

# Finite State Machines I

Andrew Lukefahr

# DFF with Synchronous Reset

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

    always_ff @(posedge clk)
    begin
        if (rst) q <= 'h0; //reset case
        else     q <= d; //non-reset case
    end

endmodule
```

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**

# What's wrong here?

```
logic [1:0] count;

always_comb begin
    count = count; //default

    if (enable && up_down) begin
        count = count + 1;
    end else if (enable && ~up_down) begin
        count = count - 1;
    end
end
```

# What's wrong here?

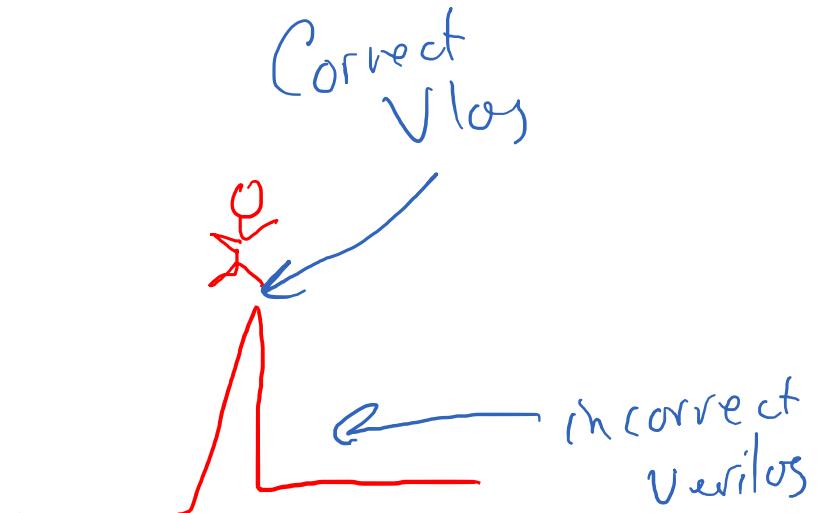
```
logic [1:0] count;

always_comb begin
    count = count; //not a default

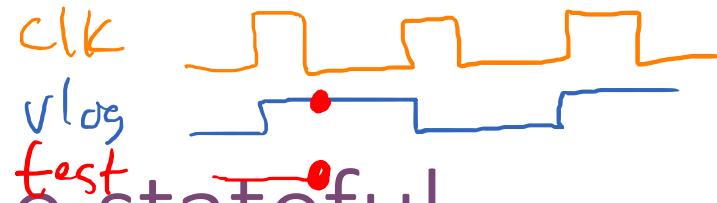
    if (enable && up_down) begin
        count = count + 1; //self reference
    end else if (enable && ~up_down) begin
        count = count - 1; //self reference
    end
end
```

# 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 (enable && up_down) begin  
        nextCount = count + 1;  
    end else if (enable && ~up_down) begin  
        nextCount = count - 1;  
    end  
end
```

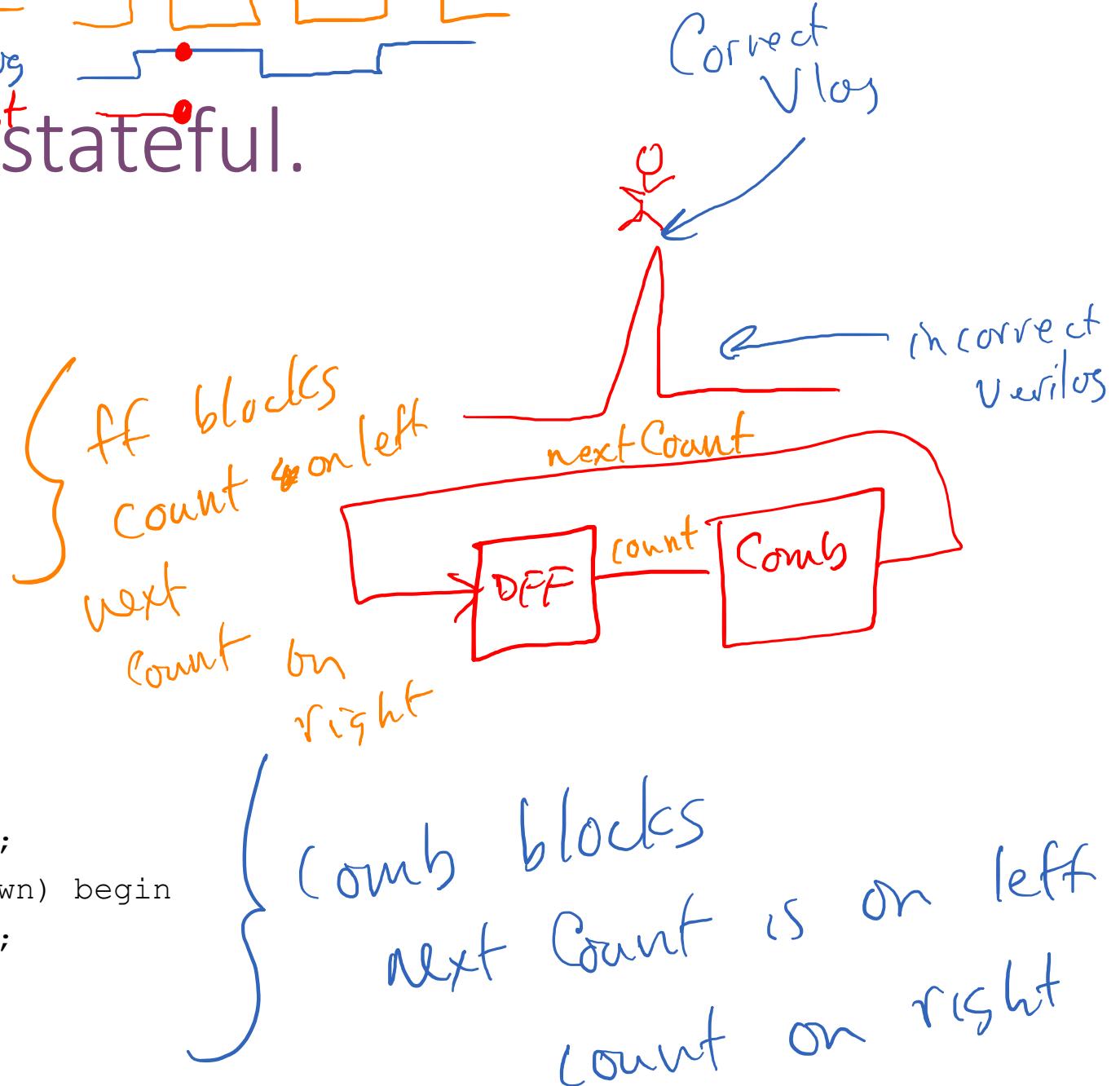


negedge  $\Rightarrow$  test.benches

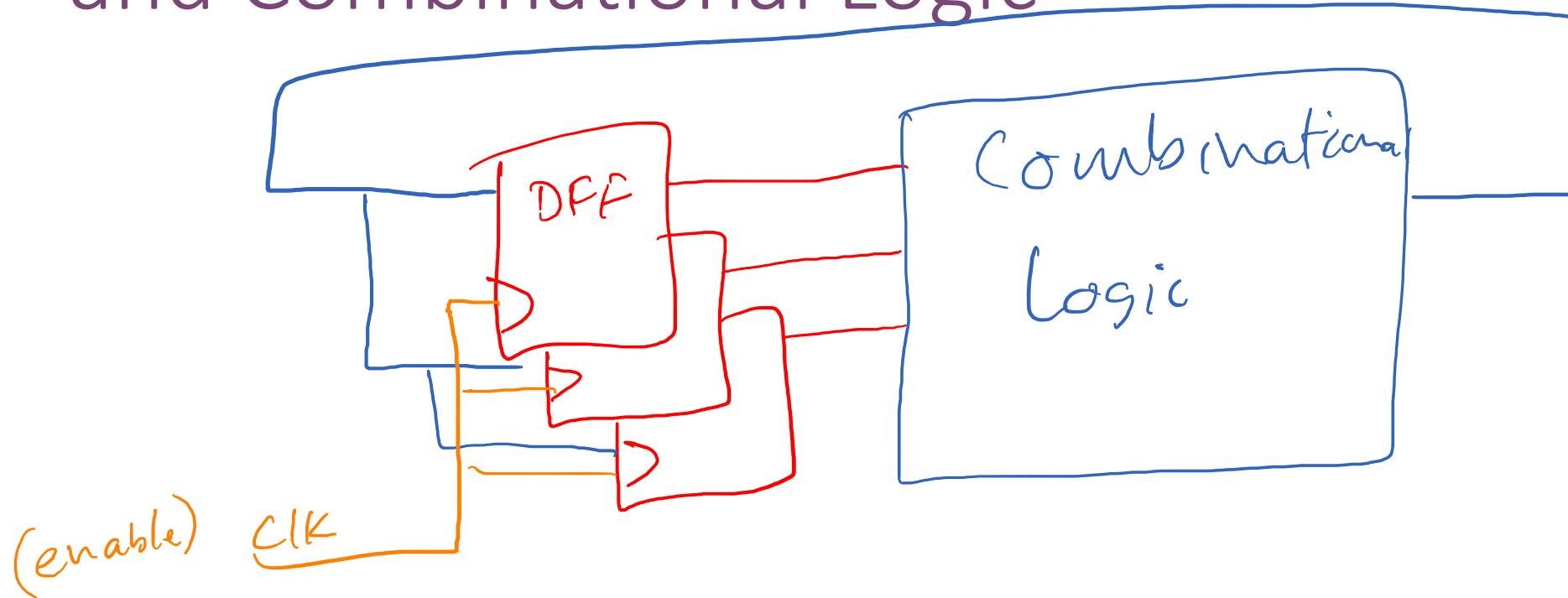


Count needs to be stateful.

```
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 (enable && up_down) begin  
        nextCount = count + 1;  
    end else if (enable && ~up_down) begin  
        nextCount = count - 1;  
    end  
end
```



# Sequential Logic uses both Flip-Flops and Combinational Logic



# Shift-Register in Verilog

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

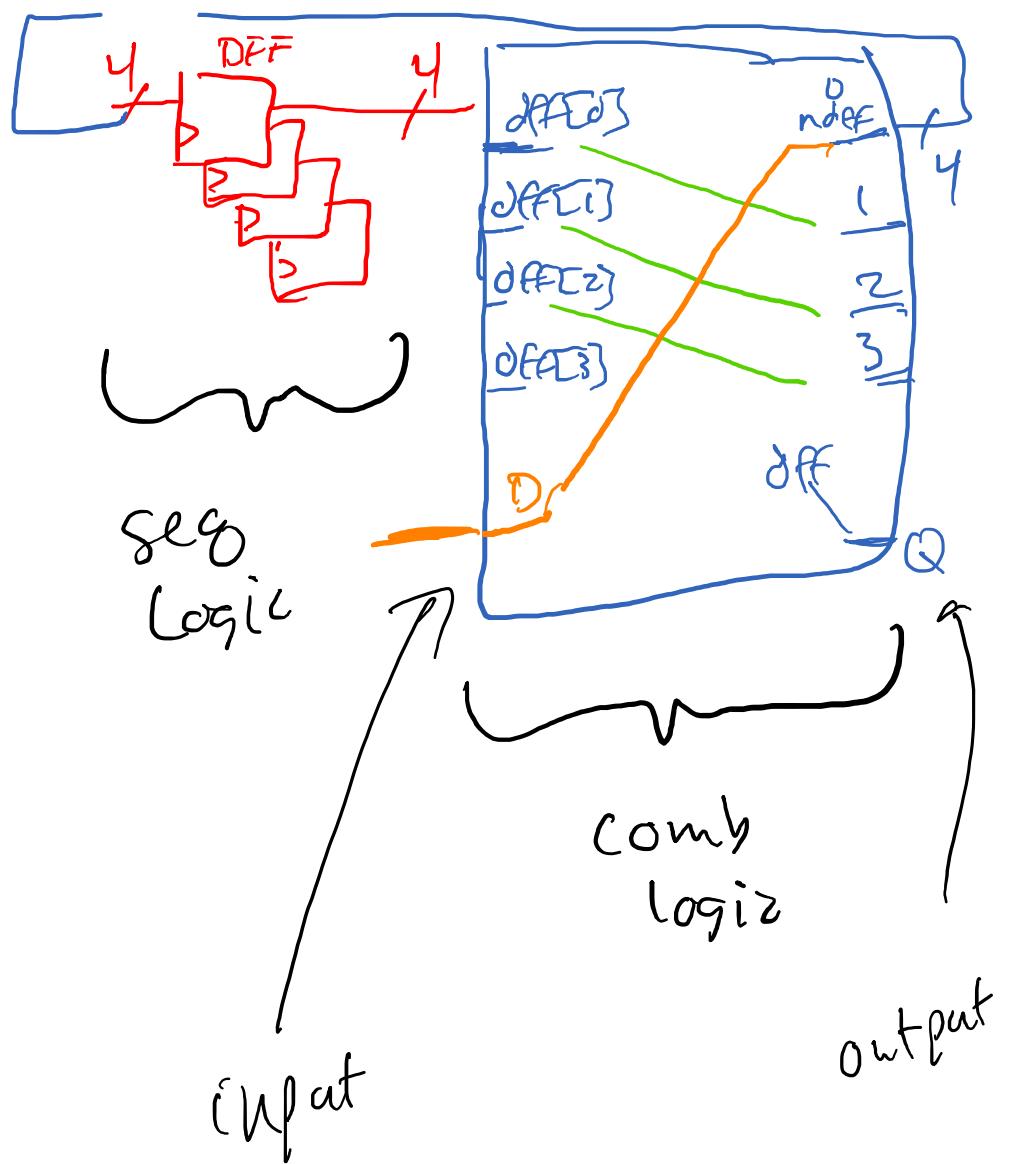
    logic [3:0] dff;
    logic [3:0] next_dff;

    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
```



```
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
```

Can I make a  
'fast'  
counter?

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

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

    always_comb begin
        sum = Q + 4'h1;
        out = sum[3];
    end

endmodule
```

<- Inputs can affect  
output or next state

```
module counter(  
    input clk, rst  
    input          out_fast, //faster output  
    output logic   out    //output  
) ;  
logic [3:0] Q;  
logic [3:0] sum;  
  
always_ff @(posedge clk) begin  
    if (rst) Q <= 4'h0;  
    else      Q <= sum;  
end  
  
always_comb begin  
    sum = Q + 4'h1;  
    out = sum[3];  
end  
  
endmodule
```

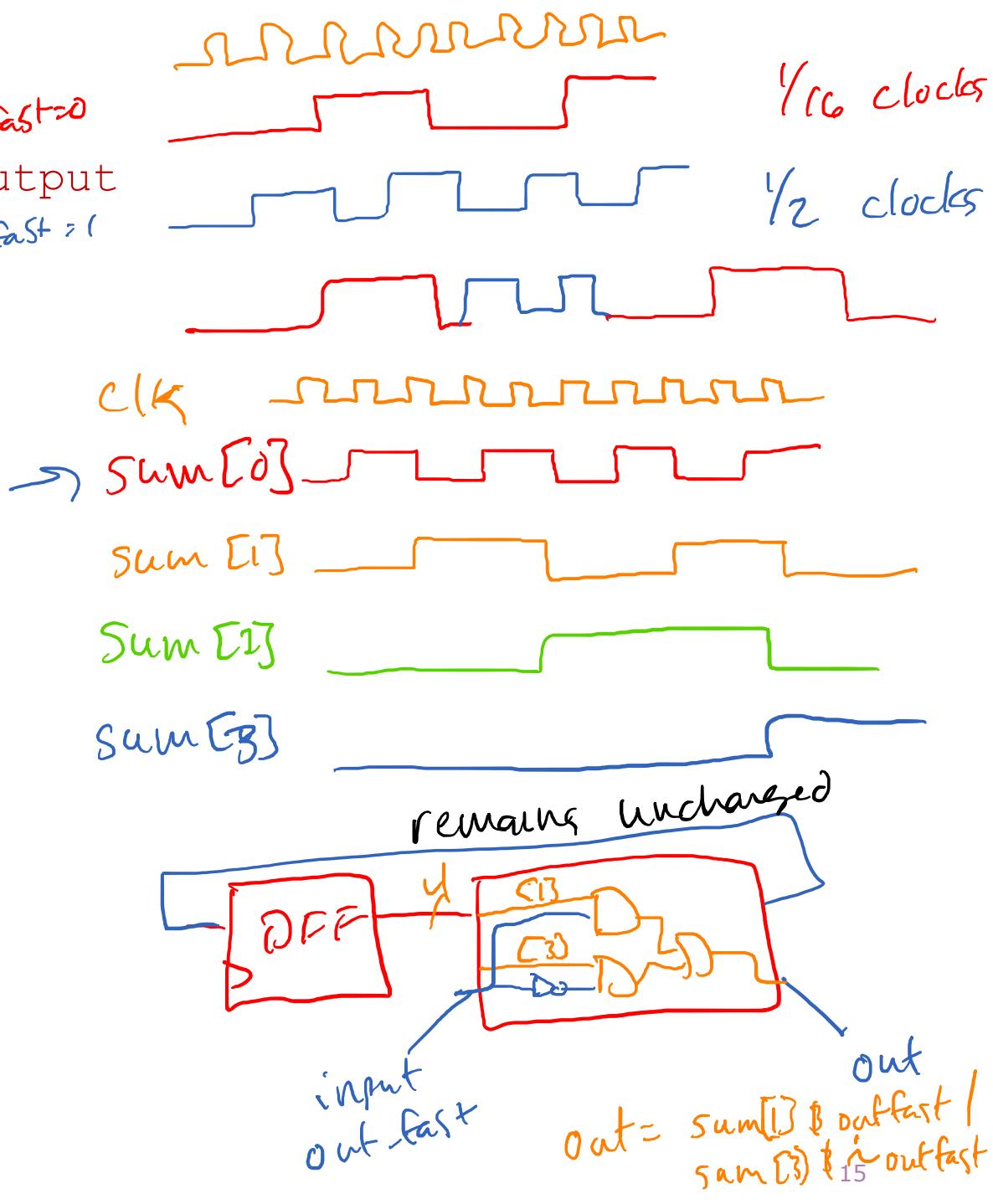
```

module counter(
    input clk, rst
    input          out_fast, //faster output
    output logic   out    //output      out_fast=0
);
    logic [3:0] Q;
    logic [3:0] sum;

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

    always_comb begin
        sum = Q + 4'h1;
        ✓ out = sum[3]; //default
        ✓ if(out_fast) out = sum[i];
    end
endmodule

```



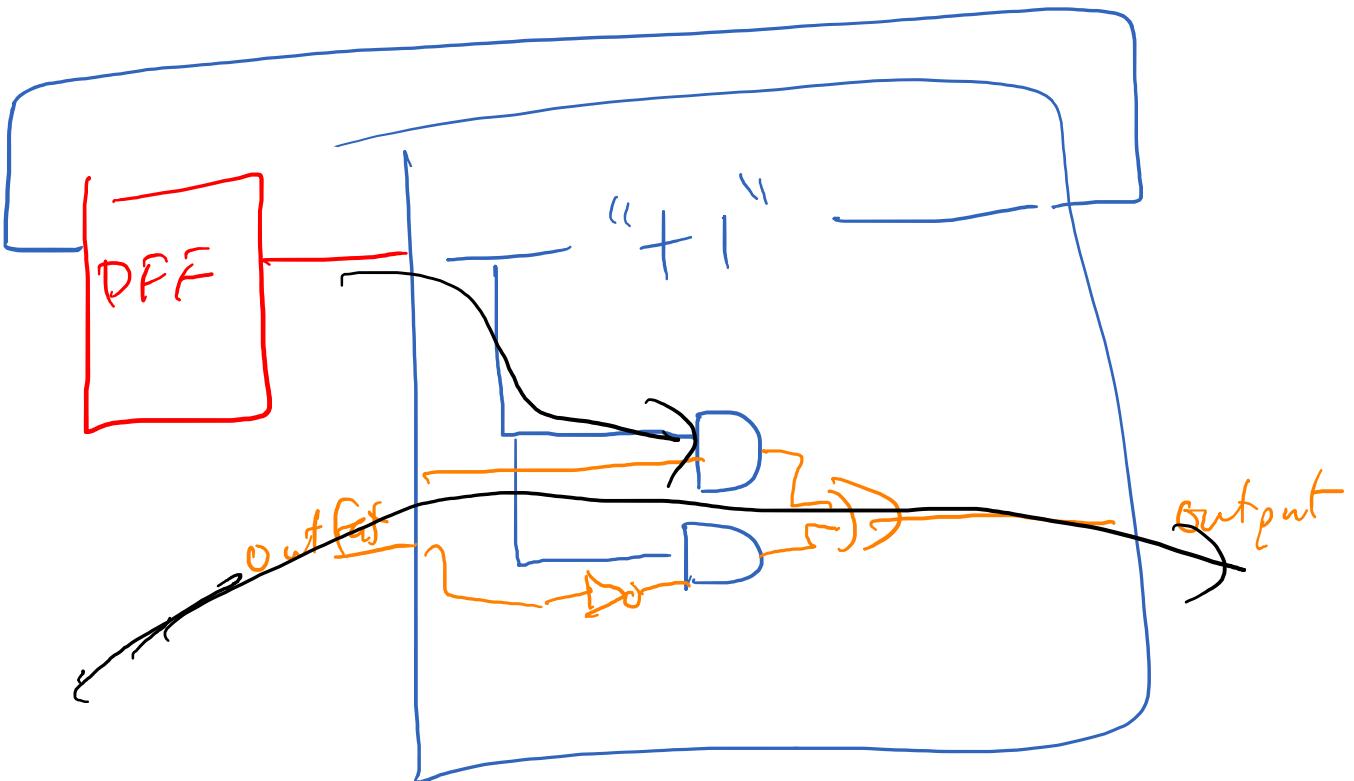
```

module counter(
    input clk, rst
    input          out_fast, //faster output
    output logic    out     //output
);
    logic [3:0] Q;
    logic [4:0] sum;

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

    always_comb begin
        sum = Q + 4'h1;
        out = sum[3];
        if (out_fast) out = sum[2];
    end
endmodule

```



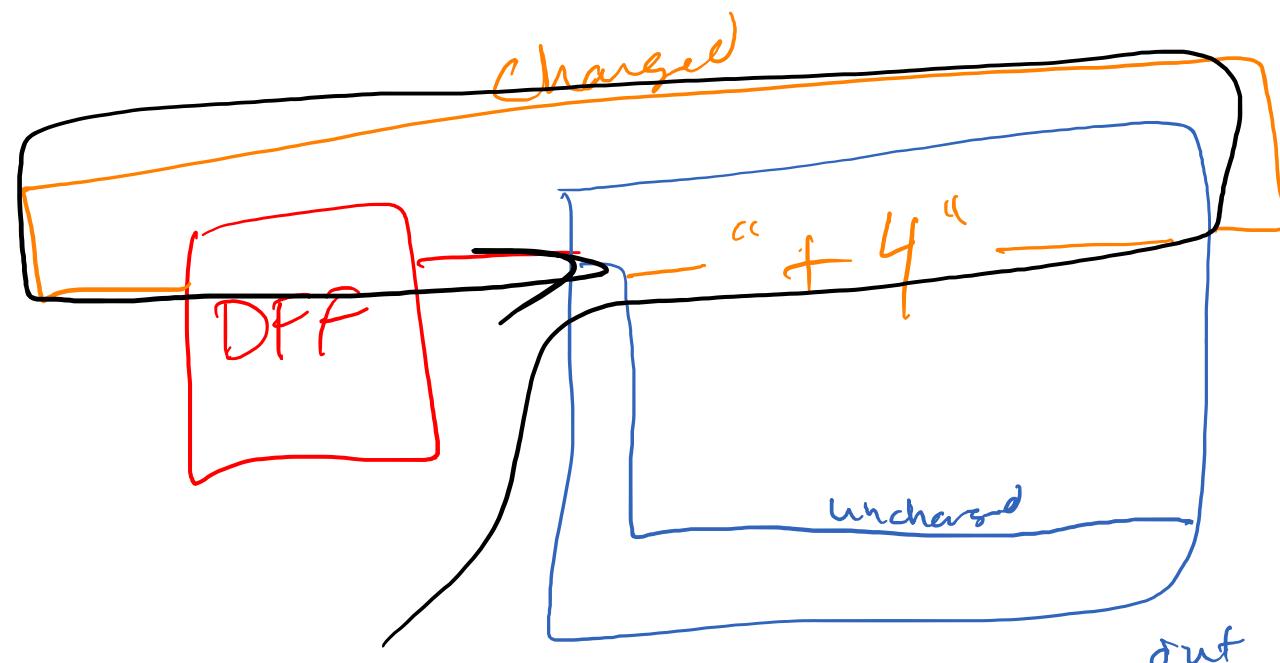
```

module counter(
    input clk, rst
    input          out_fast, //faster output
    output logic    out     //output
);
    logic [3:0] Q;
    logic [4:0] sum;

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

    always_comb begin
        sum = Q + 4'h1;
        out = sum[3];
        if (out_fast) sum = {0,Q} + 5'h4;
    end
endmodule

```



FPGA code → @ posedge clk

Testbench  
(code) → @ negedge clk  
(Always ends in - tb.sv)

→ Finite State Machines (FSMs)

State Machines

(FSMs)

Comp Sci  
Finite State Automata

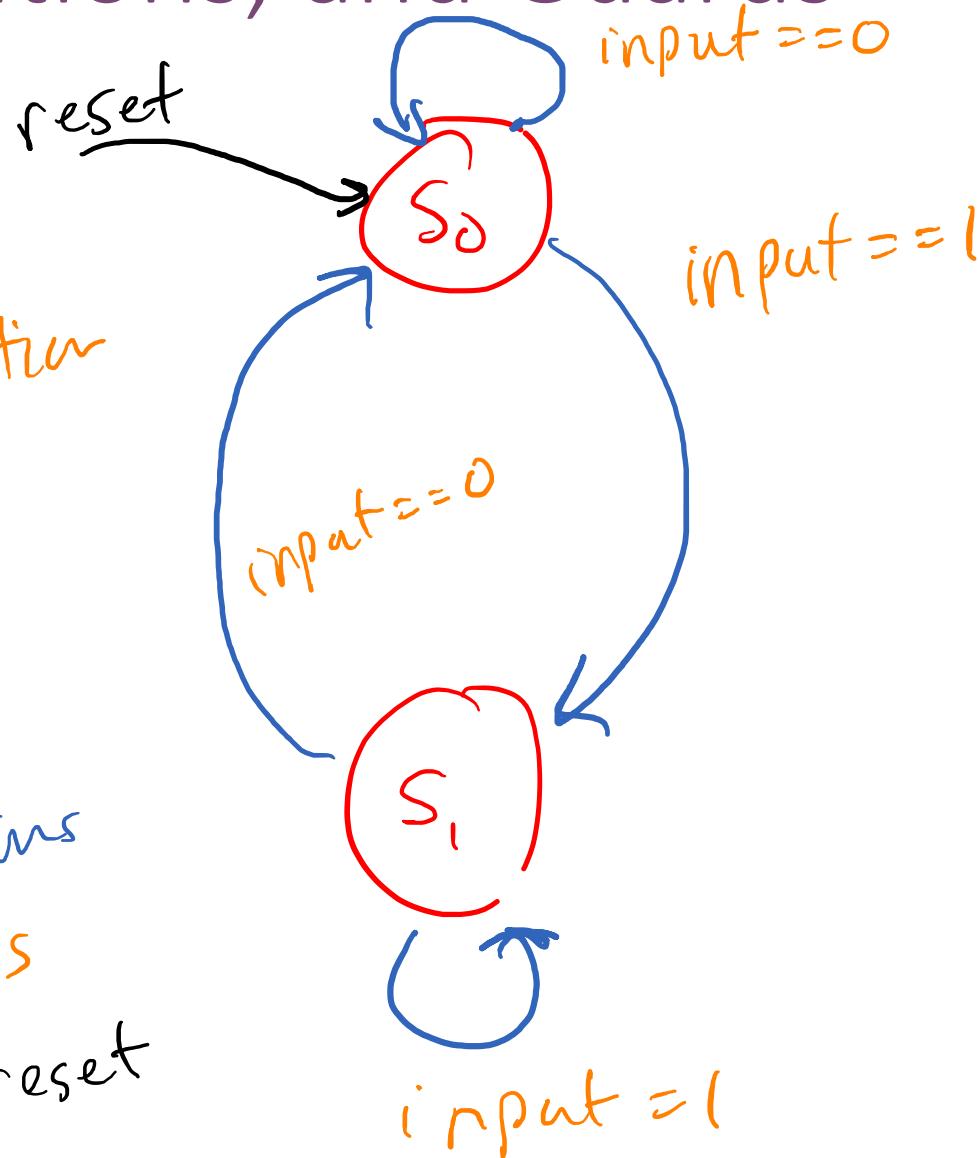
# Finite State Machines (FSMs)

- 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

guards: control which transition you take

~~$S_0 \& S_1 \Rightarrow$~~  states  
~~lives  $\Rightarrow$~~  transitions  
booleans  $\Rightarrow$  guards  
reset  $\Rightarrow$  reset



$S_0 \& S_1$  are states

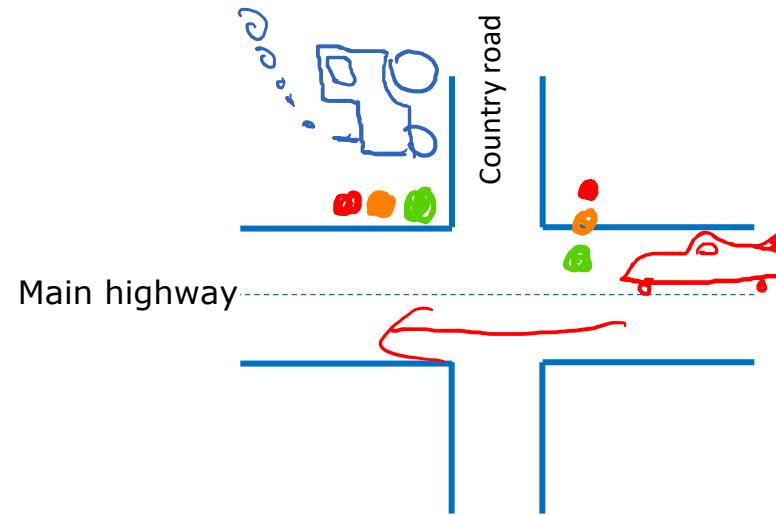
two states in this machine

transitions

$\rightarrow$  leaving one state & going to another  
(happen @posede c(k))

# 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

~~tractors~~

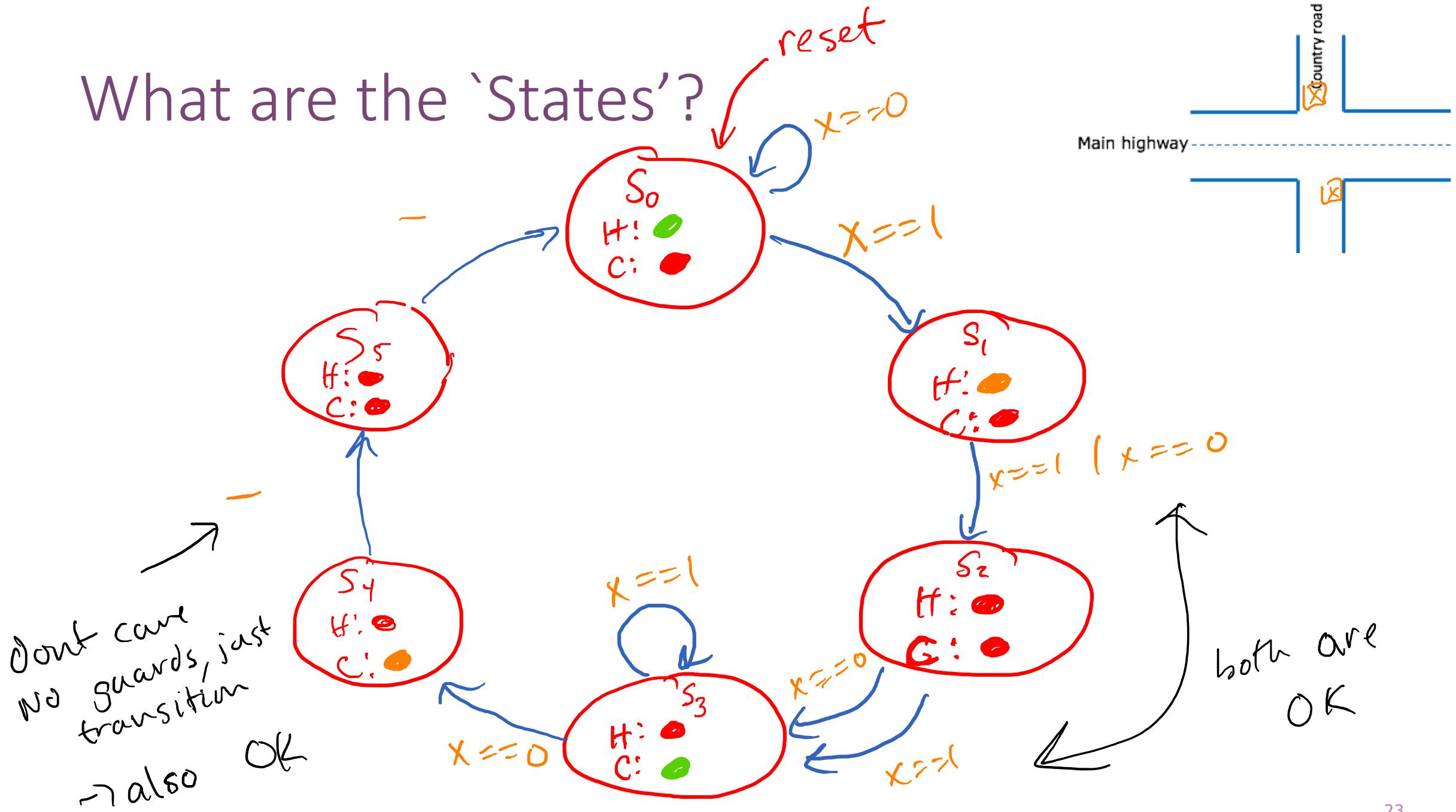
- ~~Cars~~ occasionally arrive from the country road. The traffic signal for the country road must turn **green** only long enough to let the cars on the country road go.
- When no cars are waiting on the country road, the country road traffic signal turns **yellow** then **red** and the traffic signal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

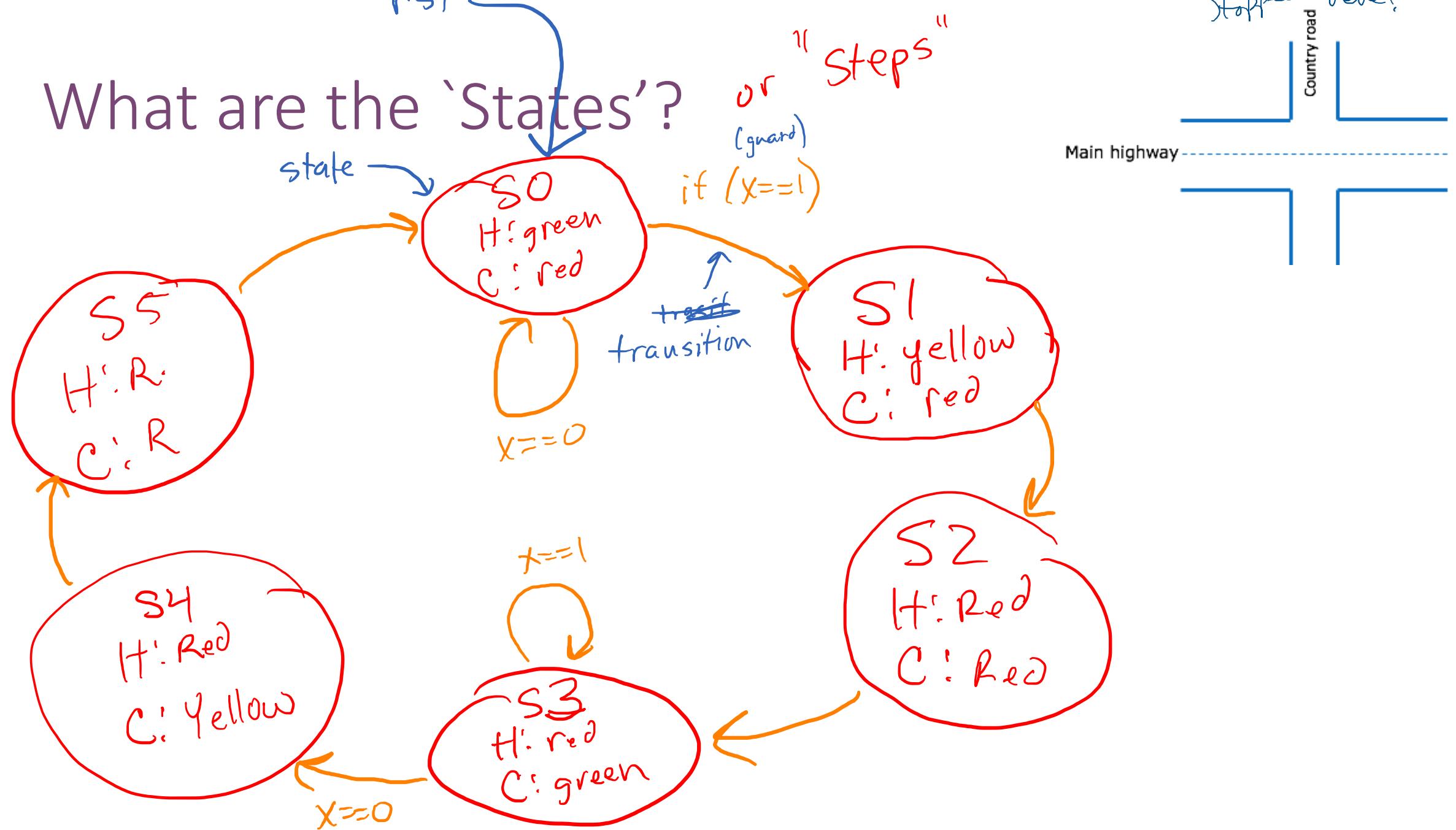
$X = 1$ , if there are cars on the country road

$X = 0$ , otherwise

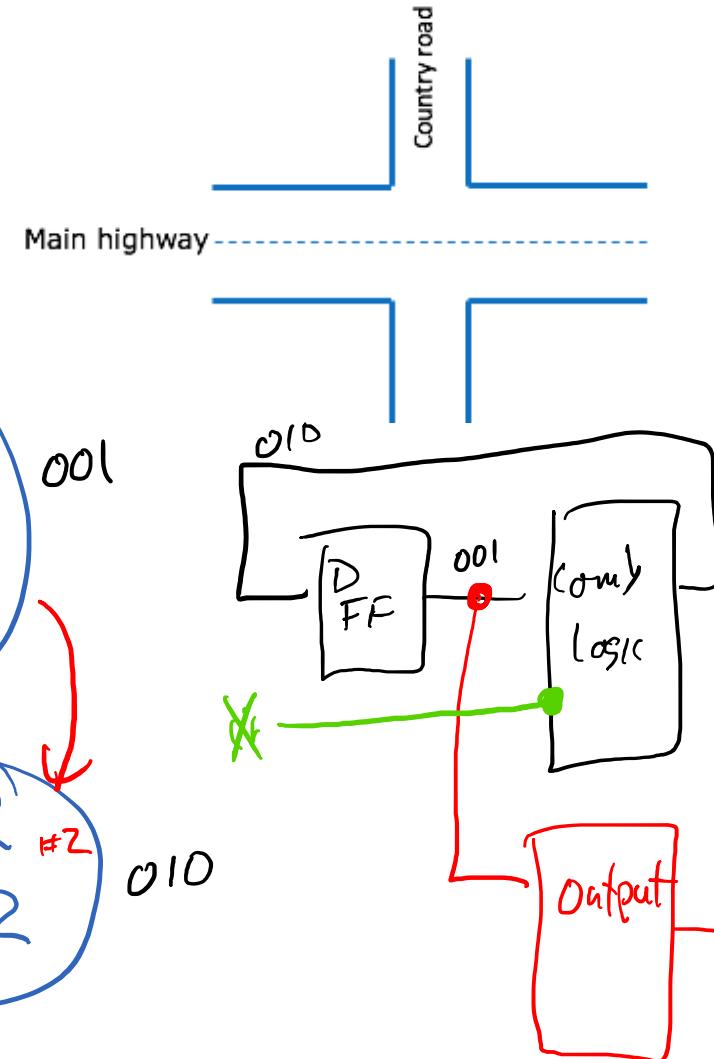
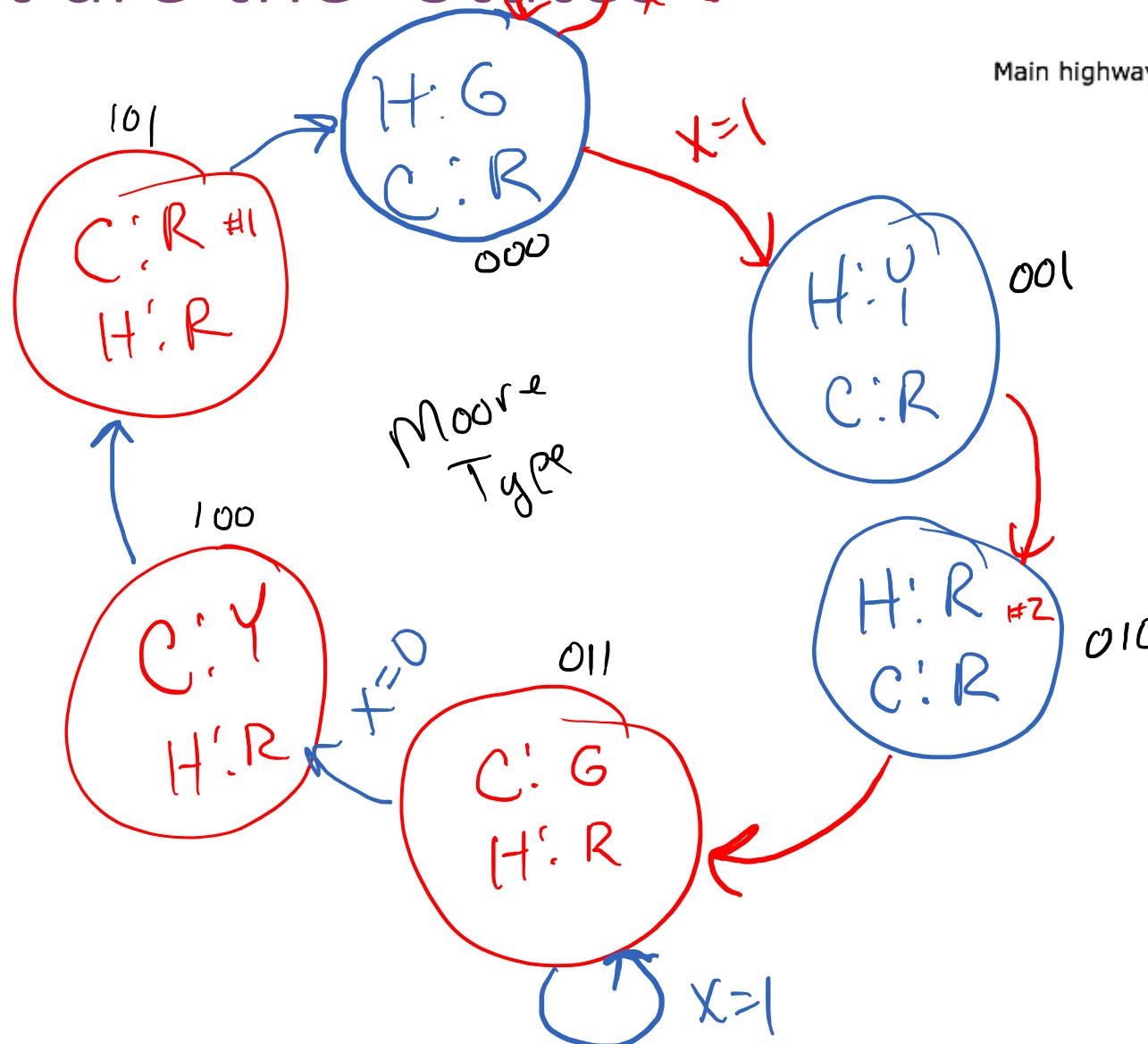
# What are the 'States'?



# What are the 'States'?

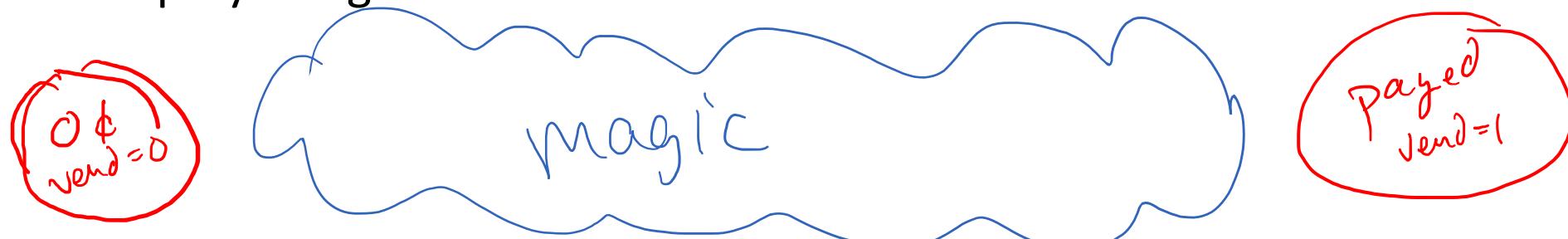


# What are the 'States'?



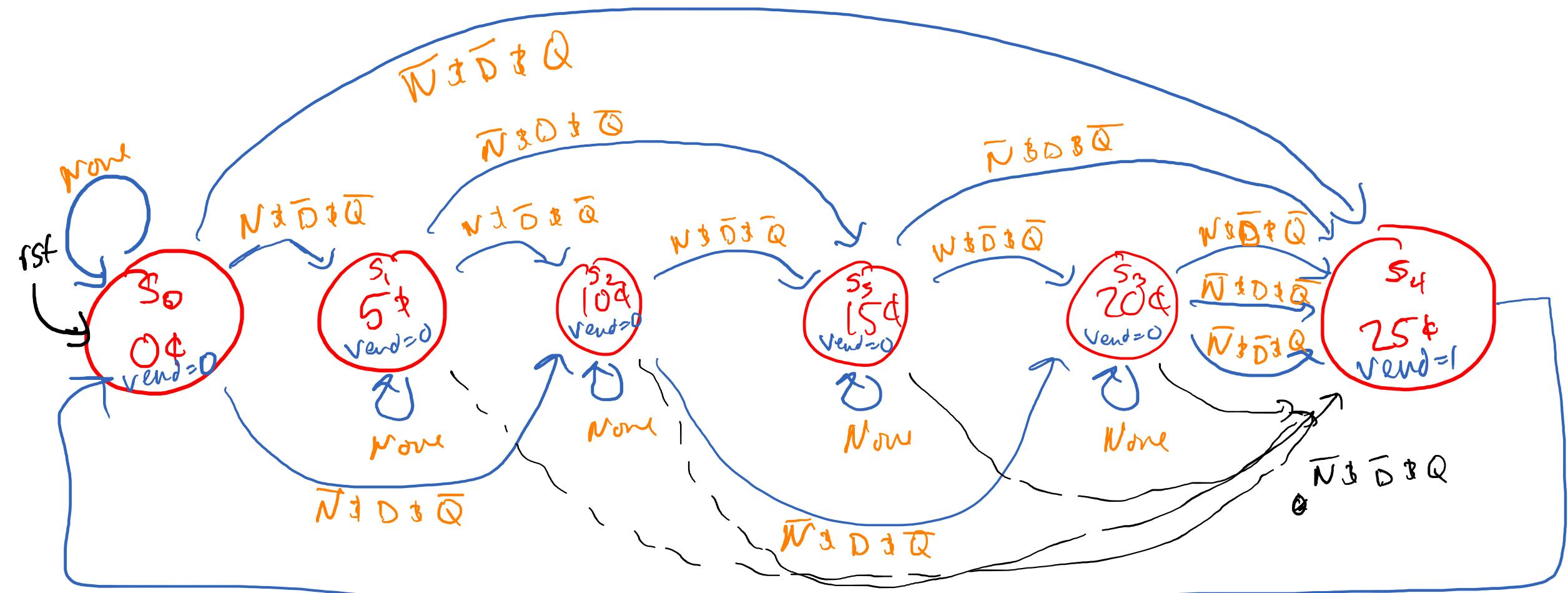
# FSM: Simple Vending Machine

- 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...



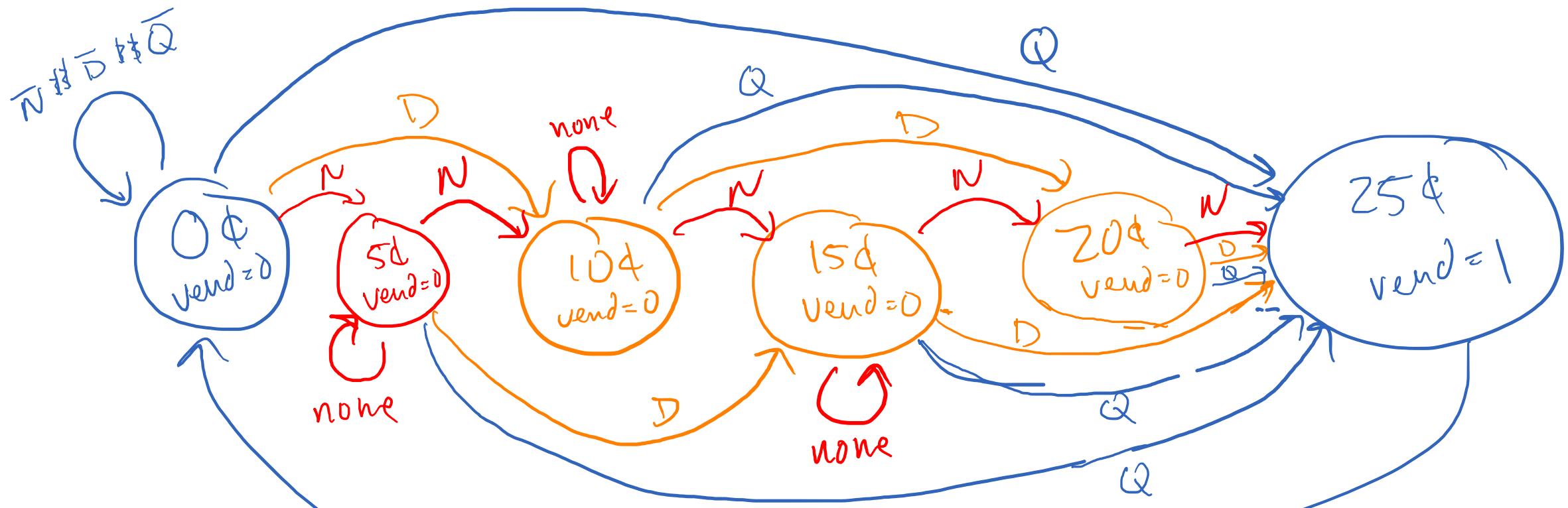
# FSM: Vending Machine

Nickles : N  
Dimes : D  
Quarters : Q



$$\text{None} = \bar{N} \bar{D} \bar{Q}$$

# FSM: Simple Vending Machine

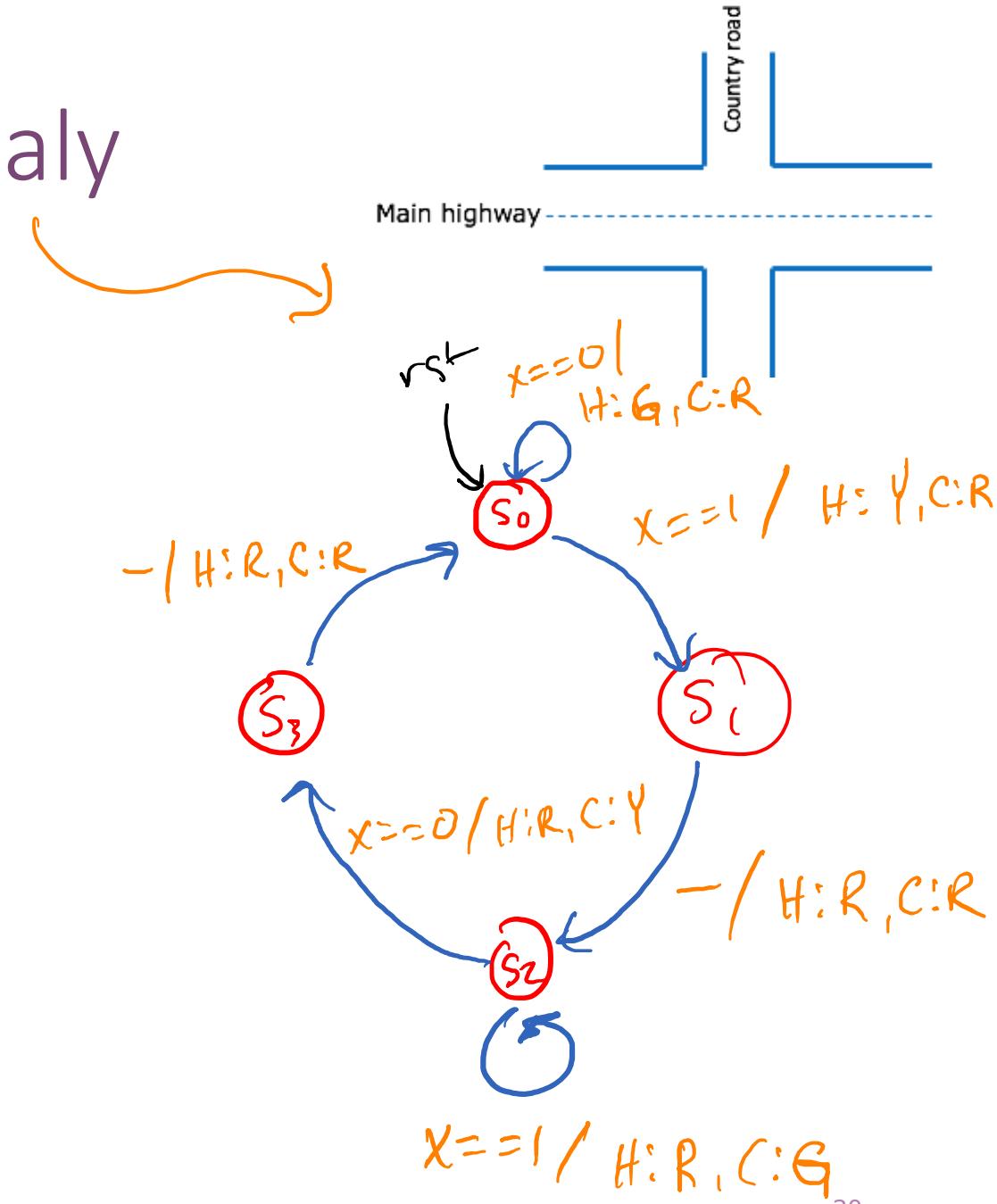
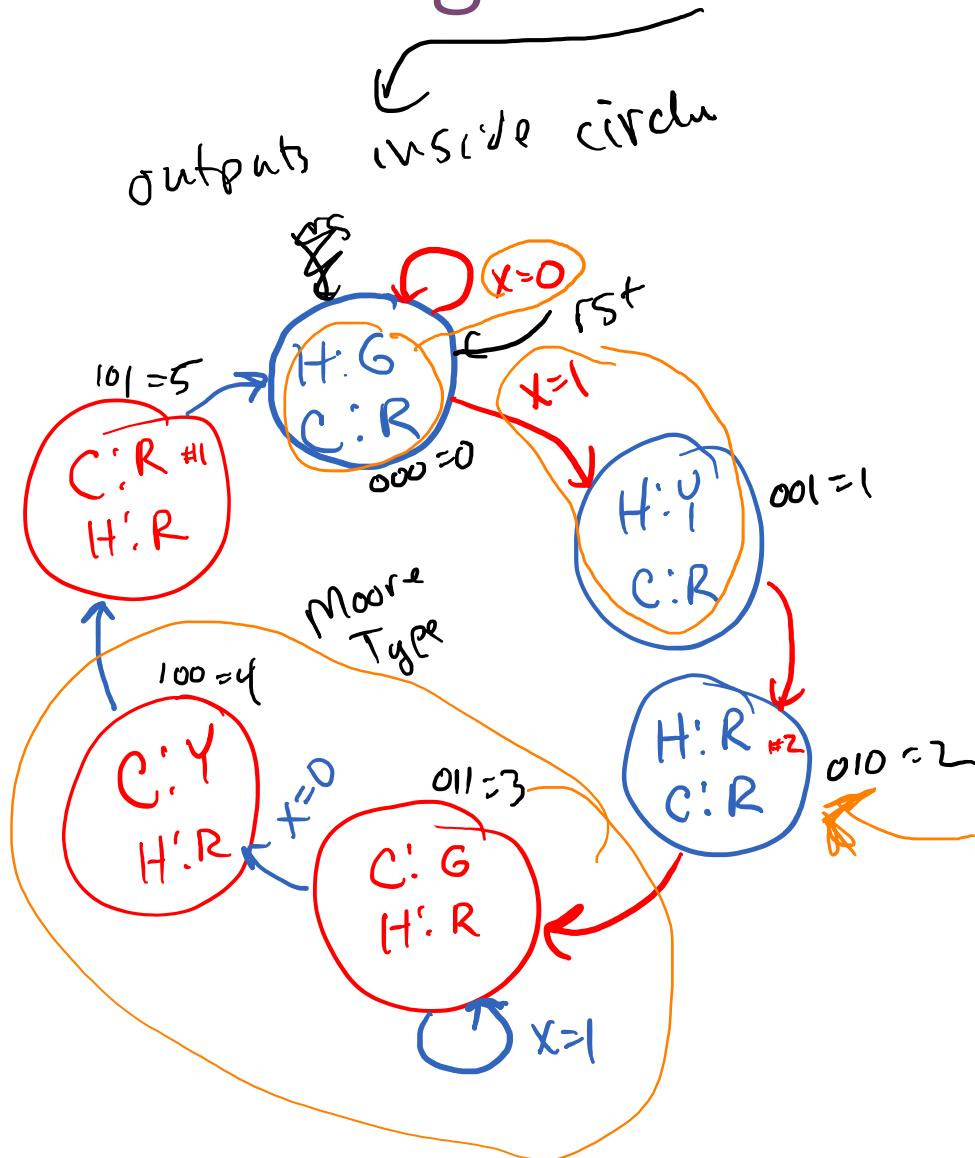


$$\text{none} = \overline{N} \ \overline{D} \ \overline{Q}$$

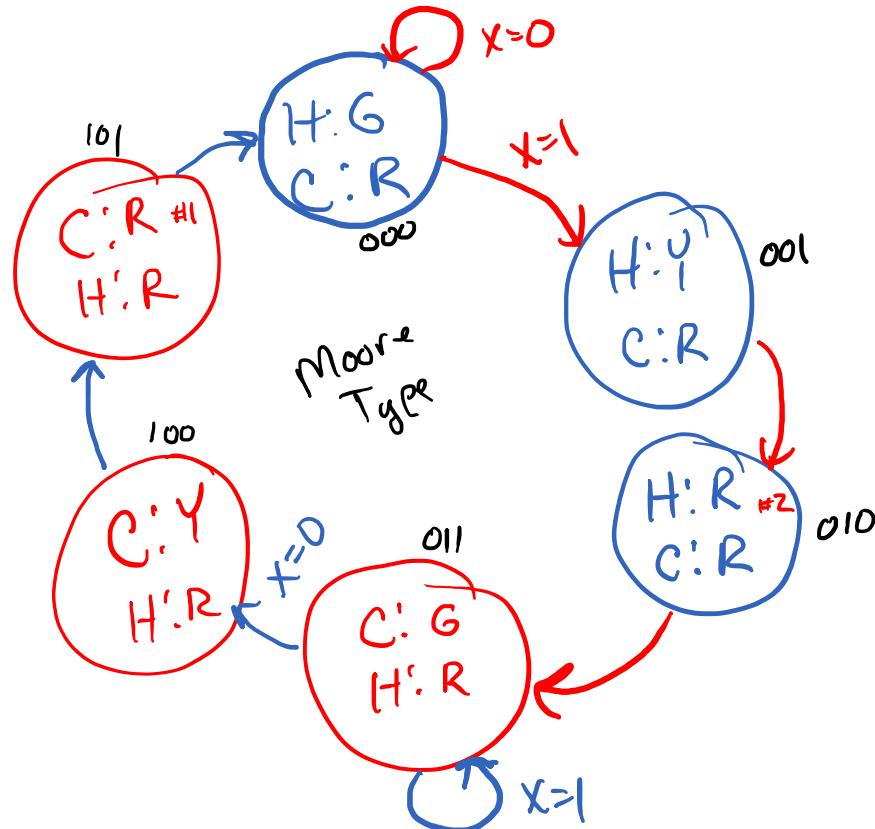
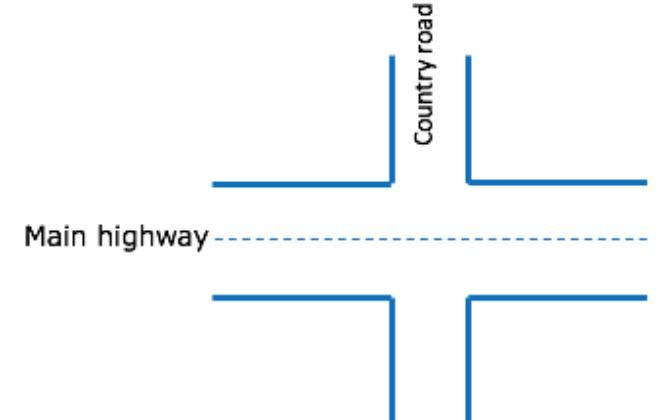
# Moore vs. Mealy Type FSMs

- 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

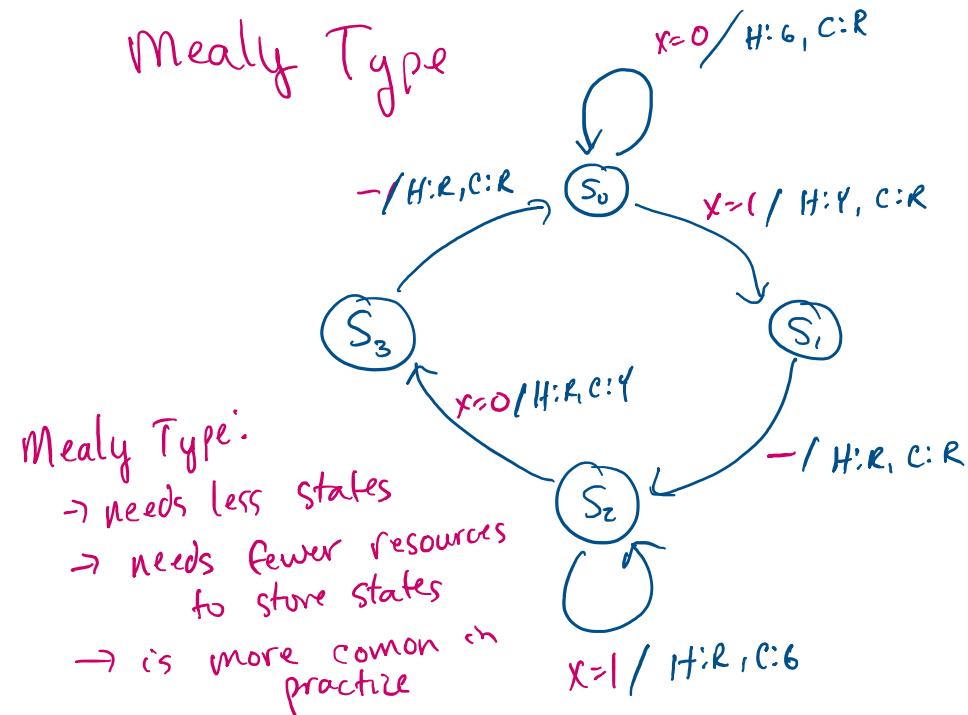
# Traffic Light: Moore vs. Mealy



# Traffic Light: Moore vs. Mealy



Mealy Type

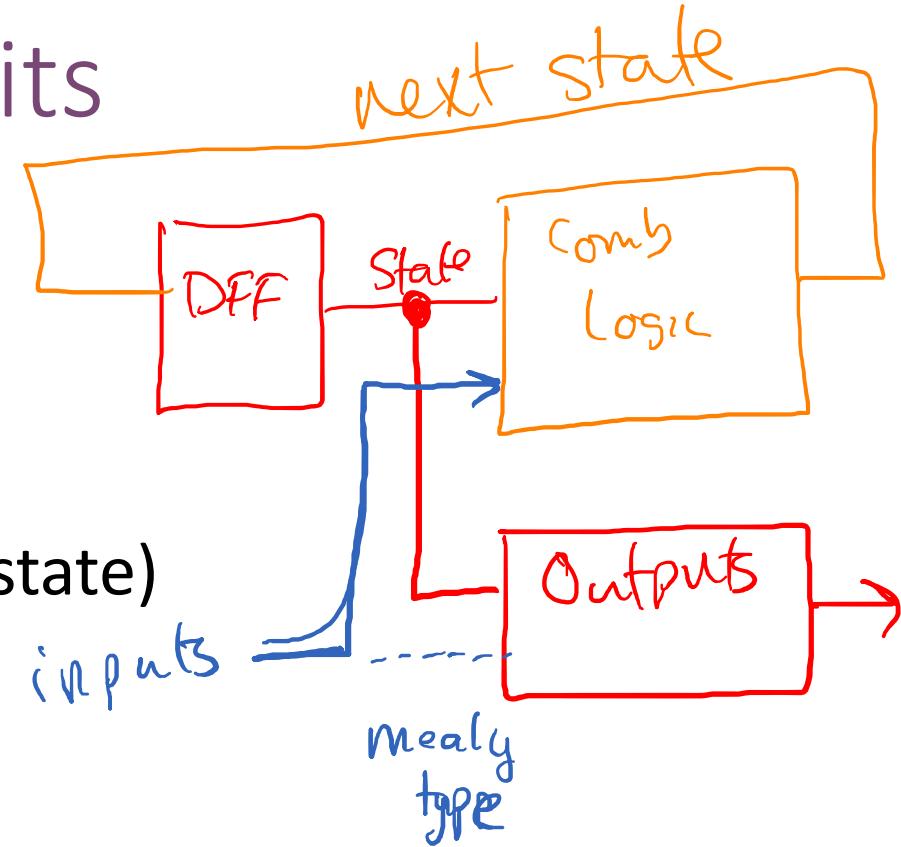


Mealy Type:

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

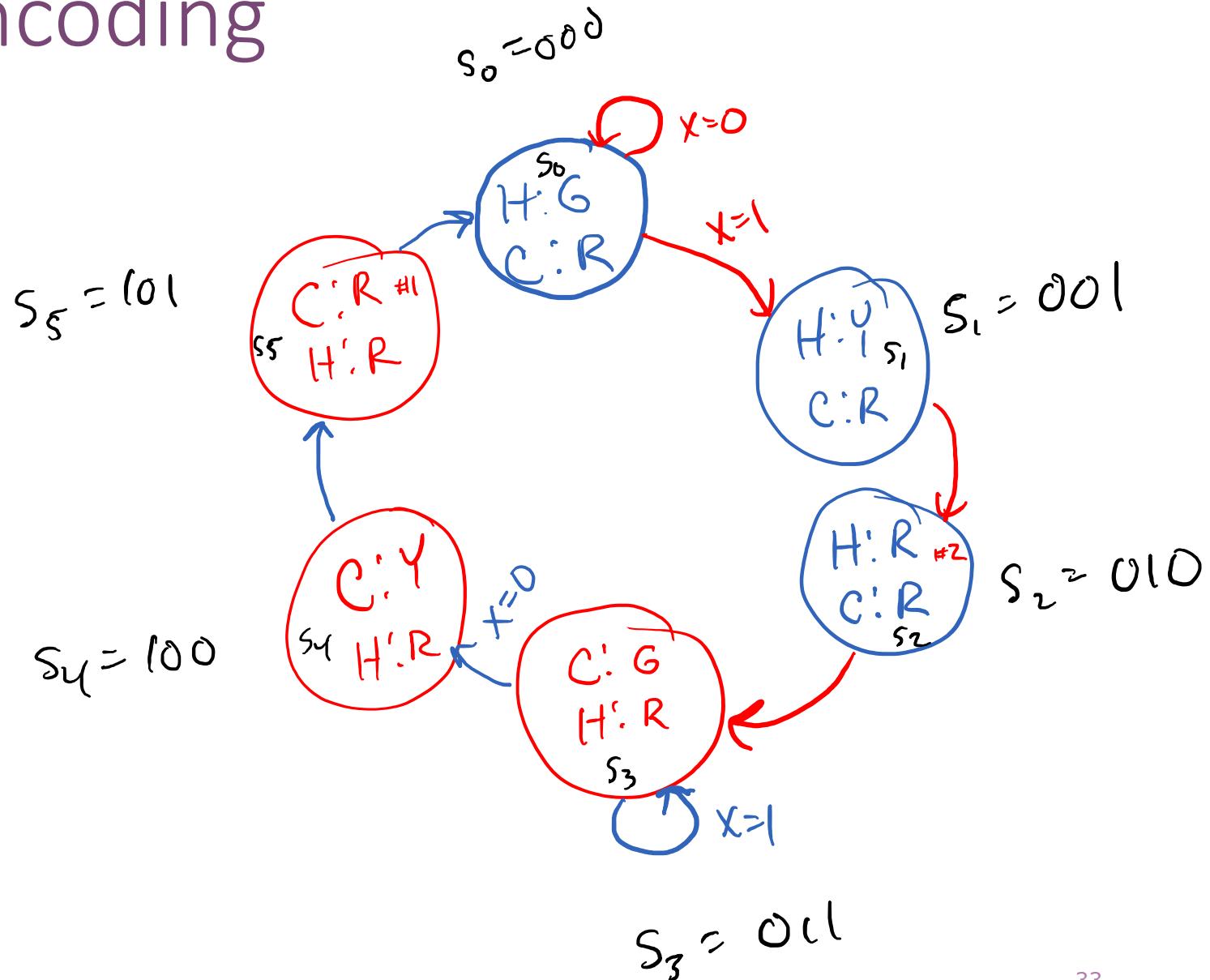
# Implementing FSMs with Circuits

- 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

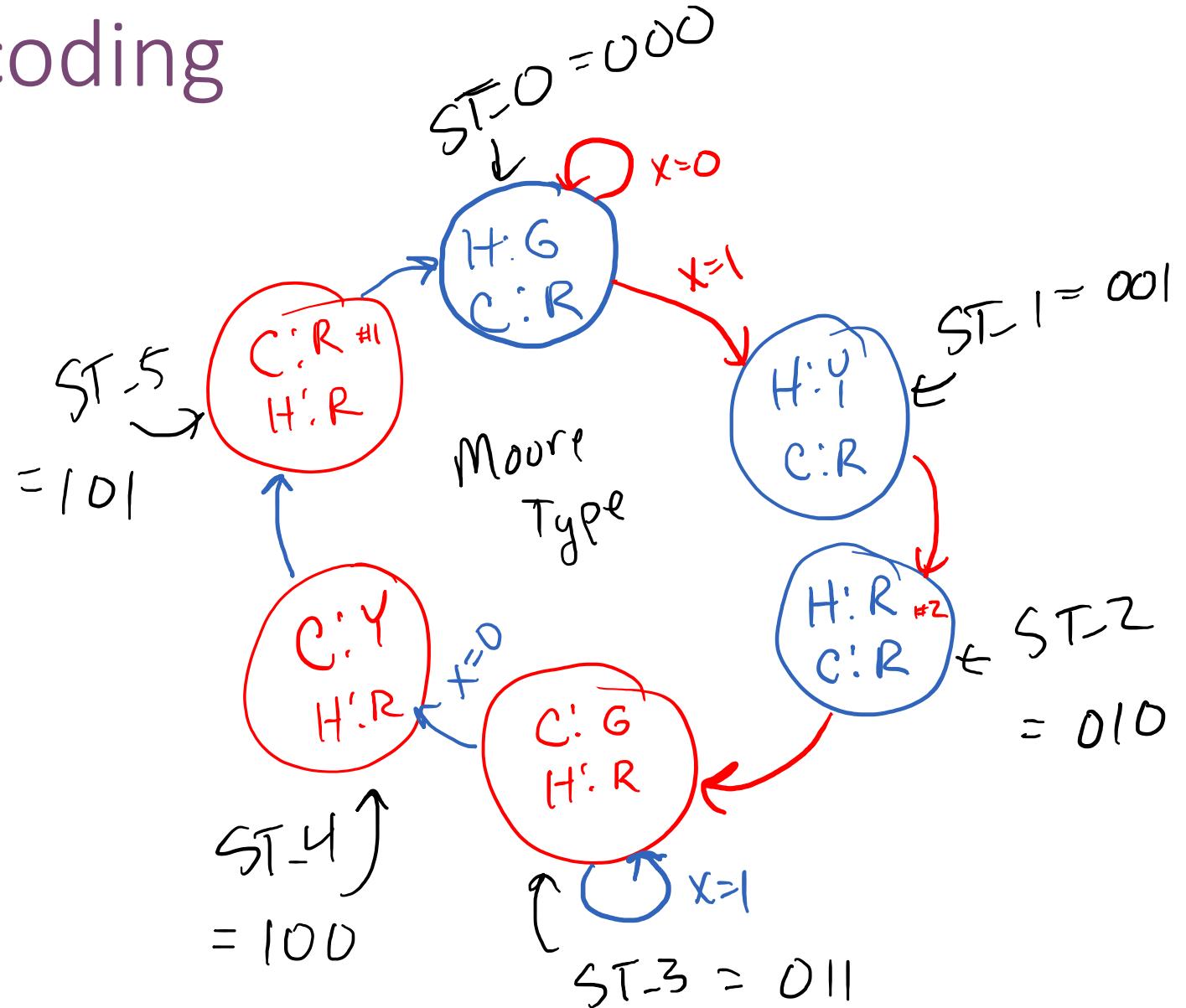


Moore type

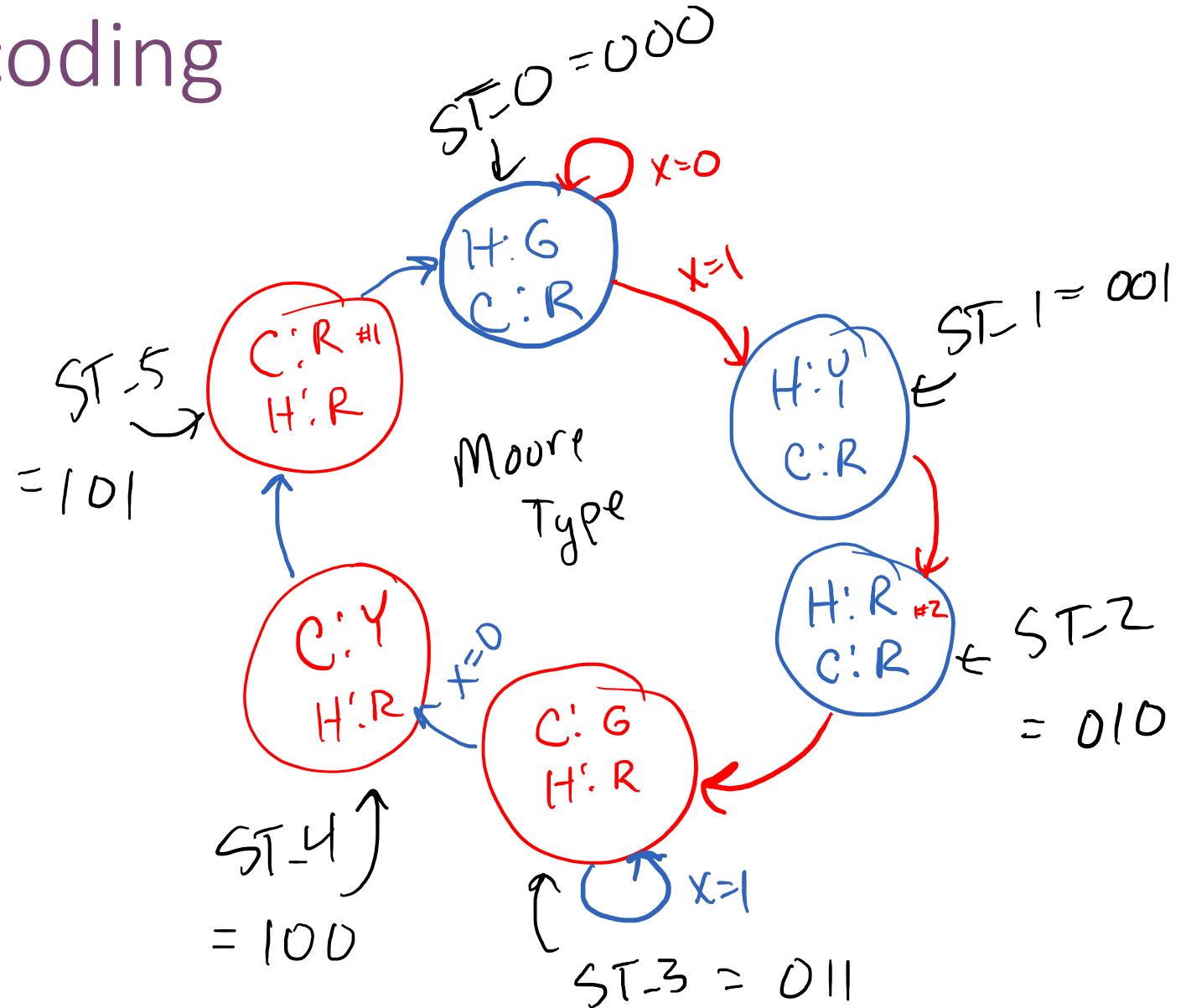
# State Transition Encoding



# State Transition Encoding

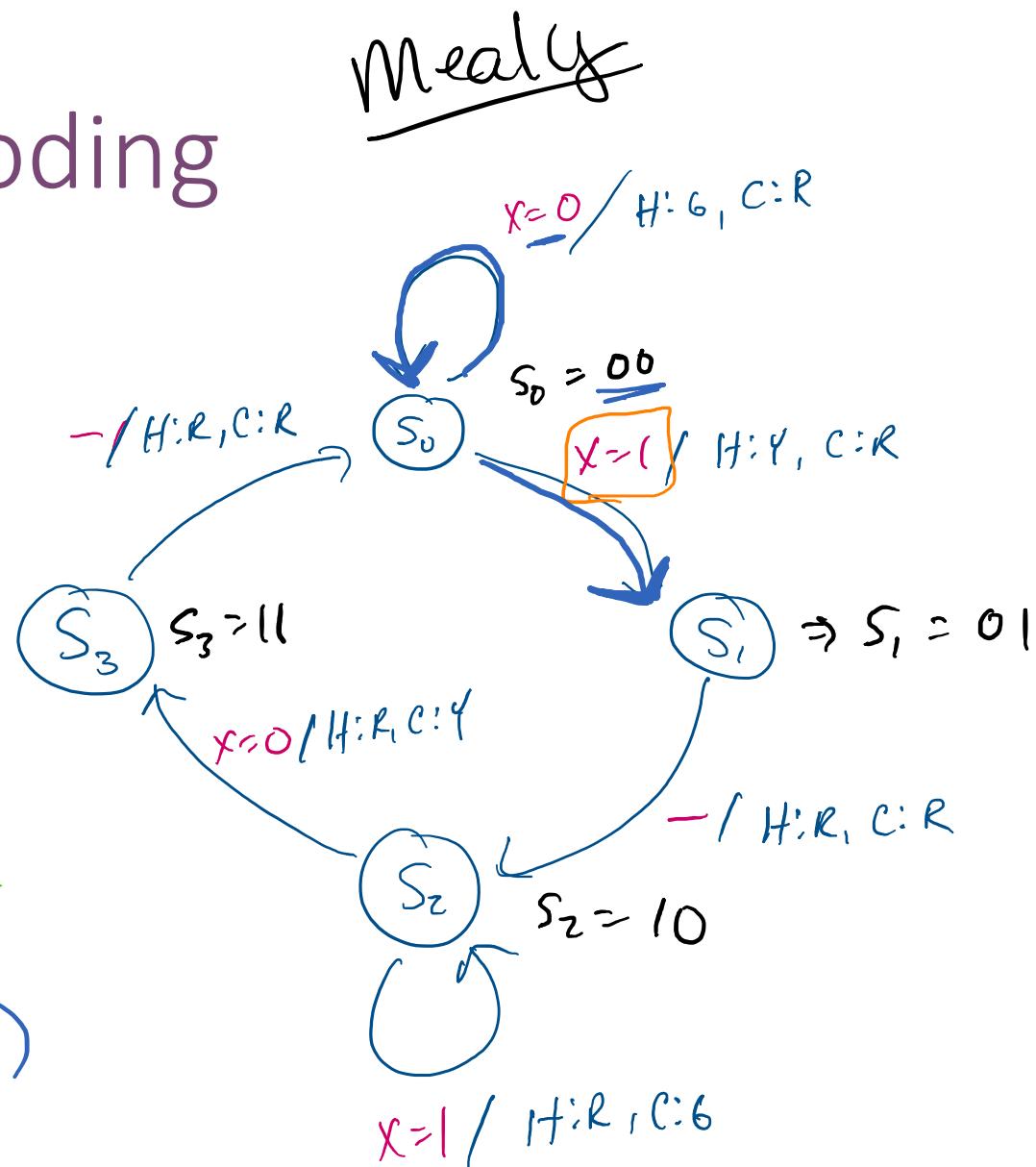


# State Transition Encoding



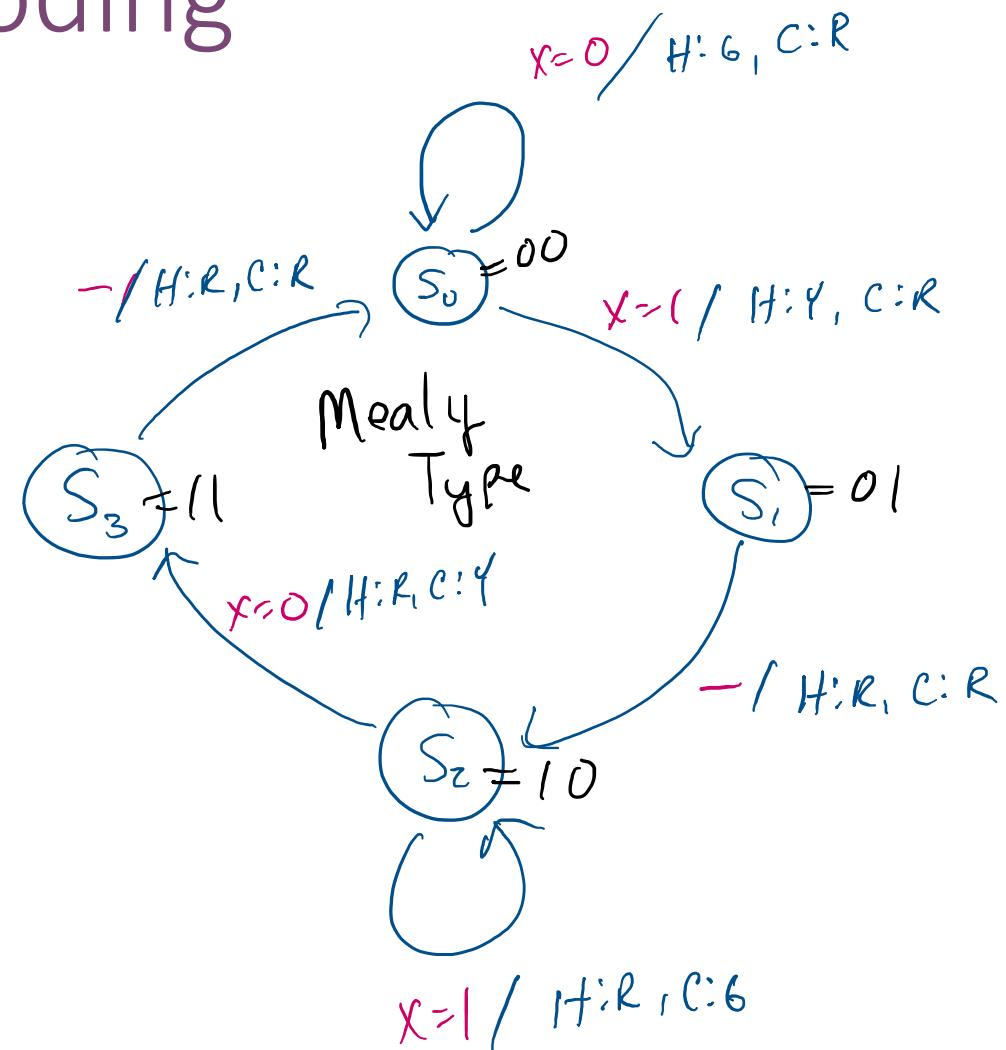
# State Transition Encoding

<u>State</u>	<u>X</u>	<u>Next State</u>
0 0 ( $S_0$ )	0	00 ( $S_0$ )
0 0 ( $S_d$ )	1	01 ( $S_1$ )
0 1	0	10 ( $S_2$ )
0 1	1	10 ( $S_2$ )
1 0	0	11 ( $S_3$ )
1 0	1	10 ( $S_2$ )
1 1	0	00 ( $S_0$ )
1 1	1	00 ( $S_0$ )



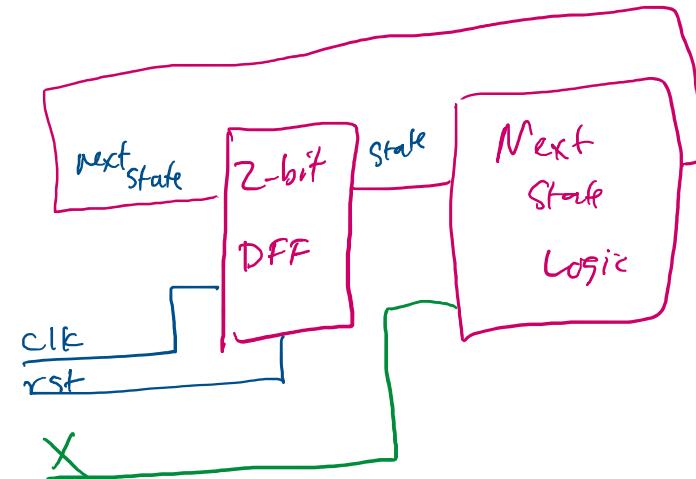
# State Transition Encoding

<u>State</u>	<u>X</u>	<u>next State</u>
00	0	00
00	1	01
01	0	10
01	1	10
10	0	11
10	1	10
11	0	00
11	1	00



# State Machine Encoding

	<u>State</u>	<u>X</u>	<u>Next State</u>
0	00	0	00
1	00	1	01
2	→ 01	0	10
3	→ 01	1	00
4	→ 10	0	11
5	→ 10	1	10
6	11	0	00
7	11	1	00



$$\text{Next State}[1] =$$

$$(\overline{\text{state}[1]} \oplus \text{state}[0] \oplus \overline{X}) \mid$$

$$(\overline{\text{state}[1]} \oplus \text{state}[0] \oplus X) \mid$$

$$(\text{state}[1] \oplus \overline{\text{state}[0]} \oplus \overline{X}) \mid$$

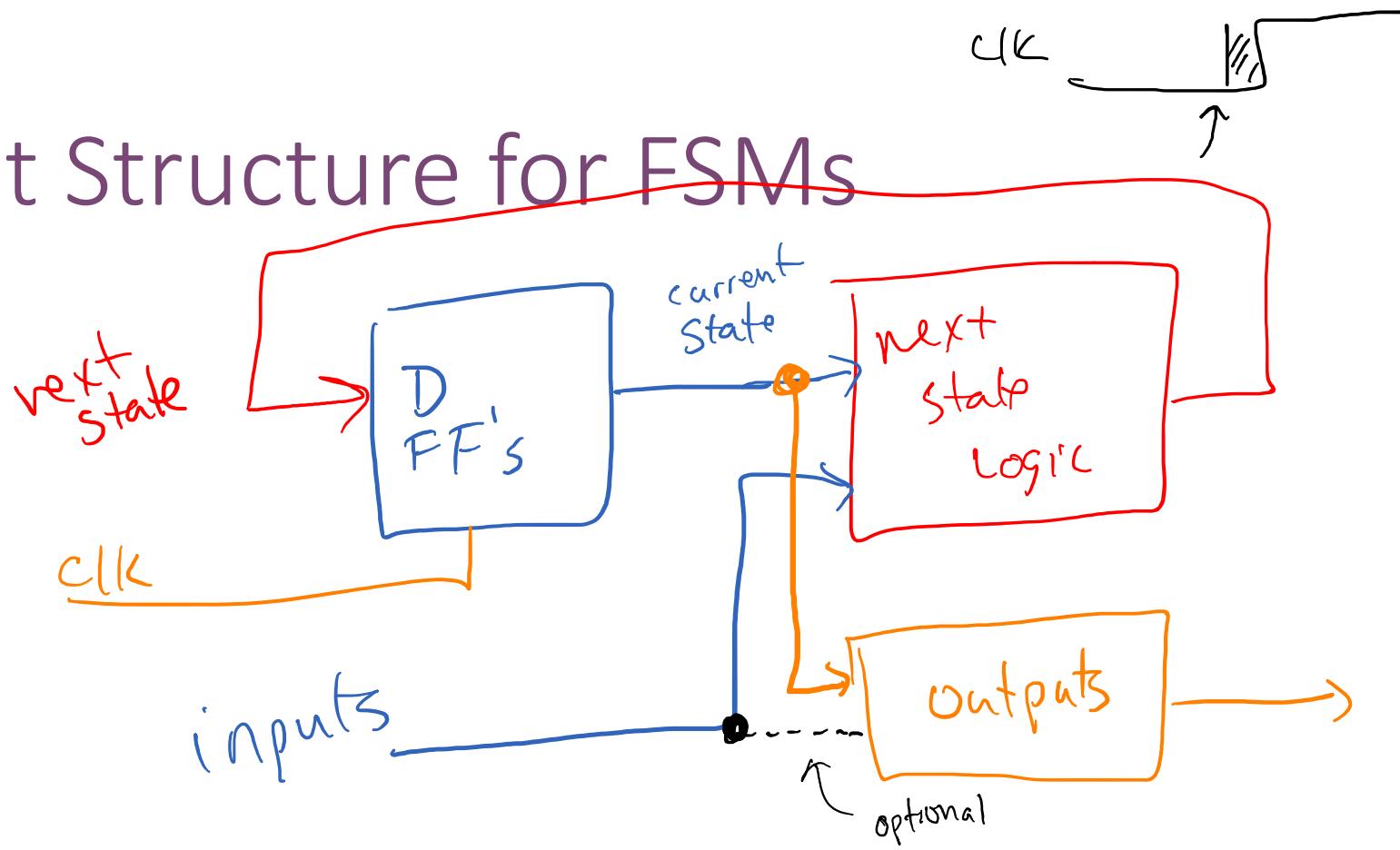
$$(\text{state}[1] \oplus \overline{\text{state}[0]} \oplus X)$$

$$\text{Next State}[0] =$$

$$(\overline{\text{state}[1]} \oplus \text{state}[0] \oplus X) \mid$$

$$(\text{state}[1] \oplus \overline{\text{state}[0]} \oplus X)$$

# Circuit Structure for FSMs

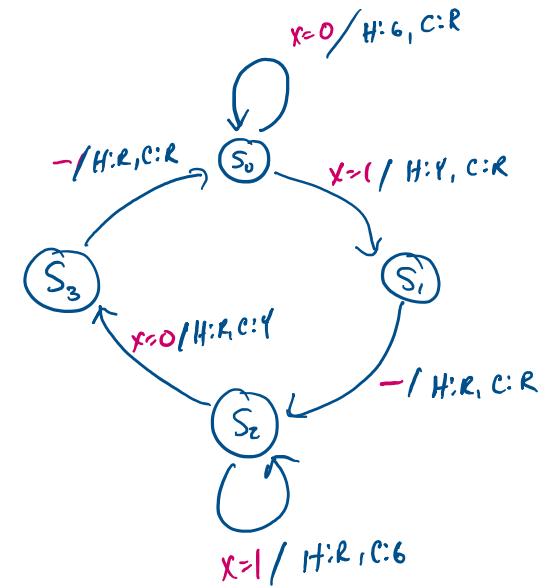


Moore Machine: outputs are a function of current state

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

# Next State Logic

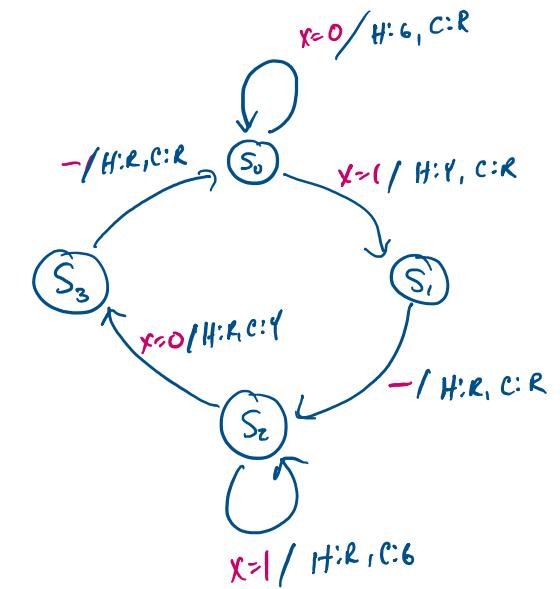
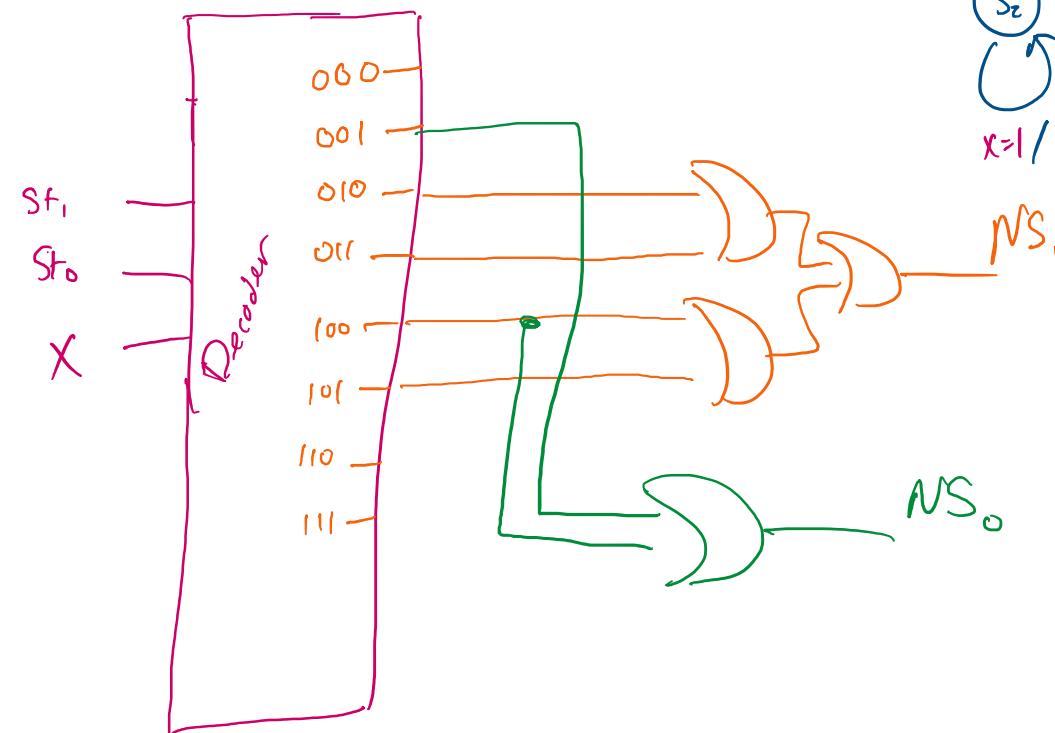
<u>State</u>	<u>X</u>	<u>Next State</u>
00	0	00
00	1	01
01	0	10
01	1	10
10	0	11
10	1	10
11	0	00
11	1	00



# Next State Logic

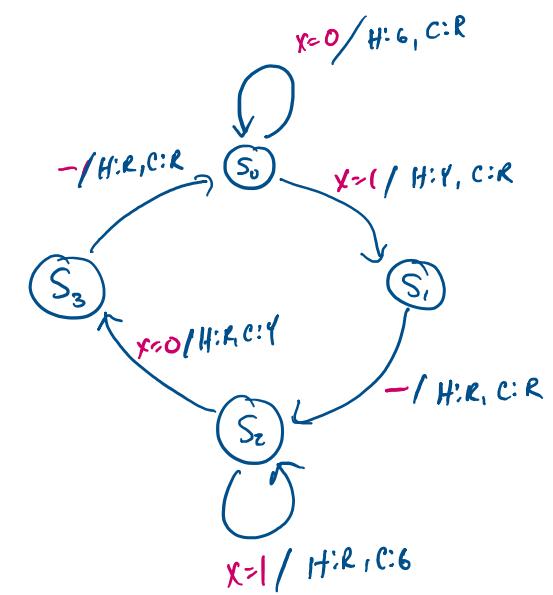
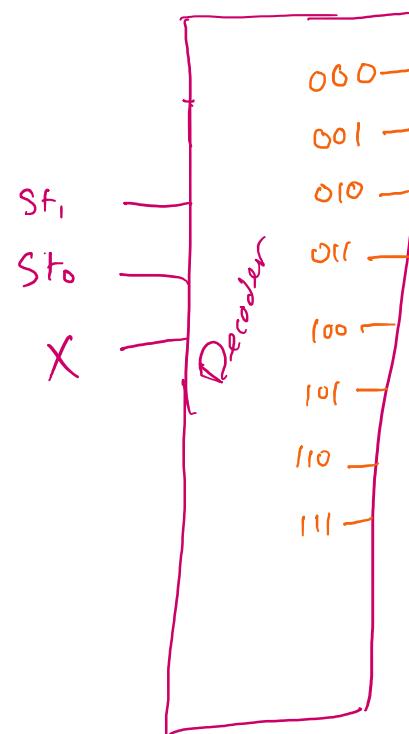
	<u>State</u>	<u>X</u>	<u>Next State</u>
0	00	0	00
1	00	1	01
2	01	0	10
3	01	1	10
4	10	0	11
5	10	1	10
6	11	0	00
7	11	1	00

↑↑      NS<sub>1</sub>      NS<sub>0</sub>



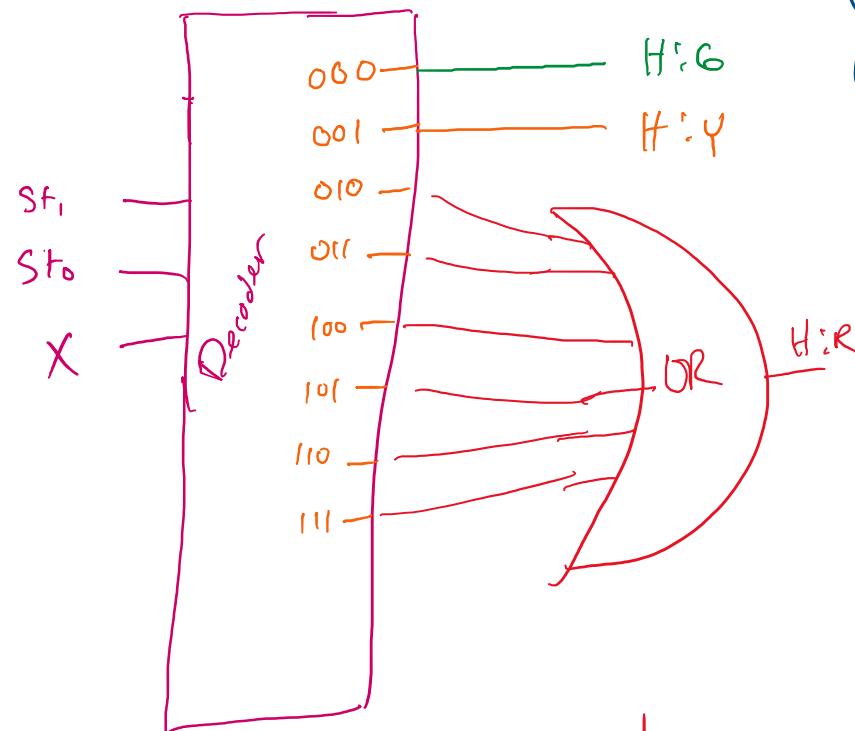
# Output Logic (Highway)

	<u>State</u>	<u>X</u>
0	00	0
1	00	1
2	01	0
3	01	1
4	10	0
5	10	1
6	11	0
7	11	1

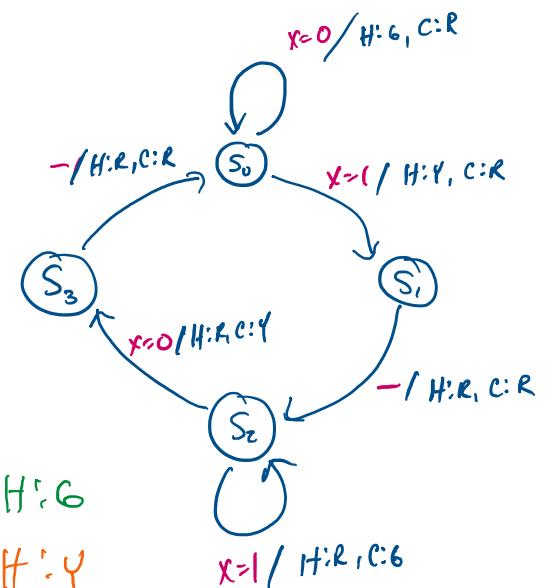


# Output Logic (Highway)

	<u>State</u>	<u>X</u>	<u>H:R</u>	<u>H:Y</u>	<u>H:G</u>
0	00	0	0	0	1
1	00	1	0	1	0
2	01	0	1	0	0
3	01	1	1	0	0
4	10	0	1	0	0
5	10	1	1	0	0
6	11	0	1	0	0
7	11	1	1	0	0

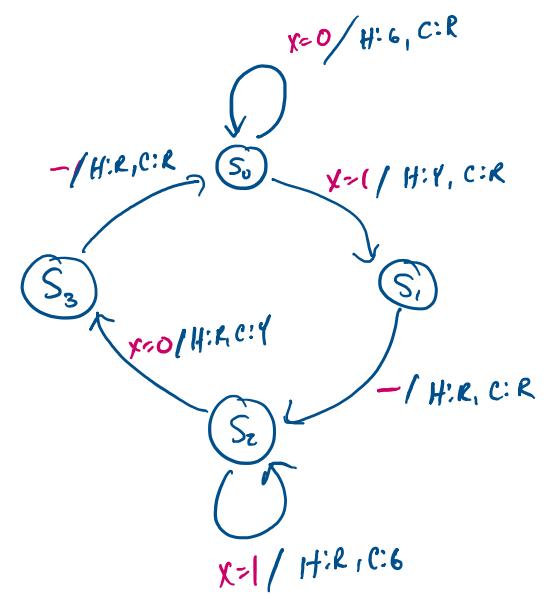
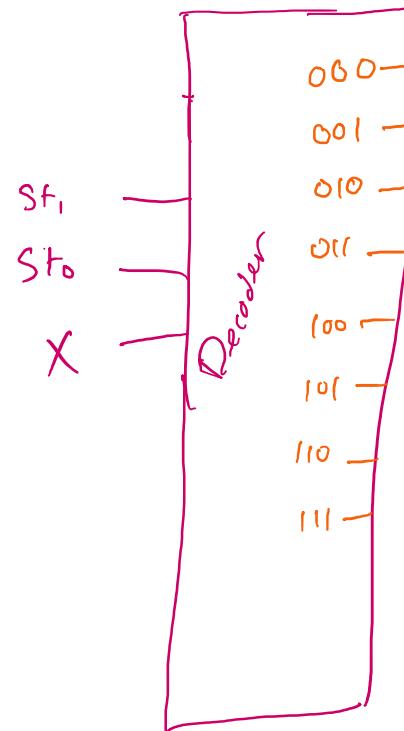


\* Simpler methods exist



# Output Logic (Country Rd)

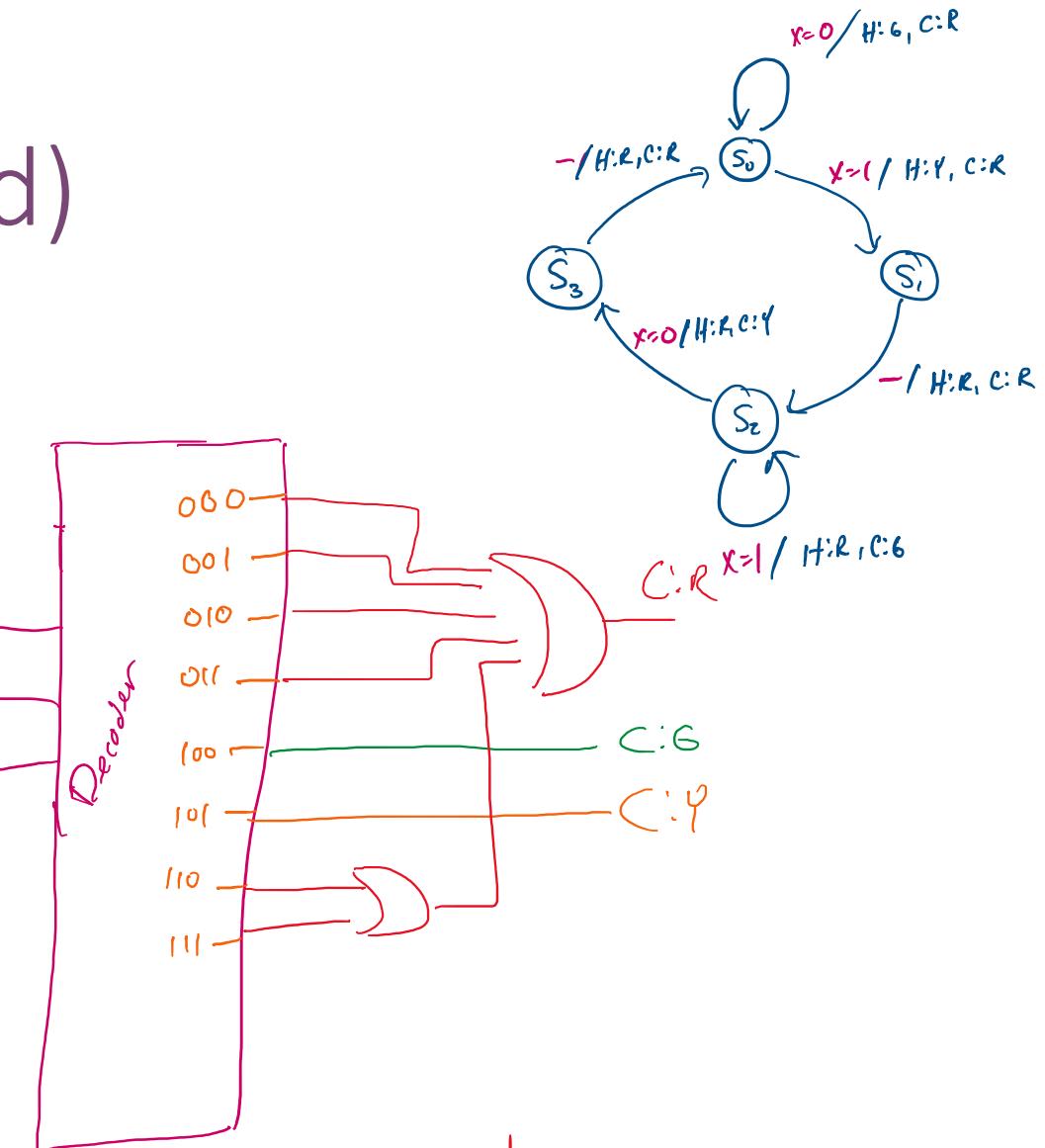
	<u>State</u>	X
0	00	0
1	00	1
2	01	0
3	01	1
4	10	0
5	10	1
6	11	0
7	11	1



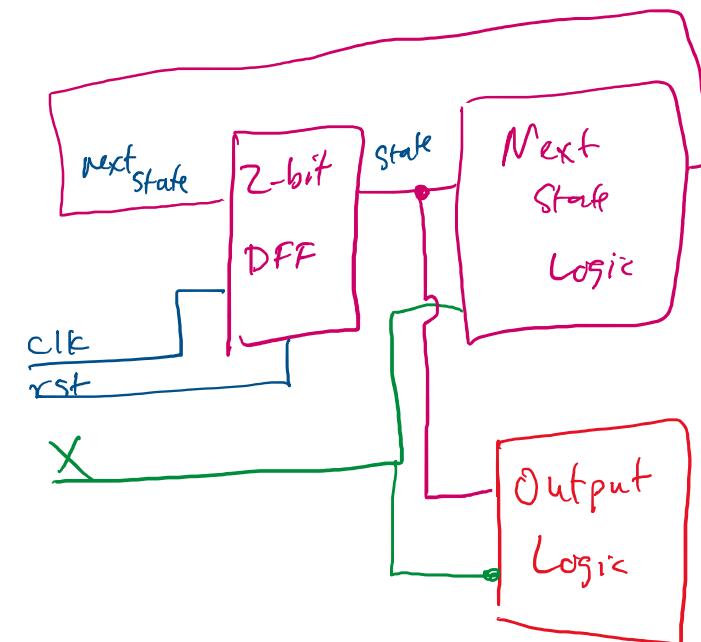
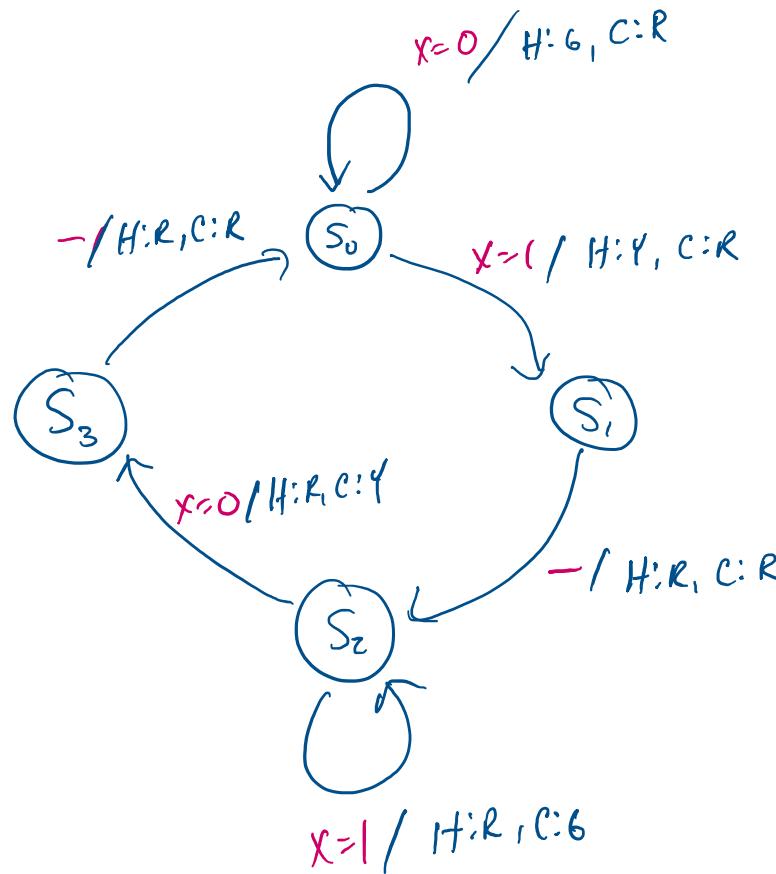
# Output Logic (Country Road)

	<u>State</u>	<u>X</u>	C:R	C:Y	C:G
0	00	0	1	0	0
1	00	1	0	0	0
2	01	0	0	0	0
3	01	1	1	0	0
4	10	0	0	0	0
5	10	1	0	0	1
6	11	0	1	0	0
7	11	1	0	0	0

\* Simpler methods exist

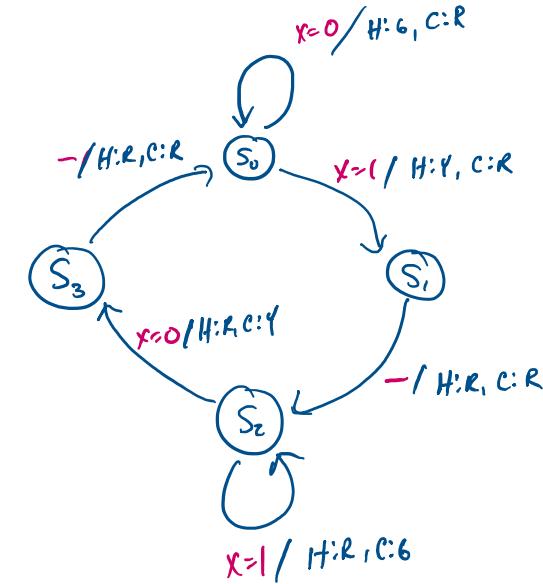


# State Machine to Logic



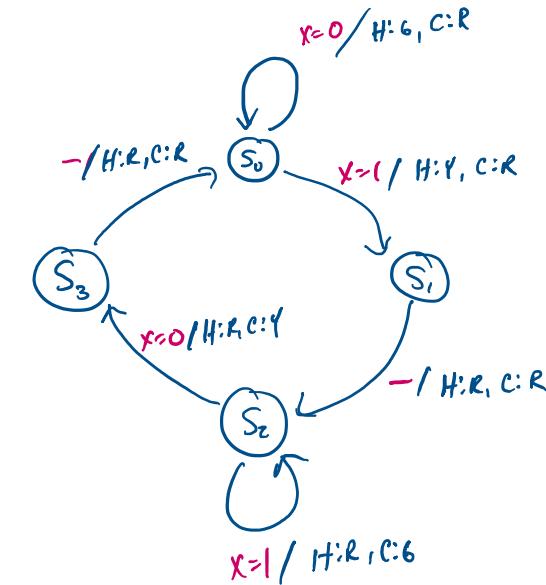
# State Machine to Verilog

- Define states?



# State Machine to Verilog

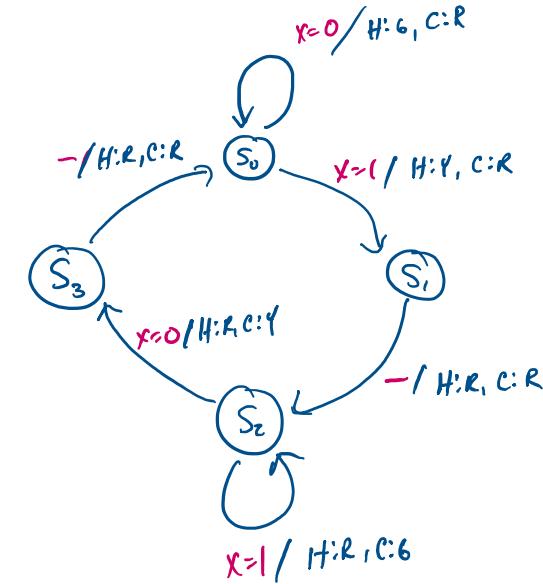
- Define states?



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

# State Machine to Verilog

- Build State Machine?

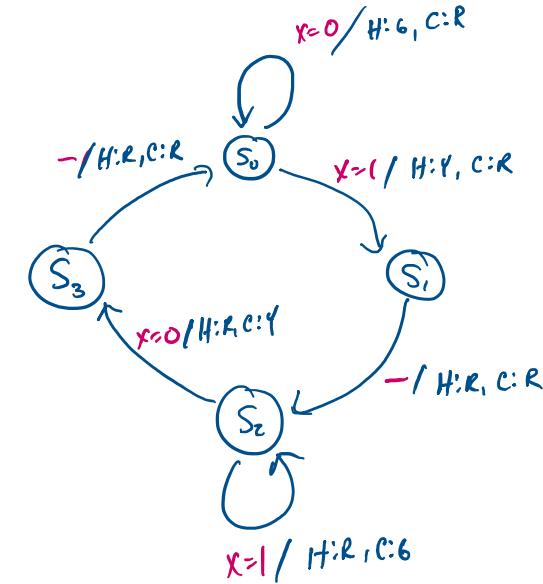


# State Machine to Verilog

- Build State Machine?

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

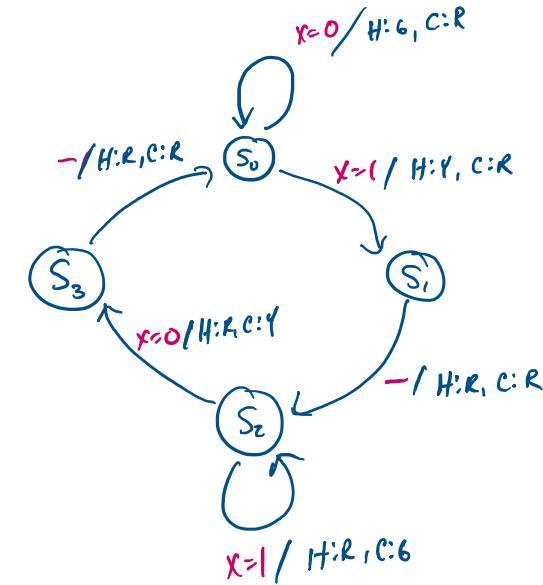
- What is nextState?



# State Machine to Verilog

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

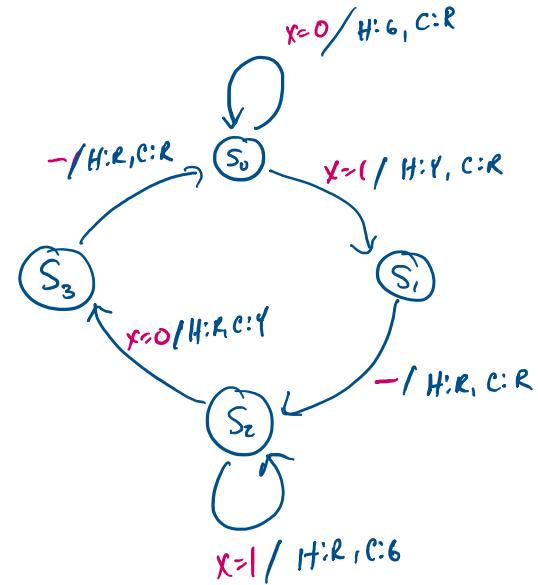
- What is nextState?



# State Machine to Verilog

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

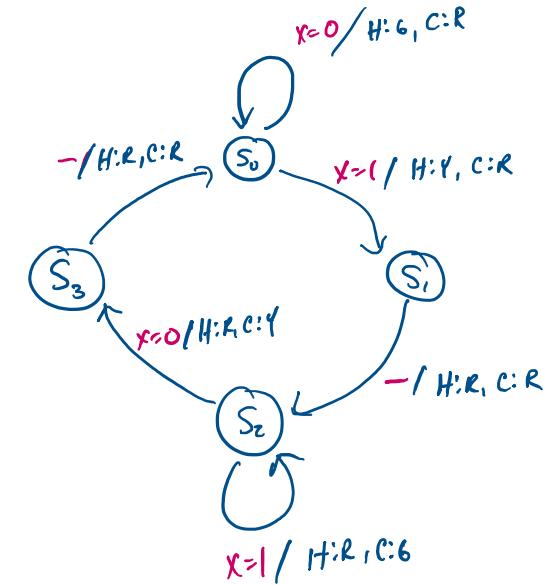
always_comb begin
    nextState = state; //default
    case(state)
        ST_0: nextState = ST_1; //goto state 1
        ST_1: nextState = ST_2;
        ST_2: nextState = ST_3;
        ST_3: nextState = ST_0; //loop
        default: nextState = ST_0; //just in case
    endcase
end
```



# State Machine to Verilog

- What is this missing?

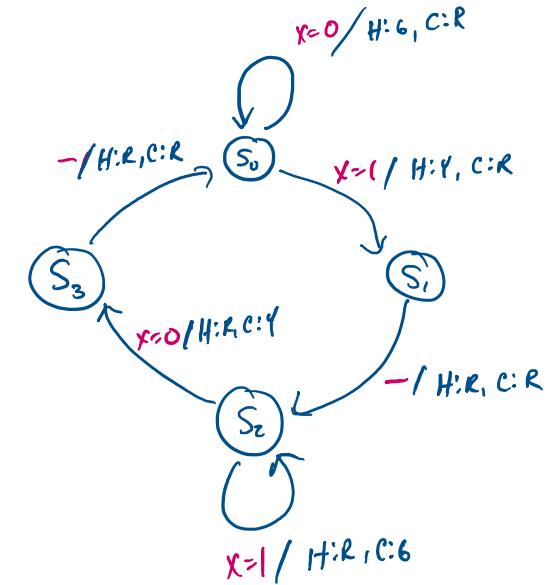
```
always_comb begin
    nextState = state; //default
    case(state)
        ST_0: nextState = ST_1; //goto state 1
        ST_1: nextState = ST_2;
        ST_2: nextState = ST_3;
        ST_3: nextState = ST_0; //loop
        default: nextState = ST_0; //just in case
    endcase
end
```



# State Machine to Verilog

- What is this missing?

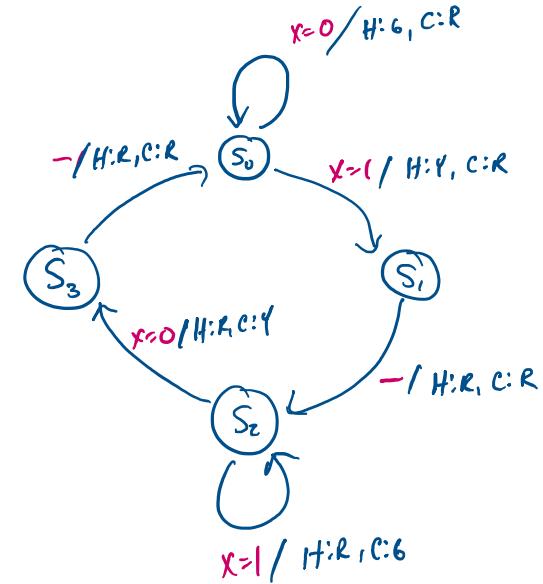
```
always_comb begin
    nextState = state; //default
    case(state)
        ST_0:
            nextState = ST_1;
        ST_1:
            nextState = ST_2;
        ST_2:
            nextState = ST_3;
        // ST_3 and default cases      endcase
    end
```



# State Machine to Verilog

- What is this missing?

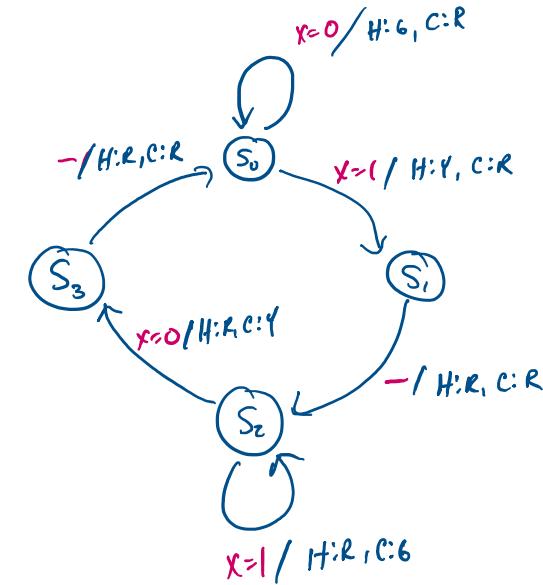
```
always_comb begin
    nextState = state; //default
    case(state)
        ST_0:
            if (X) nextState = ST_1;
        ST_1:
            nextState = ST_2;
        ST_2:
            if (~X) nextState = ST_3;
        // ST_3 and default cases    endcase
    end
```



# State Machine to Verilog

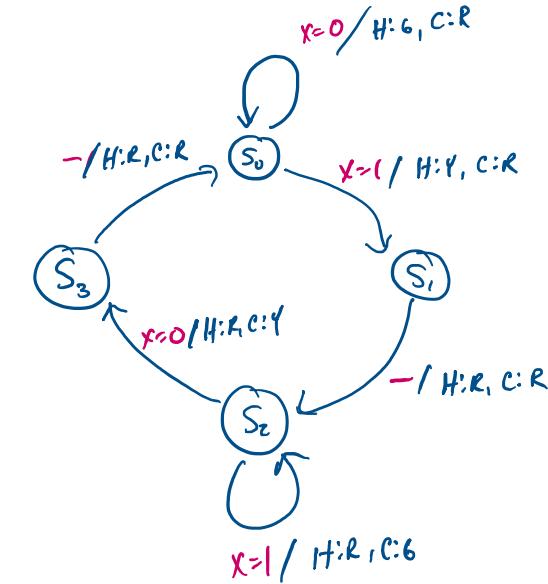
- What else is this missing?

```
always_comb begin
    nextState = state; //default
    case(state)
        ST_0:
            if (X) nextState = ST_1;
            // ST_1-3 and default cases
    endcase
end
```

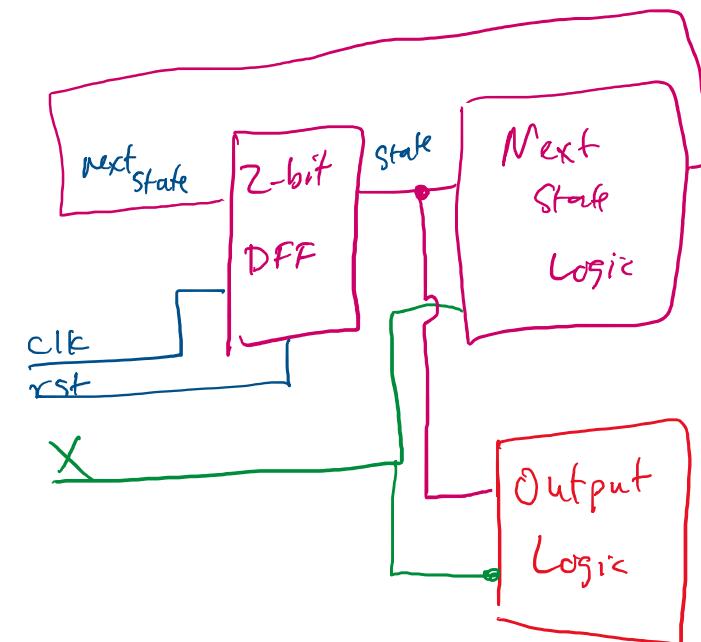
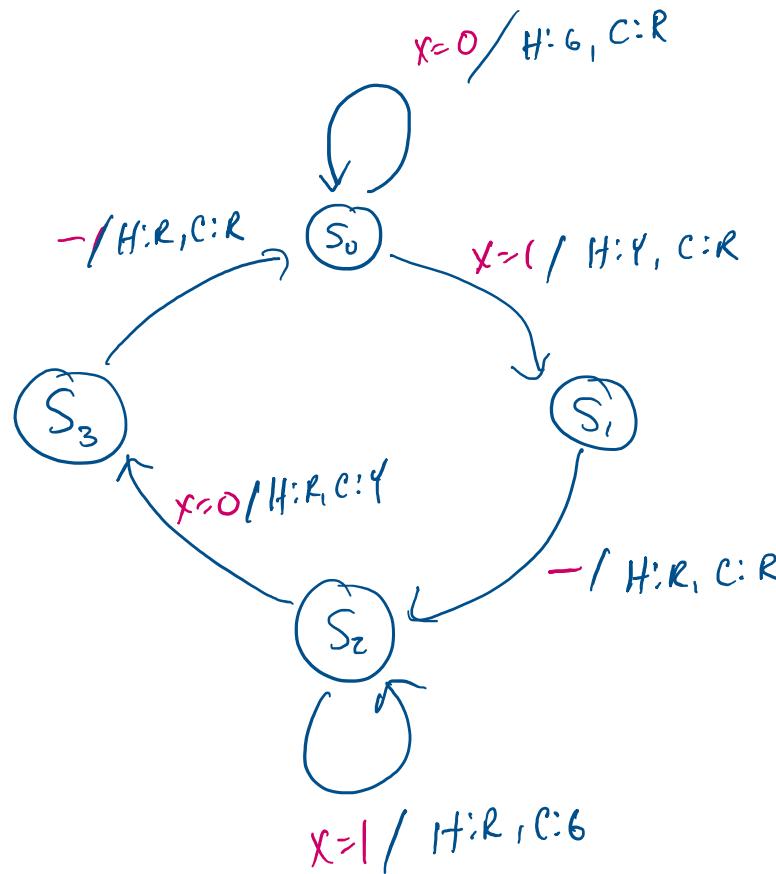


# State Machine to Verilog

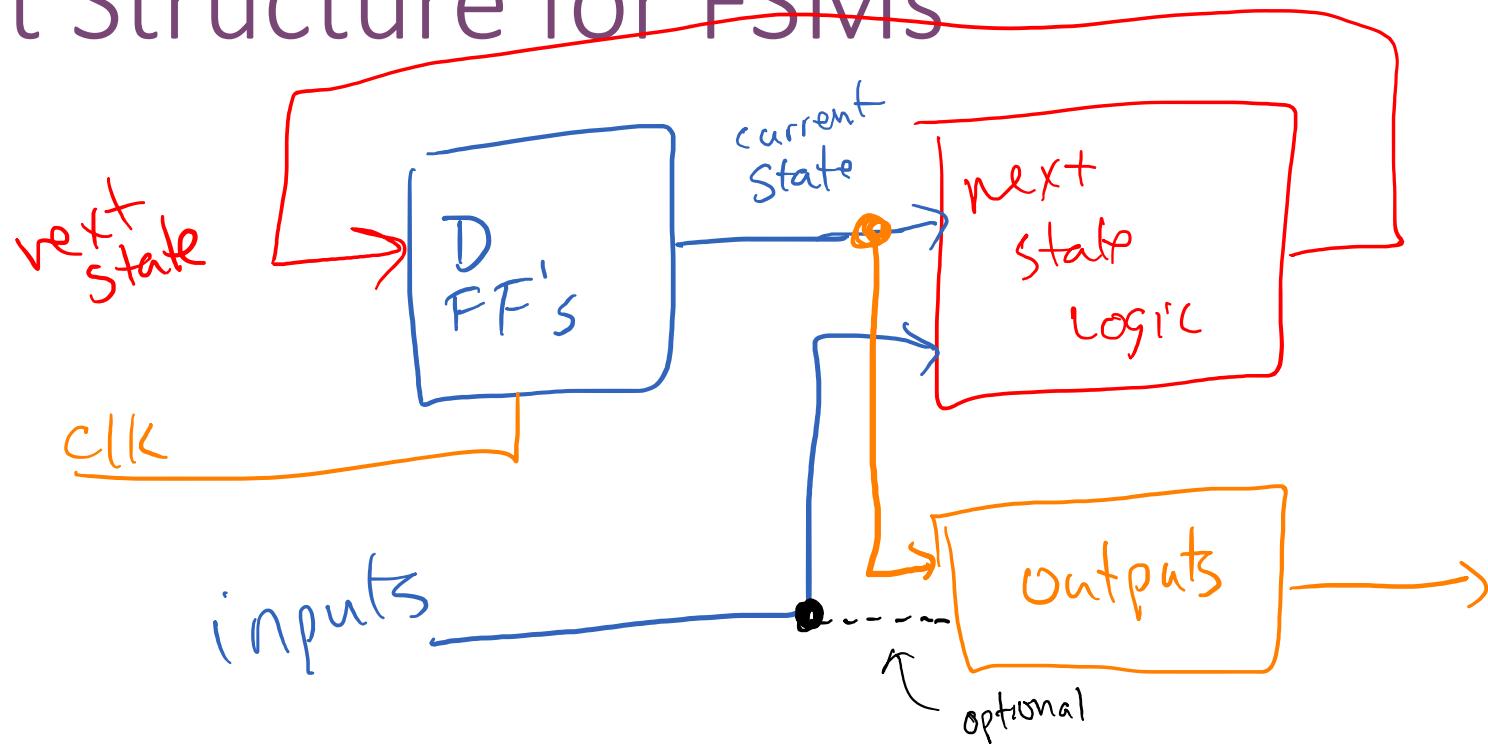
```
always_comb begin
    nextState = state; //default
    Hryg = {0,0,1}; Cryg={1,0,0};
    case(state)
        ST_0: begin
            if (X) begin
                nextState = ST_1;
                Hryg = {0,1,0};
                Cryg = {1,0,0}; //optional
            end else begin
              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