Laboratory Exercise #12
The Traffic Light Controller Lab

ECEN 248: Introduction to Digital Design

Department of Electrical and Computer Engineering
Texas A&M University

1 Introduction

The lab this week will continue the introduction to FSMs with a simple Traffic Light Controller design project. In the pre-lab, you will have the opportunity to create a state diagram of a *Mealy* machine, which implements a traffic light controller, based on provided guidelines. You will then use this state diagram to write the behavioral Verilog description of the traffic light controller. The testing of your traffic light controller will take place during the lab session using the ISE development tools and the Spartan 3E board. As with the previous lab, we will make use of the character LCD screen to display the outputs of our digital circuit.

2 Background

Background information necessary for the completion of this lab assignment will be presented in the next few subsection. The pre-lab assignment that follows will test your understanding of the background material.

2.1 The Mealy Machine

As mentioned in the previous lab, the *Mealy* machine is a Finite State Machine (FSM) in which the outputs of the machine are dependent not only on the state of the machine but also on the input to the machine. As a reminder, Figure 1 highlights the distinction between the *Mealy* and the Moore machine with a simple blue line that connects the input to the next-state logic.

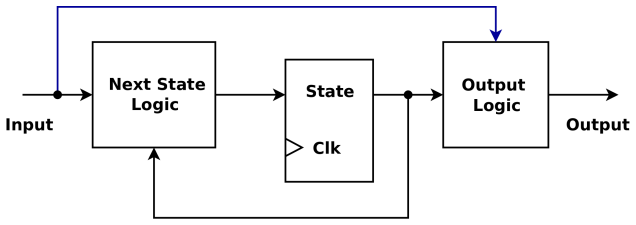

Figure 1: The Mealy Machine

As an example, consider the 2-bit saturating counter discussed previously. Let us suppose that we were asked to add an additional output signal, *Sat*, that is asserted when the counter is saturated in either the maximum or minimum state. Assuming that the counter is not considered to be in saturation when it first enters the minimum or maximum state, but rather when it remains in either of those states, we would need to use the state and the input of the machine to create this output signal. In terms of the state diagram, *Sat*

will need to be based on the state transitions of the machine as seen in Figure 2. Notice that *Sat* is defined on the transitions (i.e. edges of the graph) rather than on the states (i.e. nodes of the graph).

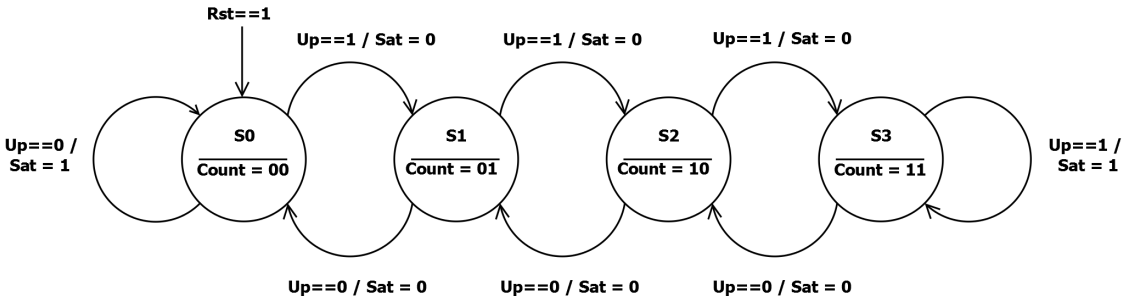

Figure 2: 2-bit Saturating Counter State Diagram

The behavioral Verilog for the modified 2-bit saturating counter is nearly identical to the previous description with the exception of an additional **always** block, which describes the combinational logic used to generate the *Sat* signal. To make clear the dependencies *Sat* has on the state and input, the **always**@(state **or** Up) clause has been used. It should be noted that **always**@(*) would have worked as well.

```
1  `timescale 1ns / 2 ns
   `default_nettype none
3
   /*This is a  behavioral Verilog description of*
5   *a 2-bit saturating counter.                  */
   module saturating_counter(
7      /*output and input are wires*/
       output wire [1:0] Count; //2-bit output
9      output reg Sat;//driven in an always block
       input wire Up; //input bit asserted for up
11     input wire Clk, Rst; //the usual inputs to a synchronous circuit

13     /*we haven't talked about parameters much but as you can see *
        *they make our code much more readable!                     */
15     parameter S0 = 2'b00,
                 S1 = 2'b01,
17               S2 = 2'b10,
                 S3 = 2'b11;
19
       /*intermeidate nets*/
21     reg [1:0] state; //4 states requires 2-bits
       reg [1:0] nextState; //will be driven in always block!
23
       /*describe Sat output logic*/
25     /*Notice that it closely resembles the next-state logic*/
```

```
    always@(state or Up) //purely combinational!
27      case(state)
            S0: begin
29              if(Up) //not saturated
                    Sat = 1'b0;
31              else //saturated
                    Sat = 1'b1;
33          end
            S1: Sat = 1'b0;
35          S2: Sat = 1'b0;
            S3: begin
37              if(Up) //saturated
                    Sat = 1'b1;
39              else //not saturated
                    Sat = 1'b0;
41          end
            //do not need a default because all states
43          //have been taken care of!
        endcase
45
    /*describe next state logic*/
47  always@(*) //purely combinational!
        case(state)
49          S0: begin
                if(Up) //count up
51                  nextState = S1;
                else //saturate
53                  nextState = S0;
            end
55          S1: begin
                if(Up) //count up
57                  nextState = S2;
                else //count down
59                  nextState = S0;
            end
61          S2: begin
                if(Up) //count up
63                  nextState = S3;
                else //count down
65                  nextState = S1;
            end
67          S3: begin
                if(Up) //saturate
69                  nextState = S3;
                else //count down
71                  nextState = S2;
            end
73          //do not need a default because all states
            //have been taken care of!
```

```
75          endcase

77      /*describe the synchronous logic to hold our state!*/
        always@(posedge Clk)
79          if(Rst) //reset state
                state <= S0;
81          else
                state <= nextState;
83
        /*describe the output logic, which in this case *
85       *happens to just be wires                        */
        assign Count = state;
87
endmodule //that's it!
```

2.2 Generating Timing Delays

In lab, you will need to implement pauses between state transitions in order to mimic the operation of a real traffic light. As you may have guessed, counters can be used to generate precise timing delays or pauses. Figure 3 depicts this concept with a state machine that after *Start* is asserted, waits in state *S1* for 1000 clock cycles and then in state *S2* for 2000 cycles. To accomplish this, the output signal from the state machine, *RstCount*, resets the counter, and in turn, the output of the counter (and input to the state machine), *Count*, controls the transitions from states *S1* and *S2*.

The actual time delay for each state can be easily calculated by multiplying the delay in terms of clock cycles by the clock period. For example, if the clock driving the aforementioned counter has a 50 MHz frequency, then the wait times in state *S1* and *S2* are given by:

$$t_{S1} = \frac{10^3}{50 * 10^6} = 20 * 10^{-6} = 20\mu s$$

$$t_{S2} = \frac{2 * 10^3}{50 * 10^6} = 40 * 10^{-6} = 40\mu s$$

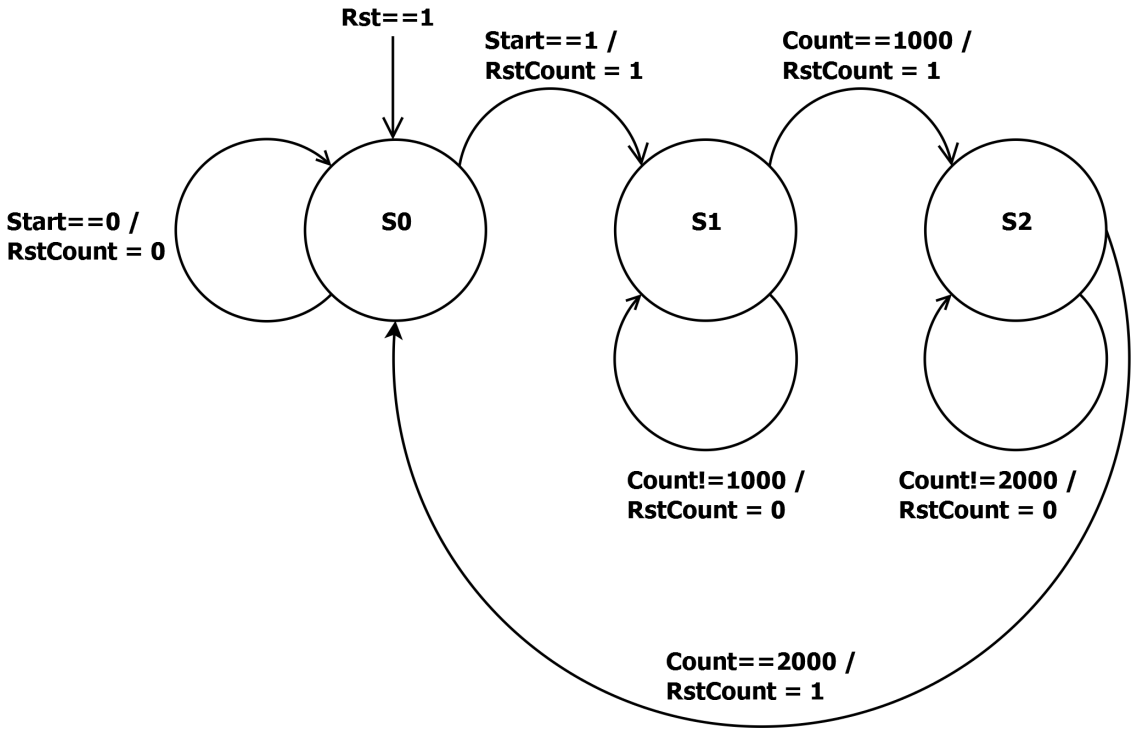

Figure 3: Counter Controlled Paused Transitions

3 Pre-lab

The objective of the pre-lab assignment is to create the state diagram and write the Verilog for the FSM that you will load onto the Spartan 3E board during lab. Therefore, the following subsections will describe the design we are attempting to create in detail. The pre-lab deliverables at the end will state what it is you are required to have prior to attending your lab session.

3.1 The Traffic Light Controller

This week you will design a traffic light controller for the highway and farm road intersection shown in Figure 4. For simplicity, we will assume that the traffic lights on opposite ends of the intersection are the same; in other words, if the light for the North bound traffic is green, then the light for the South bound traffic is also green. The traffic light controller outputs two 2-bit signals, *highwaySignal* and *farmSignal*, for

the highway road and the farm road, respectively. The following encoding is used for both signals:

$$00 : green$$
$$01 : yellow$$
$$10 : red$$

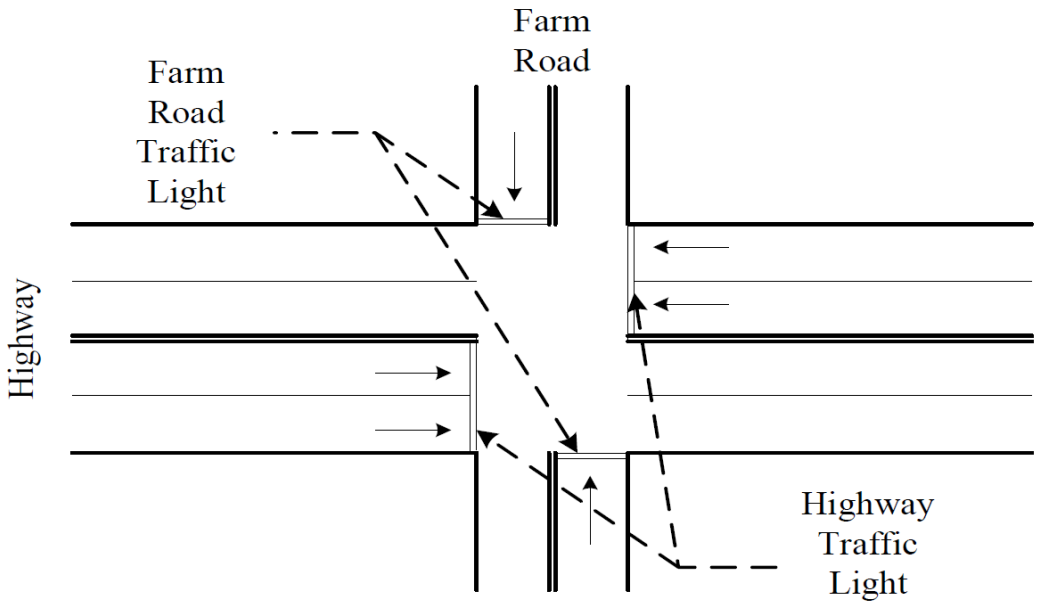

Figure 4: Highway and Farm Road Crossing

The traffic light controller will be made up of three main components as shown in Figure 5. You will be responsible for describing in Verilog the counter and FSM, while the LCD driver will be provided for you.

Obviously, to avoid accidents, the highway lights and farm road lights must not be green at the same time. Furthermore, the transitions from green on each road must be followed by 3 seconds of yellow and a 1 second period in which all directions are red. The latter allows motorists to clear the intersection prior to switching the direction of traffic. Because the traffic on the highway is much greater than that of the farm road, the traffic light must remain green on the highway longer than on the farm road. For this design, the highway light must remain green for 30 seconds, while the farm road light must remain green for only 15 seconds. Table 1 summarizes the above guidelines. Notice the partially blank column on the far right of Table 1 with the delays in terms of clock cycles. The entries in this column represent the values the counter

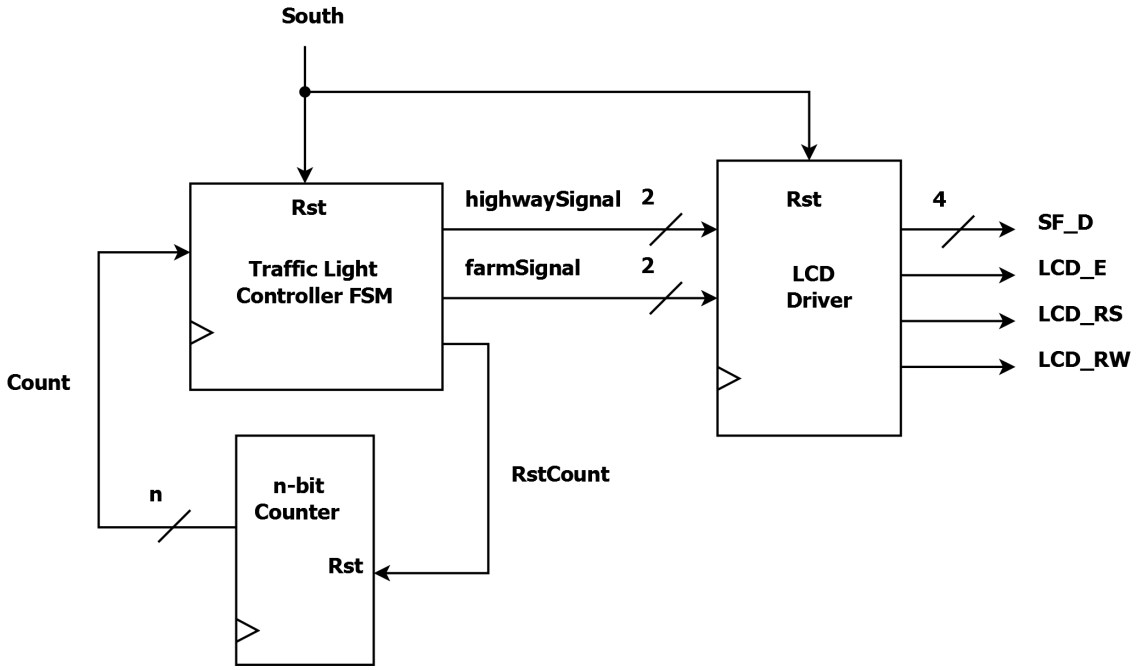

Figure 5: The Initial Traffic Light Controller Circuit

must reach in each state before transitioning to the next state. You will need to complete this column assuming a 50 MHz clock.

Table 1: Traffic Light Controller States

State	Highway Output	Farm Road Output	Delay (seconds)	Delay (cc)
S0	red	red	1	50,000,000
S1	green	red	30	
S2	yellow	red	3	
S3	red	red	1	
S4	red	green	15	
S5	red	yellow	3	

The brain of the traffic light controller is the FSM. For the the initial design, *Count* and *Rst* constitute the only input to the FSM. The output of the FSM includes the *highwaySignal* and *farmSignal* previously mentioned as well as *RstCount*, the counter reset. The operation of the FSM came be described by Table 1 and the time line in Figure 6. We can also describe this behavior using a state diagram similar to that seen

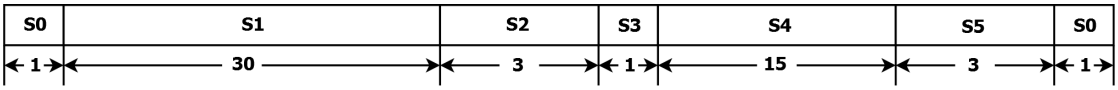

Figure 6: Time Line in Seconds of Initial FSM design

in Figure 3. One of the pre-lab exercises will require you to do just that.

3.2 Pre-lab Deliverables

Include the following items in you pre-lab submission in addition to those items mentioned in the *Policies and Procedures* document for this lab course.

1. Fill in the remaining entries in Table 1. You may find the discussion on *Generating Timing Delays* in the background section helpful.
2. Based on the column entries you just calculated, what is the necessary value of n in Figure 5?
3. Given Table 1 and Figure 6, create a state diagram for the traffic light controller FSM. Be sure to include the appropriate input and output labels. Assume *S0* is the reset state.
4. Now describe the traffic light controller FSM in Verilog using the following module interface:

```
module tlc_fsm(
    output reg [2:0] state, //output for debugging
    output reg RstCount,//use an always block
    /*another always block for these as well*/
    output reg [1:0] highwaySignal, farmSignal,
    input wire [n-1:0] Count, //use n computed earlier
    input wire Clk, Rst //clock and reset
);
```

4 Lab Procedure

Experiment 1 involves the implementation and testing of the design presented in the pre-lab, while experiment 2 requires that you extend the functionality of the pre-lab design.

4.1 Experiment 1

In experiment 1, you will synthesize the traffic light controller with the FSM you designed in the prelab. Correct functionality of the design will be determined by loading the design onto the Spartan 3E FPGA and verifying that it operates per per-lab specifications.

1. Synthesize and Implement the traffic light controller discussed in the pre-lab using ISE.

(a) Create a new ISE project called "lab12" and add the Verilog code you wrote in the pre-lab to the new project.

(b) Create a source file called "tlc_controller.v" and use the Verilog code below as a starting point for describing the circuit in Figure 5.

```
`default_nettype none

/*This module describes the top level traffic
  light controller module discussed in lab11*/
module tlc_controller(

    /*LCD interface wires make up our output!*/
    output wire LCD_E, LCD_RW, LCD_RS,
    output wire [3:0] SF_D,
    /*Let's output to J1 for debugging!*/
    output wire [3:0] J1,
    input wire Clk,
    /*the buttons provide input to our
      top-level circuit*/
    input wire South //use as reset
);

    /*intermediate nets*/
    wire SouthSync;
    wire [1:0] highwaySignal, farmSignal;
    wire RstCount;
    reg [30:0] Count;

    assign J1[3] = RstCount;//for debugging

    /*synchronize button inputs*/
    synchronizer syncSouth(SouthSync, South, Clk);

    /*instantiate FSM*/
    tlc_fsm FSM(
        .state(J1[2:0]), //wire state up to J1
        //finish things up here



        .Rst(SouthSync)//use synchronized signal
    );

    /*wire up LCD driver*/
    lcd_driver_lab11_ver1 LCD_driver(
        .Clk(Clk),
        .Rst(SouthSync),
```

```
44          .highwaySignal(highwaySignal),
            .farmSignal(farmSignal),
46          .SF_D(SF_D),
            .LCD_E(LCD_E),
48          .LCD_RS(LCD_RS),
            .LCD_RW(LCD_RW)
50      );

52      /*describe Counter with a syncrhonous reset*/

54

56

58  endmodule
```

(c) Copy the following source files from the course directory into your "lab12" directory.

- "lcd_driver_lab12_ver1.v", the LCD driver source file
- "synchronizer.v", the synchronizer module for our asynchronous inputs
- "tlc_controller.ucf", the top-level UCF

(d) Add the aforementioned source files along with the source file you just created to your ISE project. Synthesize the design, fixing any errors or warnings reported by the synthesis tool.

(e) Kick off the Implementation process, fixing all errors and warnings.

2. Load the design onto the FPGA board and test its functionality.

(a) Program the board using the bit stream created in the previous step.

(b) Press the "South" button to reset the design and observe the output on the LCD screen. Figure 7 illustrates the display format.

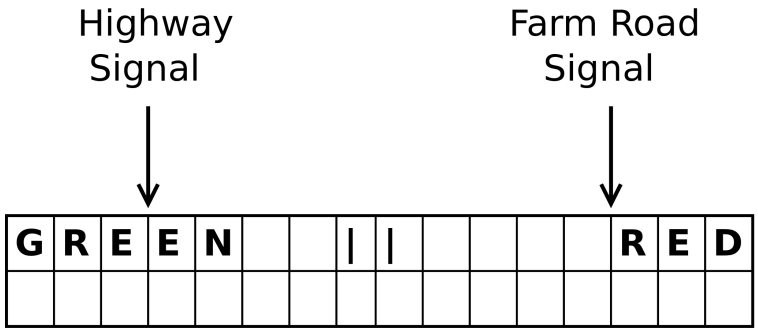

Figure 7: Display Format

(c) Verify that your traffic light controller cycles through the states appropriately.

(d) If your design works properly, **demonstrate your progress to the TA**; otherwise, you may want to use the logic analyzer for debugging. Additionally, you may find the FSM test bench file, "tlc_fsm_tb.v", to be helpful. Since simulation is quite slow, you will want to reduce the delays in your FSM module by several orders of magnitude prior to simulation. The provided test bench does not perform automatic verification so you will need to examine the waveform output to determine correct operation.

4.2 Experiment 2

The traffic light controller designed in the pre-lab is fairly inefficient. Most of the time, there are no cars on the farm road, except of course when there is an Aggie football game. Therefore, a sensor has been installed that asserts the signal, *farmSensor*, when a car is present on the farm road in either direction. In this experiment, you will modify your current design to support the new sensor.

1. Use the steps below to integrate the sensor signal into your current design.

   (a) Add an input port, *farmSensor*, to your FSM module.

   (b) Modify the FSM with the following the specifications:

   - The highway light shall remain GREEN as long as the sensor signal is LOW but must be GREEN for at least 30 seconds at a time.
   - Once transitioned to GREEN, the farm road light shall remain GREEN for at least 3 seconds.
   - As long as the sensor signal is HIGH, the farm road light shall remain GREEN but not for more than 15 additional seconds in order to give priority to the highway.
   - All signal light transitions from GREEN shall proceed to YELLOW and then to RED as before.

   (c) Modify the "tlc_controller.v" source file and the corresponding UCF to support the new signal. Use the "North" button on the Spartan 3E board to mimic the sensor output. Don't forget to properly synchronize the input signal.

2. Load the design onto the Spartan 3E FPGA and ensure proper operation.

   (a) Re-synthesize your design in ISE, correcting all warnings and errors.

   (b) Perform the Implementation process to create a bit stream for programming the FPGA. If warnings or errors are generated in this step, they are more than likely associated with your UCF modifications.

   (c) Program the Spartan 3E FPGA with the bit stream created in the previous step.
   Note: Demonstrate your progress to the TA once you have determined that your modified design is operating per specifications.

5 Post-lab Deliverables

Please include the following items in your post-lab write-up in addition to the deliverables mentioned in the *Policies and Procedures* document.

1. Include the source code with comments for **all** modules that you wrote or modified in lab. You do **not** need to include code that was provided! Remember that code without comments will not be accepted!
2. Include the state diagram for the modified traffic light controller FSM.
3. **(Honors)** Explain why we did not have to de-bounce the sensor signal, *farmSensor*, even though it was generated by a push-button for our prototyped design.

6 Important Student Feedback

The last part of lab requests your feedback. We are continually trying to improve the laboratory exercises to enhance your learning experience, and we are unable to do so without your feedback. Please include the following post-lab deliverables in your lab write-up.
Note: If you have any other comments regarding the lab that you wish to bring to your instructor's attention, please feel free to include them as well.

1. What did you like most about the lab assignment and why? What did you like least about it and why?
2. Were there any section of the lab manual that were unclear? If so, what was unclear? Do you have any suggestions for improving the clarity?
3. What suggestions do you have to improve the overall lab assignment?