LAB 4

Finite State Machines

Team 6

鄭聰明 **Jason Theodorus Pratama (109006236)**

鐵特德 **Tuguldur Tserenbaljir (109006271)**

TABLE OF CONTENTS

I. Content Addressable Memory (CAM)

A. Design

At first we did not know much about the Content Addressable Memory, but after researching about it for a while, We found out that both address and content is stored side by side. Thus, the data stored in the stored data line is n sets of m-bit data where in our case it is n-16 and m =8;

Following the slides, we design out CAM with 5 inputs, which is clk, ren,wen din and addr. And the outputs are dout and hit. While the inputs of din is m (8 bit) and our addr is 4-bit, our output will be 4-bit.

Where the inputs are called into the sequential block, and for the output especially **hit** will go out from the sequential block, while **dout** goes from the Priority Encoder. And the din that was inputted from the sequential block goes to the Comparator Block or in our design is the Combinational Block, then output the value of compare which is assigned to the address of the CAM, so if its the same it will show the output of compare as 1, then goes to the priority encoder to print out the dout or the **priorityaddress**

As for the priority encoder the input is compare and outputs match, and dout. Where if the Comparator Array matches with the CAM[Address] it will output a match signal to the sequential block and output hit, as the prio address is outputted from the priority encoder.

The Sequential Block of CAM :

```
if (ren == 1'b1) begin // Ren = 1 or Ren Wen = 1
   if(match) begin
      dout <= prioaddress;
      hit <= 1'b1;
   end
   else begin
      dout <= 4'b0;
      hit <= 1'b0;
   end
end
else begin
   if(wen == 1'b1) begin // Wen = 1
      CAM[addr] <= din;
```

```
    dout <= 4'b0;
    hit <= 1'b0;
  end
  else begin // Ren = 0 , Wen = 0
    dout <= 4'b0;
    hit <= 1'b0;
      end
    end
  end
```

Since if Ren = 1 and Wen = 1, it just want to be ren = 1, so we made a nested if else statement like above, because if we make a separate ren = 1, wen = 1 if statement. Logically, it would be mindblending and may cause bugs because you can't just set the wen to 0 if ren = 1, wen = 1.

As for the comparator :

```
always @(*)begin
  if (din == CAM[0])begin
    compare[0] = 1'b1;
  end
  else begin
    compare[0] = 1'b0;
  end
//…..
  if (din == CAM[1])begin
    compare[15] = 1'b1;
  end
  else begin
    compare[15] = 1'b0;
   End
End
```

Where this compares the value of din and CAM[addr], previously this was in a for loop, but when scrolling through discussion, for loops might cause problems in designing circuits, so we elaborated it.

Previous comparator with For Loop :

```
always @(*)begin
 For (i = 0 ; i < n ; i = i + 1) begin
        If (din == CAM[i] ) begin
                Compare[i] = 1'b1;
```

```
        end
        Else begin
                Compare[i] = 1’b0;
        end
end
```

Then Finally for Priority Encoder which is in a separated module :

```
always @(*)begin
   matching = 1'b1;
   if(in[0] == 1'b1) out = 4'b0000 ;
   else if(in[1] == 1'b1) out = 4'b0001;
   else if(in[2] == 1'b1) out = 4'b0010;
          // ….
   else if(in[14] == 1'b1) out = 4'b1110;
   else if(in[15] == 1'b1) out = 4'b1111;
   else begin
   out = 4'bxxxx;
   matching = 1'b0;
   end
end
```

Where if it matches, it will check from the lowest address to the highest. This way it will show the lowest address that matches. This way the data stored on the CAM, can be easily accessed by searching for the content itself, while the memory will retrieve the address of those respective addresses in a single clock cycle.

B. Block Diagram & State Transition

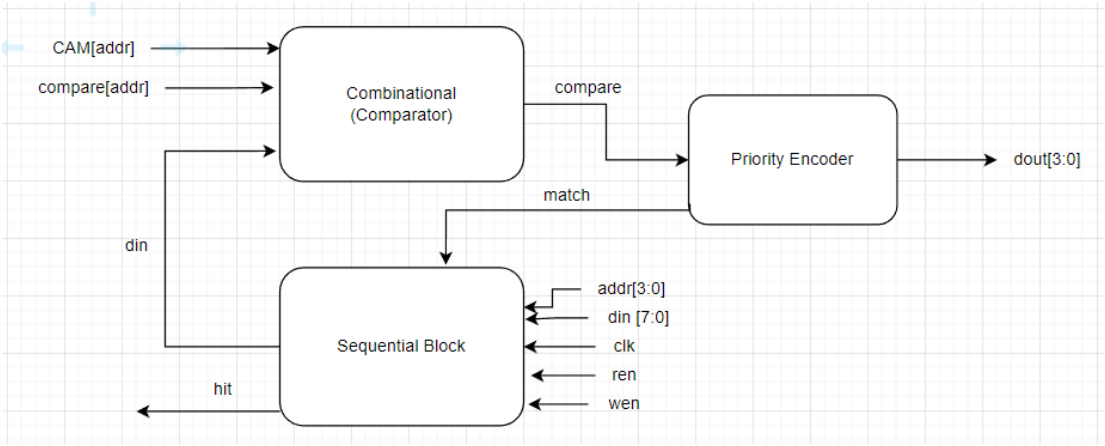


Figure 1.1 Block Diagram of CAM

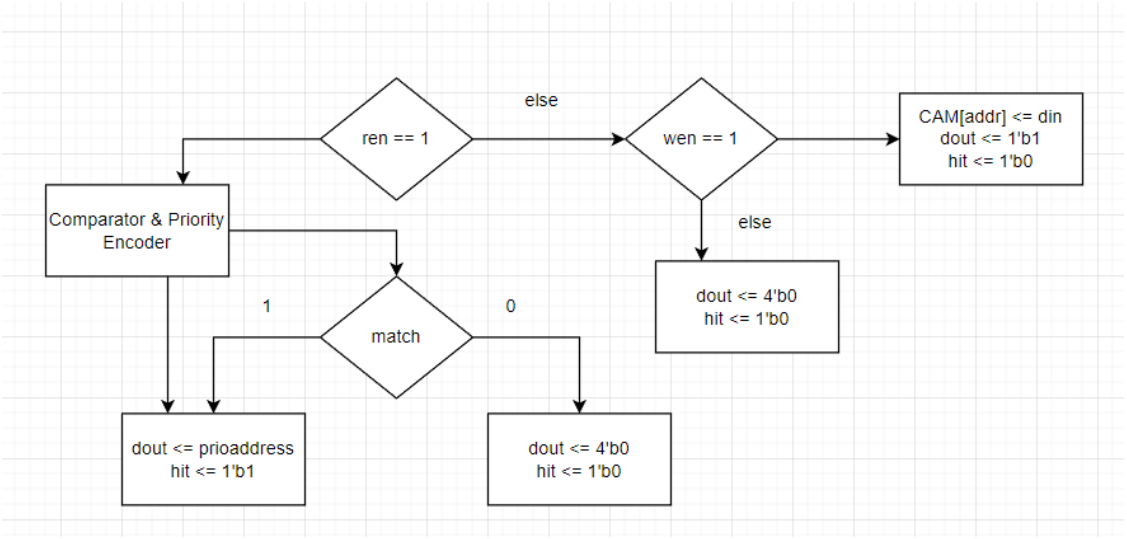
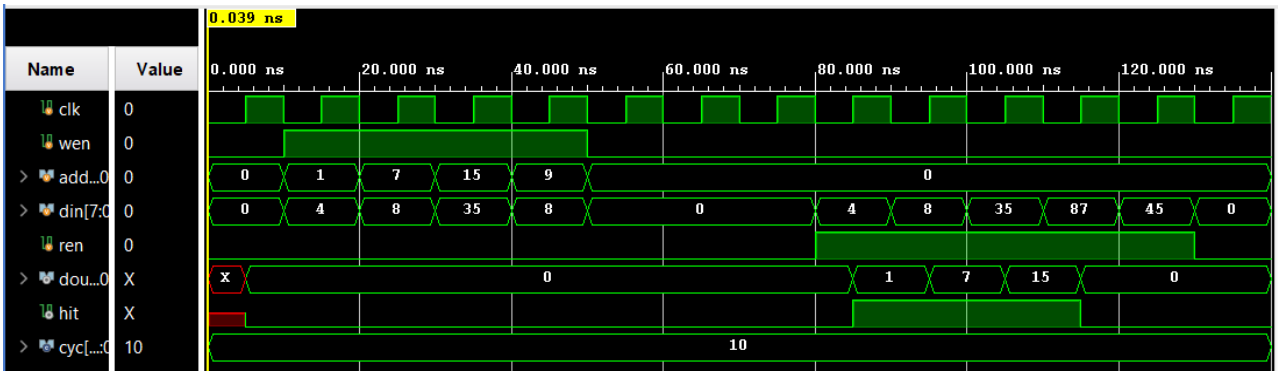


Figure 1.2 State Transition of CAM

C. Waveform & Testing



This is the correct waveform from the testbench that were made from the TA's waveform from the slides, but previously on **Figure 1.3** the dout and hit was delayed by 1 cycle.

The Waveform is like so, where the din 4 is set to 1, din 8 set to 7, din 35 set to 15, and din 8 set to 9. And when it is reading din 4,8,35,87,45,0. It will output hit, and dout 1, as address assigned to 4, and for 8 since there are double addresses (7 and 9) it will output the lowest address from the priority encoder, which is 7, and so on.

D. Problems Faced

The problems we faced was about the clock, where dout and hit was late by 1 clock cycle. The hit and dout was previously assigned on the combinational block, so we tried to put it on the sequential block, but as for the match and the priority address that is supposed to stay on the combinational because it always has to be ready before the data transmission into the sequential, has to stay in combinational, so we used assign to get the data of prio address and match, and connect it from the separate module of **Priority Encoder**

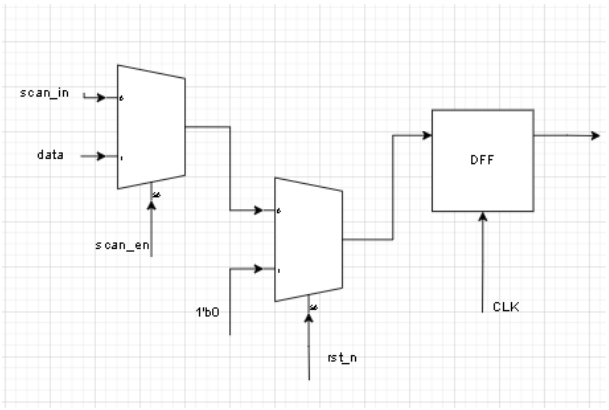
```
wire matchreg;
wire match;
wire [3:0]prioaddressreg;
wire [3:0] prioaddress;
assign prioaddress = prioaddressreg;
assign match = matchreg;
```

Figure 1.3 Manual Assign of Prio Address and match

II. Scan Chain

A. Design

In this problem 2, we have to create an Scan chain where the combinational circuit is 4-bit multiplier which is connected to our 8 DFFs respectively. However, the DFFs are Scan DFFs. So first we design our Scan DFF fist (Figure 2.1).



(Figure 2.1)

Two multiplexer are added to the DFF to make the Scan DFF. While the first multiplexer will act as an functional input for the D, the other one controls the rst_n. It will decide upon to free the DFF or continue the operation. Thus, we connect all of the flops to form a chain where the scan-in port and the last flop is connected to the scan-out port. We connect all of our SDFF (8) to our combinational multiplier (Figure 2.2). During the scan-in, the flows from the output of one flop to the scan-input of the next flip flop. After the sequence is loaded, one clock pulse or the capture phase is initiated. Then the data is shifted out and the signature is compared.

For our design to work, we first assign scan_out to temp[0] (data). For the first phase, we feed the SDFF module the data, scan enable, the next data, p,rst_n and clock.

```
 SDFF S1(temp[7],scan_en,scan_in,p[7],rst_n,clk);
//…
 SDFF S8(temp[0],scan_en,temp[1],p[0],rst_n,clk);
```

First, we check if the rst_n is 1’b0 or 1’b1, if it is 1’b0 we reset the all SDFFs. Otherwise we continue and check the scanned enabled or not. From this, we test if we are in our first phase to scan or the second phase capturing. If the scan is enabled we load the data into the DFF first. Otherwise feed the next data to the respective DFF.

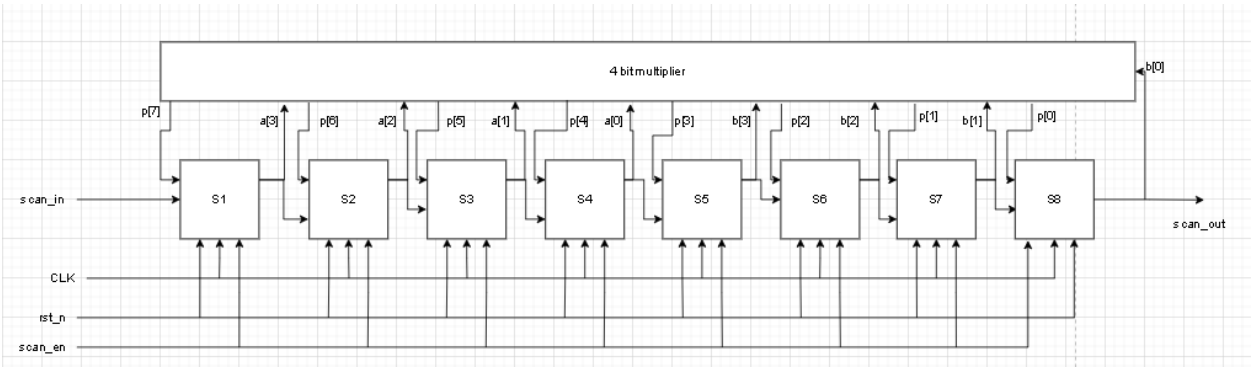
```
    if(rst_n == 1'b0)begin
        DFF <= 1'b0;
    end else begin
        if(scan_en == 1'b1)begin
            DFF<=scan_in;
        end
```

```
        else begin
            DFF <= data;
        end
    end
```

This wat the scan chain will act as a shift register where the first flip-flop is connected to the input ports of next flip-flops to come whereas the last flip-flop's scan chain si connected to the scan_out.

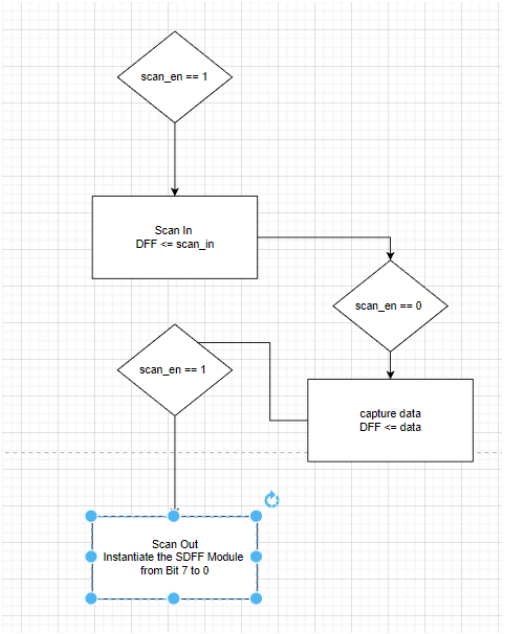
B. Block Diagram & State Transition

After we connect our SDFF to our combinational circuit, our block diagram is illustrated in Figure 2.2 ..



(Figure 2.2)

In Figure 2.3, shows our state transition where the data is shifted bit by bit. .

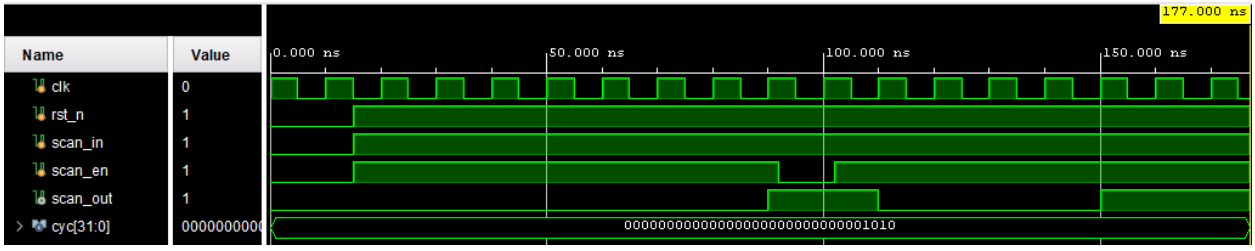


(Figure 2.3)

C. Waveform & Testing

First, we check for the positive edge clock and then start testing the 3 phases. The first phase where it is scanning in when scan is enabled. The second phase capture happens around 100 ns and the last phase scan out is preformed. We set our clock cycle to 10.

After verifying with the Lab 4 slides, all the conditions and double checking our code with our designed circuit from above we conclude our test.



To Test the Scan Chain Design, we used an online Binary Multiplication Tool, to make our jobs easier to read the waveform, so we multiply the first 4 inputs after rst_n with the next 4 inputs so, its 4x4 bit. And after multiplying it we read it from the back

For example :

//TODO//

1111 x 1111 = **011100001**

Then we read it from the back so it becomes 10000111, the last bit isn't accounted for because the output is 8 digits, while the result of multiplier is 9 digits. Output scan_out is the same as the binary multiplication method.

III. Built in Self Test

A. Design

For the built-in self test was bit easier because we reuse our Many-to-One and Scan chain design from the previous questions. We have inputs of clk, rst_n,scan_en and

scan_in, scan_out outputs. First we call our Many-to-One module and give in the variables of clk,rst_n and scan_in. Finally, we call the next module Scan Chain and give the variables clk,rst_n,scan_in,scan_en, and scan_out.

```
    Many_To_One_LFSR M(
        .clk(clk),
        .rst_n(rst_n),
        .out(scan_in)
    );

    Scan_Chain_Design SCD(
        .clk(clk),
        .rst_n(rst_n),
        .scan_in(scan_in),
        .scan_en(scan_en),
        .scan_out(scan_out)
        );
```

These modules are instantiated on the top module which is the Build_In_Self_Test module, which calls the Many_To_One_LFSR and Scan_Chain_Design. Which the value of out, which is the MSB is put into the scan in of the Scan-Chain-Design. During the clock cycle, our scan chain design will also have the three phases of scan_en, capture, scan_out. Much like the question for Scan Chain, the output of multiplier will be stored back to the SDFF's at the positive edge of the clock.

For the LFSR Itself, we modify our LFSR from the basic question so that only the MSB of the LFSR is shifted into the scan chain. When the rst_n is 1'b0, we manually set the value of the DFF to 8'b10111101.

```
module Many_To_One_LFSR(clk, rst_n, out);
always @(posedge clk) begin
     if(rst_n == 1'b0) begin
         DFF[7:0] <= 8'b10111101;
     end else begin
         DFF[7:1] <= DFF[6:0];
```

```
        DFF[0] <= (DFF [1] ^ DFF[2]) ^ (DFF[3] ^ DFF[7]);
    end

end
   assign out = DFF[7];
endmodule
```

We used the module of LFSR from the Many_To_One LFSR, but we modified it so it only outputs the most significant bit which is on DFF[7] .

B. Block Diagram & State Transition

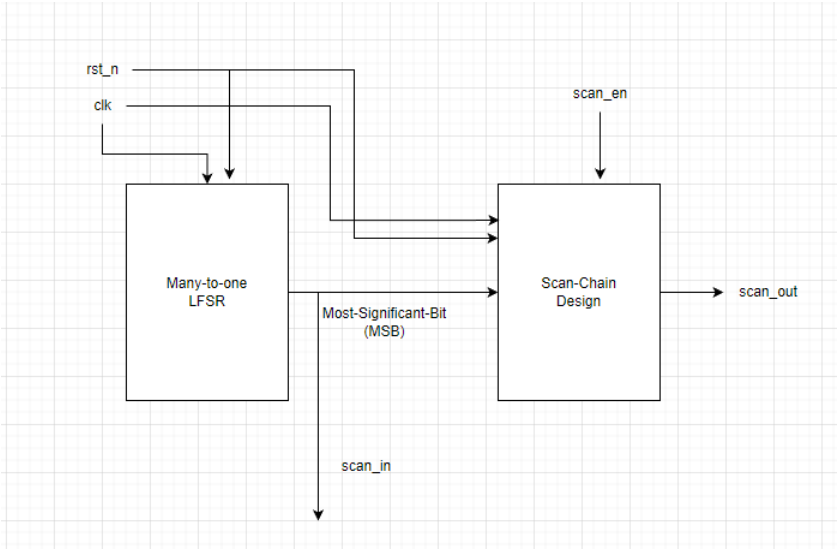


Figure 3.1 Block Diagram of Built in Self Test

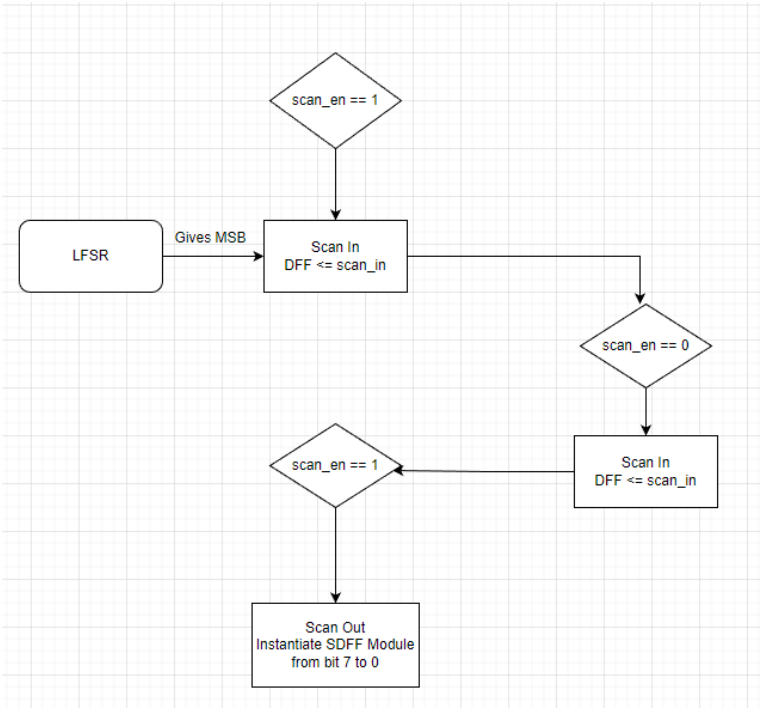
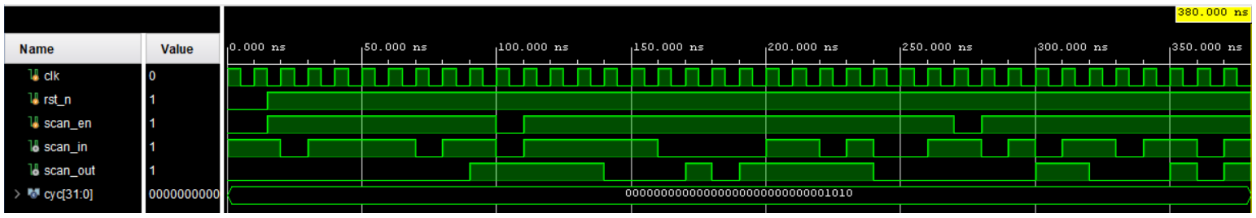


Figure 3.2 State Transition of Built in Self Test

C. Waveform & Testing



The 1st scan_en of wave outputs the initial LFSR value which in the 8 bit many-to-one from basic question 3 is 10111101, and when the scan_en is set to 0, then set to 1 again, it will output the scan_out, which is just like advanced question 2 it is the result of Multiplication, but its inverted. So 1011 x 1101 = **10001111**

Where you read it from the back so the scan_out would be 11110001, which is like the 1st scan_in. Then for the next scan_in, when scan_out is outputting. And goes on.

For the Testbench, we implemented a simple timing so the scan_out will have enough time to output all of its value and scan_in have enough time to input the LFSR value too. So it would look like this

```
  initial begin
     clk = 1'b1;
     rst_n = 1'b0;
     scan_en = 1'd0;
     #(cyc)
     @(negedge clk) begin
        rst_n = 1'd1;
        scan_en = 1'd1;
     end

     //1011 x 1101
  @(negedge) clk  * 8
     @(posedge clk) scan_en = 1'b0;
```

```
    @(posedge clk) scan_en = 1'b1;
    // Scan_Out Start
    @(negedge clk) * 8
    //Scan In Again
  @(negedge clk) * 8
    @(posedge clk) scan_en = 1'b0;
    @(posedge clk) scan_en = 1'b1;
    //Scan_Out Again
       //then wait for scan_out to finish
    #100 $finish;
```

IV. Mealy Sequence Detector

A. Design

In Problem 4, we have to detect sequence 1011, 1100,and 0111. Since we are using mealy machine and the output of the mealy machine depends on the **state** and the **input**. The design is made similarly where the output or in this case **decc** where it will be as **dec** is assigned on the combinational block instead of sequential.

Here is the Table for my Case Statements :

State	**Next_State / In = 1**	**Next_State / In = 0**	**Dec / In = 1**	**Dec / In = 0**
0S0	S1	S4	0	0
S1	S7	S2	0	0
S2	S3	WS2	0	0
S3	S0	S0	1	0
S4	S5	WS1	0	0
S5	S6	WS2	0	0
S6	S0	S0	1	0

S7	WS2	S8	0	0
S8	S0	S0	0	1
WS1	WS2	WS2	0	0
WS2	S0	S0	0	0

So since the Mealy Sequence Detector, will reset its function for every 4 bit. We would want to make it when it has reached the **state of the sequence,** we will come back to **S0** which is **empty/IDLE,** Since the states are impossible to overlap, So i implemented **WS1 and WS2** as a state to direct the wrong **inputs** for it to count to **4** and reset to **S0.**
Where the code is made from the table

Sequential Block of Mealy Sequence Detector :

```
always @(posedge clk) begin
   if (rst_n == 1'b0) begin
      decc <= 1'b0;
      state <= S0;
   end
   else begin
      state <= next_state;
   end
End
```

Where the sequential block captures the state from next state, and when reset is 0, it will set the initial state dec, and state to S0.

Combinational Block of Mealy Sequence Detector

```
always @(*) begin
   case(state)
   S0 : begin
      if (in) begin
```

```
            next_state = S1;
            decc = 1'b0;
        end
        else begin
            next_state = S4;
            decc = 1'b0;
        end
    end
//…
    WS2 : begin
        next_state = S0;
        decc = 1'b0;
    end
    endcase
end
```

The Combinational Block is made from the State Diagram **Figure 4.2** that was drawn, where the beginning state is S0.

B. Block Diagram & State Transition

The Block Diagram of the Mealy Sequence Detector would be like so.

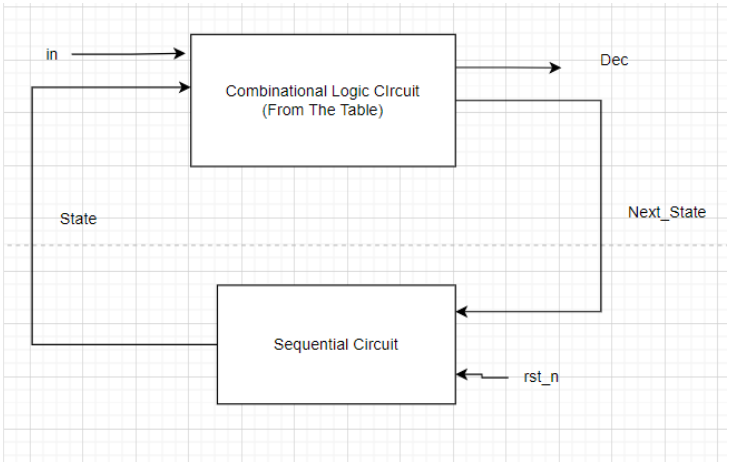


Figure 4.1 Block Diagram of Mealy Sequence Detector

And the State Transition would be like so.

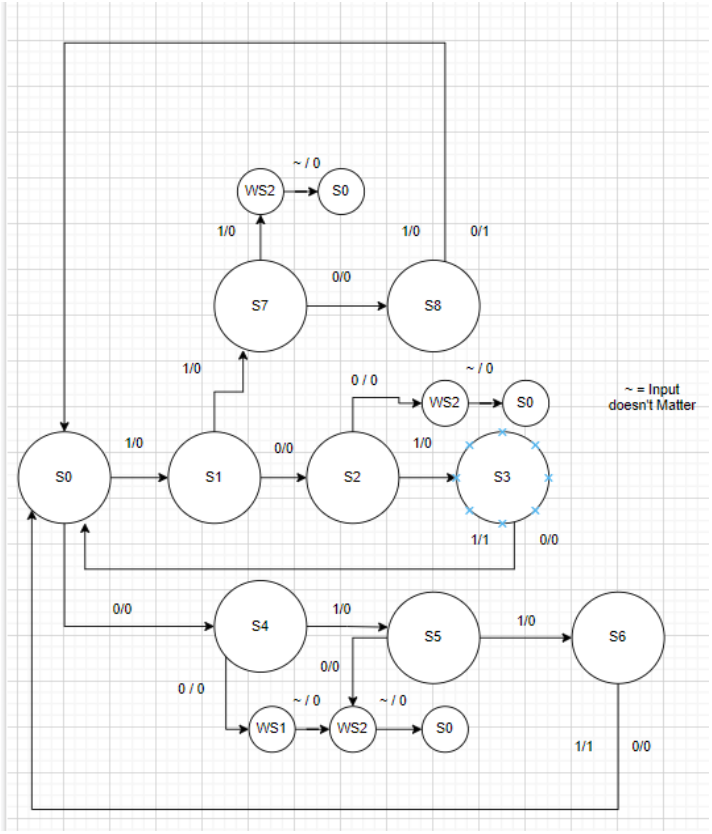
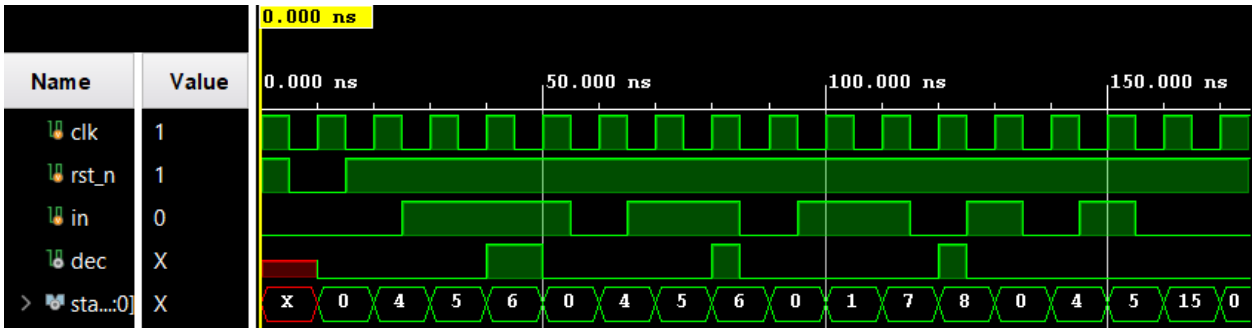


Figure 4.2 State Transition of Mealy Sequence Detector

C. Waveform & Testing

To Check the module, output the states, in order to debug it, because there was a problem where the dec, wasn't outputting on the detected sequence, so i output the state to make it easier to debug. And from this, the module Mealy Sequence Detector has been implemented correctly



As for the testbench, we just use the TA's Waveform on the given slide of Lab 4, a sneak peek of the Testbench :

```
initial begin
   @(negedge clk) rst_n = 1'b0;
   @(negedge clk) rst_n = 1'b1;
   //Given Waveform


   @(negedge clk) in = 1'b1;
   //
   @(negedge clk) in = 1'b0;
   @(negedge clk) in = 1'b1;
   @(negedge clk) in = 1'b0;
   @(negedge clk) in = 1'b0;


   #10 $finish;
end
```

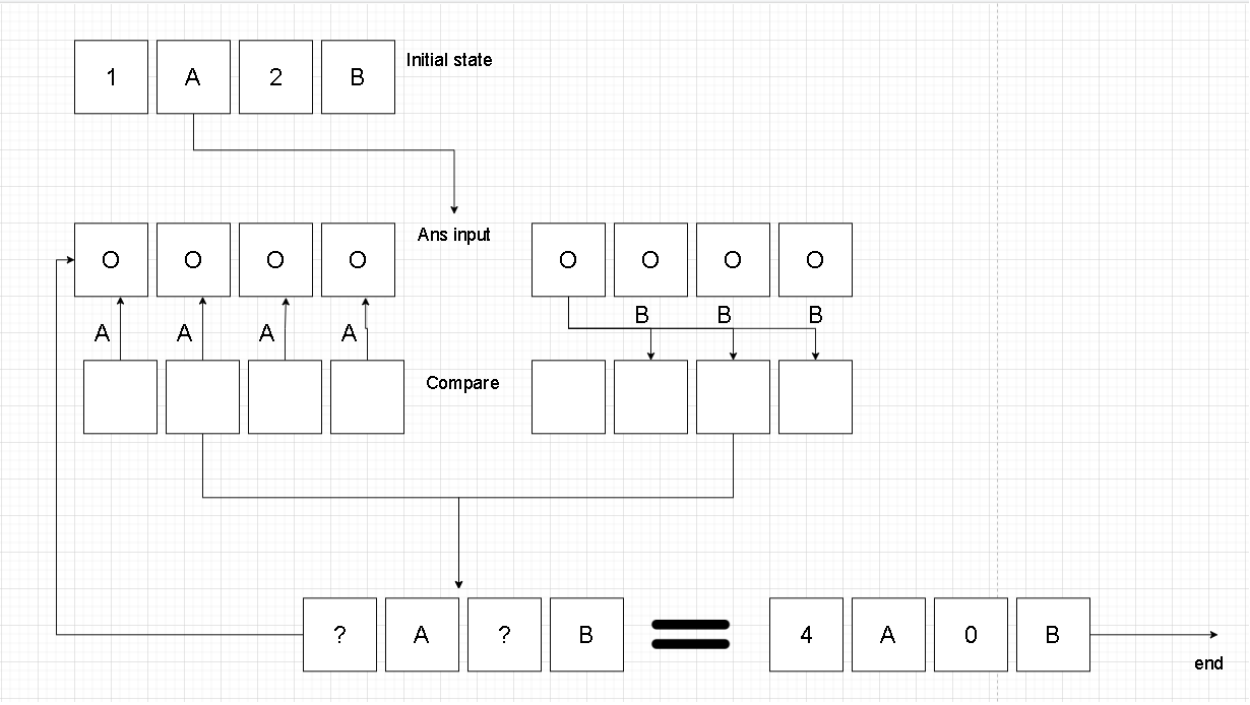
Where the bolded one, is in order to start the detector, otherwise it wont be detecting , since it haven't reset. And the rest are just inputs which is based on the negative edge clock.

Note : the states i made were different, because lack of attention to the slides where on the slide

Where the 1st checking was for the sequence 0111, i made my checking sequence to go 1011 first, just for clarification. **But the dec and waveform should be the same**, because it is a mealy machine, where it depends on the input and the state.

V. FPGA Demonstration

For the FPGA 1A2B game, our initial plan was to create an array where it inputs the 4 numbers from the user and then compares to the randomized numbers from the FPGA, and then print out the correct, wrong place number in the 7 segment. If not “4a0b” where all the answer is correct corresponding to the randomized numbers from the FPGA, it will go back to the user input guessing state (To picture more clearly Figure 5.1).



(Figure 5.1).

To start the coding challenge, we decided to split up the modules into main top module, debounce module, onepulse module, Compare module and LSFR module.

```
module onepulse(pb_debounced, clk, pb_one_pulse);
module debounce(pb_debounced, pb, clk);
module LFSR(clk, rst_n, out3,out2,out1,out0);
```

```
      module Compare(Guess,Answer, answer3,answer2,answer1,answer0,
guess3,guess2,guess1,guess0);
      module top(clk,rst_n,in,start,enter,Anode,Cathode,LED);
```

Firstly, for the Compare module, it gets the inputs of the 4-bit numbers and random numbers. In addition, it's purpose is to compare the numbers to the guessed numbers and give a feedback. In the module , it has two counters for every correct spot (A counter) and every missed placed spot (B counter). Thus, at the very end it adds all of the element of the Counters and gives a sequence to Guessed number and Answer number.

Secondly, for the onepulse and debounce module, it's main purpose is to control the buttons on the 7 segment at the initial, start, and at the end of the game.

Thirdly, for the LFSR module, it takes in the random numbers bit by bit with the clock. At the positive edge of the clock it sets the value of DFF <= 'b1011110110111101. Otherwise it continues to create an random numbers of 1's and 0's to decode later and guess. We use LFSR because it is more cost efficient and we just implemented on the previous questions.

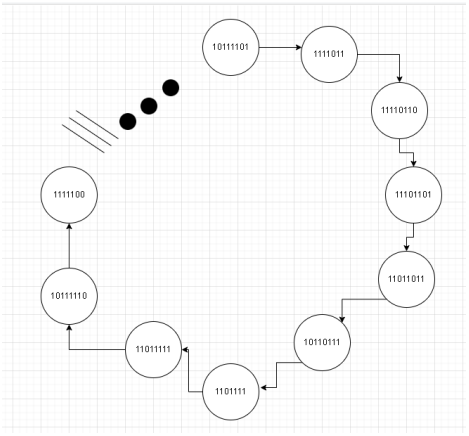
Finally, putting all of modules together in the top module. After we feed out variables to our modules successfully, it time to start the game. Before we start our game, we try to make our last digit or inputting answer digit of the User to flicker every 2.6 milliseconds with a refresh counter. In addition to the flickering clock always block, we also have an another always block. Our intention was to control the LED lights on the 7 segment. In our second always block, we also have two case blocks where on is to activate and deactivate the LEDS, and the other for patterns of the 7-segment LED display (Anode and Cathode baised respectively).

However, due to time constraint we were unable to debug our code to run correctly. Main reason why we had time constraint is that we did not have enough time to sit down and think carefully about project because we had numerous midterms and other projects to do at the time and not enough time to work on the Laboratory 4.

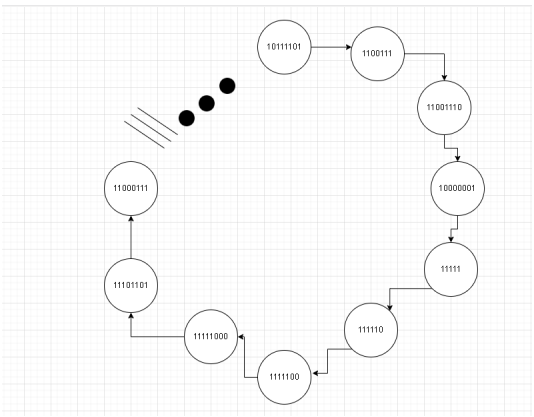
VI. Basic Questions 3 & 4

If we reset the DFFs to 8'd0 in our LFSR, it will create an infinite loop because if all of the inputs are 0's it will shift all 0's, and since our XOR gate logic of two 0's is 0. However, we could force the next cycle after we detect the 8'd0 sequence.

Figure 6.1 demonstrates the first ten states for Many to One whereas Figure 6.2 demonstrates One to Many LFSR.



(Figure 6.1)



(Figure 6.2)

VII. Contributions

Jason made the LFSR, One Pulse, and Debounce Module for the FPGA, while Tuguldur made the compare and top module. Jason is in charge of question 1 and 4. And Tuguldur was in charge of Question 2 and 3. Jason helped Tuudo debug for code 2 and 3. For the report Tuudo wrote report for FPGA, while Jason wrote report for 1,3,4 and Tudo wrote report 2.

VIII. Problems We Faced In This Lab

For Tuguldur, kept getting confused on the clock cycles and where the current state is at. Luckily, Jason was able to understand Tuguldur's code and help him debug it. Apart from the FPGA, Content-Addressable Memory and Scan Chain were the one the biggest challenge to us. For example, in Content-Addressable Memory, when ren == 1'b1, we did not have a clear idea of how to know if it matching or multi-matching or not matching at all. Just when Tuguldur Thought the Scan Chain was complete, the waveform was off by one clock cycle and it took a while to debug it.

Finally, for the FPGA, we did not have enough time to finish our 1a2b game because of other project deadlines and midterms.

IX. Things we learned from this Lab

In summary, most of the things we learned is about finite state machines, and memory, We learned how the scan chain works, and different ways to implement it. Thus, more knowledge about Content-Addressable Memory where why is it faster than a RAM and the difference between them. More hands-on practical knowledge and coding knowledge about finite machines such as moore and meely whereas in the Logic Design class, we were just familiar with the logic.