ENGR 210 / CSCI B441
"Digital Design"

Sequential Logic
FSMs

Andrew Lukefahr

Announcements

- P7 Saturating Counter is out
- P8 – Elevator Controller is out
  - This one is hard.

Always specify defaults for `always_comb`!

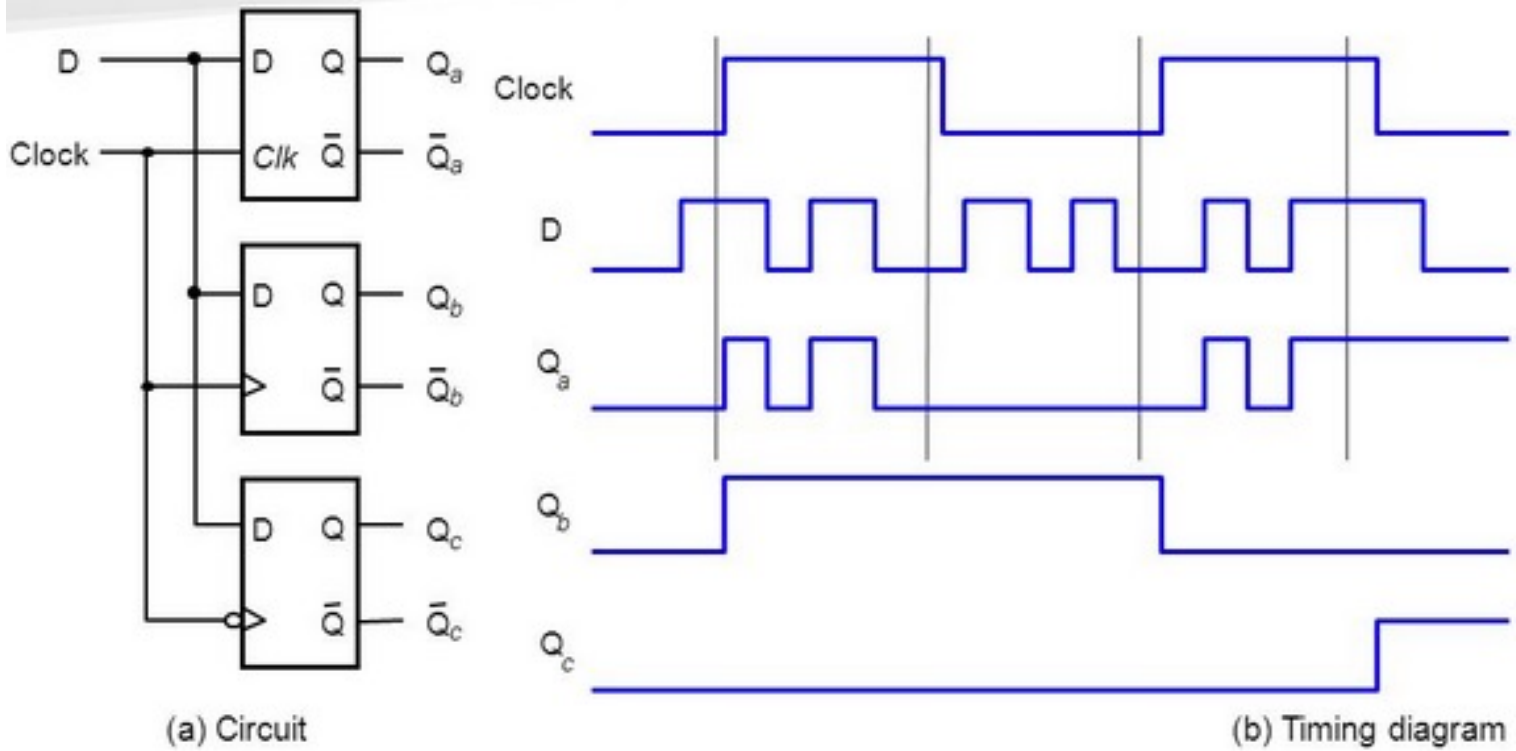
Inferred Latches are bad…

```
WARNING: [Synth 8-327] inferring latch for
variable 'count_reg'
[/home/autograder/working_dir/src/saturating_cou
nter.sv:XX]
```

BLOCKING (=) FOR `always_comb`

NON-BLOCKING (<=) for `always_ff`

D Latch versus D Flip-Flop



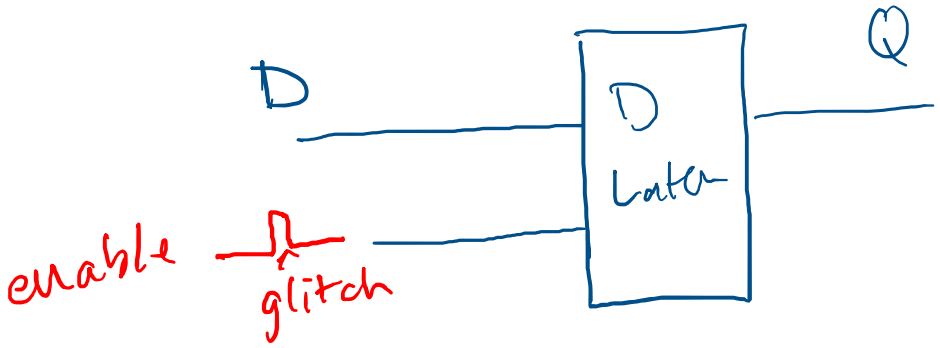
(a) Circuit

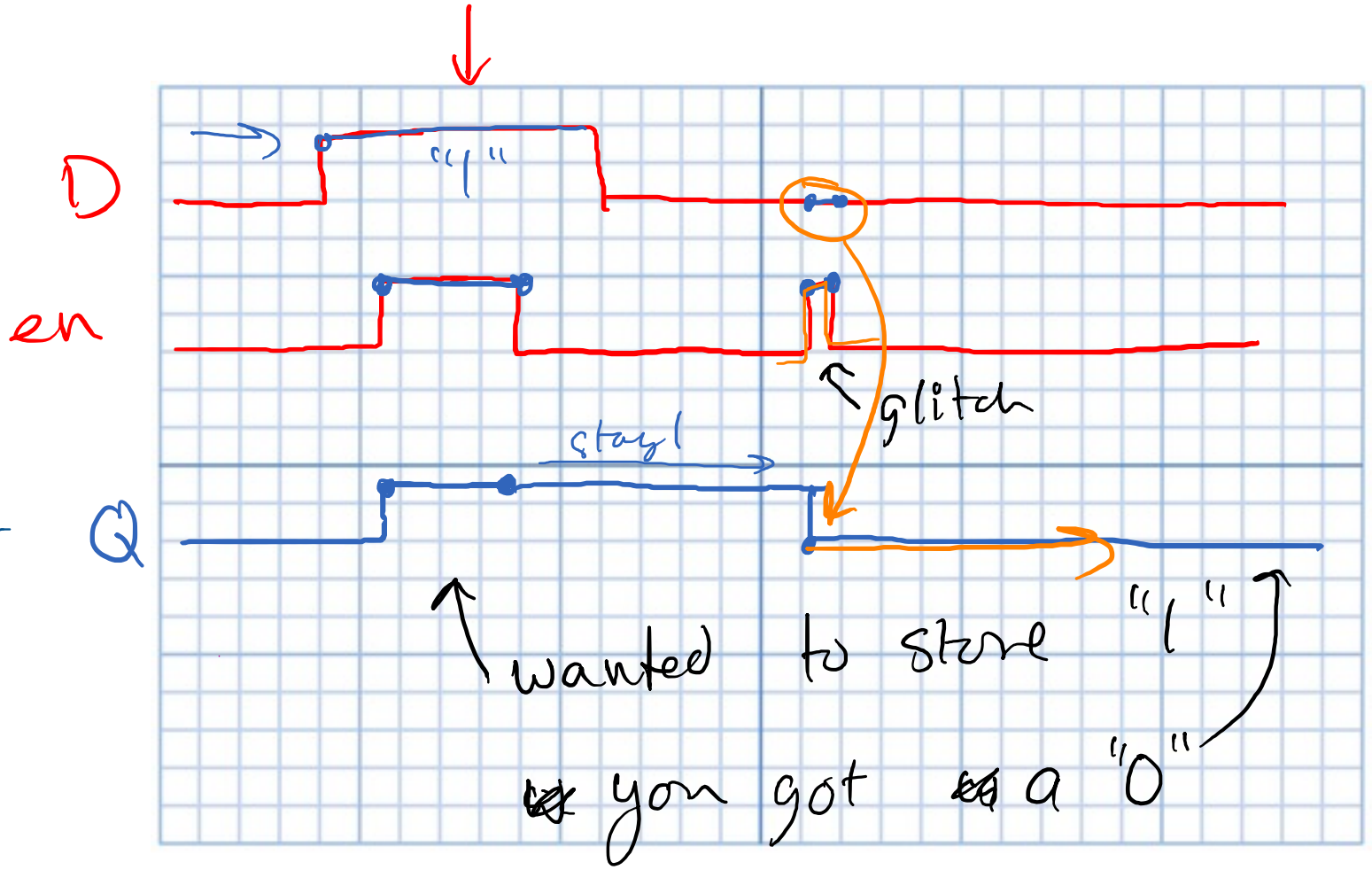
(b) Timing diagram

Comparison of level-sensitive and edge-triggered devices

51

Glitches on D-Latches





Flip-Flop in Verilog

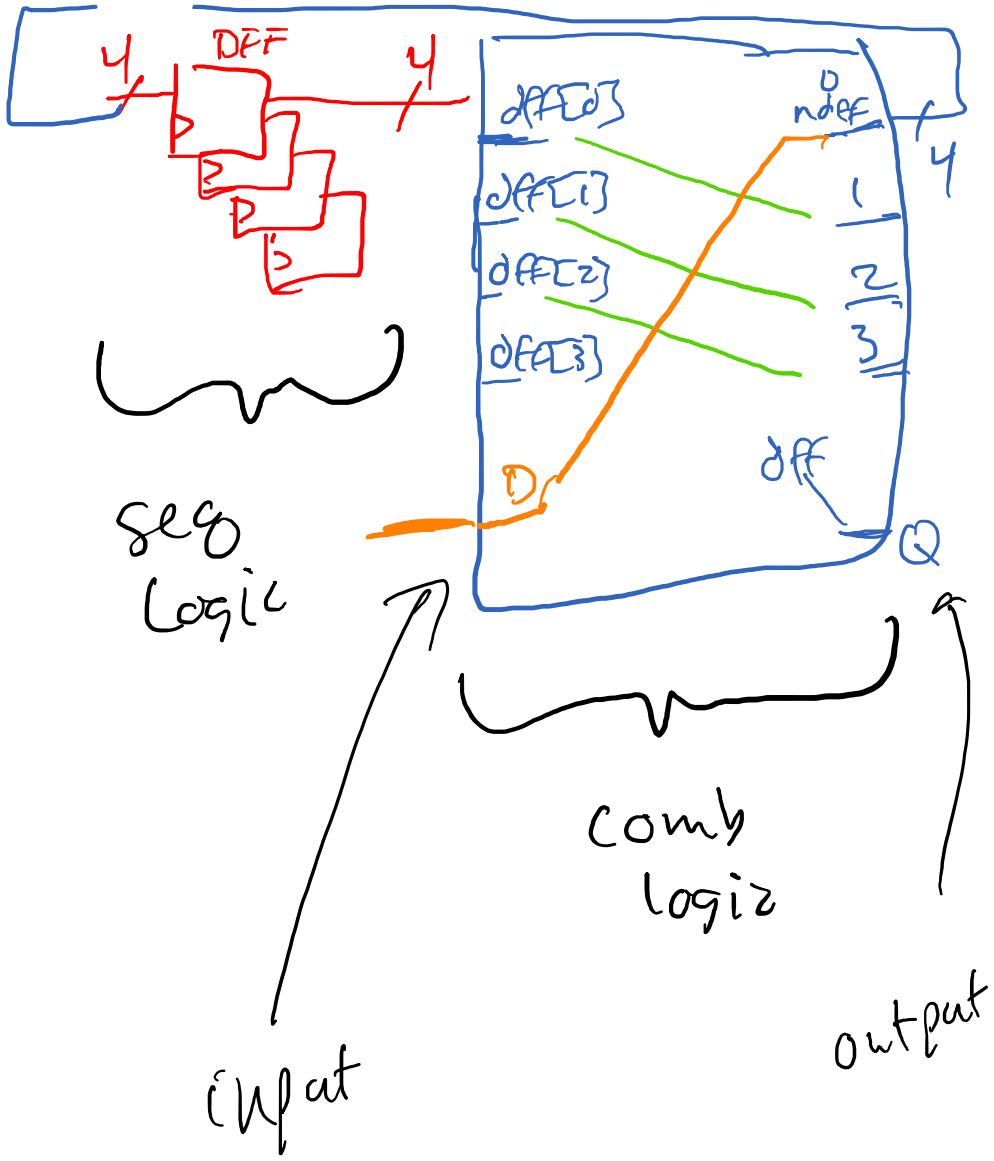
```
module d_ff (
    input          d,   //data
    input          clk, //clock
    output logic q    //output register
);

     always_ff @( posedge clk )
     begin
          q <= d; //non-blocking assign
     end

endmodule
```

Shift-Register in Verilog

```
module shift_register (
   input clk, rst, D,
   output [3:0] Q );

   logic [3:0] dff;
   logic [3:0] next_dff;

   always_ff(@posedge clk) begin
      if (rst) dff <= 4'h0;
      else     dff <= next_dff;
   end

   always_comb
      next_dff = { dff[2:0], D};

   assign Q = dff;

endmodule
```

Shift-Register in Verilog

```
module shift_register (
    input clk, rst, D,
    output [3:0] Q );

    logic [3:0] dff;
    logic [3:0] next_dff;

    always_ff(@posedge clk) begin
       if (rst) dff <= 4'h0;
       else     dff <= next_dff;
    end

    always_comb
       next_dff = { dff[2:0], D};

    assign Q = dff;

endmodule
```



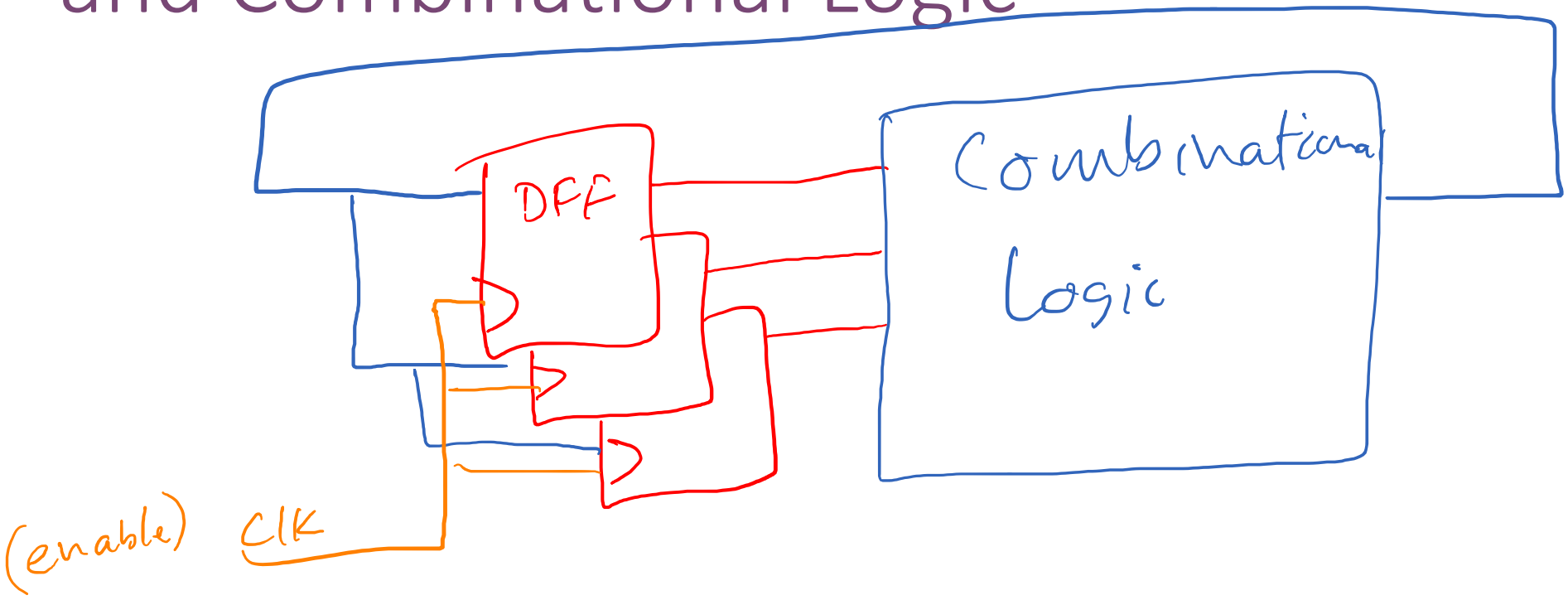
What does this module do?

```
module mystery(
    input                   clk,      //clock
    input                   rst,      //reset
    output logic   out      //output
);
    logic [3:0] Q;
    logic [3:0] sum;

    always_ff @( posedge clk ) // <- sequential logic
    begin
         if (rst) Q <= 4'h0;
         else     Q <= sum;    //non-blocking
    end

    always_comb begin // <- combinational logic
        sum = Q + 4'h1;  //blocking
        out = sum[3];
    end

endmodule
```

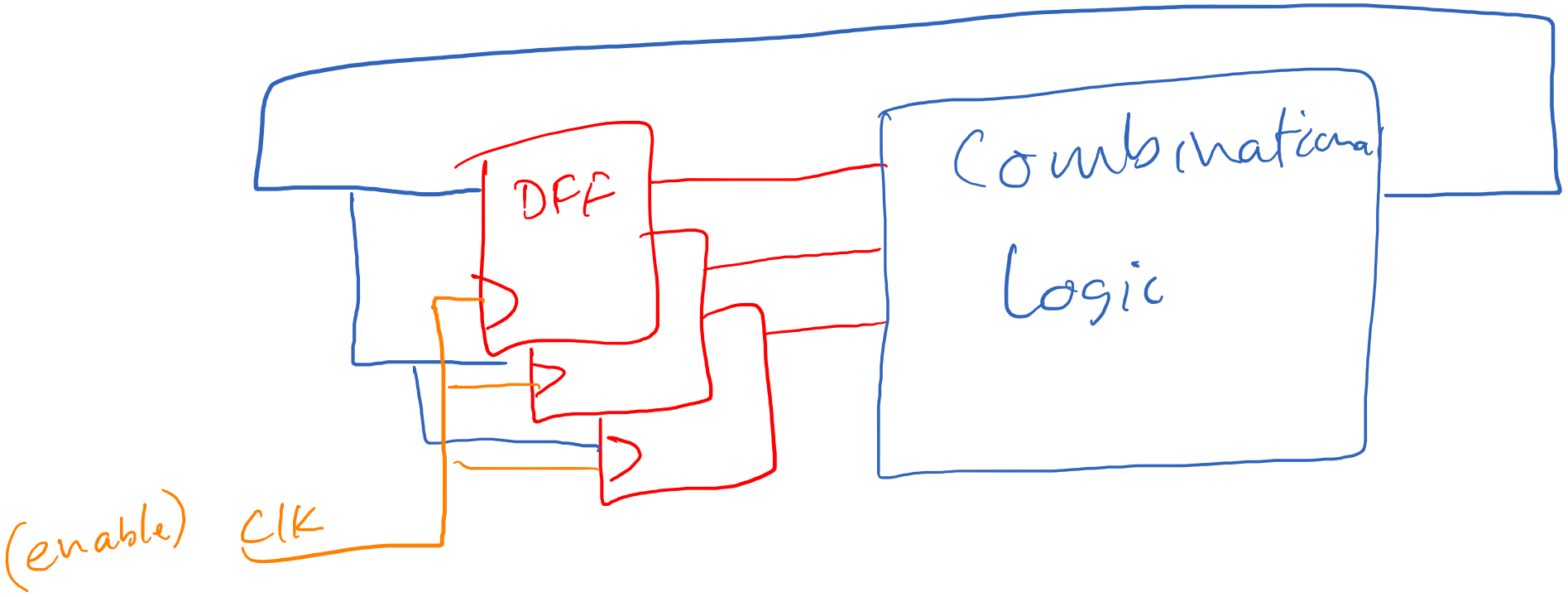
What does this module do?

```
module counter(
    input                   clk,      //clock
    input                   rst,      //reset
    output logic   out      //output
);
    logic [3:0] Q;
    logic [3:0] sum;

    always_ff @( posedge clk ) // <- sequential logic
    begin
         if (rst) Q <= 4'h0;
         else     Q <= sum;    //non-blocking
    end

    always_comb begin // <- combinational logic
        sum = Q + 4'h1;  //blocking
        out = sum[3];
    end

endmodule
```

Sequential Logic uses both Flip-Flops and Combinational Logic



Inputs can affect output or state



What's wrong here?

```
logic [1:0] count;

always_comb begin
      count = count; //default

      if (foo) begin
            count = count + 1;
      end else if (bar) begin
            count = count - 1;
      end
end
```

What's wrong here?

```
logic [1:0] count;

always_comb begin
      count = count; //not a default

      if (foo) begin
            count = count + 1; //self reference
      end else if (bar) begin
            count = count - 1; //self reference
      end
end
```

Count needs to be stateful.

```
logic [1:0] count, nextCount;

always_ff @(posedge clk) begin
        if (rst) count <= 2'h0;
        else     count <= nextCount;
end

always_comb begin
        nextCount = count; //default

        if (foo) begin
                nextCount = count + 1;
        end else if (bar) begin
                nextCount = count - 1;
        end
end
```

Finite State Machines (FSMs)

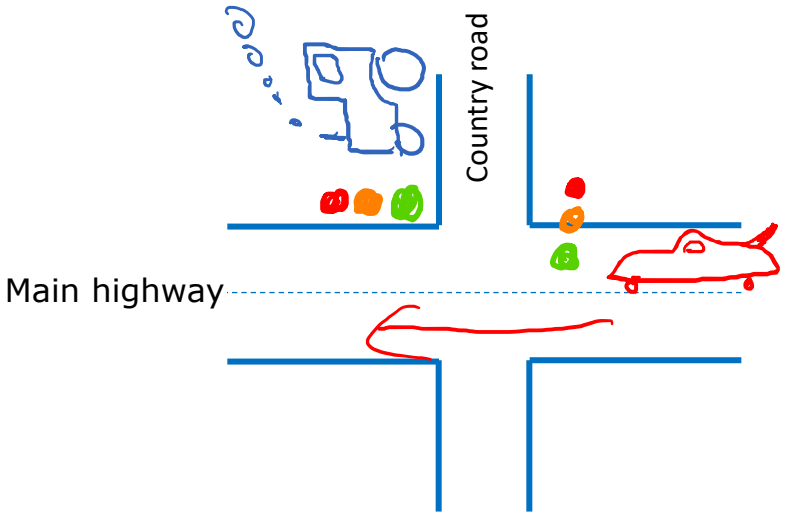
Finite State Machines (FSMs)

- A **finite-state machine (FSM)** or **finite-state automaton** (**FSA**, plural: *automata*), **finite automaton**, or simply a **state machine**, is a mathematical model of computation.
- It is an abstract machine that can be in exactly one of a finite number of *states* at any given time.
- The FSM can change from one state to another in response to some inputs; the change from one state to another is called a *transition*.[1] [wiki]

States, Transitions, and Guards

FSM: Traffic Signal Controller

- A controller for traffic at the intersection of a main highway and a country road.



- The main highway gets priority because it has more cars
  - The main highway signal remains green by default.

Traffic signal controller

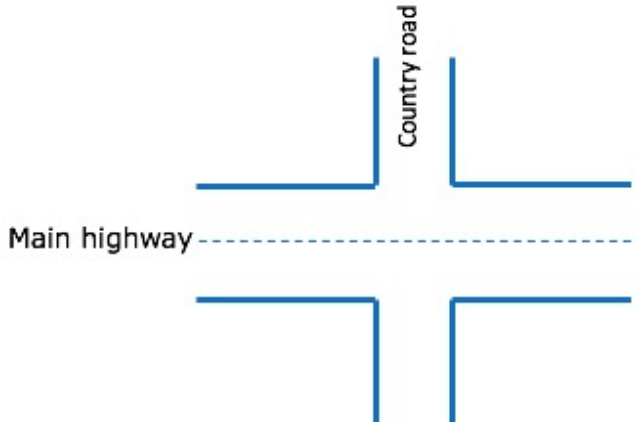
- Cars occasionally arrive from the country road. The traffic signal for the country road must turn green only long enough to let the cars on the country road go.
- When no cars are waiting on the country road, the country road traffic signal turns yellow then red and the traffic signal on the main highway turns green again.

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

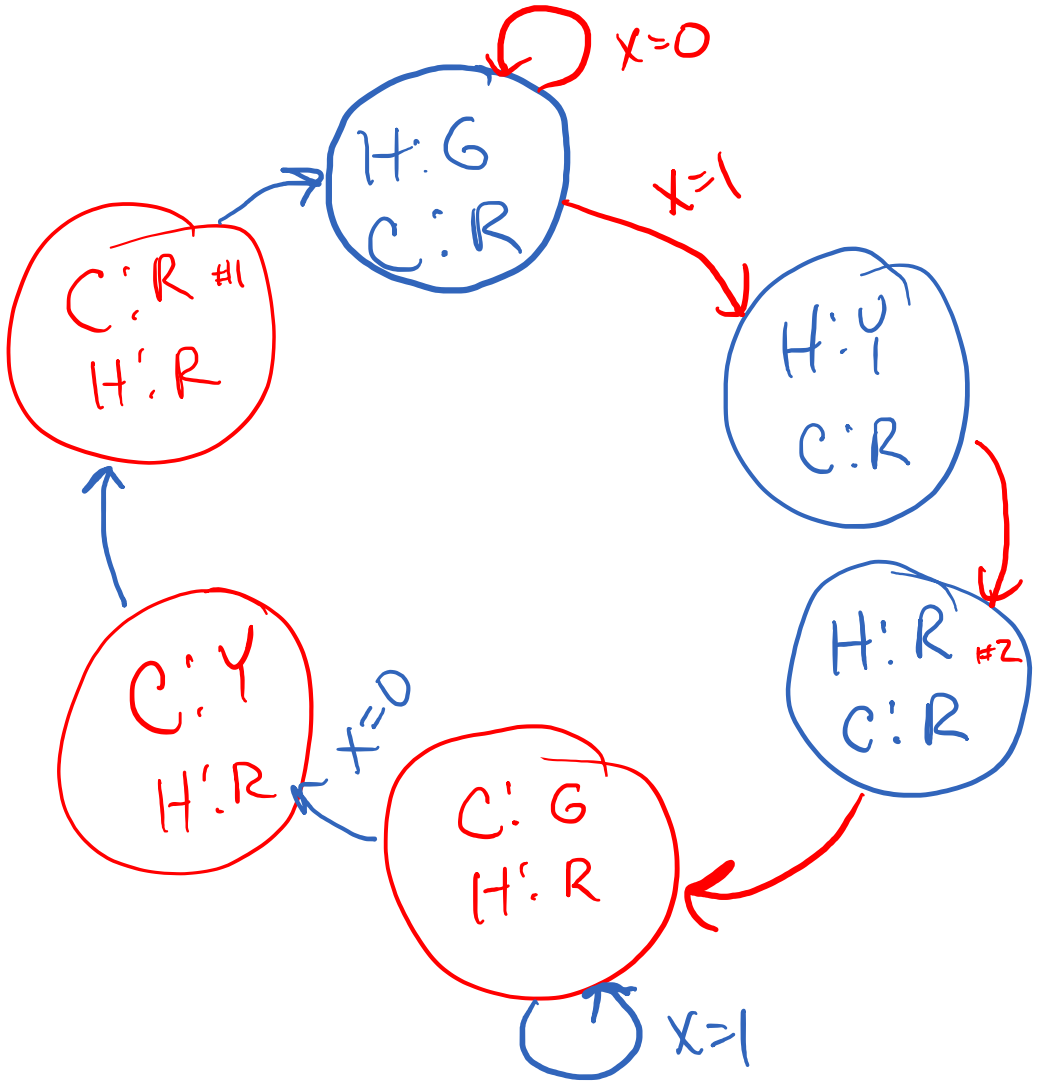
$X = 1$, if there are cars on the country road

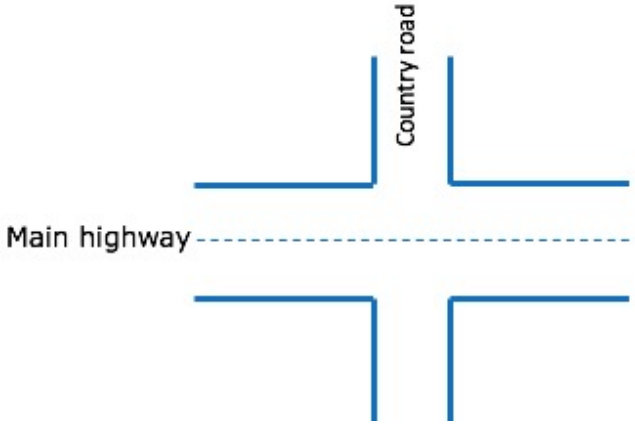
$X = 0$, otherwise

What are the `States’?



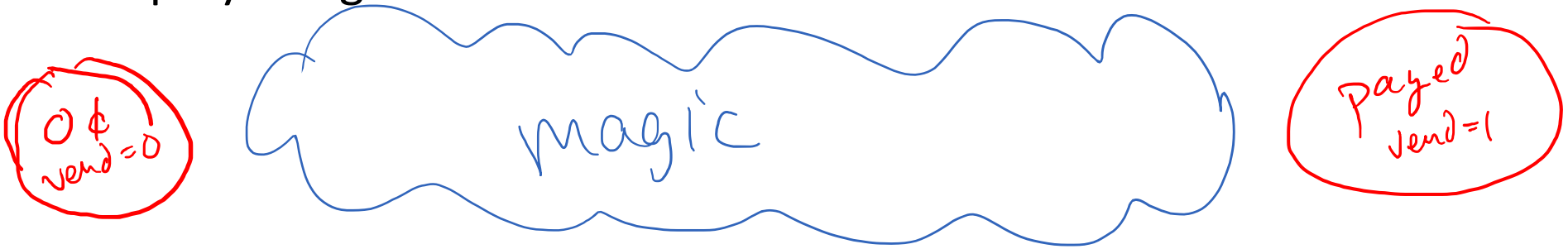
What are the `States'?





FSM: Simple Vending Machine

- You are designing a Vending Machine that dispenses Widgets for $0.25/each.
- Your machine must accept any combination of nickels (`N`), dimes (`D`), and quarters (`Q`) to pay for the Widget.
- When the correct payment is secured, you dispense the Widget (`vend`), and reset the payment.
- If a customer overpays, you keep the extra money. ☺
  - Just to simplify things…

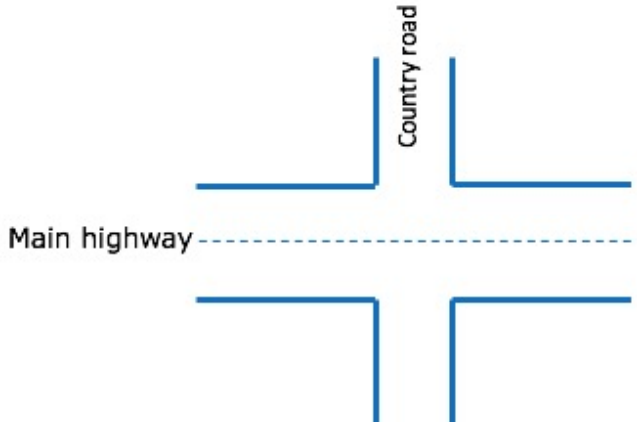


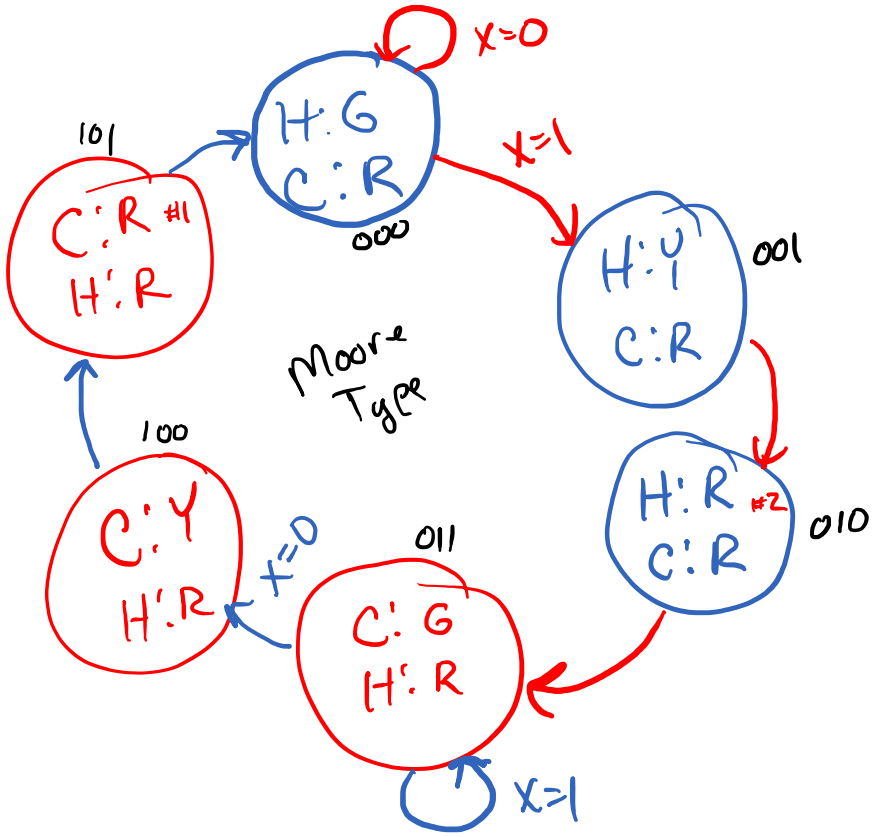
FSM: Vending Machine

Moore vs. Mealy Type FSMs

- Thus far we've done "Moore" Type
  - Moore Type: Outputs determined by the state (circle)
- Another technique: "Mealy" Type
  - Mealy Type: Output determined by the transition (arrow)
- Moore: Easier, but more states
- Mealy: Less states, more complicated transitions

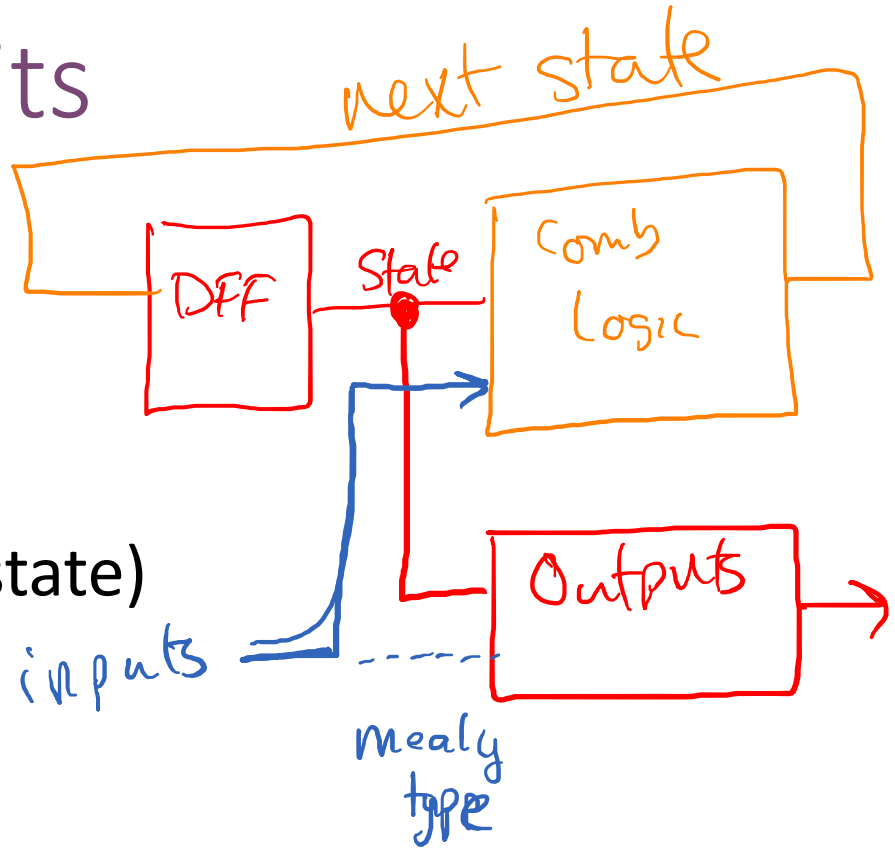
Traffic Light: Moore vs. Mealy



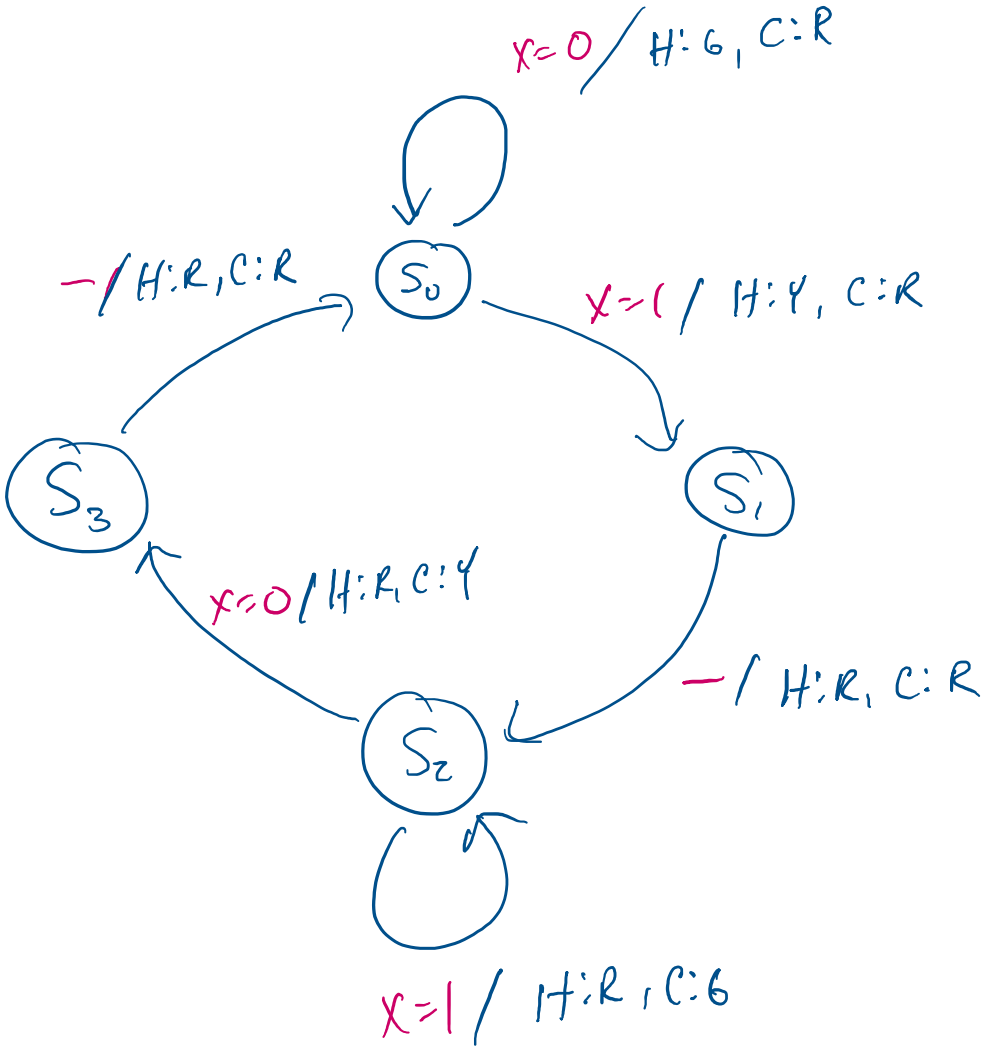


Implementing FSMs with Circuits

- Encode each state as a number
  - Store this with DFF's
- Generate state transition logic (arrow to next state)
  - Use combinational logic
- Generate output given state + inputs
  - Use combinational logic



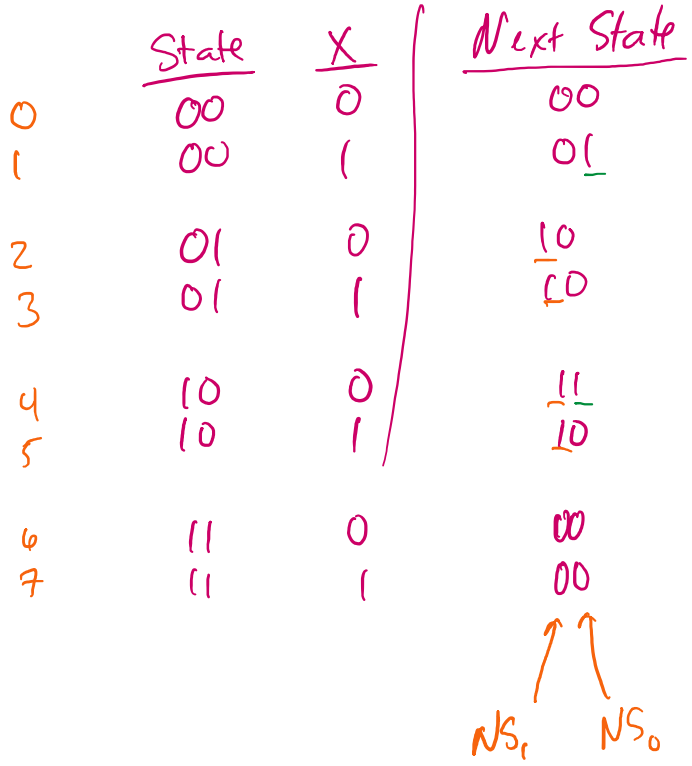
State Transition Encoding

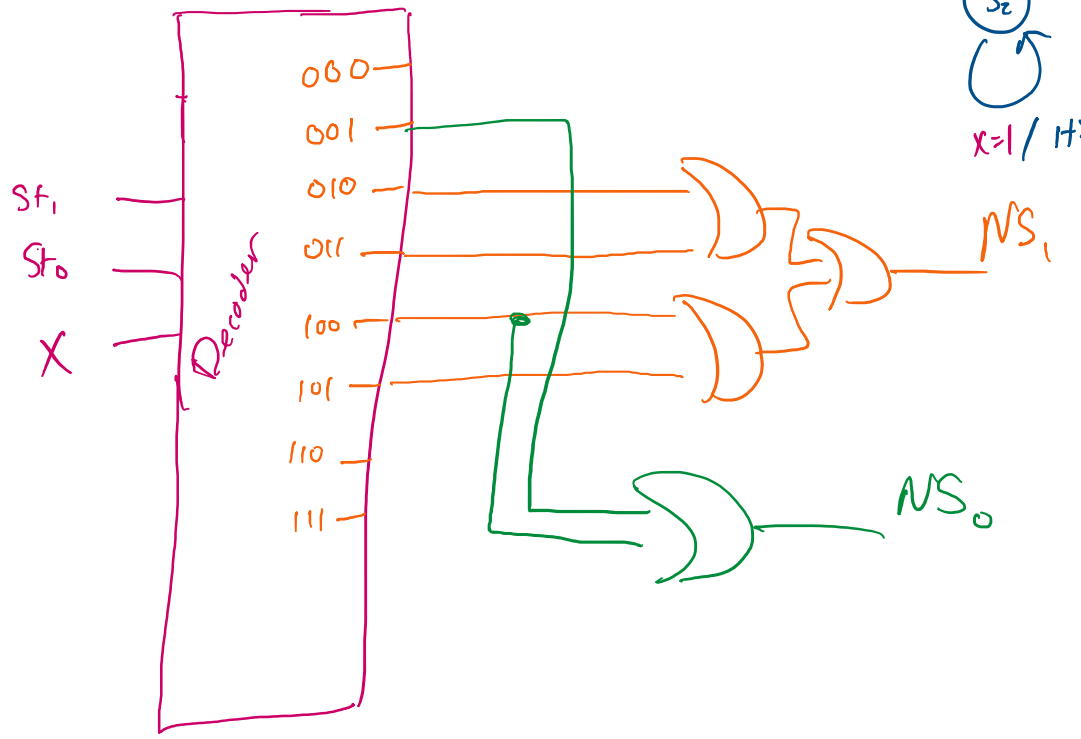


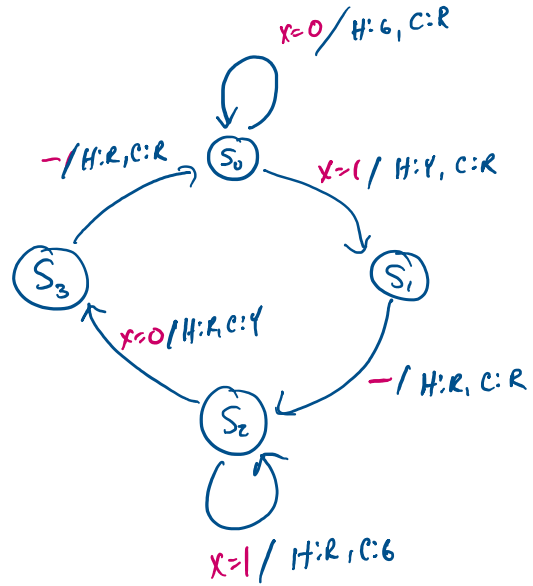
State Machine Encoding

	State	X	Next State
0	00	0	00
1	00	1	01
2	01	0	10
3	01	1	10
4	10	0	11
5	10	1	10
6	11	0	00
7	11	1	00

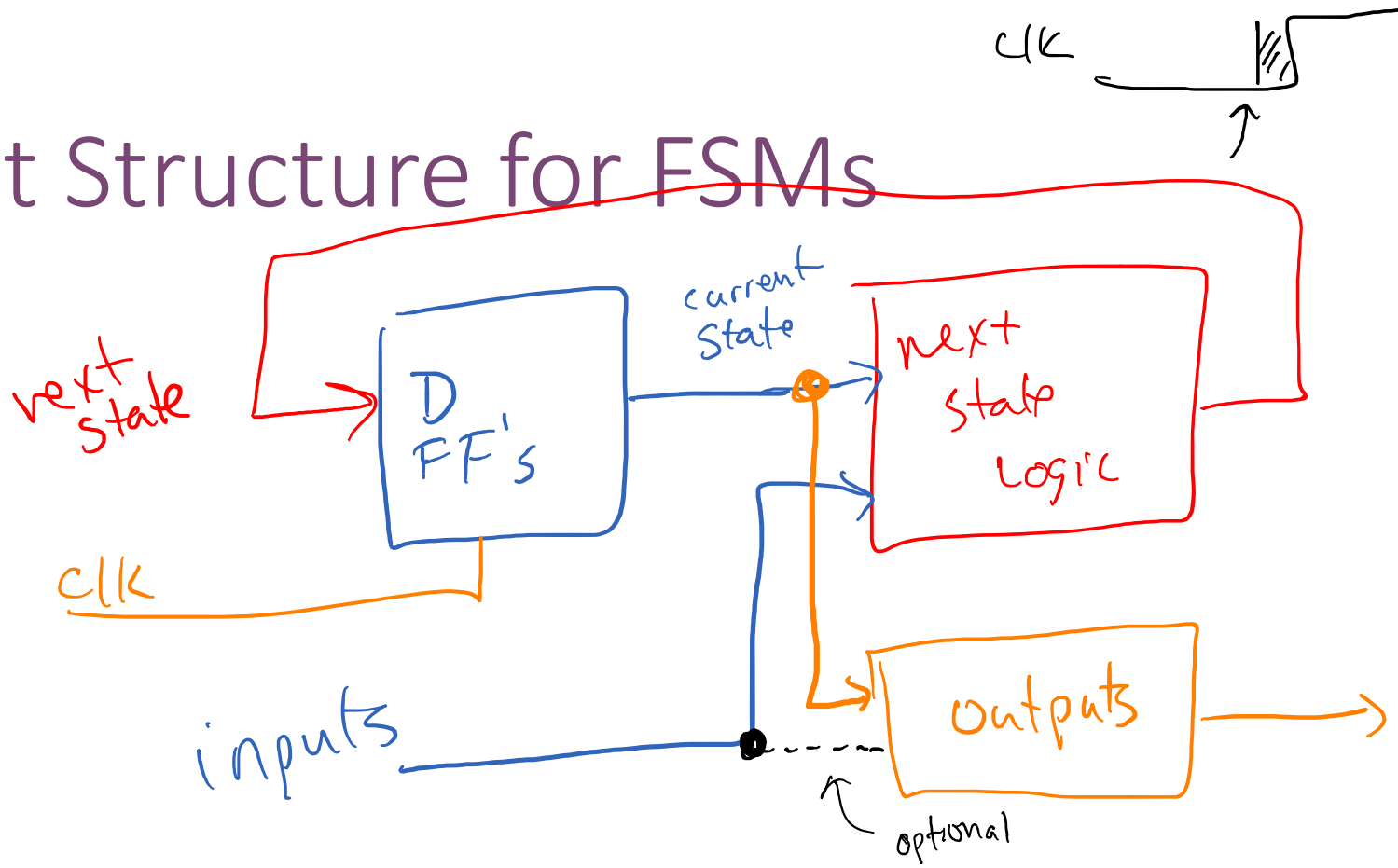
Next State Logic







Circuit Structure for FSMs

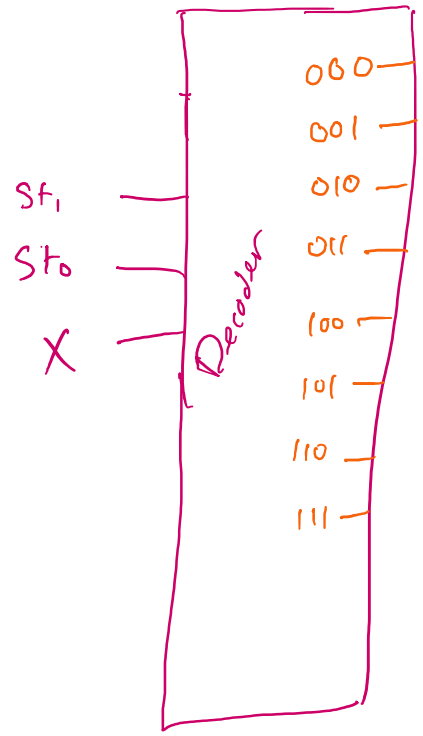


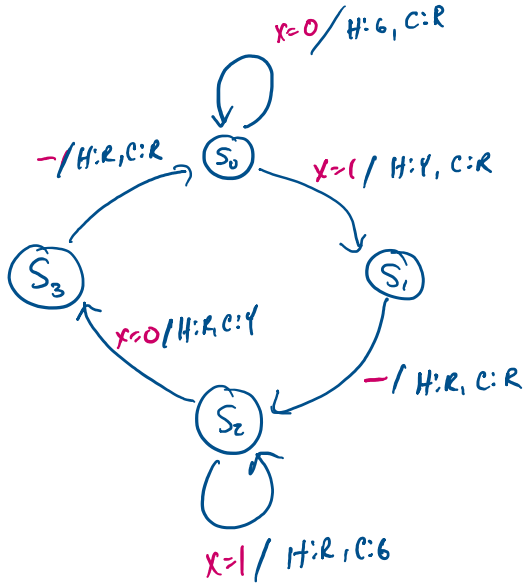
Moore Machine: outputs are a function of current state

Mealy Machine: outputs are a function of current state + inputs

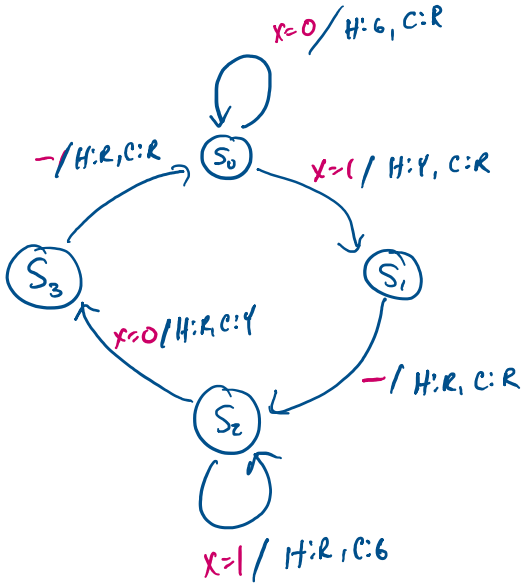
Output Logic (Highway)

	State	X
0	00	0
1	00	1
2	01	0
3	01	1
4	10	0
5	10	1
6	11	0
7	11	1

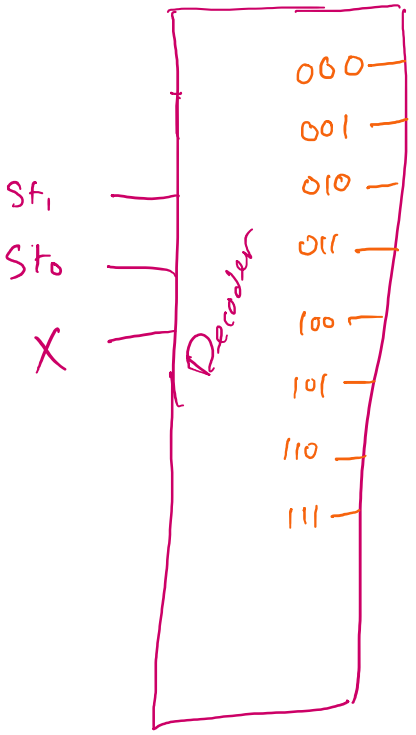




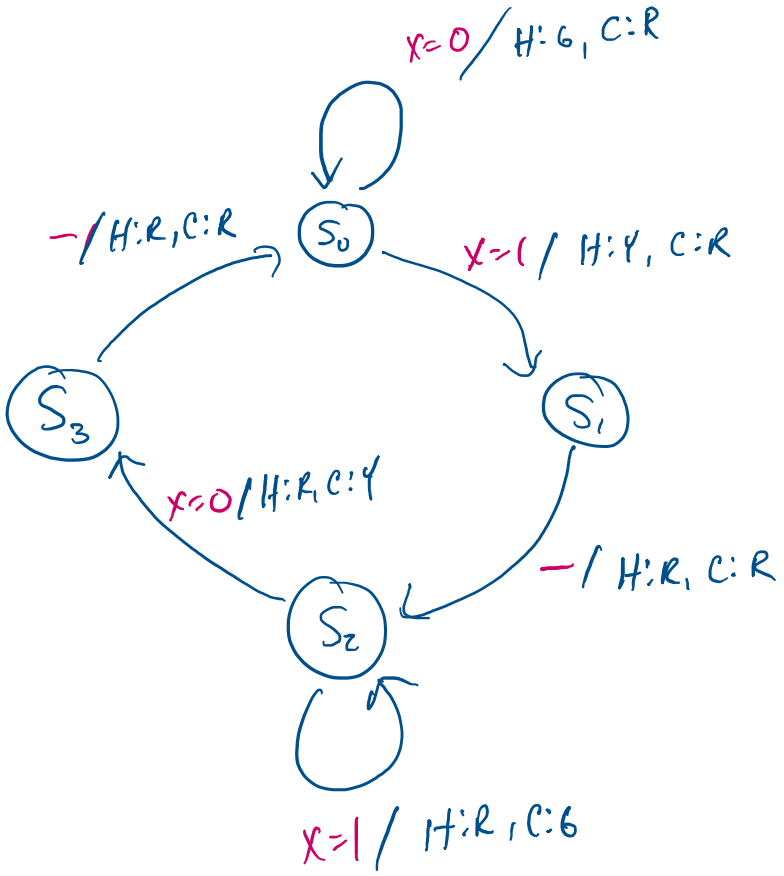
Output Logic (Country Rd)

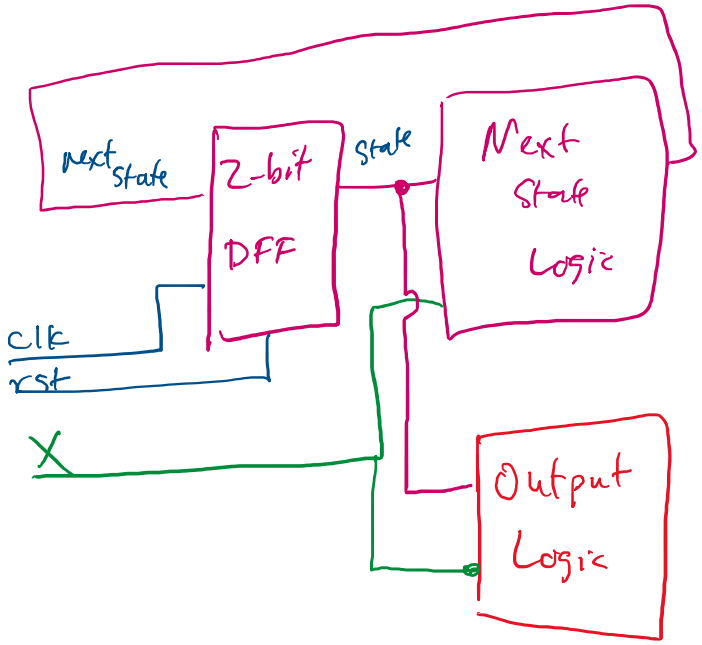


	State	X
0	00	0
1	00	1
2	01	0
3	01	1
4	10	0
5	10	1
6	11	0
7	11	1



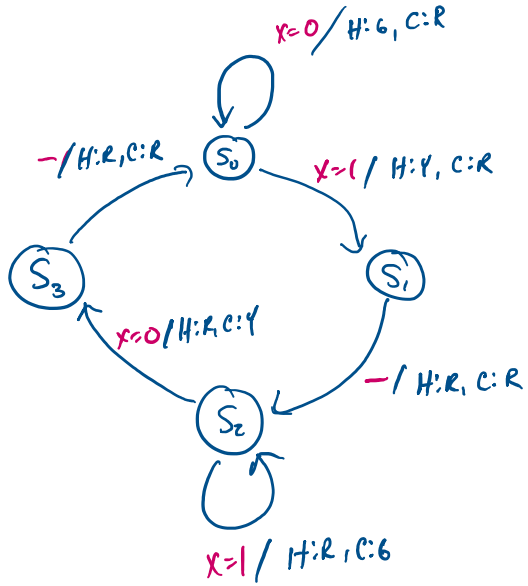
State Machine to Logic





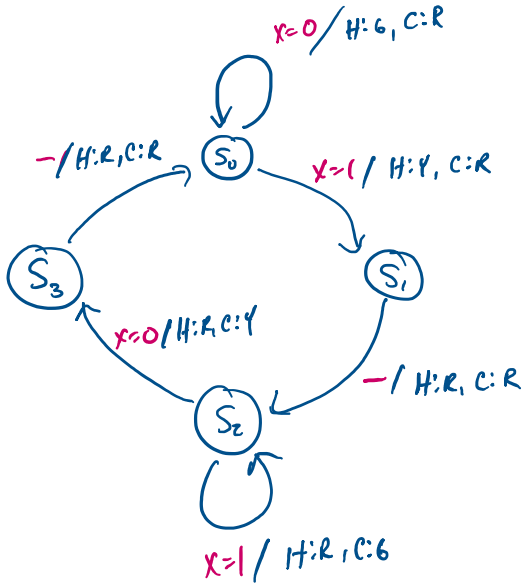
State Machine to Verilog

- Define states?



State Machine to Verilog

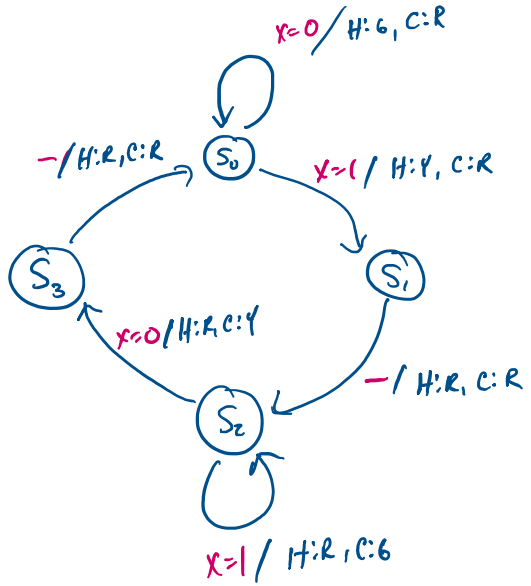
- Define states?



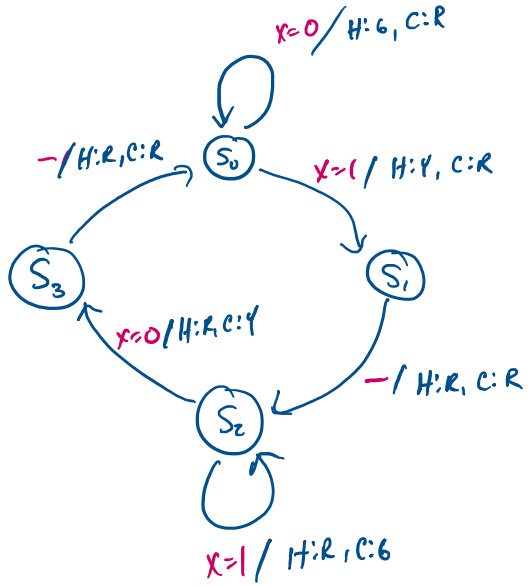

```
enum { ST_0, ST_1, ST_2, ST_3} state, nextState;
```

State Machine to Verilog

- Build State Machine?



State Machine to Verilog



- Build State Machine?

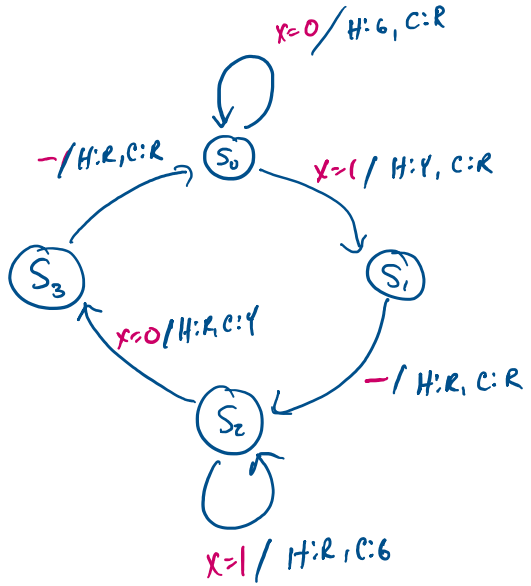
```
always_ff @(posedge clk) begin
    if (rst) state <= ST_0;
    else state <= nextState;
end
```

- What is `nextState`?

State Machine to Verilog

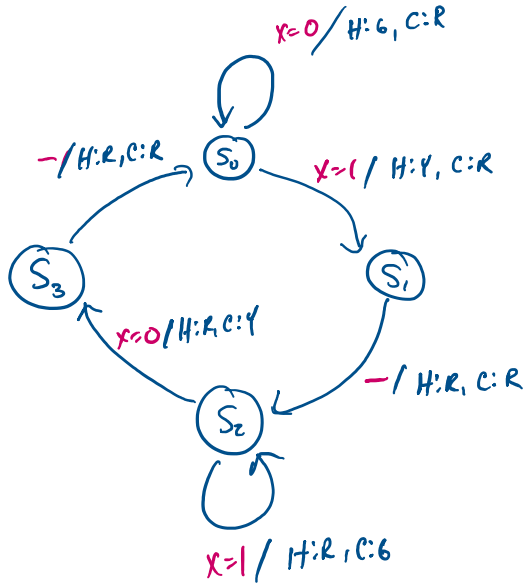
```
always_ff @(posedge clk) begin
    if (rst) state <= ST_0;
    else state <= nextState;
end
```

- What is `nextState`?

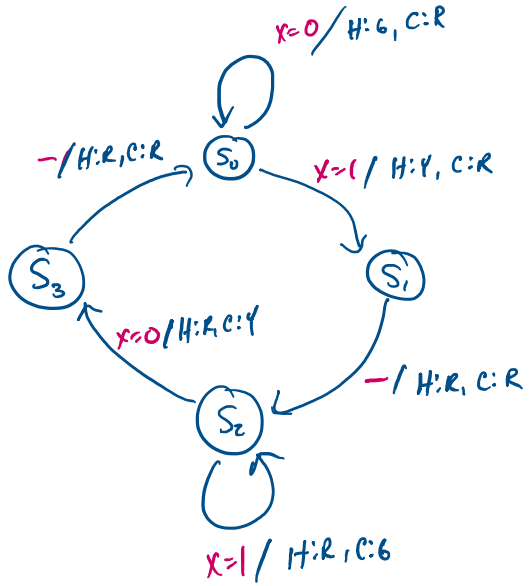


State Machine to Verilog

```
always_ff @(posedge clk) begin
    if (rst) state <= ST_0;
    else state <= nextState;
end

always_comb begin
        nextState = state; //default
        case(state)
                ST_0: nextState = ST_1; //goto state 1
                ST_1: nextState = ST_2;
                ST_2: nextState = ST_3;
                ST_3: nextState = ST_0; //loop
                default: nextState = ST_0; //just in case
        endcase
end
```



State Machine to Verilog



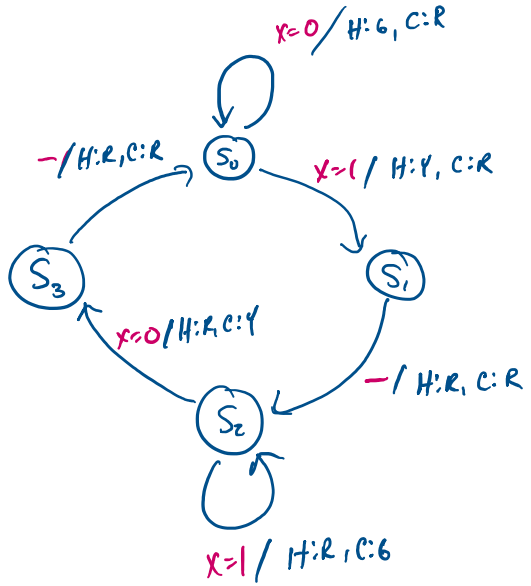
- What is this missing?

```
always_comb begin
        nextState = state; //default
        case(state)
                ST_0: nextState = ST_1; //goto state 1
                ST_1: nextState = ST_2;
                ST_2: nextState = ST_3;
                ST_3: nextState = ST_0; //loop
                default: nextState = ST_0; //just in case
        endcase
end
```

State Machine to Verilog

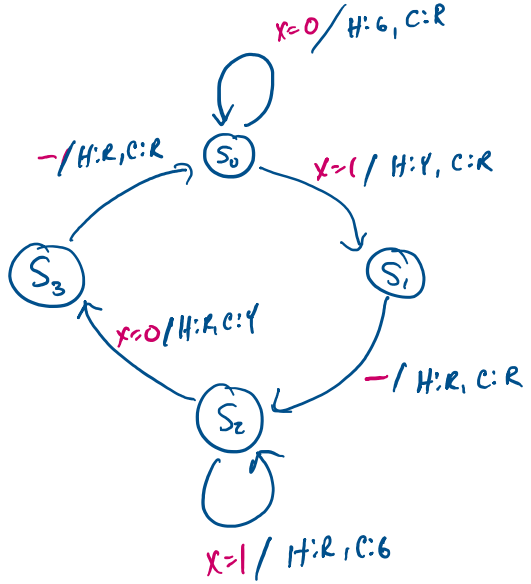
- What is this missing?

```
always_comb begin
        nextState = state; //default
        case(state)
                ST_0:
                        nextState = ST_1;
                ST_1:
                        nextState = ST_2;
                ST_2:
                        nextState = ST_3;
                // ST_3 and default cases   endcase
end
```



State Machine to Verilog

- What is this missing?

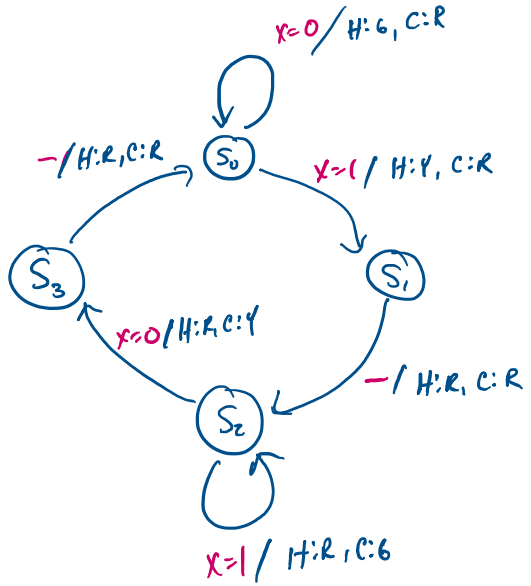



```
always_comb begin
        nextState = state; //default
        case(state)
                ST_0:
                        if (X) nextState = ST_1;
                ST_1:
                        nextState = ST_2;
                ST_2:
                        if (~X) nextState = ST_3;
                // ST_3 and default cases   endcase
end
```

State Machine to Verilog

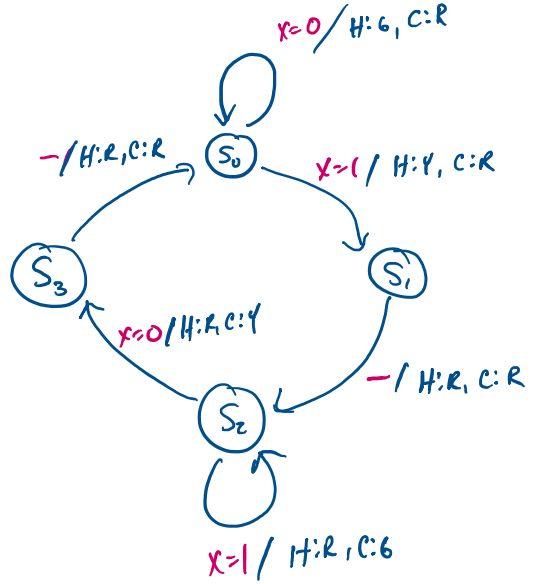
- What <u>else</u> is this missing?

```
always_comb begin
        nextState = state; //default

        case(state)
                ST_0:
                        if (X) nextState = ST_1;



                // ST_1-3 and default cases
        endcase
end
```

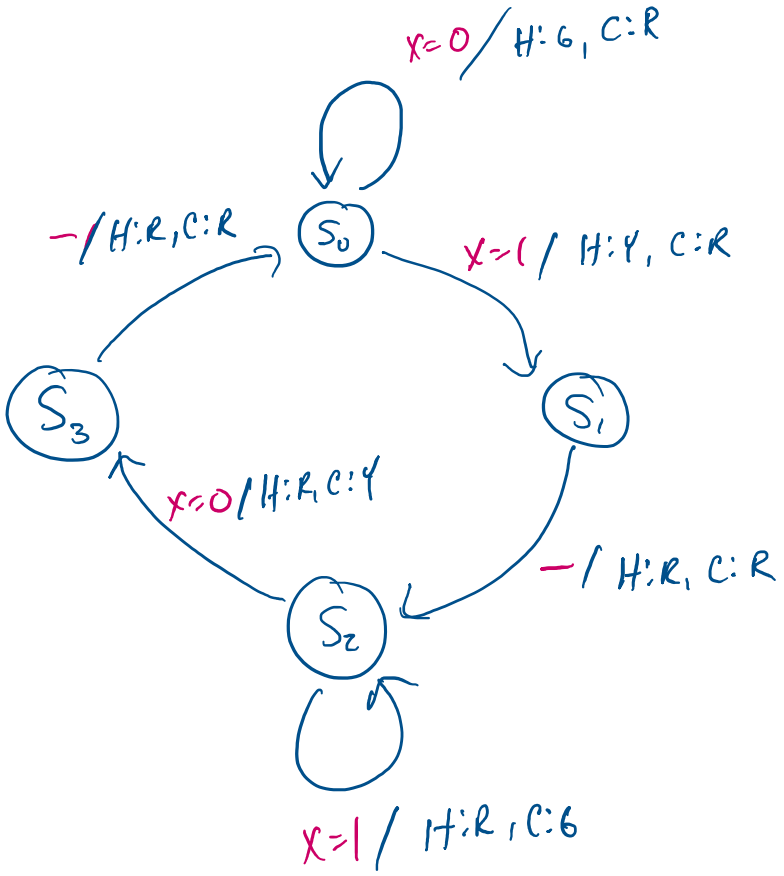


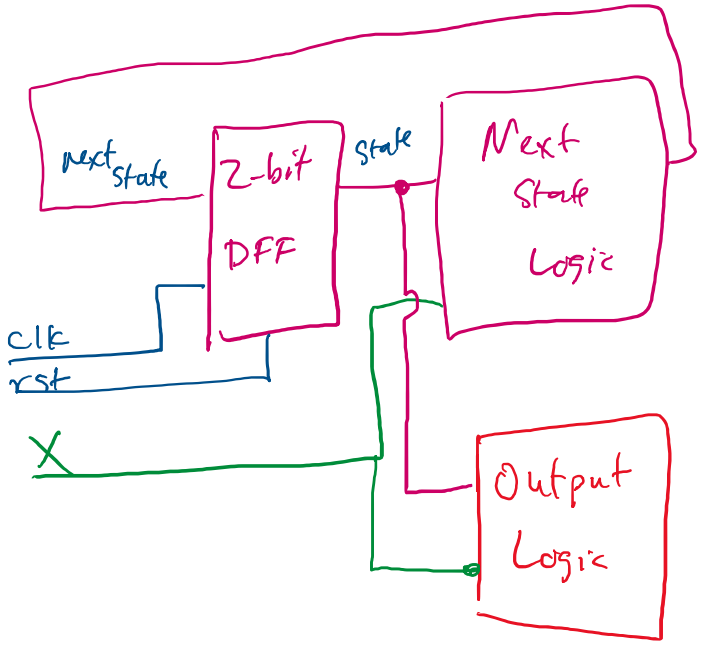
State Machine to Verilog

```
always_comb begin
        nextState = state; //default
        Hryg = {0,0,1}; Cryg={1,0,0};
        case(state)
                ST_0: begin
                        if (X) begin
                                nextState = ST_1;
                                Hryg = {0,1,0};
                                Cryg = {1,0,0); //optional
                        end else begin
                                nextState = ST_0; //optional
                                Hryg = {0,0,1}; //optional
                                Cryg = {1,0,0); //optional
                        end
                end
                // ST_1-3 and default cases
        endcase
end
```

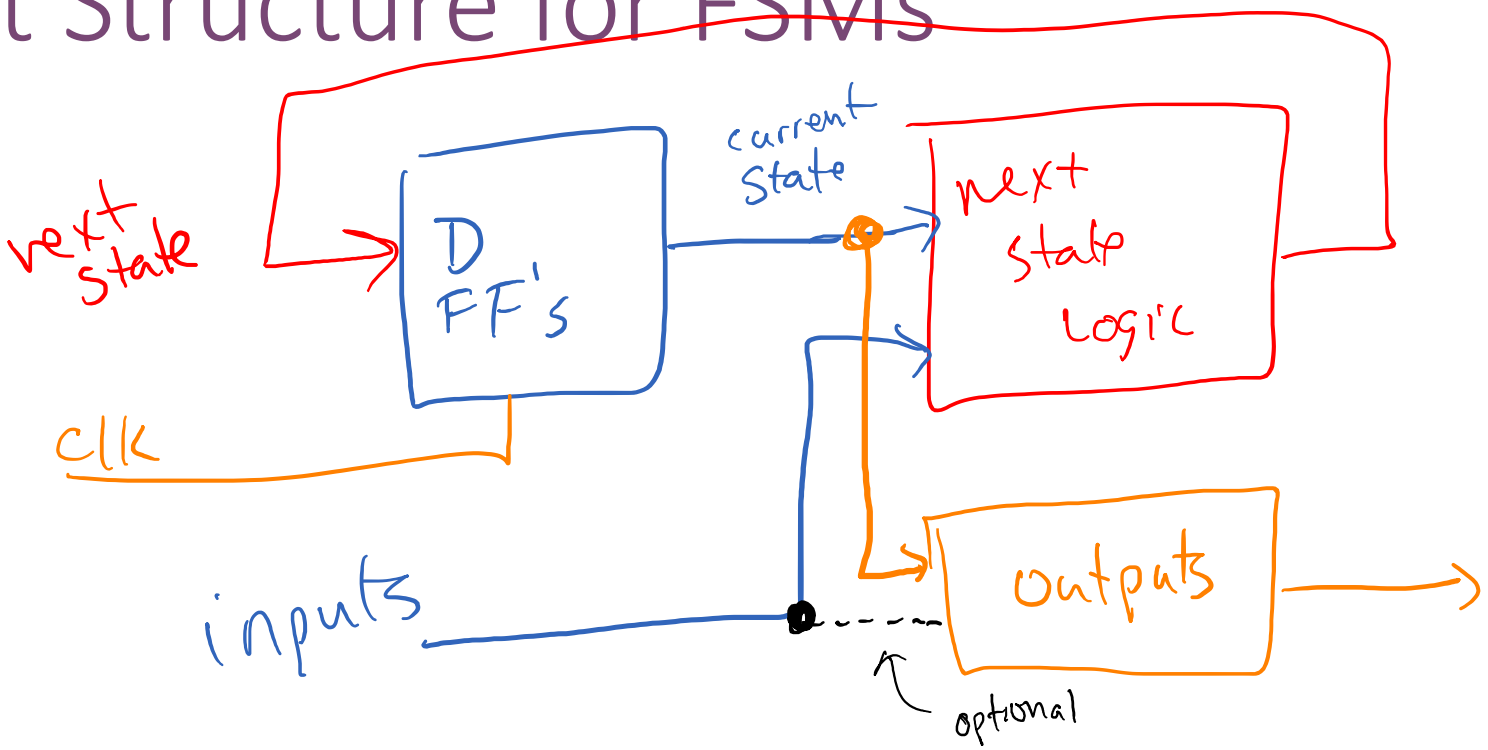


State Machine in Logic





Circuit Structure for FSMs



Moore Machine: outputs are a function of current state

Mealy Machine: outputs are a function of current state + inputs

Next Time

- More Finite State Machines (FSMs)