

ECEN2350

Project3_Report

Chengming Li

1. Project Statement

In this project, we are required to design a mealy finite state machine that emulates the behavior of taillights of a 1965 Ford Thunderbird Automobile. There are seven states I am going to emulate. They are idle light, hazard light, left turn light, right turn light, brake light, brake_left light and brake_right light. We are required to display the light via De10-lite on board LEDR[9:7] and LEDR[2:0]. Moreover, there are two control modes we need to design, manually control mode and automation mode. These two modes are controlled by the SW[9]. Within the autonomous mode, we are going to design the on board memory block to automatically run through the states with 0.2 Hz frequency.

2. Theory Statement


```

always@(*)
begin
    reset_counter = 1;
    case(CurrentState)
        Idle :begin
            if (SW[0] == 1) begin
                NextState = Hazard;
            end
            else if (SW[2] == 1) begin

                if (SW[1] == 1) begin
                    if (KEY[1] == 0) begin
                        NextState = B_Right;
                        reset_counter = 0;
                    end
                    else if (KEY[1] == 1) begin
                        NextState = B_Left;
                        reset_counter = 0;
                    end
                end
                else begin
                    NextState = Brake;
                end
            end
            else if (SW[1] == 1) begin

                if (KEY[1] == 0) begin
                    NextState = Right;
                    reset_counter = 0;
                end
                else if (KEY[1] == 1) begin
                    NextState = Left;
                    reset_counter = 0;
                end
            end
            else begin
                NextState = Idle;
            end
        end
    endcase
end

```

Figure 2. Logic behind each case statement

In the document of Project3, we've been told that the Hazard Light has the highest priority in the logic. Thus, the Hazard light goes first and determined by the $SW[0] = 1$. Then, everything about the Brake light goes next, because we must determine whether the Turn light is in the Brake state at first. If the Turn light state goes next, the state will not get into the Brake state because it only judges the status of $SW[1]$. Moreover, the $SW[2]$ drives the Brake state if

SW[1] is off. Meanwhile, the SW[2] and KEY[1] will also drive the state to Brake_Left or Brake_Right when the SW[1] = 1. After the Brake logic, the Turn Light goes next and it is determined by the status of SW[1] and KEY[1]. The state is Left Turn when the KEY[1] is not pressed and the state is Right Turn when the KEY[1] is pressed. At the end, the state will go back to Idle state if the Current State does not satisfy any previous states. In conclusion, every state in my NSL block has the same logic as shown in Figure 2.

In the Current State Logic, the only logic in this state is that updating the Current State as output, given the Next State as input. And this block is driven by the clock, which means the State will update whenever the clock hits the positive edge.

```

34      Left: begin
35          LEDR[6:0] = 4'b0;
36          if (counter == 1 || counter == 2) begin
37              LEDR[9:7] = 3'b001;
38          end
39          else if (counter == 3 || counter == 4) begin
40              LEDR[9:7] = 3'b011;
41          end
42          else if (counter == 5) begin
43              LEDR[9:7] = 3'b111;
44          end
45          else begin
46              LEDR[9:7] = 3'b0;
47          end
48      end
49      Right: begin
50          LEDR[9:3] = 4'b0;
51          if (counter == 1 || counter == 2) begin
52              LEDR[2:0] = 3'b100;
53          end
54          else if (counter == 3 || counter == 4) begin
55              LEDR[2:0] = 3'b110;
56          end
57          else if (counter == 5) begin
58              LEDR[2:0] = 3'b111;
59          end
60          else begin
61              LEDR[2:0] = 3'b0;
62          end
63      end

```

Figure 3. LEDR logic at the state of Turn

In the Output logic, this block will output the LEDR as tail lights we want given Current State as input. One logic I found is the most tough one is the flashing of Turn Light, because the LEDR does not blink simultaneously. Taking Left Turn as an example, the LEDR[9:7] will blink













like this: 000, 001, 011, 111. So, one way I created this logic is that creating a counter. The first LEDR will turn on whenever the counter counts 1 or 2. The first two LEDR will turn on whenever the counter counts to 3 or 4. And all the LEDRs will turn on whenever the counter counts to 5. After that, all LEDRs will turn off whenever the counter counts to 6 and reset the counter. Beside that, the Hazard light will flash at the same frequency and the state of clock so that the LEDRs could blink as shown in the video. In the Brake state, the LEDRs just stay on for the whole period.

```
1  DEPTH = 8;
2  WIDTH = 8;
3
4  ADDRESS_RADIX = DEC;
5  DATA_RADIX = BIN;
6
7  CONTENT
8  BEGIN
9  % Brake, LR select, TS enable, Hazard %
10 0 : 00000000; % IDLE No Brake %
11 1 : 00000001; % IDLE Brake %
12 2 : 00010010; % HAZARD %
13 3 : 00000010; % LT No Brake %
14 4 : 00010110; % LT Brake %
15 5 : 00000110; % RT No Brake %
16 6 : 00000000; % RT Brake %
17 7 : 00000000; % Unassigned %
18 END;
```

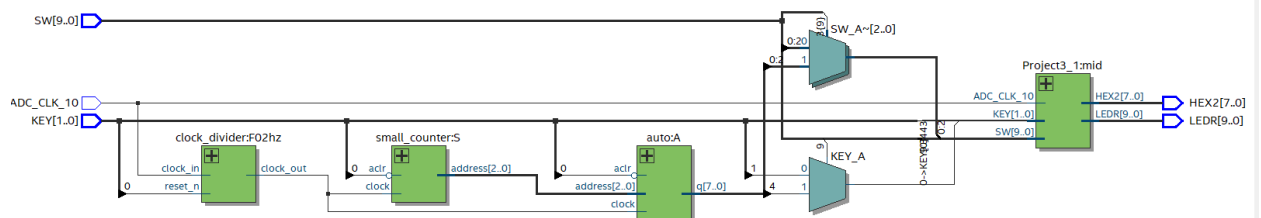
Figure 4. Mif

At the last part of this project, we are going to design the automation mode by designing the on board memory block M9K. Generally, in the automation mode, I do not have to manually control the SW or KEY to emulate the cycling of states, because the memory block does for me. For the memory block, it is generated by Quartus. And few things we are going to do is define the width and depth of the memory. Then, I could program the memory block via the .mif file by assigning the output value in each address. In this project, the depth of the memory block is 8 since I have 7 states in my NSL, and the width is 4 because we need to control 3 SW and 1 KEY.

3. Hierarchy of My Project Source Files

Entity:Instance	
 MAX 10: 10M50DAF484C7G	
▼  Project3	
>  auto:A	
 clock_divider:F02hz	
 small_counter:S	
▼  Project3_1:mid	
 counter:C1	
 CSL:CS	
 clock_divider:F5hz	
 decoder:H2	
 NSL:NS	
 OL:O	

4. Block Diagram



5. Description of Testbench Operation

```

Project3 > Source > tb_State.v
1  `timescale 1 ns / 100 ps
2  module tb_State();
3
4      reg clock = 0;
5      reg reset_n;
6      reg [9:0] SW;
7      reg [1:0] KEY;
8
9      wire [9:0] LEDR;
10     wire reset_counter;
11
12     wire [3:0] CurrentState, NextState;
13     wire [2:0] counter, hazard;
14     always #10 clock = ~clock;
15
16     CSL CS (.clock(clock), .reset_n(reset_n), .CurrentState(CurrentState), .NextState(NextState));
17
18     NSL NS (.CurrentState(CurrentState), .NextState(NextState), .SW(SW), .KEY(KEY), .reset_counter(reset_counter));
19
20     OL O (.LEDR(LEDR), .CurrentState(CurrentState), .counter(counter), .hazard(hazard));
21
22     counter C1(.clock(clock), .CurrentState(CurrentState), .counter(counter), .reset_n(reset_n), .hazard(hazard), .reset_counter(reset_counter));
23
24     initial
25     begin
26         $dumpfile("tb_State.vcd");
27         $dumpvars;
28         reset_n = 0; KEY[1] = 1;
29         #20 reset_n = 1;
30
31         #20 SW[0] = 1;          //Hazard
32         #200
33         #20 SW[0]=0; SW[1]=1; //Left Turn
34         #200
35         #20 SW[2] = 1;        //Brake_Left
36         #200
37         #20 KEY[1] = 0;       //Brake_Right
38         #200
39         #20 SW[2]=0;          //Right Turn
40         #200
41         #20 SW[1]=0;SW[2]=1;  //Brake Light
42         #100
43         #40 reset_n =0;
44         #100 $finish;
45     end
46
47     initial begin
48         $monitor($time, " CurrentState = %d , NextState = %d , LEDR = %b", CurrentState,NextState, LEDR);
49     end

```

Figure 5. Testbench

In my testbench, I instantiate three state blocks and a counter block to simulate my design in different states. From line 31 to line 44, I simulate all the states except Brake, since I think the Brake_turn light already includes the state of Brake light. Beside that, I keep each state has a 200 ns period so that I could tell the changing of LEDRs. In this project, GTKwave is the better way to see the changing of state, so I did not use the display statement to print out all the variables.

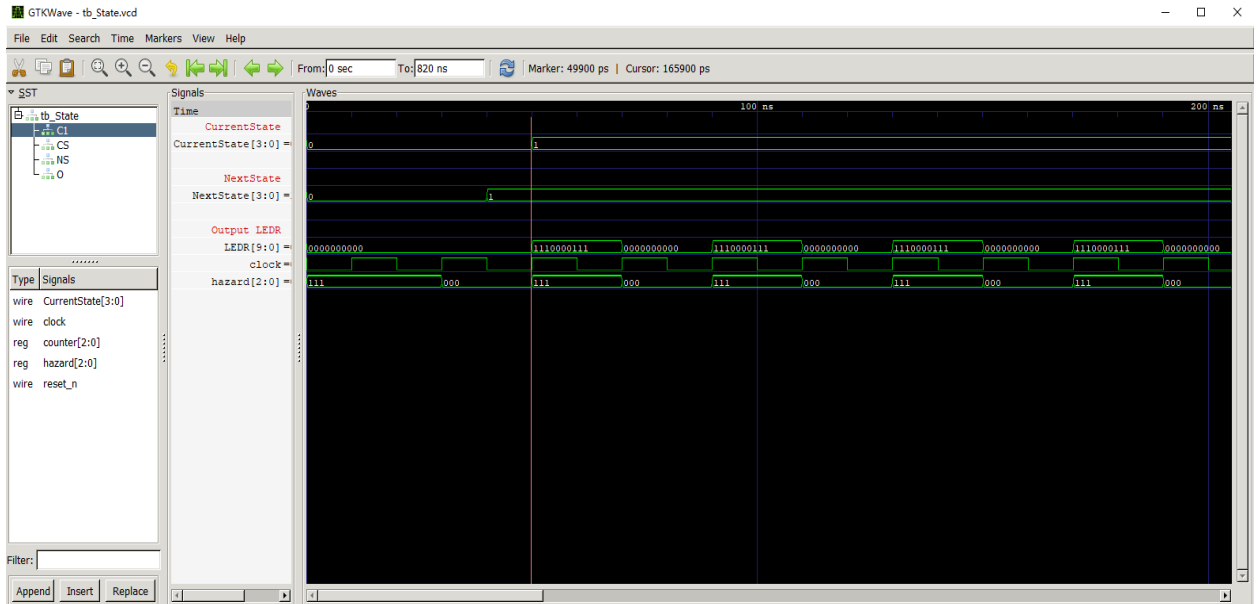


Figure 6. Hazard Light (State = 1)

The Hazard light will turn on when the SW[0] = 1. And the LEDRs will blink at the same frequency of the clock as shown in Figure 6.

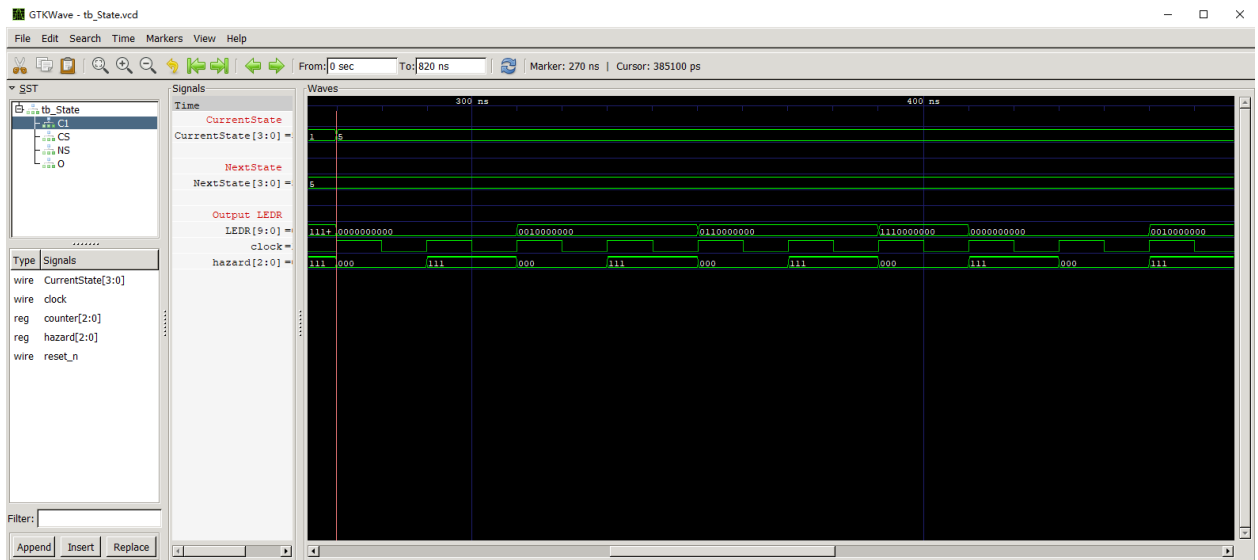


Figure 7. Left Turn Light (State = 5)

The Left Turn light will turn on when SW[1] = 1. And the LEDR[9:7] will blink in the order like this : 000, 001, 011, 111.

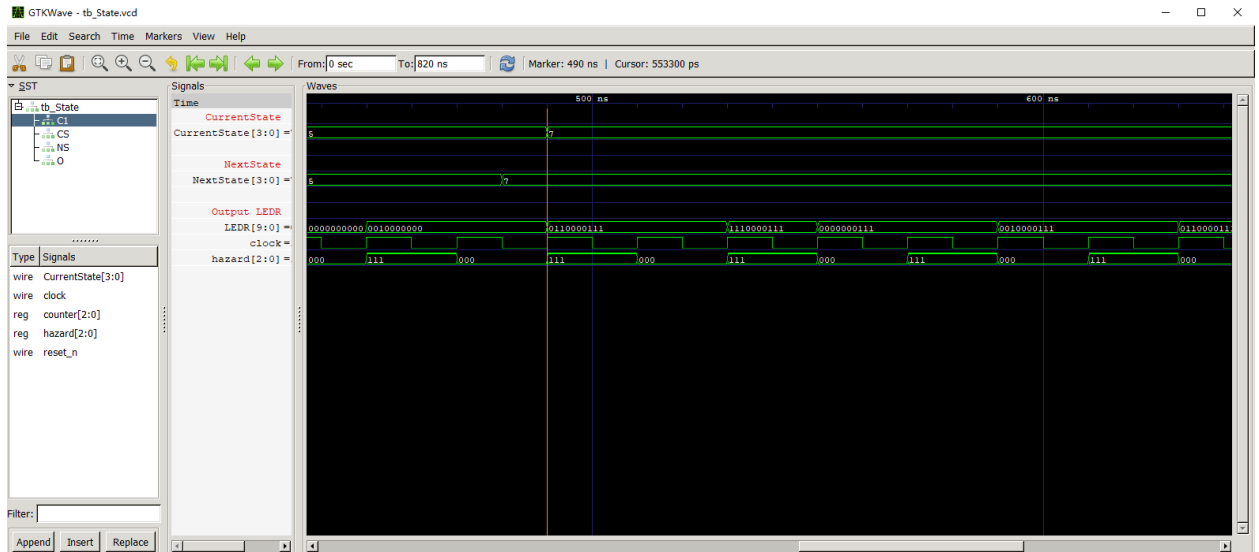


Figure 10. Brake_Left Light (State = 7)

The Brake_Left Light turns on when the KEY[2;1] = 1. The LEDR[9:7] works the same as Left_Turn light and the LEDR[2:0] stays on for the whole period.

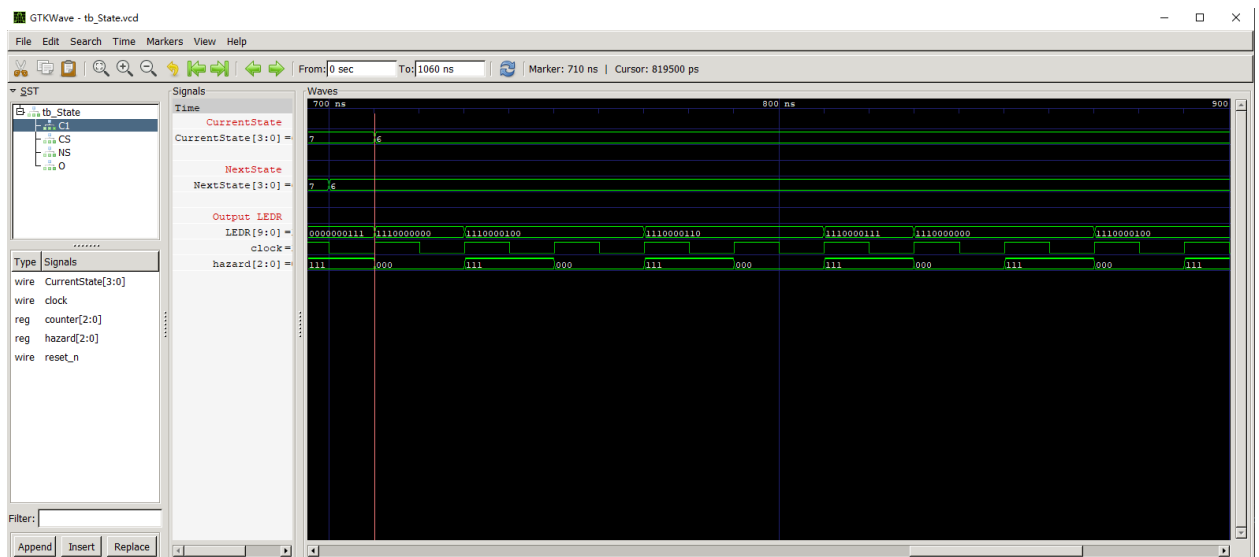


Figure 11. Brake_Right Light (State = 6)

The Brake_Right Light turns on when the KEY[2;1] = 1 and KEY[1] is pressed. The LEDR[2:0] works the same as Right_Turn light and the LEDR[9:7] stays on for the whole period.

6. Summary

Personally speaking, this project is really fun because we designed the memory block to make the automation mode without manually controlling the SW. Generally, I finish all the requirements of this project. Moreover, I have a better understanding of

Finite State Machine after I finished this project. I think FSM is also pretty useful in my other classes because it makes the code and the logic more neat.

At the point when I try to test my automation in 0.2 Hz, I noticed that the older clock divider does not have enough capability to create 0.2 Hz clock so that my automation does not cycle through the state. After I check with the Prof. Robinson, I update the width of my clock_divider, so it could create the 0.2 Hz clock to run my automation at the end.