DIGITAL SYSTEM DESIGN APPLICATIONS

Experiment 6

FINITE STATE MACHINES

Berfin Duman
040190108

Contents

1 Introduction of Mealy and Moore Machine

Mealy and Moore machines are two different types of finite state machines (FSM) or finite state machines used in the field of numerical design. The main difference between these two types is how state transitions are made and when the logical units that produce output are updated.

1.0.1 Mealy Machine:

State Transitions: In Mealy machines, state transitions depend only on inputs. That is, the current state and inputs come together to determine the next state.
Outputs: Mealy machines produce instantaneous output based on inputs as well as status. This means that the outputs are updated instantly based on the current status and inputs.

1.0.2 Moore Machine:

State Transitions: In Moore machines, state transitions depend only on the current state. That is, the next state is determined as a function of the current state.
Outputs: Moore machines produce a fixed output that is associated with the state itself. So the outputs depend only on the state and the inputs do not affect the outputs.

2 FSM1 Project

2.1 State Encoding

In Finite State Machine (FSM) design, state encoding is a process of assigning numerical values used to represent the states of a machine.
In Finite State Machine (FSM) design, binary coding, a coding method used to represent states, is achieved by expressing states with binary numbers. This method means representing each state with a binary number and defining the states of the FSM using these numbers. In this method, each state is implemented by designing a log2(#state) flip flop.
By looking at Figure 1 given in the assignment, we can see that the Algorithm can be represented by 3-bit binary numbers from log2(6), considering that it contains 6 states (A, B, C, D, E, F). I will use the state encoding given in Fig3 for the given algorithm. All states of this encoding method are given in Fig4, and I will reduce these states in Q2, Q1, Q0 and z using the Karnaugh map method.

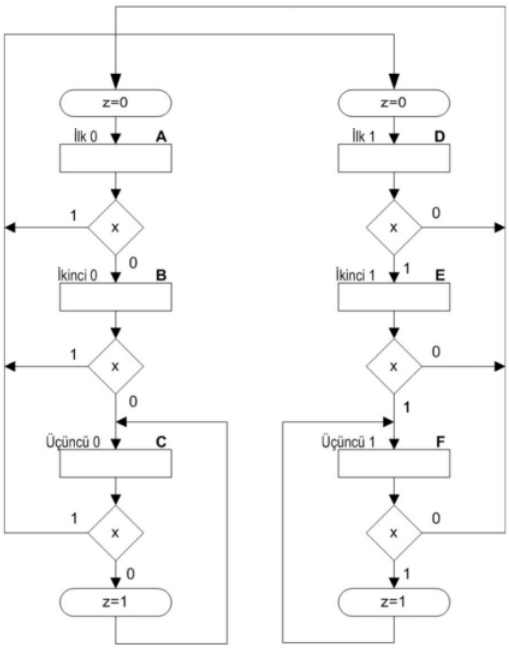


Figure 1: Algorithmic State Diagram

State	next state, output	
	x = 0	x = 1
A	B,0	D,0
B	C,0	D,0
C	C,1	D,0
D	A,0	E,0
E	A,0	F,0
F	A,0	F,1

Figure 2: State Table Without Any Reduction

state	binary code
A	000
B	001
C	010
D	011
E	100
F	101

Figure 3: State Encoding Example

x	q2	q1	q0	Q2	Q1	Q0	z
0	0	0	0	0	0	1	0
0	0	0	1	0	1	0	0
0	0	1	0	0	1	0	1
0	0	1	1	0	0	0	0
0	1	0	0	0	0	0	0
0	1	0	1	0	0	0	0
0	1	1	0	k	k	k	k
0	1	1	1	k	k	k	k
1	0	0	0	0	1	1	0
1	0	0	1	0	1	1	0
1	0	1	0	0	1	1	0
1	0	1	1	1	0	0	0
1	1	0	0	1	0	1	0
1	1	0	1	1	0	1	1
1	1	1	0	k	k	k	k
1	1	1	1	k	k	k	k

Figure 4: State Table after Encoding

2.2 Reduction of Combinatorial Parts

We define 6 states with 3 bits. We can represent 8 states in total with 3 bits. The 2 states we do not use are called arbitrary states, and we can include them if we need them while reducing, otherwise we can pretend they do not exist.
I minimized usin Karnaugh-Map method. After obtaining the minimized expressions, group terms that have common inputs. This grouping and minimizing helps to identify common subexpressions that can be implemented in a single LUT4 or decrease number of LUT4. Grouping is crucial for LUT optimization.
Let's look at the changes in lut4 before and after reduction:

$$Q1 = q1q0' + q2'q1'q0 + q2'q1x \text{This equation can realize using 7 LUT4.} \quad (1)$$

Q1= q1q0'+q2'q1'(q0+x) After reduction we realize only 5 LUT4 for Q1 (2)

$$Q2 = xq1' + q2'q1'q0 + q2'q0'x \text{This equation can realize using 7 LUT4.} \quad (3)$$

Q2= xq1'+q2'q1'(q1'+x) After reduction we realize only 5 LUT4 for Q2 (4)

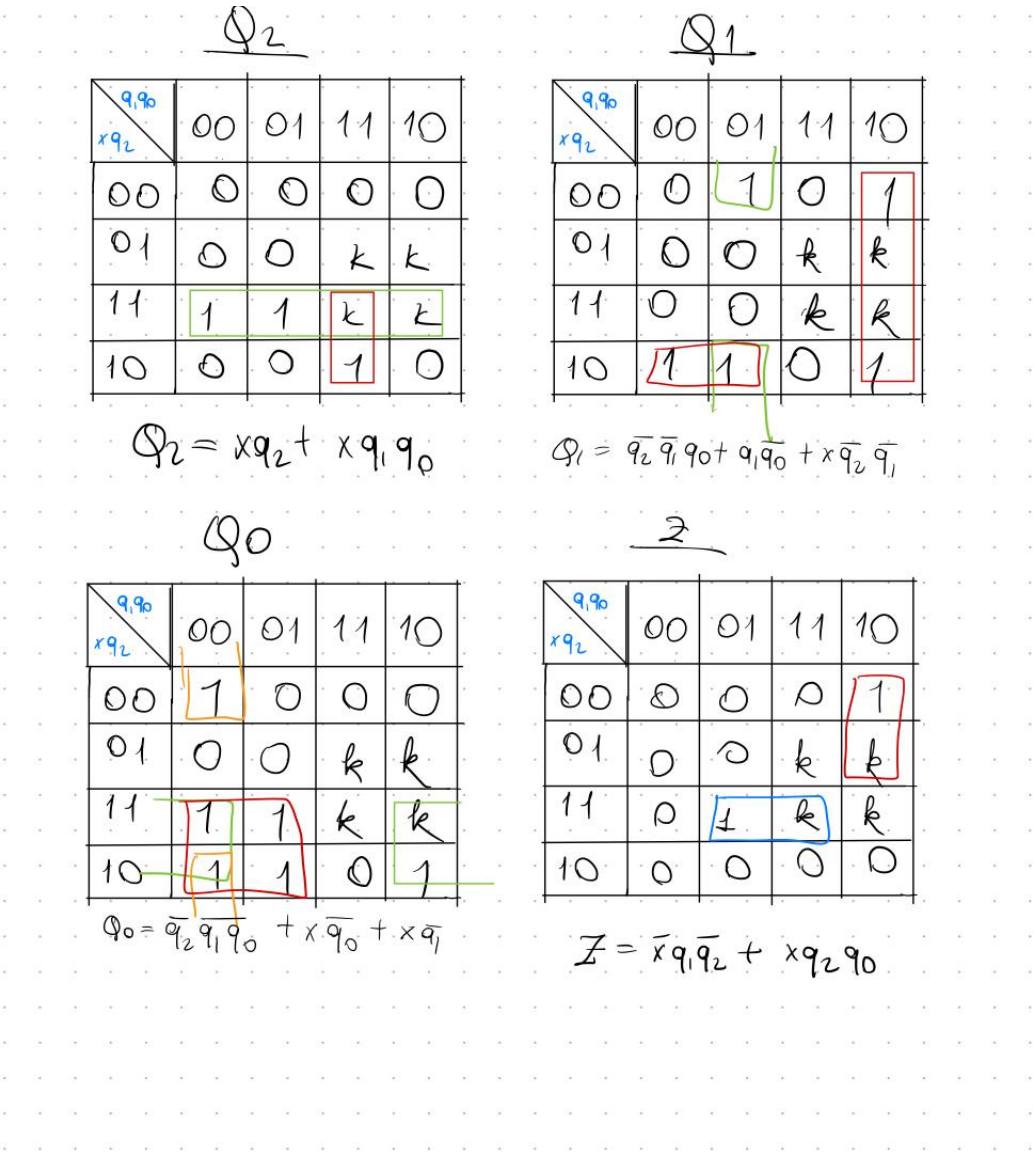


Figure 5: Reduction of flip-flops

2.3 Verilog Encoding

After collecting this preliminary information and learning the working principle of mealy and moore machines, I opened a new project in my exp6 folder to realize what I learned, created my FSM1.v file, and uploaded the constraint file of the FPGA (100T) we used throughout the semester. And I started creating the FSM1 module requested in the assignment.
Thanks to the comments I made on the FSM1 module, we can try this circuit with the machine we want by opening the relevant comment line and closing the other one. Likewise, I wrote a single testbench for both machines and the bonus part.
While assigning the combinatorial parts of the module code as requested, I wrote the flip flops in the always block, which is triggered by the positive part of the clock.
I implemented the testbench in the same way, as requested, by waiting 5 seconds after en-

tering the negative edge always block in order for it to be triggered at clock time.

Listing 1: FSM1 module

```
‘timescale 1ns / 1ps
module FSM1(
input x,clk,
//output z // could use for mealy machine
output reg z // could use for moore machine
    );
reg q2,q1,q0;
wire Q2,Q1,Q0;
wire temp_z; // could use for moore machine
initial
begin
q2=0;
q1=0;
q0=0;
end
assign Q0=(~q2&~q1&~q0)|(x& ~q0)|(x& ~q1);
assign Q1=(~q2&~q1&q0) |(q1&~q0)| (x&~q2&~q1);
assign Q2=(x&q2) | (x&q1&q0);
assign temp_z=(~x&q1&~q0)|(x&q2&q0);  // could use for moore
   machine
//assign z=(~x&q1&~q0)|(x&q2&q0); // could use for mealy
   machine
always @(posedge clk)
begin
q2<=Q2;
q1<=Q1;
q0<=Q0;
z<=temp_z; // could use for moore machine
end
endmodule
```

Listing 2: FSM_tb module testbench

```
‘timescale 1ns / 1ps
module FSM_tb;
    reg x   ;
    reg clk ;
    wire z  ;
    reg [41:0] test_signal =
       42'b010011000111000011110000011111000000111111;
    FSM1 uut(.x(x), .clk(clk), .z(z));
    //FSM1 uut(.x(x), .CLK(CLK), .z(z));
```

```
    initial begin
            clk = 0            ;
            x  = test_signal[41] ;
            #20;
    end
    always #10 clk = !clk ;
    always @(negedge clk)
    begin
    #5
      test_signal = test_signal << 1;
      x <= test_signal[41] ;

    if (test_signal == 42'b0) begin
        $finish;
    end
    end
endmodule
```

2.4 Analysis of Modules

I implemented the circuit with Mealy and Moore and got the desired schema and layout representations at each stage. Below are the RTL schematics, technology schematics and device layouts. Before this analysis i will compare behavioral and post-implementation functional simulations for Mealy and Moore, respectively.

```
## This file is a general .xdc for the Nexys A7-100T
set_property CLOCK_DEDICATED_ROUTE FALSE [get_nets clk_IBUF];
##Switches
set_property -dict { PACKAGE_PIN J15   IOSTANDARD LVCMOS33 } [get_ports { x}];
## LEDs
set_property -dict { PACKAGE_PIN H17   IOSTANDARD LVCMOS33 } [get_ports { z }];
##Buttons
set_property -dict { PACKAGE_PIN N17   IOSTANDARD LVCMOS33 } [get_ports { clk}];
```

I solved the problem by connecting the input and output variables with a constraint file and adding the last line because it caused problems in the implementation due to the clock condition.I used these constraint file for these project.

2.4.1 Compare of Simulations

Mealy Machines

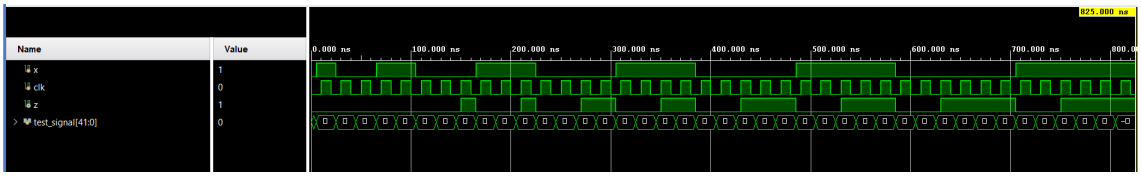


Figure 6: Behavioral Simulation of Mealy Machine of FSM1

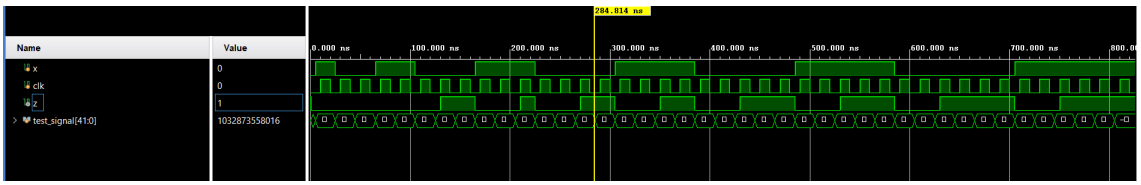


Figure 7: Post- Implementation on Function Simulation of Mealy Machine of FSM1

The results obtained from simulating post-implantation function show inaccuracies. The characteristics of the faulties outputs remain consistent. For every set of 3 identical inputs, the circuit produces a high output instead of the expected output for 4 identical inputs. The root cause of these inaccuracies lies in the Mealy machine, where both inputs and states influence the output. Consequently, the output does not wait for the next clock signal. To prevent such issues, a Moore machine can be employed.
To avert malfunctions, we can incorporate a D flip-flop into the circuit that was designed earlier. This modification transforms the circuit from a Mealy machine to a Moore machine. As can be seen, the Moore machine simulation results were as we expected.

Moore Machines

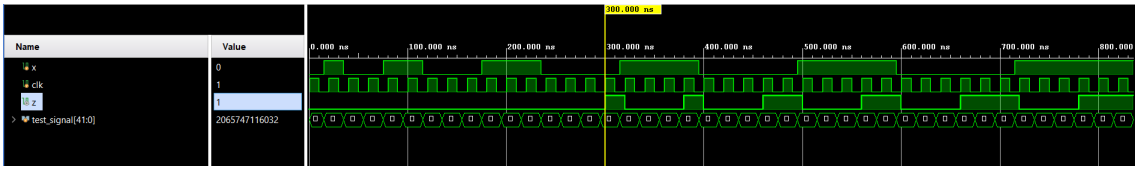


Figure 8: Behavioral Simulation of Moore Machine of FSM1

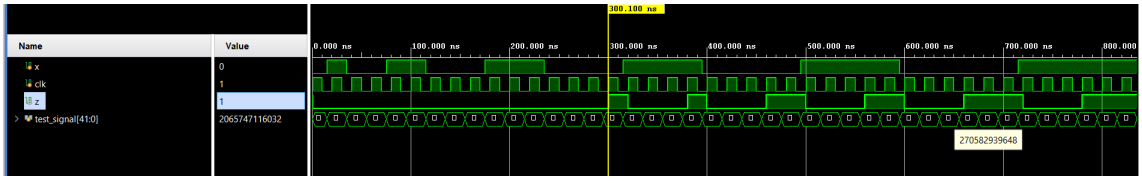


Figure 9: Post- Implementation on Function Simulation of Moore Machine of FSM1

2.4.2 Analysis Mealy Part

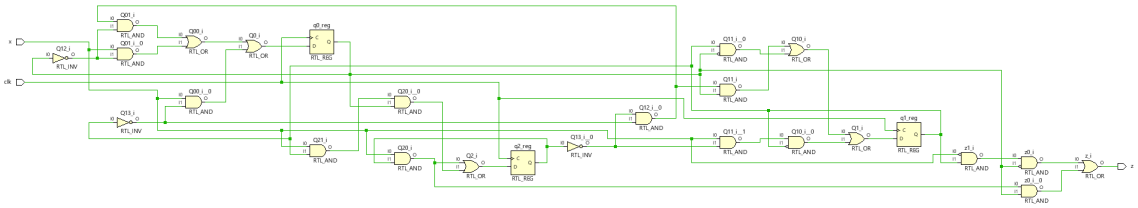


Figure 10: RTL Schematic of Moore Machine of FSM1

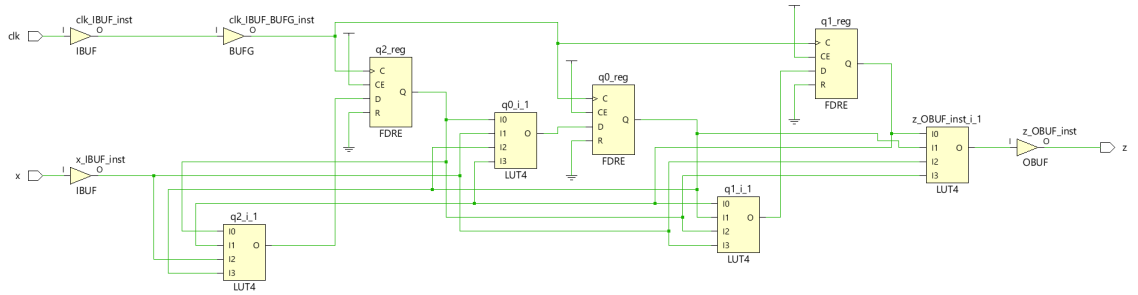


Figure 11: Technology Schematic of Moore Machine of FSM1

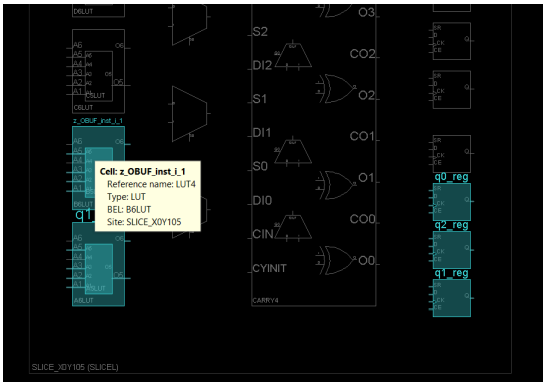


Figure 12: Layout of Mealy Machine of FSM1

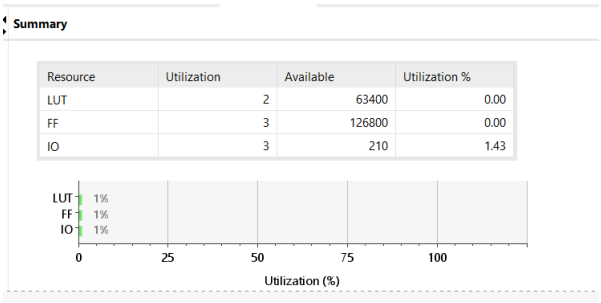


Figure 13: Utilization Summary of Moore Machine of FSM1

From Port	To Port	Max Delay	Max Process Corner	Min Delay	Min Process Corner
x	z	8.617	SLOW	2.551	FAST

Figure 14: Timing Summary of Moore Machine of FSM1

2.4.3 Analysis Moore Part

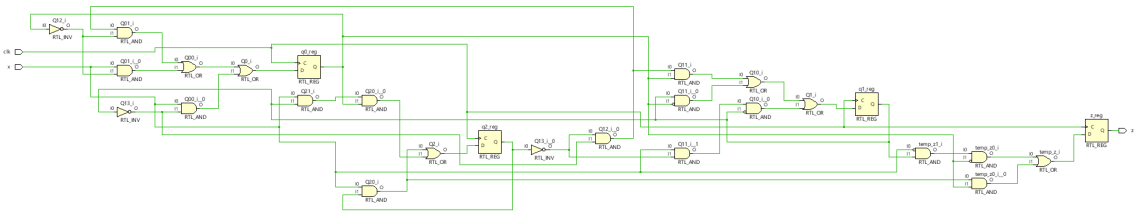


Figure 15: RTL Schematic of Moore Machine of FSM1

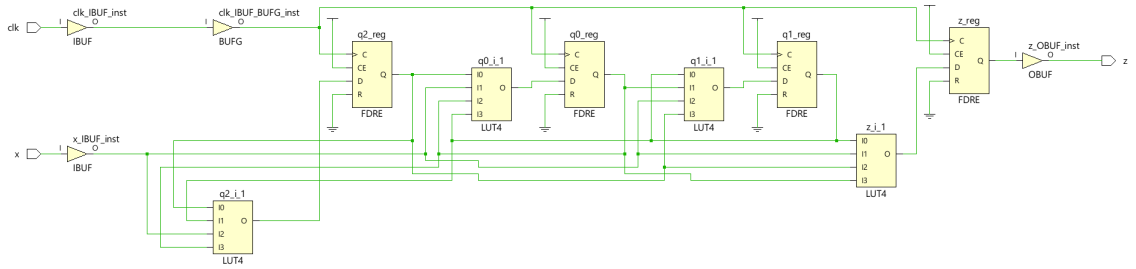


Figure 16: Technology Schematic of Moore Machine of FSM1

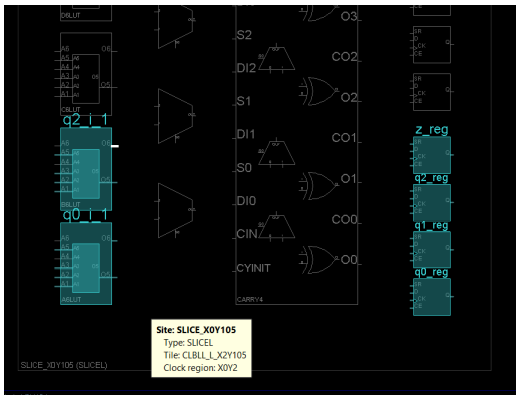


Figure 17: Layout of Moore Machine of FSM1

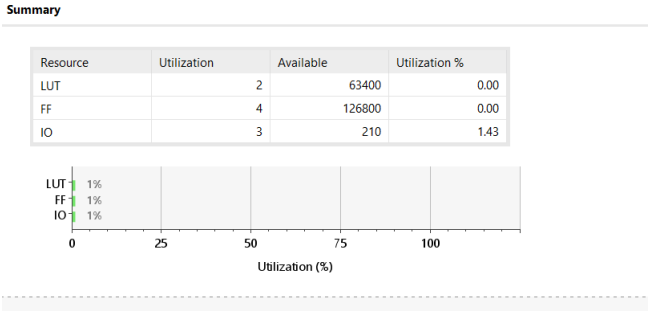


Figure 18: Utilization Summary of Moore Machine of FSM1

2.5 Stucking in Undesirable States

Now let's go back to our mealy code and see the results by replacing our initial values with arbitrary states for which we have not specified a state.
Arbitrary States= 110, 111

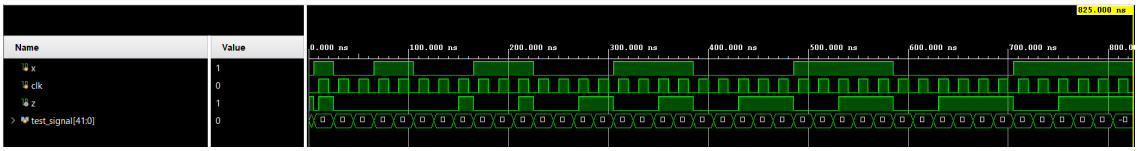


Figure 19: Circuit Simulation Initialize with Arbitrary States 110 of FSM2

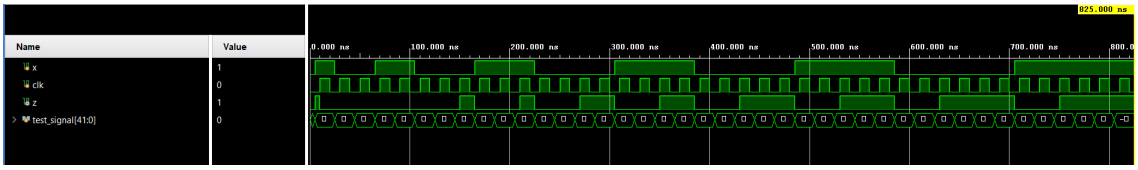


Figure 20: Circuit Simulation Initialize with Arbitrary States 111 of FSM2

According to the behavioral simulation of the circuit, the initial state is arbitrary, and the circuit can transition from arbitrary states to states encoded in the circuit. Please note that at the beginning of the simulation, it may produce incorrect results.

3 FSM2 Project

For the FSM2 project, I applied the steps in FSM1 one by one. I wrote the code for the module and used the same testbench I used in FSM1 by directly calling the FSM2 module. I created the diagrams of the module as follows:

3.1 Verilog Code

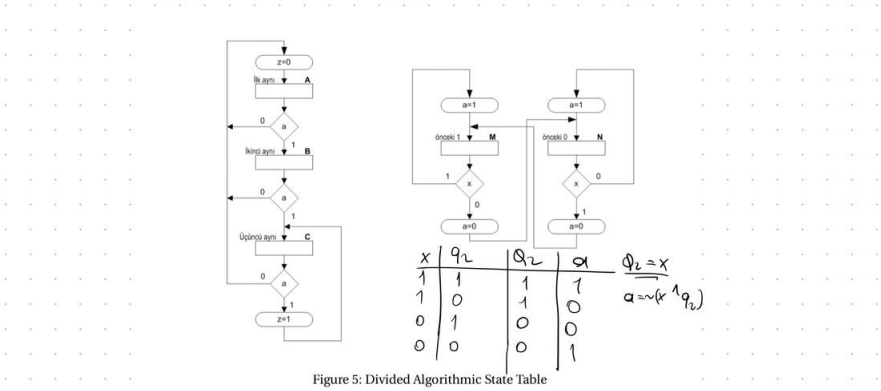


Figure 5: Divided Algorithmic State Table

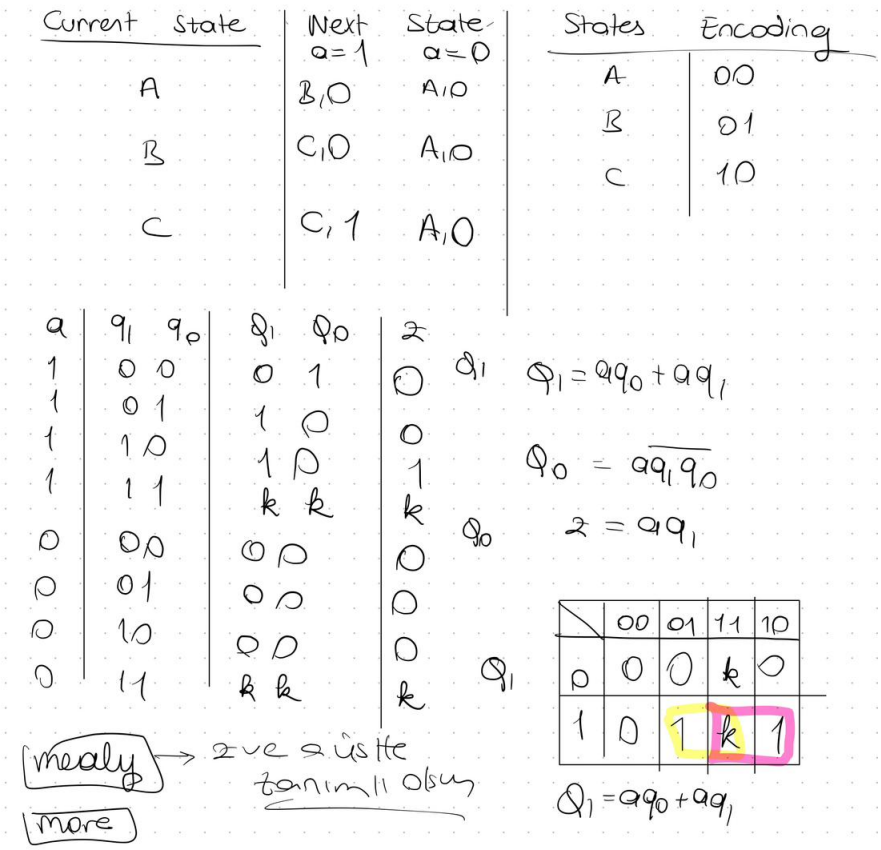


Figure 21: Hand Written Circuit of FSM2

Listing 3: FSM2 module

```
1 `timescale 1ns / 1ps
2 `timescale 1ns / 1ps
3 module FSM2(
4 input x,clk,
5 output z // could use for mealy machine
```

```
//output reg z // could use for moore machine
    );
reg q2,q1,q0;
//reg a; // could use for moore machine
wire Q2,Q1,Q0;
wire temp_a;
initial
begin
q2=0;
q1=0;
q0=0;
//a=0; // could use for moore machine
end
assign Q2=x;
assign Q0=temp_a& ~q1 & ~q0;
assign Q1=temp_a & q0 | temp_a & q1;
assign temp_a= ~(x ^ q2);
//assign temp_z=temp_a&q1;  // could use for moore machine
assign z=temp_a&q1; // could use for mealy machine
always @(posedge clk)
begin
q2<=Q2;
q1<=Q1;
q0<=Q0;
//z<=temp_z; // could use for moore machine
//a<=temp_a; // could use for moore machine
end
endmodule
```

3.2 Analysis of Modules

3.2.1 Compare of Simulations

Mealy Machines

Figure 22: Behavioral Simulation of Mealy Machine of FSM2

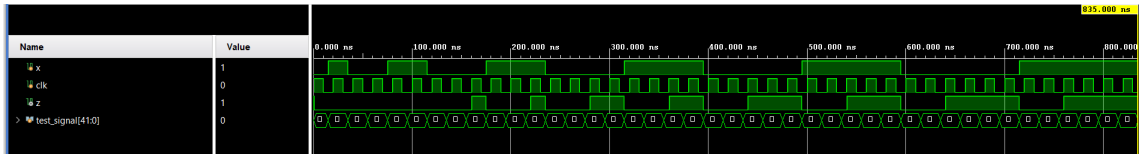


Figure 23: Post- Implementation on Function Simulation of Mealy Machine of FSM2

Moore Machines

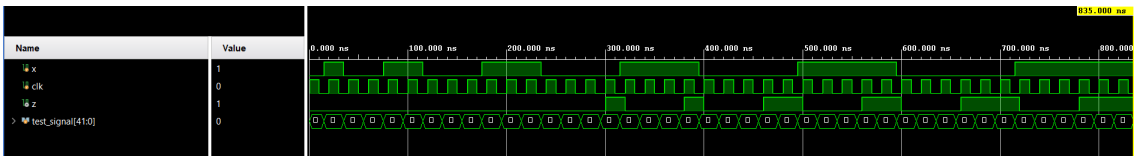


Figure 24: Behavioral Simulation of Moore Machine of FSM2

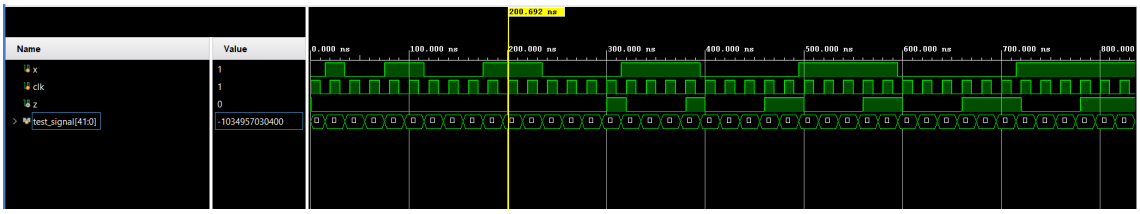


Figure 25: Post- Implementation on Function Simulation of Moore Machine of FSM2

3.2.2 Analysis Mealy Part

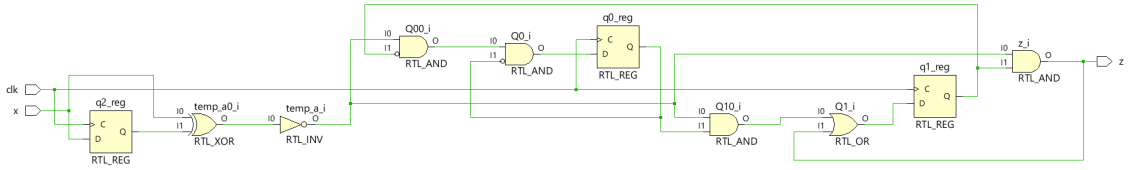


Figure 26: RTL Schematic of Mealy Machine of FSM2

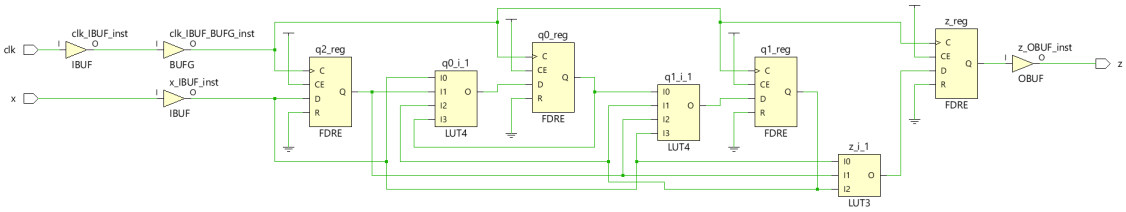


Figure 27: Technology Schematic of Mealy Machine of FSM2

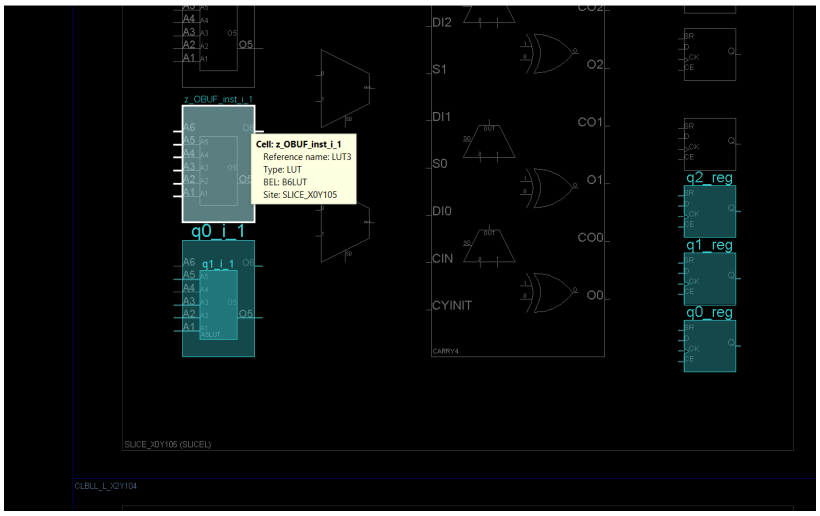


Figure 28: Layout of Mealy Machine of FSM2

Summary

Resource	Utilization	Available	Utilization %
LUT	2	63400	0.00
FF	3	126800	0.00
IO	3	210	1.43

Figure 29: Utilization Summary of Mealy Machine of FSM2

3.2.3 Analysis Moore Part

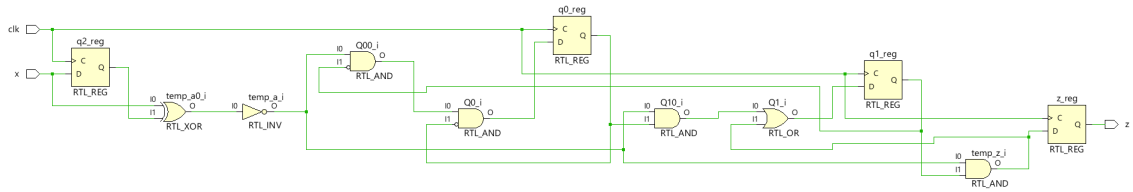


Figure 30: RTL Schematic of Moore Machine of FSM2

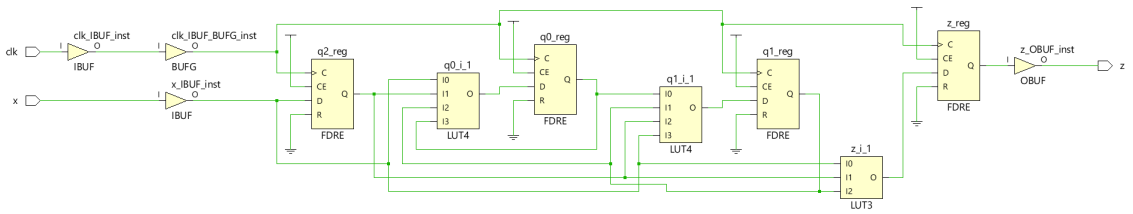


Figure 31: Technology Schematic of Moore Machine of FSM2

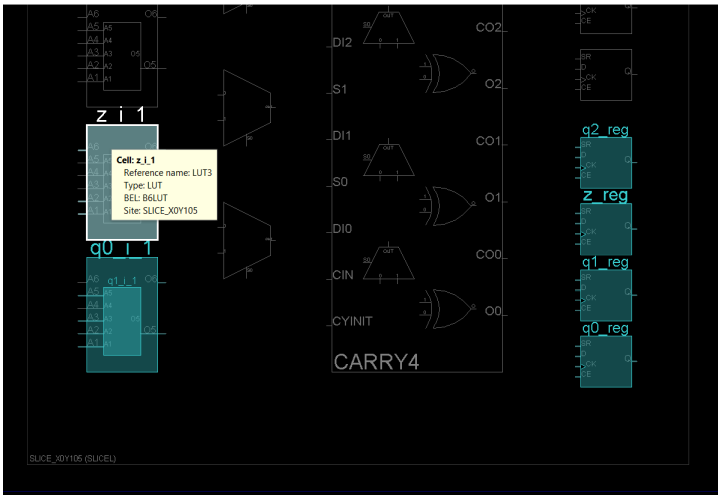


Figure 32: Layout of Moore Machine of FSM2

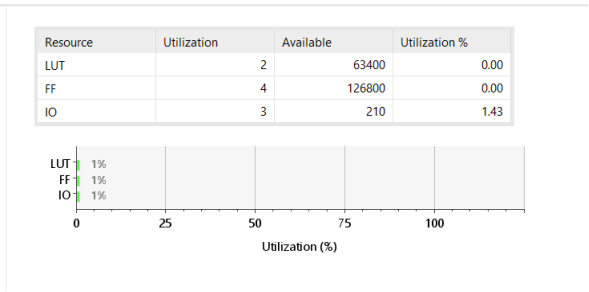


Figure 33: Utilization Summary of Moore Machine of FSM2

3.3 Stucking Undesirable States

Now let's go back to our mealy code and see the results by replacing our initial values with arbitrary states for which we have not specified a state.
Arbitrary States= 110, 111

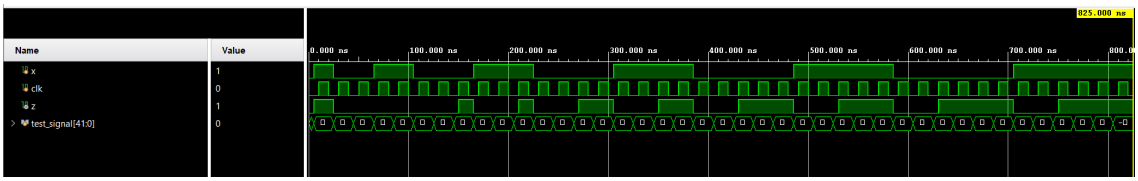


Figure 34: Circuit Simulation Initialize with Arbitrary States 110 of FSM2

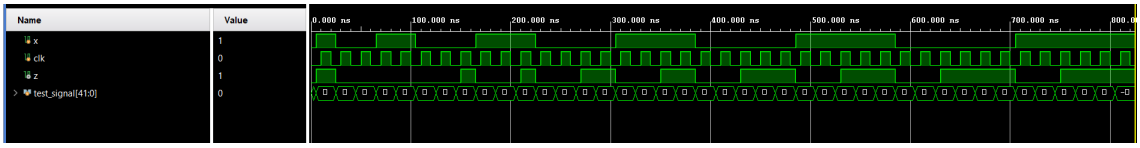
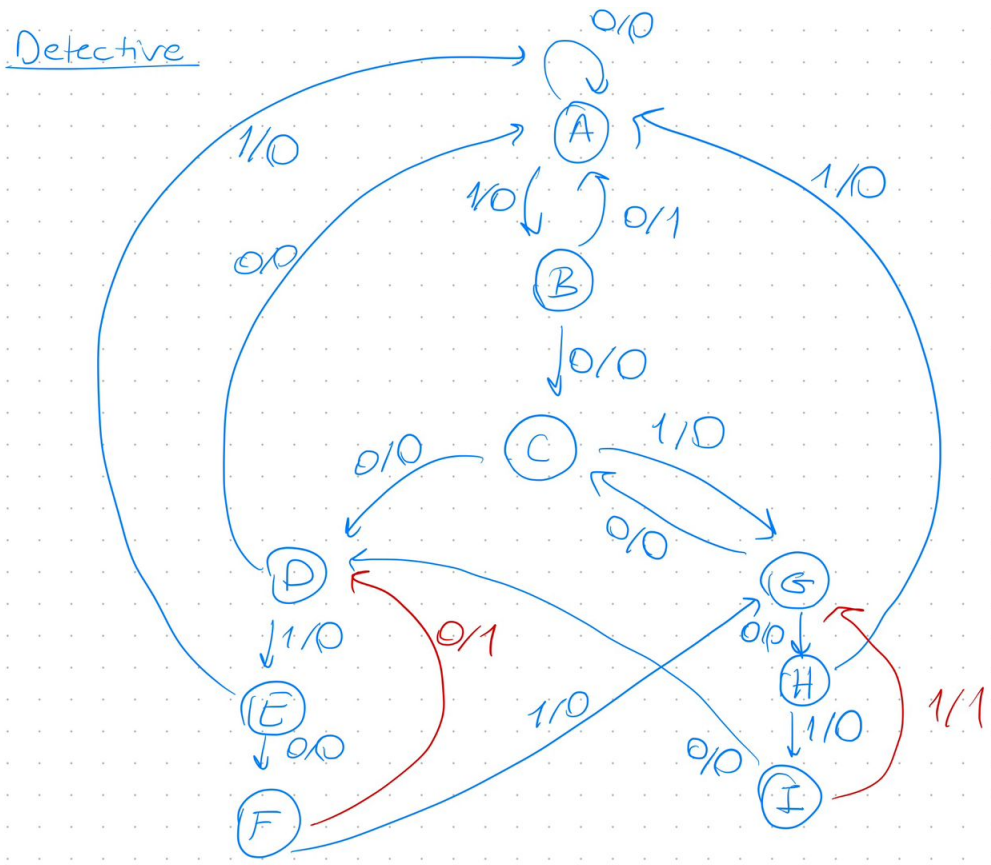


Figure 35: Circuit Simulation Initialize with Arbitrary States 111 of FSM2

According to the behavioral simulation of the circuit, the initial state is arbitrary, and the circuit can transition from arbitrary states to states encoded in the circuit. Please note that at the beginning of the simulation, it may produce incorrect results.

4 Detective Project

Similarly, I wrote the verilog code to implement the given detective algorithm, added the test code to test it, and thought that I did not see any problems with the test and that I had written the algorithm correctly, I carried out the synthesis and implementation without any problems. I have expressed 9 state log2(9) in 4 bits in binary code. I drew a situation diagram and state reduction like this:



State	binary code
A	0000
B	0001
C	0010
D	0011
E	0100
F	0101
G	0110
H	0111
I	1000

Figure 36: State Diagram and S. Table of Detective Module

Listing 4: detective module

```
module detect(
     input x, clk,
     output reg z
     );
     reg flag;
     reg [3:0] state ;
     reg [3:0] next_state ;

     initial begin
     state = 4'b0000;
     next_state = 4'b0000;
     flag = 1'b0;
     end

     always @(posedge clk) begin
     state <= next_state;
     end
         always @(state, x) begin

         case(state)
         4'b0000 : begin
         if (x == 1)begin
             z <= 1'b0;
             next_state <= 4'b0001;
         end
        else begin
            z <= 1'b0;
             next_state <= 4'b0000;
          end
          end
          4'b0001 : begin
          if (x == 1) begin
              z <= 1'b0;
              next_state <= 4'b0000;
          end
          else begin
              z <= 1'b0;
              next_state <= 4'b0010;
          end
          end
          4'b0010 : begin
          if (x == 1) begin
              z <= 1'b0;
              next_state <= 4'b0111;
```

```
45          end
46         else begin
47              z <= 1'b0;
48              next_state <= 4'b0011;
49          end
50          end
51          4'b0011 : begin
52          if (x == 1) begin
53              z <= 1'b0;
54              next_state <= 4'b0100;
55          end
56         else begin
57              z <= 1'b0;
58              next_state <= 4'b0000;
59          end
60          end
61          4'b0100 : begin
62          if (x == 1) begin
63              z <= 1'b0;
64              next_state <= 4'b0000;
65          end
66         else begin
67              z <= 1'b0;
68              next_state <= 4'b0101;
69          end
70          end
71          4'b0101 : begin
72          if (x == 1) begin
73              z <= 1'b0;
74              next_state <= 4'b0111;
75          end
76         else begin
77              z <= 1'b1;
78              next_state <= state;
79              flag <=1;
80          end
81          end
82          4'b0111 : begin
83          if (x == 1) begin
84             z <= 1'b0;
85              next_state <= 4'b1000;
86          end
87         else begin
88              z <= 1'b0;
89              next_state <= 4'b0010;
```

```
90          end
91          end
92          4'b1000 : begin
93          if (x == 1) begin
94              z <= 1'b0;
95              next_state <= 4'b0000;
96          end
97         else begin
98              z <= 1'b0;
99              next_state <= 4'b1001;
100         end
101         end
102         4'b1001 : begin
103         if (x == 1)begin
104             z <= 1'b1;
105             next_state <= state;
106             flag <=1;
107         end
108         else begin
109             z <= 1'b0;
110             next_state <= 4'b0011;
111         end
112         end
113         endcase
114         end
115
116 endmodule
```

Listing 5: detective test bench module

```
1  module FSM1_tb();
2      wire z;
3      reg x; reg clk;
4
5      reg [41:0] test =
          42'b010010000111011000000100011101101100000111;
6      detect uut(x, clk, z);
7      always #5 clk = !clk;
8      initial begin
9      x = 0; clk = 0;
10     #10;
11     end
12
13     always @(posedge clk)
14     begin
15         test = test << 1;
```

```
16          x = test[41];
17      end
18  endmodule
```

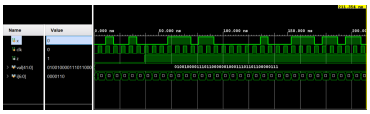


Figure 37: Behavioral Simulation of Detective

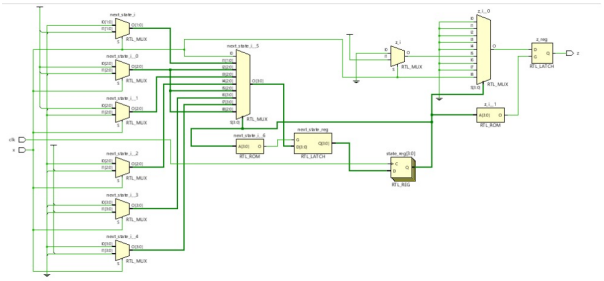


Figure 38: Rtl Schematic of Detective

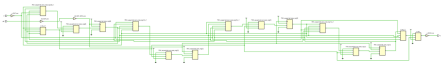


Figure 39: Tech Schematic of Detective

Resource	Utilization	Available	Utilization %
LUT	3	63400	0.00
FF	9	126800	0.01
IO	3	210	1.43

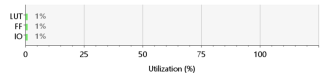


Figure 40: Implementation summary of Detective

Name	Slack	Levels	Routes	High Fanout	From	To	Total Delay	Logic Delay	Net Delay
Path 1	∞	2	1	1	z_reg/G	z	6.149	4.079	2.069
Path 2	∞	2	2	5	x	z_reg/D	3.828	1.602	2.227
Path 3	∞	2	2	5	x	FSM_seque..._reg[1]/D	3.728	1.602	2.126
Path 4	∞	2	2	5	x	FSM_seque..._reg[0]/D	3.435	1.630	1.805
Path 5	∞	2	2	5	x	FSM_seque..._reg[3]/D	3.434	1.628	1.806
Path 6	∞	2	1	5	x	FSM_seque..._reg[2]/D	3.055	1.632	1.423
Path 7	∞	1	1	1	FSM_seque..._reg[0]/G	FSM_seque..reg[0]/D	1.330	0.559	0.771
Path 8	∞	1	1	1	FSM_seque..._reg[2]/G	FSM_seque..reg[2]/D	1.325	0.559	0.766
Path 9	∞	1	1	1	FSM_seque..._reg[1]/G	FSM_seque..reg[1]/D	1.179	0.559	0.620
Path 10	∞	1	1	1	FSM_seque..._reg[3]/G	FSM_seque..reg[3]/D	0.942	0.559	0.383

Name	Slack	Levels	Routes	High Fanout	From	To	Total Delay	Logic Delay	Net Delay
Path 11	∞	1	1	1	FSM_seque..._reg[3]/G	FSM_seque..reg[3]/D	0.278	0.158	0.120
Path 12	∞	2	1	6	FSM_seque..reg[3]/C	FSM_seque..._reg[2]/D	0.313	0.192	0.121
Path 13	∞	1	1	1	FSM_seque..._reg[1]/G	FSM_seque..reg[1]/D	0.388	0.158	0.230
Path 14	∞	2	2	6	FSM_seque..reg[0]/C	FSM_seque..._reg[3]/D	0.428	0.189	0.239
Path 15	∞	2	2	6	FSM_seque..reg[0]/C	FSM_seque..._reg[0]/D	0.436	0.190	0.246
Path 16	∞	1	1	1	FSM_seque..._reg[2]/G	FSM_seque..reg[2]/D	0.526	0.158	0.368
Path 17	∞	1	1	1	FSM_seque..._reg[0]/G	FSM_seque..reg[0]/D	0.527	0.158	0.369
Path 18	∞	2	2	6	FSM_seque..reg[0]/C	FSM_seque..._reg[1]/D	0.633	0.186	0.447
Path 19	∞	2	2	6	FSM_seque..reg[0]/C	z_reg/D	0.677	0.186	0.491
Path 20	∞	2	1	1	z_reg/G	z	1.884	1.379	0.504

Figure 41: Timing report of Detective