Lecture 6: Designing Sequential Logic

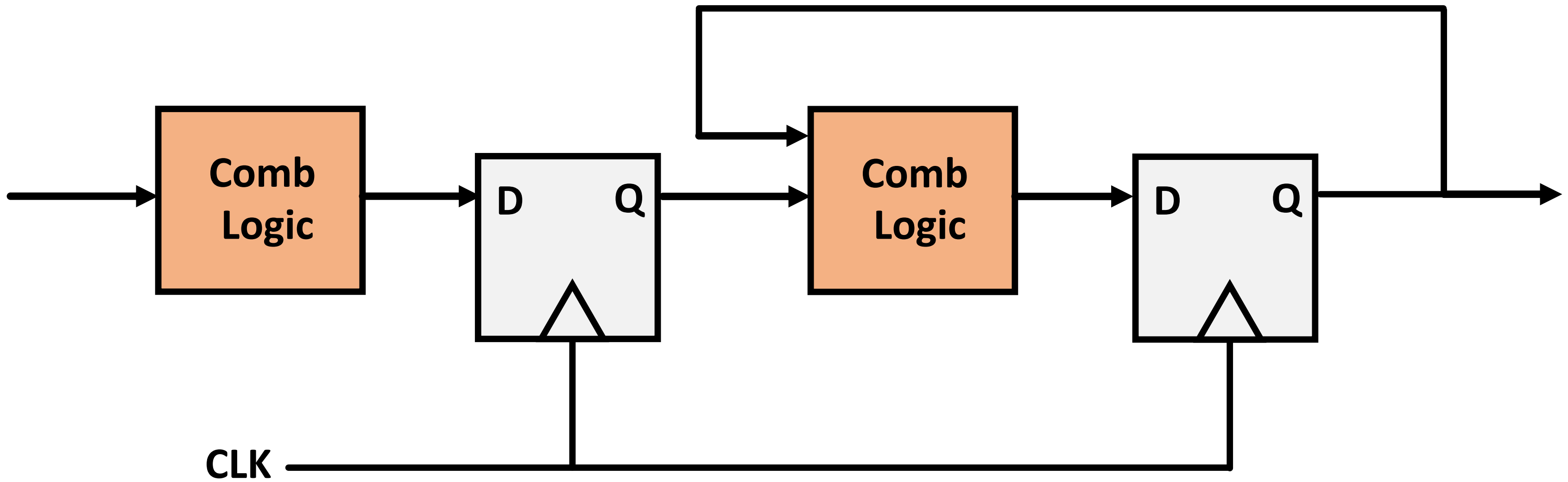
Lpset 5 out now, due Thursday

Lab 02 Part 2 due today/tomorrow!

Lab 03 out, due in a week!

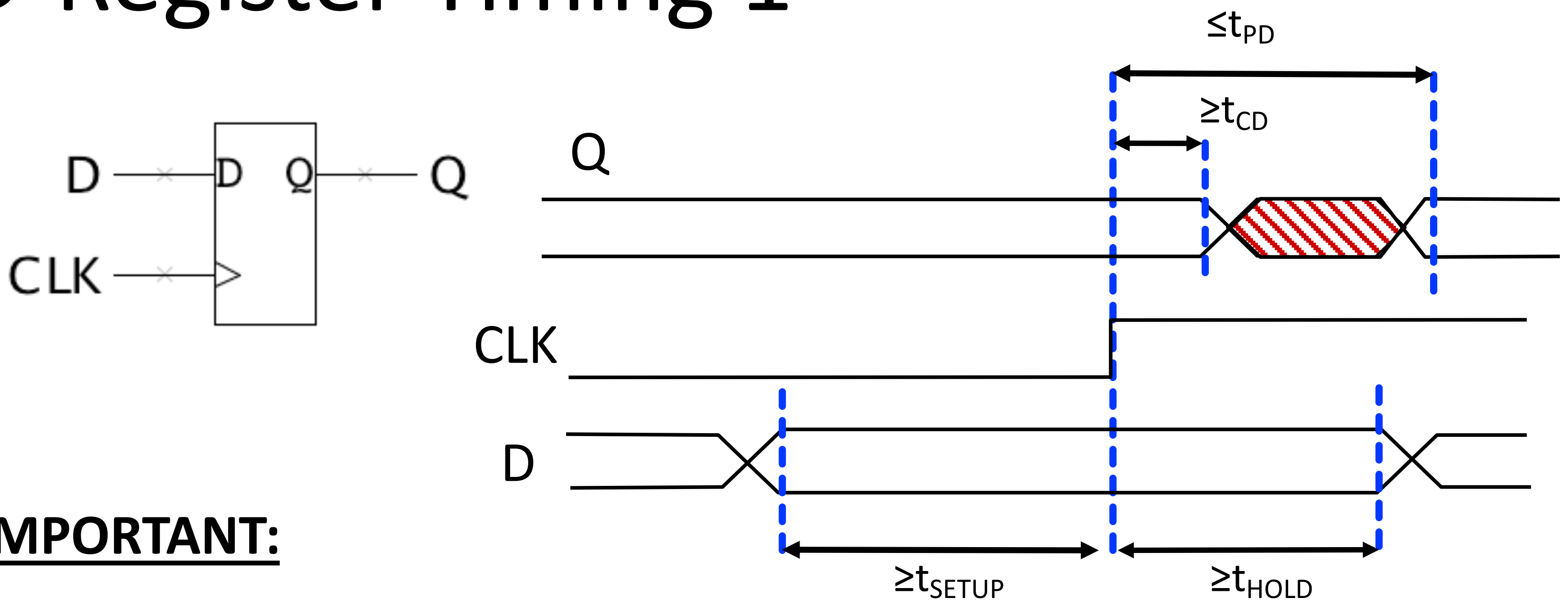
Previously on 6.111

- We've started to discuss how to design things in stages:
  - Split operations down into collections of combinational logic isolated by register/flip-flops,
  - Our designs become functions of time



D-Register Timing 1

▨ =undetermined state



IMPORTANT:

t_{PD}: maximum propagation delay, @posedge CLK D →Q

__Maximum__ time it takes for Q to change after rising edge of CLK

t_{CD}: minimum contamination delay, @posedge CLK D →Q

__Minimum__ time it takes for Q to start to change after rising edge of CLK

t_{SETUP}: setup time

How long D must be stable __before__ the rising edge of CLK

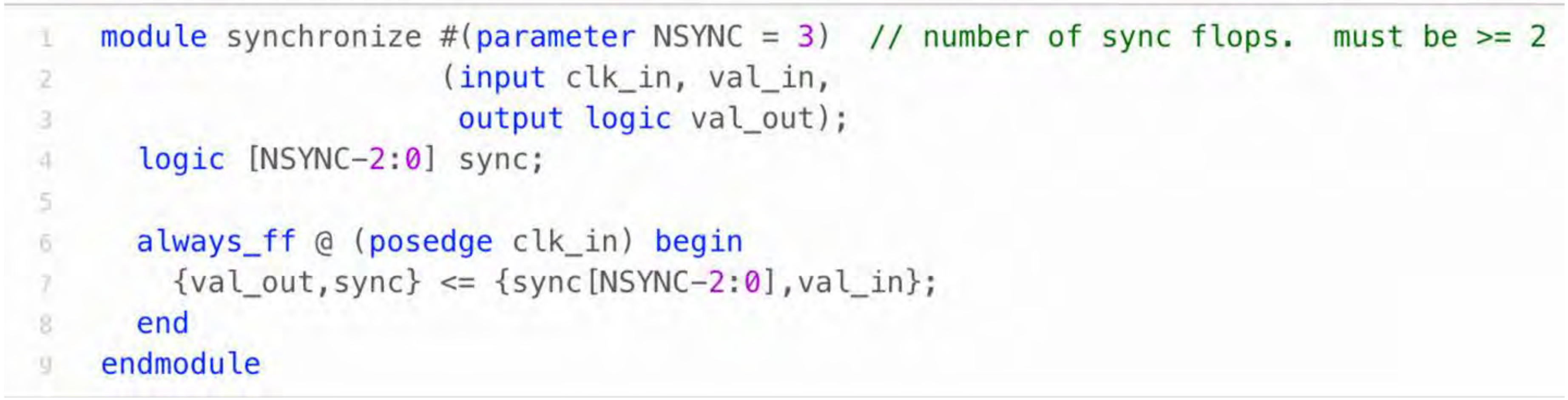
t_{HOLD}: hold time

How long D must be stable __after__ the rising edge of CLK

} ***New timing attributes for registers***

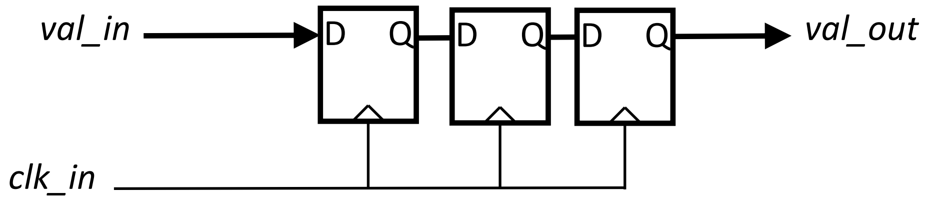
Huh?

- In Lab 3:



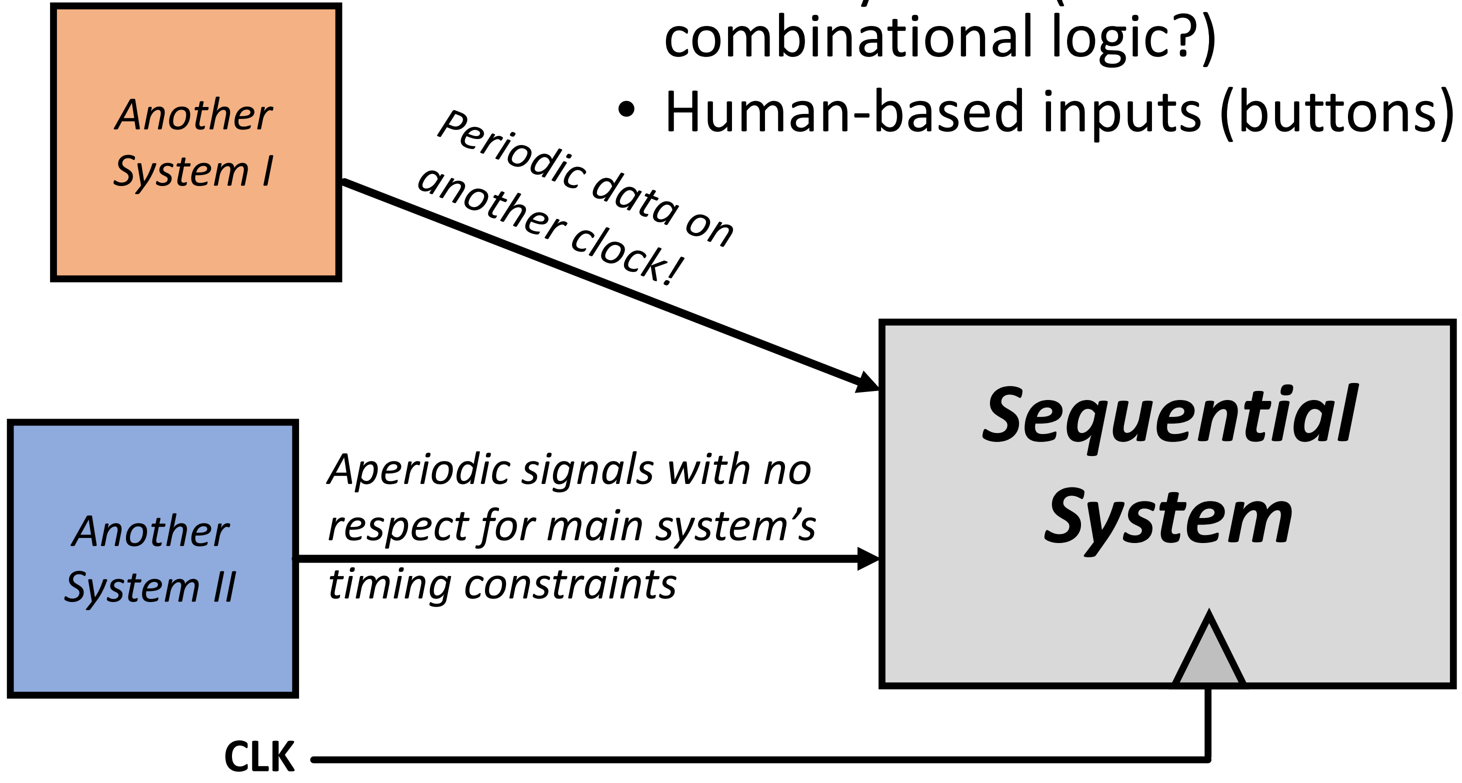
```
module synchronize #(parameter NSYNC = 3)  // number of sync flops.  must be >= 2
                   (input clk_in, val_in,
                    output logic val_out);
  logic [NSYNC-2:0] sync;

  always_ff @ (posedge clk_in) begin
    {val_out,sync} <= {sync[NSYNC-2:0],val_in};
  end
endmodule
```

- Basically builds this:



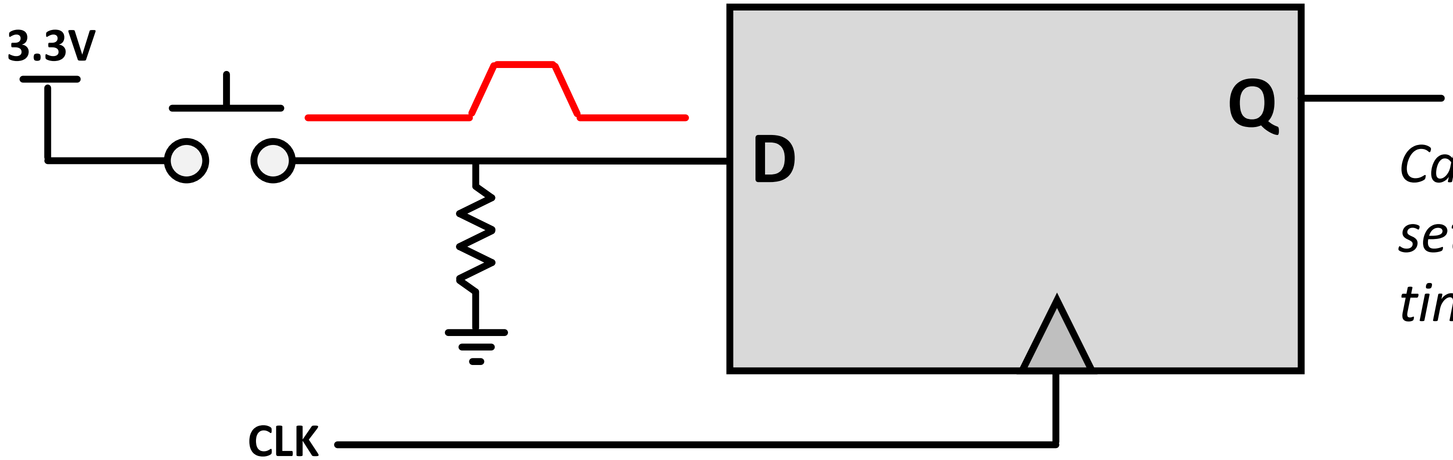
What if…?

- …we need to interface with outside equipment:
  - Other systems (on different clocks or from combinational logic?)
  - Human-based inputs (buttons)


Another System I
Periodic data on another clock!
Another System II
Aperiodic signals with no respect for main system's timing constraints
Sequential System
CLK

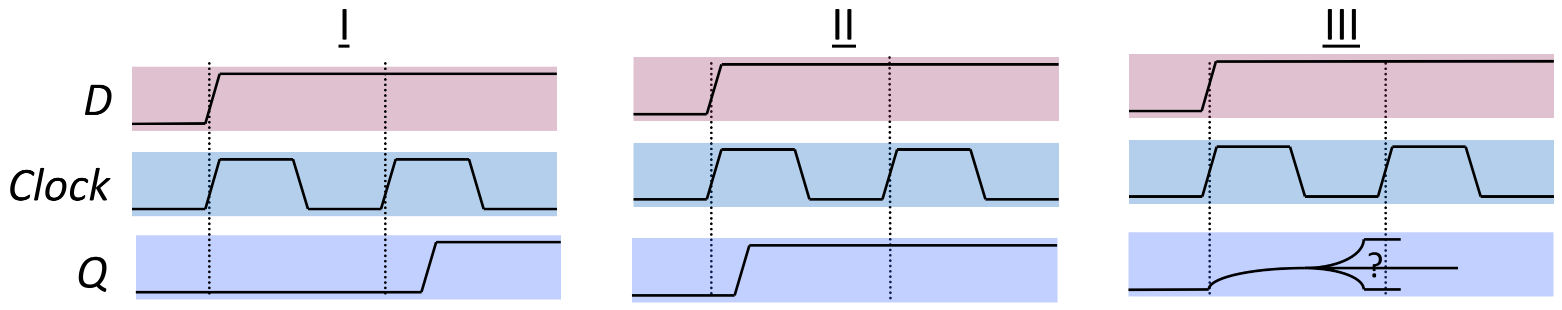
Can't guarantee setup and hold times will be met!

Example: Asynchronous Inputs in Sequential Systems



Can't guarantee setup and hold times will be met!

When an asynchronous signal causes a setup/hold violation...



Transition is missed on first clock cycle, but caught on next clock cycle.

Transition is caught on first clock cycle.

Output is metastable for an indeterminate amount of time.

Q: Which cases are problematic?

Metastability

- D-registers have issues with all that feedback and stuff going on. Can go **metastable**
- Metastability is where the system hovers between Logic High and Logic Low in an unpredictable way

Figure 2. Effects of Violating t_{SU} & t_H Requirements

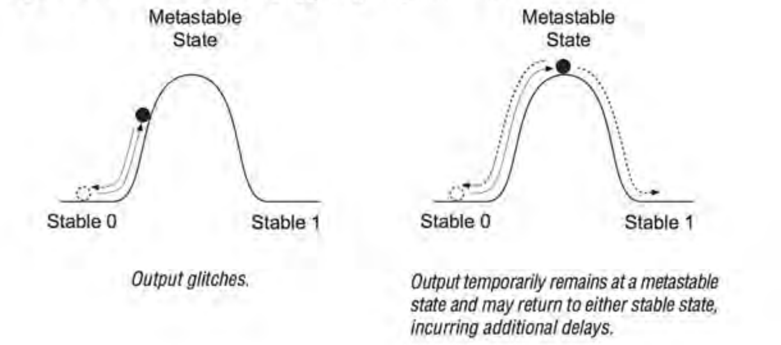
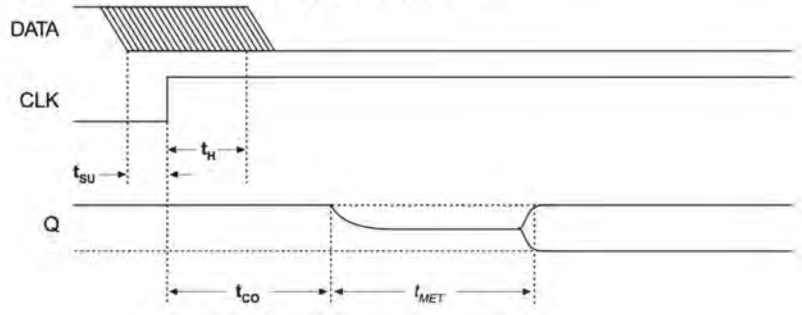
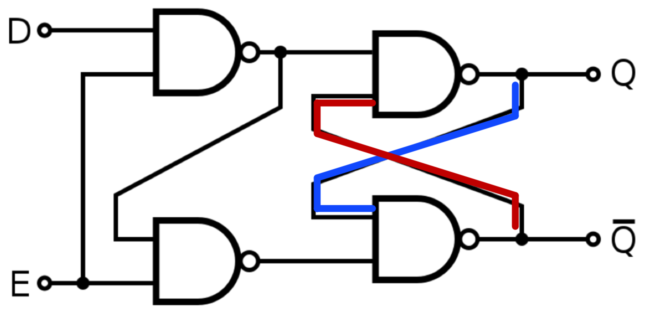

Metastable State
Stable 0
Stable 1
Output glitches.
Metastable State
Stable 0
Stable 1
Output temporarily remains at a metastable state and may return to either stable state, incurring additional delays.

Figure 1. Metastability Timing Parameters



Metastability in Altera (☹) Devices
Altera Application Note 42 (1999)



t_{co} = *"min time from clock to output"*
....think of it as t_{pd} *here (not exactly the same*

Handling Metastability

- Can't globally prevent metastability, but can isolate it!
- Stringing several registers together can isolate any freakouts!

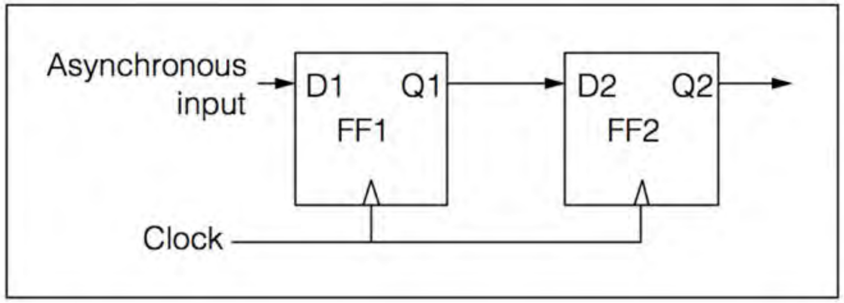
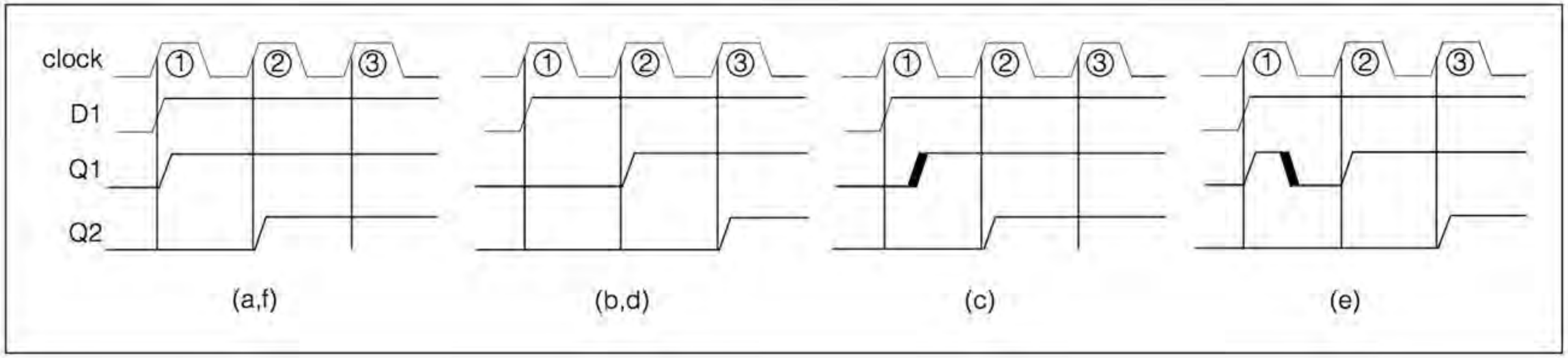


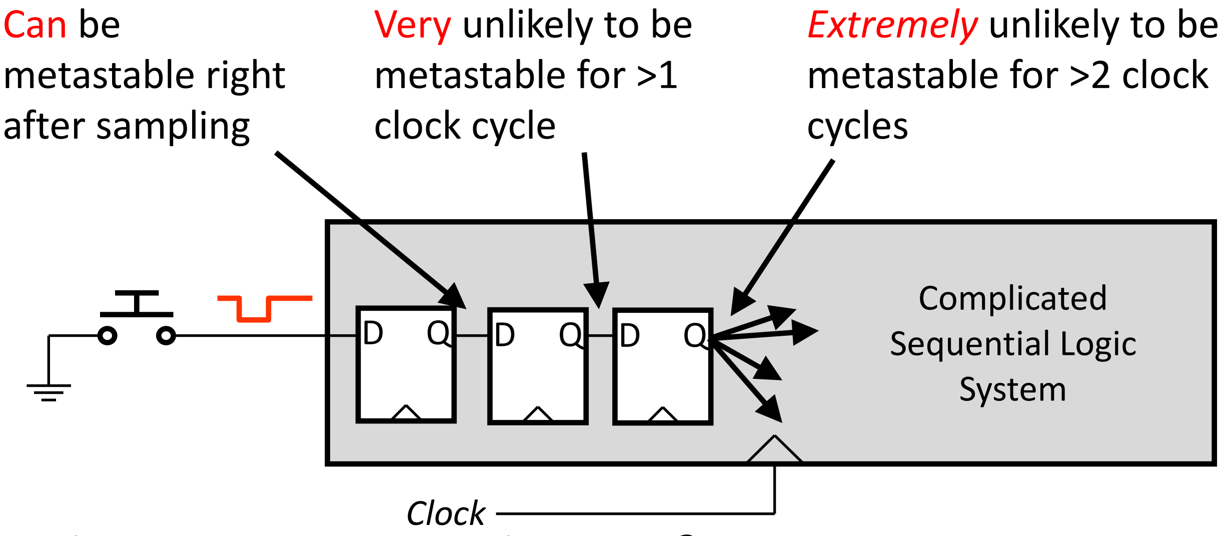
Figure 8. Two-flip-flop synchronization circuit.



"Metastability and Synchronizers: A Tutorial"
Ran Ginosar, Technion Israel Institute of Technology

Handling Metastability

- Completely preventing metastability turns out to be an impossible problem
- High gain of digital devices makes it likely that metastable conditions will resolve themselves quickly
- Solution to metastability: allow time for signals to stabilize

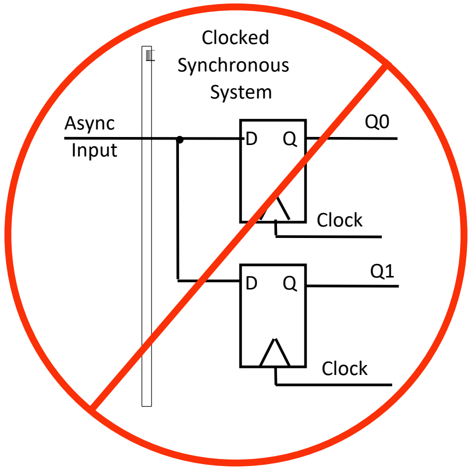

Can be metastable right after sampling
Very unlikely to be metastable for >1 clock cycle
Extremely unlikely to be metastable for >2 clock cycles
D Q
D Q
D Q
Complicated Sequential Logic System
Clock

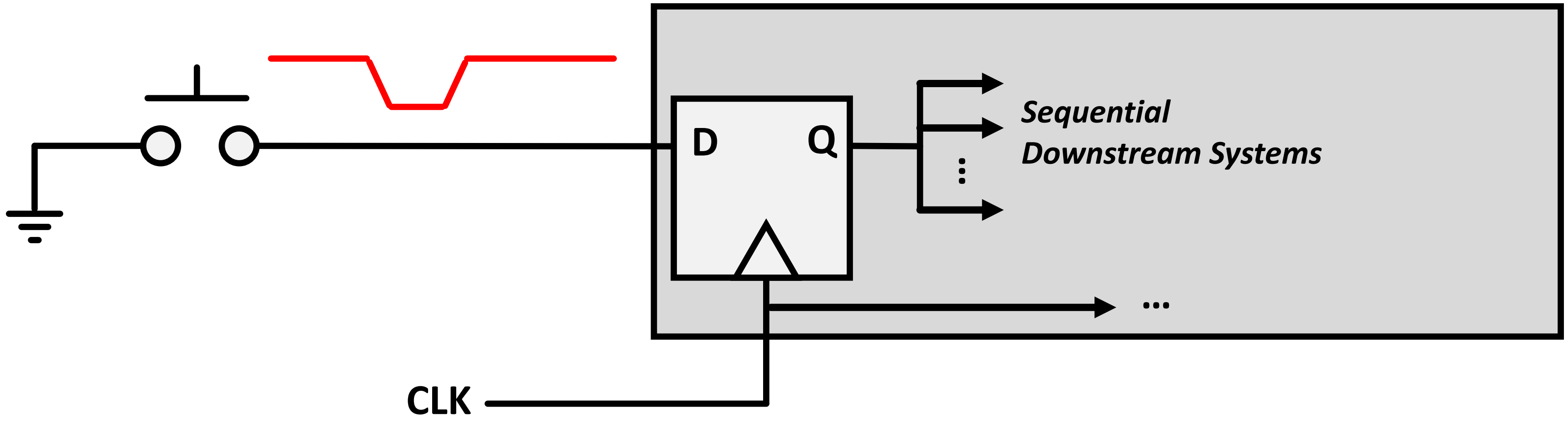
How many registers are necessary in 6.111?

- Depends on many design parameters (clock speed, device speeds, …)
- In 6.111, **a pair of synchronization** registers is sufficient
- And for simple designs…with low t_{pd} you may not even need anything

Handling Metastability

- Don't break off an asynchronous input until it has gone through some registers
- Basically: Ensure that external signals feed <u>**exactly one**</u> flip-flop





Mean Time Between Failures

Figure 1. Metastability Timing Parameters

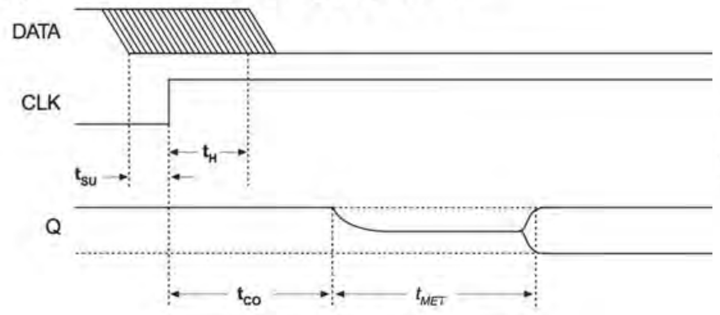
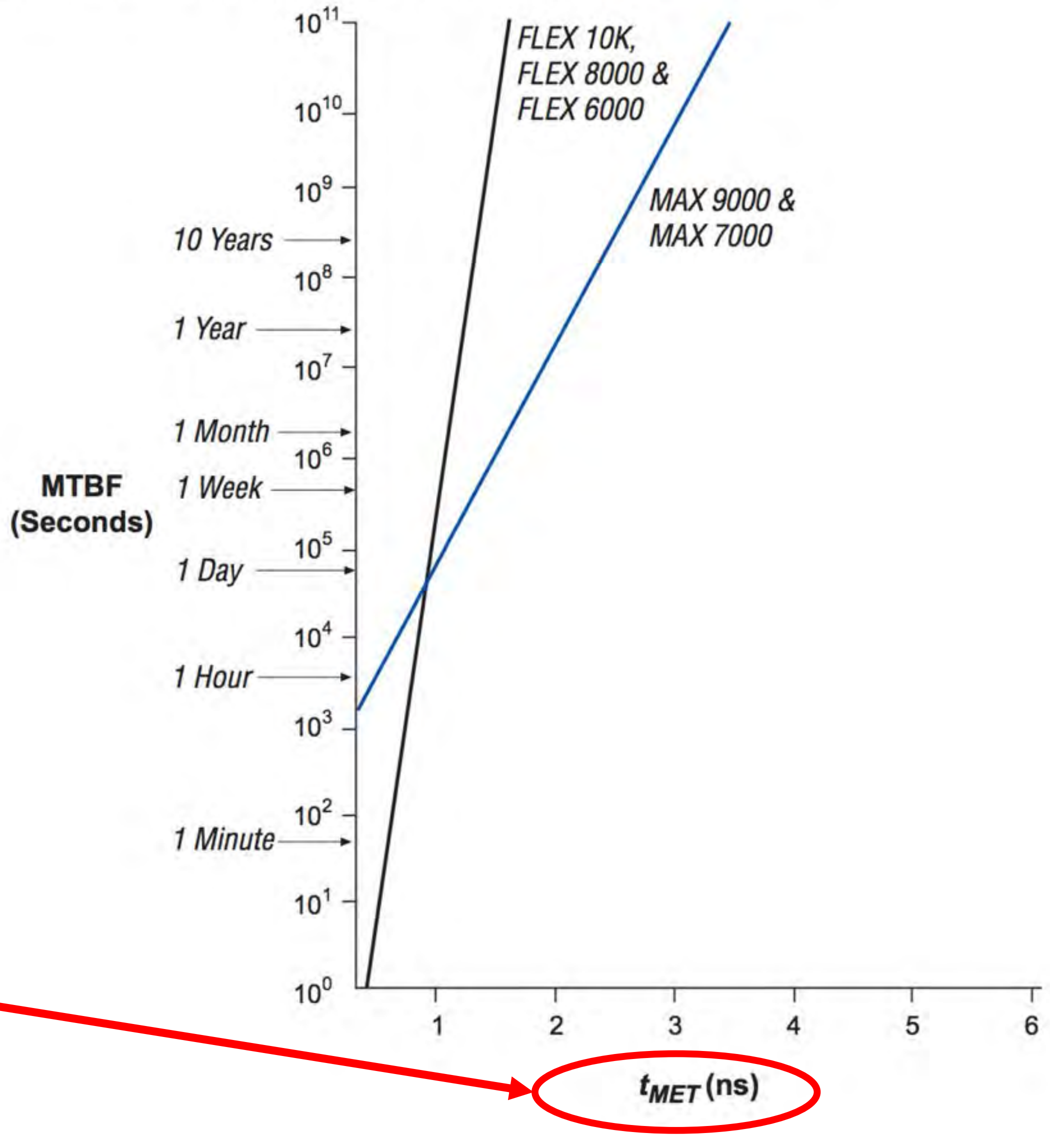


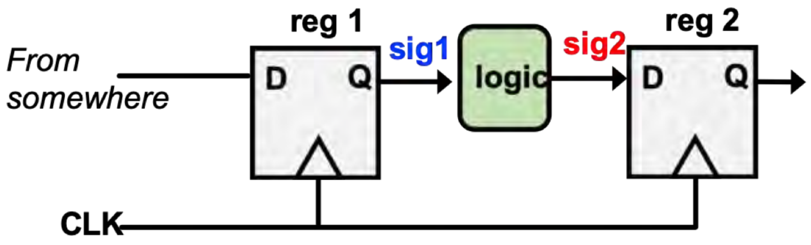
Figure 5. Metastability Characteristics of Altera Devices

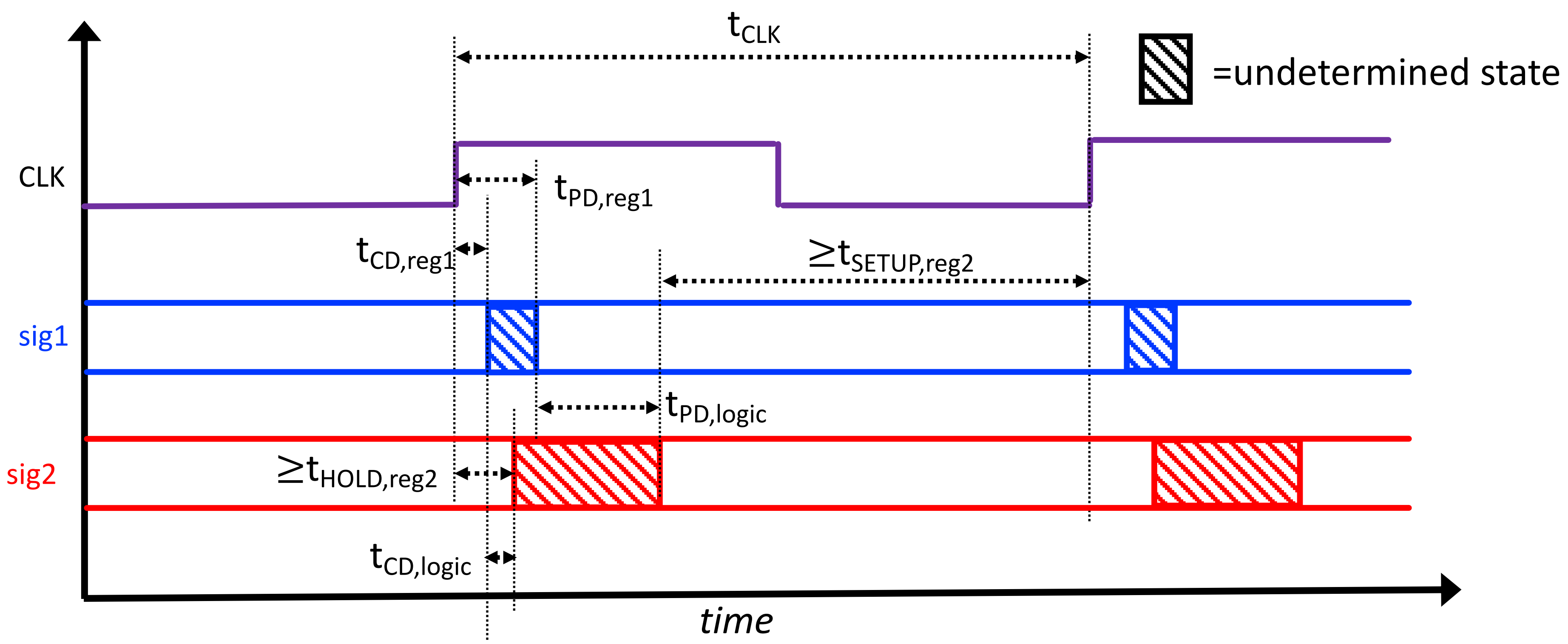


Set by user (how much extra time do you provide per cycle for metastability to dissipate)

Metastability in Altera Devices
Altera Application Note 42 (1999)

D Register Timing 2



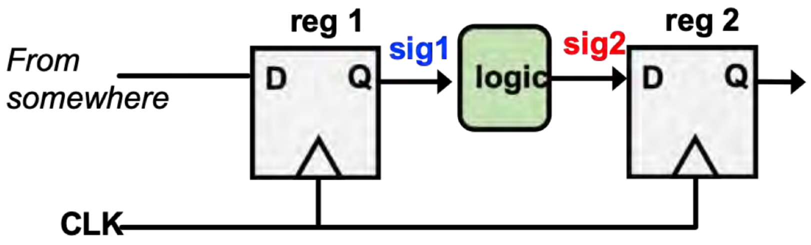


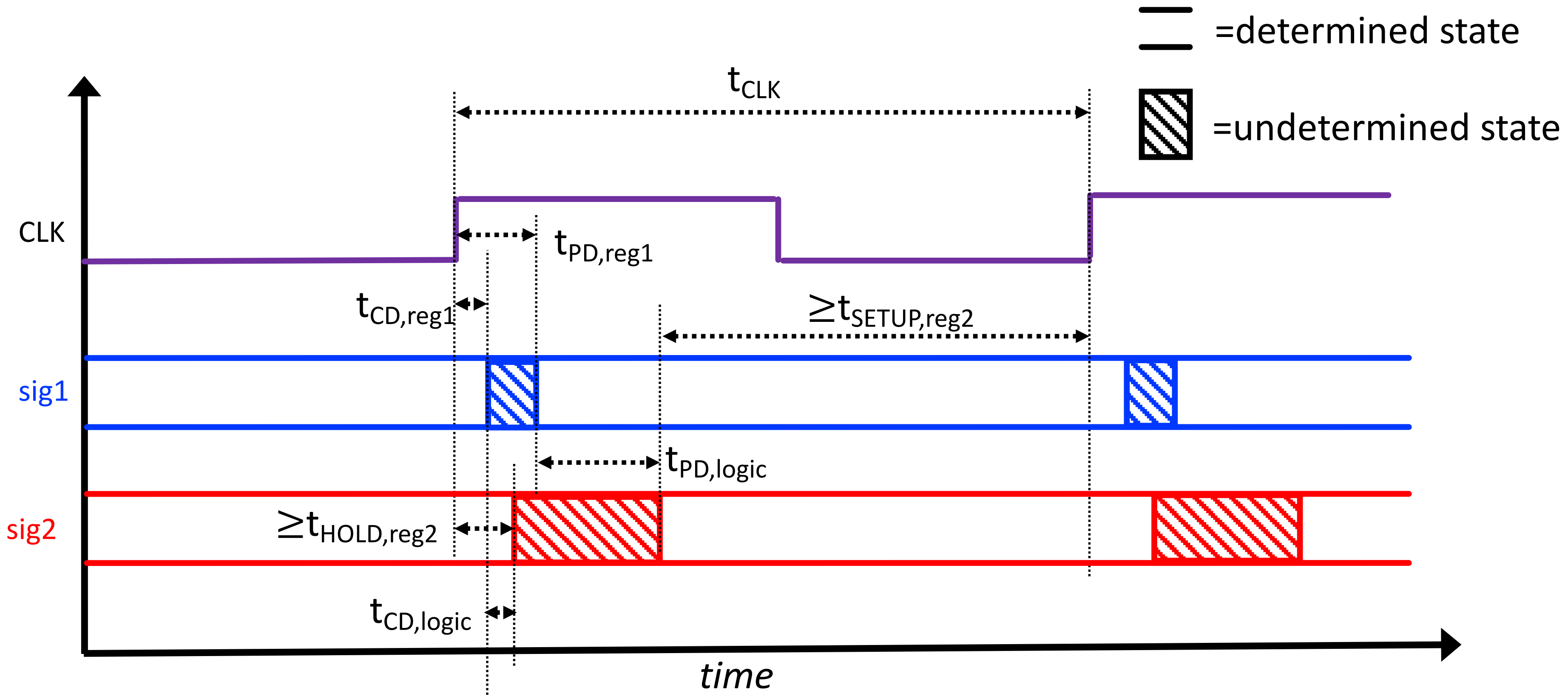
Two Requirements/ Conclusions:

$$t_{PD,reg1} + t_{PD,logic} + t_{SETUP,reg2} \leq t_{CLK}$$

$$t_{CD,reg1} + t_{CD,logic} \geq t_{HOLD,reg2}$$

D Register Timing 2





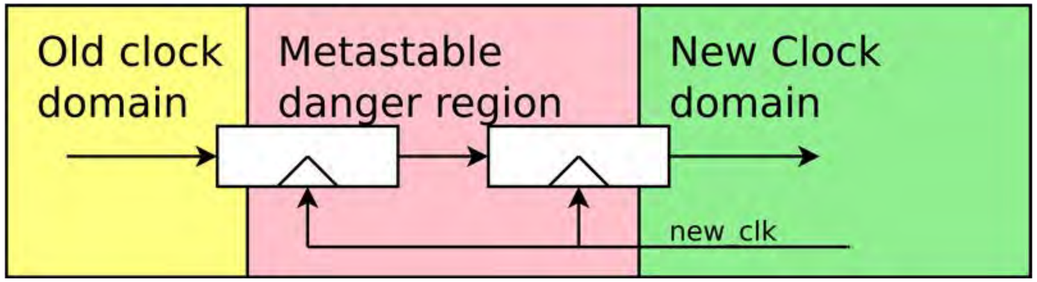
Two Requirements/ Conclusions:

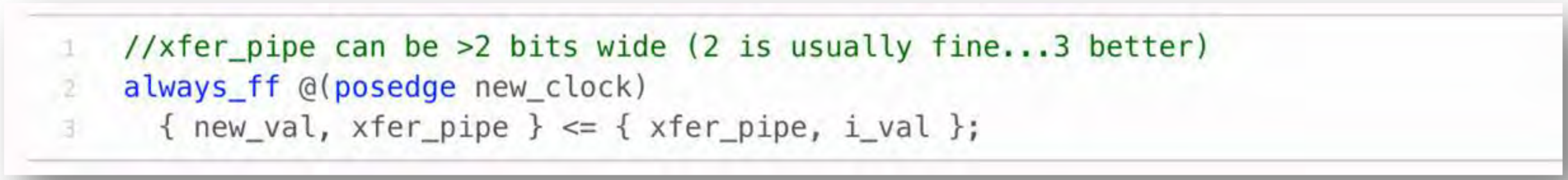
$$t_{PD,reg1} + t_{met} + t_{PD,logic} + t_{SETUP,reg2} \leq t_{CLK}$$

$$t_{CD,reg1} + t_{CD,logic} \geq t_{HOLD,reg2}$$

Clock Domain Crossing

- For example:
  - Data gets sent in at 25 MHz from one device (running on its own clock)
  - Your system runs at 50 MHz





```
//xfer_pipe can be >2 bits wide (2 is usually fine...3 better)
always_ff @(posedge new_clock)
  { new_val, xfer_pipe } <= { xfer_pipe, i_val };
```

- This only works when original clock domain frequency is <= to new clock domain frequency

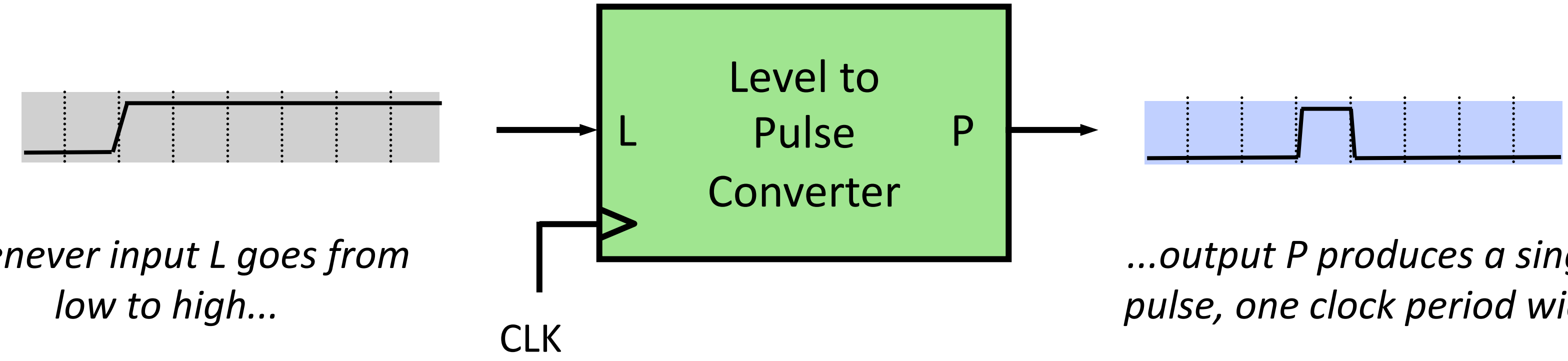
https://zipcpu.com/blog/2017/10/20/cdc.html

FSM Design

…What is a structured way to go about designing state machines?

Design Example: Level-to-Pulse

- A level-to-pulse converter produces a single-cycle pulse each time its input goes high.
- It's a **synchronous** rising-edge detector.
- Sample uses:
  - Buttons and switches pressed by humans for arbitrary periods of time
  - Single-cycle enable signals for counters

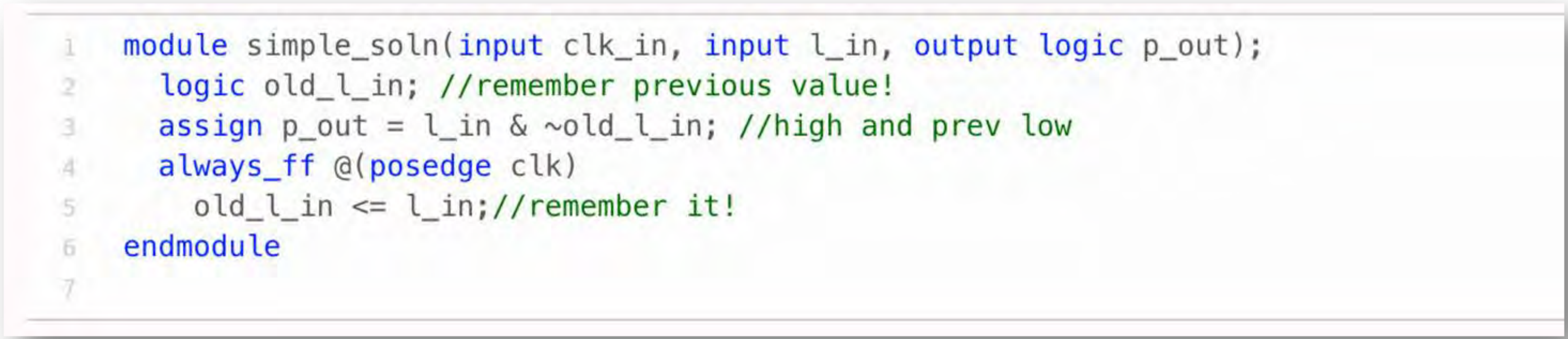


Whenever input L goes from low to high...

...output P produces a single pulse, one clock period wide.

Leve-to-Pulse

- One simple solution (~from Lab 2)
- One bit positive discrete time positive

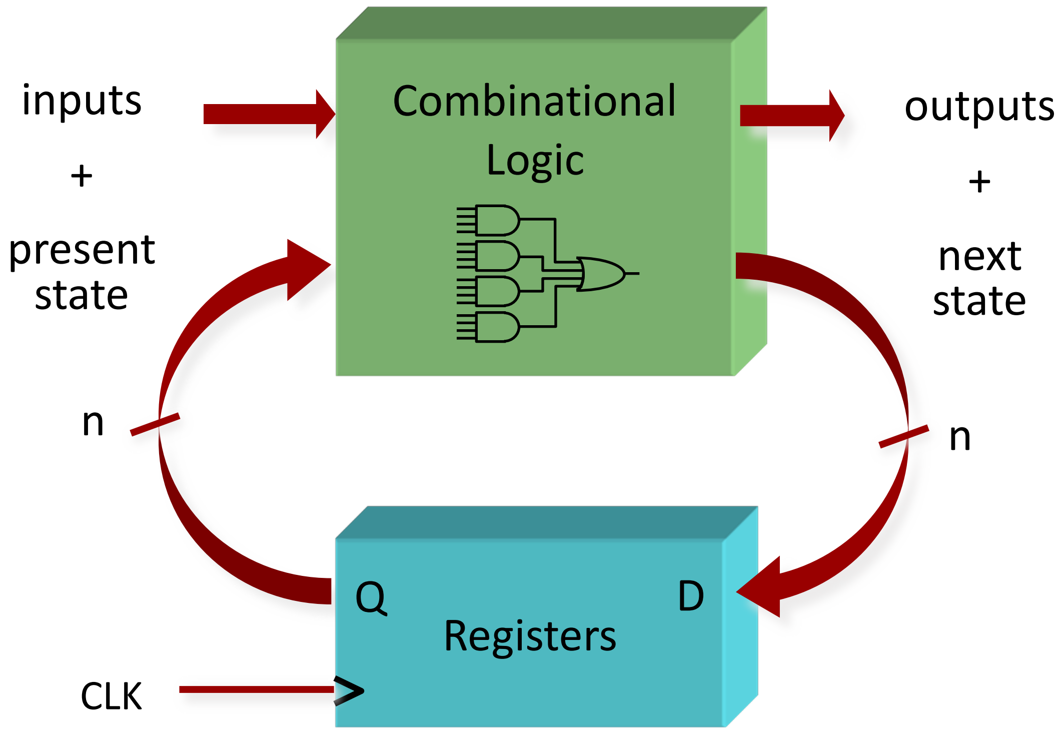


```
module simple_soln(input clk_in, input l_in, output logic p_out);
  logic old_l_in; //remember previous value!
  assign p_out = l_in & ~old_l_in; //high and prev low
  always_ff @(posedge clk)
    old_l_in <= l_in;//remember it!
endmodule
```

- Let’s try to formalize this a bit more

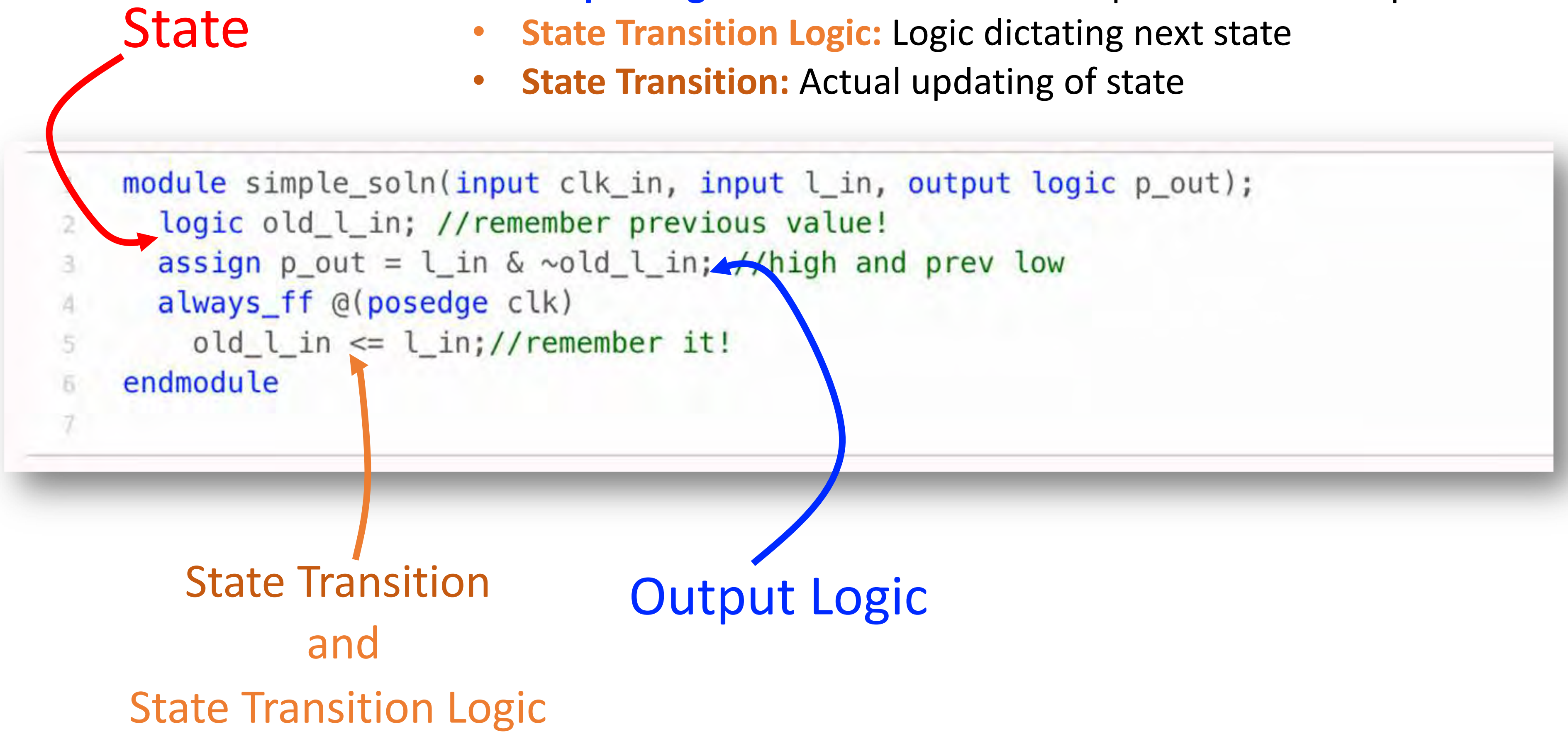
Finite State Machines

- Finite State Machines (FSMs) are a useful abstraction for sequential circuits with centralized "states" of operation
- At each clock edge, combinational logic computes *outputs* and *next state* as a function of *inputs* and *present state*



Level-to-Pulse

- **State:** how/what stores past information?
- **Output Logic:** How does state and input influence output
- **State Transition Logic:** Logic dictating next state
- **State Transition:** Actual updating of state

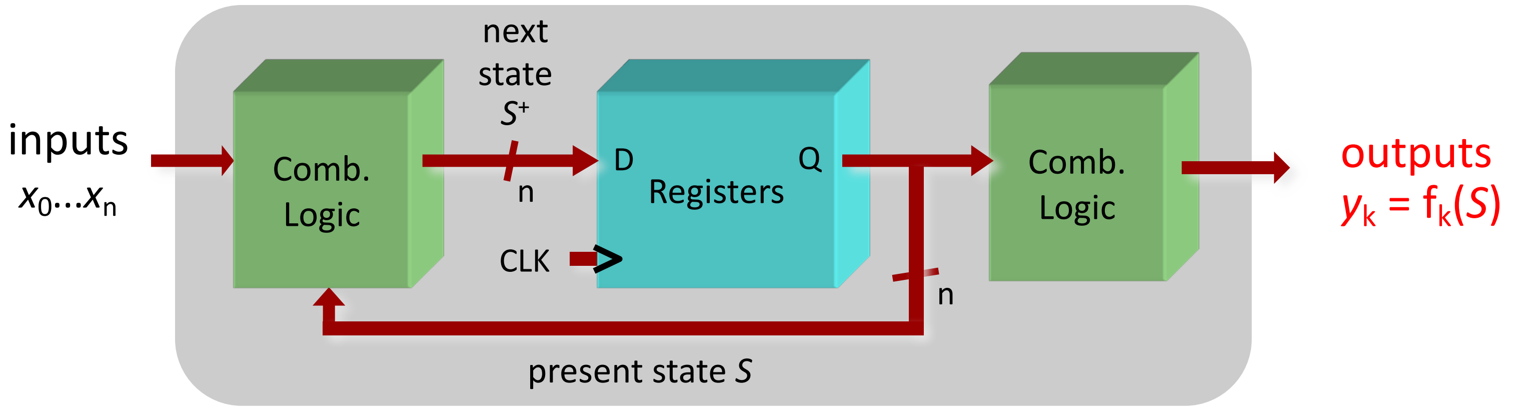


```
module simple_soln(input clk_in, input l_in, output logic p_out);
  logic old_l_in; //remember previous value!
  assign p_out = l_in & ~old_l_in; //high and prev low
  always_ff @(posedge clk)
    old_l_in <= l_in;//remember it!
endmodule
```

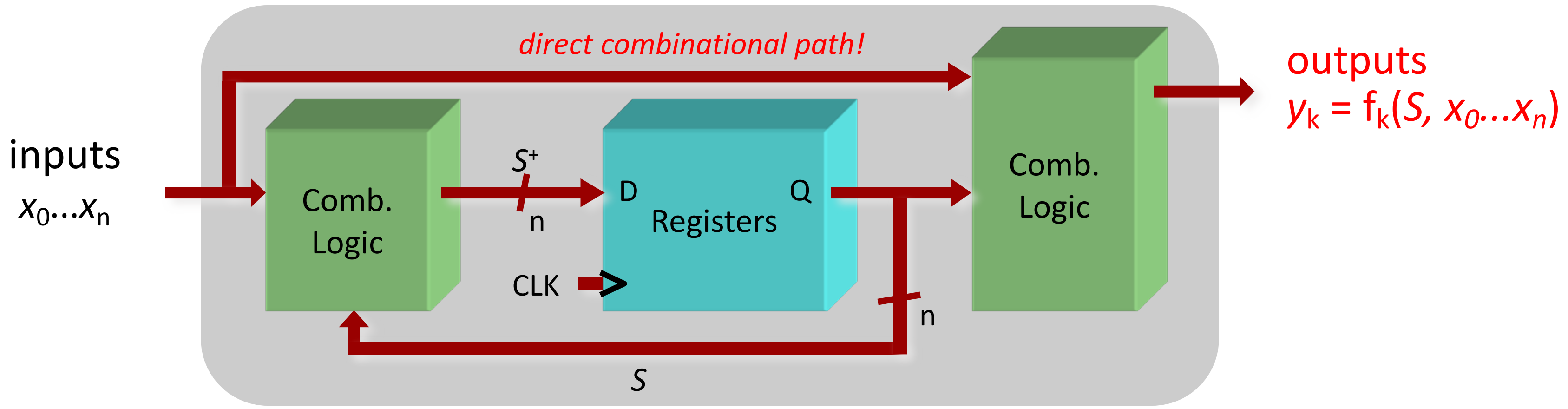
Let’s Formalize it: Two Types of FSMs

Moore and **Mealy** FSMs : different output generation

- Moore FSM:



- Mealy FSM:



Moore

- Edward F. Moore
- 1925-2003
- Virginia Tech
- Worked with Claude Shannon

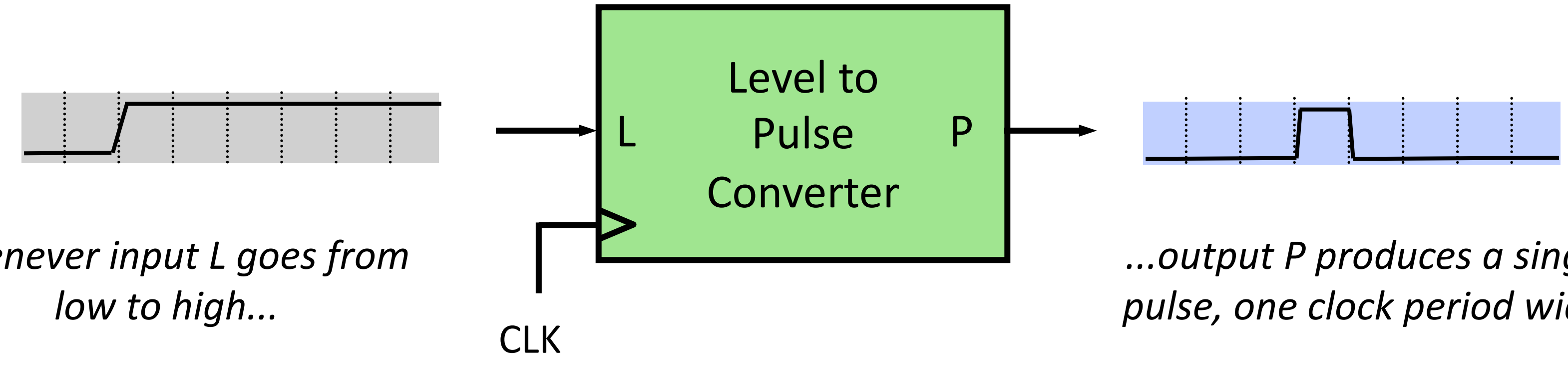
- Not same Moore as Moore's Law…that was Gordon Moore from Intel

Mealy

- George H. Mealy
- 1927-2010
- Harvard, Bell Labs

Design Example: Level-to-Pulse

- A level-to-pulse converter produces a single-cycle pulse each time its input goes high.
- It's a synchronous rising-edge detector.
- Sample uses:
    - Buttons and switches pressed by humans for arbitrary periods of time
    - Single-cycle enable signals for counters

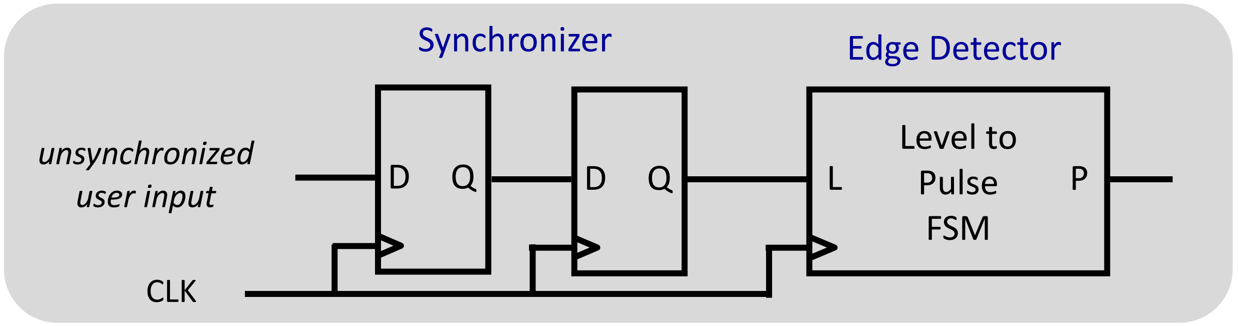


Whenever input L goes from low to high...

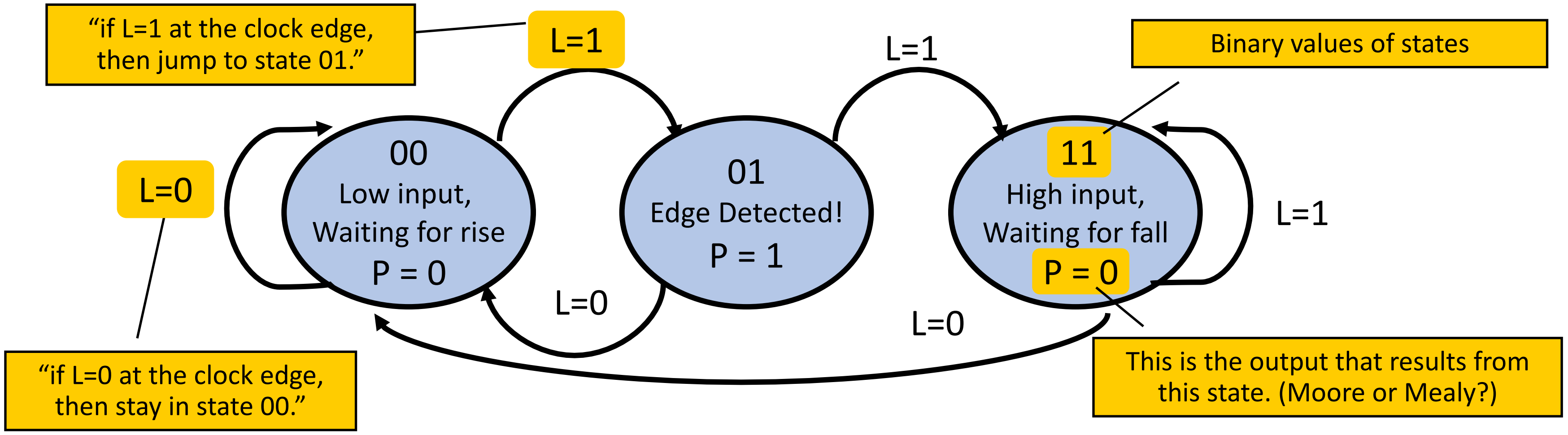
...output P produces a single pulse, one clock period wide.

Step 1: State Transition Diagram

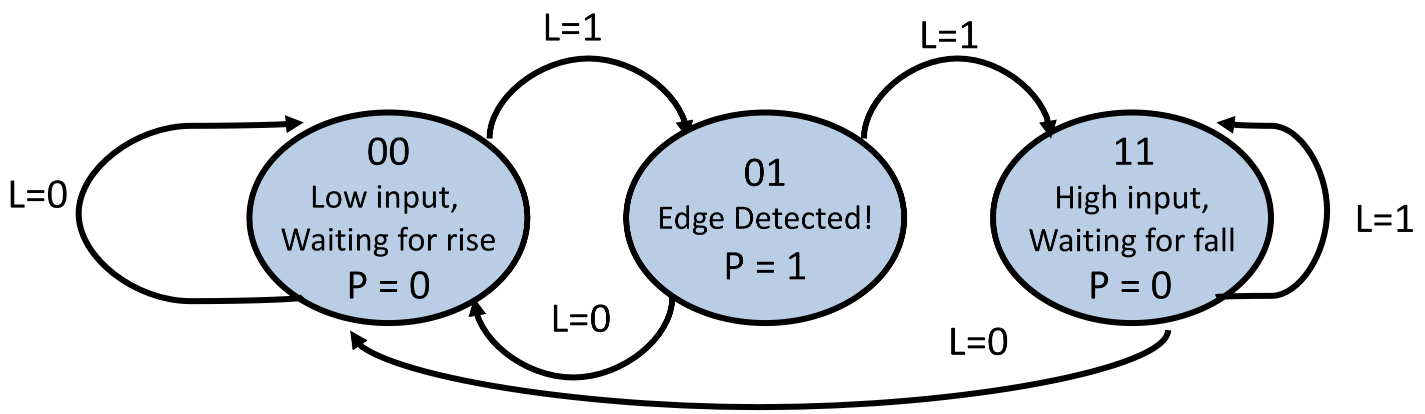
- Block diagram of desired system:



- State transition diagram is a useful FSM representation and design aid:


"if L=1 at the clock edge, then jump to state 01."
L=1
L=1
Binary values of states
L=0
00 Low input, Waiting for rise P = 0
01 Edge Detected! P = 1
11 High input, Waiting for fall P = 0
L=1
L=0
L=0
"if L=0 at the clock edge, then stay in state 00."
This is the output that results from this state. (Moore or Mealy?)

Valid State Transition Diagrams



- Arcs leaving a state are mutually exclusive, i.e., for any combination input values there's at most one applicable arc
- Arcs leaving a state are collectively exhaustive, i.e., for any combination of input values there's at least one applicable arc**
- So for each state: for any combination of input values there's exactly one applicable arc (no ambiguity)
- Often a starting state is specified
- Each state specifies values for all outputs (in the case of Moore)

Choosing State Representation

Choice #1: binary encoding

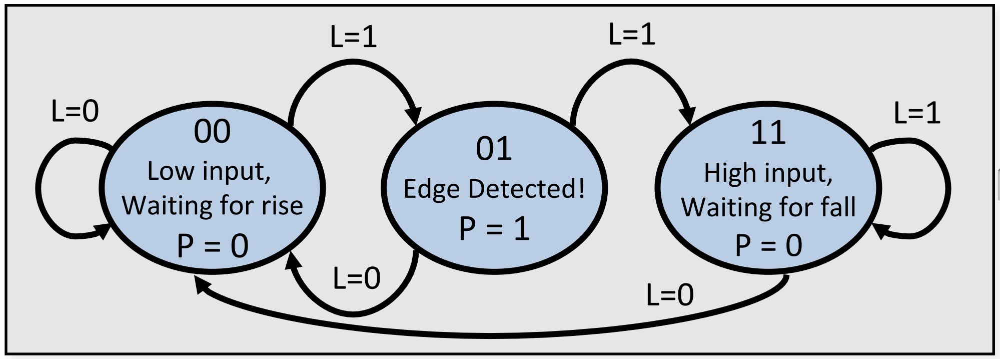
For N states, use ceil($\log_2 N$) bits to encode the state with each state represented by a unique combination of the bits. Tradeoffs: most efficient use of state registers, but requires more complicated combinational logic to detect when in a particular state.

Choice #2: "one-hot" encoding

For N states, use N bits to encode the state where the bit corresponding to the current state is 1, all the others 0. Tradeoffs: more state registers, but often much less combinational logic since state decoding is trivial.

Step 2: Logic Derivation

Transition diagram is readily converted to a state transition table (just a truth table)



L=0
L=1
L=1
L=1
00 Low input, Waiting for rise P = 0
01 Edge Detected! P = 1
11 High input, Waiting for fall P = 0
L=0
L=0

Current State		In	Next State		Out
S_1	S_0	L	S_1^+	S_0^+	P
0	0	0	0	0	0
0	0	1	0	1	0
0	1	0	0	0	1
0	1	1	1	1	1
1	1	0	0	0	0
1	1	1	1	1	0

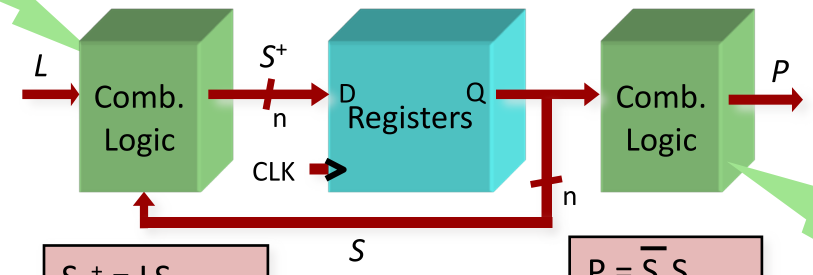
- Combinational logic ***could*** be derived using Karnaugh maps, but we'll let an HDL do that for us

for S_1^+:

L \ S_1S_0	00	01	11	10
0	0	0	0	X
1	0	1	1	X

for S_0^+:

L \ S_1S_0	00	01	11	10
0	0	0	0	X
1	1	1	1	X



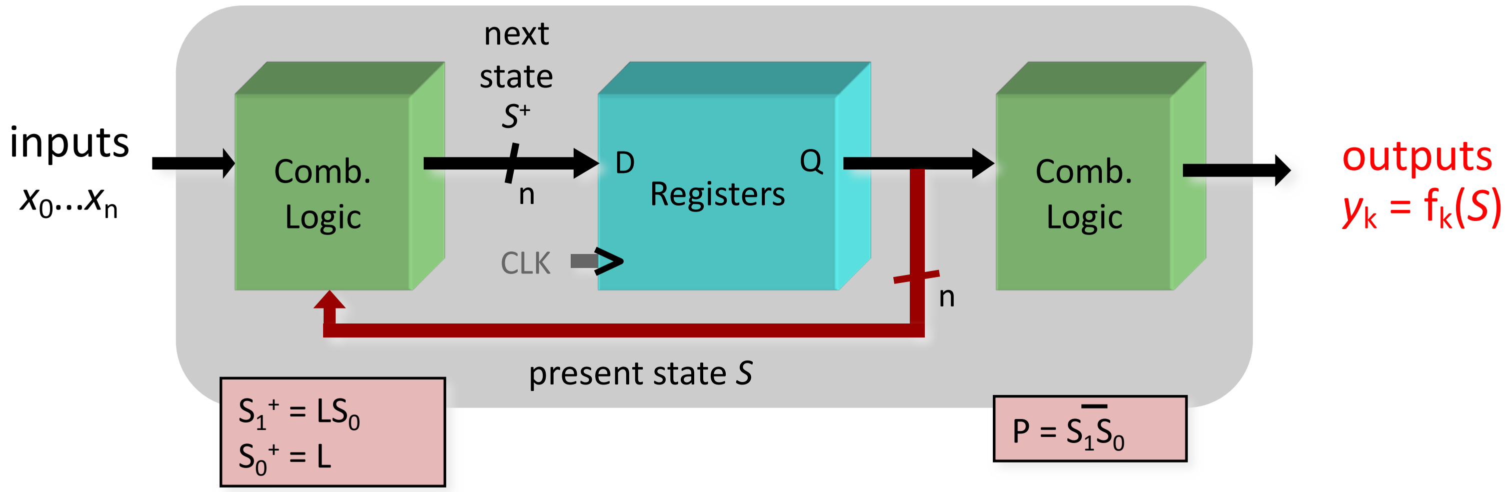
$S_1^+ = LS_0$
$S_0^+ = L$

$P = \overline{S_1}S_0$

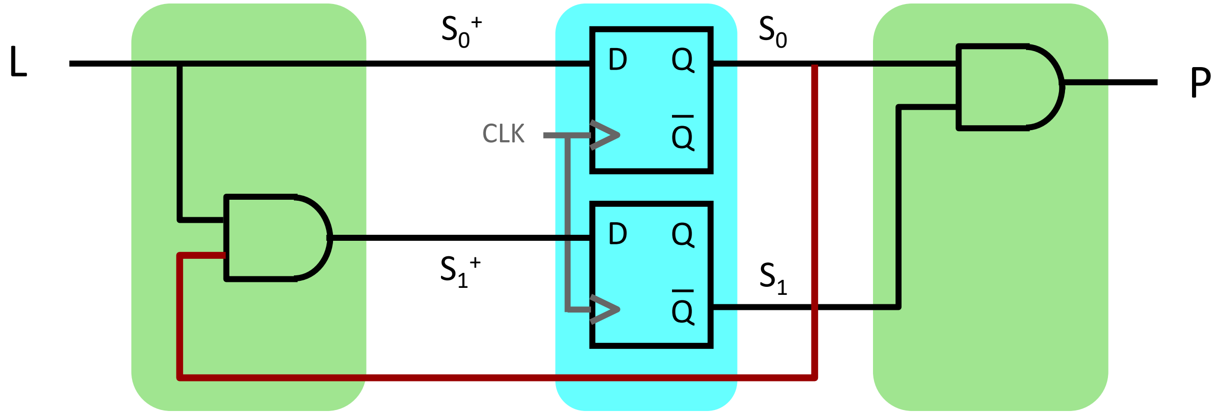
for P:

S_0 \ S_1	0	1
0	0	X
1	1	0

Moore Level-to-Pulse Converter

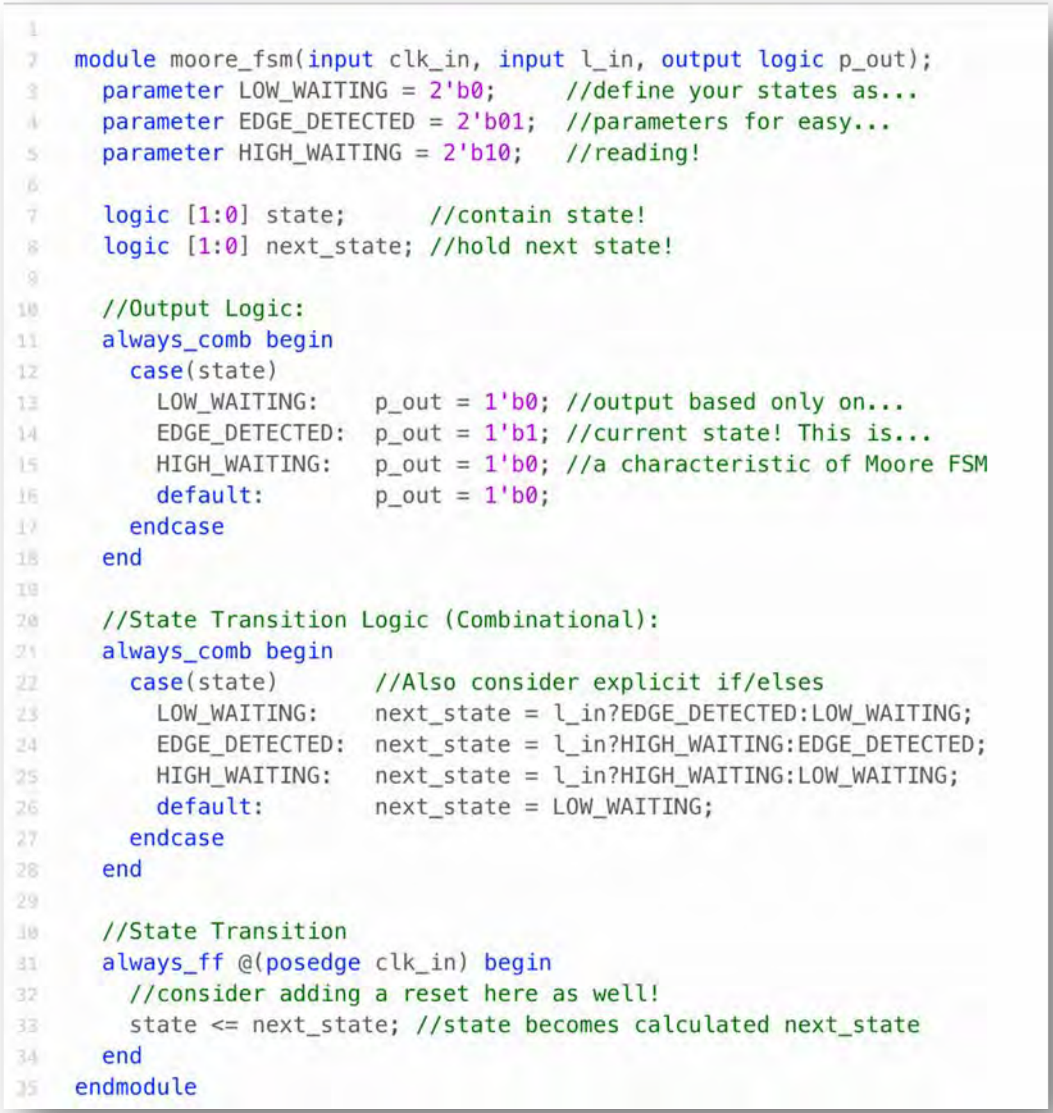


Moore FSM circuit implementation of level-to-pulse converter:



Moore Level-to-Pulse Converter (SystemVerilog)

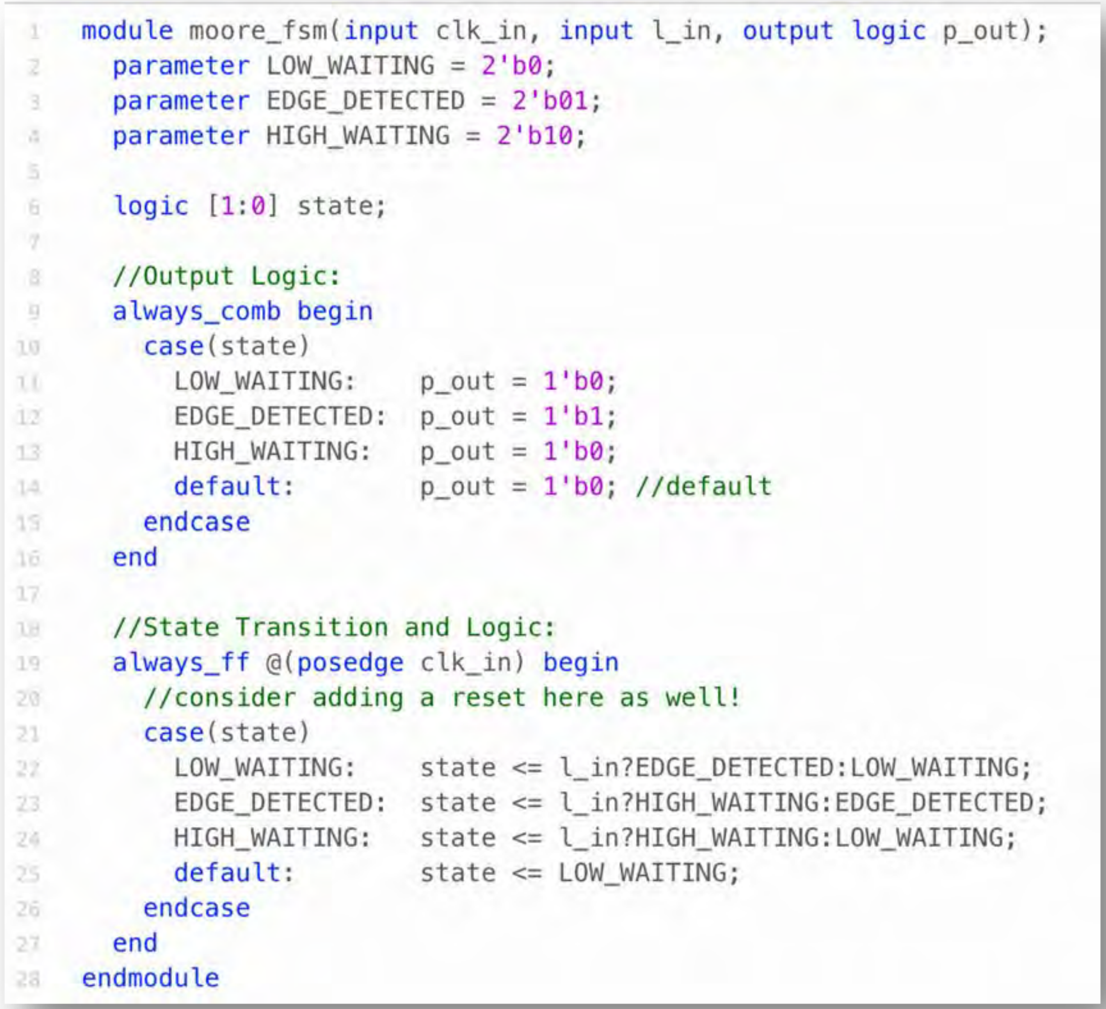
- An example of a **very explicit** Moore FSM implementation of the level-to-pulse converter:



```
module moore_fsm(input clk_in, input l_in, output logic p_out);
  parameter LOW_WAITING = 2'b0;     //define your states as...
  parameter EDGE_DETECTED = 2'b01;  //parameters for easy...
  parameter HIGH_WAITING = 2'b10;   //reading!

  logic [1:0] state;      //contain state!
  logic [1:0] next_state; //hold next state!

  //Output Logic:
  always_comb begin
    case(state)
      LOW_WAITING:    p_out = 1'b0; //output based only on...
      EDGE_DETECTED:  p_out = 1'b1; //current state! This is...
      HIGH_WAITING:   p_out = 1'b0; //a characteristic of Moore FSM
      default:        p_out = 1'b0;
    endcase
  end

  //State Transition Logic (Combinational):
  always_comb begin
    case(state)       //Also consider explicit if/elses
      LOW_WAITING:    next_state = l_in?EDGE_DETECTED:LOW_WAITING;
      EDGE_DETECTED:  next_state = l_in?HIGH_WAITING:EDGE_DETECTED;
      HIGH_WAITING:   next_state = l_in?HIGH_WAITING:LOW_WAITING;
      default:        next_state = LOW_WAITING;
    endcase
  end

  //State Transition
  always_ff @(posedge clk_in) begin
    //consider adding a reset here as well!
    state <= next_state; //state becomes calculated next_state
  end
endmodule
```

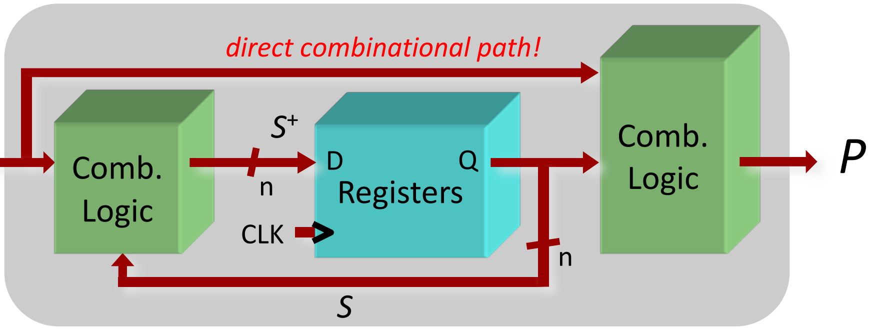
Moore Level-to-Pulse Converter (SystemVerilog)

- Merging **State Transition Logic** and **State Transition** into one block
- Some people like this more (me)



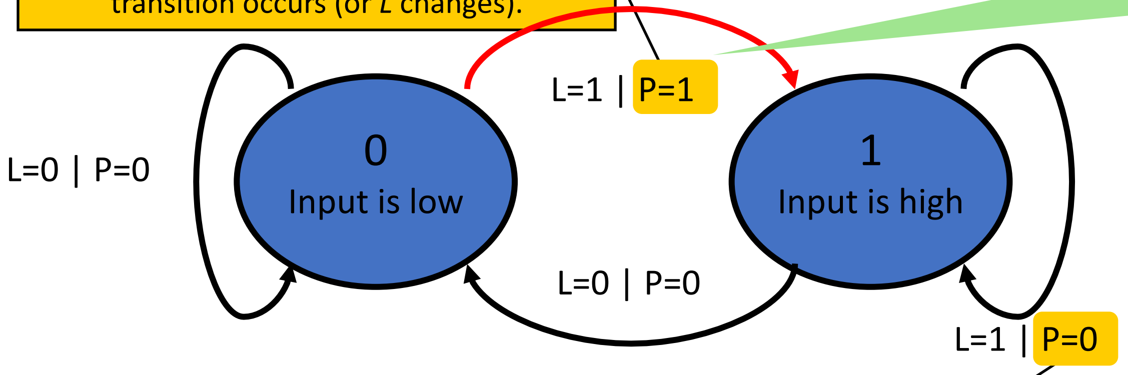
```
module moore_fsm(input clk_in, input l_in, output logic p_out);
  parameter LOW_WAITING = 2'b0;
  parameter EDGE_DETECTED = 2'b01;
  parameter HIGH_WAITING = 2'b10;

  logic [1:0] state;

  //Output Logic:
  always_comb begin
    case(state)
      LOW_WAITING:    p_out = 1'b0;
      EDGE_DETECTED:  p_out = 1'b1;
      HIGH_WAITING:   p_out = 1'b0;
      default:        p_out = 1'b0; //default
    endcase
  end

  //State Transition and Logic:
  always_ff @(posedge clk_in) begin
    //consider adding a reset here as well!
    case(state)
      LOW_WAITING:    state <= l_in?EDGE_DETECTED:LOW_WAITING;
      EDGE_DETECTED:  state <= l_in?HIGH_WAITING:EDGE_DETECTED;
      HIGH_WAITING:   state <= l_in?HIGH_WAITING:LOW_WAITING;
      default:        state <= LOW_WAITING;
    endcase
  end
endmodule
```

Design of a Mealy Level-to-Pulse

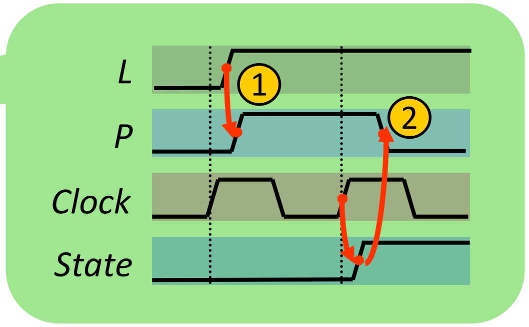


- Since outputs are determined by state *and* inputs, Mealy FSMs may need fewer states than Moore FSM implementations

1. When *L*=1 and *S*=0, this output is asserted immediately and until the state transition occurs (or *L* changes).



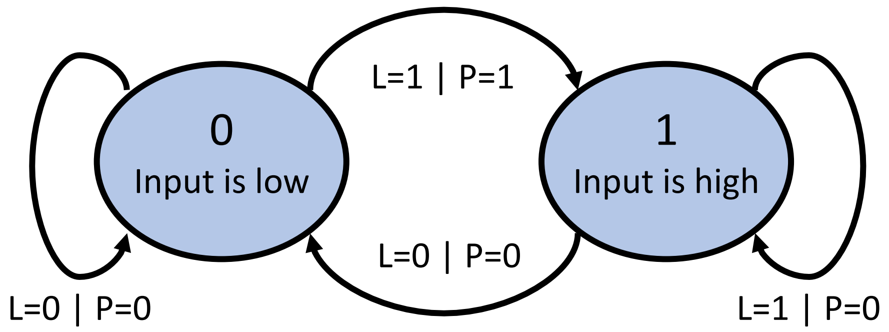
2. While in state S=1 and as long as *L* remains at *1, this* output is asserted until next clock.



Output transitions immediately.

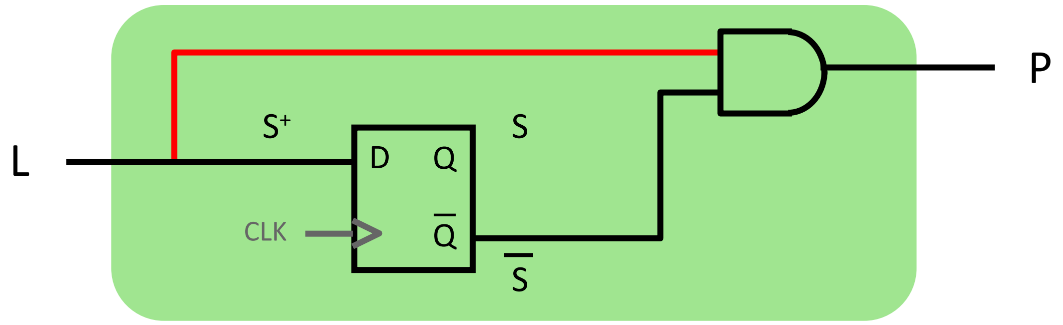
State transitions at the clock edge.

Mealy Level-to-Pulse Converter



Pres. State	In	Next State	Out
S	L	S^+	P
0	0	0	0
0	1	1	1
1	1	1	0
1	0	0	0

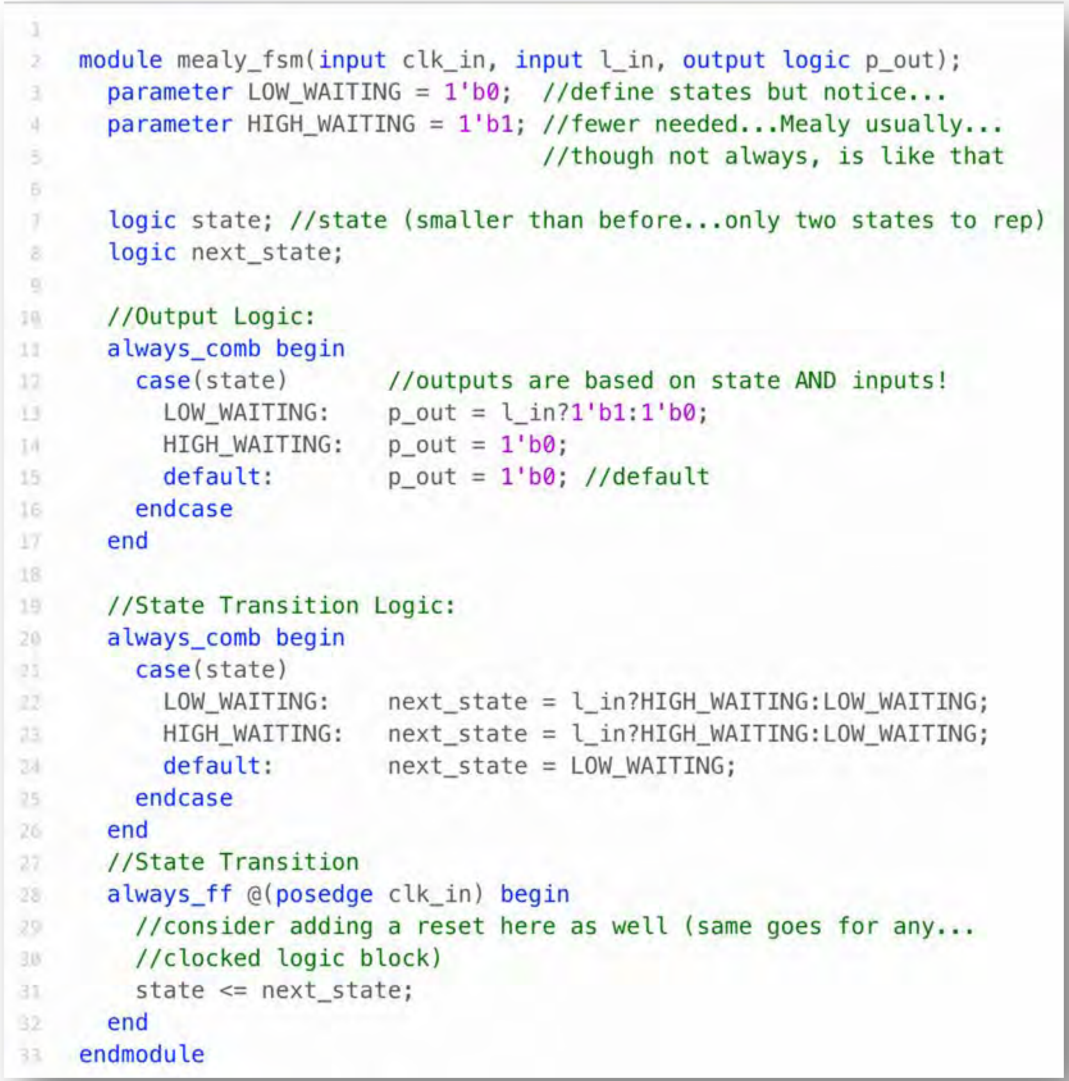
Mealy FSM circuit implementation of level-to-pulse converter:



- FSM’s state simply remembers the previous value of L
- Circuit benefits from the Mealy FSM’s implicit single-cycle assertion of outputs during state transitions

Mealy Level-to-Pulse Converter (SystemVerilog)

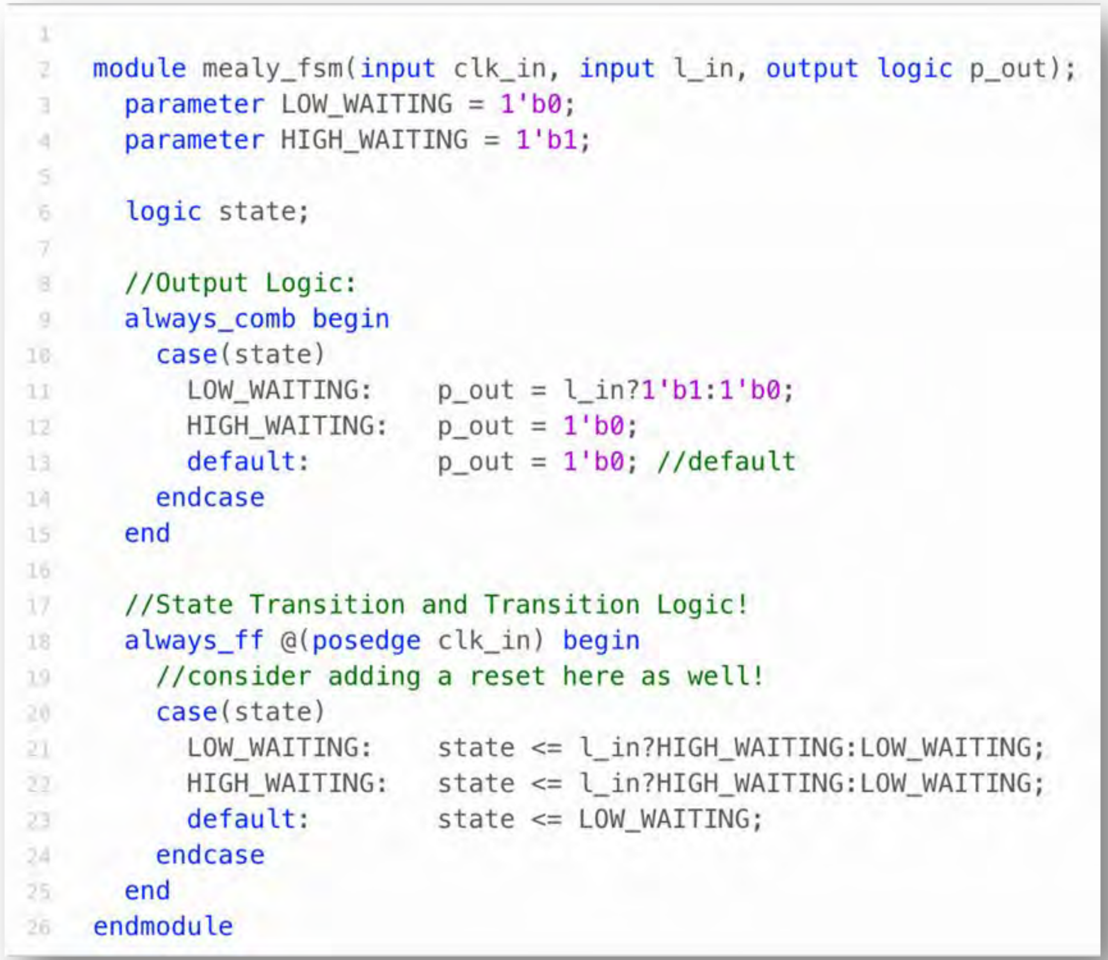
- An example of a **very explicit** Mealy FSM implementation of the level-to-pulse converter:



```
module mealy_fsm(input clk_in, input l_in, output logic p_out);
  parameter LOW_WAITING = 1'b0;  //define states but notice...
  parameter HIGH_WAITING = 1'b1; //fewer needed...Mealy usually...
                                 //though not always, is like that

  logic state; //state (smaller than before...only two states to rep)
  logic next_state;

  //Output Logic:
  always_comb begin
    case(state)         //outputs are based on state AND inputs!
      LOW_WAITING:      p_out = l_in?1'b1:1'b0;
      HIGH_WAITING:     p_out = 1'b0;
      default:          p_out = 1'b0; //default
    endcase
  end

  //State Transition Logic:
  always_comb begin
    case(state)
      LOW_WAITING:      next_state = l_in?HIGH_WAITING:LOW_WAITING;
      HIGH_WAITING:     next_state = l_in?HIGH_WAITING:LOW_WAITING;
      default:          next_state = LOW_WAITING;
    endcase
  end
  //State Transition
  always_ff @(posedge clk_in) begin
    //consider adding a reset here as well (same goes for any...
    //clocked logic block)
    state <= next_state;
  end
endmodule
```

Mealy Level-to-Pulse Converter (SystemVerilog)

- Merging **State Transition Logic** and **State Transition** into one block

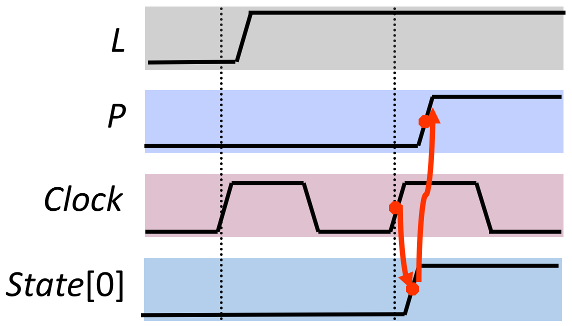


```
module mealy_fsm(input clk_in, input l_in, output logic p_out);
  parameter LOW_WAITING = 1'b0;
  parameter HIGH_WAITING = 1'b1;

  logic state;

  //Output Logic:
  always_comb begin
    case(state)
      LOW_WAITING:    p_out = l_in?1'b1:1'b0;
      HIGH_WAITING:   p_out = 1'b0;
      default:        p_out = 1'b0; //default
    endcase
  end

  //State Transition and Transition Logic!
  always_ff @(posedge clk_in) begin
    //consider adding a reset here as well!
    case(state)
      LOW_WAITING:    state <= l_in?HIGH_WAITING:LOW_WAITING;
      HIGH_WAITING:   state <= l_in?HIGH_WAITING:LOW_WAITING;
      default:        state <= LOW_WAITING;
    endcase
  end
endmodule
```

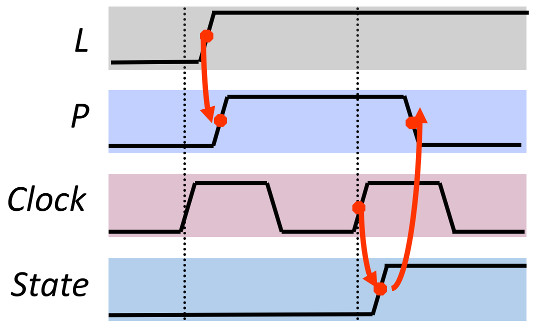
Moore/Mealy Trade-Offs

- How are they different?
  - Moore: outputs = f(state) only
  - Mealy outputs = f(state *and input*)
  - Mealy outputs generally occur one cycle earlier than a Moore:

Moore: delayed assertion of P



Mealy: immediate assertion of P

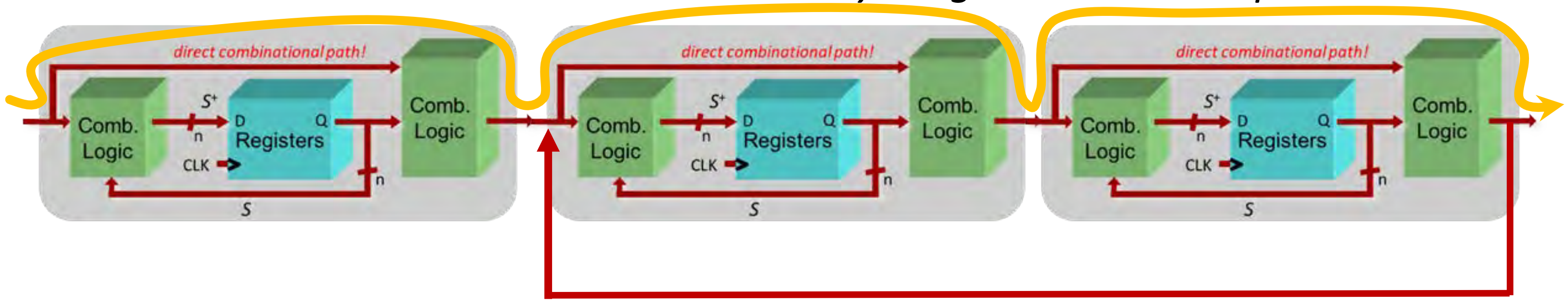


- Compared to a Moore FSM, a Mealy FSM ***might***...
  - Be more difficult to conceptualize and design (both at circuit level and in HDL)
  - Have fewer states
  - Be expressed using fewer lines of Verilog

Moore/Mealy Trade-Offs

- Moore:
  - Usually more states
  - Each state has a particular output
- Mealy:
  - Fewer states, outputs are specified on edges of diagram
  - Potential Dangers:

Really-long combinatorial paths!

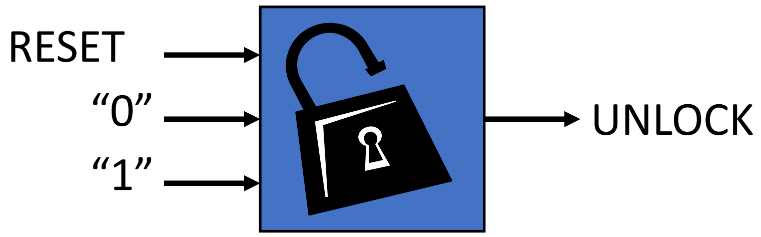


Possible cyclic logic paths
Combinatorial logic driving itself asynchronously through really hard-to-debug pathways!

FSM Example

GOAL:

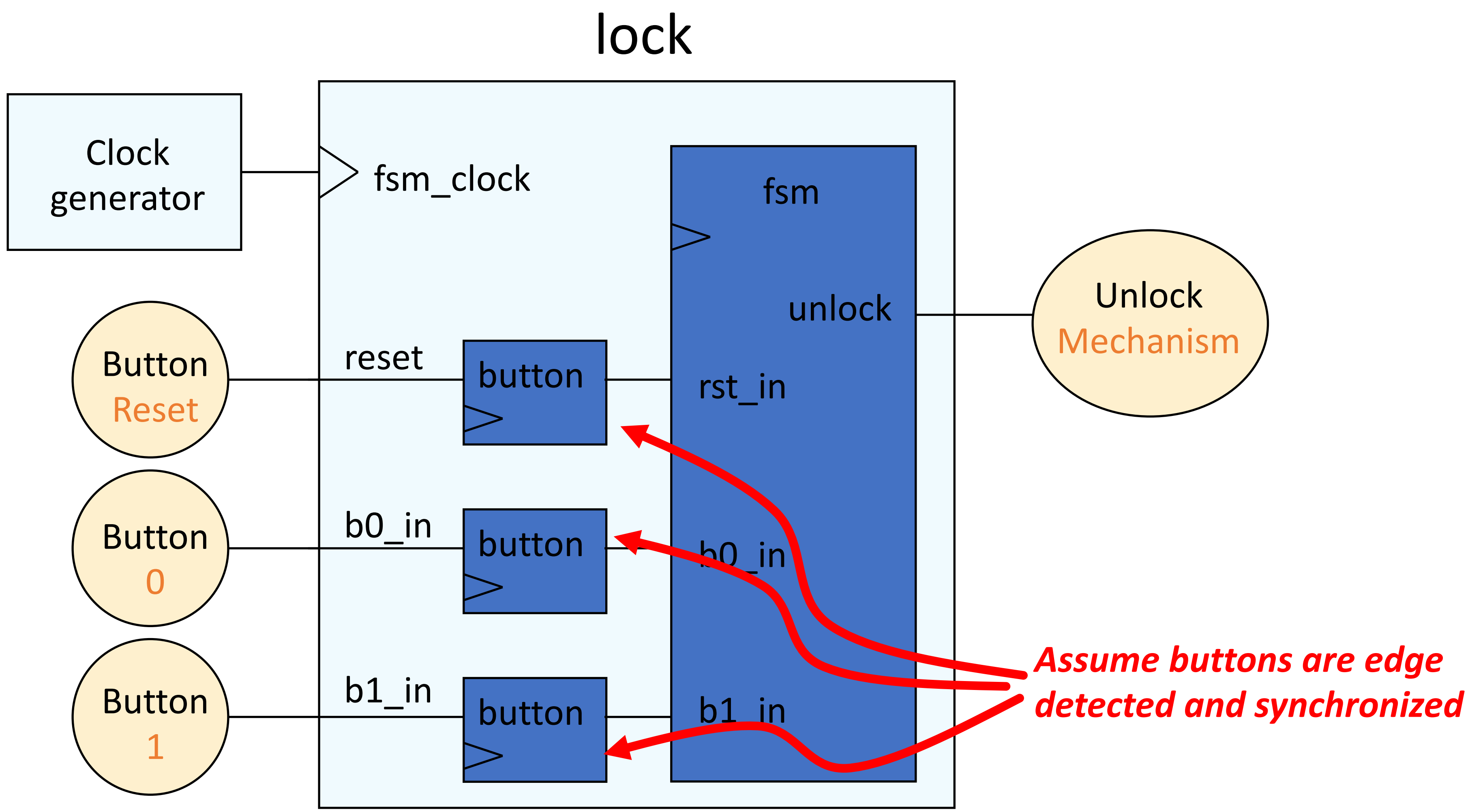
- Build an electronic combination lock with a reset button, two number buttons (0 and 1), and an unlock output signal. The combination will **always be 01011**.
- Use a sliding window of the last five entries



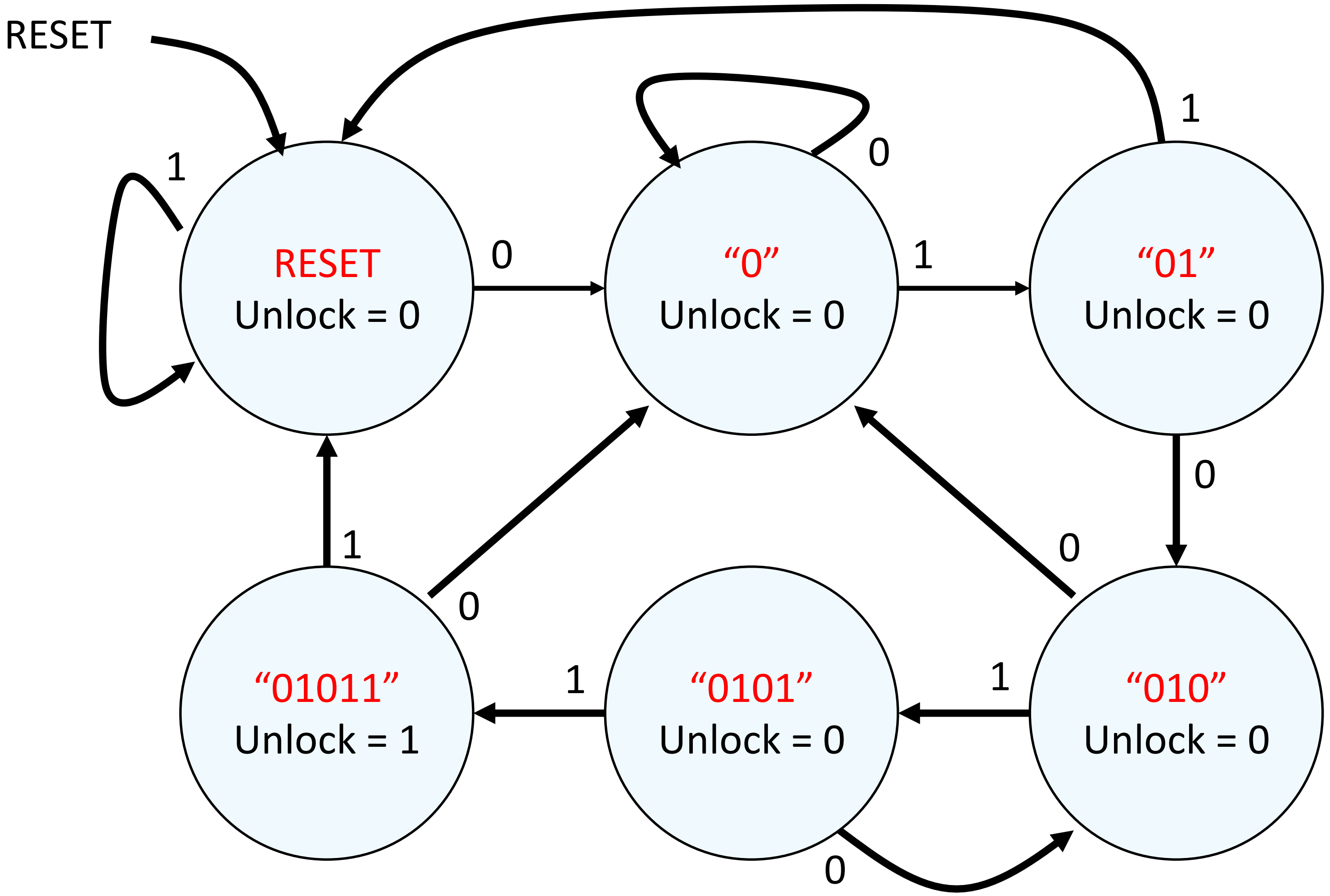
STEPS:

1. Design lock FSM (block diagram, state transitions)
2. Write SystemVerilog module(s) for FSM

Step 1A: Block Diagram

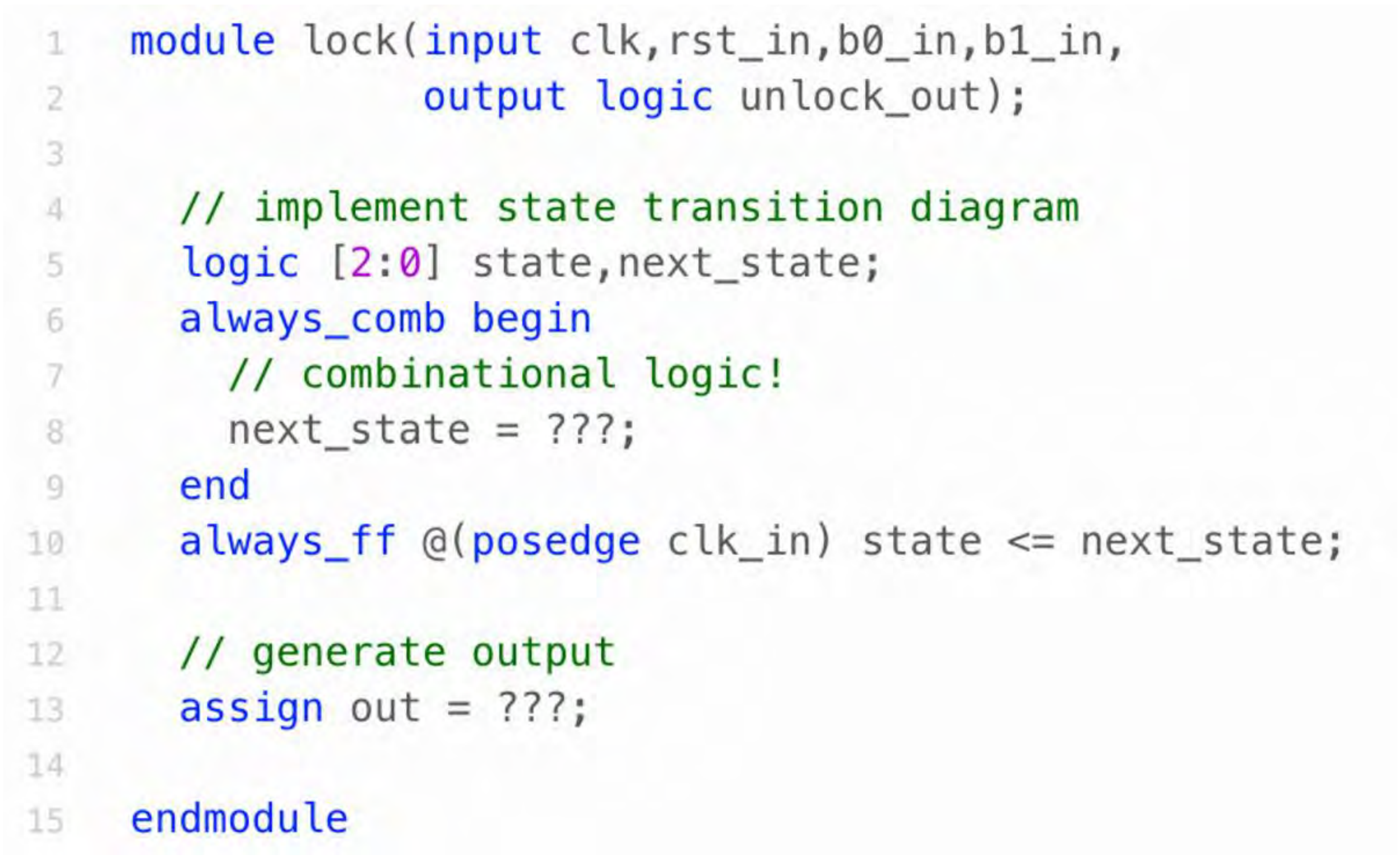


Step 1B: State transition diagram



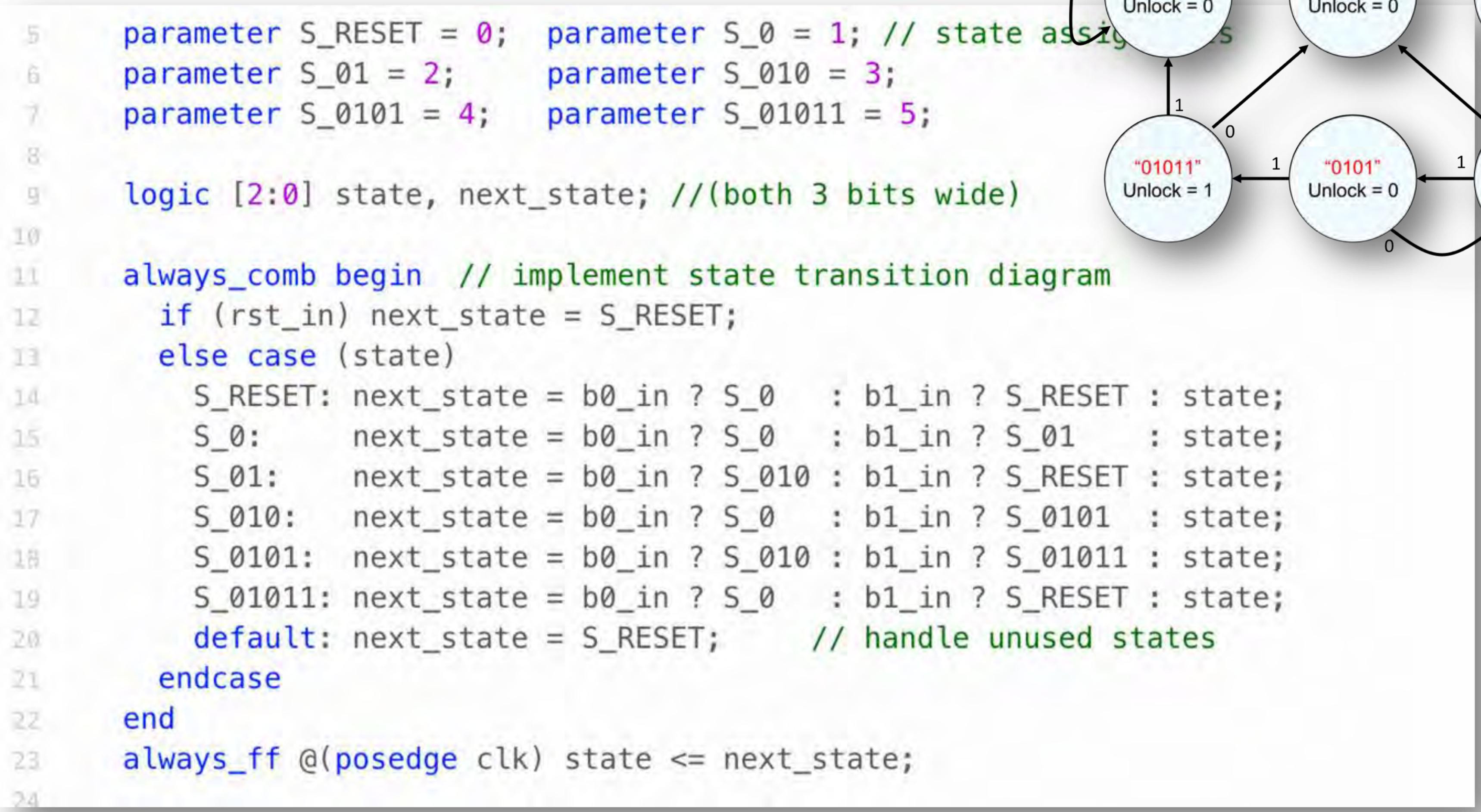
6 states → 3 bits

Step 2: Write Verilog

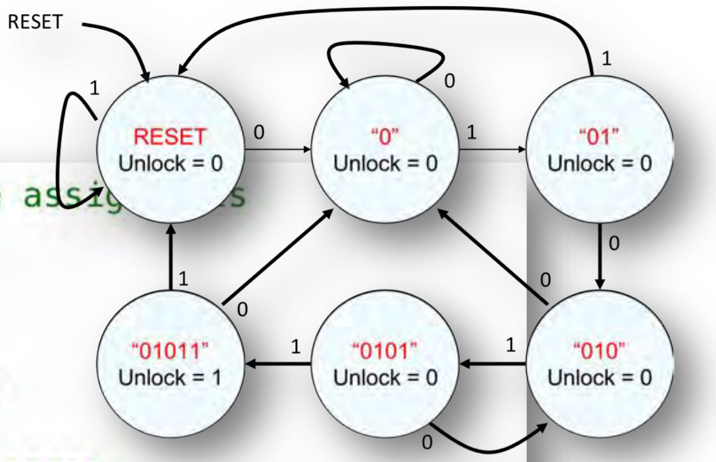


```
module lock(input clk,rst_in,b0_in,b1_in,
            output logic unlock_out);

  // implement state transition diagram
  logic [2:0] state,next_state;
  always_comb begin
    // combinational logic!
    next_state = ???;
  end
  always_ff @(posedge clk_in) state <= next_state;

  // generate output
  assign out = ???;

endmodule
```

Step 2B: state transition diagram

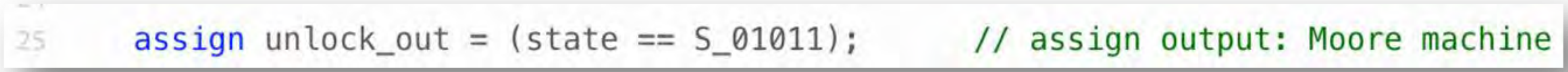


```
parameter S_RESET = 0;  parameter S_0 = 1; // state assig
parameter S_01 = 2;     parameter S_010 = 3;
parameter S_0101 = 4;   parameter S_01011 = 5;

logic [2:0] state, next_state; //(both 3 bits wide)

always_comb begin  // implement state transition diagram
  if (rst_in) next_state = S_RESET;
  else case (state)
    S_RESET: next_state = b0_in ? S_0   : b1_in ? S_RESET : state;
    S_0:     next_state = b0_in ? S_0   : b1_in ? S_01    : state;
    S_01:    next_state = b0_in ? S_010 : b1_in ? S_RESET : state;
    S_010:   next_state = b0_in ? S_0   : b1_in ? S_0101  : state;
    S_0101:  next_state = b0_in ? S_010 : b1_in ? S_01011 : state;
    S_01011: next_state = b0_in ? S_0   : b1_in ? S_RESET : state;
    default: next_state = S_RESET;     // handle unused states
  endcase
end
always_ff @(posedge clk) state <= next_state;
```



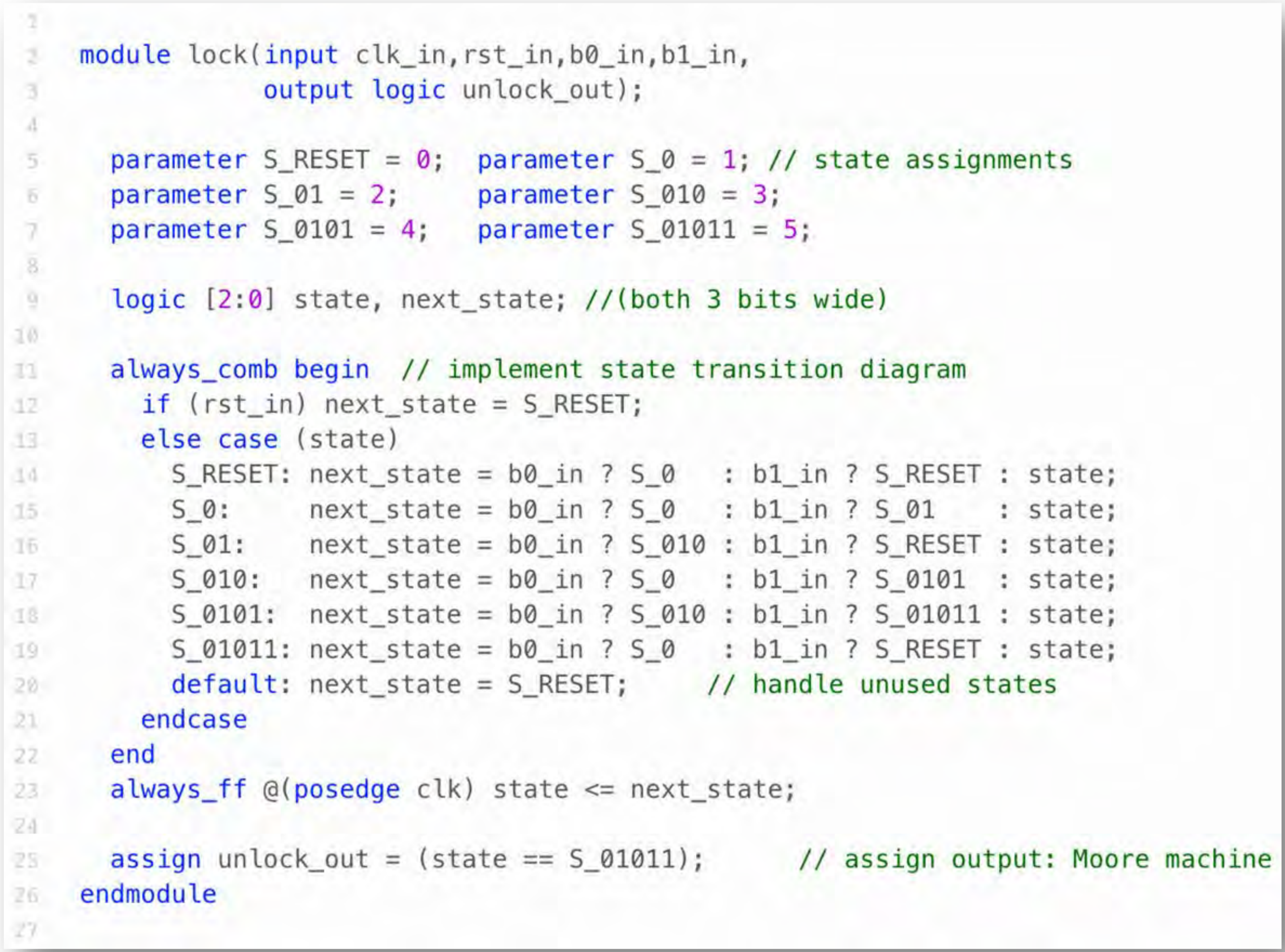
Step 2C: generate output

```
// it's a Moore machine!  Output only depends on current state
```



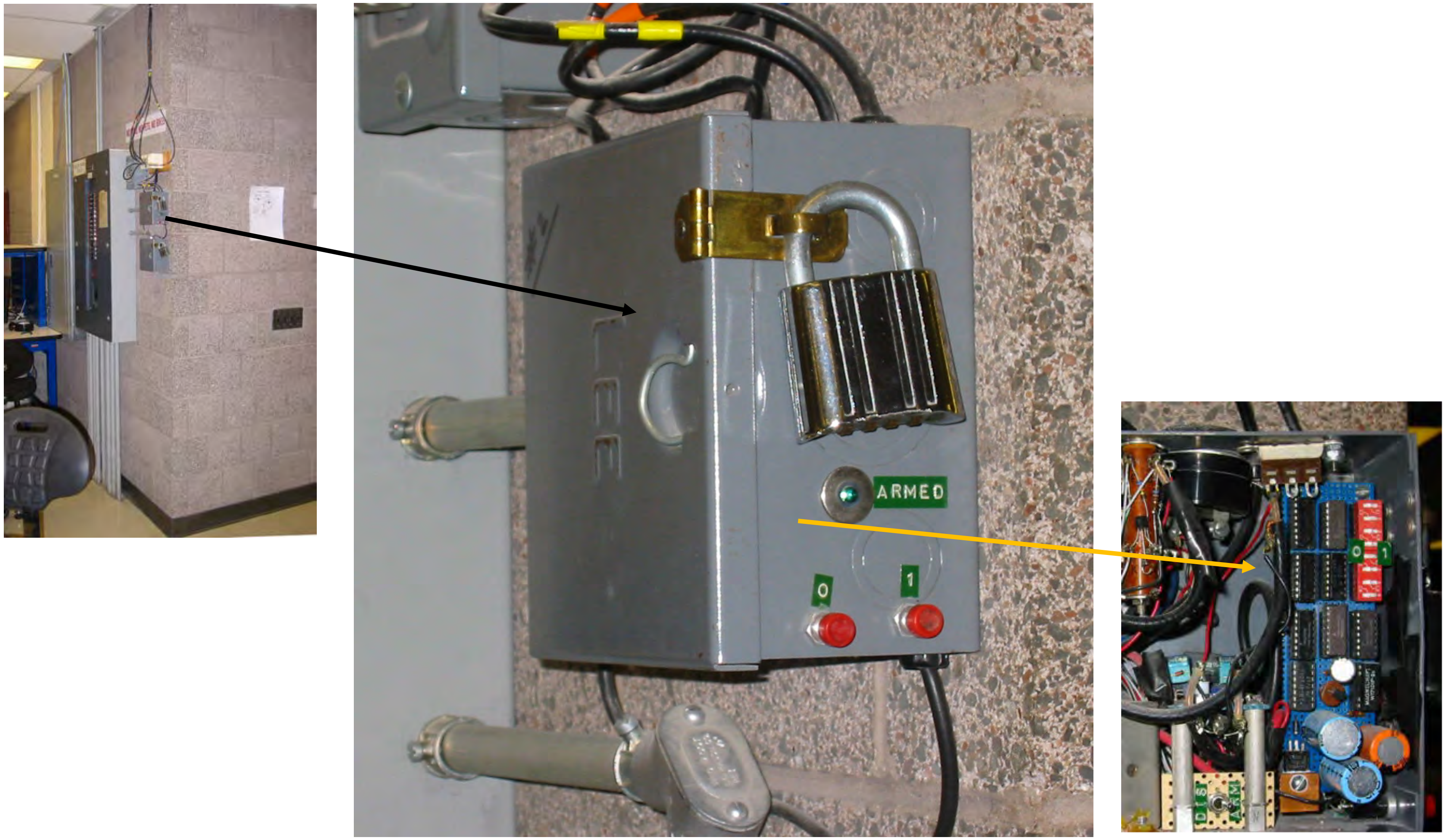
```
    assign unlock_out = (state == S_01011);       // assign output: Moore machine
```

Step 2: final Verilog implementation



```
module lock(input clk_in,rst_in,b0_in,b1_in,
            output logic unlock_out);

  parameter S_RESET = 0;  parameter S_0 = 1; // state assignments
  parameter S_01 = 2;     parameter S_010 = 3;
  parameter S_0101 = 4;   parameter S_01011 = 5;

  logic [2:0] state, next_state; //(both 3 bits wide)

  always_comb begin  // implement state transition diagram
    if (rst_in) next_state = S_RESET;
    else case (state)
      S_RESET: next_state = b0_in ? S_0   : b1_in ? S_RESET : state;
      S_0:     next_state = b0_in ? S_0   : b1_in ? S_01    : state;
      S_01:    next_state = b0_in ? S_010 : b1_in ? S_RESET : state;
      S_010:   next_state = b0_in ? S_0   : b1_in ? S_0101  : state;
      S_0101:  next_state = b0_in ? S_010 : b1_in ? S_01011 : state;
      S_01011: next_state = b0_in ? S_0   : b1_in ? S_RESET : state;
      default: next_state = S_RESET;     // handle unused states
    endcase
  end
  always_ff @(posedge clk) state <= next_state;

  assign unlock_out = (state == S_01011);      // assign output: Moore machine
endmodule
```

Real FSM Security System

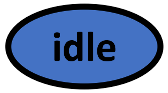


The 6.111 Vending Machine (example from circa 2000…slightly updated)

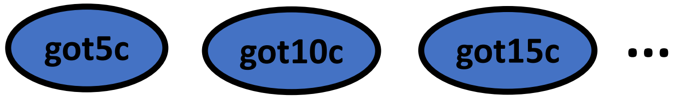
- Lab assistants demand a new soda machine for the 6.111 lab. You design the FSM controller.
- **<u>All selections are $0.30.</u>**
- The machine makes change. (Dimes and nickels only.)
- Inputs: limit 1 per clock
  - Q - quarter inserted
  - D - dime inserted
  - N - nickel inserted
- Outputs: limit 1 per clock
  - DC - dispense can
  - DD - dispense dime
  - DN - dispense nickel

What States are in the System?

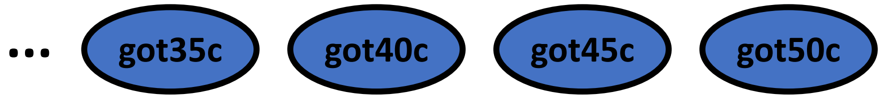
- A starting (idle) state:



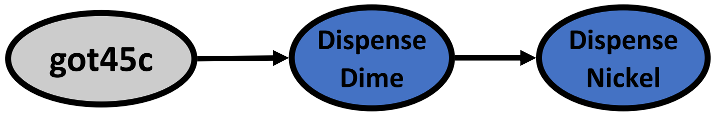
- A state for each possible amount of money captured:



- What's the maximum amount of money captured before purchase?
  25 cents (just shy of a purchase) + one quarter (largest coin)

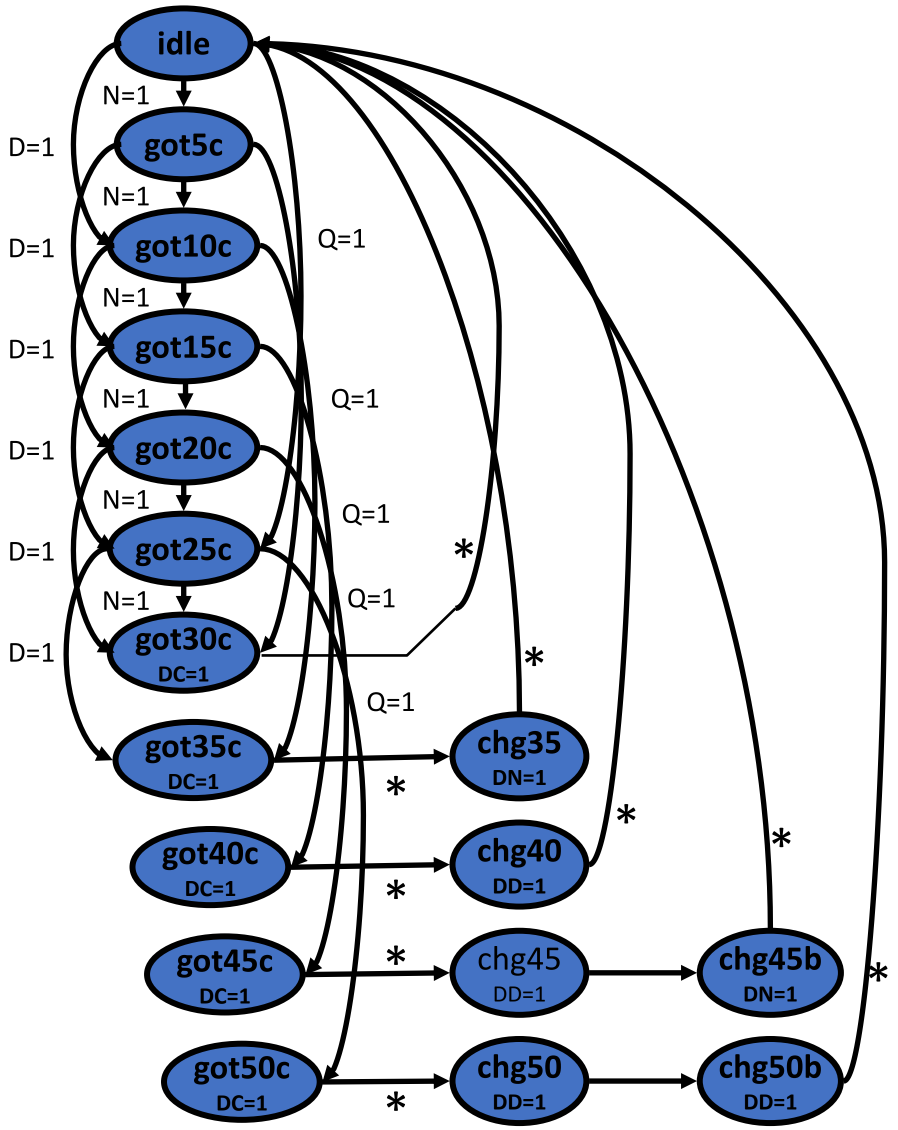


- States to dispense change (one per coin dispensed):



A Moore Vender

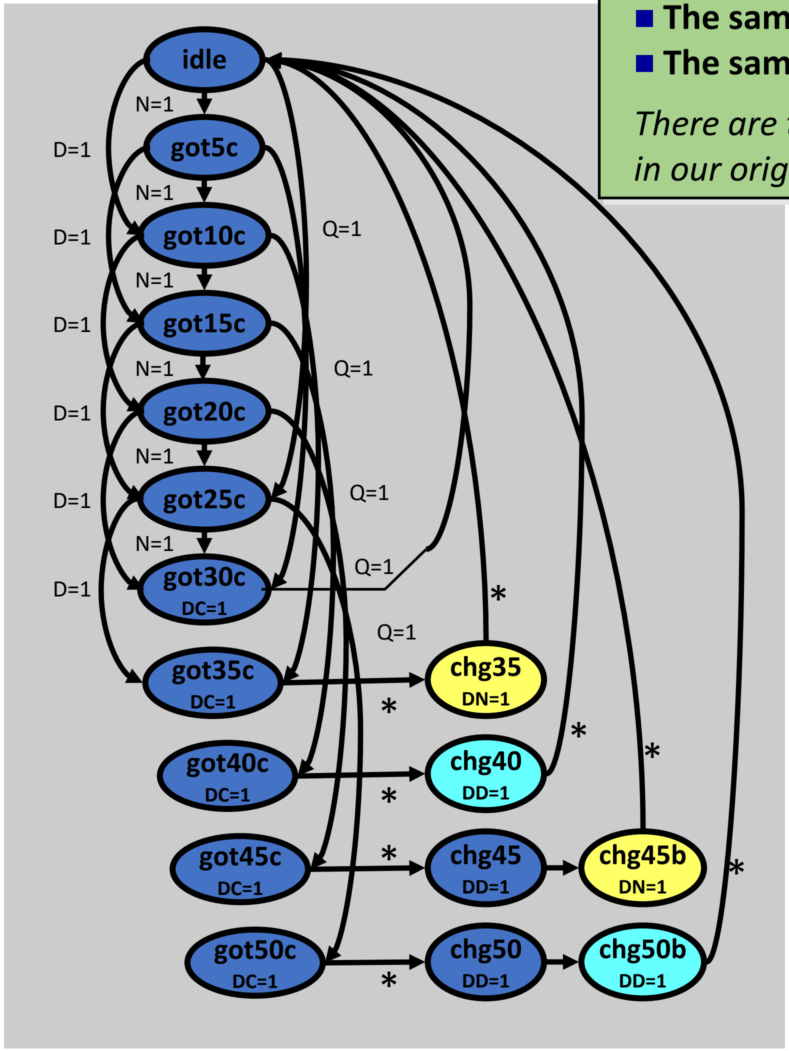
Here's a first cut at the state transition diagram.



State Reduction

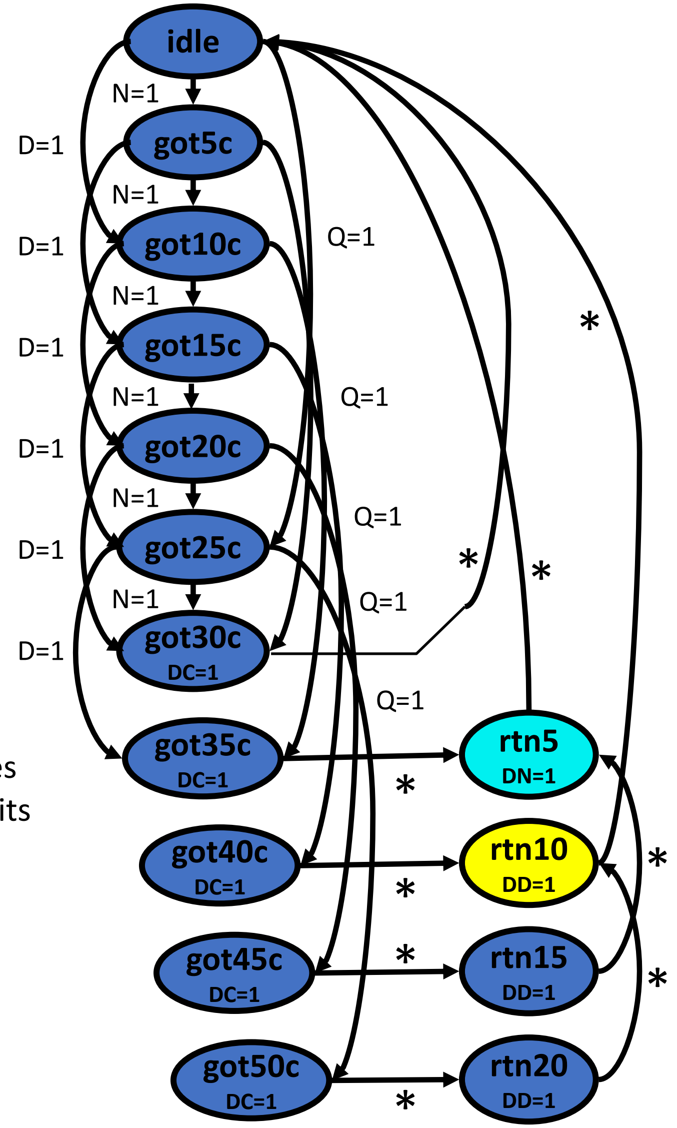
Duplicate states have:
- **The same outputs, and**
- **The same transitions**

There are two duplicates in our original diagram.

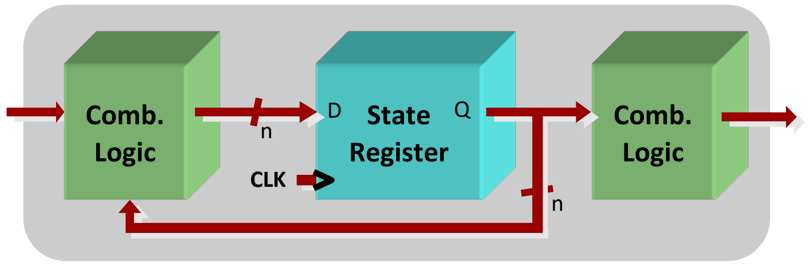


17 states
5 state bits

15 states
4 state bits



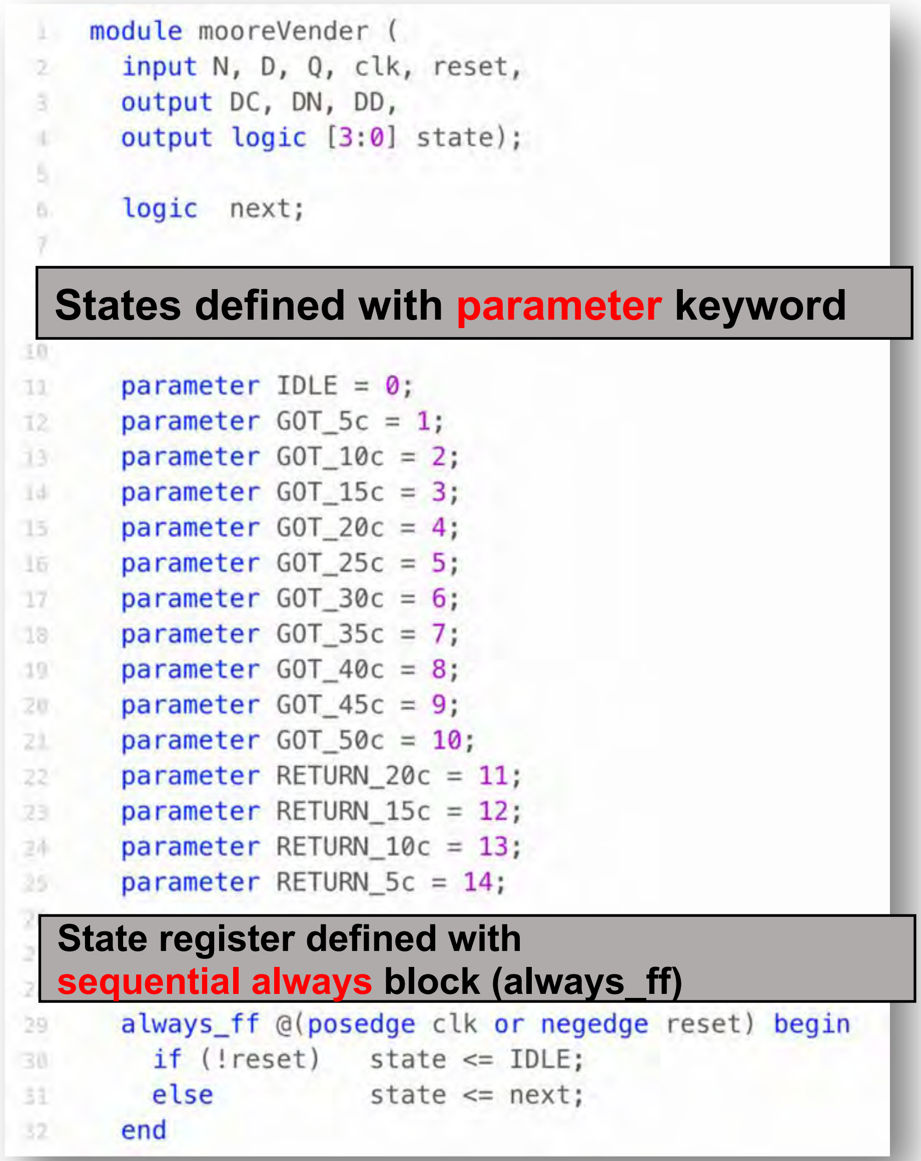
Verilog for the Moore Vender



- **State register**
  (sequential always block)

So triggered on posedge clock

- **Next-state combinational logic**
  (comb. always block with case)
- **Output combinational logic block**
  (comb. always block *or* assign statements)



```
module mooreVender (
  input N, D, Q, clk, reset,
  output DC, DN, DD,
  output logic [3:0] state);

  logic  next;
```

States defined with parameter keyword

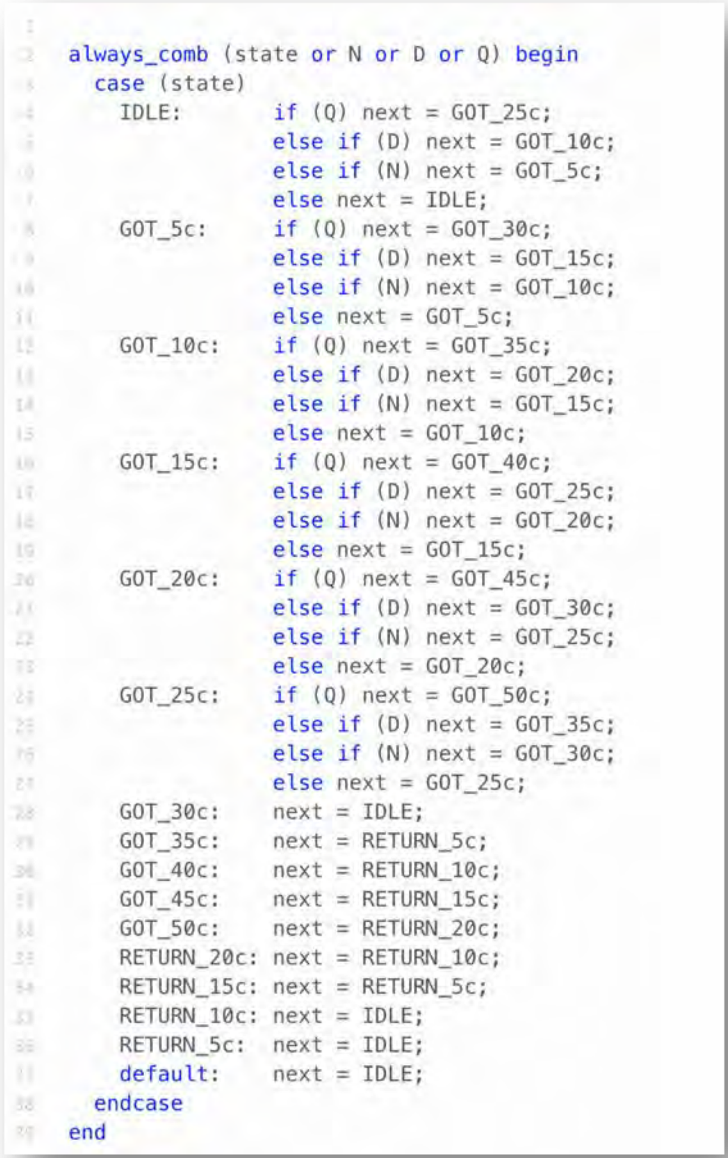
```
  parameter IDLE = 0;
  parameter GOT_5c = 1;
  parameter GOT_10c = 2;
  parameter GOT_15c = 3;
  parameter GOT_20c = 4;
  parameter GOT_25c = 5;
  parameter GOT_30c = 6;
  parameter GOT_35c = 7;
  parameter GOT_40c = 8;
  parameter GOT_45c = 9;
  parameter GOT_50c = 10;
  parameter RETURN_20c = 11;
  parameter RETURN_15c = 12;
  parameter RETURN_10c = 13;
  parameter RETURN_5c = 14;
```

State register defined with sequential always block (always_ff)

```
  always_ff @(posedge clk or negedge reset) begin
    if (!reset)    state <= IDLE;
    else           state <= next;
  end
```

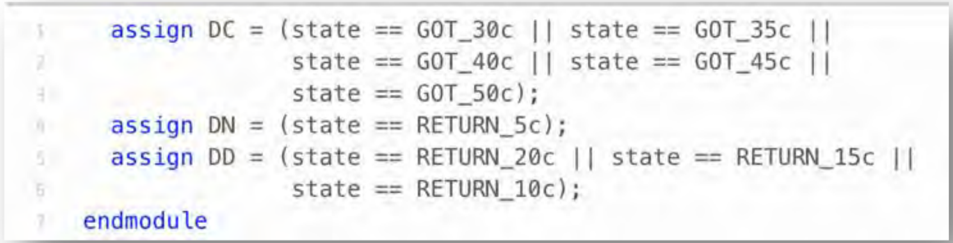
Verilog for the Moore Vender

Next-state logic within a combinational always block



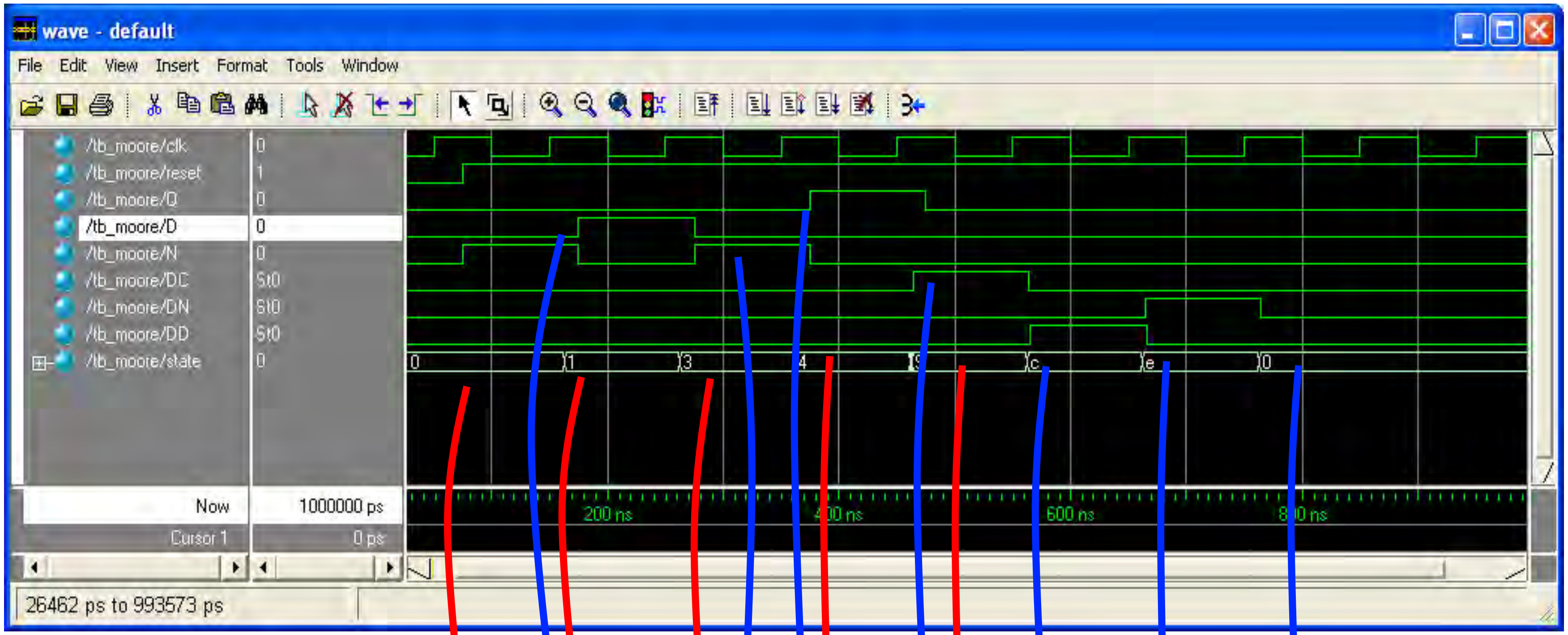
```
always_comb (state or N or D or Q) begin
  case (state)
    IDLE:         if (Q) next = GOT_25c;
                  else if (D) next = GOT_10c;
                  else if (N) next = GOT_5c;
                  else next = IDLE;
    GOT_5c:       if (Q) next = GOT_30c;
                  else if (D) next = GOT_15c;
                  else if (N) next = GOT_10c;
                  else next = GOT_5c;
    GOT_10c:      if (Q) next = GOT_35c;
                  else if (D) next = GOT_20c;
                  else if (N) next = GOT_15c;
                  else next = GOT_10c;
    GOT_15c:      if (Q) next = GOT_40c;
                  else if (D) next = GOT_25c;
                  else if (N) next = GOT_20c;
                  else next = GOT_15c;
    GOT_20c:      if (Q) next = GOT_45c;
                  else if (D) next = GOT_30c;
                  else if (N) next = GOT_25c;
                  else next = GOT_20c;
    GOT_25c:      if (Q) next = GOT_50c;
                  else if (D) next = GOT_35c;
                  else if (N) next = GOT_30c;
                  else next = GOT_25c;
    GOT_30c:      next = IDLE;
    GOT_35c:      next = RETURN_5c;
    GOT_40c:      next = RETURN_10c;
    GOT_45c:      next = RETURN_15c;
    GOT_50c:      next = RETURN_20c;
    RETURN_20c:   next = RETURN_10c;
    RETURN_15c:   next = RETURN_5c;
    RETURN_10c:   next = IDLE;
    RETURN_5c:    next = IDLE;
    default:      next = IDLE;
  endcase
end
```

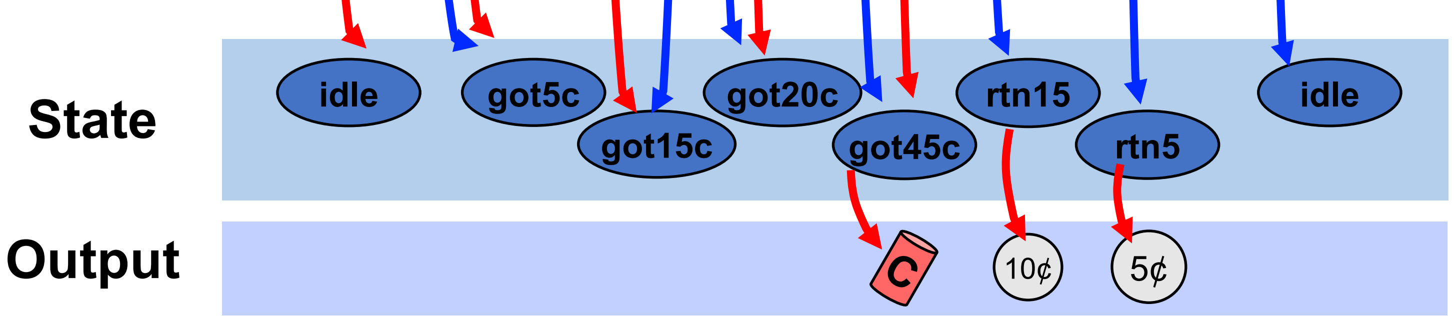
Combinational output assignment



```
  assign DC = (state == GOT_30c || state == GOT_35c ||
               state == GOT_40c || state == GOT_45c ||
               state == GOT_50c);
  assign DN = (state == RETURN_5c);
  assign DD = (state == RETURN_20c || state == RETURN_15c ||
               state == RETURN_10c);
endmodule
```

Simulation of Moore Vender

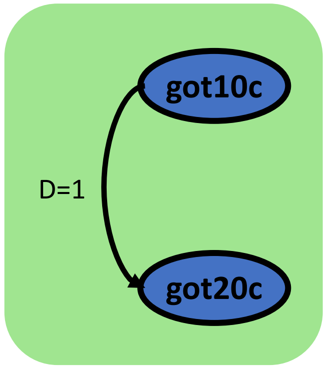




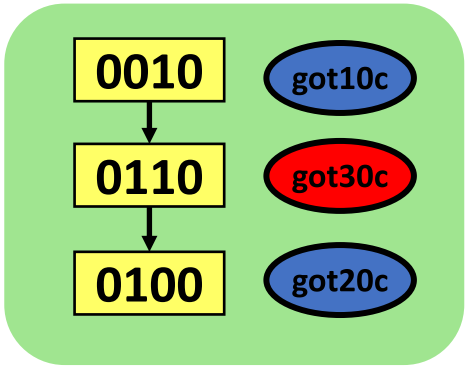
FSM Output Glitching

- **FSM state bits may not transition at precisely the same time**
- **Combinational logic for outputs may contain hazards**
- **Result: your FSM outputs may glitch!**

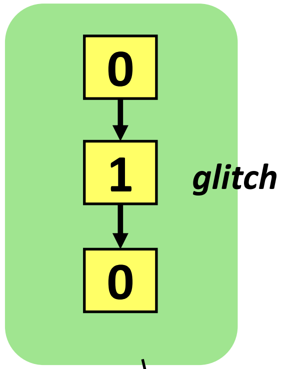
during this state transition...

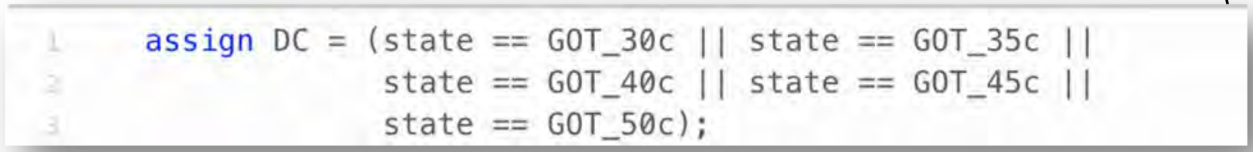


...the state registers may transtion like this...



*...causing the DC output to **glitch** like this!*



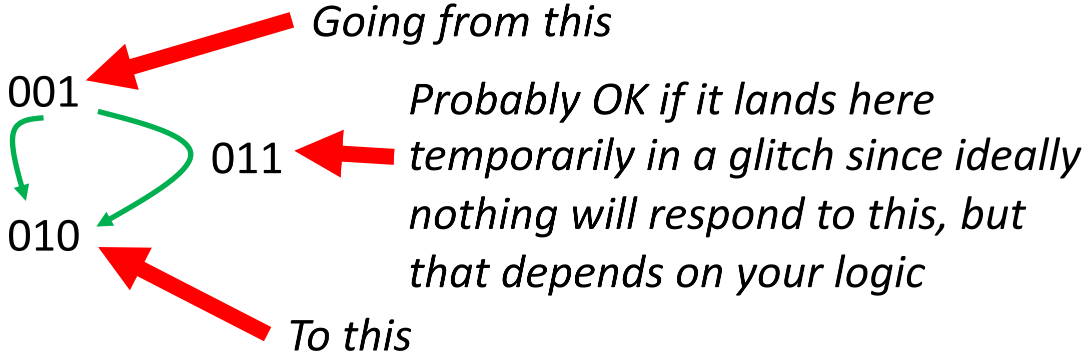


```
assign DC = (state == GOT_30c || state == GOT_35c ||
             state == GOT_40c || state == GOT_45c ||
             state == GOT_50c);
```

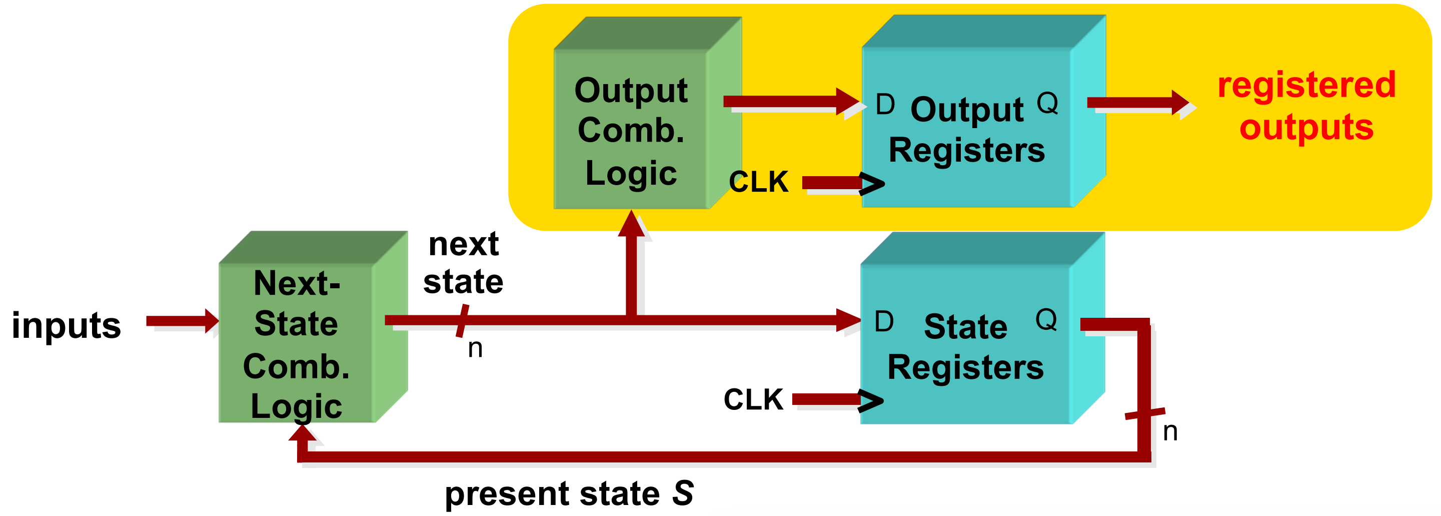
If the soda dispenser is glitch-sensitive, your customers can get a 20-cent soda!

One way to fix Glitches:

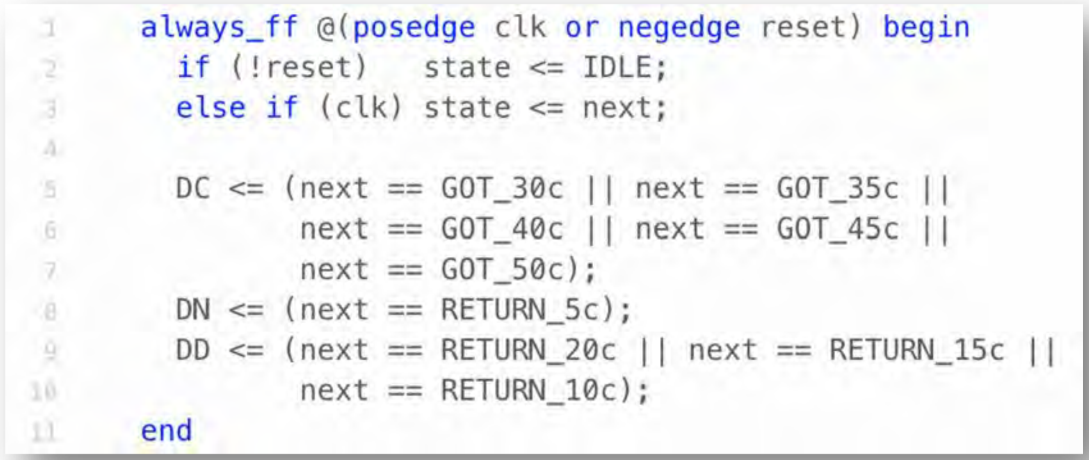
- Don't have to have state 3 (3'b011) go into state 4 (3'b100). Use different state naming/use different numbers!!! *A rose by any other name would smell as sweet*
- Perhaps a Gray code (??):
  - Count up like: 000, 001, 011, 010, 110, 111, 101, 100, …
  - Have the really important/glitch-sensitive states only require transitions of one bit
- One-hot encoding:


Going from this
001
011
Probably OK if it lands here temporarily in a glitch since ideally nothing will respond to this, but that depends on your logic
010
To this

Another Solution:
Registered FSM Outputs are Glitch-Free



- Move output generation into the sequential always block
- Calculate outputs based on <u>next</u> state
- Delays outputs by one clock cycle. Problematic in some application.



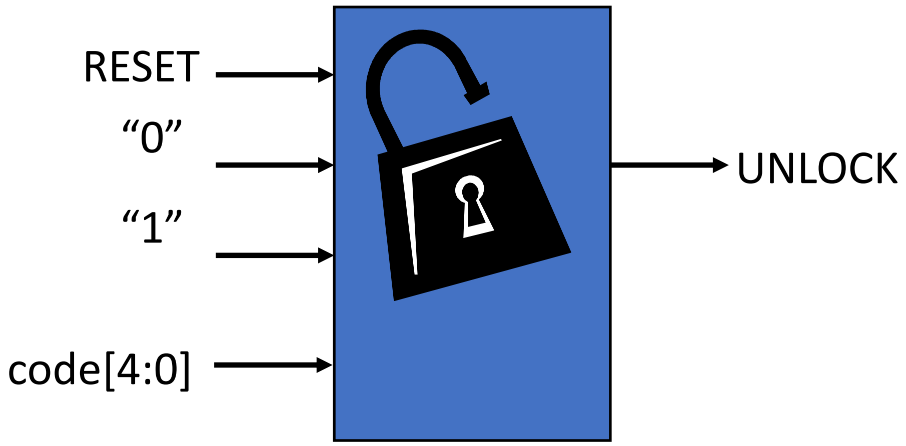
```
always_ff @(posedge clk or negedge reset) begin
  if (!reset)   state <= IDLE;
  else if (clk) state <= next;

  DC <= (next == GOT_30c || next == GOT_35c ||
         next == GOT_40c || next == GOT_45c ||
         next == GOT_50c);
  DN <= (next == RETURN_5c);
  DD <= (next == RETURN_20c || next == RETURN_15c ||
         next == RETURN_10c);
end
```

Note this is inside an edged always with non-blocking assigns!
This will synthesize to registered outputs!

Let’s Do a Better Lock

GOAL:

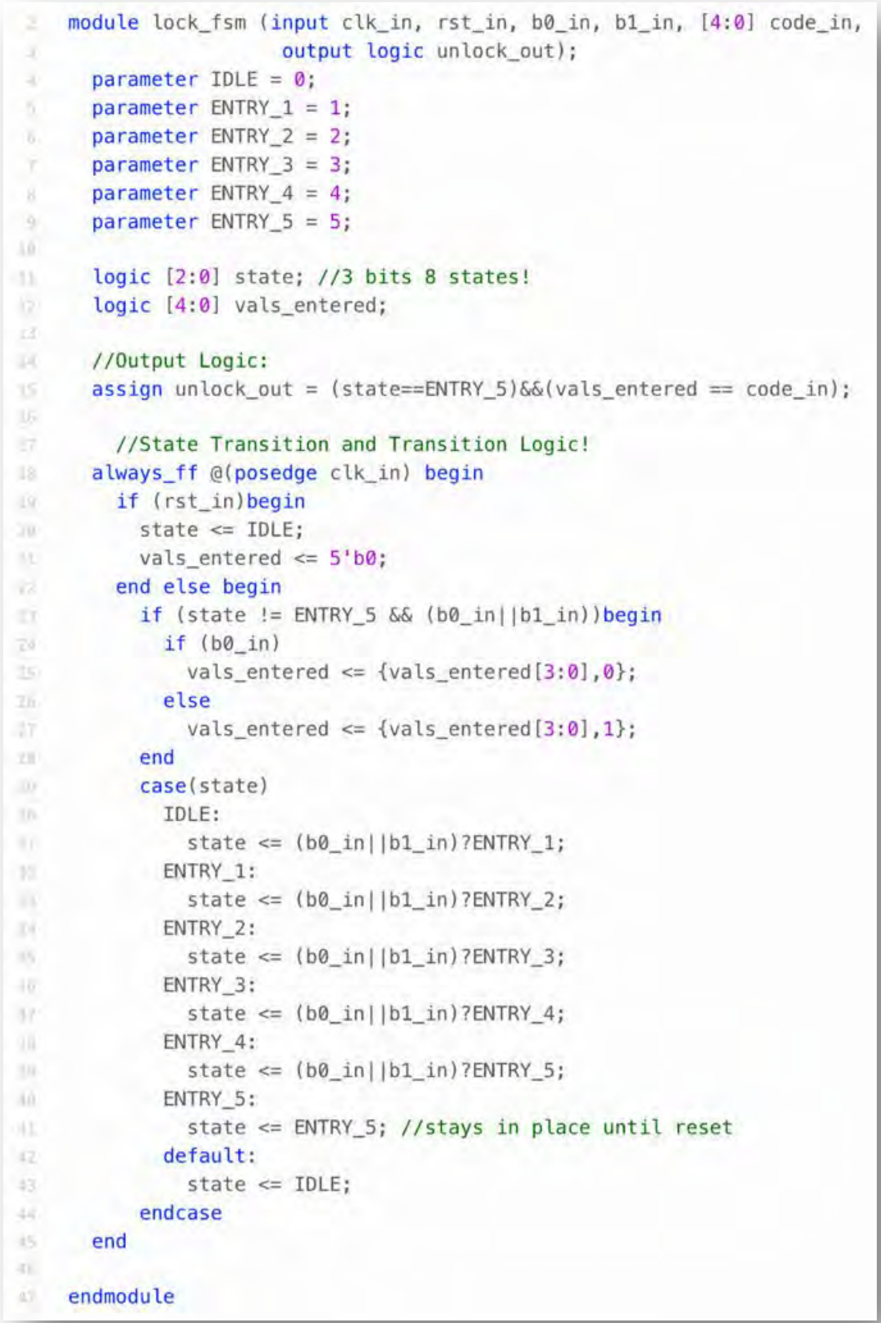
- Build an electronic combination lock with a reset button, two number buttons (0 and 1), and an unlock output signal. The combination is 5 bits **specified by the user!**.
- After each attempt, reset must be pushed to try again



By departing a bit from a rigid FSM structure we can maybe express ourselves more effectively!!

Implementation 1

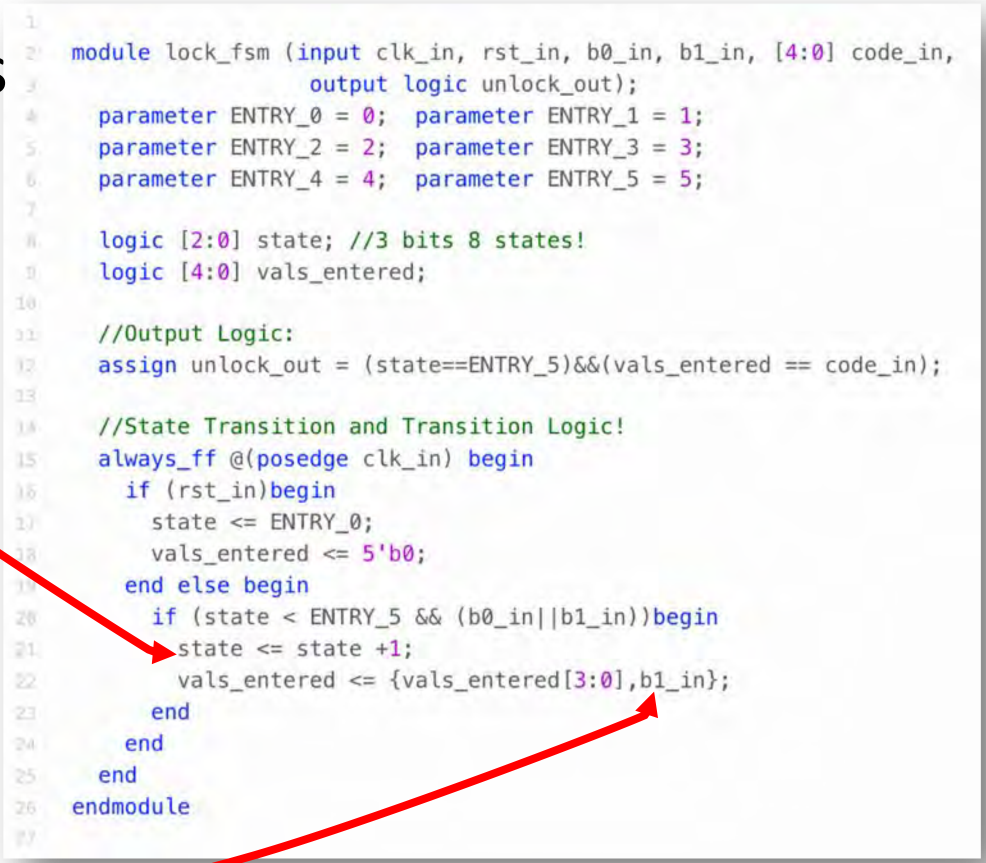
- Rather verbose Moore FSM
- Output is Moore-ish (but also additional information)



```
module lock_fsm (input clk_in, rst_in, b0_in, b1_in, [4:0] code_in,
                  output logic unlock_out);
  parameter IDLE = 0;
  parameter ENTRY_1 = 1;
  parameter ENTRY_2 = 2;
  parameter ENTRY_3 = 3;
  parameter ENTRY_4 = 4;
  parameter ENTRY_5 = 5;

  logic [2:0] state; //3 bits 8 states!
  logic [4:0] vals_entered;

  //Output Logic:
  assign unlock_out = (state==ENTRY_5)&&(vals_entered == code_in);

    //State Transition and Transition Logic!
  always_ff @(posedge clk_in) begin
    if (rst_in)begin
      state <= IDLE;
      vals_entered <= 5'b0;
    end else begin
      if (state != ENTRY_5 && (b0_in||b1_in))begin
        if (b0_in)
          vals_entered <= {vals_entered[3:0],0};
        else
          vals_entered <= {vals_entered[3:0],1};
      end
      case(state)
        IDLE:
          state <= (b0_in||b1_in)?ENTRY_1;
        ENTRY_1:
          state <= (b0_in||b1_in)?ENTRY_2;
        ENTRY_2:
          state <= (b0_in||b1_in)?ENTRY_3;
        ENTRY_3:
          state <= (b0_in||b1_in)?ENTRY_4;
        ENTRY_4:
          state <= (b0_in||b1_in)?ENTRY_5;
        ENTRY_5:
          state <= ENTRY_5; //stays in place until reset
        default:
          state <= IDLE;
      endcase
  end

endmodule
```

Implementation 1

- Very similar to previous implementation, but took advantage of ordering of states



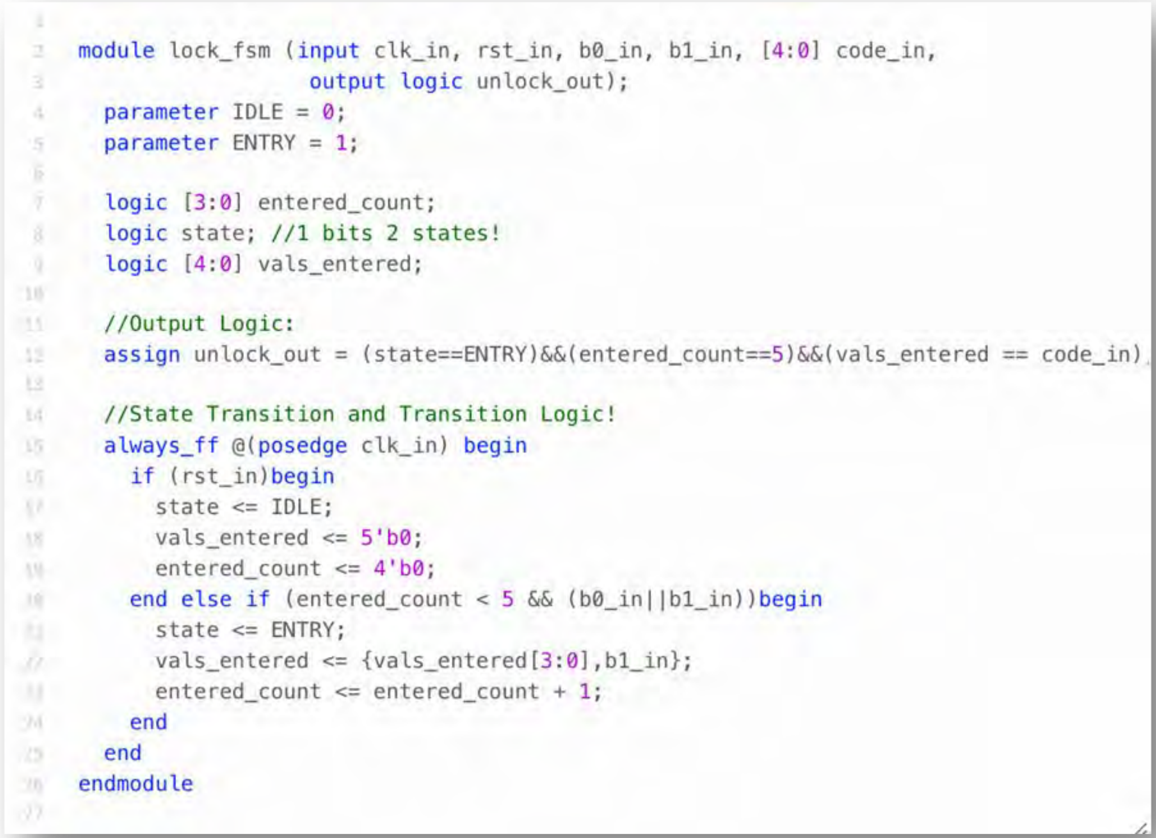
```
module lock_fsm (input clk_in, rst_in, b0_in, b1_in, [4:0] code_in,
                 output logic unlock_out);
  parameter ENTRY_0 = 0;  parameter ENTRY_1 = 1;
  parameter ENTRY_2 = 2;  parameter ENTRY_3 = 3;
  parameter ENTRY_4 = 4;  parameter ENTRY_5 = 5;

  logic [2:0] state; //3 bits 8 states!
  logic [4:0] vals_entered;

  //Output Logic:
  assign unlock_out = (state==ENTRY_5)&&(vals_entered == code_in);

  //State Transition and Transition Logic!
  always_ff @(posedge clk_in) begin
    if (rst_in)begin
      state <= ENTRY_0;
      vals_entered <= 5'b0;
    end else begin
      if (state < ENTRY_5 && (b0_in||b1_in))begin
        state <= state +1;
        vals_entered <= {vals_entered[3:0],b1_in};
      end
    end
  end
endmodule
```

Cute but sneaky way to add in which button is getting pushed

Implementation 2

- Use fewer explicit states, but maybe need to remember stuff somehow
- Bring in additional variables…in theoretical sense those are also states

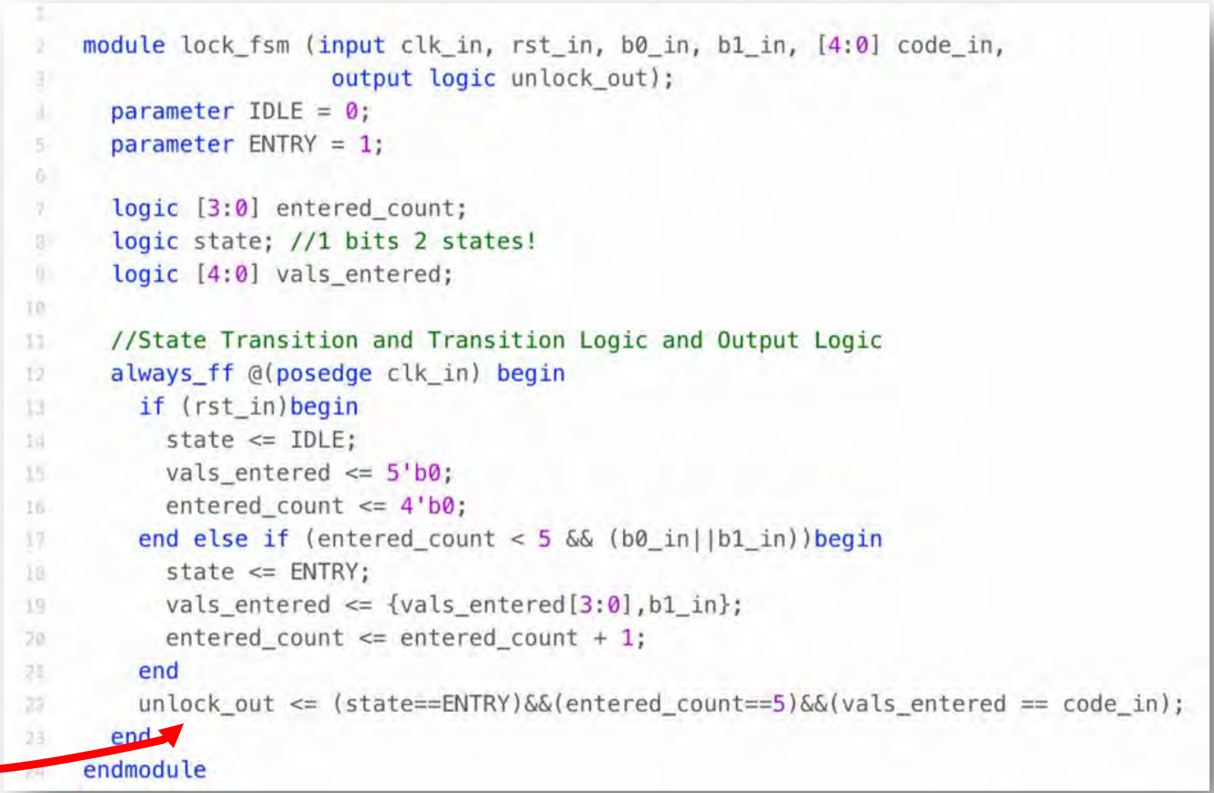
A rose by any other name…



```
module lock_fsm (input clk_in, rst_in, b0_in, b1_in, [4:0] code_in,
                  output logic unlock_out);
  parameter IDLE = 0;
  parameter ENTRY = 1;

  logic [3:0] entered_count;
  logic state; //1 bits 2 states!
  logic [4:0] vals_entered;

  //Output Logic:
  assign unlock_out = (state==ENTRY)&&(entered_count==5)&&(vals_entered == code_in)

  //State Transition and Transition Logic!
  always_ff @(posedge clk_in) begin
    if (rst_in)begin
      state <= IDLE;
      vals_entered <= 5'b0;
      entered_count <= 4'b0;
    end else if (entered_count < 5 && (b0_in||b1_in))begin
      state <= ENTRY;
      vals_entered <= {vals_entered[3:0],b1_in};
      entered_count <= entered_count + 1;
    end
  end
endmodule
```

Implementation 2

- If I'm worried about glitching…
- Move unlock_out assignment to within sequential block
- Becomes registered, far less likely to glitch!



```
module lock_fsm (input clk_in, rst_in, b0_in, b1_in, [4:0] code_in,
                  output logic unlock_out);
  parameter IDLE = 0;
  parameter ENTRY = 1;

  logic [3:0] entered_count;
  logic state; //1 bits 2 states!
  logic [4:0] vals_entered;

  //State Transition and Transition Logic and Output Logic
  always_ff @(posedge clk_in) begin
    if (rst_in)begin
      state <= IDLE;
      vals_entered <= 5'b0;
      entered_count <= 4'b0;
    end else if (entered_count < 5 && (b0_in||b1_in))begin
      state <= ENTRY;
      vals_entered <= {vals_entered[3:0],b1_in};
      entered_count <= entered_count + 1;
    end
    unlock_out <= (state==ENTRY)&&(entered_count==5)&&(vals_entered == code_in);
  end
endmodule
```

Summary

- Other options (?):
  - Mealy (both of previous ones were basically MOORE state machines)
- No matter what many solutions will theoretically be a Mealy or Moore FSM…whether you structure it as such is up to you.
- Thinking about problems as Mealy/Moore FSM is a powerful way to get started on a solution
- Sometimes it can be too-restrictive and/or not scale the best, however
- The best choice is usually something in between!

Ray Tracer

Sam Gross, Adam Lerer - Spring 2007

Pong that you Live In

Nicholas Waltman Mike Wang, 2018