Gabriel Emerson
ELEC 4200 – Lab 5
09/23/21

Lab 5: Finite State Machines

Goals:

The goal for this lab is to become familiar with finite state machines. This mainly includes a couple different types of finite state machines, and why they are better than the other for different purposes. We also continue this lab with writing our own testbench files for each task which will verify our program when running it in simulation. This is the first lab where we have to write all of our main program and also write our own testbench.

Design Process:

Task 1 is a straightforward creation of the Mealy Finite State Machine. This means our machine output will depend on both current state and current input. We can implement this by viewing our PreLab task of drawing a state diagram of the machine, and then putting the code into 3 different always blocks. This is how we will proceed with designing of all the Finite State Machines. The task 1 block diagram is shown in Figure 1 and the desired output of task 1 simulation is shown in Figure 2.

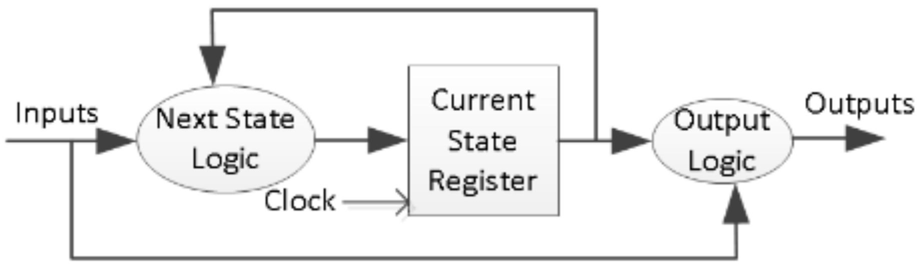


Figure 1

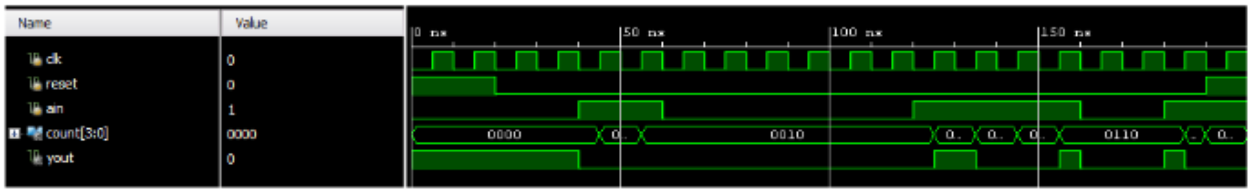


Figure 2

Task 2 is similar to task 1 except we use the design of a Moore machine instead of the previous Mealy machine. We also are trying to only get a high output after specific series of inputs as defined by table 1. Our block diagram of the Moore machine is shown in Figure 3 and the expected output for the task is shown in Figure 4.

Series Input Value	**Output Value**
01 → 00	0
11 → 00	1
10 → 00	Toggle

Table 1

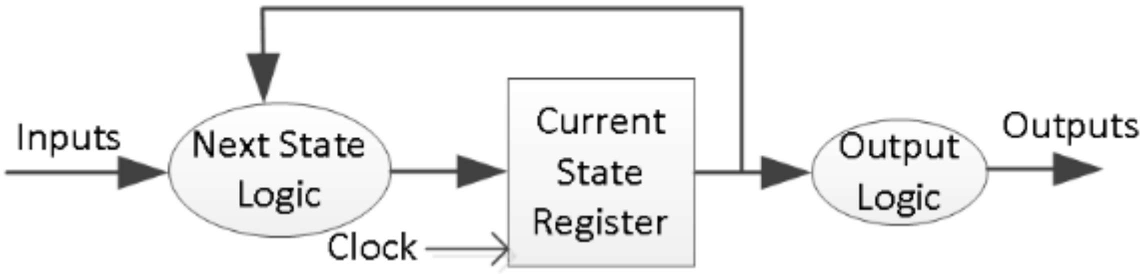


Figure 3

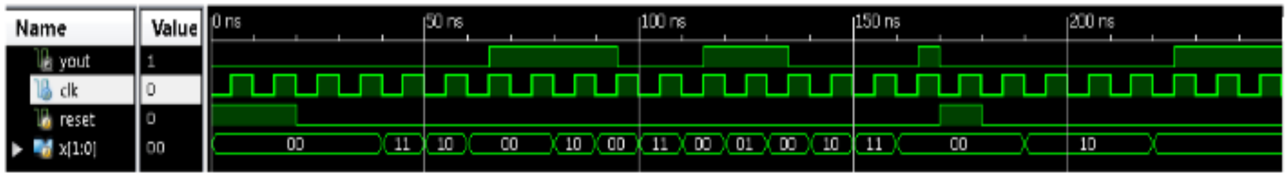


Figure 4

Task 3 introduces using Finite State Machines with a ROM memory file. In this task we are asked to use a Mealy machine with a ROM fle. This is used to count in a specific series not normal counting. This program should count 000, 001, 011, 101, 111, 010 and then repeat starting back at 000. This means we need a counter to show which spot we are at in the ROM file and also the value of output that we should be showing to the user. The diagram for the Mealy with ROM is shown in Figure 5 and the expected output is shown in Figure 6.

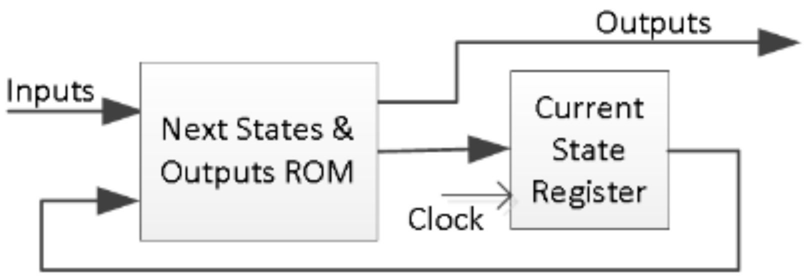


Figure 5

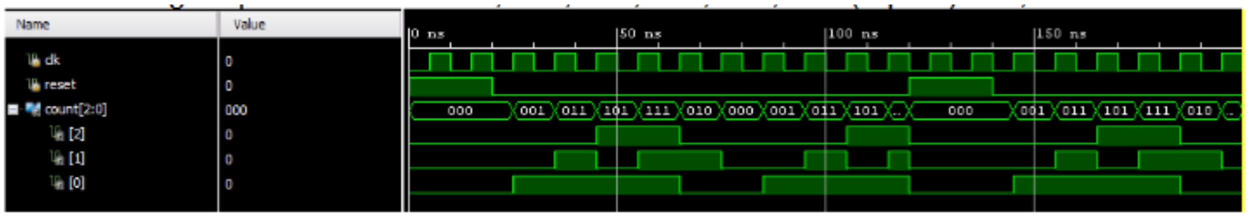
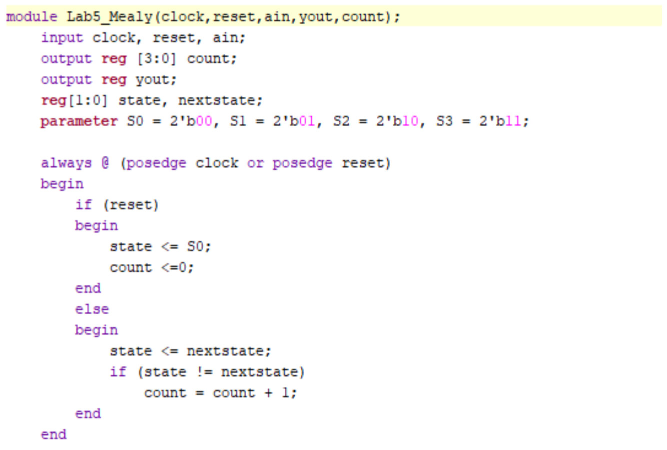


Figure 6

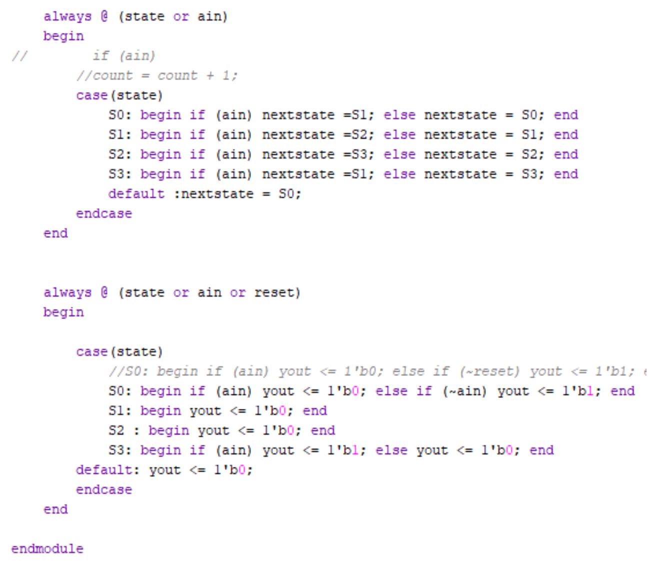
Detailed Design:

Task 1 was designing a Mealy state machine according to the state diagram we drew in the Prelab. This means we want our output to only be 1 if we have received a sequence of 1's that is divisible by 3. The design of this circuit was not too difficult if the state diagram was done correctly. We were also tasked with building the testbench for each task. Due to our code being too long, we could not get it to fit into one image. The code for task 1 is shown in Figures 7 & 8, and the testbench is shown in Figures 9 & 10.



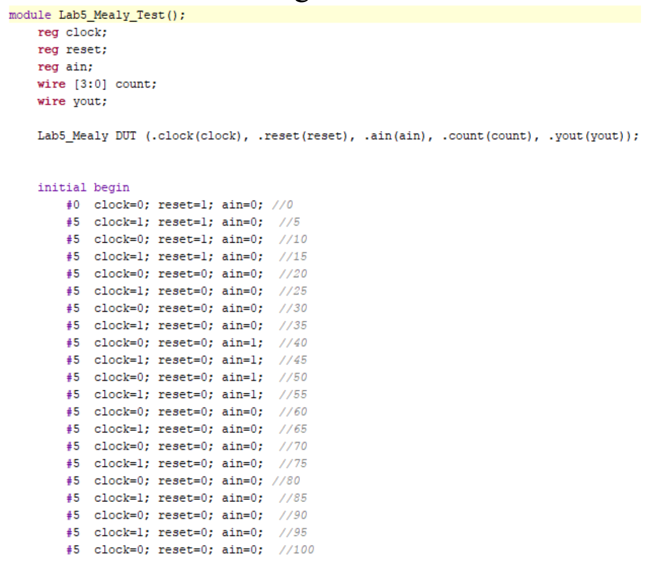
```
module Lab5_Mealy(clock,reset,ain,yout,count);
    input clock, reset, ain;
    output reg [3:0] count;
    output reg yout;
    reg[1:0] state, nextstate;
    parameter S0 = 2'b00, S1 = 2'b01, S2 = 2'b10, S3 = 2'b11;

    always @ (posedge clock or posedge reset)
    begin
        if (reset)
        begin
            state <= S0;
            count <=0;
        end
        else
        begin
            state <= nextstate;
            if (state != nextstate)
                count = count + 1;
        end
    end
```

Figure 7



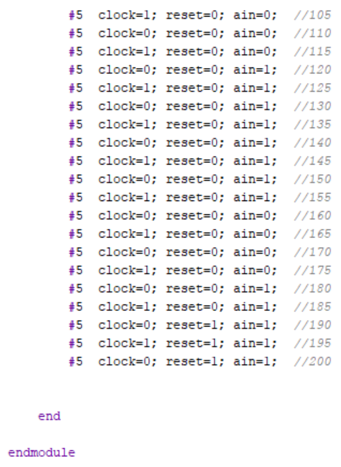
```
    always @ (state or ain)
    begin
//          if (ain)
        //count = count + 1;
        case(state)
            S0: begin if (ain) nextstate =S1; else nextstate = S0; end
            S1: begin if (ain) nextstate =S2; else nextstate = S1; end
            S2: begin if (ain) nextstate =S3; else nextstate = S2; end
            S3: begin if (ain) nextstate =S1; else nextstate = S3; end
            default :nextstate = S0;
        endcase
    end


    always @ (state or ain or reset)
    begin

        case(state)
            //S0: begin if (ain) yout <= 1'b0; else if (~reset) yout <= 1'b1;
            S0: begin if (ain) yout <= 1'b0; else if (~ain) yout <= 1'b1; end
            S1: begin yout <= 1'b0; end
            S2 : begin yout <= 1'b0; end
            S3: begin if (ain) yout <= 1'b1; else yout <= 1'b0; end
        default: yout <= 1'b0;
        endcase
    end

endmodule
```

Figure 8



```
module Lab5_Mealy_Test();
    reg clock;
    reg reset;
    reg ain;
    wire [3:0] count;
    wire yout;

    Lab5_Mealy DUT (.clock(clock), .reset(reset), .ain(ain), .count(count), .yout(yout));


    initial begin
        #0  clock=0; reset=1; ain=0; //0
        #5  clock=1; reset=1; ain=0;  //5
        #5  clock=0; reset=1; ain=0;  //10
        #5  clock=1; reset=1; ain=0;  //15
        #5  clock=0; reset=0; ain=0;  //20
        #5  clock=1; reset=0; ain=0;  //25
        #5  clock=0; reset=0; ain=0;  //30
        #5  clock=1; reset=0; ain=0;  //35
        #5  clock=0; reset=0; ain=1;  //40
        #5  clock=1; reset=0; ain=1;  //45
        #5  clock=0; reset=0; ain=1;  //50
        #5  clock=1; reset=0; ain=1;  //55
        #5  clock=0; reset=0; ain=0;  //60
        #5  clock=1; reset=0; ain=0;  //65
        #5  clock=0; reset=0; ain=0;  //70
        #5  clock=1; reset=0; ain=0;  //75
        #5  clock=0; reset=0; ain=0; //80
        #5  clock=1; reset=0; ain=0;  //85
        #5  clock=0; reset=0; ain=0;  //90
        #5  clock=1; reset=0; ain=0;  //95
        #5  clock=0; reset=0; ain=0;  //100
```

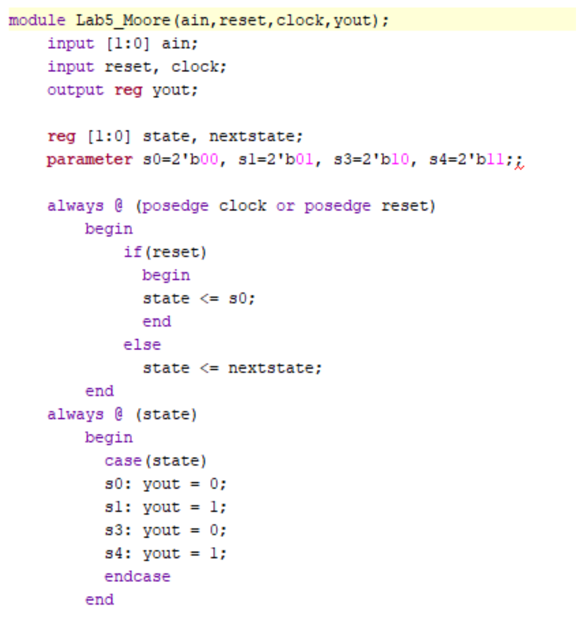
Figure 9



```
        #5  clock=1; reset=0; ain=0;  //105
        #5  clock=0; reset=0; ain=0;  //110
        #5  clock=1; reset=0; ain=0;  //115
        #5  clock=0; reset=0; ain=1;  //120
        #5  clock=1; reset=0; ain=1;  //125
        #5  clock=0; reset=0; ain=1;  //130
        #5  clock=1; reset=0; ain=1;  //135
        #5  clock=0; reset=0; ain=1;  //140
        #5  clock=1; reset=0; ain=1;  //145
        #5  clock=0; reset=0; ain=1;  //150
        #5  clock=1; reset=0; ain=1;  //155
        #5  clock=0; reset=0; ain=0;  //160
        #5  clock=1; reset=0; ain=0;  //165
        #5  clock=0; reset=0; ain=0;  //170
        #5  clock=1; reset=0; ain=0;  //175
        #5  clock=0; reset=0; ain=1;  //180
        #5  clock=1; reset=0; ain=1;  //185
        #5  clock=0; reset=1; ain=1;  //190
        #5  clock=1; reset=1; ain=1;  //195
        #5  clock=0; reset=1; ain=1;  //200


    end

endmodule
```

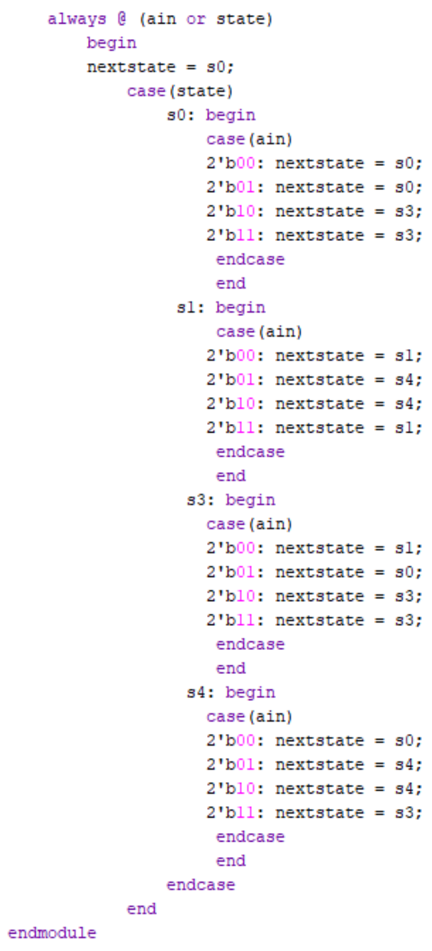
Figure 10

Task 2 design was similar but instead used a Moore machine to change outputs only after a specific sequence of inputs. The state diagram from PreLab was drawn following the outputs of Table 1 above. Then we used this state diagram to design a simple (but long) program that follows input sequences to get the right output. The code is shown in Figures 11 & 12, and the testbenches designed are shown in the Figures 13 & 14.



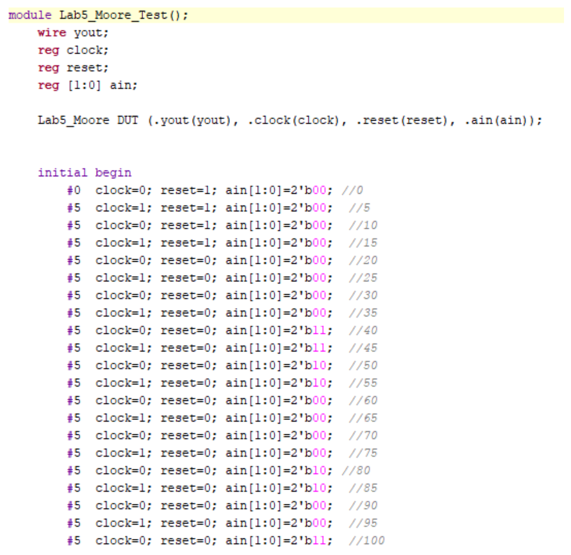
```
module Lab5_Moore(ain,reset,clock,yout);
    input [1:0] ain;
    input reset, clock;
    output reg yout;

    reg [1:0] state, nextstate;
    parameter s0=2'b00, s1=2'b01, s3=2'b10, s4=2'b11;;

    always @ (posedge clock or posedge reset)
        begin
            if(reset)
              begin
              state <= s0;
              end
            else
              state <= nextstate;
        end
    always @ (state)
        begin
          case(state)
          s0: yout = 0;
          s1: yout = 1;
          s3: yout = 0;
          s4: yout = 1;
          endcase
        end
```

Figure 11



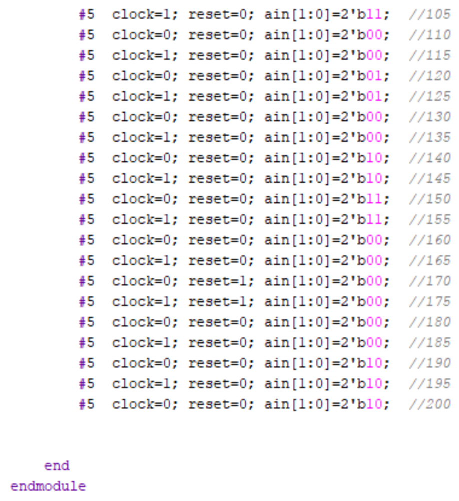
```
    always @ (ain or state)
        begin
        nextstate = s0;
            case(state)
                s0: begin
                    case(ain)
                    2'b00: nextstate = s0;
                    2'b01: nextstate = s0;
                    2'b10: nextstate = s3;
                    2'b11: nextstate = s3;
                     endcase
                     end
                 s1: begin
                     case(ain)
                    2'b00: nextstate = s1;
                    2'b01: nextstate = s4;
                    2'b10: nextstate = s4;
                    2'b11: nextstate = s1;
                     endcase
                     end
                  s3: begin
                    case(ain)
                    2'b00: nextstate = s1;
                    2'b01: nextstate = s0;
                    2'b10: nextstate = s3;
                    2'b11: nextstate = s3;
                     endcase
                     end
                  s4: begin
                    case(ain)
                    2'b00: nextstate = s0;
                    2'b01: nextstate = s4;
                    2'b10: nextstate = s4;
                    2'b11: nextstate = s3;
                     endcase
                     end
                endcase
            end
endmodule
```

Figure 12



```
module Lab5_Moore_Test();
    wire yout;
    reg clock;
    reg reset;
    reg [1:0] ain;

    Lab5_Moore DUT (.yout(yout), .clock(clock), .reset(reset), .ain(ain));


    initial begin
        #0  clock=0; reset=1; ain[1:0]=2'b00; //0
        #5  clock=1; reset=1; ain[1:0]=2'b00;  //5
        #5  clock=0; reset=1; ain[1:0]=2'b00;  //10
        #5  clock=1; reset=1; ain[1:0]=2'b00;  //15
        #5  clock=0; reset=0; ain[1:0]=2'b00;  //20
        #5  clock=1; reset=0; ain[1:0]=2'b00;  //25
        #5  clock=0; reset=0; ain[1:0]=2'b00;  //30
        #5  clock=1; reset=0; ain[1:0]=2'b00;  //35
        #5  clock=0; reset=0; ain[1:0]=2'b11;  //40
        #5  clock=1; reset=0; ain[1:0]=2'b11;  //45
        #5  clock=0; reset=0; ain[1:0]=2'b10;  //50
        #5  clock=1; reset=0; ain[1:0]=2'b10;  //55
        #5  clock=0; reset=0; ain[1:0]=2'b00;  //60
        #5  clock=1; reset=0; ain[1:0]=2'b00;  //65
        #5  clock=0; reset=0; ain[1:0]=2'b00;  //70
        #5  clock=1; reset=0; ain[1:0]=2'b00;  //75
        #5  clock=0; reset=0; ain[1:0]=2'b10; //80
        #5  clock=1; reset=0; ain[1:0]=2'b10;  //85
        #5  clock=0; reset=0; ain[1:0]=2'b00;  //90
        #5  clock=1; reset=0; ain[1:0]=2'b00;  //95
        #5  clock=0; reset=0; ain[1:0]=2'b11;  //100
```

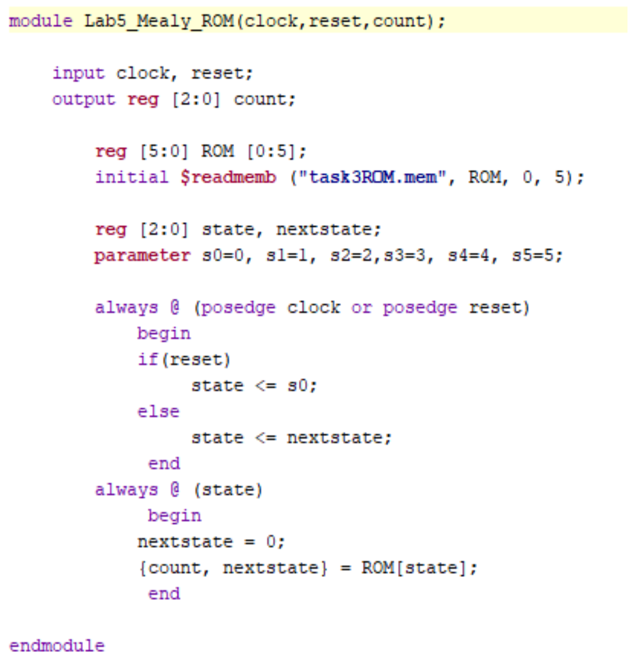
Figure 13



```
        #5  clock=1; reset=0; ain[1:0]=2'b11;  //105
        #5  clock=0; reset=0; ain[1:0]=2'b00;  //110
        #5  clock=1; reset=0; ain[1:0]=2'b00;  //115
        #5  clock=0; reset=0; ain[1:0]=2'b01;  //120
        #5  clock=1; reset=0; ain[1:0]=2'b01;  //125
        #5  clock=0; reset=0; ain[1:0]=2'b00;  //130
        #5  clock=1; reset=0; ain[1:0]=2'b00;  //135
        #5  clock=0; reset=0; ain[1:0]=2'b10;  //140
        #5  clock=1; reset=0; ain[1:0]=2'b10;  //145
        #5  clock=0; reset=0; ain[1:0]=2'b11;  //150
        #5  clock=1; reset=0; ain[1:0]=2'b11;  //155
        #5  clock=0; reset=0; ain[1:0]=2'b00;  //160
        #5  clock=1; reset=0; ain[1:0]=2'b00;  //165
        #5  clock=0; reset=1; ain[1:0]=2'b00;  //170
        #5  clock=1; reset=1; ain[1:0]=2'b00;  //175
        #5  clock=0; reset=0; ain[1:0]=2'b00;  //180
        #5  clock=1; reset=0; ain[1:0]=2'b00;  //185
        #5  clock=0; reset=0; ain[1:0]=2'b10;  //190
        #5  clock=1; reset=0; ain[1:0]=2'b10;  //195
        #5  clock=0; reset=0; ain[1:0]=2'b10;  //200


    end
endmodule
```

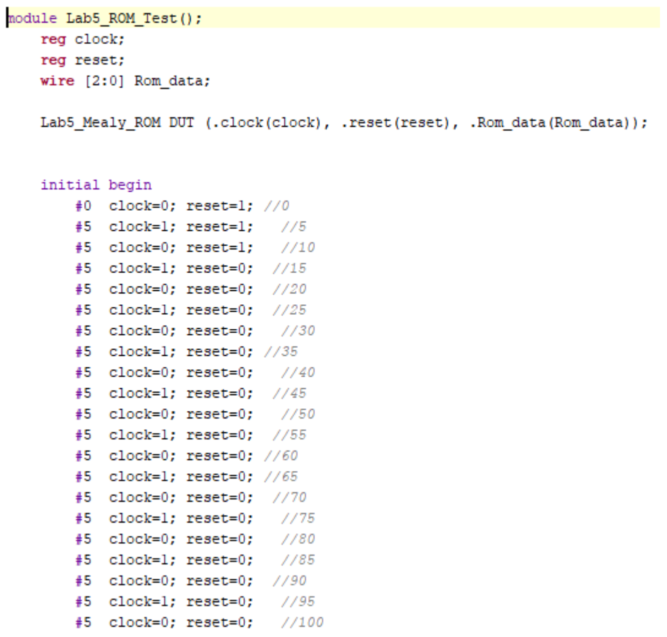
Figure 14

Task 3 also used a Mealy machine but instead of using the same format we use a ROM memory file to input what the count sequence to be. We still use the always blocks, but instead of creating our own next state, the nextstate and output are stored in the memory file. The code is shown in Figure 15, the testbench is shown in Figures 16 & 17, and the memory file is shown in Figure 18.



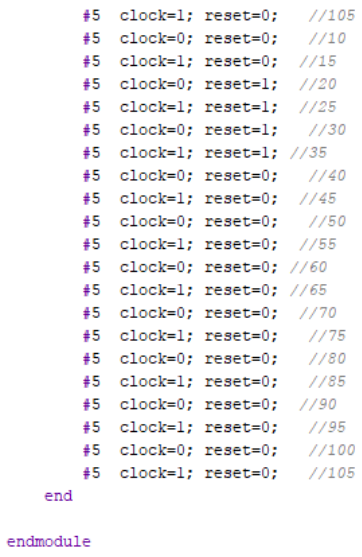
```
module Lab5_Mealy_ROM(clock,reset,count);

    input clock, reset;
    output reg [2:0] count;

        reg [5:0] ROM [0:5];
        initial $readmemb ("task3ROM.mem", ROM, 0, 5);

        reg [2:0] state, nextstate;
        parameter s0=0, s1=1, s2=2,s3=3, s4=4, s5=5;

        always @ (posedge clock or posedge reset)
            begin
            if(reset)
                 state <= s0;
            else
                 state <= nextstate;
             end
        always @ (state)
             begin
            nextstate = 0;
            {count, nextstate} = ROM[state];
             end

endmodule
```

Figure 15



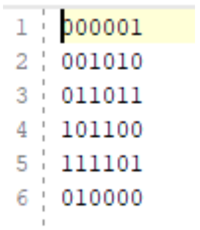
```
module Lab5_ROM_Test();
    reg clock;
    reg reset;
    wire [2:0] Rom_data;

    Lab5_Mealy_ROM DUT (.clock(clock), .reset(reset), .Rom_data(Rom_data));


    initial begin
        #0  clock=0; reset=1; //0
        #5  clock=1; reset=1;   //5
        #5  clock=0; reset=1;   //10
        #5  clock=1; reset=0;  //15
        #5  clock=0; reset=0;  //20
        #5  clock=1; reset=0;  //25
        #5  clock=0; reset=0;   //30
        #5  clock=1; reset=0; //35
        #5  clock=0; reset=0;   //40
        #5  clock=1; reset=0;  //45
        #5  clock=0; reset=0;   //50
        #5  clock=1; reset=0;  //55
        #5  clock=0; reset=0; //60
        #5  clock=1; reset=0; //65
        #5  clock=0; reset=0;  //70
        #5  clock=1; reset=0;   //75
        #5  clock=0; reset=0;   //80
        #5  clock=1; reset=0;   //85
        #5  clock=0; reset=0;  //90
        #5  clock=1; reset=0;   //95
        #5  clock=0; reset=0;   //100
```

Figure 16



```
        #5  clock=1; reset=0;   //105
        #5  clock=0; reset=0;   //10
        #5  clock=1; reset=0;  //15
        #5  clock=0; reset=1;  //20
        #5  clock=1; reset=1;  //25
        #5  clock=0; reset=1;   //30
        #5  clock=1; reset=1; //35
        #5  clock=0; reset=0;   //40
        #5  clock=1; reset=0;  //45
        #5  clock=0; reset=0;   //50
        #5  clock=1; reset=0;  //55
        #5  clock=0; reset=0; //60
        #5  clock=1; reset=0; //65
        #5  clock=0; reset=0;  //70
        #5  clock=1; reset=0;   //75
        #5  clock=0; reset=0;   //80
        #5  clock=1; reset=0;   //85
        #5  clock=0; reset=0;  //90
        #5  clock=1; reset=0;   //95
        #5  clock=0; reset=0;   //100
        #5  clock=1; reset=0;   //105
    end

endmodule
```

Figure 17



```
000001
001010
011011
101100
111101
010000
```

Figure 18

Verification:

Each task is verified by using both a Simulation (using our own testbench) and uploading it to the board and visually verifying the functionality of the task. All tasks included an expected output for the simulation to try to replicate.

Task 1 can be verified after our simulation matches that of the expected results. Then we upload to the board and view if our results are only 1 if our inputs received a sequence of 1's that are divisible by 3. The simulation output is shown in Figure 19.

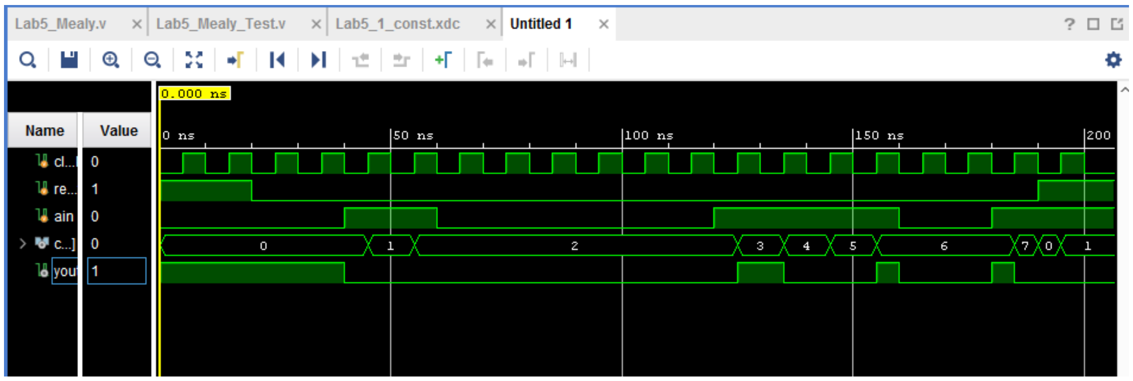


Figure 19

Task 2 can be verified by checking our simulation with the expected results simulation given to us in the lab writeup. We then upload our program to the board and visually check if the input sequence changes the output according to Table 1. Our simulation results are shown in Figure 20.

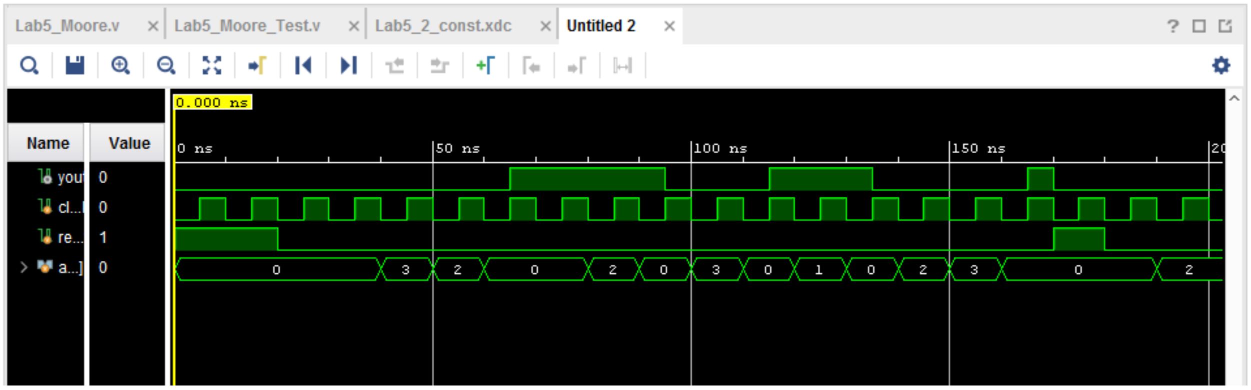


Figure 20

Task 3 can also be verified by simply checking our simulation with the expected results given to us in the writeup. After we verify this works, we upload the program to the board and visually verify that our program is counting correctly and is at the specific number shown in the ROM file. Our simulation results are shown in Figure 21.

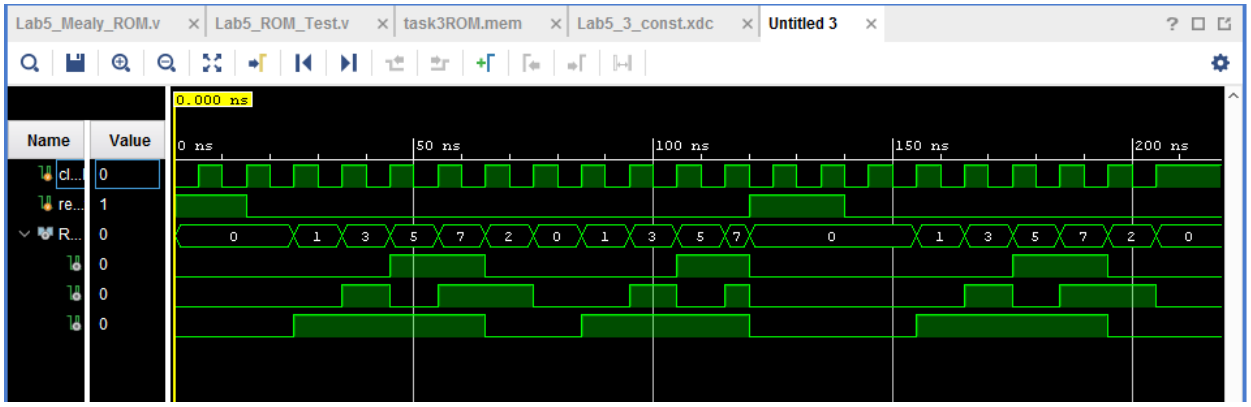


Figure 21

Conclusion:

What I learned in this lab is different methods of implementing Mealy and Moore Finite State Machines. We designed and implemented sequence detector, a sequence generator, and code converters using the two and three always blocks styles. I also reenforced the ideas behind using ROM files to write specific outputs on the board, and writing our own testbenches to simulate our original program directly. This lab went much better than last since I was able to finish in the correct time.