ENGR 210 / CSCI B441
“Digital Design”

Final Review

Andrew Lukefahr

Announcements

- Spi due Friday!

- Exam next Friday

12:40 - 2:40

In Same room

General Topics

- Truth Tables / Boolean Logic / Logic Gates
- Combinational Logic
- ALU
- RS, D Latch, D Flip Flop Circuits
- State Machines
- SPI
- FPGAs

Testbenches

Textbook Mapping

Chapt / Section

- Section 1.2
- Section 2.2 (skip 2.2.2)
- Section 2.3 } unsigned & signed arith
- Section 2.4 }
- Section 3.1 } comb
- Section 3.2 }
- Section 4.1 } seq logic
- Section 4.2 }
- Section 4.3 }

SystemVerilog on Exam (?)

- Oh, yes!
- Big part of the exam is writing code.

- Don’t care about minor syntax errors
  - Missing commas / semi-colons (`;`)

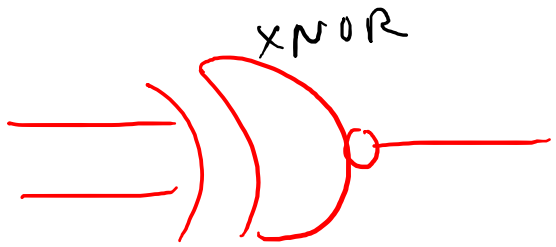
- Minor point deductions for:
  - Blocking vs. Non-blocking operators
  - Missing delays (`#`)

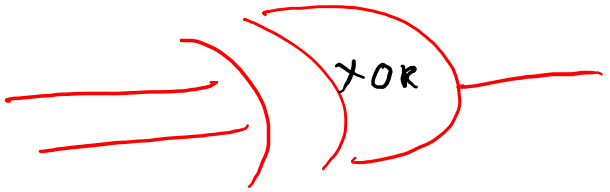
- Major point deductions for logical errors!

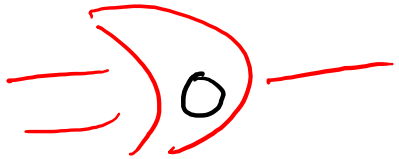
A “Cheat” Sheet is Allowed

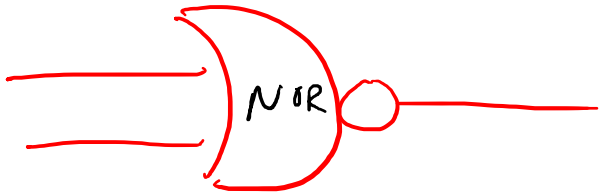
- 2-sided
- 8.5”x11” paper
- Handwritten (not photocopied)

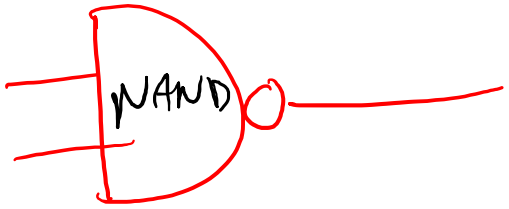
Logic Gates



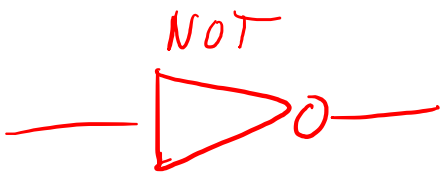


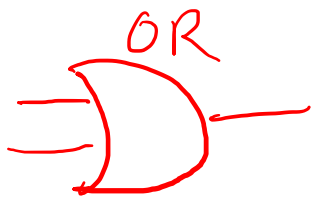


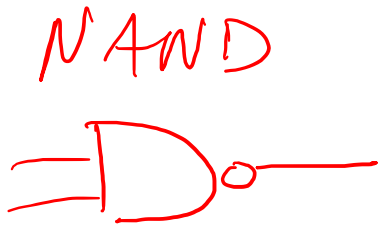


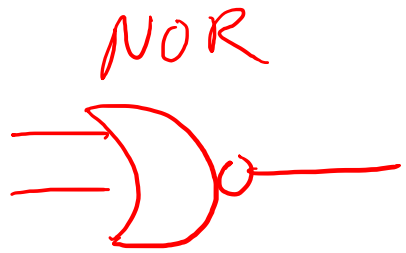


Logic Gates







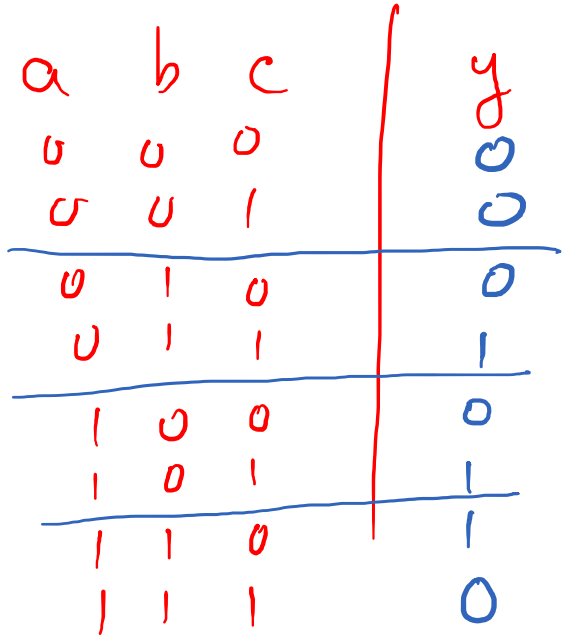


Boolean Equation -> Truth Table

Write the truth table for the following Boolean equation:

$$y = (ab + ca) \oplus bc$$

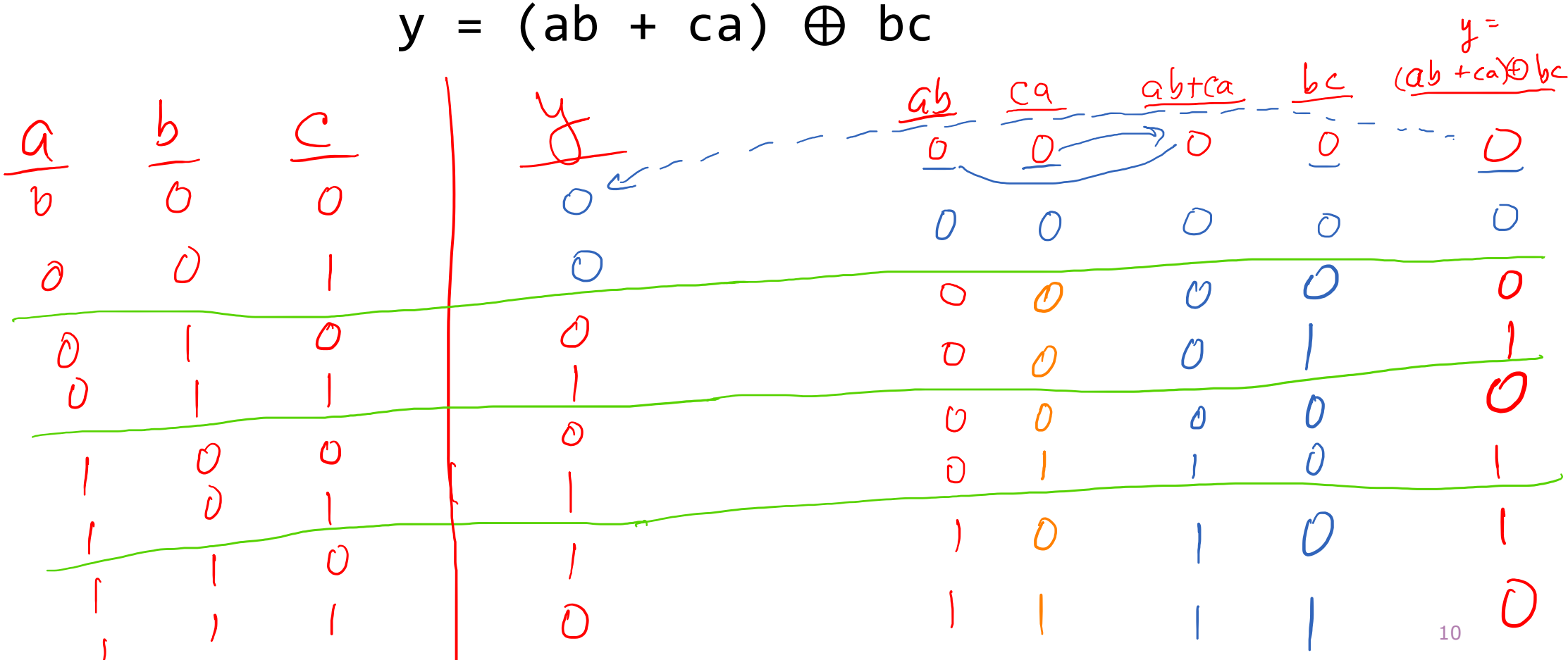
```
y = ( (a & b) | (c & a)) ^ (b & c);
```



Boolean Equation -> Truth Table

Write the truth table for the following Boolean equation:

$$y = (ab + ca) \oplus bc$$



Truth Table -> Boolean Equation

A	B	C	D	X	Y
0	0	0	0	0	0
0	0	0	1	0	0
0	0	1	0	0	0
0	0	1	1	1	0
0	1	0	0	0	1
0	1	0	1	0	0
0	1	1	0	0	0
0	1	1	1	1	0
1	0	0	0	0	0
1	0	0	1	0	1
1	0	1	0	0	0
1	0	1	1	1	0
1	1	0	0	1	0
1	1	0	1	1	0
1	1	1	0	1	1
1	1	1	1	1	0

y_4

y_9

y_{14}

$y = y_4 \mid y_9 \mid y_{14}$

$y = (\overline{A} \,\&\, B \,\&\, \overline{C} \,\&\, \overline{D} \mid$

$A \,\&\, \overline{B} \,\&\, \overline{C} \,\&\, D \mid$

$A \,\&\, B \,\&\, C \,\&\, \overline{D})$

Truth Table -> Boolean Equation

* not need to be minimized

A	B	C	D	X	Y
0	0	0	0	0	0
0	0	0	1	0	0
0	0	1	0	0	0
0	0	1	1	1	0
0	1	0	0	0	1
0	1	0	1	0	0
0	1	1	0	0	0
0	1	1	1	1	0
1	0	0	0	0	0
1	0	0	1	0	1
1	0	1	0	0	0
1	0	1	1	1	0
1	1	0	0	1	0
1	1	0	1	1	0
1	1	1	0	1	1
1	1	1	1	1	0

$X = X_1 \mid X_2 \mid X_3 \mid X_4 \mid X_5 \mid X_6 \mid X_7$

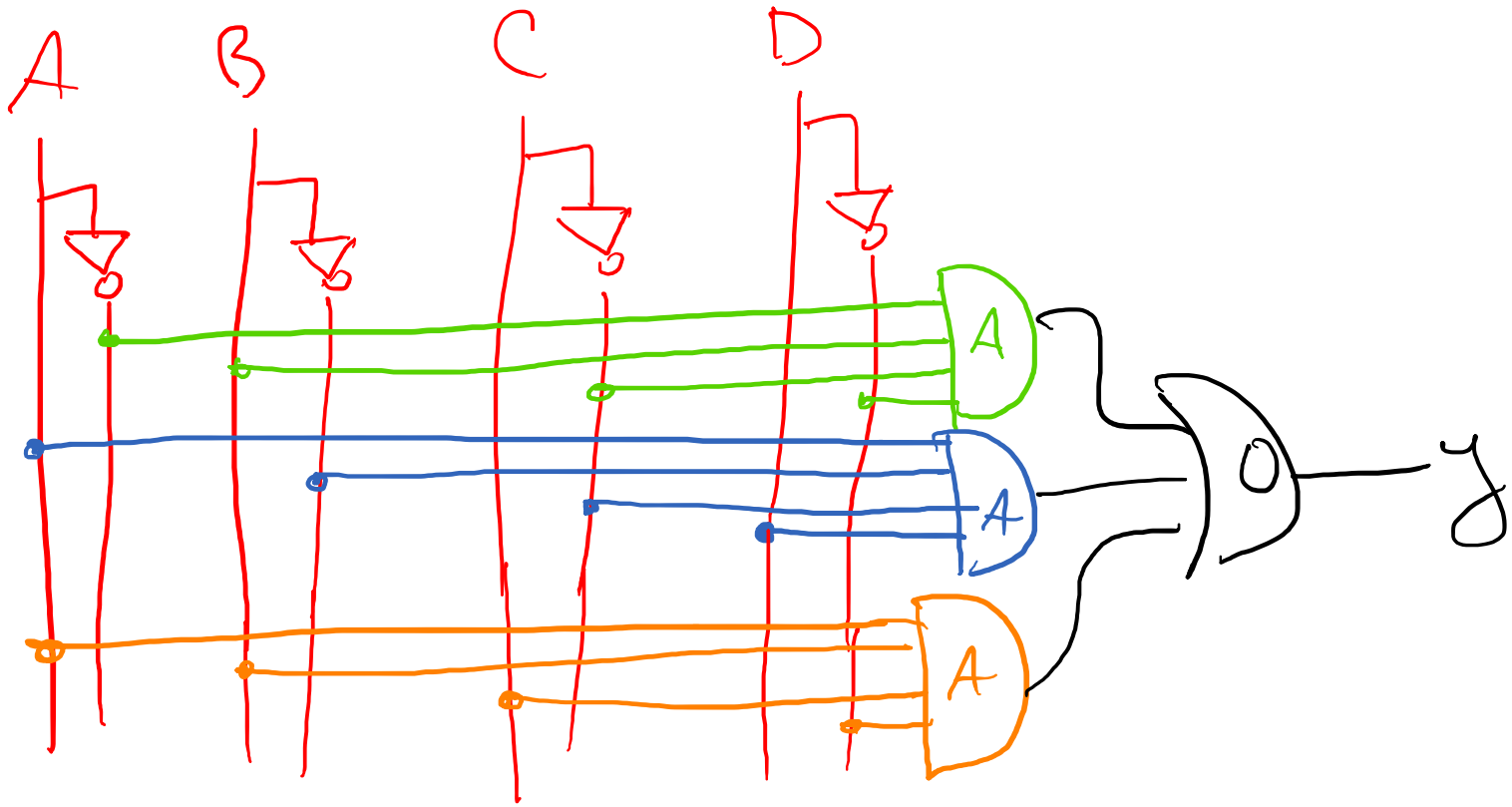
$X = \bar{a}\bar{b}cd \mid \bar{a}bcd \mid a\bar{b}cd \mid$

$ab\bar{c}\bar{d} \mid ab\bar{c}d \mid abc\bar{d} \mid abcd$

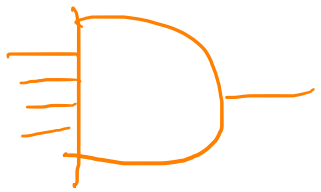
$y = \bar{a}b\bar{c}\bar{d} \mid a\bar{b}\bar{c}d \mid$

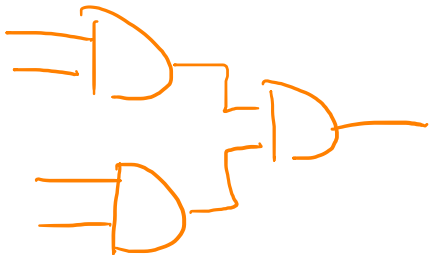
$abc\bar{d}$

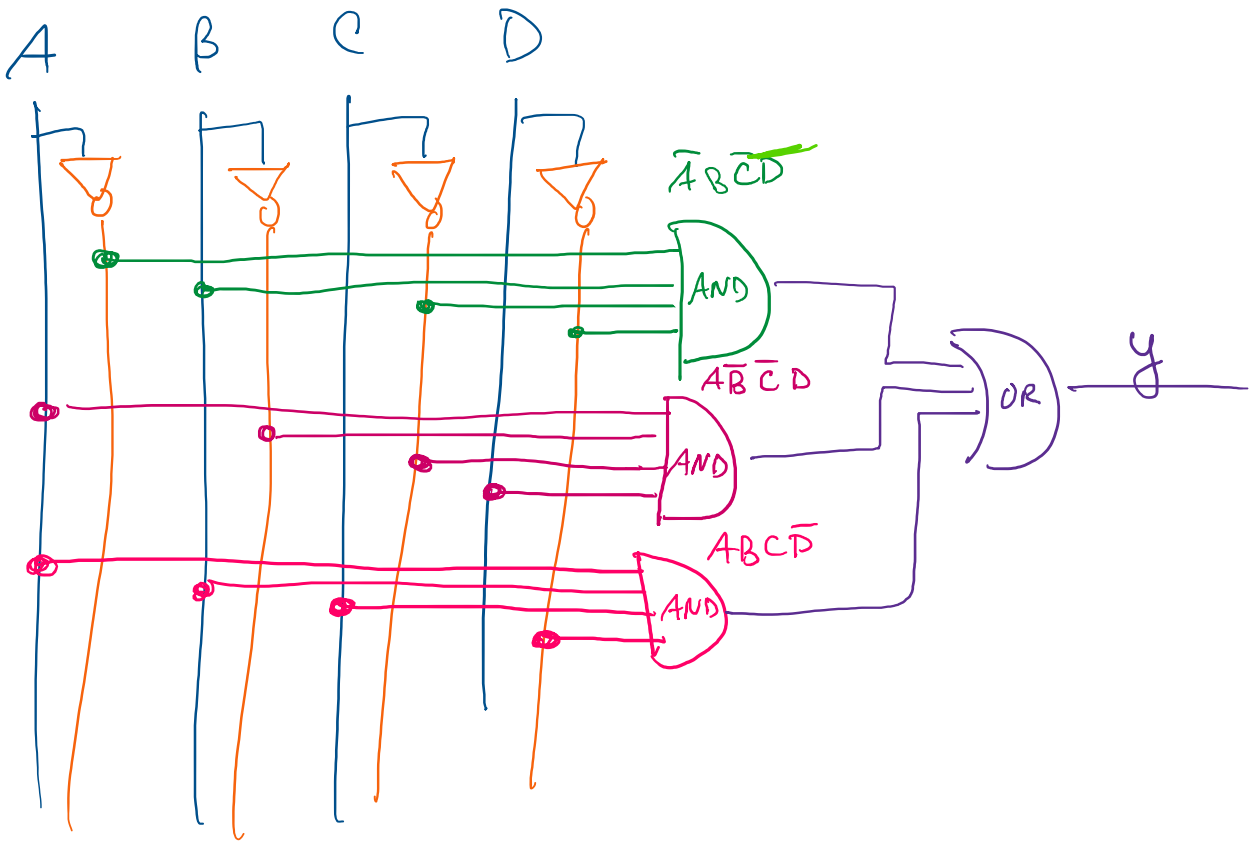
$$y = \bar{A}B\bar{C}\bar{D} + A\bar{B}\bar{C}D + ABC\bar{D}$$



$y = \bar{A}B\bar{C}\bar{D} + A\bar{B}\bar{C}D + ABC\bar{D}$

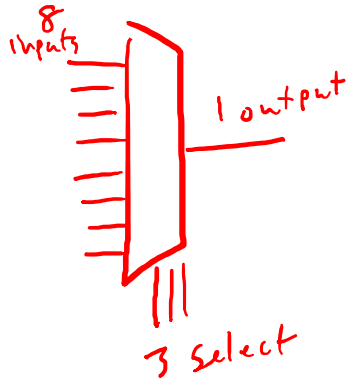




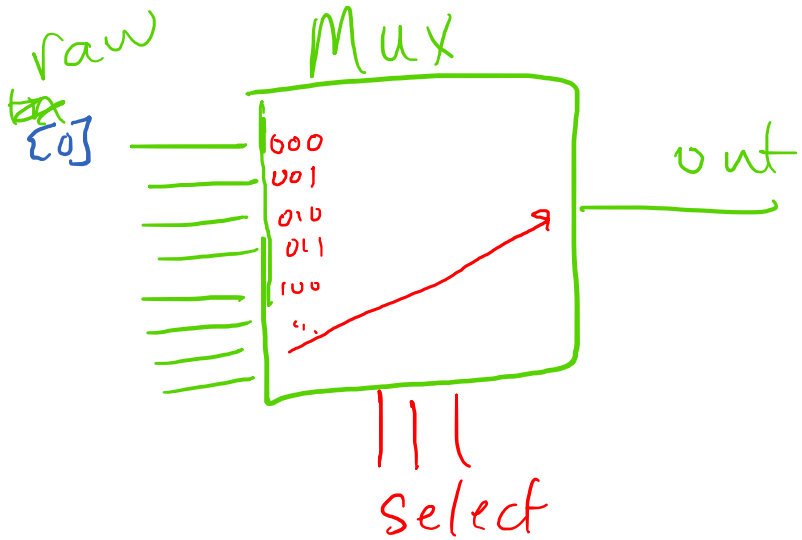


Multiplexer in Verilog

```
module mux (
     input [7:0] raw,
     input [2:0] select,
     output logic out
);

always_comb begin
    case (select)
        3'h0: out = raw[0];
        3'h1: out = raw[1];
        3'h2: out = raw[2];
        ...
    endcase
end

endmodule
```



Multiplexer in Verilog




```
module mux (
     input [7:0] raw,
     input [2:0] select,
     output out
);
```

or "output reg out"

reg outR;

always_comb begin

case (select)

3'h0: outR = raw[0];
3'h1: outR = raw[1];
3'h2: outR = raw[2];
3'h3: ... [3]
3'h4: ... raw[4]
3'h5: [5]
3'h6: [6]
3'h7: [7]

endcase

endmodule

assign out = outR;

Multiplexer in Verilog

```
module mux (
     input [7:0] raw,
     input [2:0] select,
     output logic out
);


always_comb begin
          out = 0; // default
          case (select)
                    3'h0: out = raw[0];
                    3'h1: out = raw[1];
                    3'h2: out = raw[2];
                    3'h3: out = raw[3];
                    3'h4: out = raw[4];
                    3'h5: out = raw[5];
                    3'h6: out = raw[6];
                    3'h7: out = raw[7];
          endcase
end
endmodule
```

Multiplexer in Verilog

```
module mux (
    input [7:0] raw,
    input [2:0] select,
    output out
);

assign out = raw[ select ]; // :)

endmodule
```

MUX Testbench

```
module mux (
    input [7:0] raw,
    input [2:0] select,
    output out
);
```

```
module mux_tb();
  logic [7:0] raw;
  logic [2:0] select;
  logic out; // wire
mux DUT( .raw(raw), .select(select), .out(out));
integer i, j
initial begin
   for(i=0; i<256; i++) begin
        raw = i;
        for(j=0; j<8; j++) begin
            sel = j;
            #1
        end assert(out = raw[j]) else $fatal(1, "bad mux");
   end
end module
```

MUX Testbench

```
module testbench;
logic [7:0] raw;
logic [2:0] sel;
logic out

mux m0 (.raw(raw), .select(sel), .out(out));

integer i, j;
initial  begin
    for (i = 0; i < 256; i++) begin
        for (j = 0; j < 8; j++) begin
            raw = i; sel = j;
            #1
            assert(out == raw[sel]) else $fatal(1, "Bad Out");
        end
    end
end

endmodule
```

Task MuxCheck ([7:0] rawT [2:0] selT outT)

raw = rawT; sel = selT;
#1
assert(out = outT)...

end task

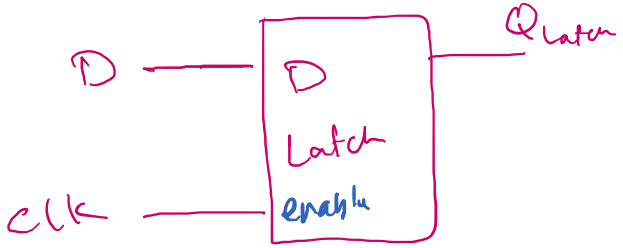
<- Could also use Task here!

Usually a testbench problem on Exam...

"RS, $\overline{RS}$, D Latch circuits on cheat sheet"

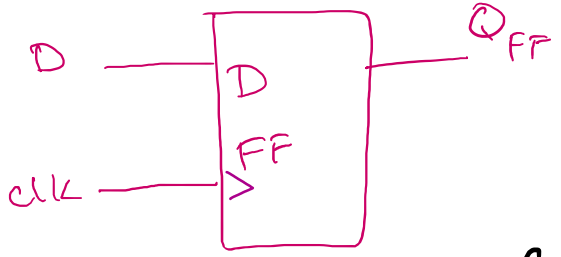
Timing Diagram

"level sensative"

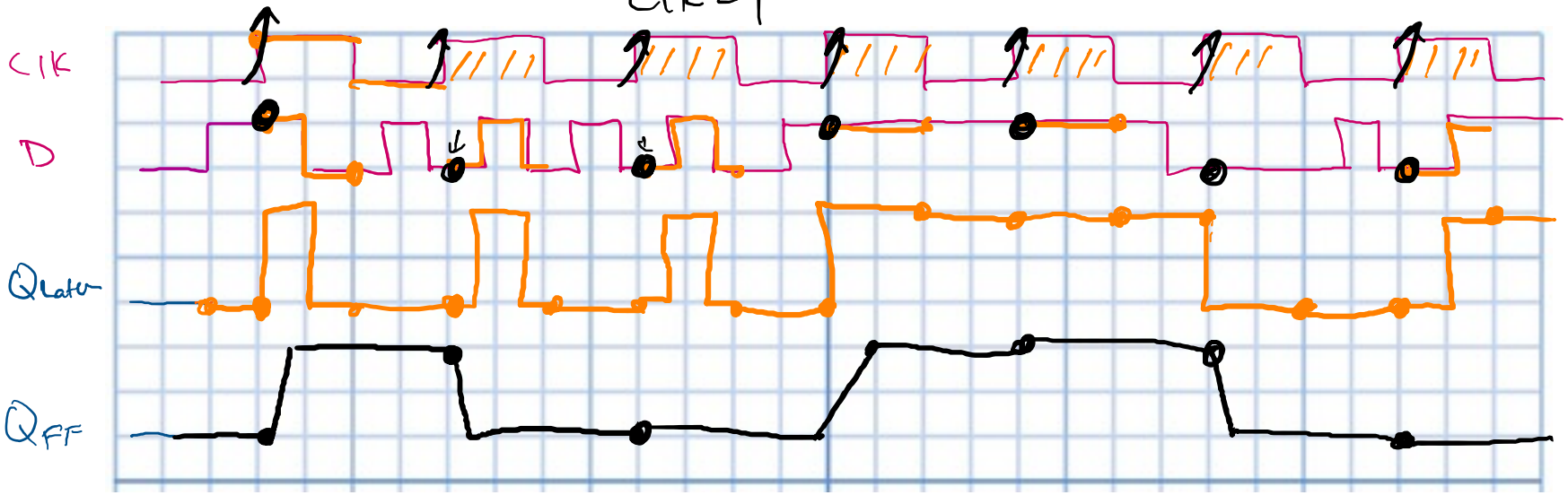


output follows input when clk = 1

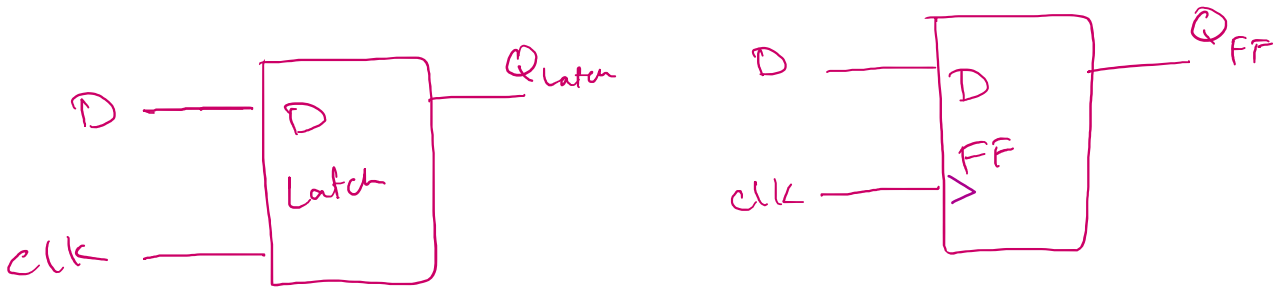
"edge sensative"

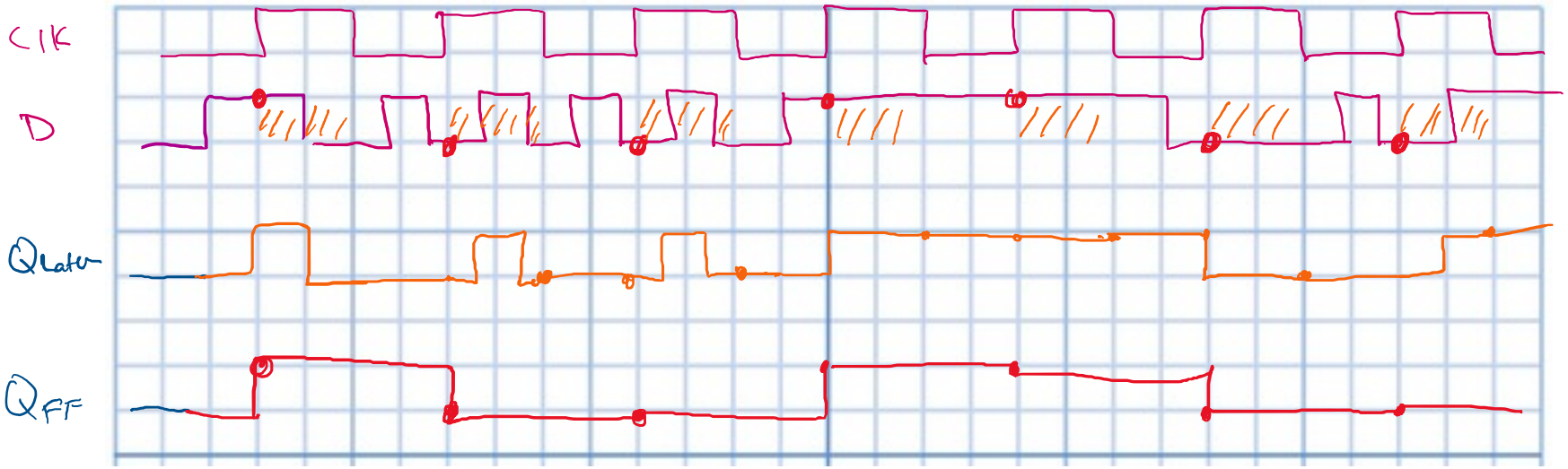


output follows input when clk is arising edge

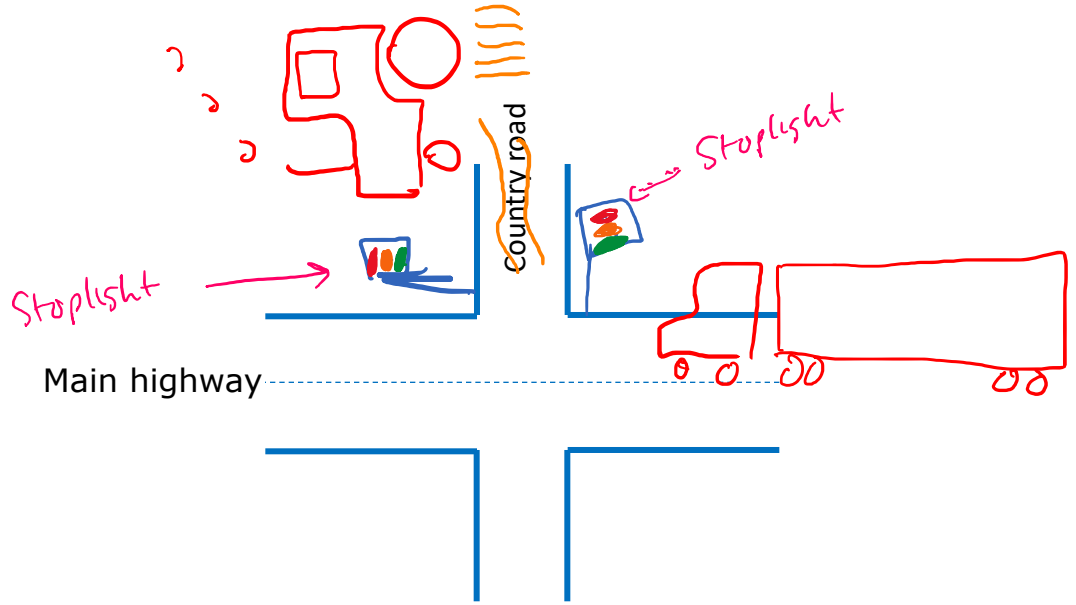


Timing Diagram





Example: Traffic Signal Controller



The main highway gets priority, should be normally green.

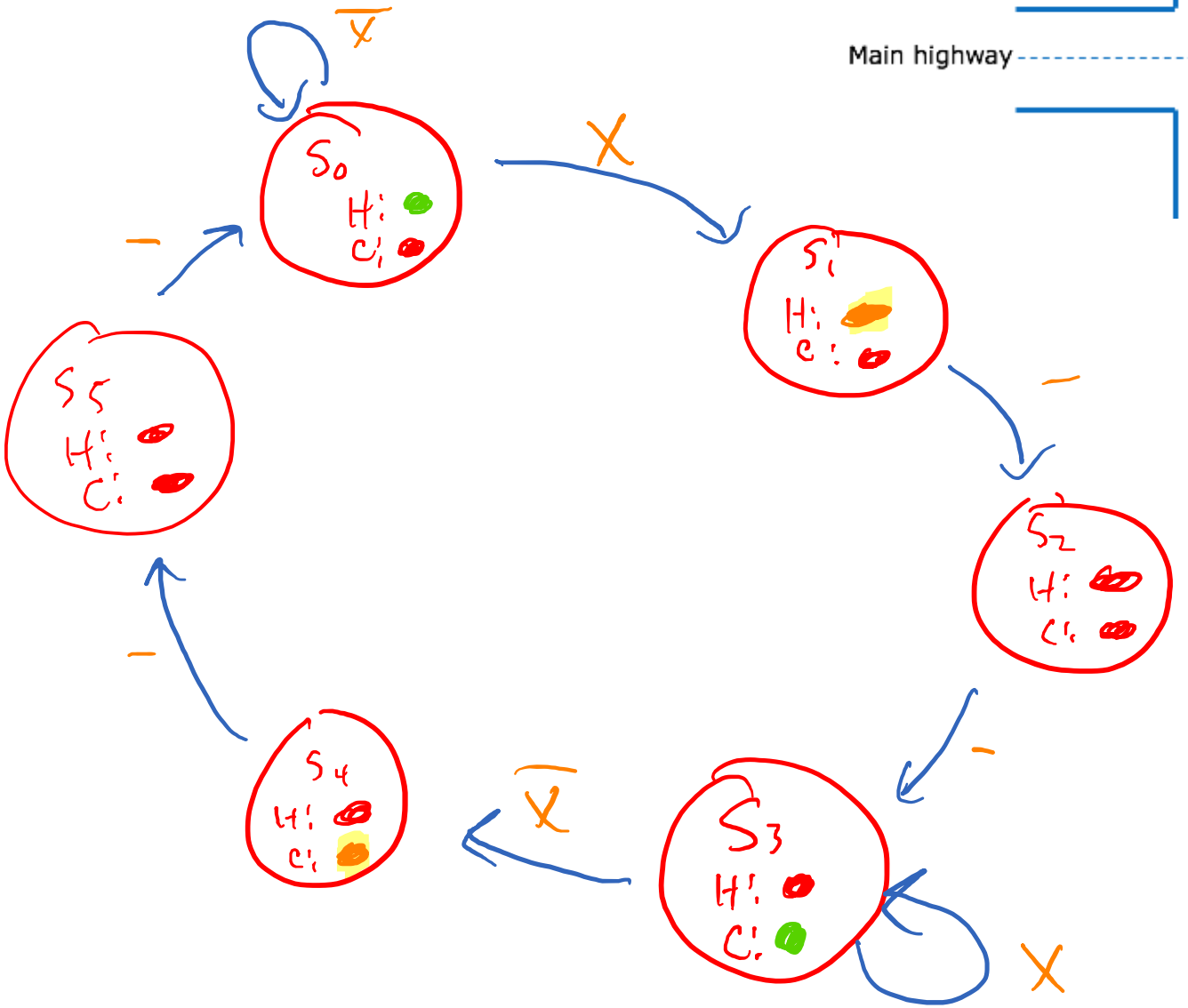
Cars from a country cause the traffic signal to turn green only long enough to let the cars on the country road go.

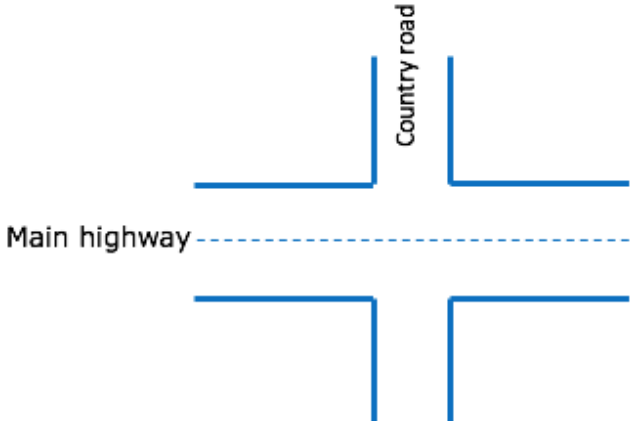
There is a sensor to detect cars waiting on the country road. The sensor sends a signal *X* as input to the controller:

X = 1, if there are cars on the country road

X = 0, otherwise

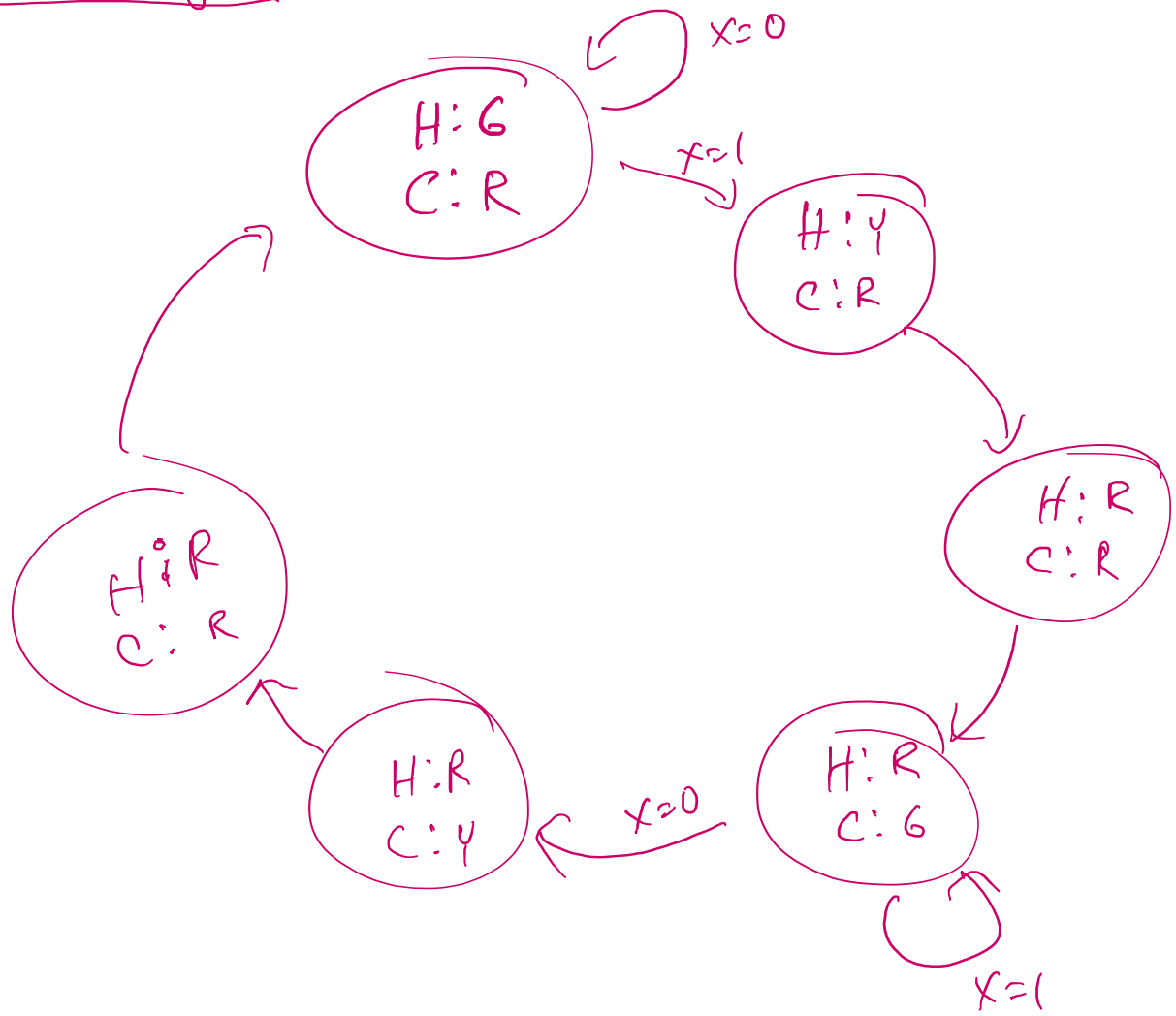
What are the `States'?



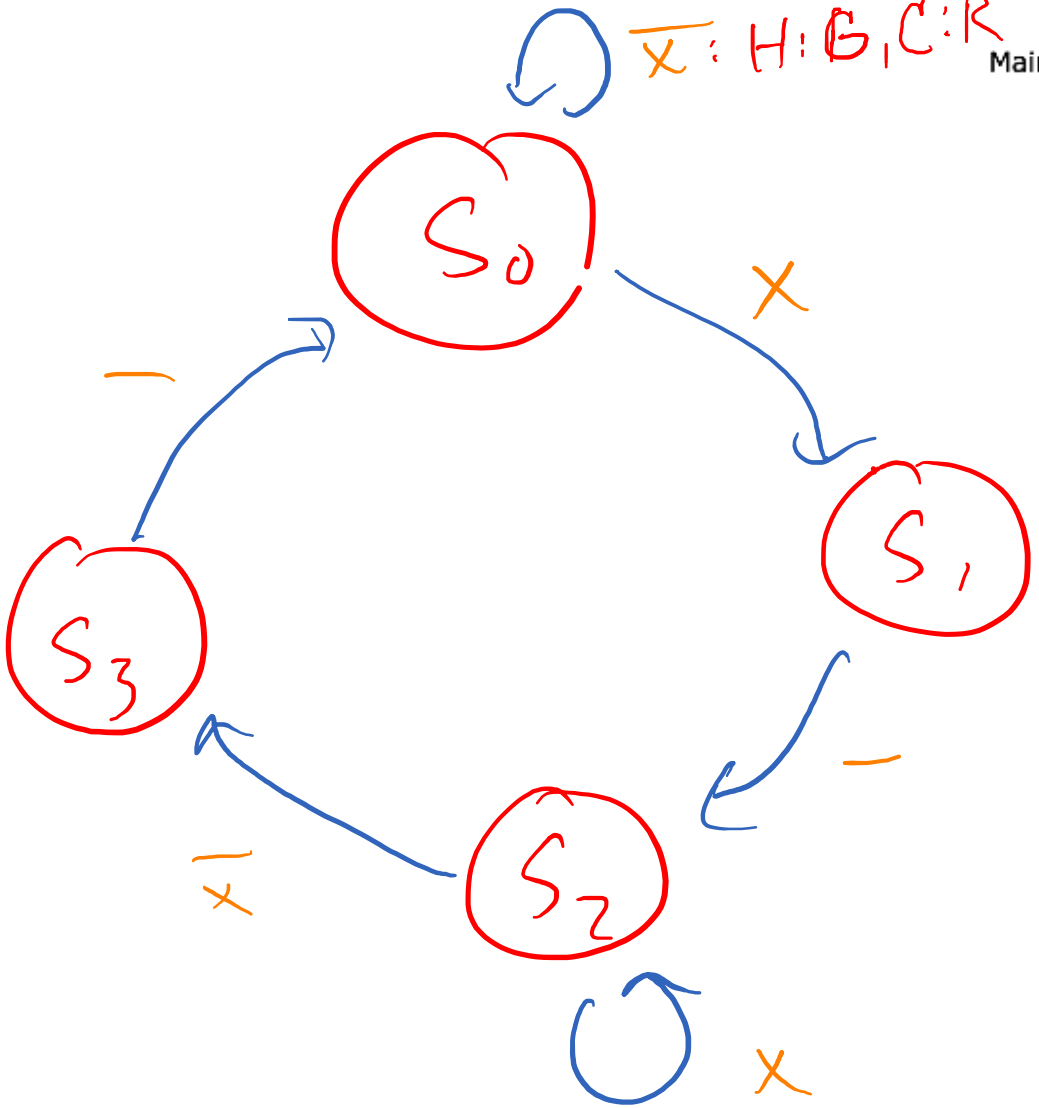


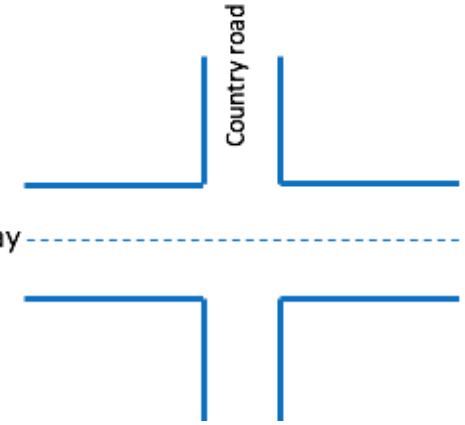
What are the `States'?

Moore Type



What are the `States'?

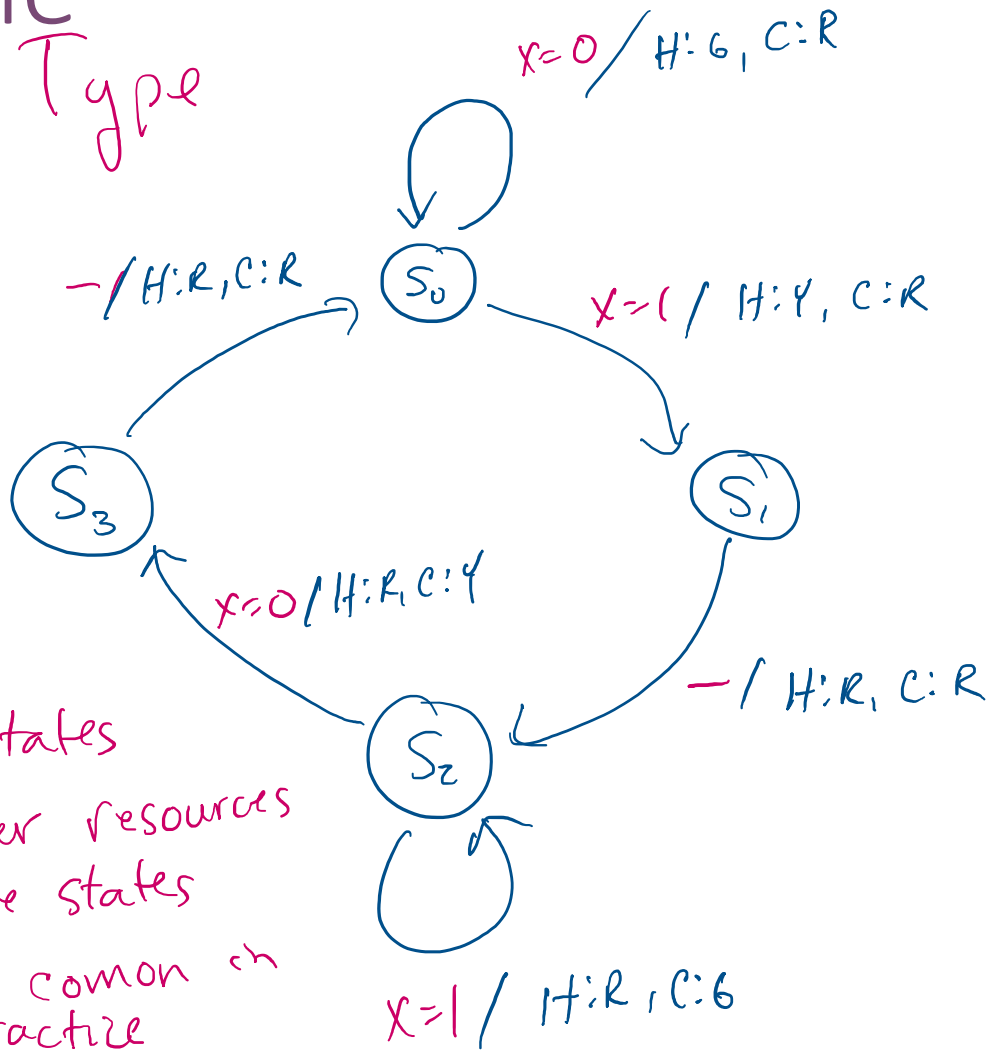




Incomplete

State Machine

Mealy Type



Mealy Type:

- → needs less states
- → needs fewer resources to store states
- → is more comon in practice

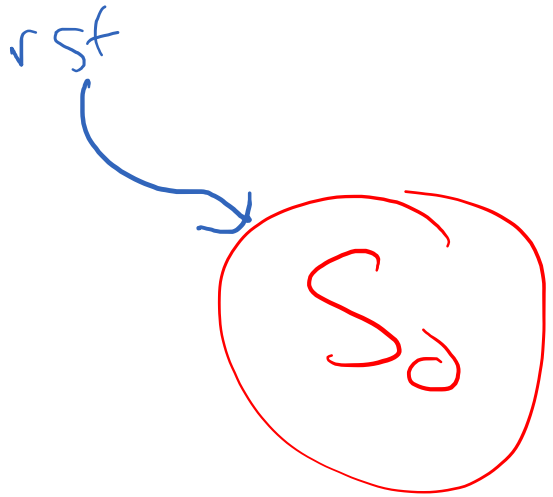
What's the FSM here?

```
module ReverseEngr (
    input clk, rst,
    input X,
    output Y,
);

enum {S0, S1} state, nextstate;

always_ff (@posedge clk) begin
    if (rst) state <= S0;
    else
       state <= nextstate;
end
```

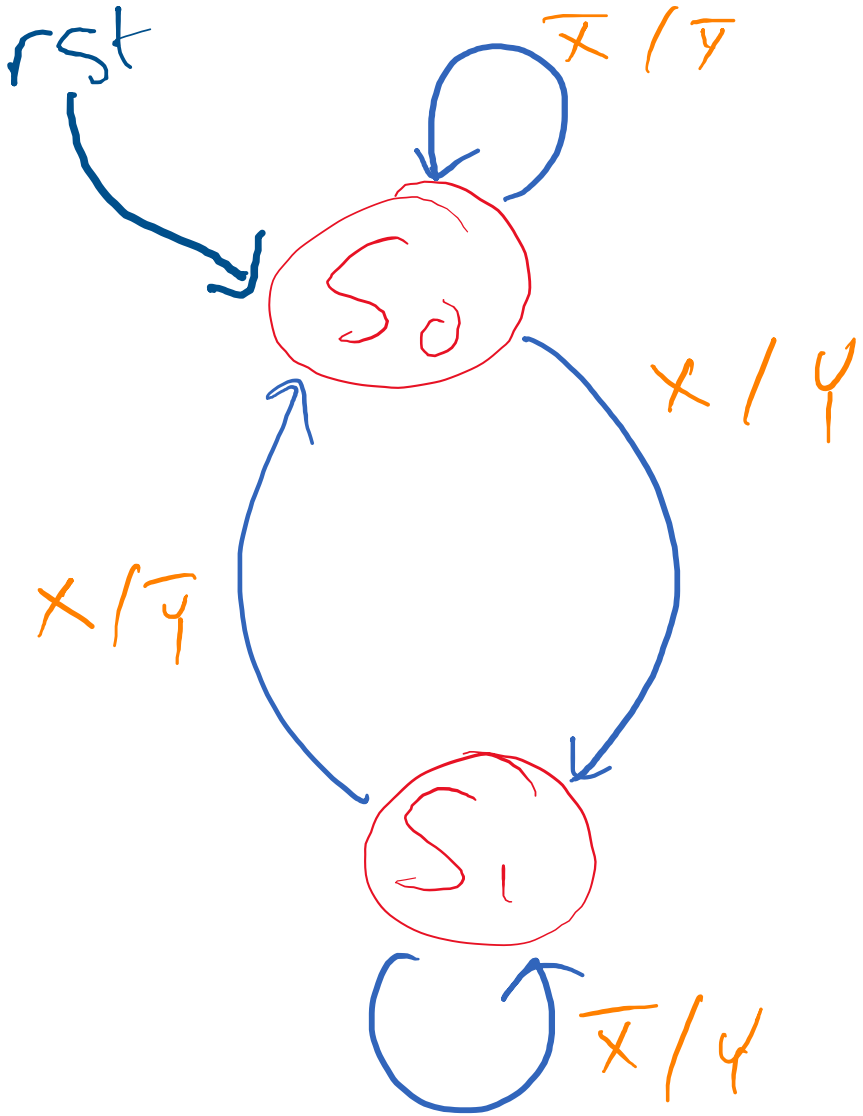
```
always_comb begin
    Y = 1'b0;
    case (state)
        S0: begin
            if (x) begin
                Y = 1;
                nextstate = S1
            end else begin
              nextstate = S0;
            end

        S1: begin
            if (X)
                nextstate = S0;
            else begin
                Y = 1;
                nextstate = S1;
            end
        end
    endcase
end
endmodule
```

What's the FSM here?

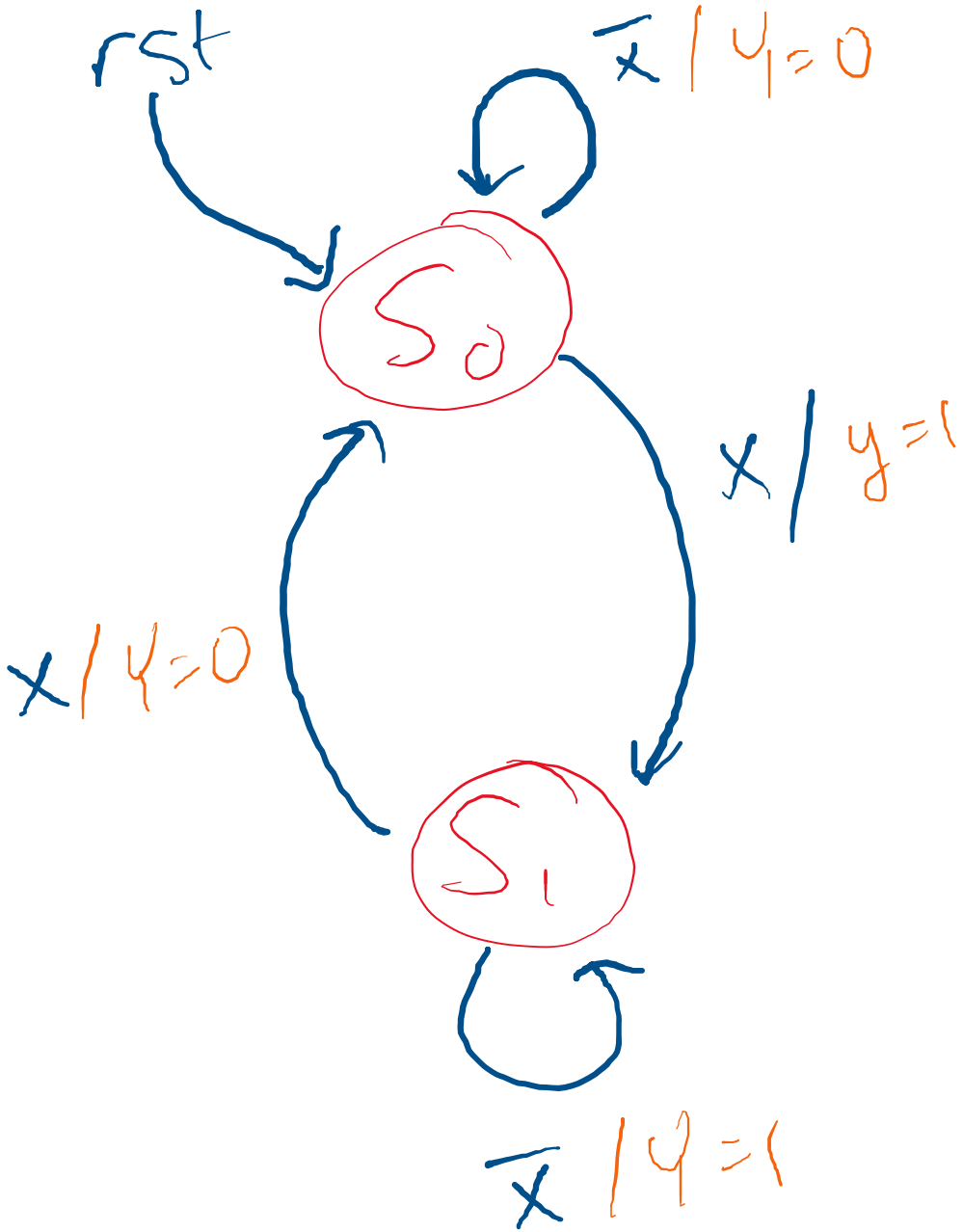
```
module ReverseEngr (
    input clk, rst,
    input X,
    output Y,
);

enum {S0, S1} state, nextstate;

always_ff (@posedge clk) begin
    if (rst) state <= S0;
    else
       state <= nextstate;
end
```



What's the FSM here?




```
always_comb begin
    Y = 1'b0;
    case (state)
        S0: begin
            if (x) begin
                Y = 1;
                nextstate = S1
            end else begin
              nextstate = S0;
            end

        S1: begin
            if (X)
                nextstate = S0;
            else begin
                Y = 1;
                nextstate = S1;
            end
        end
    endcase
end
endmodule
```


rst
$\bar{x}$ / y=0
S_0
x / y=1
x / y=0
S_1
$\bar{x}$ / y=1

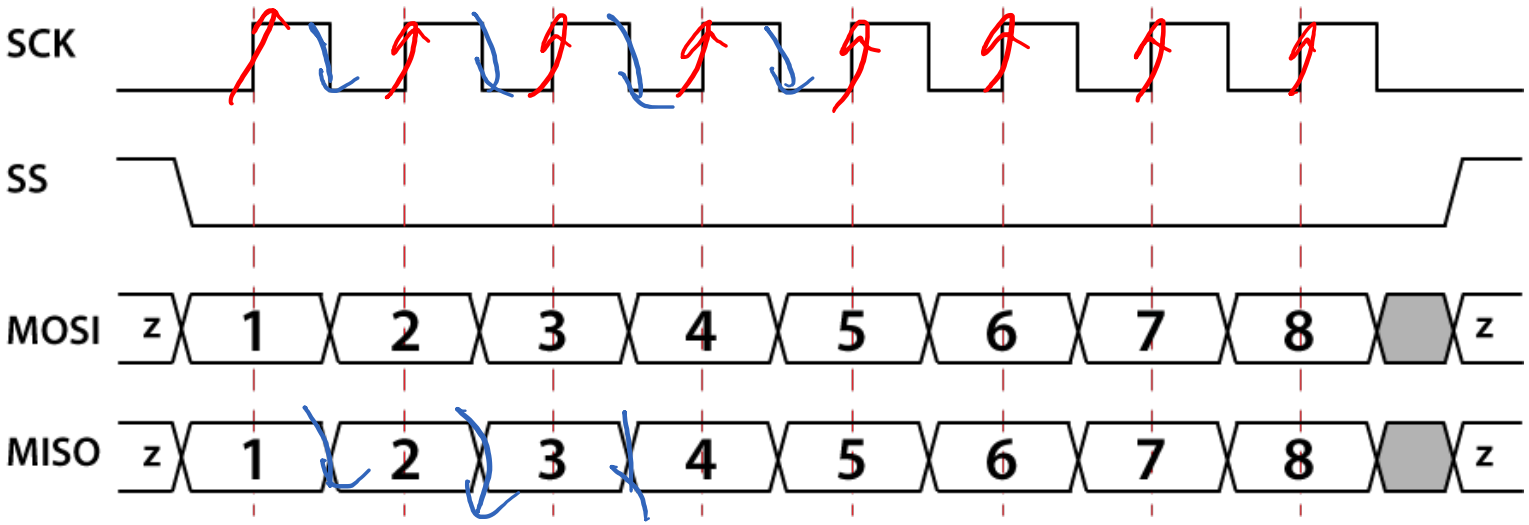
SPI

- Incoming Data:

rising edge

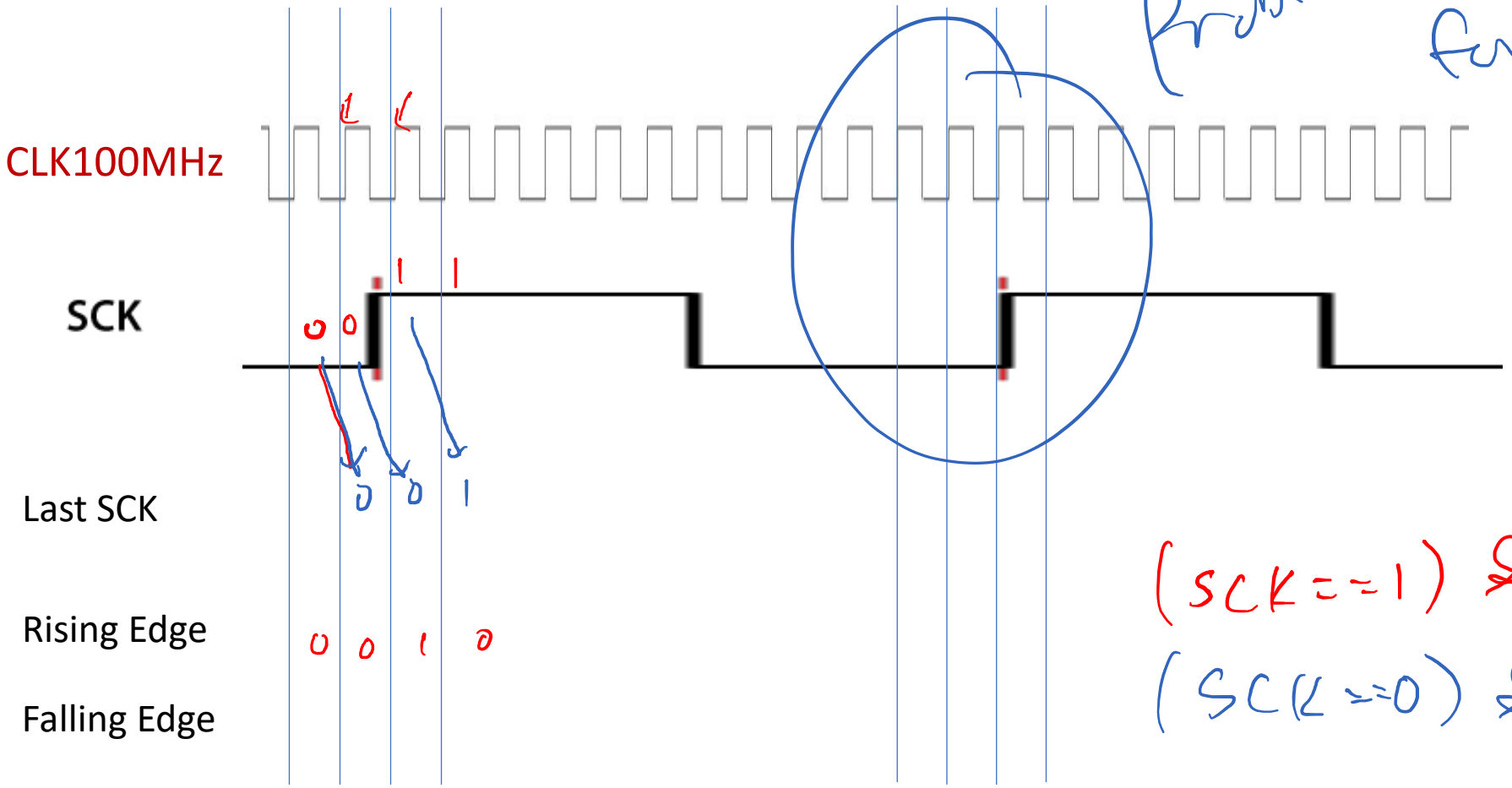
- Outgoing Data:

falling edge



always_ff @(posedge clk)
last sclk <= sck;

Finding SCLK Edges



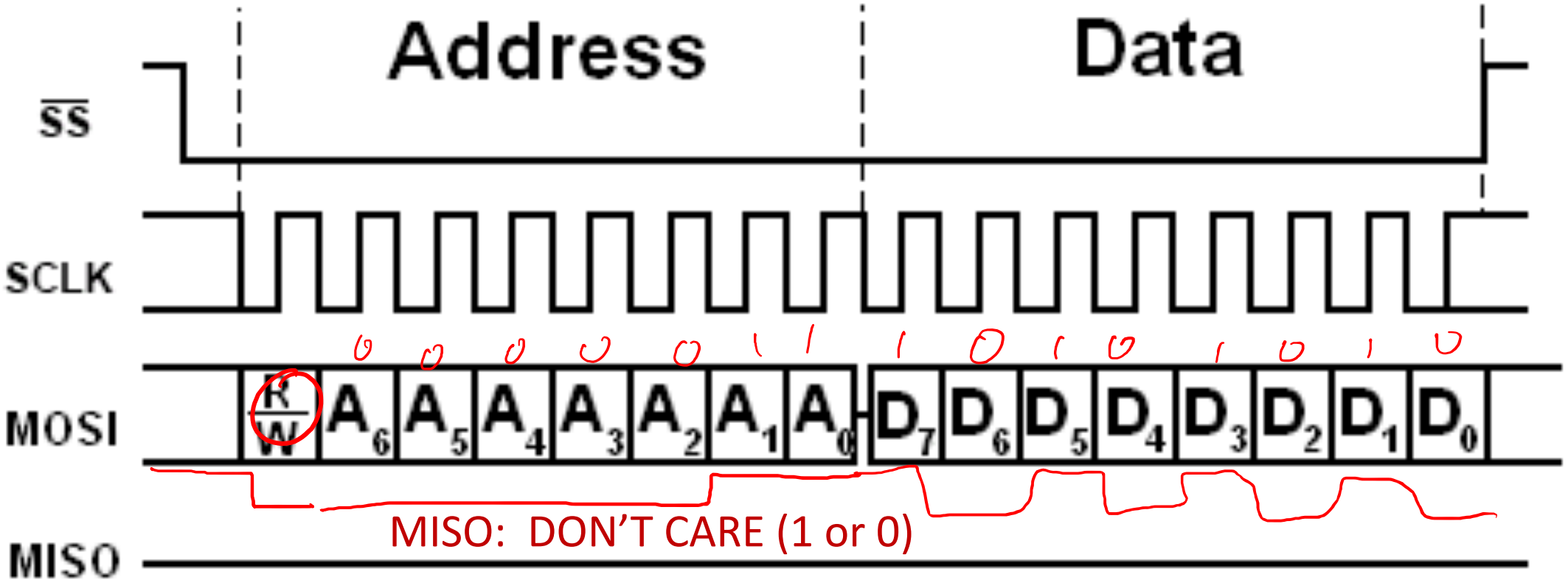
Problem ... ignore for now.

(sck == 1) && (last_sck == 0)

(sck == 0) && (last_sck == 1)

Write Protocol

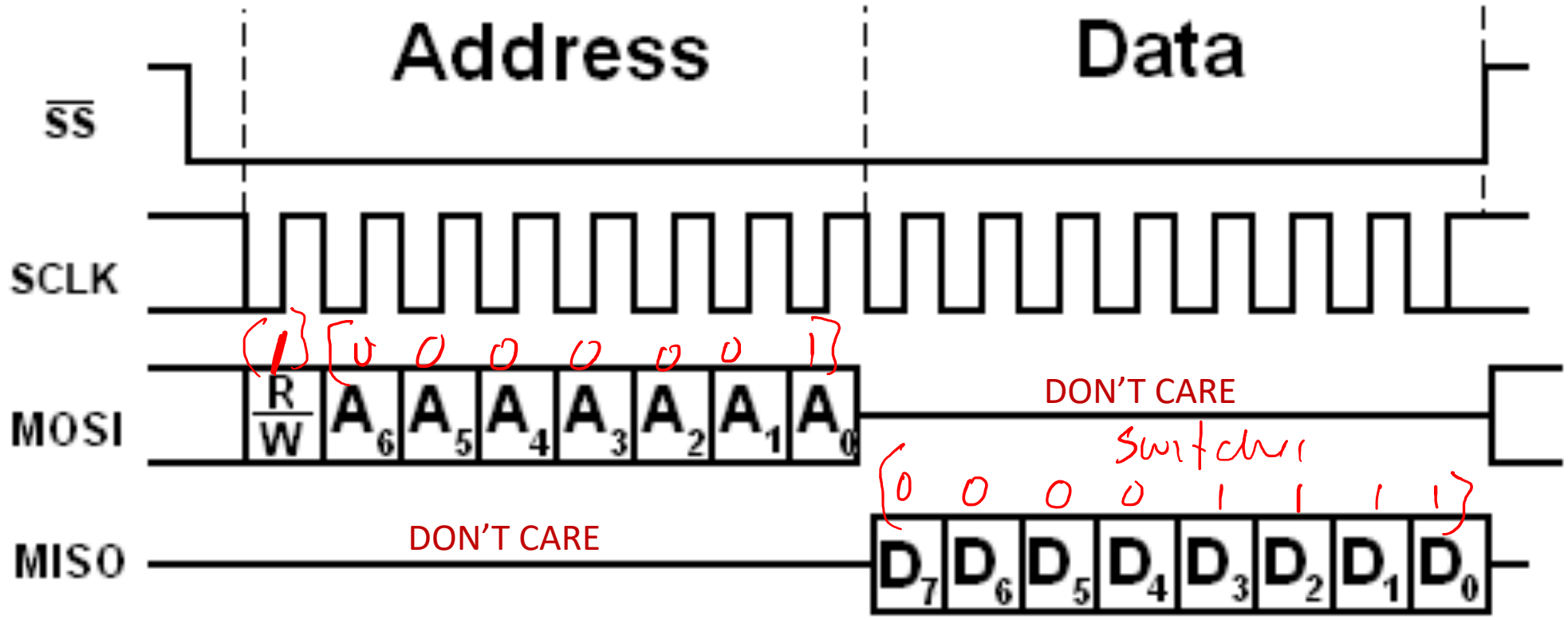
Operation: WRITE (`'h0`)
Address: LEDs (`'h3`)
Data: ON-OFF-ON-OFF (`'b10101010`)



[pyroelectro]

Read Protocol

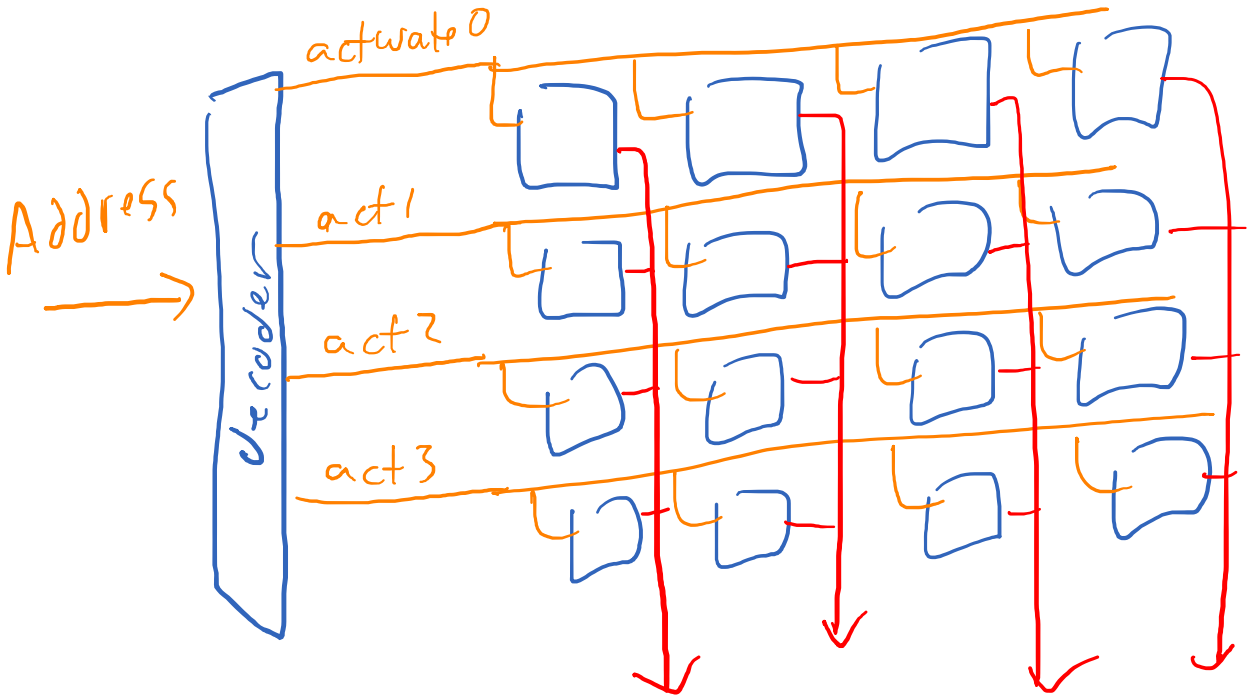
Operation: READ (`'h1`)
Address: SWITCHES (`'h1`)
Data: ??



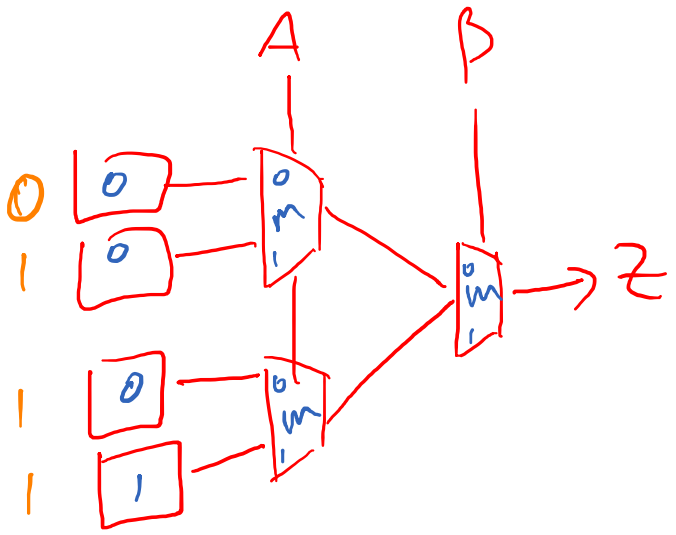
[pyroelectro]

Fpgas

Memory Array



Lookup Table



Z = A & B

A	B	Z
0	0	0
0	1	0
1	0	0
1	1	1

Z = A | B

A	B	Z
0	0	0
0	1	1
1	0	1
1	1	1