Gabriel Emerson
ELEC 4200 – Lab 7
10/14/21

Lab 7: Traffic Light Controller

Goals:

The goal for this lab is to get more experience in different situations of using a finite state machine. In this lab in particular, we explore how finite state machines can be used to simulate/create traffic light signals. The creation of a traffic light controller is very common for developing skills with finite state machines, we used the Mealy design machine to implement all three tasks of this lab. We also got more experience with the use of the IP catalog and Clock wizard to get the timing exactly right for the changing of lights.

Design Process:

Task 1 is the simplest version of the three tasks. With this first task we are to implement the simplest version of the traffic light controller. This first light only has a red and green light instead of using all three (red, green, and yellow lights). This means our circuit should just be red going one direction, and green the other. Then after 3 seconds, the two directions will swap. Table 1 below shows the instructions for the controller. Then Figure 1 shows the picture of the lights for viewing that was given in the lab writeup.

North-South road	West-East road	ON time
red	green	3 sec
green	red	3 sec

Table 1

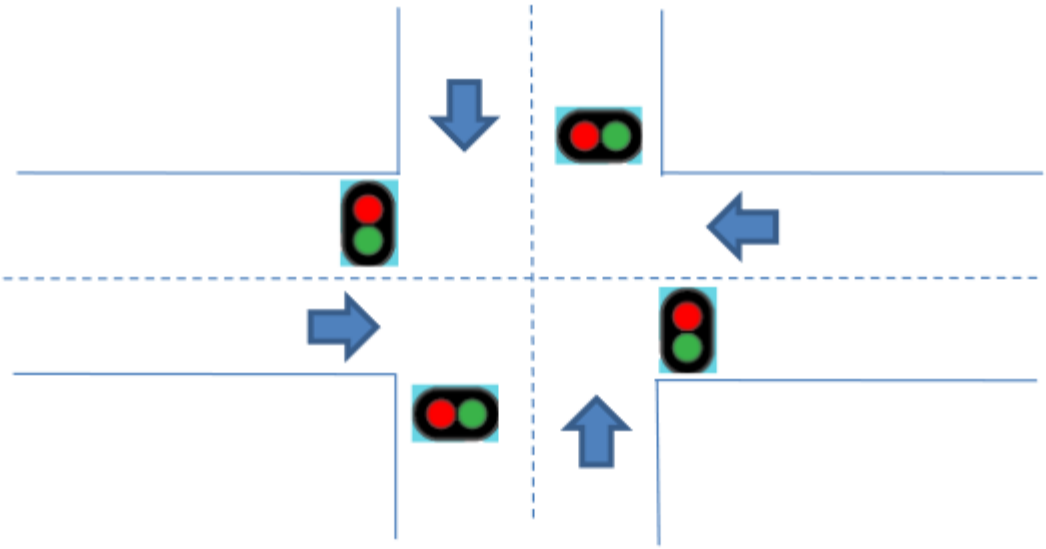

Figure 1

Task 2 was the advancement of task 1. It was the modification of the two light signal in task 1 and adding the yellow light into both directions. For this design, we will do very similar actions to that of the previous task but instead of just swapping from red to green in both directions every three seconds, we will have whichever signal is green, before it swaps to red, will enter a state of yellow for one second, then will proceed to the red state. This means for the other direction that it will have to wait another second before swapping to green in order to avoid both lights being green or yellow at the same time. The Table below shows the states and how they are implemented. Then Figure 2 shows the signals image given in the lab writeup.

North-South road	West-East road	ON time (seconds)
Red	Green	3
Red	Yellow	1
Red	Red	1
Green	Red	3
Yellow	Red	1
Red	Red	1
Red	Green	3
…	…	…

Table 2

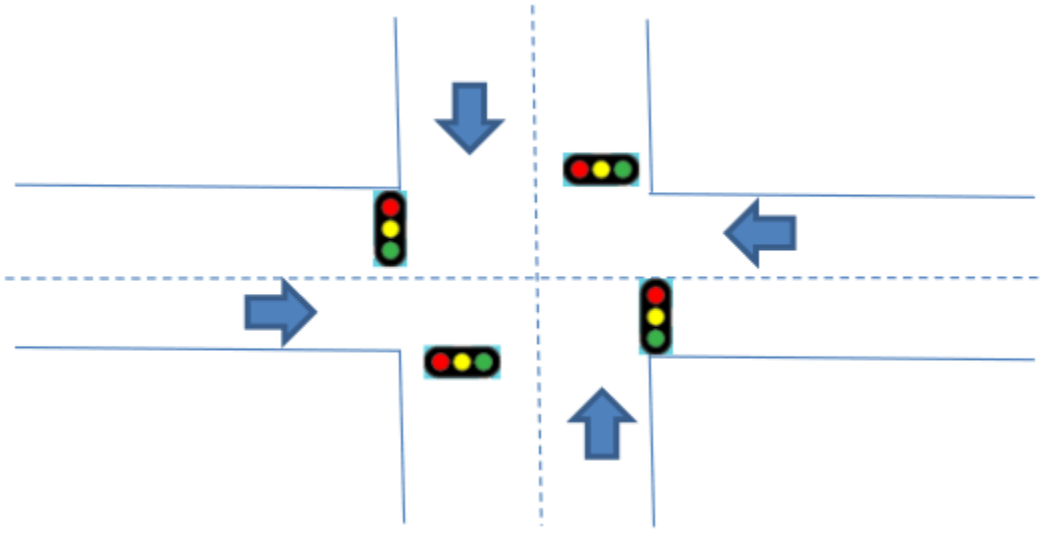

Figure 2

Task 3 is once again a step up from the previous task. This time instead of including another light in the signal, we include a pedestrian crossing state. When this state occurs, all lights go to red for safe pedestrian crossing. Then after a few seconds, the pedestrian has crossed, and we can resume the state of the lights. This added state is different from adding another light to a signal, because this does not modify previous states, but instead adds an interrupt-like button that when pressed will shift everything to a different state for a short period of time, before coming back and resuming where we were. Due to this, we can use Table 2 to determine the signal states, and Figure 3 shows the light image given in the writeup.

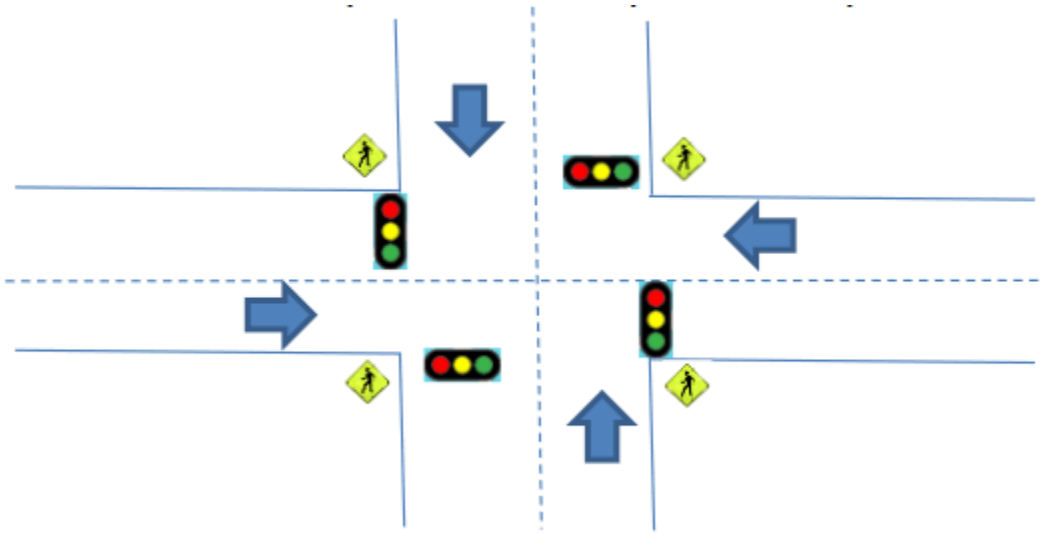

Figure 3

Detailed Design:

Task 1 was the hardest task to get all the way done first, because this task had no other to base itself off of. We were tasked with using a Mealy machine to implement our design of the 2 light system. Using the state diagram drawn before the start of the lab, all that needed to be done was follow the state diagram after getting down the framework for a Mealy finite state machine. The code used was too long to show in one complete image, so I had to take several images of the code, that we will call Figure 4.

```
module Lab7_1(clock_in, reset, clock_out, NS_Green, NS_Red, EW_Green, EW_Red);
    input clock_in, reset;
    output clock_out;
    output reg NS_Green;
    output reg NS_Red;
    output reg EW_Green;
    output reg EW_Red;

    wire top_clock_out;
    reg [2:0] count=0;
    reg change_occured=0;
    parameter NS_Go = 2'b00, EW_Go = 2'b01;
    reg[1:0] state, nextstate;

    clk_wiz_0
   (
  // Clock out ports
    .clk_out1(top_clock_out),     // output clk_out1
    .clk_in1(clock_in));      // input clk_in1
    divider(.clockin(top_clock_out), .clockout(clock_out) );
```

```
        always @ (posedge clock_out or posedge reset)
        begin
            if (reset)
            begin
                state <= NS_Go;
                count <=0;
            end
            else
            begin
                if (count<2)
                begin
                    count <= count + 1;
                end
                else
                begin
                    count <= 0;
                    state <= nextstate;
                end
            end
        end

        always @ (state)
        begin
            case(state)

                NS_Go: nextstate = EW_Go;
                EW_Go:nextstate = NS_Go;
            default :nextstate =EW_Go;
            endcase
        end

        always @ (state or count or reset)
        begin
            case(state)
                NS_Go: begin NS_Green =1; NS_Red =0; EW_Green = 0; EW_Red =1; end
                EW_Go: begin NS_Green =0; NS_Red =1; EW_Green = 1; EW_Red =0; end
            default: begin NS_Green =0; NS_Red =1; EW_Green = 1; EW_Red =0; end
            endcase
        end
    endmodule

module divider(clockin,clockout);
    input clockin;
    output reg clockout; // output clock after dividing the input clock by divisor
    reg [31:0] count=32'b0;
    parameter DIVISOR = 5000000;
    // The frequency of the output clock_out = The frequency of the input clk_in divided by DIVISOR
    // For example: clock_in = 5Mhz, if you want to get 100kHz signal to blink LEDs
    // You will modify the DIVISOR parameter value to 50.

    always @ (posedge clockin) //count until you get to the value of 50(DIVISOR)
        begin
            if (count == DIVISOR - 1)
                count <= 32'b0;
            else
                count <= count + 1;
        end

    always @ (posedge clockin)
        begin
            if (count < DIVISOR/2)   //clock_out changes every 25(DIVISOR/2) cycles, which make its fre
                clockout <= 1;
            else
                clockout <= 0;
        end
endmodule
```

Figure 4

Task 2 was easier to begin but had more bugs than the first task. Since this task practically builds off the previous one, I simply used the framework for that and added in another state for the yellow light. This was once again easy to follow using the state diagrams drawn before the start of the lab. The code is shown below. Once again the code was too long to fit into an image, so the code has been broken up into several images below named Figure 5.

```
module Lab7_2(clock_in, reset, clock_out, NS_Green, NS_Yellow, NS_Red, EW_Green, EW_Yellow, EW_Red);
    input clock_in, reset;
    output clock_out;
    output reg NS_Green;
    output reg NS_Yellow;
    output reg NS_Red;
    output reg EW_Green;
    output reg EW_Yellow;
    output reg EW_Red;

    wire top_clock_out;
    reg [1:0] count_green = 0;
    reg [1:0] count_yellow = 0;
    reg [1:0] count_red = 0;

    reg change_occured = 0;
    parameter State_NS_Green = 3'b000, State_NS_Yellow= 3'b001, State_NS_Red= 3'b010, State_EW_Green = 3'b011, State_EW_Yellow = 3'b100, State_EW_Red = 3'b101;
    reg[2:0] state, nextstate;

    clk_wiz_0
   (
    // Clock out ports
    .clk_out1(top_clock_out),     // output clk_out1
    .clk_in1(clock_in));      // input clk_in1
    divider(.clockin(top_clock_out), .clockout(clock_out) );
always @ (posedge clock_out or posedge reset)
begin
    if (reset)
    begin state <= State_NS_Green;count_green <=0;count_yellow <=0; count_red <=0;end
    else
    begin
        if (count_green<2)
        begin count_green <= count_green + 1; end
        else
        begin
            count_green <= 0; if (state == State_NS_Green) begin state <= nextstate; end else if (state == State_EW_Green)begin state <= nextstate; end
        end
        if (count_yellow<1) begin count_yellow <= count_yellow + 1; end
        else
        begin
            count_yellow <= 0;
            begin if (state == State_NS_Yellow) /*NS YELLOW*/ begin state <= nextstate; end  else if (state == State_EW_Yellow) /* NS RED */begin state <= nextstate; end
            end
        end
        if (count_red<1) begin count_red <= count_red + 1; end
        else
        begin
            count_red <= 0;
            begin if (state == State_NS_Red) begin state <= nextstate; end else if (state == State_EW_Red)begin state <= nextstate; end
            end
        end
    end
end

always @ (state)
begin
    case(state)
        State_NS_Green: nextstate = State_NS_Yellow;
        State_NS_Yellow: nextstate = State_NS_Red;
        State_NS_Red: nextstate = State_EW_Green;
        State_EW_Green: nextstate = State_EW_Yellow;
        State_EW_Yellow: nextstate = State_EW_Red;
        State_EW_Red: nextstate = State_NS_Green;
    default :nextstate = State_EW_Red;
    endcase
end

    always @ (state or count_red or count_yellow or count_green or reset)
    begin
        case(state)
            State_NS_Green: begin NS_Green =1; NS_Yellow =0; NS_Red =0; EW_Green = 0; EW_Yellow =0; EW_Red = 1; end
            State_NS_Yellow: begin NS_Green =0; NS_Yellow =1; NS_Red =0; EW_Green = 0; EW_Yellow =0; EW_Red = 1; end
            State_NS_Red: begin NS_Green =0; NS_Yellow =0; NS_Red =1; EW_Green = 0; EW_Yellow =0; EW_Red = 1; end
            State_EW_Green: begin NS_Green =0; NS_Yellow =0; NS_Red =1; EW_Green = 1; EW_Yellow =0; EW_Red = 0; end
            State_EW_Yellow: begin NS_Green =0; NS_Yellow =0; NS_Red =1; EW_Green = 0; EW_Yellow =1; EW_Red = 0; end
            State_EW_Red: begin NS_Green =0; NS_Yellow =0; NS_Red =1; EW_Green = 0; EW_Yellow =0; EW_Red = 1; end
        default: begin NS_Green =0; NS_Yellow =0; NS_Red =0; EW_Green = 0; EW_Yellow =0; EW_Red = 0; end
        endcase
    end
endmodule

module divider(clockin,clockout);

    input clockin;
    output reg clockout;
    reg [31:0] count=32'b0;
    parameter DIVISOR = 5000000;
    // The frequency of the output clock_out = The frequency of the input clk_in divided by DIVISOR
    // For example: clock_in = 5Mhz, if you want to get 100kHz signal to blink LEDs
    // You will modify the DIVISOR parameter value to 50.

    always @ (posedge clockin) //count until you get to the value of 50(DIVISOR)
        begin
            if (count == DIVISOR - 1)
                count <= 32'b0;
            else
                count <= count + 1;
        end

    always @ (posedge clockin)
        begin
            if (count < DIVISOR/2)   //clock_out changes every 25(DIVISOR/2) cycles, which make its frequency 100kHz. This means it will go high once every 10us.
                clockout <= 1;
            else
                clockout <= 0;
        end

endmodule
```

Figure 5

Task 3 was once again following the state diagrams drawn before the lab. This was easy to implement following the last task. We can essentially use the last task, but also setup an interrupt-like button that when pressed will stop everything and go to a new state. Then after a few seconds, it will resume the program of lights. The state diagram will be very similar to the one before. The code below is broken up due to the length of the code, but all the code together is found in Figure 6.

```
module Lab7_3(clock_in, reset, pedestrian, clock_out, NS_Green, NS_Yellow, NS_Red, EW_Green, EW_Yellow, EW_Red);
    input clock_in, reset, pedestrian;
    output clock_out;

    output reg NS_Green;
    output reg NS_Yellow;
    output reg NS_Red;
    output reg EW_Green;
    output reg EW_Yellow;
    output reg EW_Red;

    wire top_clock_out;
    reg [1:0] count_green =0;
    reg [1:0] count_yellow =0;
    reg [1:0] count_red =0;
    reg [1:0] pedestrian_count =0;

    reg change_occured=0;
    parameter State_Pedestrian = 3'b110, State_NS_Green = 3'b000, State_NS_Yellow= 3'b001, State_NS_Red= 3'b010, State_EW_Green = 3'b011, State_EW_Yellow = 3'b100, State_EW_Red = 3'b101;
    reg[2:0] state, nextstate;

    clk_wiz_0
    (
     // Clock out ports
    .clk_out1(top_clock_out),
    .clk_in1(clock_in));
    divider(.clockin(top_clock_out), .clockout(clock_out) );
    always @ (posedge clock_out or posedge reset or posedge pedestrian)
    begin
        if (reset)
        begin state <= State_NS_Green;count_green <=0;count_yellow <=0; count_red <=0;end
        else if (pedestrian)
        begin
           // pedestrian_count = 0;
           state <= State_Pedestrian;
           //nextstate <= State_NS_Green;

           // count_green <=0;count_yellow <=0; count_red <=0;
        end
        else
        begin
           if (count_green<2)
            begin count_green <= count_green + 1; end
           else
            begin
                count_green <= 0; if (state == State_NS_Green) begin state <= nextstate; end else if (state == State_EW_Green)begin state <= nextstate; end
            end
            if (count_yellow<1) begin count_yellow <= count_yellow + 1; end
            else
            begin
                count_yellow <= 0;
                begin if (state == State_NS_Yellow) /*NS YELLOW*/ begin state <= nextstate; end  else if (state == State_EW_Yellow) /* NS RED */begin state <= nextstate; end
                end
            end
            if (count_red<1) begin count_red <= count_red + 1; end
            else
            begin
                count_red <= 0;
                begin if (state == State_NS_Red) begin state <= nextstate; end else if (state == State_EW_Red)begin state <= nextstate; end
                end
            end
           if (pedestrian_count < 1) begin pedestrian_count <= pedestrian_count + 1; end
           else
           begin
                pedestrian_count <=0;
                begin
                if (state == State_Pedestrian)
                    state <= nextstate;
                end
           end
        end
    end
```

```
            always @ (state)
            begin
                case(state)
                    State_NS_Green: nextstate = State_NS_Yellow;
                    State_NS_Yellow: nextstate = State_NS_Red;
                    State_NS_Red: nextstate = State_EW_Green;
                    State_EW_Green:  nextstate = State_EW_Yellow;
                    State_EW_Yellow:  nextstate = State_EW_Red;
                    State_EW_Red: nextstate = State_NS_Green;
                    State_Pedestrian: nextstate = State_NS_Green;
                    default :nextstate =State_EW_Red;
                endcase
            end

            always @ (state or pedestrian or count_red or count_yellow or count_green or reset)
            begin
                case(state)
                    State_NS_Green: begin NS_Green =1; NS_Yellow =0; NS_Red =0; EW_Green = 0; EW_Yellow =0; EW_Red = 1; end
                    State_NS_Yellow: begin NS_Green =0; NS_Yellow =1; NS_Red =0; EW_Green = 0; EW_Yellow =0; EW_Red = 1; end
                    State_NS_Red: begin NS_Green =0; NS_Yellow =0; NS_Red =1; EW_Green = 0; EW_Yellow =0; EW_Red = 1; end
                    State_EW_Green: begin NS_Green =0; NS_Yellow =0; NS_Red =1; EW_Green = 1; EW_Yellow =0; EW_Red = 0; end
                    State_EW_Yellow: begin NS_Green =0; NS_Yellow =0; NS_Red =1; EW_Green = 0; EW_Yellow =1; EW_Red = 0; end
                    State_EW_Red: begin NS_Green =0; NS_Yellow =0; NS_Red =1; EW_Green = 0; EW_Yellow =0; EW_Red = 1; end
                    State_Pedestrian: begin NS_Green =0; NS_Yellow =0; NS_Red =1; EW_Green = 0; EW_Yellow =0; EW_Red = 1; end
                default: begin NS_Green =0; NS_Yellow =0; NS_Red =0; EW_Green = 0; EW_Yellow =0; EW_Red = 0; end
                endcase
            end
        endmodule

module divider(clockin,clockout);

    input clockin;
    output reg clockout;
    reg [31:0] count=32'b0;
    parameter DIVISOR = 5000000;
    // The frequency of the output clock_out = The frequency of the input clk_in divided by DIVISOR
    // For example: clock_in = 5Mhz, if you want to get 100kHz signal to blink LEDs
    // You will modify the DIVISOR parameter value to 50.

    always @ (posedge clockin) //count until you get to the value of 50(DIVISOR)
        begin
            if (count == DIVISOR - 1)
                count <= 32'b0;
            else
                count <= count + 1;
        end

    always @ (posedge clockin)
        begin
            if (count < DIVISOR/2)   //clock_out changes every 25(DIVISOR/2) cycles, which make its frequency 100kHz. This means it will go high once every 10us.
                clockout <= 1;
            else
                clockout <= 0;
        end
endmodule
```

Figure 6

Verification:

All three of these tasks were verified but uploading each one to the board, and visually verifying the functionality following the LED's. Each task had a bit of a different output, but they can be checked by finding each state on the board and how they get there and comparing to the state diagrams drawn for each task. The tasks functionality are as follows:

Task 1 should have two lights for each direction (NS & WE). At any time One should be red and the other green, then after 3 seconds, switch.

Task 2 should have 3 lights for each direction (NS & WE). At any time One direction will be red and the other green, then after 3 seconds the green will go yellow for 1 second, then red and the red light will go 4 seconds, then green (4 seconds because 3 + 1 = 4).

Task 3 should have 3 lights for each direction (NS & WE) and a pedestrian crossing button. The directions of signals are the same as Task 2 above, but at any time a crossing button may be pressed, then all lights go red and the crossing signal is green for 3 seconds, then resume.

Conclusion:

What I learned in this lab is how often finite state machines are used in real world systems. Most systems used in real world jump from state to state depending on possible inputs and the previous state. I also learned how easy it is to write a fsm if you understand the framework of fsm and have previously drawn a state diagram of the machine you are designing. We also got more experience with the Clock Wizard in IP Catalog and using it to get an exact number of seconds for the lights to switch. This was used to get more experience and design three kinds of traffic light controllers.