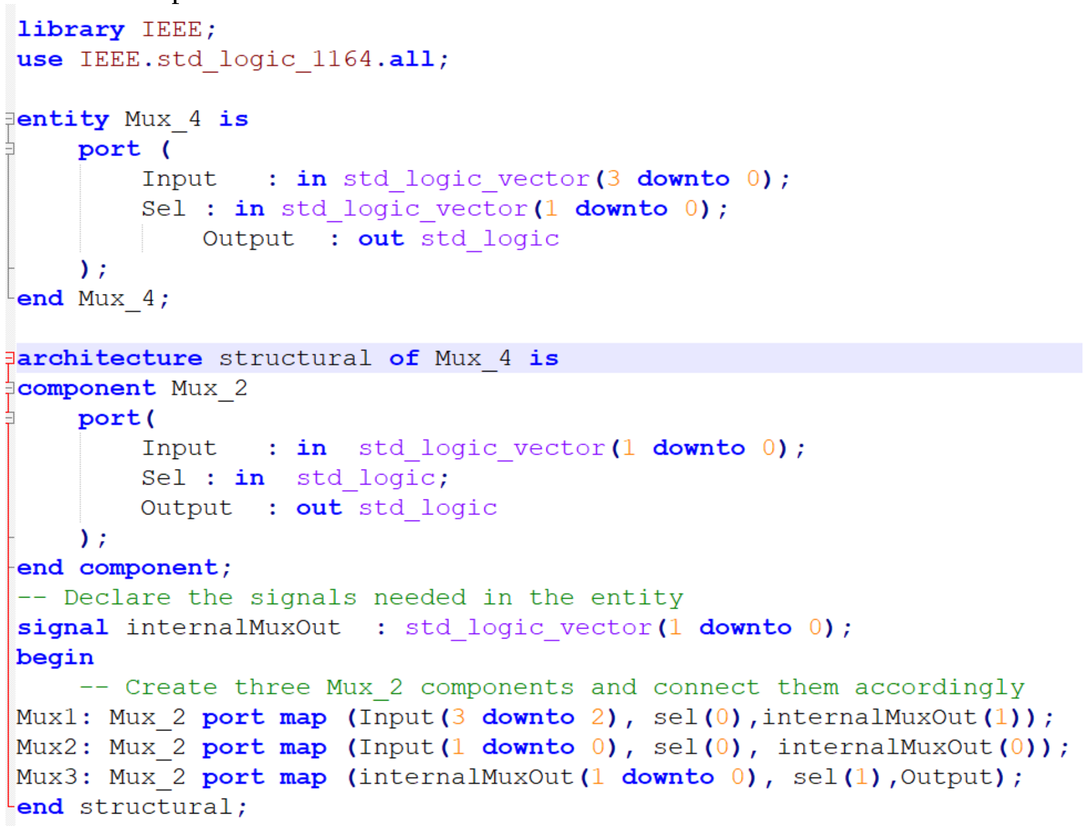
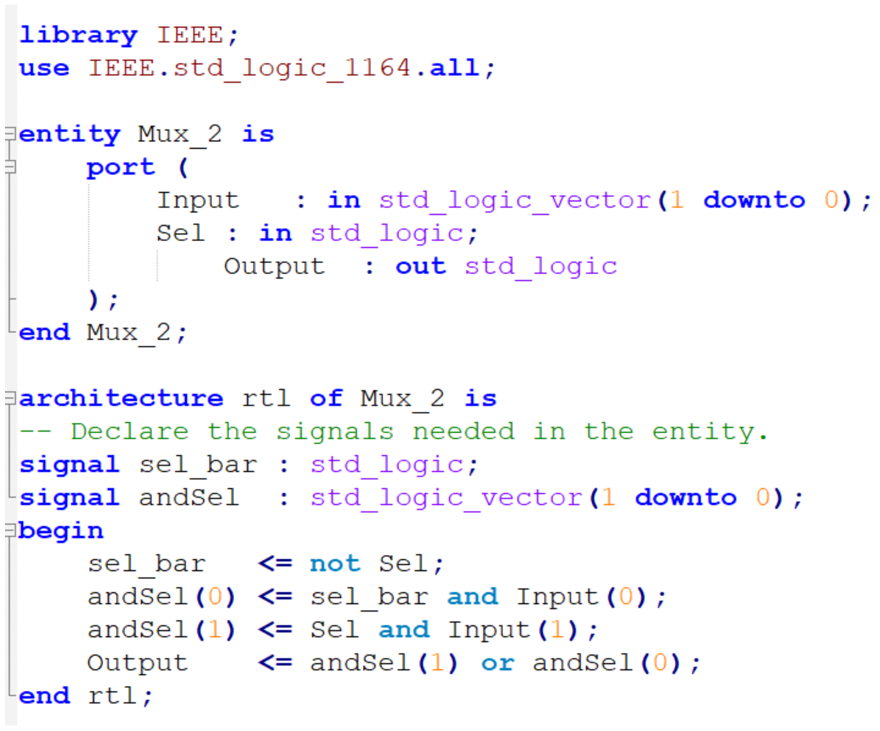
Reference Solutions: Assignment #1 – ECED 4260

1. A 4-to-1 multiplexer using 2-to-1 multiplexer as components

VHDL examples:



```vhdl
library IEEE;
use IEEE.std_logic_1164.all;

entity Mux_4 is
    port (
        Input   : in std_logic_vector(3 downto 0);
        Sel : in std_logic_vector(1 downto 0);
            Output  : out std_logic
    );
end Mux_4;

architecture structural of Mux_4 is
component Mux_2
    port(
        Input   : in  std_logic_vector(1 downto 0);
        Sel : in  std_logic;
        Output  : out std_logic
    );
end component;
-- Declare the signals needed in the entity
signal internalMuxOut  : std_logic_vector(1 downto 0);
begin
    -- Create three Mux_2 components and connect them accordingly
Mux1: Mux_2 port map (Input(3 downto 2), sel(0),internalMuxOut(1));
Mux2: Mux_2 port map (Input(1 downto 0), sel(0), internalMuxOut(0));
Mux3: Mux_2 port map (internalMuxOut(1 downto 0), sel(1),Output);
end structural;
```



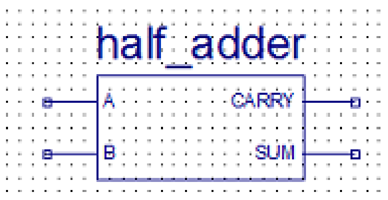
```vhdl
library IEEE;
use IEEE.std_logic_1164.all;

entity Mux_2 is
    port (
        Input   : in std_logic_vector(1 downto 0);
        Sel : in std_logic;
            Output  : out std_logic
    );
end Mux_2;

architecture rtl of Mux_2 is
-- Declare the signals needed in the entity.
signal sel_bar : std_logic;
signal andSel  : std_logic_vector(1 downto 0);
begin
    sel_bar   <= not Sel;
    andSel(0) <= sel_bar and Input(0);
    andSel(1) <= Sel and Input(1);
    Output    <= andSel(1) or andSel(0);
end rtl;
```

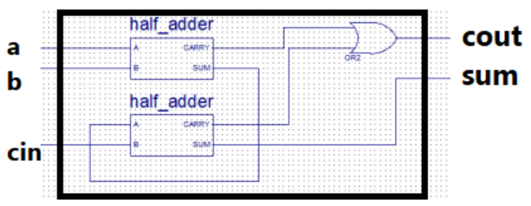
Verilog MUX:

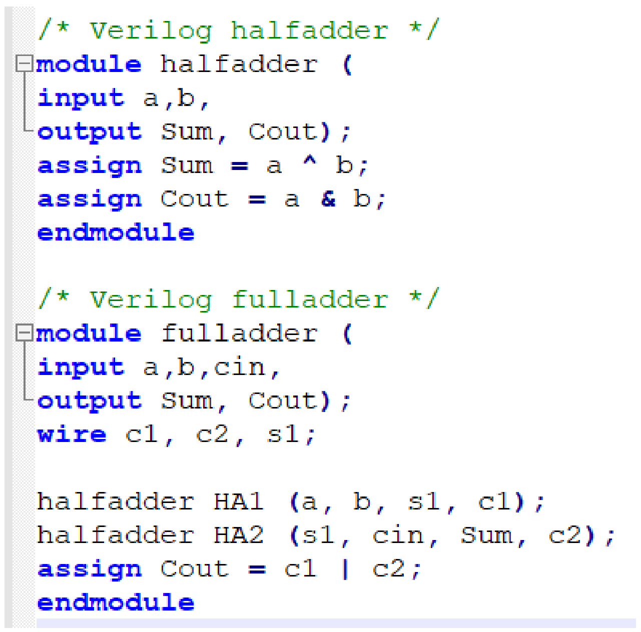
```
`timescale 1ns / 1ps

module Mux_4 (
        input [3:0] in,
        input [1:0] sel,
        output out);

        // Declare wires needed for entity
        wire [1:0] internalMuxOut;

        // Create the 3 mux components needed
        Mux_2 m1 (in[3:2], sel[0], internalMuxOut[1]);
        Mux_2 m2 (in[1:0], sel[0], internalMuxOut[0]);
        Mux_2 m3 (internalMuxOut[1:0], sel[1], out);

endmodule
```

```
`timescale 1ns / 1ps

module Mux_2 (
        input [1:0] in,
        input sel,
        output out);

        wire sel_bar;
        wire [1:0] andsel;

        assign sel_bar = ~sel;
        assign andsel[0] = sel_bar & in[0];
        assign andsel[1] = sel & in[1];
        assign out = andsel[1] | andsel[0];

endmodule
```

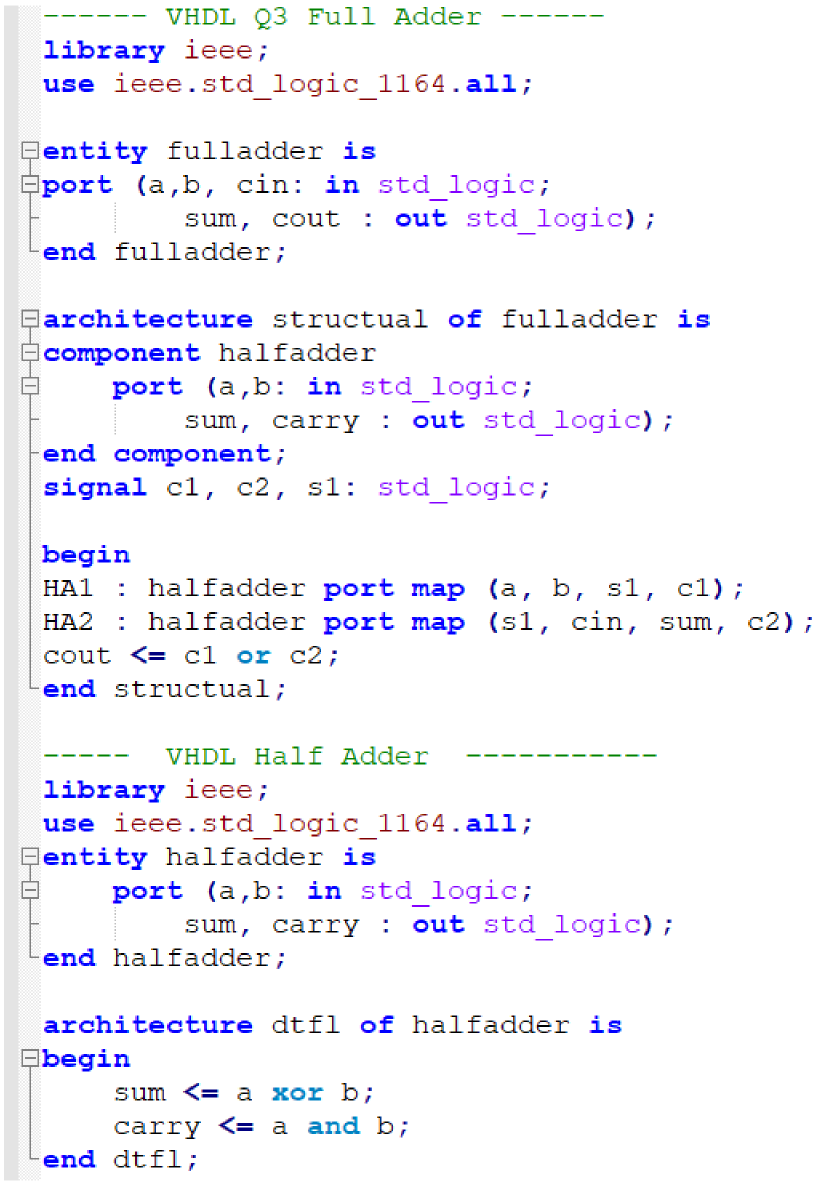
2. Half Adder and Full Adder:





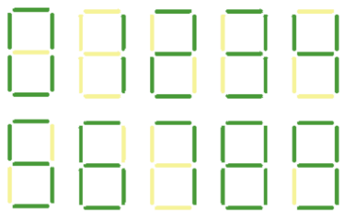


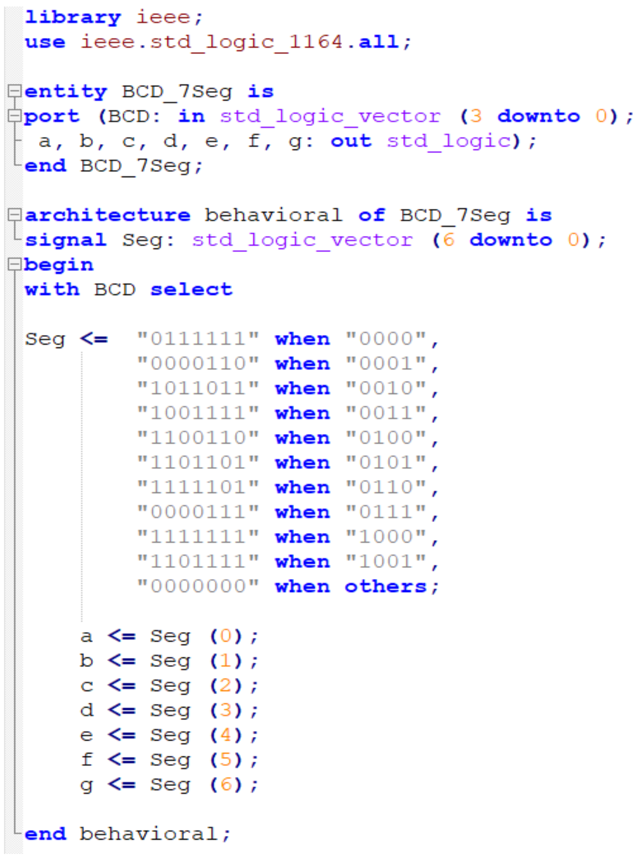
```verilog
/* Verilog halfadder */
module halfadder (
input a,b,
output Sum, Cout);
assign Sum = a ^ b;
assign Cout = a & b;
endmodule

/* Verilog fulladder */
module fulladder (
input a,b,cin,
output Sum, Cout);
wire c1, c2, s1;

halfadder HA1 (a, b, s1, c1);
halfadder HA2 (s1, cin, Sum, c2);
assign Cout = c1 | c2;
endmodule
```



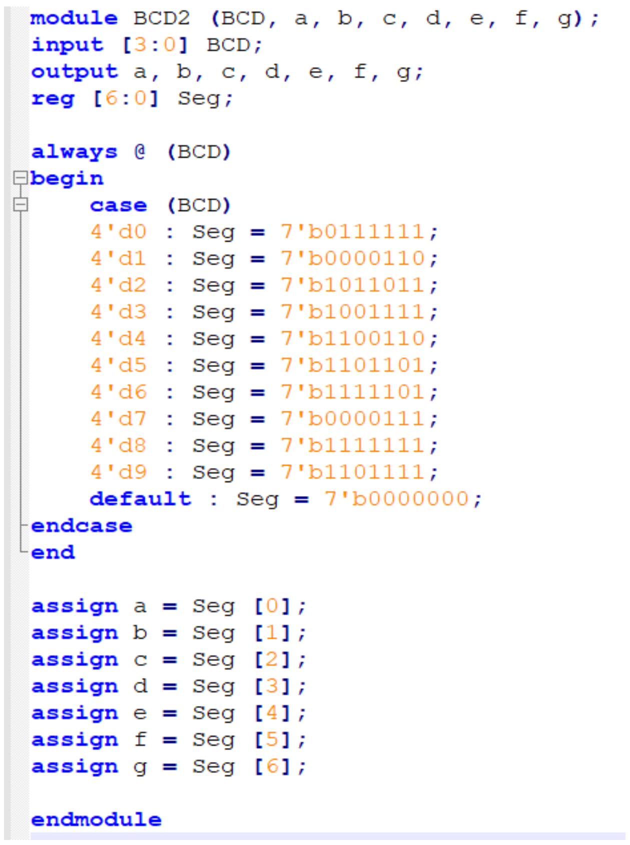
```vhdl
------ VHDL Q3 Full Adder ------
library ieee;
use ieee.std_logic_1164.all;

entity fulladder is
port (a,b, cin: in std_logic;
        sum, cout : out std_logic);
end fulladder;

architecture structual of fulladder is
component halfadder
    port (a,b: in std_logic;
        sum, carry : out std_logic);
end component;
signal c1, c2, s1: std_logic;

begin
HA1 : halfadder port map (a, b, s1, c1);
HA2 : halfadder port map (s1, cin, sum, c2);
cout <= c1 or c2;
end structual;

-----  VHDL Half Adder  -----------
library ieee;
use ieee.std_logic_1164.all;
entity halfadder is
    port (a,b: in std_logic;
        sum, carry : out std_logic);
end halfadder;

architecture dtfl of halfadder is
begin
    sum <= a xor b;
    carry <= a and b;
end dtfl;
```

3. Design a Binary Coded Decimal (BCD) – to - 7-segment display decoder using VHDL or Verilog. The seven outputs of the decoder, ***a*** to ***g***, are active high (i.e. a ‘1’ will light up the corresponding segment)





```
library ieee;
use ieee.std_logic_1164.all;

entity BCD_7Seg is
port (BCD: in std_logic_vector (3 downto 0);
 a, b, c, d, e, f, g: out std_logic);
end BCD_7Seg;

architecture behavioral of BCD_7Seg is
signal Seg: std_logic_vector (6 downto 0);
begin
with BCD select

Seg <=   "0111111" when "0000",
         "0000110" when "0001",
         "1011011" when "0010",
         "1001111" when "0011",
         "1100110" when "0100",
         "1101101" when "0101",
         "1111101" when "0110",
         "0000111" when "0111",
         "1111111" when "1000",
         "1101111" when "1001",
         "0000000" when others;

    a <= Seg (0);
    b <= Seg (1);
    c <= Seg (2);
    d <= Seg (3);
    e <= Seg (4);
    f <= Seg (5);
    g <= Seg (6);

end behavioral;
```

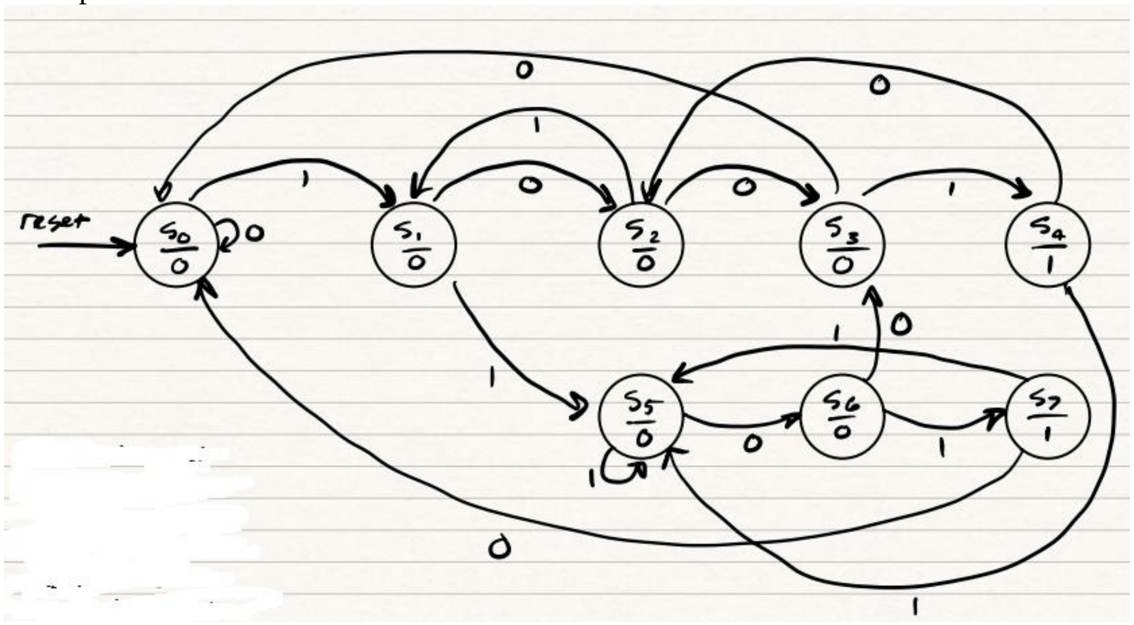


```
module BCD2 (BCD, a, b, c, d, e, f, g);
input [3:0] BCD;
output a, b, c, d, e, f, g;
reg [6:0] Seg;

always @ (BCD)
begin
    case (BCD)
    4'd0 : Seg = 7'b0111111;
    4'd1 : Seg = 7'b0000110;
    4'd2 : Seg = 7'b1011011;
    4'd3 : Seg = 7'b1001111;
    4'd4 : Seg = 7'b1100110;
    4'd5 : Seg = 7'b1101101;
    4'd6 : Seg = 7'b1111101;
    4'd7 : Seg = 7'b0000111;
    4'd8 : Seg = 7'b1111111;
    4'd9 : Seg = 7'b1101111;
    default : Seg = 7'b0000000;
endcase
end

assign a = Seg [0];
assign b = Seg [1];
assign c = Seg [2];
assign d = Seg [3];
assign e = Seg [4];
assign f = Seg [5];
assign g = Seg [6];

endmodule
```

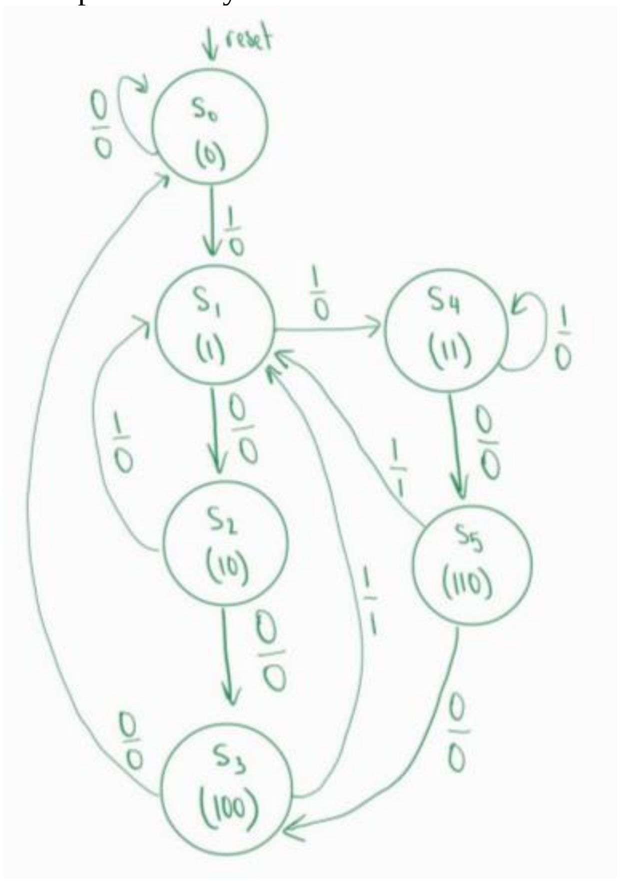
.

4 (a). Reference state diagrams:

Example 1. Moore FSM



Example 2. Mealy FSM



4 (b) (omitted) State optimization, binary state assignment, next-state and output logics.
4 (c)(d) Example Verilog Mealy FSM:

```
// overlapping pattern detector module
module overlappingPatternDetector(
    input wire clk, // input clock signal
    input wire rst, // input reset signal
    input wire in,  // input bit
    output reg out, // output match flag
);
    // State encoding
    parameter S0 = 3'b000, S1 = 3'b001, S2 = 3'b010,
              S3 = 3'b011, S4 = 3'b100, S5 = 3'b101;

    // internal variables to store states
    reg [2:0] state, next_state;

    // State transition logic
    always @(posedge clk or posedge rst) begin
        if (rst)
            state <= S0;  // Reset state
        else
            state <= next_state;  // Move to next state
    end

    // Next state logic
    always @(*) begin
        case (state)
            S0: next_state = (in) ? S1 : S0;  // S0 -> S1 if input = 1, else
stay in S0
            S1: next_state = (in) ? S4 : S2;  // S1 -> S4 if input = 1, else
S2
            S2: next_state = (in) ? S1 : S3;  // S2 -> S1 if input = 1, else
S3
            S3: next_state = (in) ? S1 : S0;  // S3 -> S1 if input = 1, else
S0
            S4: next_state = (in) ? S4 : S5;  // S4 -> S5 if input = 0, else
stay in S4
            S5: next_state = (in) ? S1 : S3;  // S5 -> S1 if input = 1, else
S3
            default: next_state = S0;         // Default state is S0

        endcase
    end

    // Output logic
    always @(posedge clk or posedge rst) begin
        if (rst)
            out <= 0; // reset output
        else
            out <= (state == S3 && in) || (state == S5 && in);  // Output
logic
    end

endmodule
```

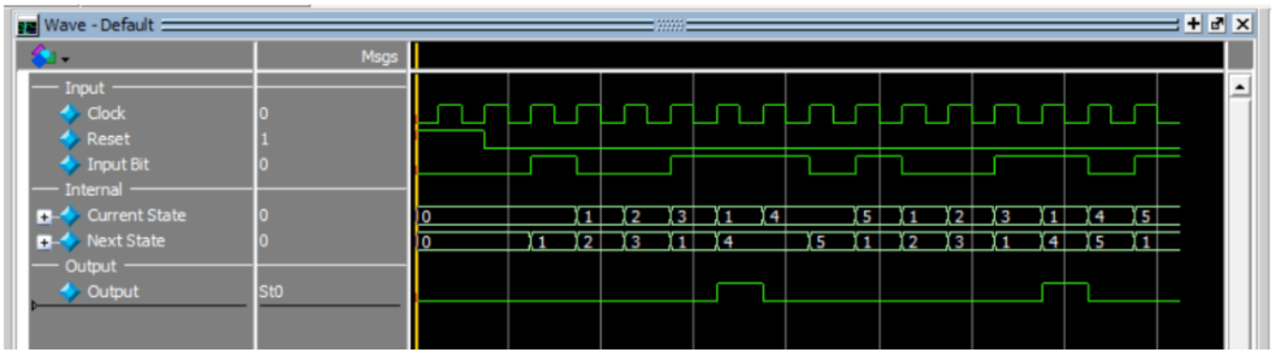
Please note:

You can code your FSM based on the optimized design from 4 (b):
For example:

```
module [redacted](clk,rst,I,O);
    input  clk,rst,I;
    output O;
    reg O;
    reg [2:0] state;

    parameter s0 = 3'b000,
                      s1 = 3'b001,
              s2 = 3'b010,
              s3 = 3'b101,
              s4 = 3'b100;

    always@(posedge clk or posedge rst)
    begin
        if(rst == 1'b1)
          begin
            state <= s0;
            O  <= 1'b0;
          end
        else
          begin
            case(state)
            s0:
              begin
```

……

Sample waveform:


Wave - Default
Msgs
Input
Clock
Reset
Input Bit
Internal
Current State
Next State
Output
Output
0
1
0
0
0
St0
0
1
2
3
1
4
5
1
2
3
1
4
5
0
1
2
3
1
4
5
1
2
3
1
4
5
1