Assignment-2

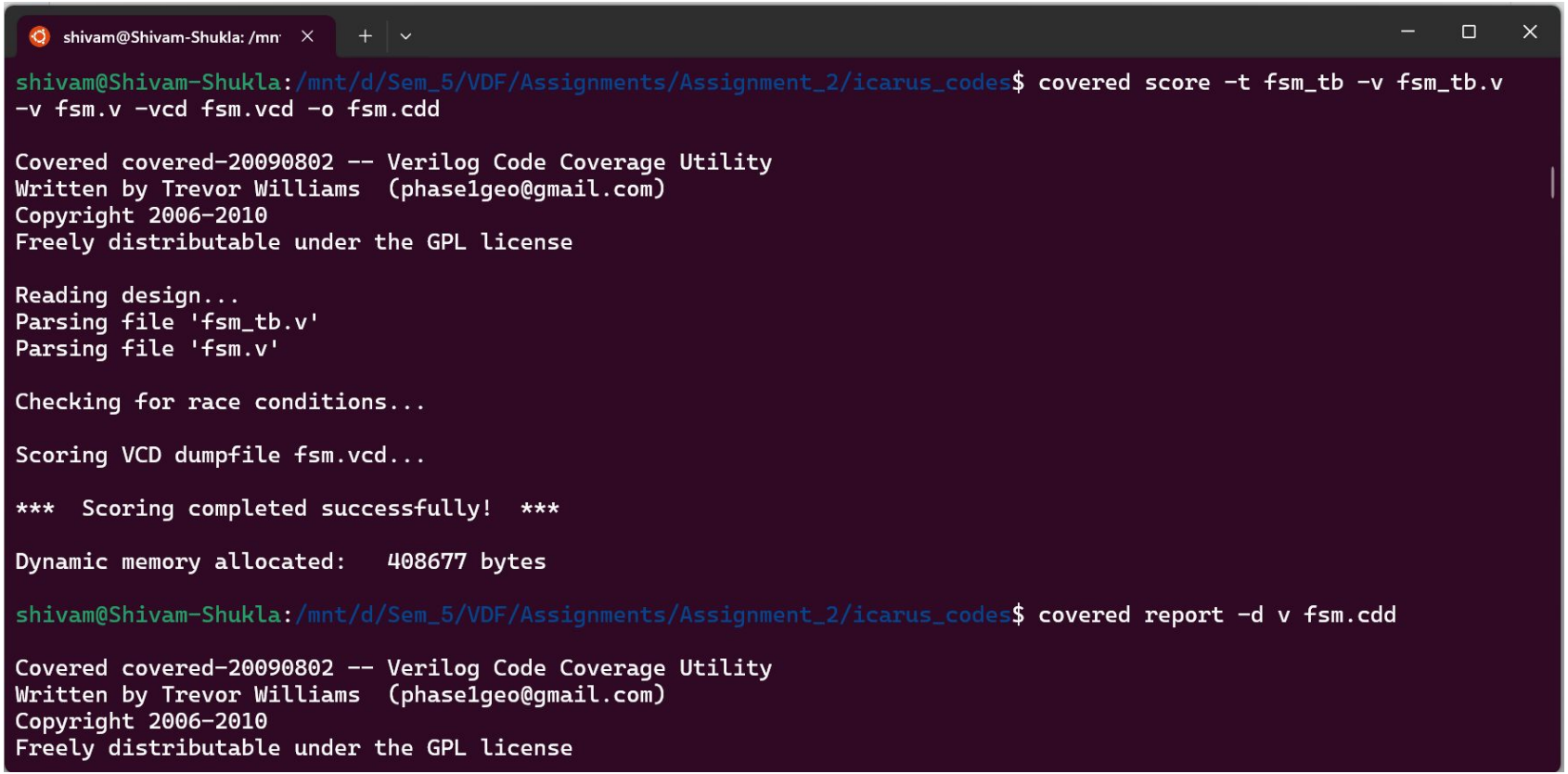
Name: Shivam Shukla
Roll No.: 2022478
Advisor: Sneh Saurabh
Date: 24/09/2024
Course: VDF

INDRAPRASTHA INSTITUTE *of*
INFORMATION TECHNOLOGY
DELHI

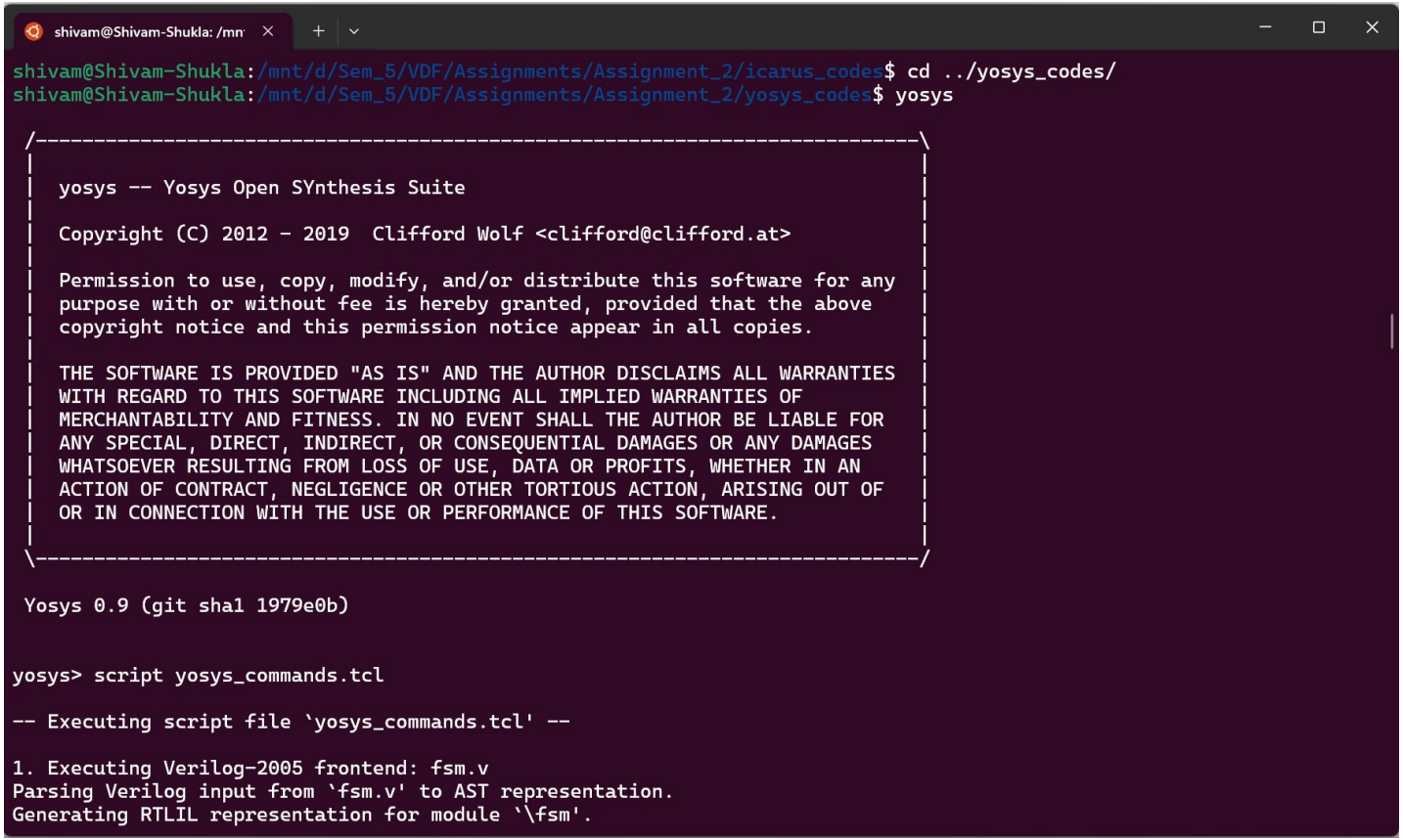
Q1) Snippet of the log files to prove that I have indeed run the tools

```
shivam@Shivam-Shukla: /mn
Welcome to Ubuntu 22.04.4 LTS (GNU/Linux 5.15.153.1-microsoft-standard-WSL2 x86_64)

 * Documentation:  https://help.ubuntu.com
 * Management:     https://landscape.canonical.com
 * Support:        https://ubuntu.com/pro


This message is shown once a day. To disable it please create the
/home/shivam/.hushlogin file.
shivam@Shivam-Shukla:~$ cd /mnt/d/Sem_5/VDF/Assignments/Assignment_2/icarus_codes
shivam@Shivam-Shukla:/mnt/d/Sem_5/VDF/Assignments/Assignment_2/icarus_codes$ iverilog -Wall -o test fsm.v fsm_tb.v
shivam@Shivam-Shukla:/mnt/d/Sem_5/VDF/Assignments/Assignment_2/icarus_codes$ vvp test
VCD info: dumpfile fsm.vcd opened for output.
shivam@Shivam-Shukla:/mnt/d/Sem_5/VDF/Assignments/Assignment_2/icarus_codes$ gtkwave fsm.vcd

GTKWave Analyzer v3.3.104 (w)1999-2020 BSI


(gtkwave:205294): dconf-WARNING **: 15:25:50.655: failed to commit changes to dconf: Could not connect: No such fi
le or directory
[0] start time.
[834] end time.
```

Q1) Snippet of the log files to prove that I have indeed run the tools



Q1) Snippet of the log files to prove that I have indeed run the tools



Q2 (a) Verilog model using binary encoding

```
module fsm (
 input clk,            //Clock signal
 input reset,          //Reset signal
 input [1:0] in,       //2-bit input signal
 output reg [1:0] out   //2-bit output signal
);

//States
parameter S1 = 3'b000;
parameter S2 = 3'b001;
parameter S3 = 3'b010;
parameter S4 = 3'b011;
parameter S5 = 3'b100;

//State reg and next state reg
reg [2:0] state_reg, next_reg;
```

```
//State transition
always @(posedge clk or posedge reset)
begin
 if (reset) begin
 state_reg <= S1;  //Reset to initial state
S1
 end else begin
 state_reg <= next_reg;  //Transition to
next state
 end
 end

always @(*) begin
 //Default values
 next_reg=state_reg;
 out=2'b00;
```

```
//Switch cases for states and outputs
case (state_reg)
S1: begin
case (in)
2'b00: begin
out=2'b01;
next_reg=S1;
end
2'b01: begin
out=2'b00;
next_reg=S1;
end
2'b10: begin
out=2'b01;
next_reg=S2;
end
```

Q2 (a) Verilog model using binary encoding

```
2'b11: begin
out=2'b00;
next_reg=S3;
end
endcase
end
S2: begin
case (in)
2'b00: begin
out=2'b11;
next_reg=S1;
end
2'b01: begin
out=2'b11;
next_reg=S3;
end
2'b10: begin
out=2'b10;
next_reg=S2;
end
```

```
2'b11: begin
out=2'b10;
next_reg=S4;
end
endcase
end
S3: begin
case (in)
2'b00: begin
out=2'b00;
next_reg=S1;
end
2'b01: begin
out=2'b01;
next_reg=S5;
end
2'b10: begin
out=2'b00;
next_reg=S3;
end
```

```
2'b11: begin
out=2'b00;
next_reg=S4;
end
endcase
end
S4: begin
case (in)
2'b00: begin
out=2'b11;
next_reg=S3;
end
2'b01: begin
out=2'b01;
next_reg=S5;
end
2'b10: begin
out=2'b01;
next_reg=S5;
end
```

Q2 (a) Verilog model using binary encoding

```
2'b11: begin
out=2'b11;
next_reg=S4;
end
endcase
end
S5: begin
case (in)
2'b00: begin
out=2'b00;
next_reg=S5;
end
2'b01: begin
out=2'b00;
next_reg=S1;
end
```

```
2'b10: begin
out=2'b10;
next_reg=S1;
end
2'b11: begin
out=2'b11;
next_reg=S5;
end
endcase
end
endcase
end
endmodule
```

PS: I've combined the code snippets in sequence, with each section following after the white rectangles. Can also find the code here

Q2 (b) The testbench to test the Verilog model.

```
module fsm_tb;
 reg clk;
 reg reset;
 reg [1:0] in;
 wire [1:0] out;

 //Instantiate our verilog top level fsm
 fsm dut(
 .clk(clk),
 .reset(reset),
 .in(in),
 .out(out)
 );

//Generating clk
 initial
 begin
 clk=0;
 forever #10 clk = ~clk;    //To toggle clk
 end

 //Tests
 initial
 begin

 $dumpfile("fsm.vcd");      //Dumpfile
 $dumpvars(0,fsm_tb);

 //Defining reset and in
 reset = 1; in = 2'b00;
 #20 reset = 0;

 //All possible paths are tested by input
combinations
 #20 in=2'b00; //S1
 #20 in=2'b01; //S1
 #20 in=2'b10; //S2
 #20 in=2'b10; //S2
 #20 in=2'b01; //S3
 #20 in=2'b10; //S3
 #20 in=2'b11; //S4
 #20 in=2'b11; //S4
 #20 in=2'b10; //S5
 #20 in=2'b00; //S5
 #20 in=2'b11; //S5
 #20 in=2'b01; //S1

 reset=1;
 #0.5 reset=0;

 #20 in=2'b11; //S3
 #20 in=2'b00; //S1
 #20 in=2'b11; //S3
 #20 in=2'b01; //S5
 #20 in=2'b10; //S1

 reset=1;
 #0.5 reset=0;
```

Q2 (b) The testbench to test the Verilog model.

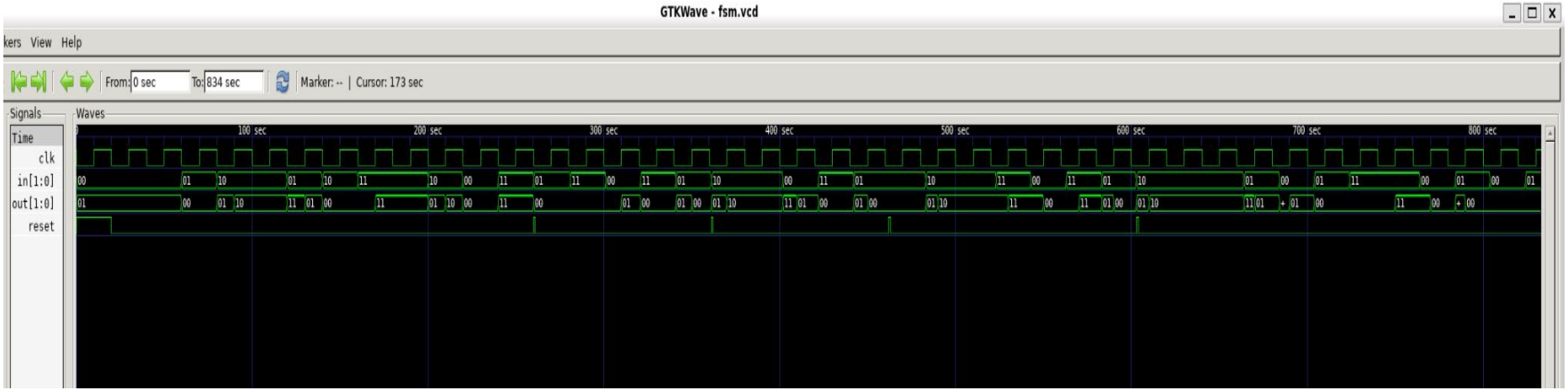
```
#20 in=2'b10; //S2
#20 in=2'b00; //S1
#20 in=2'b11; //S3
#20 in=2'b01; //S5
#20 in=2'b01; //S1

reset=1;
#0.5 reset=0;

#20 in=2'b10; //S2
#20 in=2'b10; //S2
#20 in=2'b11; //S4
#20 in=2'b00; //S3
#20 in=2'b11; //S4
#20 in=2'b01; //S5
#20 in=2'b10; //S1
```

```
reset=1;
#0.5 reset=0;

#20 in=2'b10;
#20 in=2'b10;
#20 in=2'b01;
#20 in=2'b00;
#20 in=2'b01;
#20 in=2'b11;
#20 in=2'b11;
#20 in=2'b00;
#20 in=2'b01;
#20 in=2'b00;
#20 in=2'b01;

//End simulation
#10 $finish;
end
```

```
 //signal verification
 always @(posedge clk or posedge reset)
begin
 if (reset) begin
 end else begin
 end
 end
endmodule
```

PS: I've combined the code snippets in sequence, with each section following after the white rectangles. Can also find the code here

Q2 (c) The simulation output waveform



The waveform shows the FSM simulation with clock (clk), reset (reset), input (in), and output (out).

- Clock (clk): Toggles regularly, driving state transitions.
- Reset (reset): When high, the FSM resets to the initial state (S1), and the output goes to its default.
- Input (in): The 2-bit input (00, 01, 10, 11) changes, causing the FSM to move between states.
- Output (out): Changes as per FSM logic, producing the correct output for each state and input.

The simulation output waveform confirms correct transitions and outputs, matching the Verilog FSM design.

Q2 (d) Line Coverage report for the above testbench.

The testbench and FSM design achieved 100% line coverage, meaning every line of code was executed in the simulation. This confirms that the testbench fully tests all parts of the FSM design.

```
shivam@Shivam-Shukla: /mn

shivam@Shivam-Shukla:/mnt/d/Sem_5/VDF/Assignments/Assignment_2/icarus_codes$ covered report -d v fsm.cdd

Covered covered-20090802 -- Verilog Code Coverage Utility
Written by Trevor Williams  (phase1geo@gmail.com)
Copyright 2006-2010
Freely distributable under the GPL license

                              :::::::::::::::::::::::::::::::::::::::::::::::::
                              ::                                             ::
                              ::   Covered -- Verilog Coverage Verbose Report ::
                              ::                                             ::
                              :::::::::::::::::::::::::::::::::::::::::::::::::


~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~
~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~   GENERAL INFORMATION   ~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~
~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~
* Report generated from CDD file : fsm.cdd

* Reported by                    : Module

~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~
~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~   LINE COVERAGE RESULTS   ~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~
~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~
Module/Task/Function      Filename                Hit/ Miss/Total    Percent hit
---------------------------------------------------------------------------------------------------------------------
  $root                   NA                        0/    0/    0      100%
  fsm_tb                  fsm_tb.v                 55/    0/   55      100%
  fsm                     fsm.v                    47/    0/   47      100%
---------------------------------------------------------------------------------------------------------------------
  Accumulated                                     102/    0/  102      100%
```

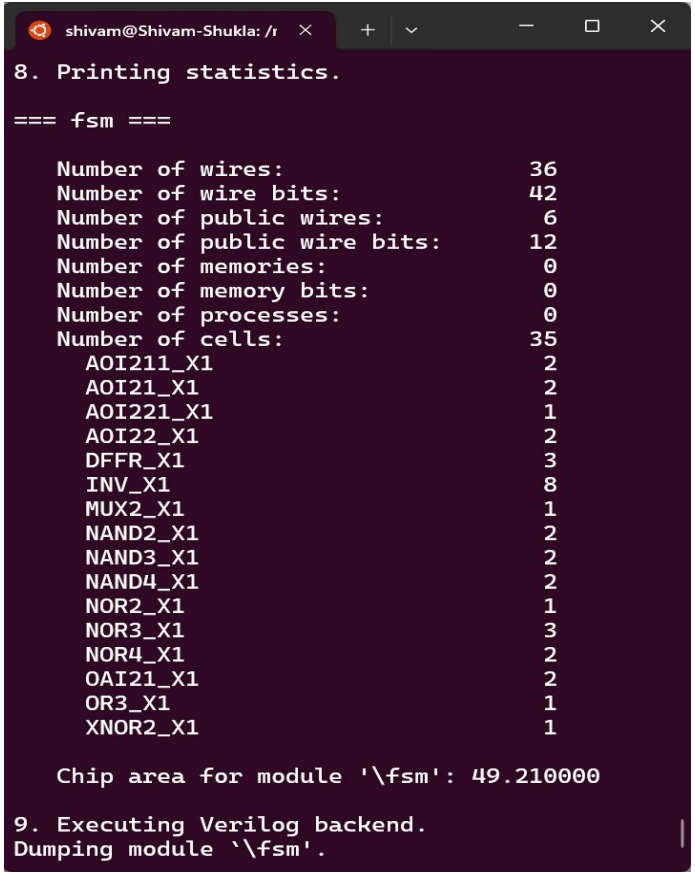
Q2 (e) Code synthesis using YOSYS

Synthesized the FSM code using YOSYS and NanGate_45nm_OCL_v2010_12 technology library.

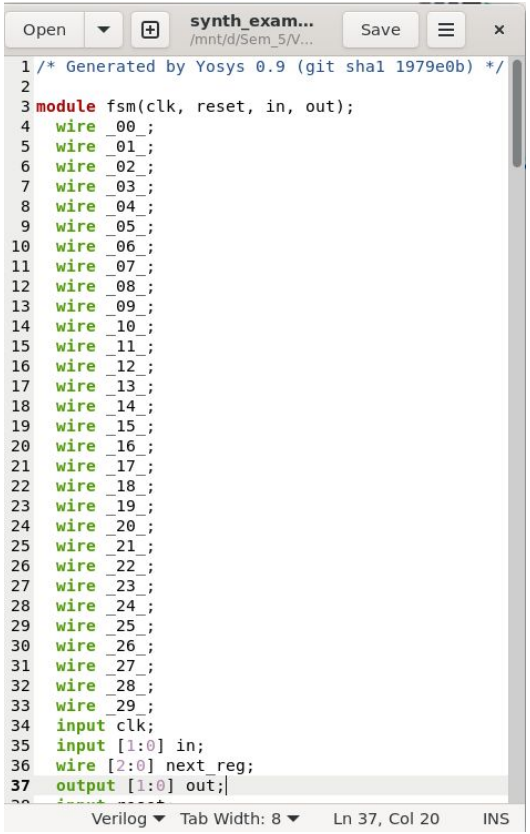
The statistics shows the hardware structure, including:

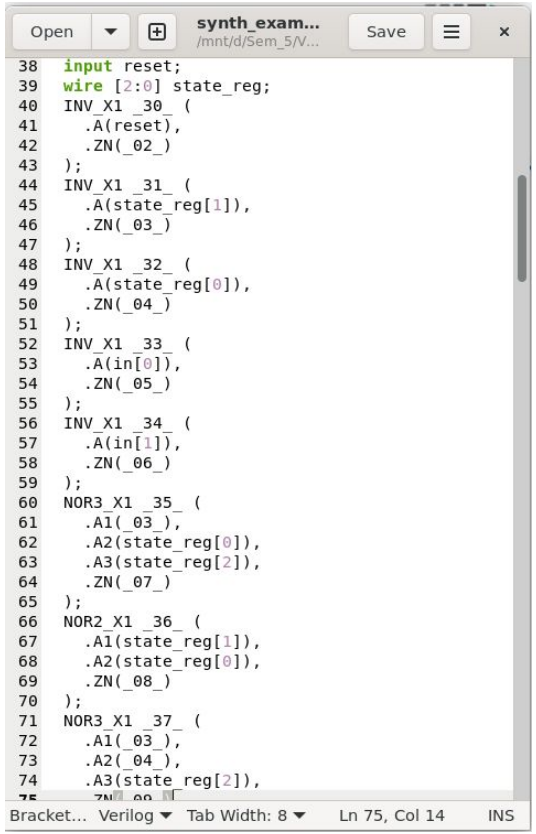
- Wires: 36 wires and 42 wire bits connect the logic elements.
- Logic Cells: 35 cells, including inverters, NOR, NAND, AOI gates, and 3 flip-flops for storing state.
- Area: The total area is 49.21, showing moderate complexity.

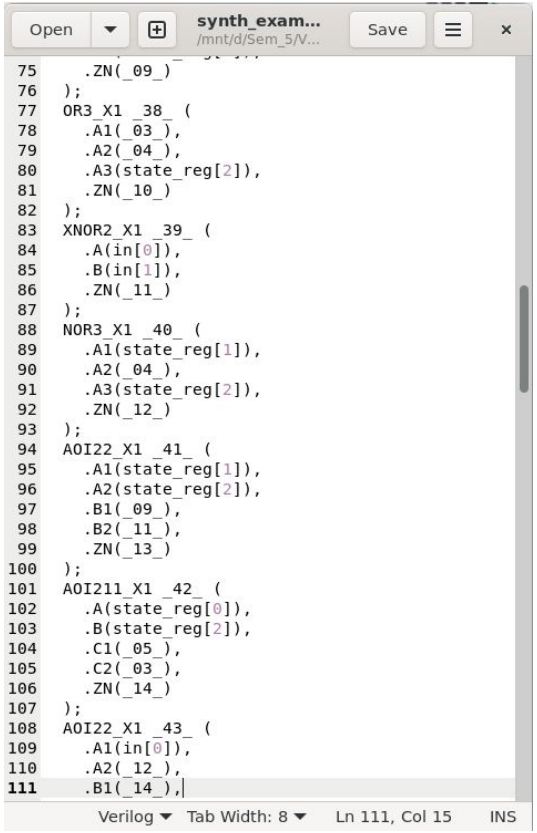
The FSM uses basic gates and flip-flops efficiently, with 3 bits representing the state. The logic is optimized, balancing complexity and area.



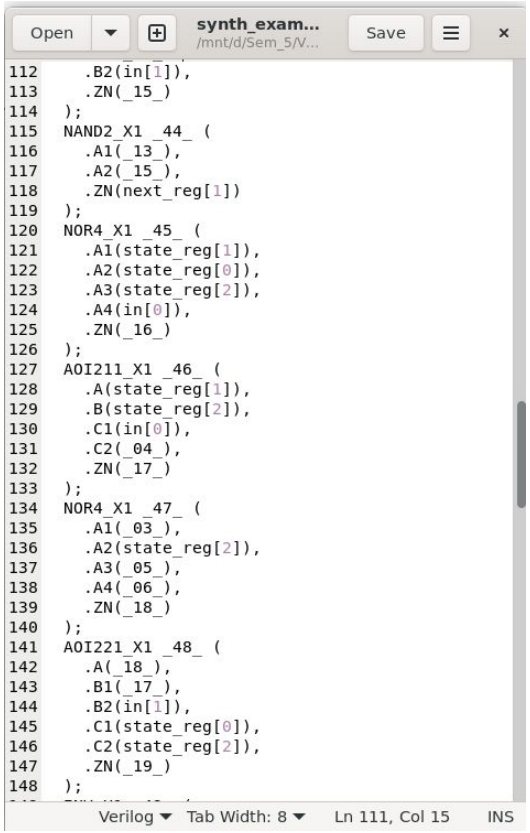
Q2 (e) The synthesized netlist

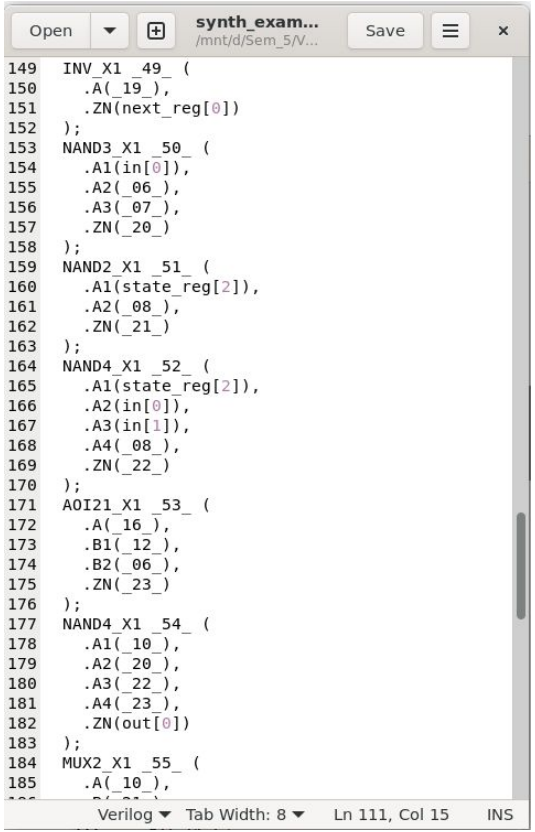


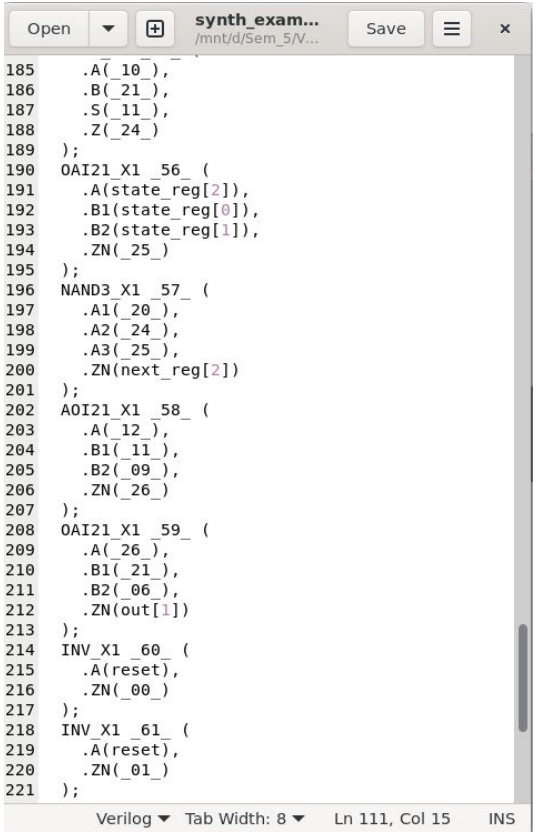




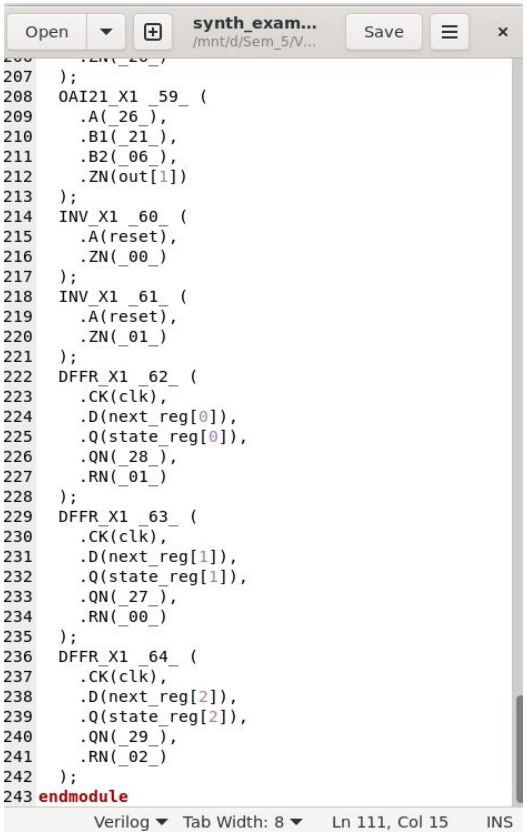
Q2 (e) The synthesized netlist







Q2 (e) The synthesized netlist



The netlist is an optimized version of the FSM that translates FSM into basic gates and flip-flops, making it ready for hardware implementation.

- Key Components: Inputs are clk, reset, and a 2-bit input (in), while the output is a 2-bit signal (out). The FSM transitions between states based on the state register and inputs.
- Flip-Flops (DFFR_X1): Three D flip-flops store the FSM state, updating on the clock's rising edge and resetting when required.
  .
- Logic Gates: Combinational gates like inverters, NORs, AOI, and multiplexers handle state transitions and output generation.

The result confirms that the design is efficient and optimized for further use in hardware.

Q2 (f) Re-write the Verilog code using one-hot encoding

```
module fsm_opt (
 input clk,            //Clock signal
 input reset,          //Reset signal
 input [1:0] in,       //2-bit input signal
 output reg [1:0] out   //2-bit output signal
);

//States bits increased
parameter S1 = 5'b00001;
parameter S2 = 5'b00010;
parameter S3 = 5'b00100;
parameter S4 = 5'b01000;
parameter S5 = 5'b10000;

//State reg and next state reg
reg [4:0] state_reg, next_reg;
//bits increased to 5
```

```
//State transition
always @(posedge clk or posedge reset)
begin
 if (reset) begin
 state_reg <= S1;  //Reset to initial state
S1
 end else begin
 state_reg <= next_reg;  //Transition to
next state
 end
 end

always @(*) begin
 //Default values
 next_reg=state_reg;
 out=2'b00;
```

```
//Switch cases for states and outputs
case (state_reg)
S1: begin
case (in)
2'b00: begin
out=2'b01;
next_reg=S1;
end
2'b01: begin
out=2'b00;
next_reg=S1;
end
2'b10: begin
out=2'b01;
next_reg=S2;
end
```

Q2 (f) Re-write the Verilog code using one-hot encoding

```
2'b11: begin
out=2'b00;
next_reg=S3;
end
endcase
end
S2: begin
case (in)
2'b00: begin
out=2'b11;
next_reg=S1;
end
2'b01: begin
out=2'b11;
next_reg=S3;
end
2'b10: begin
out=2'b10;
next_reg=S2;
end
```

```
2'b11: begin
out=2'b10;
next_reg=S4;
end
endcase
end
S3: begin
case (in)
2'b00: begin
out=2'b00;
next_reg=S1;
end
2'b01: begin
out=2'b01;
next_reg=S5;
end
2'b10: begin
out=2'b00;
next_reg=S3;
end
```

```
2'b11: begin
out=2'b00;
next_reg=S4;
end
endcase
end
S4: begin
case (in)
2'b00: begin
out=2'b11;
next_reg=S3;
end
2'b01: begin
out=2'b01;
next_reg=S5;
end
2'b10: begin
out=2'b01;
next_reg=S5;
end
```

Q2 (f) Re-write the Verilog code using one-hot encoding

```
2'b11: begin
out=2'b11;
next_reg=S4;
end
endcase
end
S5: begin
case (in)
2'b00: begin
out=2'b00;
next_reg=S5;
end
2'b01: begin
out=2'b00;
next_reg=S1;
end
```

```
2'b10: begin
out=2'b10;
next_reg=S1;
end
2'b11: begin
out=2'b11;
next_reg=S5;
end
endcase
end
endcase
end
endmodule
```

PS: I've combined the code snippets in sequence, with each section following after the white rectangles. Can also find the code here

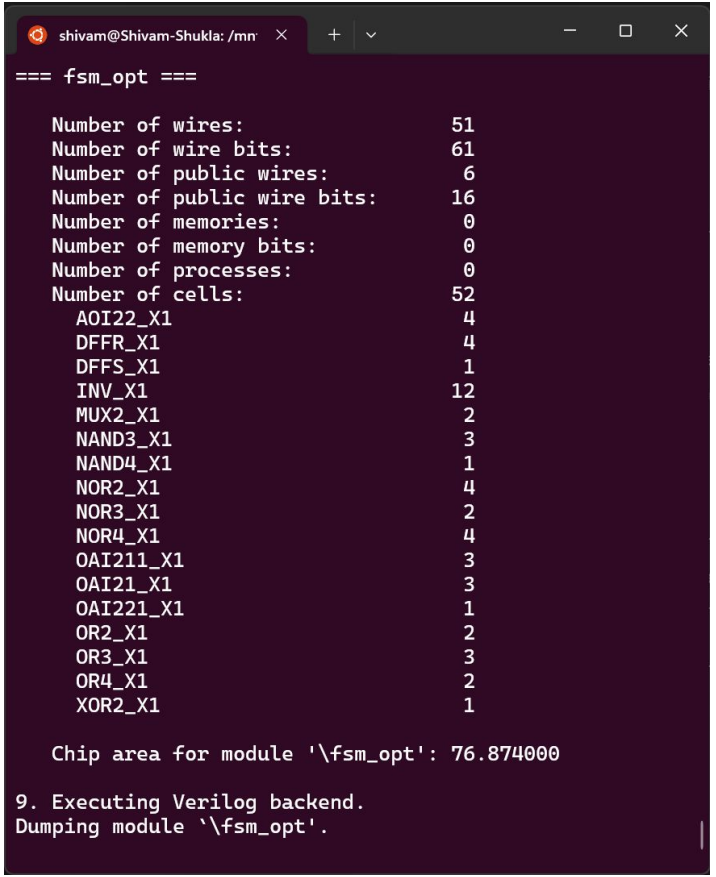
Q2 (f) One-hot encoding code synthesis using YOSYS

Synthesized the FSM using YOSYS with one-hot encoding and NanGate_45nm_OCL_v2010_12 technology library.

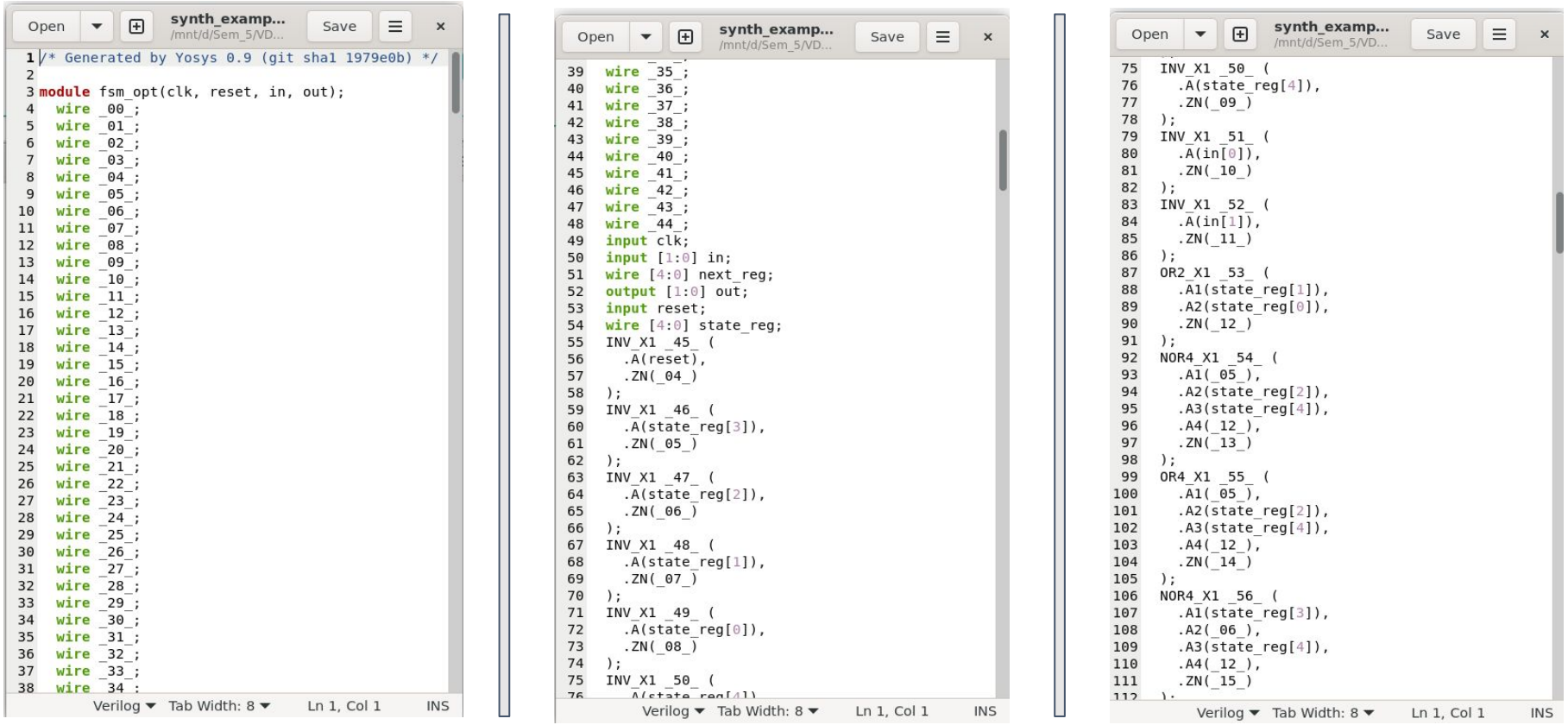
The synthesized netlist reveals the following hardware structure:

- Wires: 51 wires and 61 wire bits connect various logic elements.
- Logic Cells: 52 cells, including inverters, NOR, NAND, AOI gates, and 5 flip-flops to store the state.
- Area: The total area is 76.874, indicating an increase in complexity compared to binary encoding.

The FSM design efficiently utilizes basic gates and flip-flops with one-hot encoding. Each state is represented with a unique bit, optimizing state transitions while slightly increasing area usage.



Q2 (f) The synthesized netlist

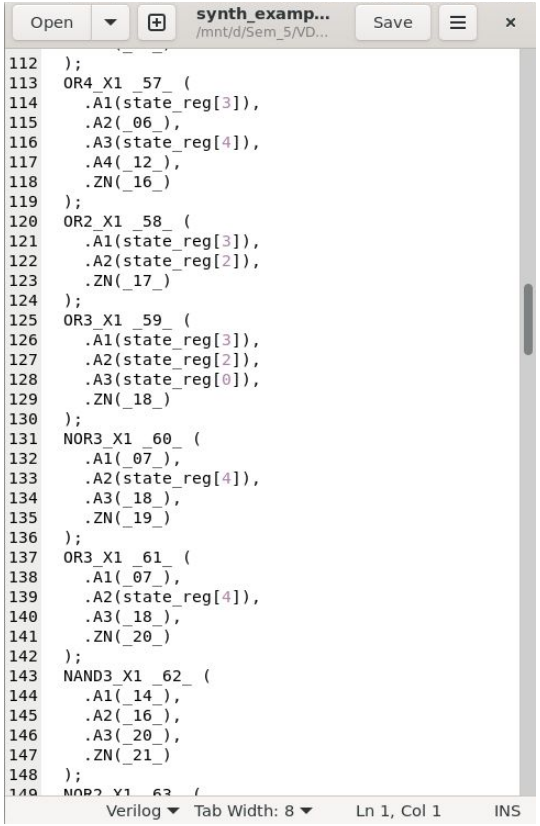


Open synth_examp... /mnt/d/Sem_5/VD... Save

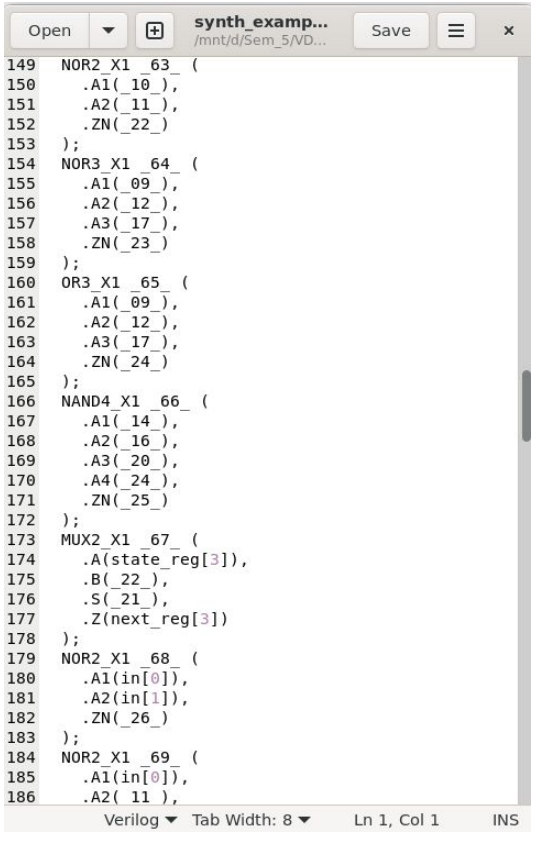
```
/* Generated by Yosys 0.9 (git sha1 1979e0b) */

module fsm_opt(clk, reset, in, out);
  wire _00_;
  wire _01_;
  wire _02_;
  wire _03_;
  wire _04_;
  wire _05_;
  wire _06_;
  wire _07_;
  wire _08_;
  wire _09_;
  wire _10_;
  wire _11_;
  wire _12_;
  wire _13_;
  wire _14_;
  wire _15_;
  wire _16_;
  wire _17_;
  wire _18_;
  wire _19_;
  wire _20_;
  wire _21_;
  wire _22_;
  wire _23_;
  wire _24_;
  wire _25_;
  wire _26_;
  wire _27_;
  wire _28_;
  wire _29_;
  wire _30_;
  wire _31_;
  wire _32_;
  wire _33_;
  wire _34_;
  wire _35_;
  wire _36_;
  wire _37_;
  wire _38_;
  wire _39_;
  wire _40_;
  wire _41_;
  wire _42_;
  wire _43_;
  wire _44_;
  input clk;
  input [1:0] in;
  wire [4:0] next_reg;
  output [1:0] out;
  input reset;
  wire [4:0] state_reg;
  INV_X1 _45_ (
    .A(reset),
    .ZN(_04_)
  );
  INV_X1 _46_ (
    .A(state_reg[3]),
    .ZN(_05_)
  );
  INV_X1 _47_ (
    .A(state_reg[2]),
    .ZN(_06_)
  );
  INV_X1 _48_ (
    .A(state_reg[1]),
    .ZN(_07_)
  );
  INV_X1 _49_ (
    .A(state_reg[0]),
    .ZN(_08_)
  );
  INV_X1 _50_ (
    .A(state_reg[4]),
    .ZN(_09_)
  );
  INV_X1 _51_ (
    .A(in[0]),
    .ZN(_10_)
  );
  INV_X1 _52_ (
    .A(in[1]),
    .ZN(_11_)
  );
  OR2_X1 _53_ (
    .A1(state_reg[1]),
    .A2(state_reg[0]),
    .ZN(_12_)
  );
  NOR4_X1 _54_ (
    .A1(_05_),
    .A2(state_reg[2]),
    .A3(state_reg[4]),
    .A4(_12_),
    .ZN(_13_)
  );
  OR4_X1 _55_ (
    .A1(_05_),
    .A2(state_reg[2]),
    .A3(state_reg[4]),
    .A4(_12_),
    .ZN(_14_)
  );
  NOR4_X1 _56_ (
    .A1(state_reg[3]),
    .A2(_06_),
    .A3(state_reg[4]),
    .A4(_12_),
    .ZN(_15_)
  );
```

Verilog Tab Width: 8 Ln 1, Col 1 INS

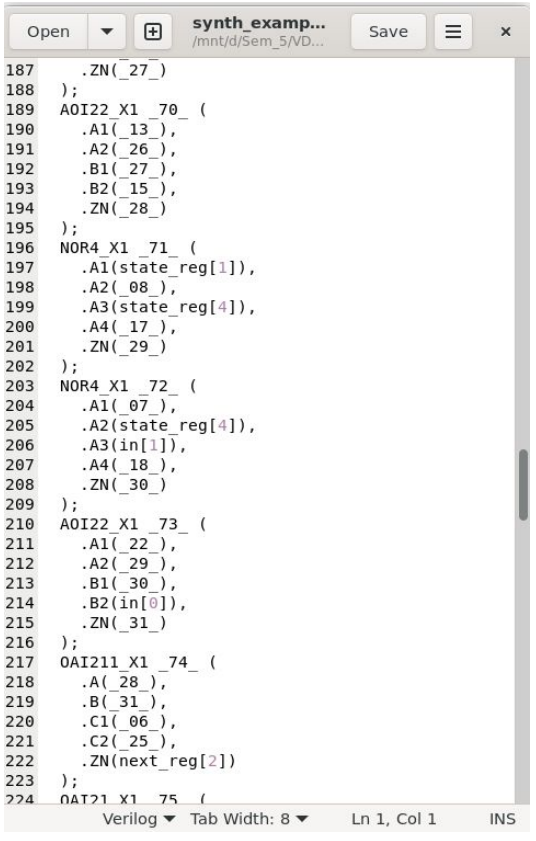
Q2 (f) The synthesized netlist



```
112  );
113  OR4_X1 _57_ (
114    .A1(state_reg[3]),
115    .A2(_06_),
116    .A3(state_reg[4]),
117    .A4(_12_),
118    .ZN(_16_)
119  );
120  OR2_X1 _58_ (
121    .A1(state_reg[3]),
122    .A2(state_reg[2]),
123    .ZN(_17_)
124  );
125  OR3_X1 _59_ (
126    .A1(state_reg[3]),
127    .A2(state_reg[2]),
128    .A3(state_reg[0]),
129    .ZN(_18_)
130  );
131  NOR3_X1 _60_ (
132    .A1(_07_),
133    .A2(state_reg[4]),
134    .A3(_18_),
135    .ZN(_19_)
136  );
137  OR3_X1 _61_ (
138    .A1(_07_),
139    .A2(state_reg[4]),
140    .A3(_18_),
141    .ZN(_20_)
142  );
143  NAND3_X1 _62_ (
144    .A1(_14_),
145    .A2(_16_),
146    .A3(_20_),
147    .ZN(_21_)
148  );
```

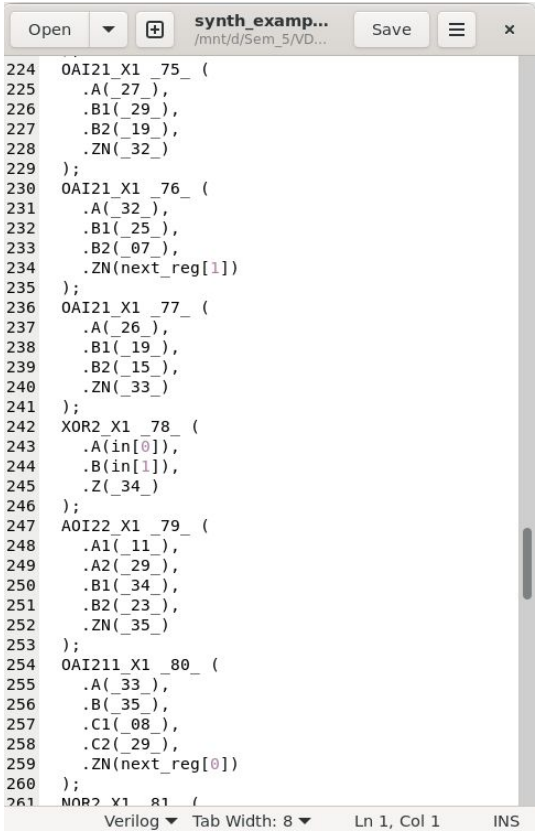


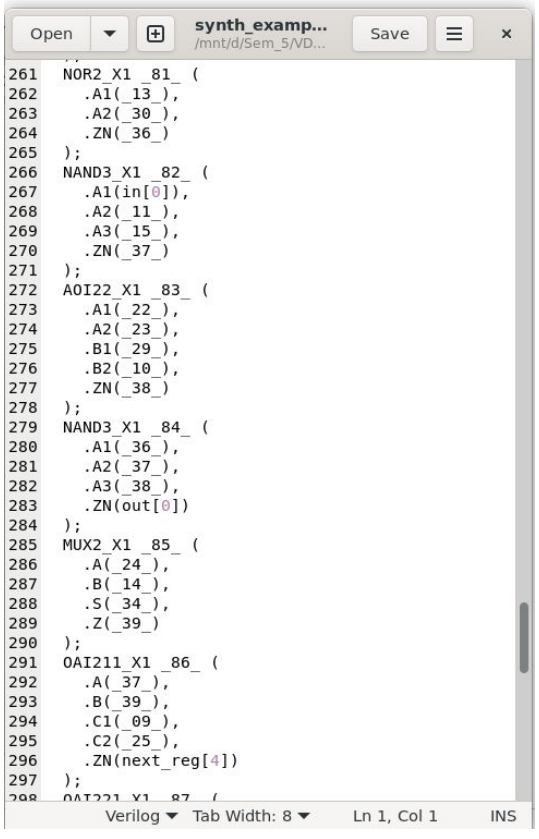
```
149  NOR2_X1 _63_ (
150    .A1(_10_),
151    .A2(_11_),
152    .ZN(_22_)
153  );
154  NOR3_X1 _64_ (
155    .A1(_09_),
156    .A2(_12_),
157    .A3(_17_),
158    .ZN(_23_)
159  );
160  OR3_X1 _65_ (
161    .A1(_09_),
162    .A2(_12_),
163    .A3(_17_),
164    .ZN(_24_)
165  );
166  NAND4_X1 _66_ (
167    .A1(_14_),
168    .A2(_16_),
169    .A3(_20_),
170    .A4(_24_),
171    .ZN(_25_)
172  );
173  MUX2_X1 _67_ (
174    .A(state_reg[3]),
175    .B(_22_),
176    .S(_21_),
177    .Z(next_reg[3])
178  );
179  NOR2_X1 _68_ (
180    .A1(in[0]),
181    .A2(in[1]),
182    .ZN(_26_)
183  );
184  NOR2_X1 _69_ (
185    .A1(in[0]),
186    .A2(_11_),
```

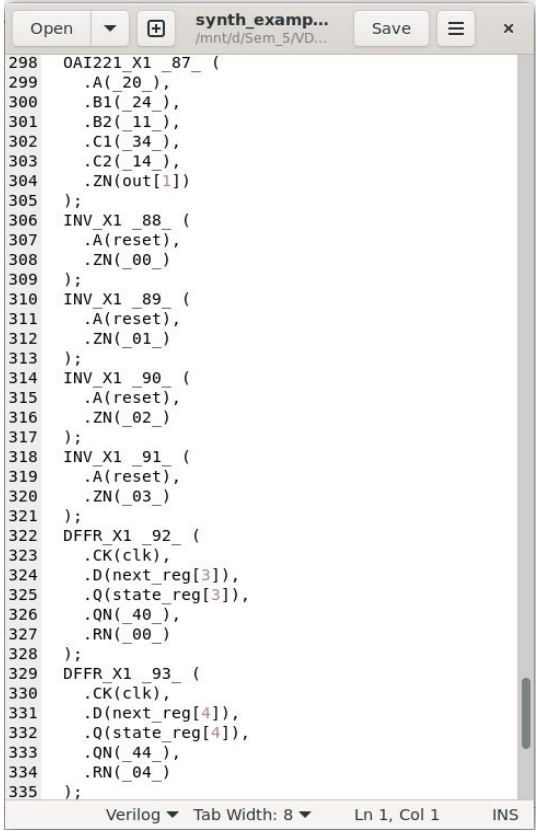


```
187    .ZN(_27_)
188  );
189  AOI22_X1 _70_ (
190    .A1(_13_),
191    .A2(_26_),
192    .B1(_27_),
193    .B2(_15_),
194    .ZN(_28_)
195  );
196  NOR4_X1 _71_ (
197    .A1(state_reg[1]),
198    .A2(_08_),
199    .A3(state_reg[4]),
200    .A4(_17_),
201    .ZN(_29_)
202  );
203  NOR4_X1 _72_ (
204    .A1(_07_),
205    .A2(state_reg[4]),
206    .A3(in[1]),
207    .A4(_18_),
208    .ZN(_30_)
209  );
210  AOI22_X1 _73_ (
211    .A1(_22_),
212    .A2(_29_),
213    .B1(_30_),
214    .B2(in[0]),
215    .ZN(_31_)
216  );
217  OAI211_X1 _74_ (
218    .A(_28_),
219    .B(_31_),
220    .C1(_06_),
221    .C2(_25_),
222    .ZN(next_reg[2])
223  );
```

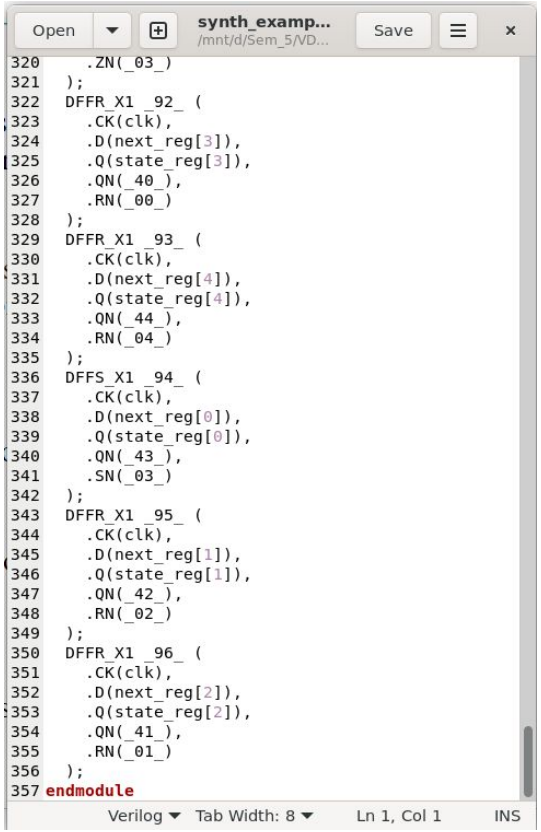
Q2 (f) The synthesized netlist







Q2 (f) The synthesized netlist



The netlist is optimized for one-hot encoding, with five flip-flops representing the FSM states.

- Inputs: Clock (clk), reset (reset), and a 2-bit input (in).
- Outputs: A 2-bit output (out) driven by state transitions.
- Flip-Flops: Five D flip-flops store the FSM state, with each state represented by one bit in the state register.
- Logic Gates: Inverters, NOR, NAND, AOI gates, and multiplexers manage state transitions and output logic.

The design efficiently translates the FSM to hardware using simple combinational logic and flip-flops, balancing hardware complexity and performance.

Q2 (f) Comparison of One-Hot vs Binary Encoded FSM:

- Wires & Wire Bits: One-hot encoding uses 51 wires and 61 wire bits, whereas binary encoding uses 36 wires and 42 wire bits. One-hot requires more connections due to the increased number of states represented by individual bits.

- Logic Cells: One-hot uses 52 cells, while binary encoding uses 35. This indicates a higher complexity in one-hot encoding as each state has its own flip-flop.

- Flip-Flops: One-hot uses 5 flip-flops, representing each state individually, compared to 3 flip-flops in binary encoding, which requires fewer flip-flops due to compact state representation.

- Area: One-hot encoding consumes more area (76.874 vs 49.21) due to the increased hardware complexity of using more flip-flops and gates for each state.

Key Inference:
One-hot encoding simplifies state transition logic but increases hardware complexity and area, while binary encoding is more area-efficient, using fewer flip-flops at the cost of slightly more complex transition logic.

PS: All the codes can be found here .

Thank You

INDRAPRASTHA INSTITUTE *of*
INFORMATION TECHNOLOGY
DELHI