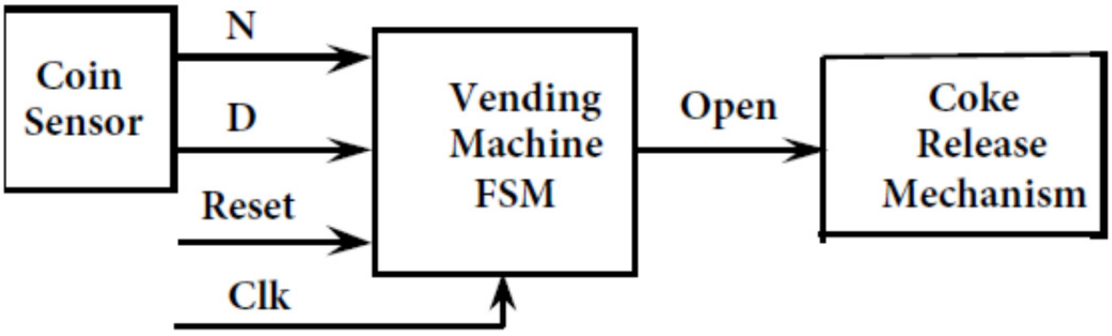
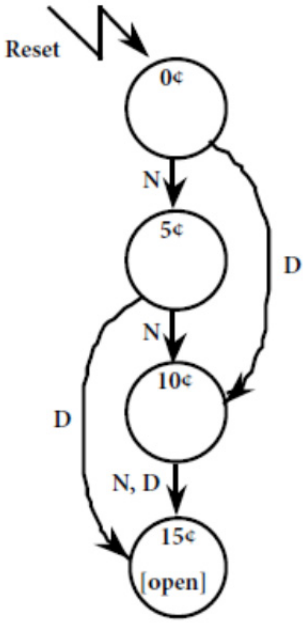
CSE 308: Digital System Design | Homework 5 Solution

Problem 1: You are working as a Hardware Design Engineer at ASML in the Netherlands (just suppose!). Your boss asked you to design a vending machine with the following specifications.

- delivers a coke after 15 cents deposited
- has a single coin slot for dimes and nickels
- does not return change



Block diagram of the vending machine

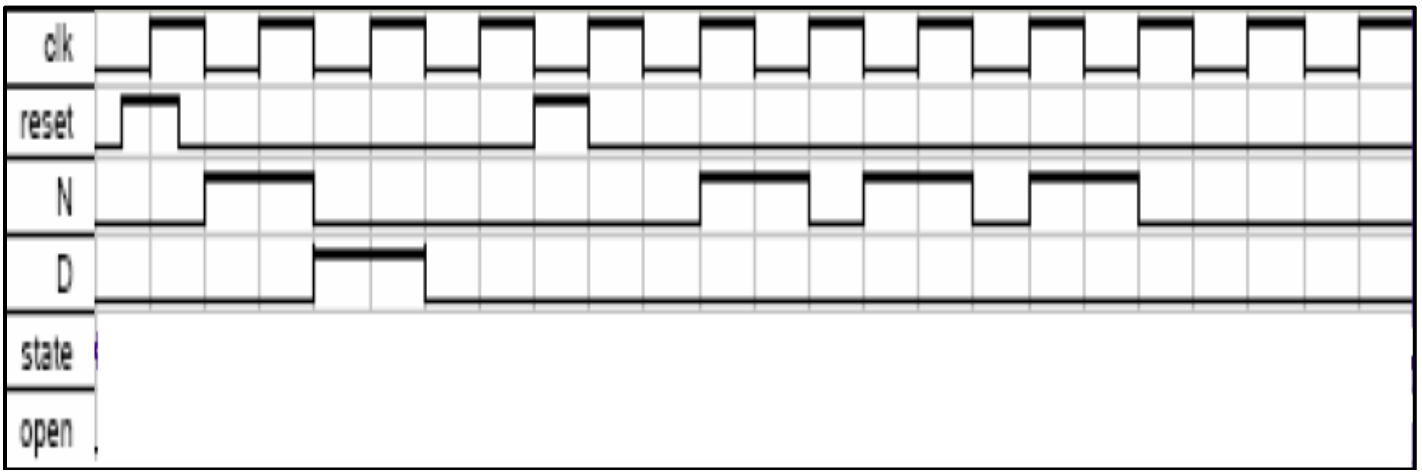


Present State	Inputs D	Inputs N	Next State	Output Open
0¢	0	0	0¢	0
	0	1	5¢	0
	1	0	10¢	0
	1	1	X	X
5¢	0	0	5¢	0
	0	1	10¢	0
	1	0	15¢	0
	1	1	X	X
10¢	0	0	10¢	0
	0	1	15¢	0
	1	0	15¢	0
	1	1	X	X
15¢	X	X	15¢	1

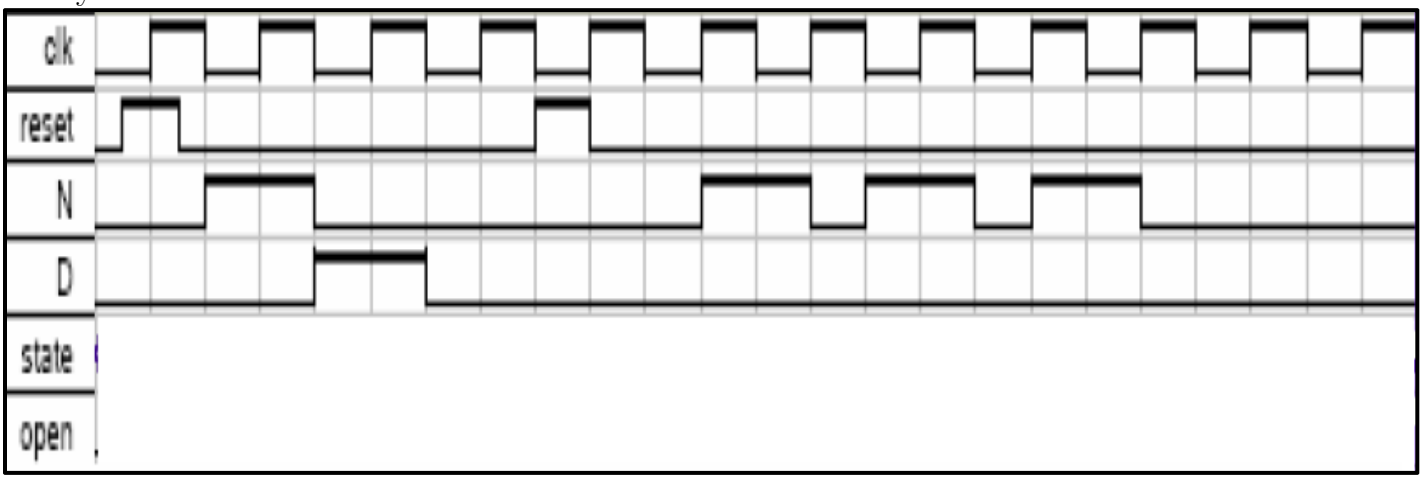
State diagram and state table of the vending machine

(a) Draw a Moore FSM for the above vending machine.
(b) Draw a Mealy FSM for the above vending machine.
(c) Implement the FSMs in (a) and (b) in Verilog.
(d) Show simulation results of your FSMs from (c) for the following values of clk, reset, N, and D.

Moore FSM



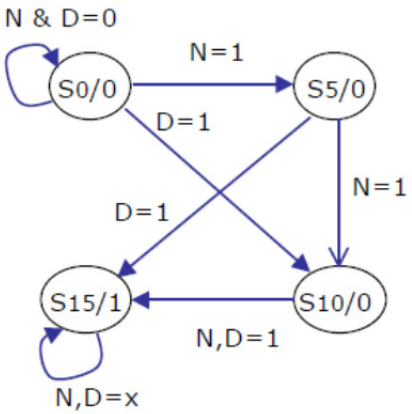
Mealy FSM



Solution:

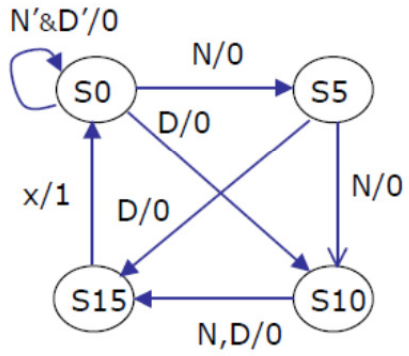
Vending Machine: Verilog (Moore FSM)

```
module vending_moore(N, D, clk, reset, open);
// Moore FSM for a vending machine
 input N, D, clk, reset;
 output open;
 reg [1:0] state;
 parameter S0=2'b00, S5=2'b01, S10=2'b10,
S15=2'b11;

// Define the sequential block
 always @(posedge reset or posedge clk)
  if (reset) state <= S0;
 else
   case (state)
          S0:  if (N) state <= S5;
                 else if (D) state <= S10;
                      else state <= S0;
          S5:  if (N)          state <= S10;
                 else if (D) state <= S15;
                      else state <= S5;
          S10: if (N) state <= S15;
                 else if (D) state <= S15;
                      else state <= S10;
          S15: state <= S15;
    endcase
// Define output during S3
          assign open = (state == S15);
endmodule
```



Vending Machine: Verilog (Mealy FSM)

```
module vending_mealy(N, D, clk, reset, open);
// Mealy FSM for a vending machine
  input N, D, clk, reset;
  output open;
  reg [1:0] pstate, nstate;
  reg open;
  parameter S0=2'b00, S5=2'b01, S10=2'b10, S15=2'b11;

// Next state and ouptut combinational logic
    always @(N or D or pstate or reset)
     if (reset)
          begin nstate = S0; open = 0; end
           else case (pstate)
              S0:  begin open = 0; if (N) nstate = S5;
                       else if (D) nstate = S10;
                          else nstate = S0; end
              S5:  begin open = 0; if (N)nstate = S10;
                       else if (D) nstate = S15;
                          else nstate = S5; end
             S10: if (N | D) begin nstate = S15; open = 0; end
                       else begin nstate = S10; open = 0; end
             S15: begin nstate = S0; open = 1; end
          endcase
// FF logic, use nonblocking assignments "<="
          always @(posedge clk)
                    pstate <= nstate;
endmodule
```

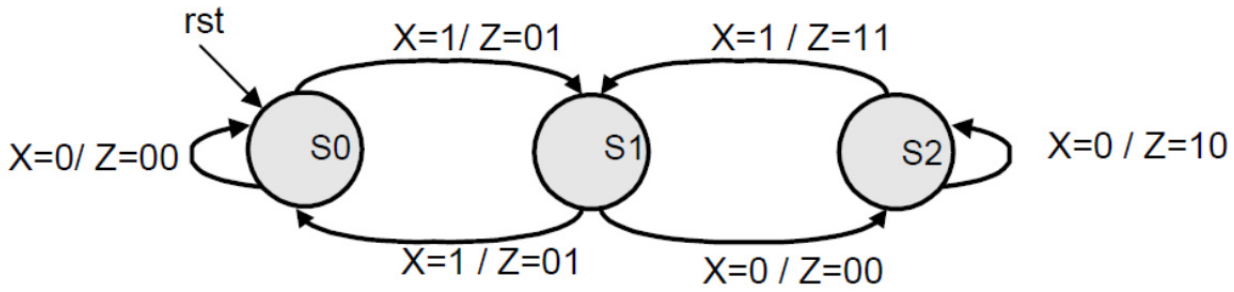


Vending Machine: Simulation Results

Moore FSM

Mealy FSM

Problem 2:



(a) Complete the Verilog code given below for the FSM shown above. The FSM has a synchronous **reset**, which forces the FSM into state S0, when **reset** is zero. State transitions occur on the positive edge of **clock**. If an invalid state is entered, the **state** should go to 2b'xx and Z should go to 2'bxx until the FSM is reset.

```
module FSM(output reg [1:0] Z, input X, clock, reset);
  reg [1:0]       state, nstate;
  parameter       S0 = 2'00, S1 = 2'b01, S2 = 2'b10;

// implement the state register
always @ (                                                    ) begin



  end
// implement the combinational next state and output logic
always @ (                                                          ) begin









end
endmodule
```

(b) Repeat part (a), but implement the FSM using continuous assignments for the output and flip-flop input equations, and the interface to the D flip-flop shown below for the state register.

module dff(q, d, clk, rst, en);

```
module FSM(output [1:0] Z, input X, clock, reset);
wire [1:0]      state, nstate;

// implement the state register



// implement the flip-flop input equations



// implement the output equations



endmodule
```

(c) Show your state table and how you derived your input and output.

	nstate		Z	
state	X=0	X=1	X=0	X=1
00				
01				
10				
11				

(d) Complete the testbench for the FSM developed in part (b) that exhaustively tests all of the transitions shown in the state diagram. Your testbench should include a monitor statement that prints the current simulation time and all inputs and outputs when any input (except the clock) or output changes. . Your clock should have a period of 10 time units. Comment your test bench so that it is easy to see which state transitions are tested when applying a particular input value. For example, use "// S0 to S1" when testing the transition from S0 to S1.

```
module t_FSM();

// Declare your nets and variables



// Set up your clock



// Set up your monitor statement



// Instantiate your FSM


// Apply the inputs to test the transitions









endmodule
```

Solution:

(a)

```
module FSM(output reg [1:0] Z, input X, clock, reset);
  reg [1:0]      state, nstate;
  parameter      S0 = 2'00, S1 = 2'b01, S2 = 2'b10;

// implement the state register
  always @ (         posedge clock                              ) begin
     if (~reset) state <= S0;
     else state <= nstate;

  end
// implement the combinational next state and output logic
  always @ (         state,  x                                          ) begin
     case (state)
              S0:      if(x) begin  nstate = S1;  z = 2'b01;  end
                       else  begin  nstate = S0;  z = 2'b00;  end
              S1:      if(x) begin  nstate = S0;  z = 2'b01;  end
                       else  begin  nstate = S2;  z = 2'b00;  end
              S2:      if(x) begin  nstate = S1;  z = 2'b11;  end
                       else  begin  nstate = S2;  z = 2'b10;  end
              default:       begin  nstate = 2'bxx;  z = 2'bxx;  end
     endcase


  end
endmodule
```

(b)

```
module FSM(output [1:0] Z, input X, clock, reset);
  wire [1:0]       state, nstate;

// implement the state register
      dff  DFF[1:0]  (state, nstate, clock, ~reset, 1'b1);


// implement the flip-flop input equations
      assign nstate[0] = x & ~state[0];
      assign nstate[1] = ~x & (state[1] | state[0]);

// implement the output equations
      assign z[0] = x;
      assign z[1] = state[1];

endmodule
```

(c)

	nstate		Z	
state	X=0	X=1	X=0	X=1
00	00	01	00	01
01	10	00	00	01
10	10	01	10	11
11	xx	xx	xx	xx

Equations for z[0] and z[1] derived by inspection.

Equations for nstate[0] and nstate[1] are derived by k-maps below:

K-map for nstate[0]

s[1] s[0] x	00	01	11	10
0	0	0	x	0
1	1	0	x	1

nstate[0] = x & ~state[0]

K-map for nstate[1]

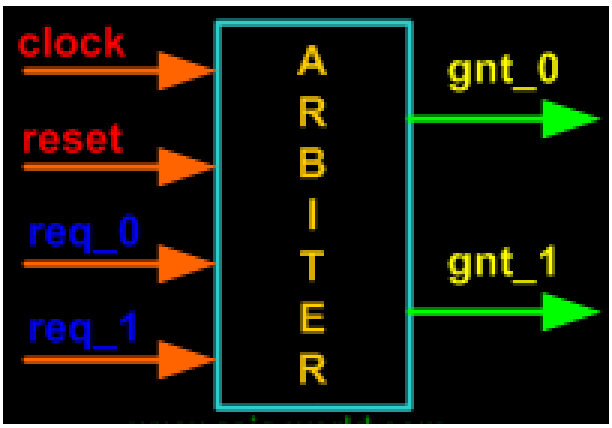
s[1] s[0] x	00	01	11	10
0	0	1	x	1
1	0	0	x	0

nstate[1] = ~x & state[0] | ~x & state[1]

(d)

```
module t_FSM();

// Declare your nets and variables
    wire [1:0] z;
    reg x, clk, rst;


// Set up your clock
    initial begin
            clk = 0;
            forever #5 clk = ~clk;
    end
// Set up your monitor statement

    initial $monitor("%t: %b, %b, %b \n", $time, rst, x, z);


// Instantiate your FSM

    fsm DUT(z, x, clk, rst);

// Apply the inputs to test the transitions
    initial begin
            rst = 0;  x = 0;          //reset to S0
            #10 rst = 1; x=0;         //S0 to S0
            #10 rst = 1; x=1;         //S0 to S1
            #10 rst = 1; x=0;         //S1 to S2
            #10 rst = 1; x=0;         //S2 to S2
            #10 rst = 1; x=1;         //S2 to S1
            #10 rst = 1; x=1;         //S1 to S0
    end




endmodule
```

Problem 3: Implement the following simple Arbiter in Verilog. The Arbiter has got two request inputs and two grant outputs, as shown in the signal diagram below.



- When req_0 is asserted, gnt_0 is asserted
- When req_1 is asserted, gnt_1 is asserted
- When both req_0 and req_1 are asserted then gnt_0 is asserted i.e. higher priority is given to req_0 over req_1.

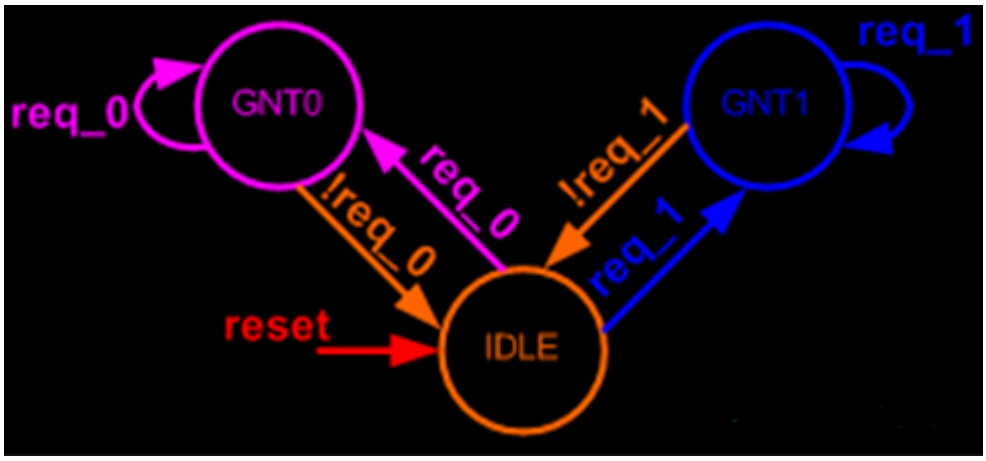
The Arbiter has got following states.

- **IDLE:** In this state the Arbiter waits for the assertion of req_0 or req_1 and drives both gnt_0 and gnt_1 to inactive state (low). This is the default state of the Arbiter; it is entered after the reset and also during fault recovery condition.
- **GNT0:** The Arbiter enters this state when req_0 is asserted, and remains here as long as req_0 is asserted. When req_0 is de-asserted, the Arbiter returns to the IDLE state.
- **GNT1:** The Arbiter enters this state when req_1 is asserted, and remains there as long as req_1 is asserted. When req_1 is de-asserted, the Arbiter returns to the IDLE state.

(a) Draw an FSM for the above Arbiter.
(b) Implement the FSM in (a) in Verilog using two Always blocks.
(c) Implement the FSM in (a) in Verilog using single Always block.

Solution:

(a)



(b)

```
 1 //------------------------------------------------------
 2 // This is FSM demo program using twoalways block
 3 // Design Name : fsm_using_two_always
 4 // File Name   : fsm_using_two_always.v
 5 //------------------------------------------------------
 6 module fsm_using_two_always (
 7 clock      , // clock
 8 reset      , // Active high, syn reset
 9 req_0      , // Request 0
10 req_1      , // Request 1
11 gnt_0      , // Grant 0
12 gnt_1
13 );
14 //-------------Input Ports-----------------------------
15 input   clock,reset,req_0,req_1;
16  //-------------Output Ports----------------------------
17 output  gnt_0,gnt_1;
18 //-------------Input ports Data Type-------------------
19 wire    clock,reset,req_0,req_1;
20 //-------------Output Ports Data Type------------------
21 reg     gnt_0,gnt_1;
22 //-------------Internal Constants----------------------
23 parameter SIZE = 3           ;
24 parameter IDLE  = 3'b001,GNT0 = 3'b010,GNT1 = 3'b100 ;
25 //-------------Internal Variables----------------------
26 reg   [SIZE-1:0]          state        ;// Seq part of the FSM
27 reg   [SIZE-1:0]          next_state   ;// combo part of FSM
28 //----------Code startes Here------------------------
29 always @ (state or req_0 or req_1)
30 begin : FSM_COMBO
31  next_state = 3'b000;
```

```
 case(state)
   IDLE : if (req_0 == 1'b1) begin
                next_state = GNT0;
              end else if (req_1 == 1'b1) begin
                next_state= GNT1;
              end else begin
                next_state = IDLE;
              end
   GNT0 : if (req_0 == 1'b1) begin
                next_state = GNT0;
              end else begin
                next_state = IDLE;
              end
   GNT1 : if (req_1 == 1'b1) begin
                next_state = GNT1;
              end else begin
                next_state = IDLE;
              end
   default : next_state = IDLE;
  endcase
end
//----------Seq Logic-----------------------------
always @ (posedge clock)
begin : FSM_SEQ
  if (reset == 1'b1) begin
    state <=  #1  IDLE;
  end else begin
    state <=  #1  next_state;
  end
end
//----------Output Logic-----------------------------
always @ (posedge clock)
begin : OUTPUT_LOGIC
if (reset == 1'b1) begin
  gnt_0 <=  #1  1'b0;
  gnt_1 <=  #1  1'b0;
end
else begin
  case(state)
    IDLE : begin
                  gnt_0 <=  #1  1'b0;
                  gnt_1 <=  #1  1'b0;
               end
   GNT0 : begin
                   gnt_0 <=  #1  1'b1;
                   gnt_1 <=  #1  1'b0;
                end
   GNT1 : begin
                   gnt_0 <=  #1  1'b0;
                   gnt_1 <=  #1  1'b1;
                end
   default : begin
                    gnt_0 <=  #1  1'b0;
                    gnt_1 <=  #1  1'b0;
                  end
  endcase
end
```

```
89 end // End Of Block OUTPUT_LOGIC
90
91 endmodule // End of Module arbiter
```

(c)

```
 1 //====================================================
 2 // This is FSM demo program using single always
 3 // for both seq and combo logic
 4 // Design Name : fsm_using_single_always
 5 // File Name   : fsm_using_single_always.v
 6 //====================================================
 7 module fsm_using_single_always (
 8 clock      , // clock
 9 reset      , // Active high, syn reset
10 req_0      , // Request 0
11 req_1      , // Request 1
12 gnt_0      , // Grant 0
13 gnt_1
14 );
15 //=============Input Ports=============================
16 input   clock,reset,req_0,req_1;
17  //=============Output Ports============================
18 output  gnt_0,gnt_1;
19 //=============Input ports Data Type====================
20 wire    clock,reset,req_0,req_1;
21 //=============Output Ports Data Type===================
22 reg     gnt_0,gnt_1;
23 //=============Internal Constants=======================
24 parameter SIZE = 3           ;
25 parameter IDLE  = 3'b001,GNT0 = 3'b010,GNT1 = 3'b100 ;
26 //=============Internal Variables=======================
27 reg   [SIZE-1:0]          state        ;// Seq part of the FSM
28 reg   [SIZE-1:0]          next_state   ;// combo part of FSM
29 //==========Code startes Here============================
30 always @ (posedge clock)
31 begin : FSM
32 if (reset == 1'b1) begin
33   state <=  #1  IDLE;
34   gnt_0 <= 0;
35   gnt_1 <= 0;
36 end else
37  case(state)
38    IDLE : if (req_0 == 1'b1) begin
39                state <=  #1  GNT0;
40                gnt_0 <= 1;
41              end else if (req_1 == 1'b1) begin
42                gnt_1 <= 1;
43                state <=  #1  GNT1;
44              end else begin
45                state <=  #1  IDLE;
46              end
47    GNT0 : if (req_0 == 1'b1) begin
48                state <=  #1  GNT0;
49              end else begin
50                gnt_0 <= 0;
```

```
51                 state <=  #1  IDLE;
52               end
53    GNT1 : if (req_1 == 1'b1) begin
54                 state <=  #1  GNT1;
55               end else begin
56                 gnt_1 <= 0;
57                 state <=  #1  IDLE;
58               end
59    default : state <=  #1  IDLE;
60 endcase
61 end
62
63 endmodule // End of Module arbiter
```