. However, this does not have to be the case and you can choose different names for the variables in the top module.

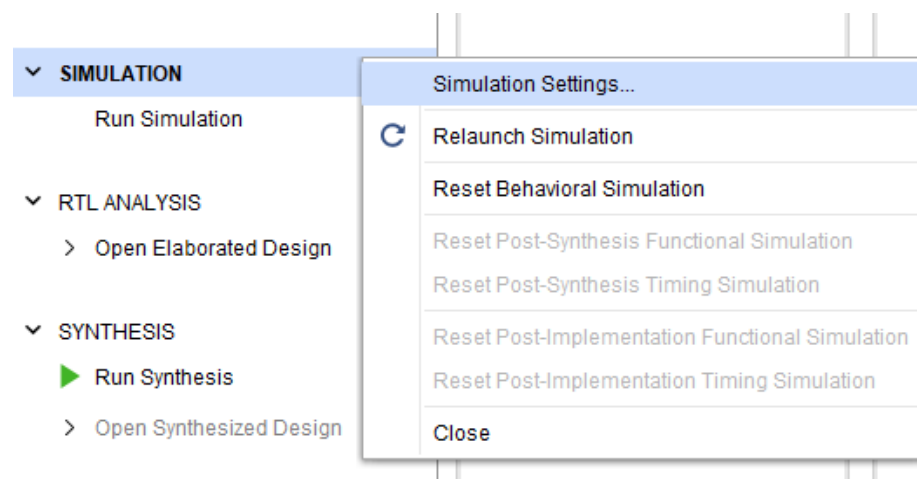
The *sys_clk* is declared as a register because its state will be explicitly varied in the test bench code. This is done in line 20, where every 5ns its state is toggled. This is performed in the always block. The register *sys_clk* must be first initialized and this is done when we declare the register. If you don't initialize the register, then the always code will not work properly because the simulator will not know what state to toggle. Since *sys_clkp* will always be in the opposite state than *sys_clk*, this variable is declared as a wire and its state is described in the *assign* code on line number 18.

Lines 23 through 37 describe the different states that *button* vector will go through. At time 0ns, we initialize the *button* vector to 4b'1111 state. Since the buttons are active low, state 4b'1111 means that none of the buttons are pressed. We held the buttons in this initial state for 200ns. The reason for this is that the FPGA will go through some initialization for the first ~150ns and you might observe erroneous results. It is best to wait 200ns before proceeding with testing the FSM. After another 20 ns, the *button* vector is set to 4b'1011, i.e. the 3rd button is pressed. After another 20 ns, the *button* vector is set to 4b'1101, i.e. the 2nd button is pressed. The rest of the code is easy to determine from the syntax.

Running the Simulator in Vivado

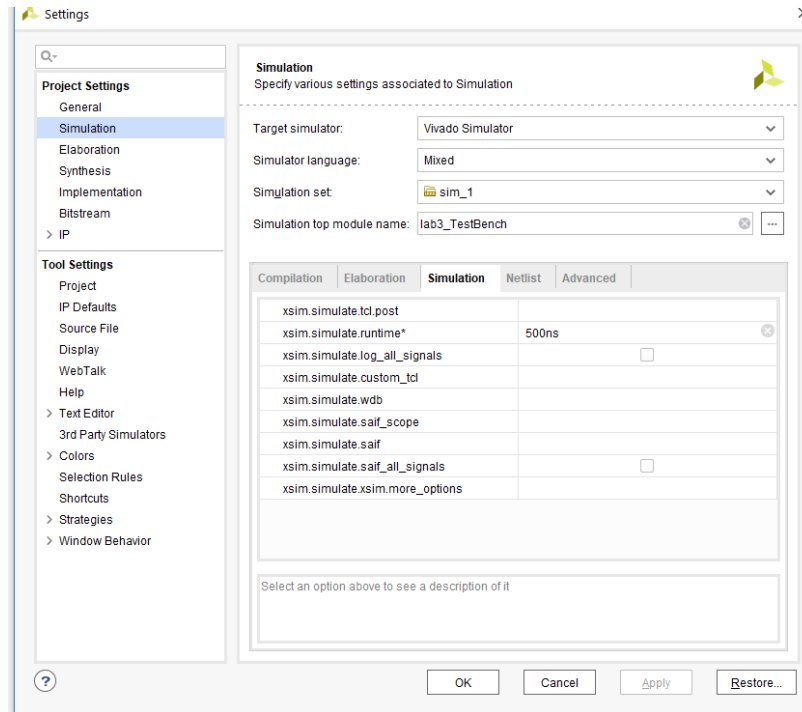
The next step is to simulate the test bench code that was added to your project. Here are the steps you will need to follow:

1. Place your mouse over the **SIMULATION** option under the *Project Manager* tasks of the *Flow Navigator* pane. Right click on the mouse while hovering over **SIMULATION** and select the **Simulation Settings** option. See the window below regarding this step:



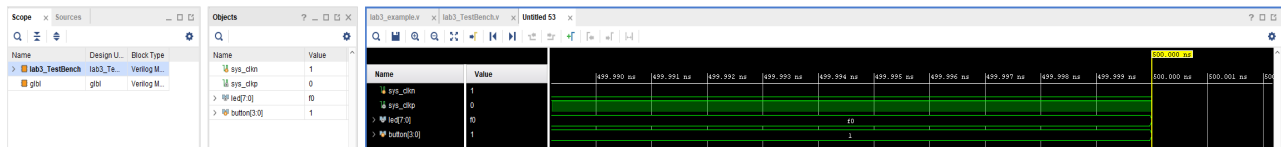
2. A **Settings** window will appear showing the **Simulation** properties form. In the **Simulation top module name:** enter **lab3_TestBench**. Next, select the **Simulation** tab, and set the

Simulation Run Time value to 500 ns. The window should look like the one below. Click **OK**.




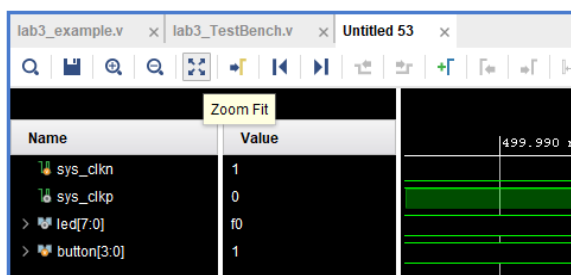
- Click on **Run Simulation > Run Behavioral Simulation** under the *Project Manager* tasks of the *Flow Navigator* pane.

The testbench and source files will be compiled, and the simulator will run if there are no errors in your Verilog code. You will see a simulator output like the one shown below.

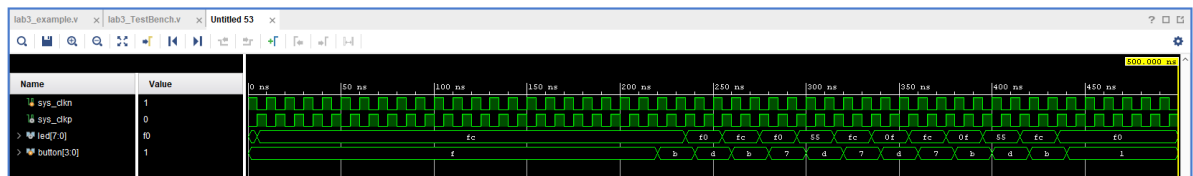


There are four main views: (i) *Scope*, where the testbench hierarchy as well as the different instances are displayed, (ii) *Objects*, where top-level signals are displayed, (iii) the waveform window, and (iv) *Tcl Console* where the simulation activities are displayed. Notice that since the testbench used is self-checking, the results are displayed after the simulation is completed.

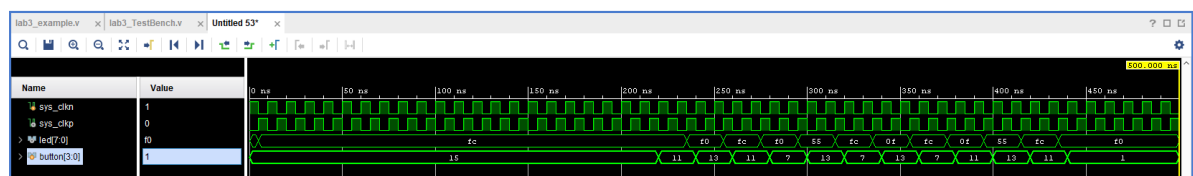
- Click on the *Zoom Fit* button, , located left of the waveform window to see the entire waveform.



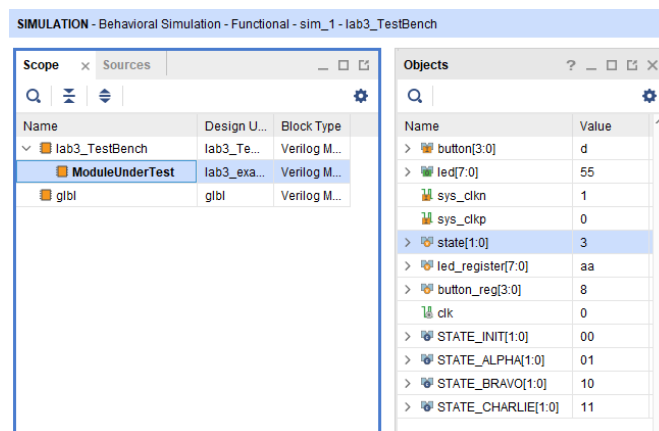
- You can float the simulation waveform window by clicking on the Float button on the upper right-hand side of the view. This will allow you to have a wider window to view the simulation waveforms. To reintegrate the floating window back into the GUI, simply click on the Dock Window button. The results from this simulation should look like this:



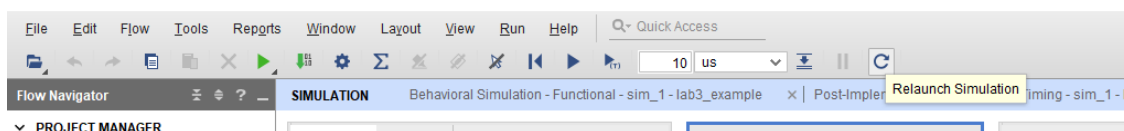
- Next, you will change the display format of your result. Select **button[3:0]** in the waveform window, right-click, select *Radix*, and then select *Unsigned Decimal* to view this variable in *integer* format. You can use other formats if desired in the future. Here is how the new result window should look like:



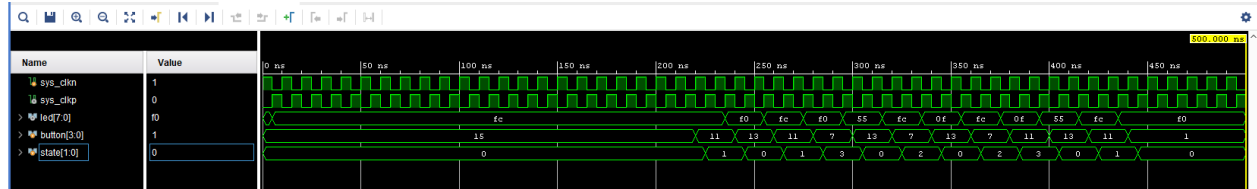
- Next, we will add additional variable to observe in the simulation window. You can observe any of the variables from the different modules that were instantiated in the test bench code. In this example, we can observe the different variables in the **lab3_example** module. To view these different variables, click on **ModuleUnderTest** in the **Scope** window. You should see the following variables in the **Objects** window.



- Select **state[1:0]** variable and drag it into the waveform window.
- In the simulator tool buttons ribbon bar, click on the **Relaunch Simulation** button. Here is where you can locate the **Relaunch Simulation** button.



11. Click on the *Zoom Fit* button and observe the output signal waveforms. Here is how the new waveform should look like:



Look at the **state[1:0]** output waveform. Trace through these different states as a function of the different **button[3:0]** inputs. Verify that these states correctly represent the FSM described in the block diagram.

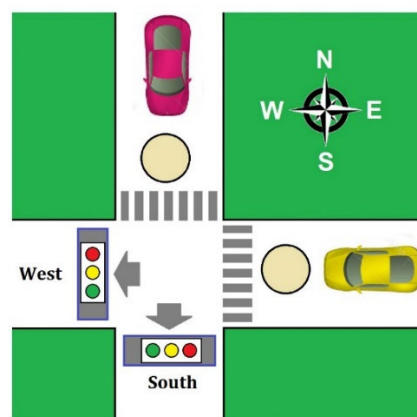
Checkpoint 1 (60 points)

- Make sure you go through all the steps described in this tutorial. Once you finish the tutorial, save the simulation plot window for lab3_TestBench.v file and include it in your report. You can press **Print Screen** button on your keyboard, which will capture the information displayed on your screen. You can paste this image in the work document. Then double click on the pasted image and use the crop function in the Word document to only select the simulation results. Make sure that your results are easy to view and understand. (10 points)
- Next, synthesize, implement and generate a bit file for lab3_example.v. These steps will generate a bit file which you will load in the FPGA via FrontPanel graphical user interface. Demonstrate to the TA that the FSM code is working properly by pressing different buttons. Explain to the TA the observed results, i.e the LED output, and relate them to the FSM block diagram. (10 points)
- Next, go to the **SIMULATION** bar in the *Flow Navigator* plane. Left click on **Run Simulation** and select **Run Post-Implementation Timing Simulation**. This simulation is run on your design after the synthesis and implementation process is completed. Print the simulation results and include them in your report. List three differences that you observe between the results from the “Behavior Simulation” and “Post-Implementation Timing Simulation”? What are the reasons for these differences between the two simulations? Use the zoom button to help you find these differences in the simulation windows. (20 points)
- Look at the waveform results at time 220ns from the start of the simulation. This is the time when the buttons change from state 4b’1111 to 4b’1011. Use the zoom in button to closely examine the results. How long does it take for the LEDs to change its output state measured from the time the buttons change from state 4b’1111 to 4b’1011? Do you see any erroneous states that the LEDs transition through? Use the zoom button and take a snapshot of these states. Include them in the report. Why do you have these erroneous states? (10 points)
- Compare these timing results to the ones from the “Behavioral simulation”. Report the time it takes for the LEDs to change its output measured from the onset the buttons change from state 4b’1111 to 4b’1011. Why is this time different compared to the one from the “Post-Implementation Timing Simulation”? (10 points)

Checkpoint 2 (60 points)

It is important that you develop the necessary debugging and simulation skills to efficiently write FSM. This will be a key skill that will enable you to efficiently complete your future lab assignments when you will develop interfaces with the various sensors. To this end, you will design a state machine for the traffic lights at a simple four-way intersection with pedestrian crosswalks. Here is the problem description for the FSM:

- The intersection that you are controlling is composed of two sets of lights for cars and one set of lights for pedestrians. The car lights control vehicle movements in either the east-west direction or the north-south, while the pedestrian lights control movement for pedestrians moving in all directions.
- There are eight lights to operate. The red, yellow, and green lights in the north-south direction will be designated as R1, Y1, G1. Similarly, the lights in the east-west direction will be called R2, Y2, and G2. There are red and green lights for the pedestrian, and you can name them as R3 and G3.
- The car lights should cycle through their state such that the green light is on for 1 seconds, the yellow light is on for ~0.5 second, and the red light is on for at least 1.5 seconds at a time.
- If a pedestrian pushes a button, then the pedestrian green light will be enabled only after the car light that is currently in either the green state or the yellow state finishes its state sequence and becomes red. In other words, the push button will not immediately turn the car lights to red but will rather wait until the car lights eventually return to red.
- The pedestrian green light will be on for 1 seconds. After this sequence, the car lights have to continue from where they left off before the button was pressed. If the north-south signal was green before the pedestrian light was on, then after the pedestrian light has gone through its green state to its red state, the east-west signal should turn to green.



Traffic Lights Intersection

Write the FSM for this traffic controller in Verilog. You will use the LEDs on the sensor board to indicate the car and pedestrian light signals. Once you complete your FSM machine demonstrate your results to the TA. Simulate the behavior of the FSM in Vivado and

demonstrate these results to the TA. Note, to be able to simulate your Verilog FSM code, you will probably have to run the FSM from a faster clock. Otherwise you will have to wait for a long time to observe the results.

Include your FSM Verilog code and test bench code in your final report. You should include only one copy for the entire report, which will contain answers to the questions in the lab and the Verilog codes for your FSM and test bench.