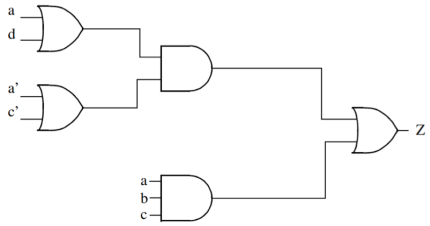
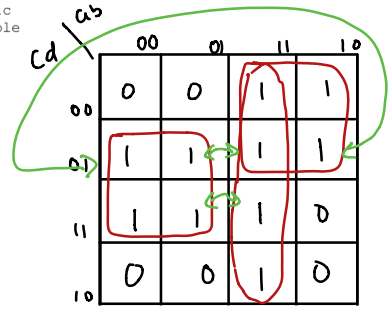
1. 1.1

For the network shown below, find any/all static 1-hazards. For any 1-hazard found specify the conditions which will cause the hazard to appear at the output (i.e. specify the logic values of the variables which are constant and the variable which is changing)



$(a+d)(a'+c') + (abc)$



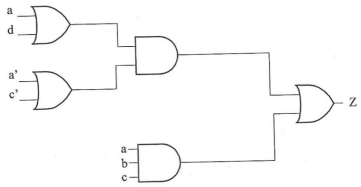
i) 0101 ↔ 1101

ii) 0111 ↔ 1111

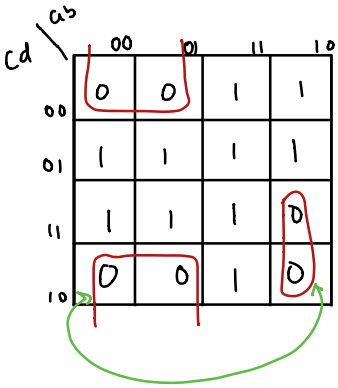
iii) 1001 ↔ 0001

2. 1.2

For the network shown below, find any/all static 0-hazards. For any 0-hazard found specify the conditions which will cause the hazard to appear at the output (i.e. specify the logic values of the variables which are constant and the variable which is changing) [10 points].



$(a+d)(a'+c') + (abc)$



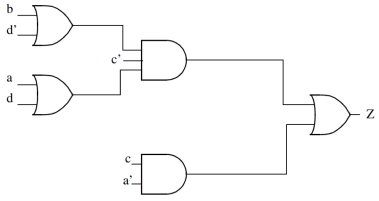
A = 0
B = 0
C = 1
D = 1

↔

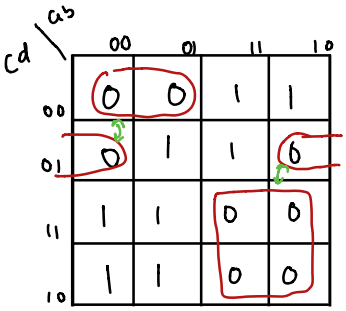
1
0
1
1

4. 1.4

For the network shown below, find all static 0-hazards. For each 0-hazard found specify the conditions which will cause the hazard to appear at the output (i.e. specify the logic values of the variables which are constant and clearly specify the variable that is assumed to be changing). If there are no 0-hazards found, use a K'map to show why this is the case.



$(b+d')(a+d)(c')+(c\,a')$



A = 0
B = 0
C = 0
D = 0

↔

0
0
0
1

A = 1
B = 0
C = 0
D = 1

↔

1
0
1
1

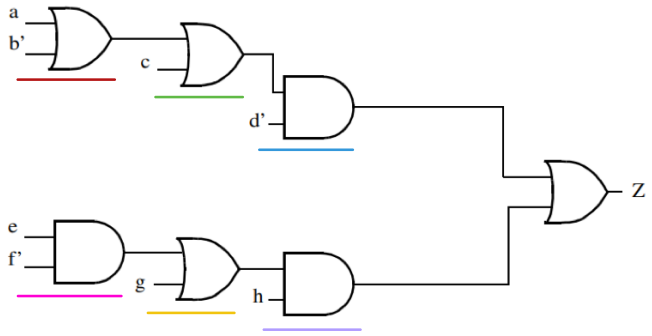
Nolan Anderson

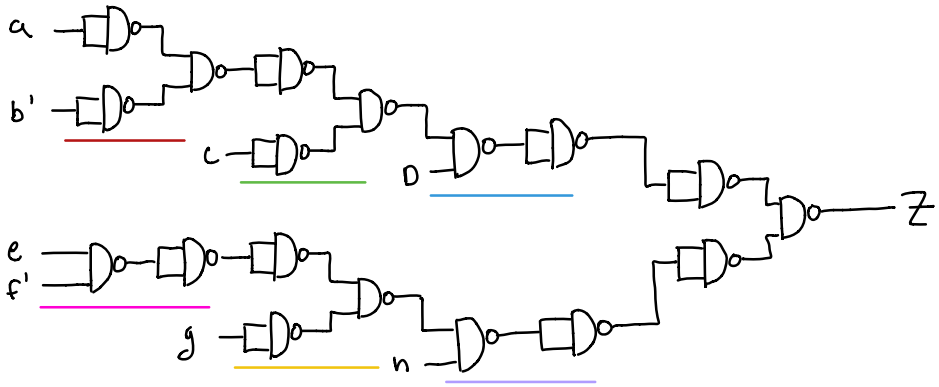
CPE 322

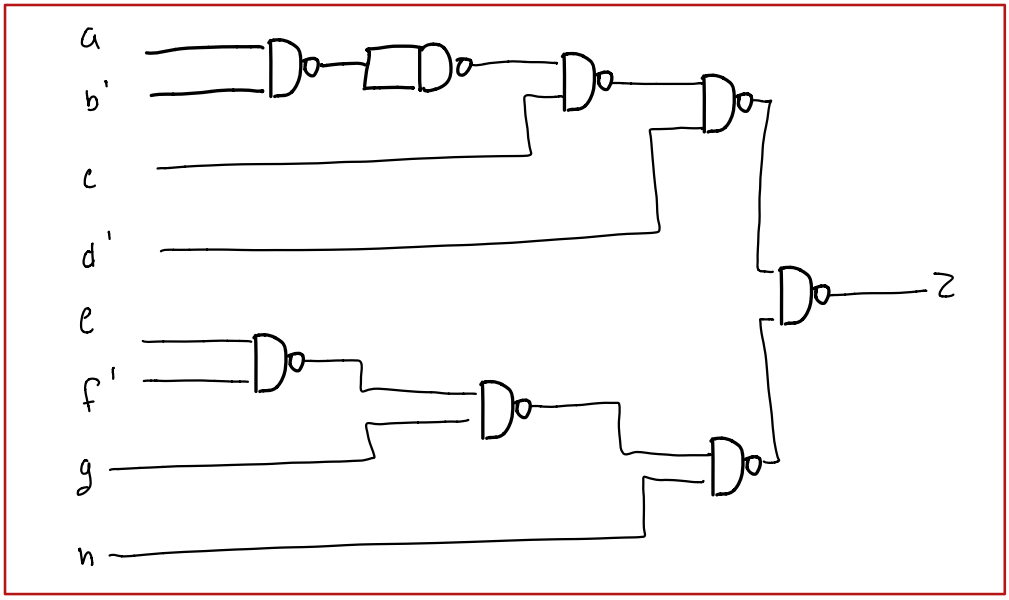
Midterm Corrections

3.1

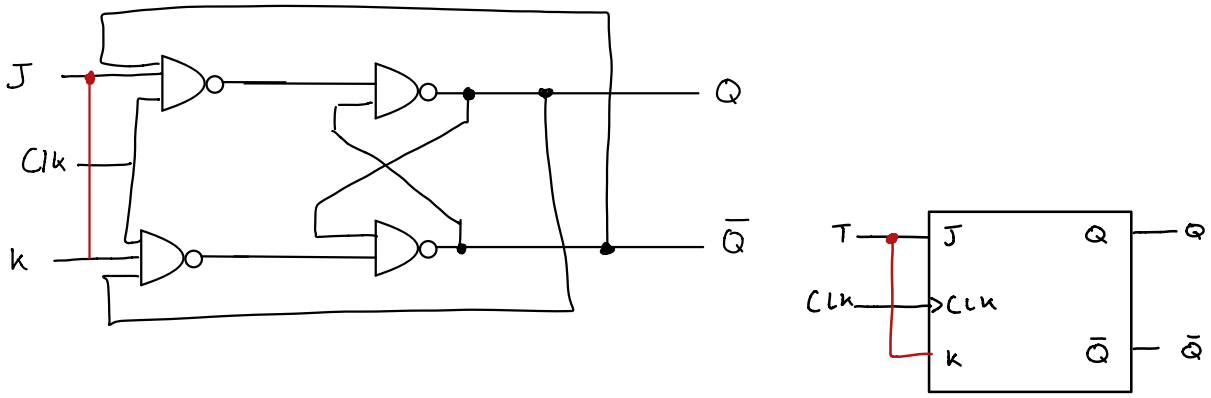
Draw an equivalent multi-level network that is composed entirely of multiple two-input **NAND** gates. Assume that all input variables (a through h) are available in their true and complemented form.





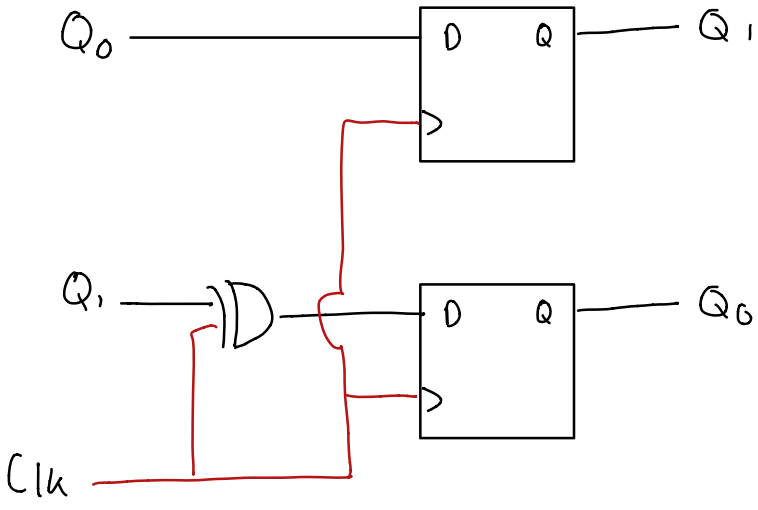


3.2 Use a single rising edge triggered JK Flip-Flop with no additional components (other than connecting wires) to create a rising edge triggered T flip flop (see Fig. 1-13 of the textbook for an example).



3.3

A 2-bit Gray Code accumulator counts 00=>01=>11=>10=>00 and continues repeatedly. Using 2 rising-edge triggered DQ Flip Flops and a single 2-input XOR gate, draw a circuit that continually generates this pattern.



1. Write a short Verilog description of a transparent D Latch with D and G inputs and Q output.

4.1

module DLATCH (**input** D, G, **output** Q)

```
module DLATCH(input D, G, output Q)
input D, G;          // D is data, G is clock
output Q;            // output is Q
always @ (D or G)
    if(G)
        Q <= D;
    end
    else
        Q <= Q;
    end
end
endmodule
```

---

2. Write a short Verilog description of a rising-edge triggered T Flip-Flop (see Fig. 1-13 on pp. 16 of the textbook for definition), with both Q and QN outputs (QN is the inverse of Q). In addition to its normal T and CLK inputs, add an active-low asynchronous SETN input that sets Q high when SETN is low (regardless of clock edge).

4.2

module TFF (**input** T, CLK, SETN, **output** Q, QN)

```
module TFF(input T, CLK, SETN, output Q, QN)
always @ (posedge CLK)
    if(!SETN)
        Q <= 1;              // SETN input that sets Q high when SETN is low.
    else
        if(T)
            QN <= !Q;        // QN gets the inverse of Q
        else
            Q <= Q;          // If Q is low it gets Q
        end
    end
end
endmodule
```

---

3. Write a short Verilog description of a falling-edge triggered JK Flip-Flop, with J and K inputs, and an active-high asynchronous clear input CLR that sets the output Q immediately to 0.

4.3

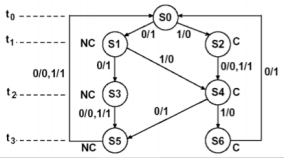
module JKFF (**input** J, K, CLK, CLR, **output** Q)

```
module JKFF(input J, K, CLK, CLR, output Q)
always @ (posedge CLR or negedge CLR)
begin
    if(CLR)
        Q <= 0;
    else
        if(J & K)
            Q <= !Q;
        else if(J & !K)
            Q <= 1;
        else if(!J & K)
            Q <= 0;
        else
            Q <= Q;
        end
    end
end

endmodule
```

Problem 5 Retry:

1. Using two always @() procedures, write a Verilog model of the following state graph shown below.
For state encoding, use 3'd0 for S0, 3'd1 for S1, … 3'd6 for S6.

5.1



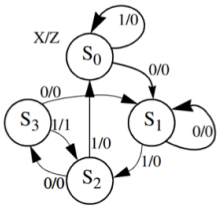

```
module fsm_prob5_1 (input X, CLK, output Z)begin
reg [2:0] state;
reg [2:0] next_state;
parameter S0 = 3'd0, S1 = 3'd1, S2 = 3'd2, S3 = 3'd3, S4 = 3'd4, S5 = 3'd5;
always @(X, state)
begin
    case(state)
        S0:
            if(X)
                Z = 2'b'01;
            else
                Z = 2'b'00;
            end
        S1:
            if(X)
                Z = 2'b'00;
            else
                Z = 2'b'01;
            end
        S2:
            if(X)
                Z = 2'b'01;
            else
                Z = 2'b'00;
            end
        S3:
            if(X)
                Z = 2'b'01;
            else
                Z = 2'b'00;
            end
        S4:
            if(X)
                Z = 2'b'00;
            else
                Z = 2'b'01;
            end
        S5:
            if(X)
                Z = 2'b'01;
            else
                Z = 2'b'00;
            end
        S6:
            if(X == 0)
                Z = 2'b'01;
            end
        default:
            next_state == S0;
    endcase
end
```

```
always @(posedge CLK)
begin
    case(state)
        S0:
            if(X)
                next_state = S2;
            else
                next_state = S1;
            end
        S1:
            if(X)
                next_state = S4;
            else
                next_state = S3;
            end
        S2:
            if(X)
                next_state = S4;
            else
                next_state = S4;
            end
        S3:
            if(X)
                next_state == S5;
            else
                next_state == S5;
            end
        S4:
            if(X)
                next_state == S6;
            else
                next_state == S5;
            end
        S5:
            if(X)
                next_state == S0;
            else
                next_state == S0;
            end
        S6:
            if(X == 0)
                next_state == S0;
            end
        default: next_state == S0;
    endcase
end

endmodule
```

5.2

```
module fsm_prob5_2(input X, CLK, output Z)begin
reg [1:0] state;
reg [1:0] next_state;
parameter S0 = 2'd0, S1 = 2'd1, S2 = 2'd2, S3 = 2'd3;

always @(X, state)
begin
    case(state)
        S0:
            if(X)
                Z = 2'b'00;
            else
                Z = 2'b'00;
            end
        S1:
            if(X)
                Z = 2'b'00;
            else
                Z = 2'b'00;
            end
        S2:
            if(X)
                Z = 2'b'00;
            else
                Z = 2'b'00;
            end
        S3:
            if(X)
                Z = 2'b'01;
            else
                Z = 2'b'00;
            end
    endcase
end

always @(posedgeCLK)
begin
    case(state)
        S0:
            if(X)
                next_state = S0;
            else
                next_state = S1;
            end
        S1:
            if(X)
                next_state = S2;
            else
                next_state = S1;
            end
        S2:
            if(X)
                next_state = S0;
            else
                next_state = S3;
            end
        S3:
            if(X)
                next_state == S2;
            else
                next_state == S1;
            end
    endcase
end

endmodule
```

2. Using two always @() procedures, write a Verilog model of the following state graph shown below.
For state encoding, use 2'd0 for S0, 2'd1 for S1, 2'd2 for S2, and 2'd3 for S3.



3. Create a valid Verilog model for a 3-to-8 decoder, with a 3-bit input SEL and an 8-bit output ONEHOT. The 3-bit input SEL enables a 1 on the output bit in the ONEHOT output vector at bit position SEL, with all other bits set to 0.

5.3

```
module three2eight_dec (input [2:0] SEL, output[7:0]ONEHOT)begin
    always @(SEL)
        case (SEL)
            3'b000 : ONEHOT = 8'b00000001;
            3'b001 : ONEHOT = 8'b00000010;
            3'b010 : ONEHOT = 8'b00000100;
            3'b011 : ONEHOT = 8'b00001000;
            3'b100 : ONEHOT = 8'b00010000;
            3'b101 : ONEHOT = 8'b00100000;
            3'b110 : ONEHOT = 8'b01000000;
            3'b111 : ONEHOT = 8'b10000000;
            default : ONEHOT = 8'b00000000;
        endcase
    end
endmodule

```

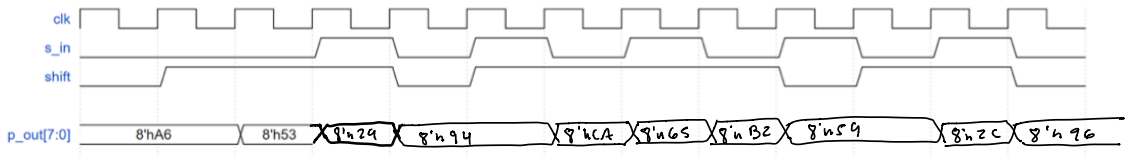
4. Create a valid Verilog 5-bit priority encoder, where the highest bit position on the 5-bit input VEC[4:0] that contains a 1 is identified as the 3-bit output SEL[2:0]. For example, under the input condition VEC[4:0] = 5'b01011, the SEL output would be 3'd3 (since bit position 3 is the highest one containing a 1).

```
module prio_enc (input [4:0] VEC, output[2:0]SEL)begin
assign SEL =
            (input[4] == 1'b1) ? 3'b100;
            (input[3] == 1'b1) ? 3'b011:
            (input[2] == 1'b1) ? 3'b010:
            (input[1] == 1'b1) ? 3'b001:
            (input[0] == 1'b1) ? 3'b000: 3'bxxx;
endmodule
```

INSTRUCTIONS: In this problem set, all state machines are intended to behave as Moore state machines; the p_out[7:0] and accum[7:0] outputs must change in value one clock cycle after the inputs change, as shown in the first 3 provided values in the timing diagrams.

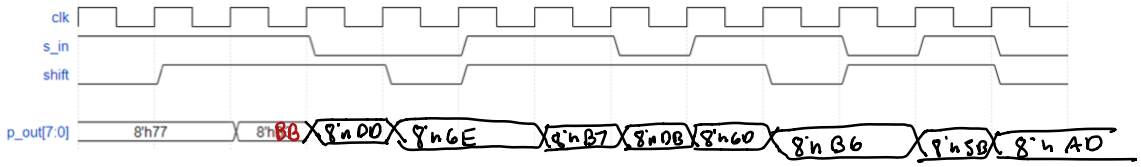
6.1

1. Complete the following timing diagram of a serial-in/parallel-out shift register, where serial data shifts in bit-by-bit from the **s_in** input into the most-significant bit side (bit 7; newest/most-recent) towards the least-significant bit side (bit 0; oldest/least-recent). Data only shifts when the **shift** input is high; when shift is low **p_out**[7:0] does not change in value. You can either complete the **p_out**[7:0] signal values in hexadecimal or binary.



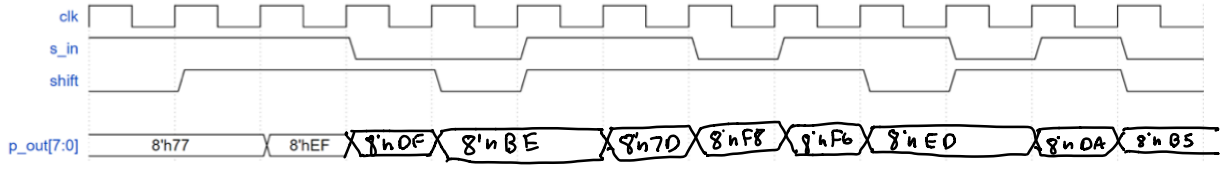
6.2

2. Complete the following timing diagram of a serial-in/parallel-out shift register, where serial data shifts in bit-by-bit from the **s_in** input into the most-significant bit side (bit 7; newest/most-recent) towards the least-significant bit side (bit 0; oldest/least-recent). Data only shifts when the **shift** input is high; when shift is low **p_out**[7:0] does not change in value. You can either complete the **p_out**[7:0] signal values in hexadecimal or binary.



6.3

3. **NOTE: THIS PROBLEM STATEMENT DIFFERS FROM #1 & #2…** Complete the following timing diagram of a serial-in/parallel-out shift register, where serial data shifts in bit-by-bit from the **s_in** input into the ***least-significant*** bit side (bit 0; newest/most-recent) towards the ***most-significant*** bit side (bit 0; oldest/least-recent). Data only shifts when the **shift** input is high; when shift is low **p_out**[7:0] does not change in value. You can either complete the **p_out**[7:0] signal values in hexadecimal or binary.



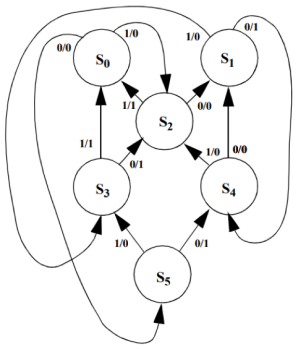
INSTRUCTIOS: FOR THE FOLLOWING 4 STATE MACHINES, DO THE FOLLOWING:

1) **Create an equivalent Algorithmic State Chart representation, using rectangular boxes to show states, diamonds to show input/decision blocks, and ovals to represent conditional outputs. See the Module/Slides for Algorithmic State Machine Representation for examples.**
2) **Write a behavioral Verilog HDL model that implements the state transition graph. In your model include a synchronous active high reset input signal that will always place the design in state S0 on the active edge of the next clock pulse regardless of the current state of the network.**

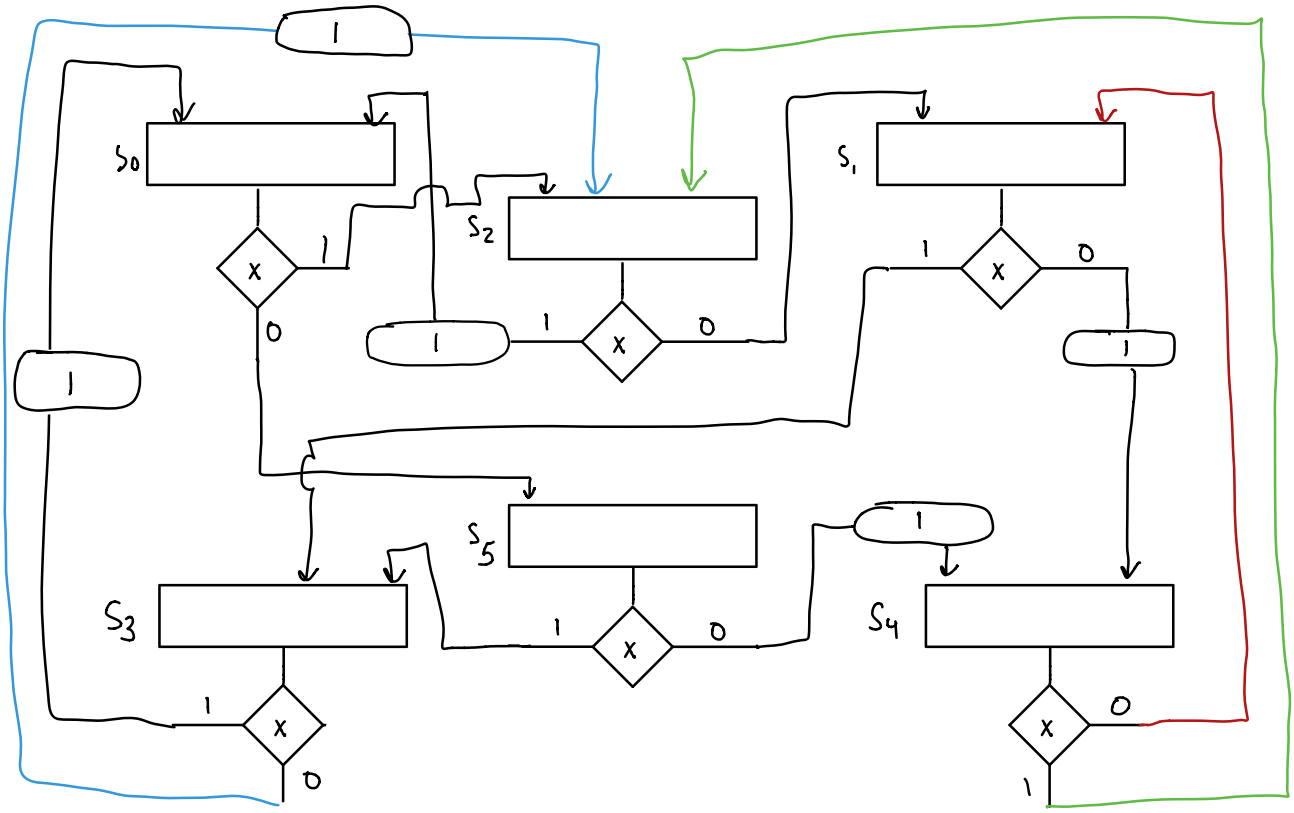
1)

Present State	Next State X=0	Next State X=1	Output X=0	Output X=1
S_0	S_5 S_1	S_2	0	0
S_1	S_4	S_3	1	0
S_2	S_1	S_0	0	1
S_3	S_2	S_0	1	1
S_4	S_1	S_2	0	0
S_5	S_4	S_3	1	0

A: 7A

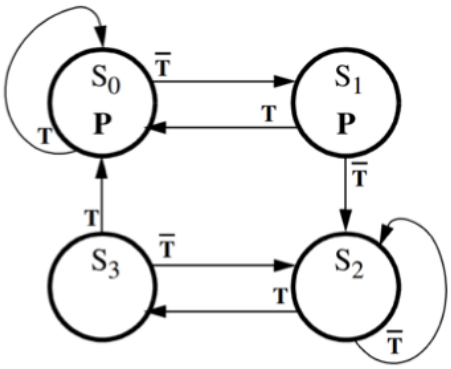



```
module SevenA(input data_in, clk, reset, output data_out)
input clk, data_in, reset;
output reg [1:0] data_out;
reg [1:0] state;
parameter S0 = 0, S1 = 1, S2 = 2, S3 = 3, S4 = 4, S5 = 5;

always @(posedge clk or posedge reset)
    begin
        if(reset)
            state <= S0;
        else
            case(state)
                S0:
                    if(data_in)
                        state <= S2;
                    else
                        state <= S5;
                    end
                S1:
                    if(data_in)
                        state <= S3;
                    else
                        state <= S4;
                    end
                S2:
                    if(data_in)
                        state <= S0;
                    else
                        state <= S1;
                    end
                S3:
                    if(data_in)
                        state <= S0;
                    else
                        state <= S2;
                    end
                S4:
                    if(data_in)
                        state <= S2;
                    else
                        state <= S1;
                    end
                S5:
                    if(data_in)
                        state <= S3;
                    else
                        state <= S4;
                    end
                default: state <= S0;
            endcase
        end
    end

always @(state)
    begin
        case(state)
            S0:
                if(data_in)
                    data_out = 2b'00;
                else
                    data_out = 2b'00;
                end
            S1:
                if(data_in)
                    data_out = 2b'00;
                else
                    data_out = 2b'01;
                end
            S2:
                if(data_in)
                    data_out = 2b'01;
                else
                    data_out = 2b'00;
                end
            S3:
                if(data_in)
                    data_out = 2b'01;
                else
                    data_out = 2b'01;
                end
            S4:
                if(data_in)
                    data_out = 2b'00;
                else
                    data_out = 2b'00;
                end
            S5:
                if(data_in)
                    data_out = 2b'00;
                else
                    data_out = 2b'01;
                end
        endcase
    end
endmodule
```

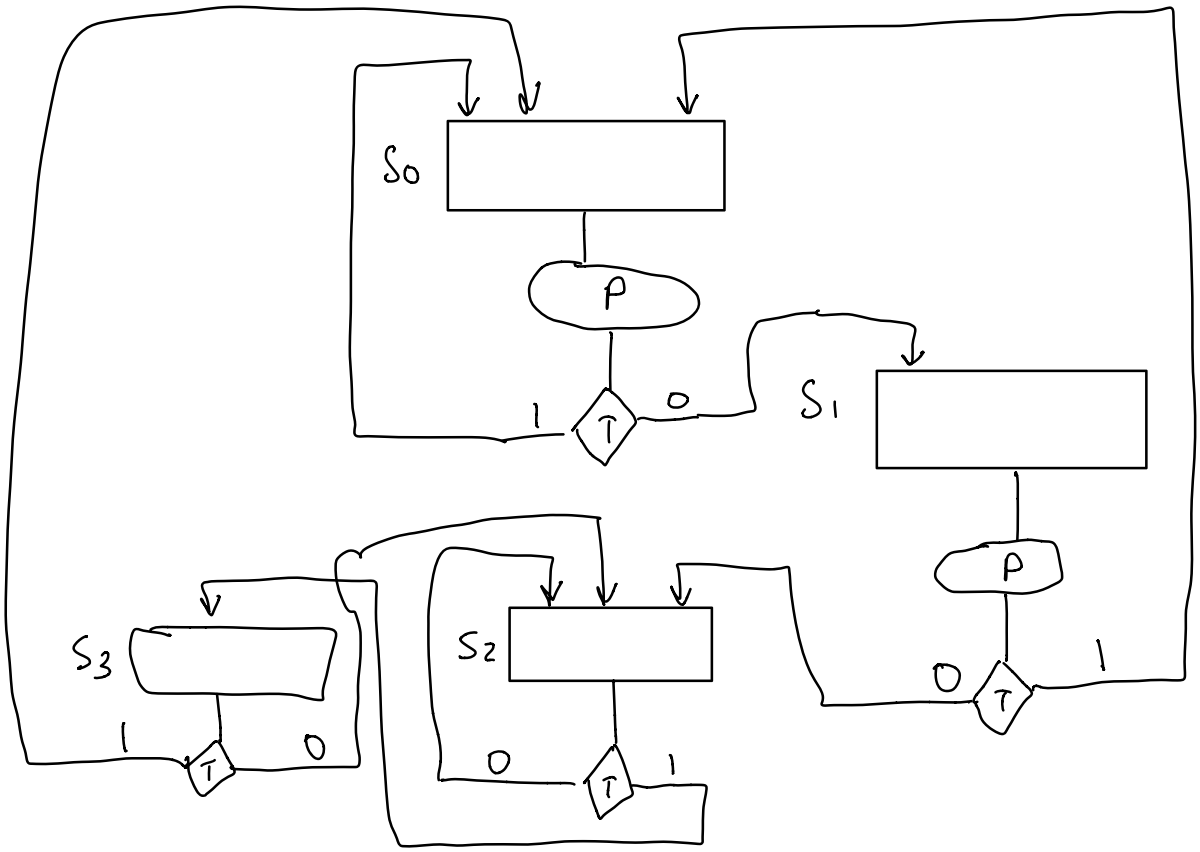


7B

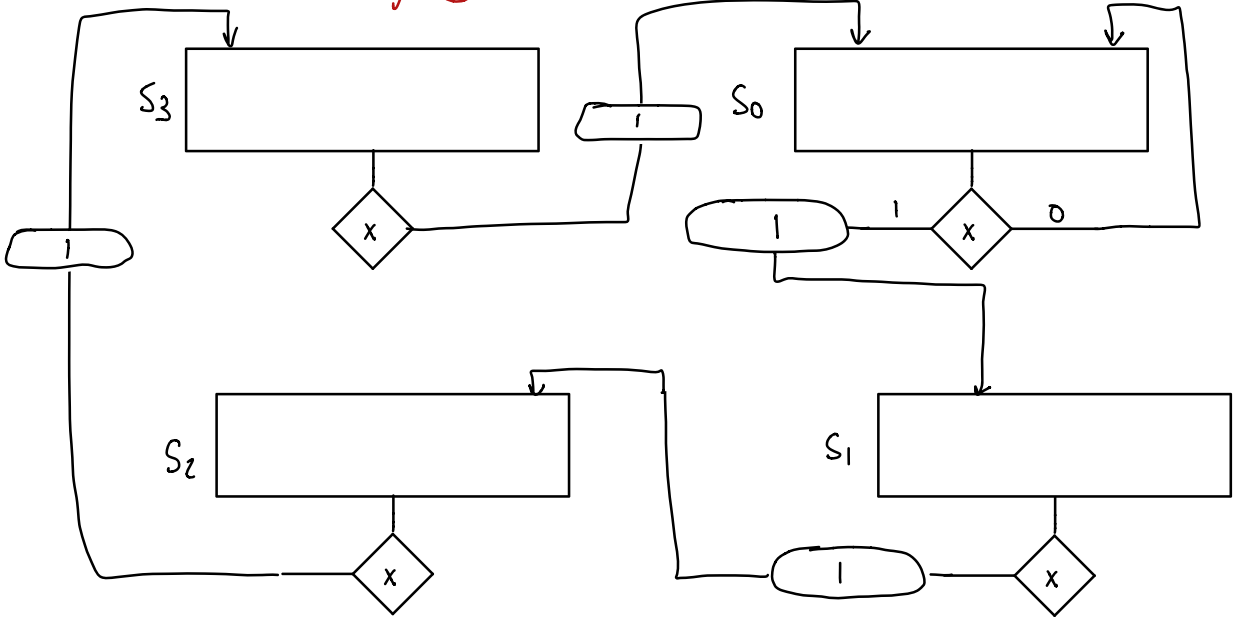
B:



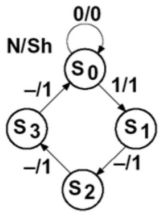

```
module SevenB(input T, CLK, output Z)begin
reg [1:0] state;
parameter S0 = 2'd0, S1 = 2'd1, S2 = 2'd2, S3 = 2'd3;

always @(posedge CLK)
begin
    case(state)
        S0:
            if(T) begin
                state = S0;
                Z = 2'b01;
            else
                state = S1;
                Z = 2b'01;
            end
        S1:
            if(T) begin
                state = S0;
                Z = 2'b01;
            else
                state = S2;
                Z = 2b'01;
            end
        S2:
            if(T) begin
                state = S3;
                Z = 2'b00;
            else
                state = S2;
                Z = 2b'00;
            end
        S3:
            if(T) begin
                state = S0;
                Z = 2'b00;
            else
                state = S2;
                Z = 2b'00;
            end
    endcase
end
```



7C



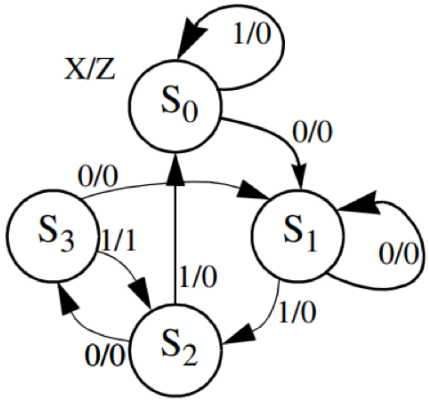
C:



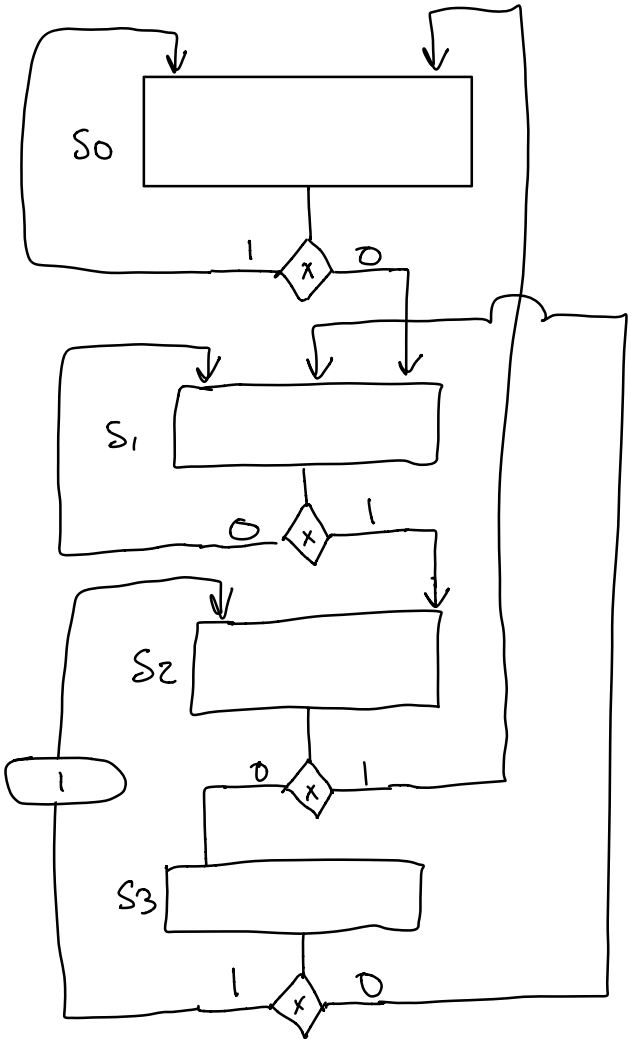

```
module SevenC(input data_in, clk, reset, output data_out)
input clk, data_in, reset;
output reg [1:0] data_out;
reg [1:0] state;
parameter S0 = 0, S1 = 1, S2 = 2, S3 = 3;

always @(posedge clk or posedge reset)
    begin
        if(reset)
            state <= S0;
        else
            case(state)
                S0:
                    if(data_in)
                        state <= S1;
                    else
                        state <= S0;
                    end
                S1:
                    if(data_in || !data_in)
                        state <= S2;
                end
                S2:
                    if(data_in || !data_in)
                        state <= S3;
                end
                S3:
                    if(data_in || !data_in)
                        state <= S0;
                end
                default: state <= S0;
            endcase
        end
    end
always @(state)
    begin
        case(state)
            S0:
                if(data_in)
                    data_out = 2b'01;
                else
                    data_out = 2b'00;
                end
            S1:
                if(data_in || !data_in)
                    data_out = 2b'01;
            end
            S2:
                if(data_in || !data_in)
                    data_out = 2b'01;
            end
            S3:
                if(data_in || !data_in)
                    data_out = 2b'01;
            end
        endcase
    end
endmodule
```

7D

D:

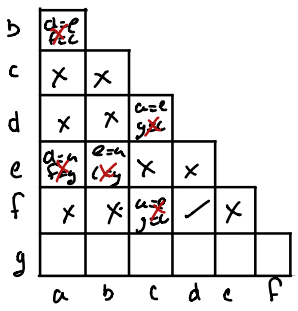



```
module SevenD(input X, CLK, output Z)begin
reg [1:0] state;
reg [1:0] next_state;
parameter S0 = 2'd0, S1 = 2'd1, S2 = 2'd2, S3 = 2'd3;

always @(X, state)
begin
    case(state)
        S0:
            if(X)
                Z = 2b'00;
            else
                Z = 2b'00;
            end
        S1:
            if(X)
                Z = 2b'00;
            else
                Z = 2b'00;
            end
        S2:
            if(X)
                Z = 2b'00;
            else
                Z = 2b'00;
            end
        S3:
            if(X)
                Z = 2b'01;
            else
                Z = 2b'00;
            end
    endcase
end

always @(posedgeCLK)
begin
    case(state)
        S0:
            if(X)
                next_state = S0;
            else
                next_state = S1;
            end
        S1:
            if(X)
                next_state = S2;
            else
                next_state = S1;
            end
        S2:
            if(X)
                next_state = S0;
            else
                next_state = S3;
            end
        S3:
            if(X)
                next_state == S2;
            else
                next_state == S1;
            end
    endcase
end

endmodule
```



8.1

present state	next state X=0	1	present output X=0	1
a	d	f	1	0
b	e	c	1	0
c	a	g	0	1
d	e	c	0	1
e	a	g	1	1
f	e	c	0	1

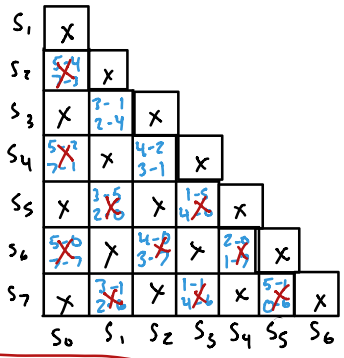


Present	Next 0	1	Present output
a	d	d	1 0
b	e	c	1 0
c	a	a	0 1
d	e	c	0 1
e	a	a	1 1

8.2

2.

present state		next state X=0	1	present output
a	S_0	S_5	S_7	0
b	S_1	S_3	S_2	1
c	S_2	S_4	S_3	0
d	S_3	S_1	S_4	1
e	S_4	S_2	S_1	0
f	S_5	S_5	S_0	1
g	S_6	S_0	S_7	0
h	S_7	S_1	S_6	1



Present State	Next State		Present output
S_0	S_5	S_7	0
S_1	S_1	S_2	1
S_2	S_2	S_1	0
S_5	S_5	S_0	1
S_6	S_0	S_7	0
S_7	S_1	S_6	1

$S_3 \rightarrow S_1$

$S_4 \rightarrow S_2$

10.2

2. Develop a Verilog model for a 6-to-1 multiplexer where the general inputs represent a 6-bit bus and the output is a single signal. When S[2:0] is 6, choose I[2] as the output, and when S[2:0] is 7, choose I[3] as the output. If using a case statement, please use a default clause; if using an if/else statement, assign O in all clauses including a final "else" clause to avoid creating a latch. Label the inputs I[5:0], S[2:0], and the output O.

```
1  module MUX6TO1(input [5:0] I, [2:0] S, output O)
2  reg [5:0] O;
3  always @ (inp or S)
4      case(S)
5        3'b000 : O = inp[0];
6        3'b001 : O = inp[1];
7        3'b010 : O = inp[2];
8        3'b011 : O = inp[3];
9        3'b100 : O = inp[4];
10       3'b101 : O = inp[5];
11       default : O = 8'bx;
12     endcase
13 endmodule
```

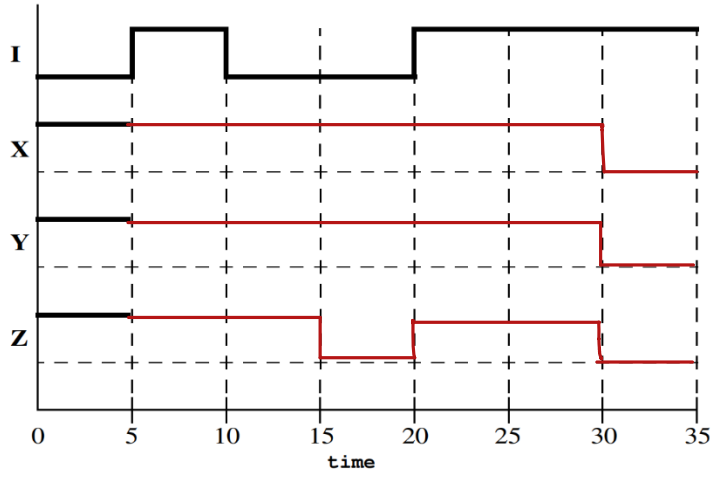
Problem 11 Retry (first part only):

11.1

1.

Complete the Timing Diagram for the simulation of the following Verilog Model where input I is driven in the manner that is shown on the diagram.

```
module v_model(input I, output X,Y, output reg Z);
   assign #10 X = ~I;
   not #(10) G1 (Y,I);
   always @(I) Z <= #10 ~I;
endmodule;
```



2.

11.2

Complete the following timing diagram for the output signal, **Z** and the internal input signal, **state**. Assume that a basic functional RTL Simulation is to be performed.

```
module fsm_network (input clk, X, Reset, output reg Z);
    reg state, next_state;

    always @ (state, X)
        case (state)
            0 : if (X) begin Z=1;  next_state=1; end
                else  begin Z=0; next_state=0; end

            1 : if (X) begin Z=0;  next_state=1; end
                else  begin Z=1; next_state=1; end
        endcase

    always @ (posedge clk)
        if (Reset) state=0;
        else state=next_state;

endmodule
```

State = next state