

# Intro to Digital Design

## L6: More FSMs, Synchronous Timing Constraints

**Instructor:** Naomi Alterman

**Teaching Assistants:**

Derek de Leuw

Isabel Froelich

Kevin Hernandez

Sathvik Kanuri

Aadithya Manoj

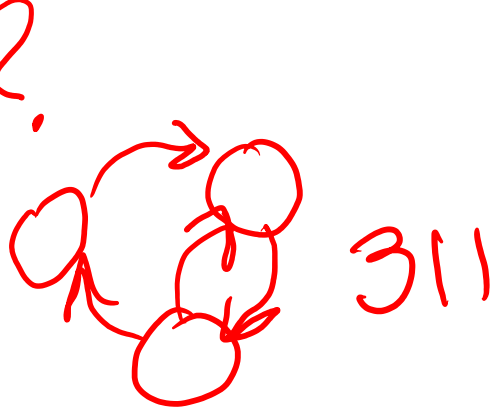
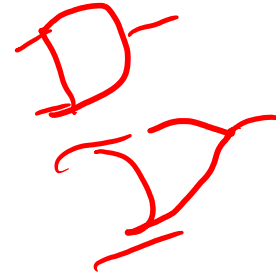
# Administrivia

- ❖ Lab 6 – Connecting multiple FSMs in Tug of War game
  - *Bigger* step up in difficulty from Lab 5
  - Putting together complex system – interconnections!
  - Bonus points for smaller resource usage

# Outline

- ❖ **FSM Design Example**
- ❖ Synchronous Timing Constraints

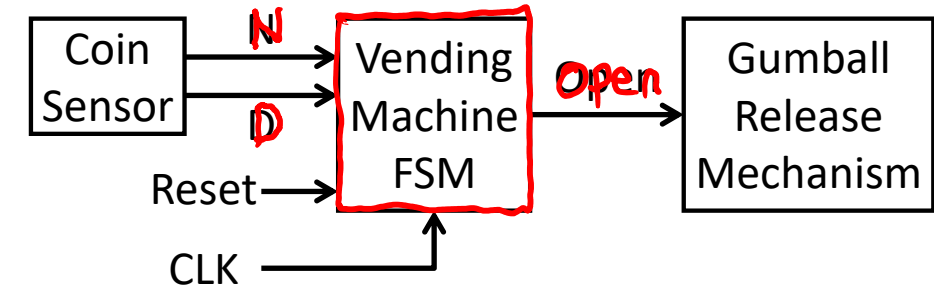
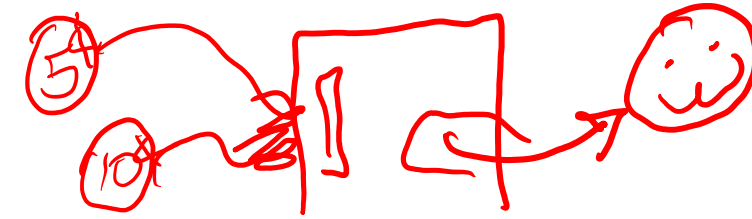
# FSM Design Process

- 1) Understand the problem?
- 2) Draw the state diagram 
- 3) Use state diagram to produce truth table *kmaps*
- 4) Use truth table to implement combinational logic 

# Vending Machine Example

## ❖ Vending machine description/behavior:

- Single coin slot for dimes and nickels
- Releases gumball after  $\geq 10$  cents deposited
- Gives no change



## ❖ State Diagram:



# Vending Machine State Table

$S_P$   $S_N$

$$NS \approx PS \bar{N} \bar{D} + \bar{P} S N$$

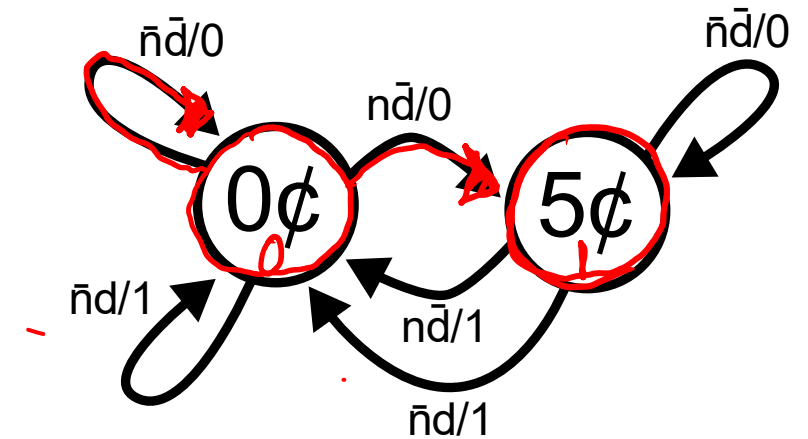
PS	N	D	NS	Open
0	0	0	0	0
0	0	1	0	1
0	1	0	1	0
0	1	1	X	X
1	0	0	1	0
1	0	1	0	1
1	1	0	0	1
1	1	1	X	X

PS,N

D	00	01	11	10
0	0	1	0	1
1	0	X	X	0

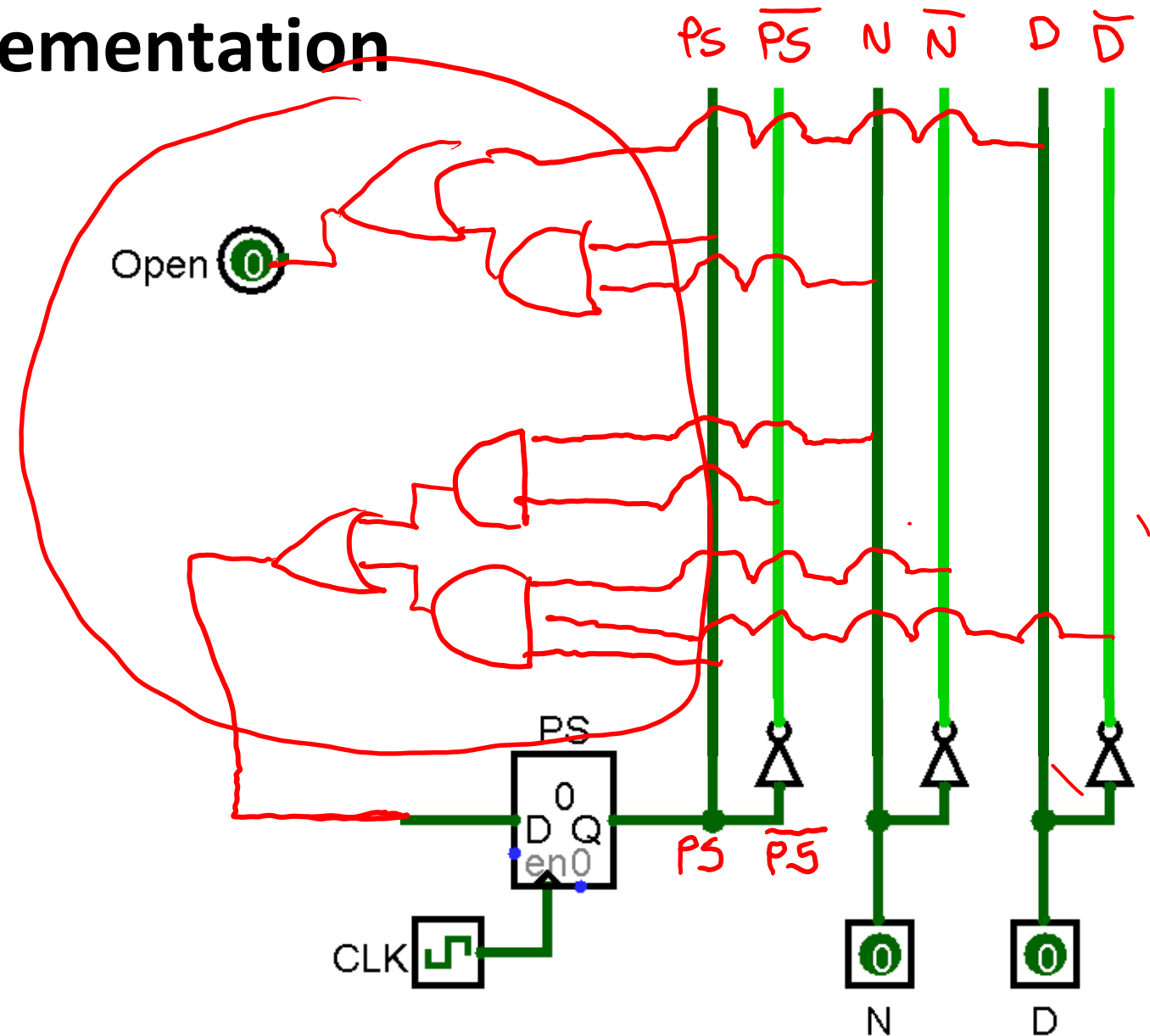
PS,N

D	00	01	11	10
0				
1				



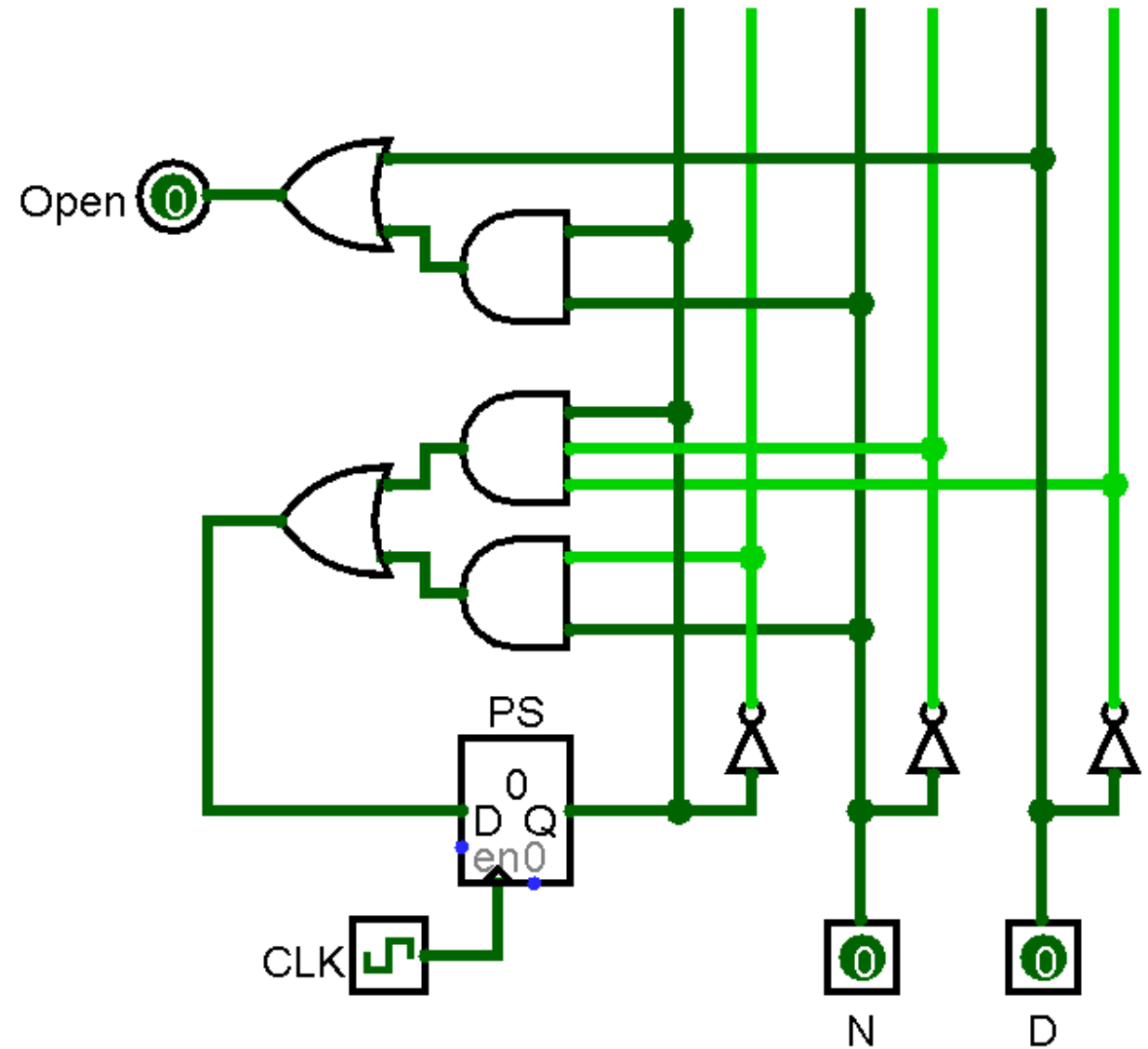
# Vending Machine Implementation

- ❖  $\text{Open} = \overline{D} + PS \cdot N$
- ❖  $\text{NS} = \overline{PS} \cdot N + PS \cdot \overline{N} \cdot \overline{D}$



# Vending Machine Implementation

- ❖  $\text{Open} = D + \text{PS} \cdot N$
- ❖  $\text{NS} = \overline{\text{PS}} \cdot N + \text{PS} \cdot \overline{N} \cdot \overline{D}$





# FSMs in Verilog (1/3) : Declarations

- ❖ Let's examine the components of the Verilog FSM example module on the next few slides

```
module vendingMachineFSM (clk, reset, n, d, open);  
    input  logic clk, reset, n, d;  
    output logic open;  
  
    // State Encodings and variables  
    // ps = Present State, ns = Next State  
    enum logic {C0 = 1'b0, C5 = 1'b1} ps, ns;  
    ...
```

# FSMs in Verilog (2/3) : Combinational Logic

...

```
// Next State Logic
```

```
always_comb
```

```
  case (ps)
```

```
    C0: if (n & ~d) ns = C5;
```

```
        else          ns = C0;
```

```
    C5: if (n | d) ns = C0;
```

```
        else          ns = C5;
```

```
endcase
```

```
// Output Logic - could have been in "always" block
```

```
// or part of Next State Logic.
```

```
assign open = ((ps == C0) & d) | ((ps == C5) & (n | d)) ;
```

...

## FSMs in Verilog (3/3) : State

...


```
// Sequential Logic (DFFs)
always_ff @(posedge clk)
  if (reset)
    ps <= C0;
  else
    ps <= ns;
```

endmodule

# Bits of state  
are really minimal!

# FSM Testbench (1/2)

```
module vendingMachineFSM_tb();  
    logic clk, reset, n, d;  
    logic open;
```



```
    vendingMachineFSM dut (.clk, .reset, .n, .d, .open);
```

```
    // Set up the clock
```



```
    parameter CLOCK_PERIOD=100;
```

```
    initial begin
```

```
        clk <= 0;
```

```
        forever #(CLOCK_PERIOD/2) clk <= ~clk;
```

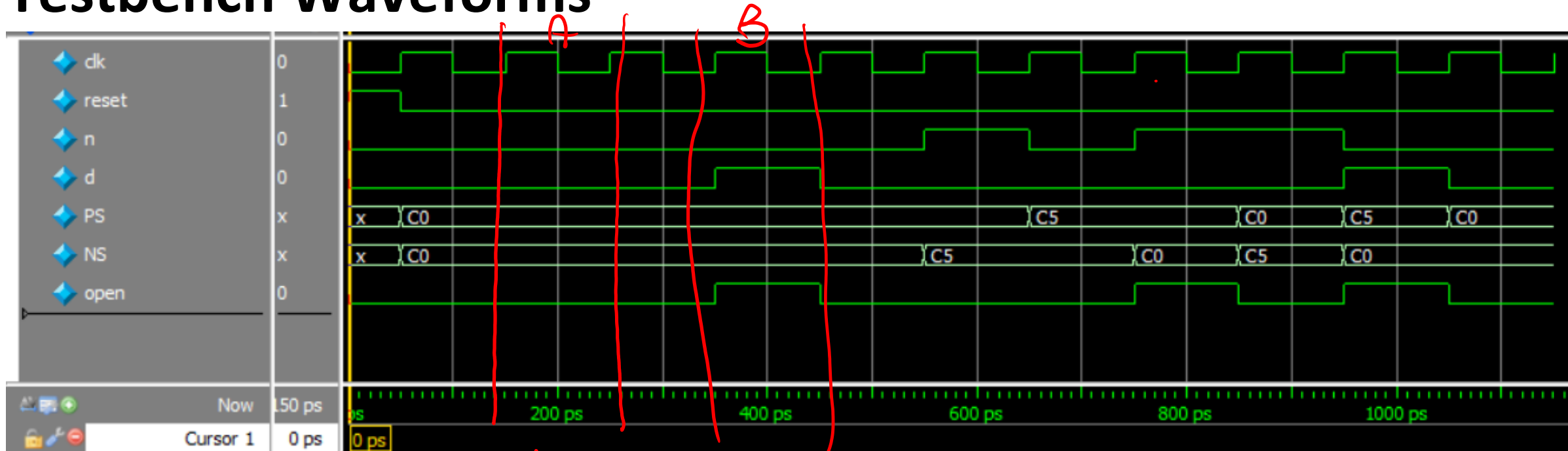
```
    end
```

```
    ...
```

# FSM Testbench (2/2)

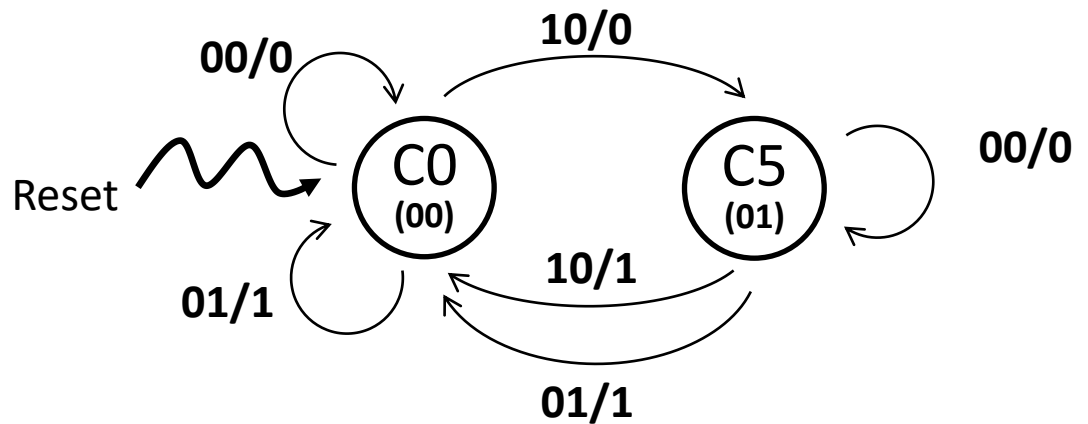
```
// Set up the inputs to the design (each line is a clock cycle)
initial begin
    reset <= 1; n <= 0; d <= 0; @(posedge clk);
        reset <= 0; @(posedge clk);
            @(posedge clk);
            @(posedge clk);
            d <= 1; @(posedge clk);
            d <= 0; @(posedge clk);
            n <= 1; @(posedge clk);
            n <= 0; @(posedge clk);
            n <= 1; @(posedge clk);
                @(posedge clk);
            n <= 0; d<=1; @(posedge clk);
                d<=0; @(posedge clk);
        $stop; // End the simulation
end
```

# Testbench Waveforms



A tests FSM when in steady state

- ❖ What is the min # of clock cycles to *completely* test this FSM? 🤔



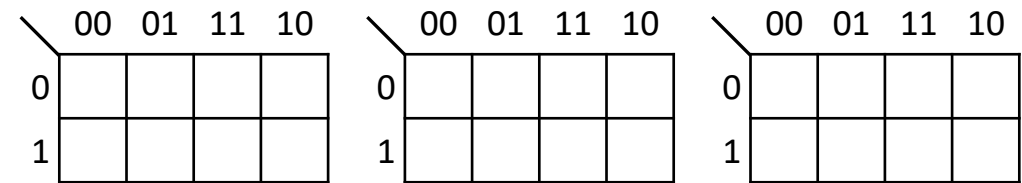
# More Practice: String Recognizer FSM

❖ Recognize the string 101 with the following behavior

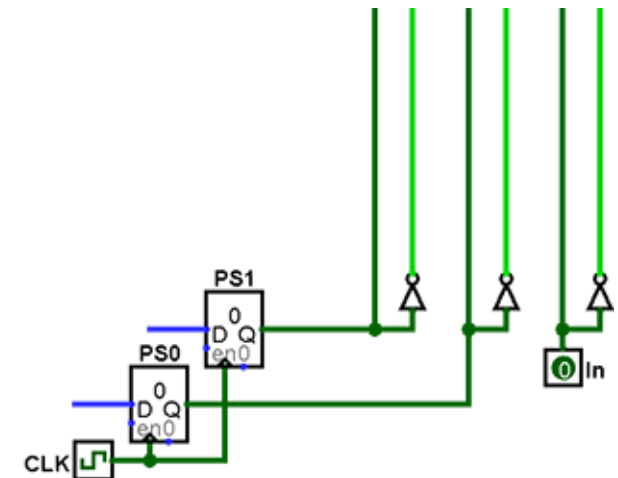
■ Input: 1 0 0 1 0 1 0 1 1 0 0 1 0

■ Output: 0 0 0 0 0 1 0 1 0 0 0 0 0

❖ State diagram to implementation:



Out



# Miso Moment





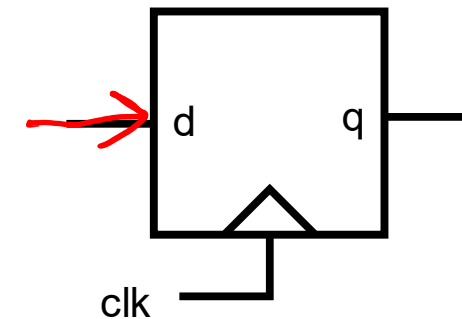
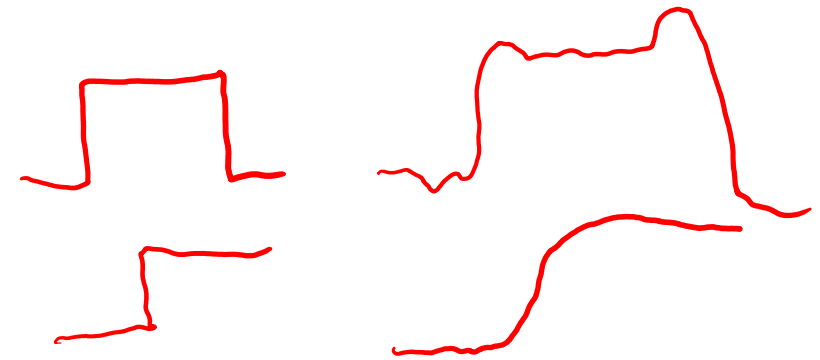
# Outline

- ❖ FSM Design Example
- ❖ **Synchronous Timing Constraints**

# Reminder: Flip Flops

- ❖ A single bit of memory
- ❖ Copy d to q on the rising edge of the clock signal

```
module DFF (q, d, clk);  
    output logic q; // q is state-holding  
    input  logic d, reset, clk;  
  
    always_ff @(posedge clk) begin  
        q <= d;  
    end  
  
endmodule
```



# Flip-Flop Timing Terminology (1/2)

- ❖ Camera Analogy: non-blurry digital photo
  - *Don't move* while camera shutter is opening
  - *Don't move* while camera shutter is closing
  - *Check for blurriness* once image appears on the display

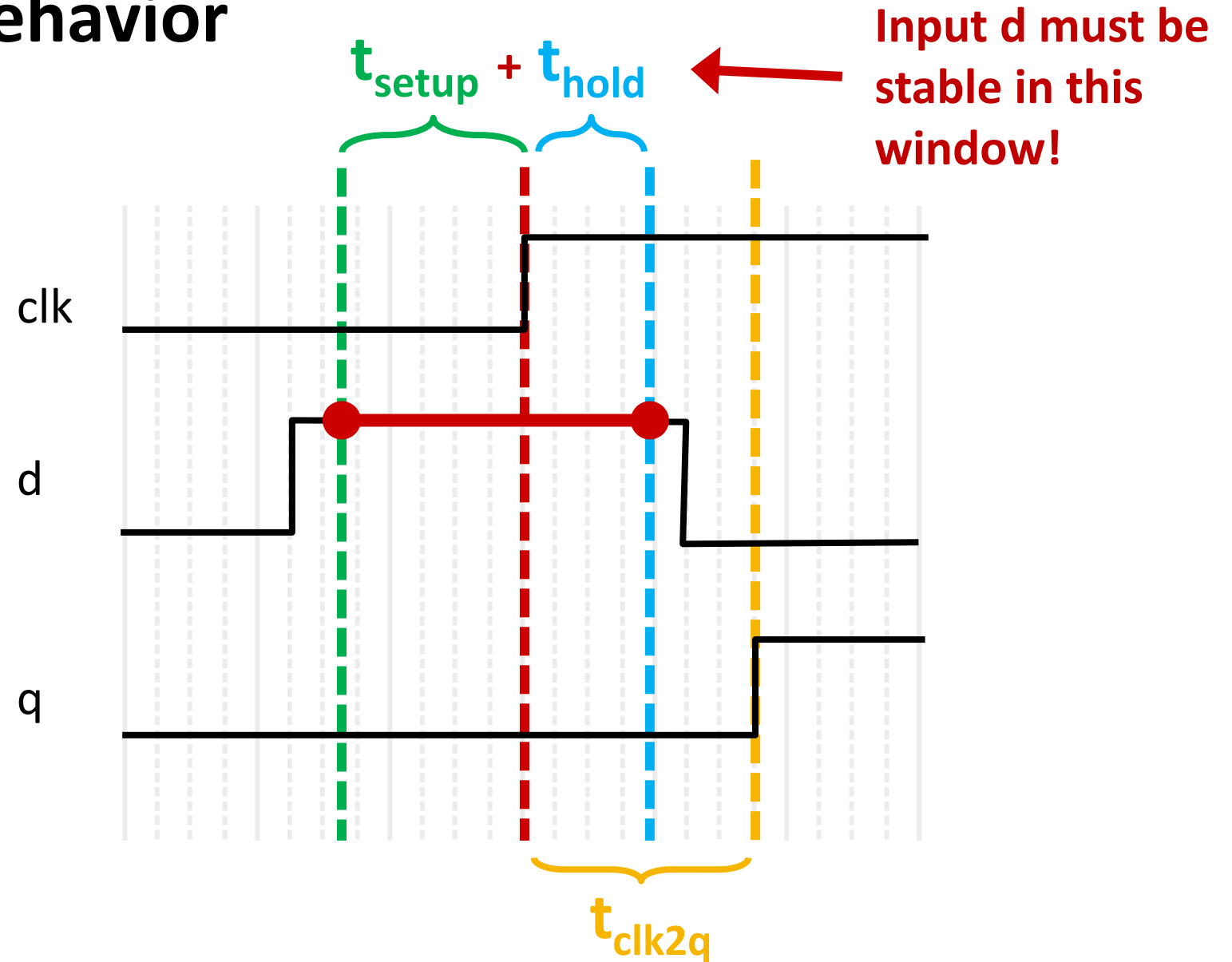
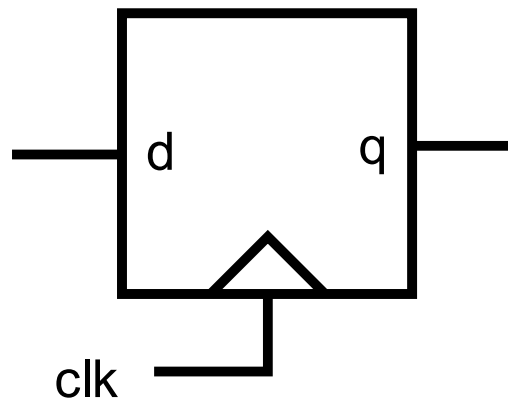


# Flip-Flop Timing Terminology (2/2)

❖ Now applied to sequential logic elements:

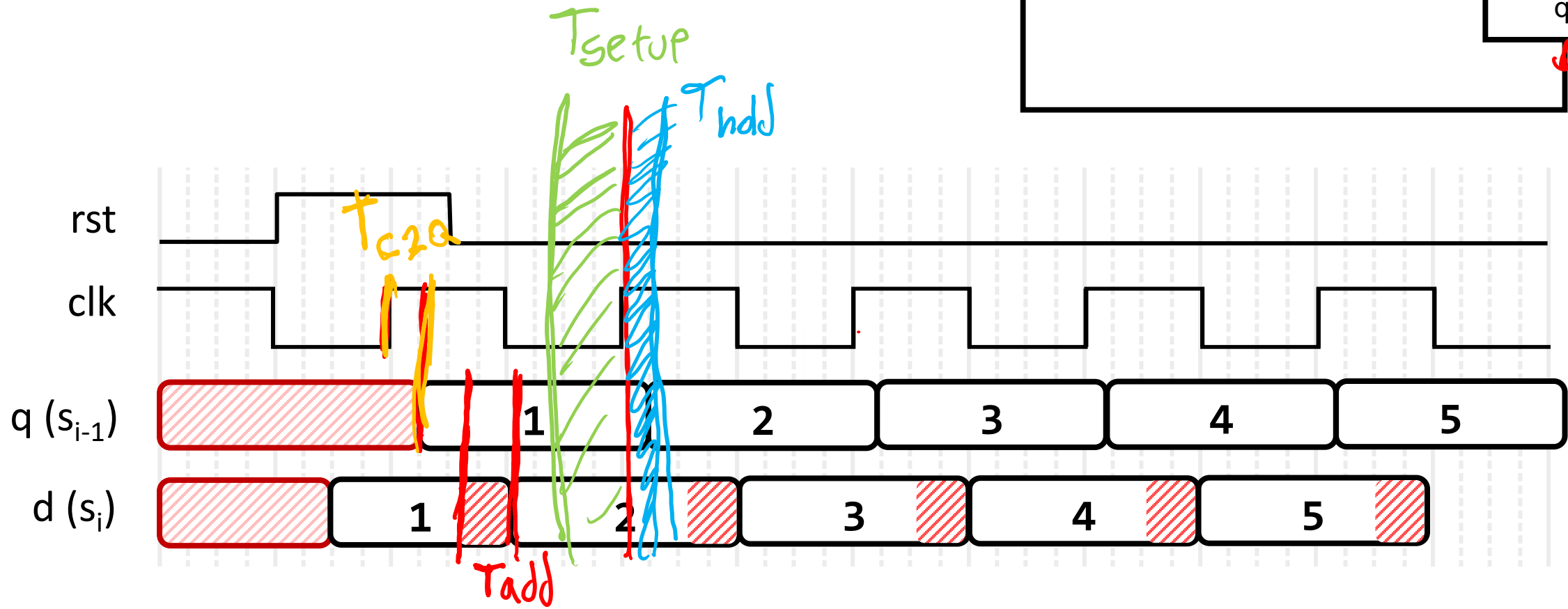
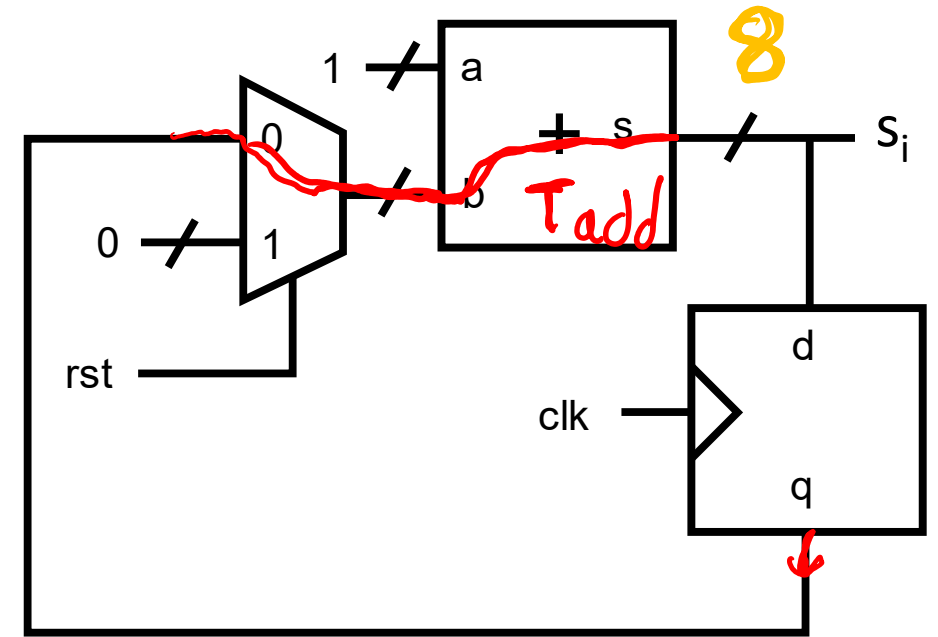
- *Setup Time*: how long the input must be stable *before* the CLK trigger for proper input read
- *Hold Time*: how long the input must be stable *after* the CLK trigger for proper input read
- *“CLK-to-Q” Delay*: how long it takes the output to change, measured from the CLK trigger

# Flip-Flop Timing Behavior

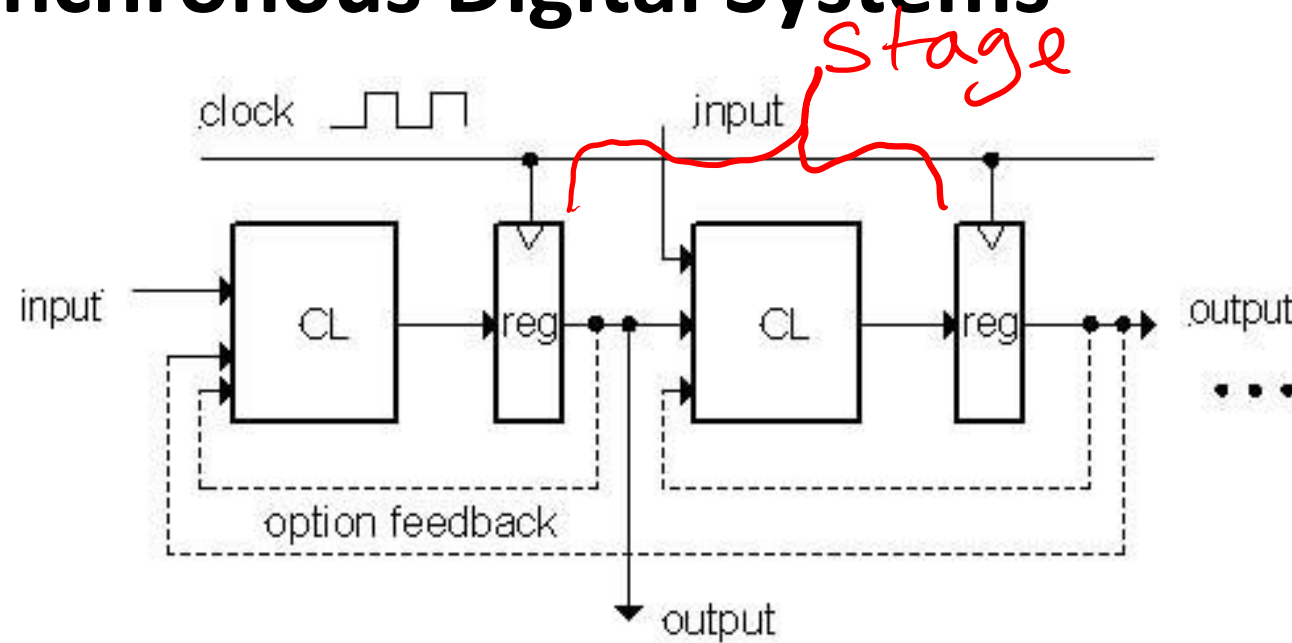


# Stopwatch: timing analysis

- As bits ripple through adder,  $S_i$  is temporarily wrong!
- BUT! Register always captures correct value
- In good circuits, instability never happens around rising edge of CLK



# Model for Synchronous Digital Systems



- ❖ Combinational logic blocks separated by registers
  - Clock signal connects only to sequential logic elements
  - Feedback is optional depending on application
- ❖ How do we ensure proper behavior?
  - How fast can we run our clock?

# When Can the Input Change?

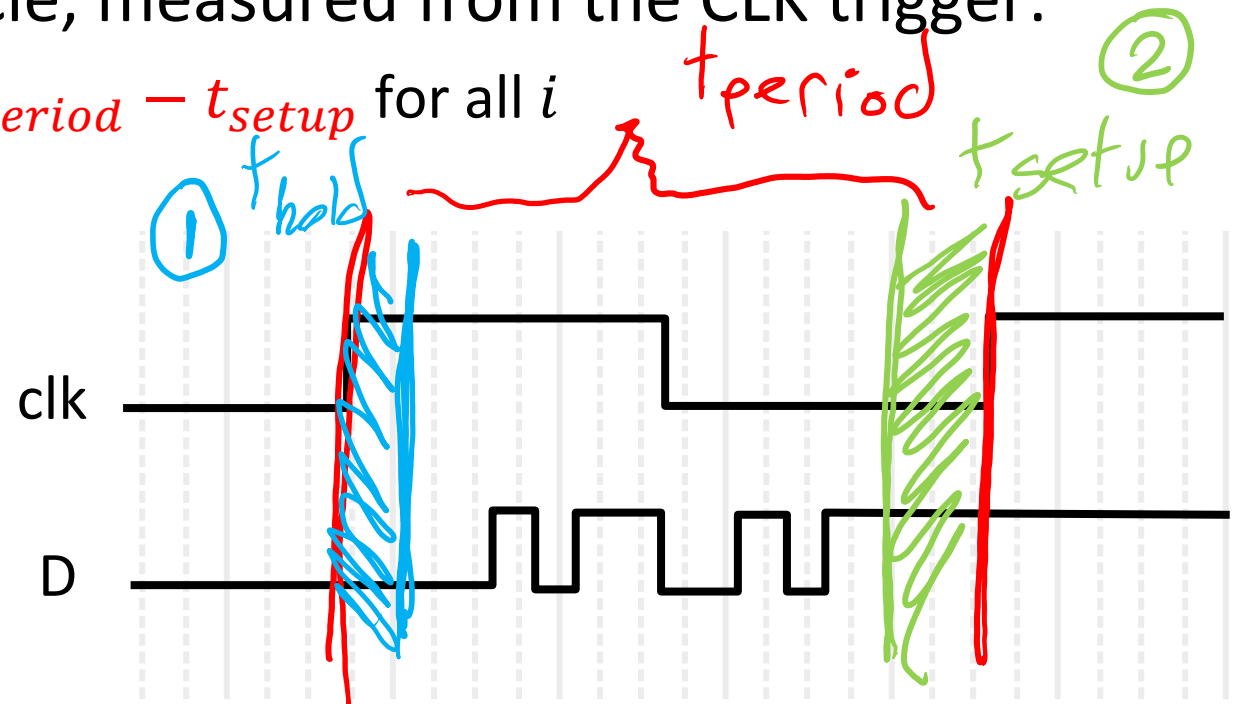
- ❖ When a register input changes shouldn't violate hold time ( $t_{hold}$ ) or setup time ( $t_{setup}$ ) constraints within a clock period ( $t_{period}$ )
- ❖ Let  $t_{input,i}$  be the time it takes for the input of a register to change for the  $i$ -th time in a single clock cycle, measured from the CLK trigger:

- Then we need  $t_{hold} \leq t_{input,i} \leq t_{period} - t_{setup}$  for all  $i$

- Two separate constraints!

$$t_{hold} \leq t_{min}$$

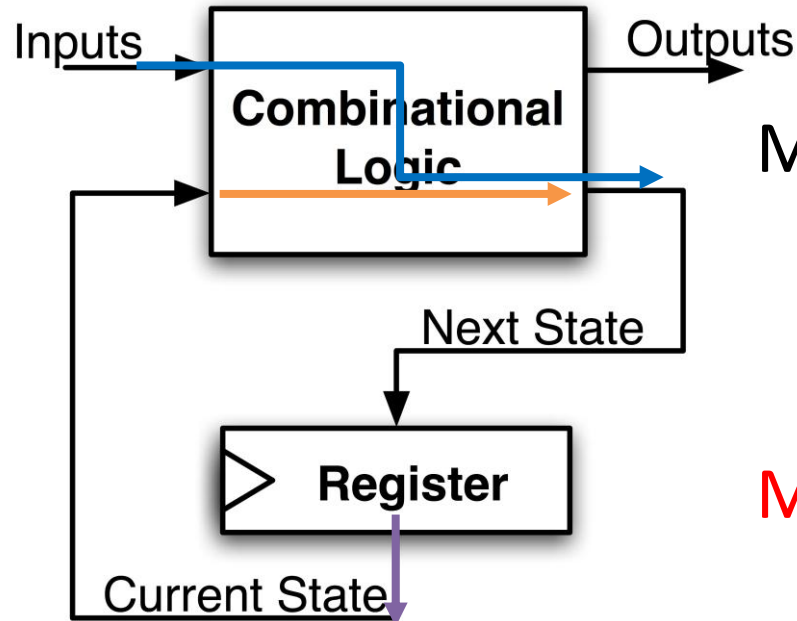
$$t_{max} \leq t_{period} - t_{setup}$$





# Minimum Delay

- ❖ If shortest path to register input is too short, might violate hold time constraint
  - Input could change before state is “locked in”
  - Particularly problematic with *asynchronous* signals

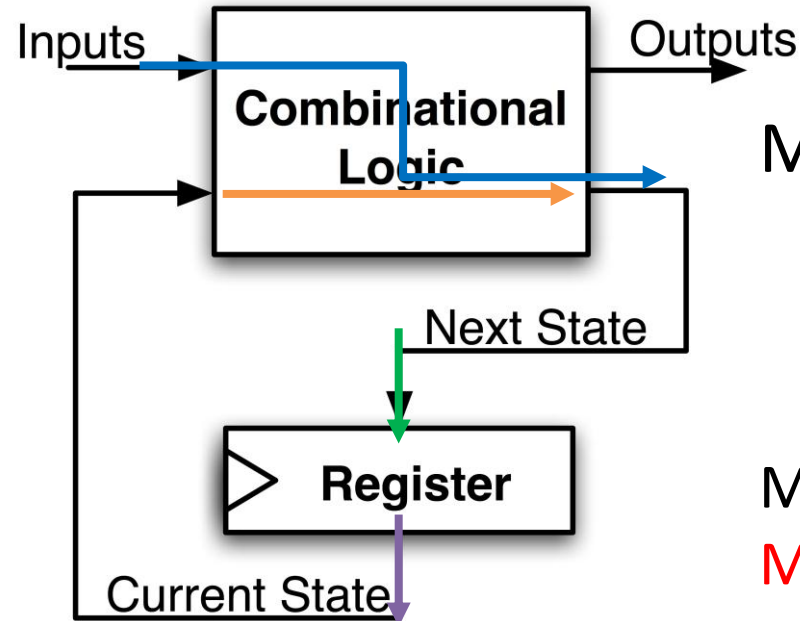


$$\text{Min Delay} = \min(\text{CLK-to-Q Delay} + \text{Min CL Delay}, \text{Min CL Delay})$$

$$\text{Min Delay} \geq \text{Hold Time}$$

# Maximum Clock Frequency

- ❖ What is the max frequency of this circuit?
  - Limited by how much time needed to get correct Next State to Register ( $t_{setup}$  constraint)



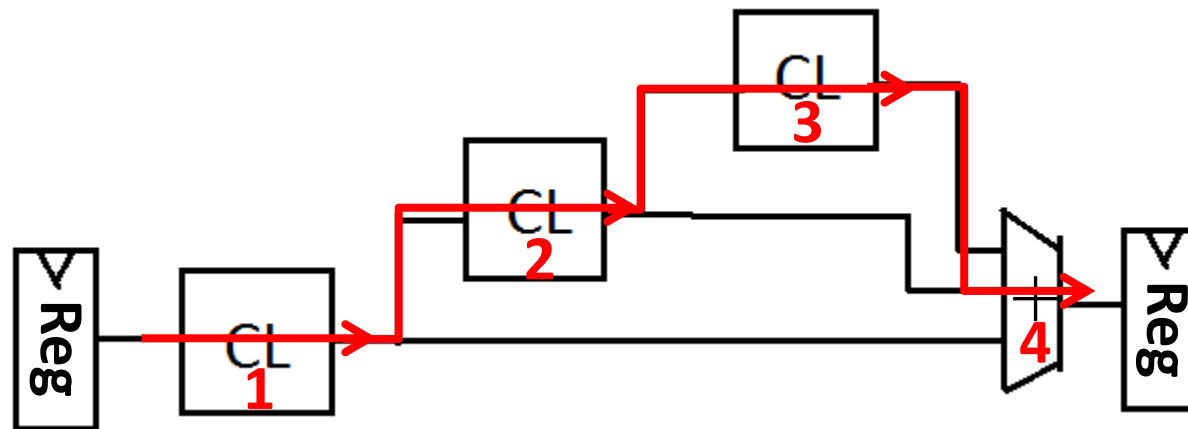
$$\text{Max Delay} = \max(\text{CLK-to-Q Delay} + \text{Max CL Delay}, \text{Max CL Delay})$$

$$\text{Min Period} = \text{Max Delay} + \text{Setup Time}$$

$$\text{Max Freq} = 1/\text{Min Period}$$

# The Critical Path

- ❖ The *critical path* is the longest delay between *any* two registers in a circuit
- ❖ The clock period must be *longer* than this critical path, or the signal will not propagate properly to that next register



**Critical Path =**

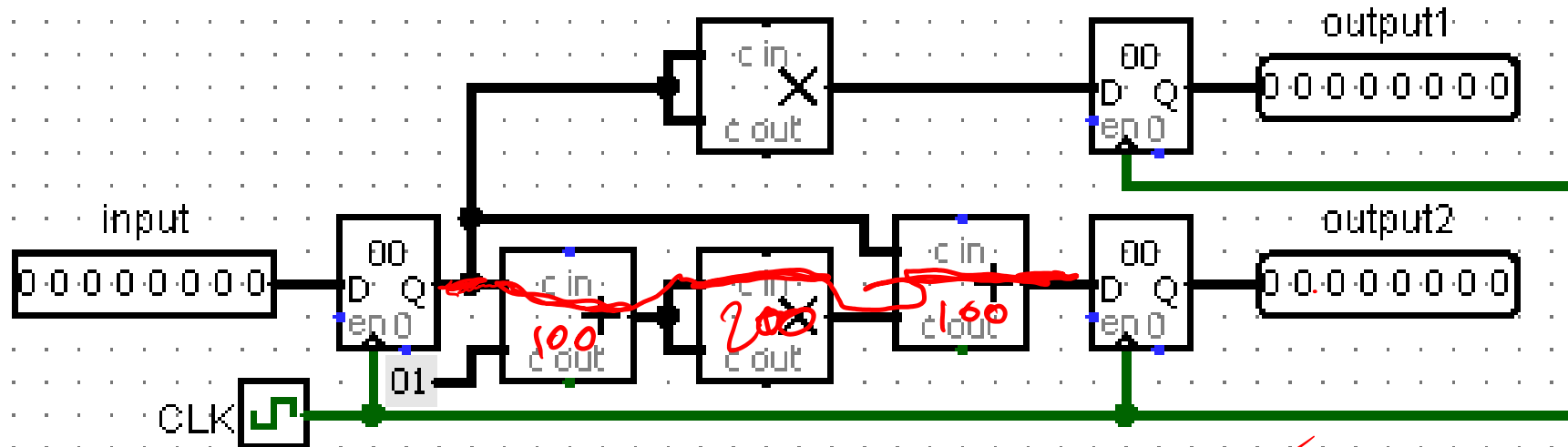
CLK-to-Q Delay  
+ CL Delay 1  
+ CL Delay 2  
+ CL Delay 3  
+ Adder Delay  
+ Setup Time

# Practice Question

$$t_{\text{hold}} \leq t_{\text{CLmin}}$$

$$t_{\text{CLmax}} \leq t_{\text{period}} - t_{\text{setup}}$$

- ❖ We want to run on 1 GHz processor.  $t_{\text{add}} = 100$  ps.  $t_{\text{mult}} = 200$  ps.  $t_{\text{setup}} = t_{\text{hold}} = 50$  ps. What is the maximum  $t_{\text{clk-to-q}}$  we can use?



$$t_{\text{clk-to-q}} + 100 + 200 + 100 \leq 1000 - 50 \quad 950$$

$$t_{\text{clk-to-q}} \leq 550$$

(A) 550 ps

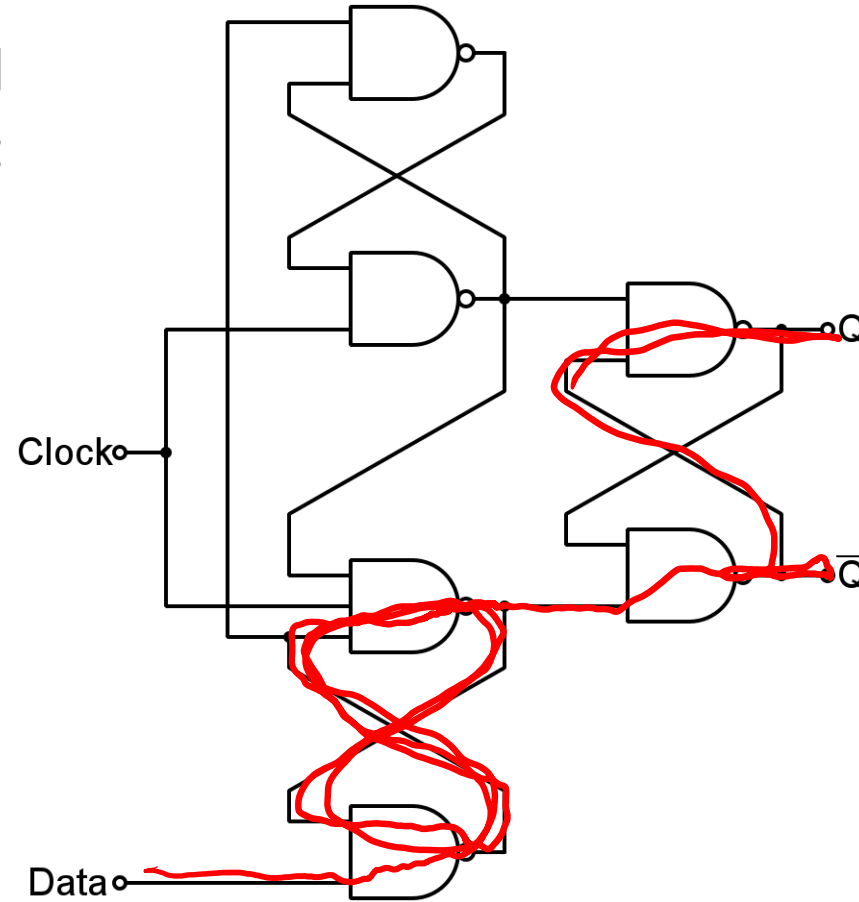
(B) 750 ps

(C) 500 ps

(D) 700 ps

# Where Do Timing Constraints Come From?

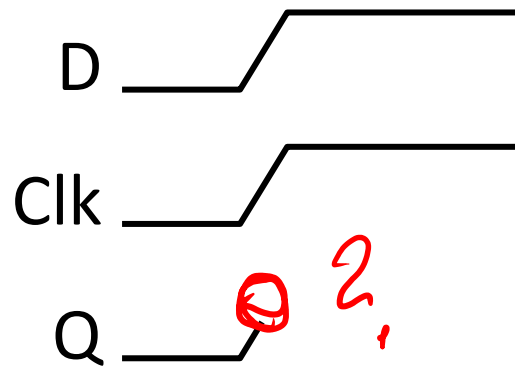
## Edge-triggered D flip-flop:



By Nolanjshettle at English Wikipedia, CC BY-SA 3.0,  
<https://commons.wikimedia.org/w/index.php?curid=40852354>

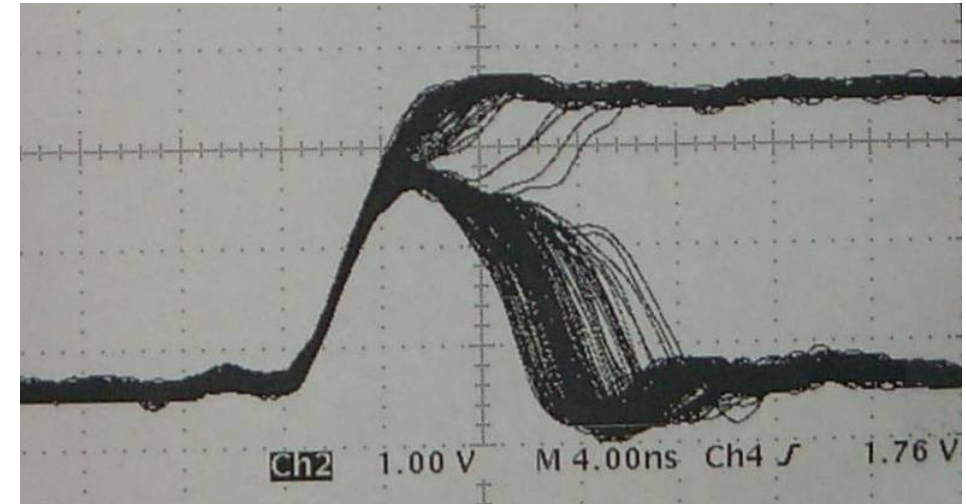
# Flip-Flop Realities: External Inputs

- ❖ External inputs aren't synchronized to the clock
  - If not careful, can violate timing constraints
- ❖ What happens if input changes around clock trigger?

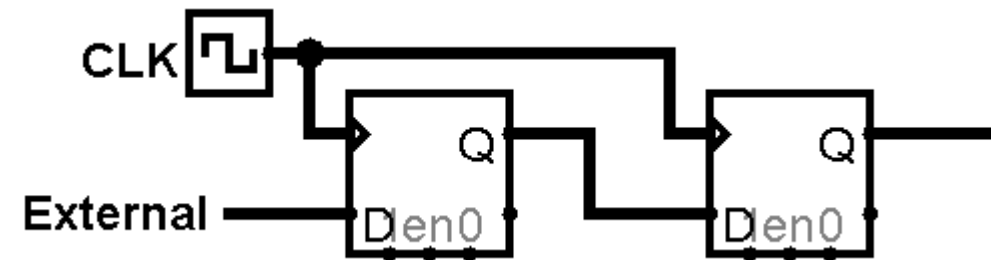


# Flip-Flop Realities: Metastability

- ❖ **Metastability** occurs when a digital feedback loop settles into an **unstable equilibrium** storing a **non-binary** voltage
  - Can last for a potentially unbounded amount of time
  - Will randomly decay to a '0' or a '1'....probably
- ❖ State elements can help reject transients
  - Longer chains = more rejection, but longer signal delay



<https://www.cl.cam.ac.uk/teaching/1011/SysOnChip/slides/sp3soc/parts/zhp6e41885b8.html>



# Summary of Timing Terms

- ❖ **Clock:** steady square wave that synchronizes system
- ❖ **Flip-flop:** one bit of state that samples every rising edge of CLK (positive edge-triggered)
- ❖ **Register:** several bits of state that samples on rising edge of CLK (positive edge-triggered); often has a RESET
- ❖ **Setup Time:** when input must be stable *before* CLK trigger
- ❖ **Hold Time:** when input must be stable *after* CLK trigger
- ❖ **CLK-to-Q Delay:** how long it takes output to change from CLK trigger



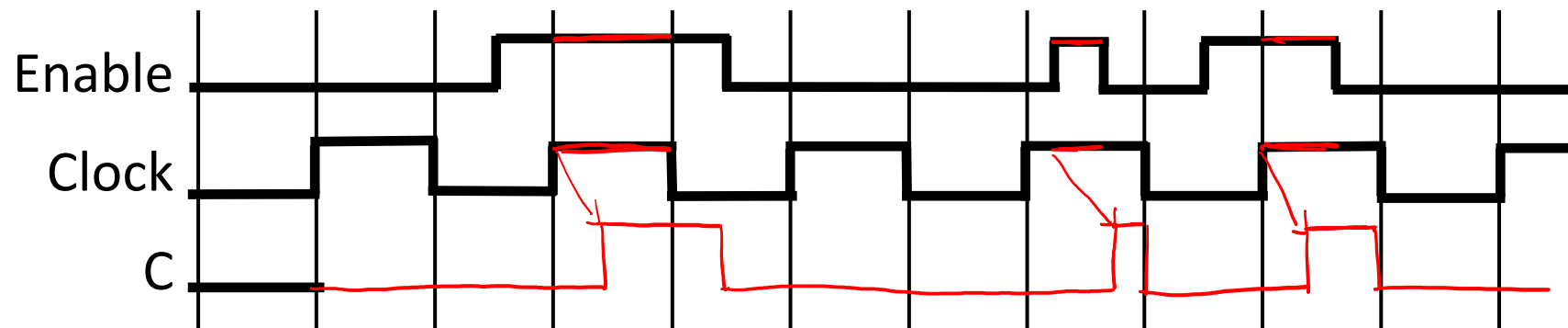
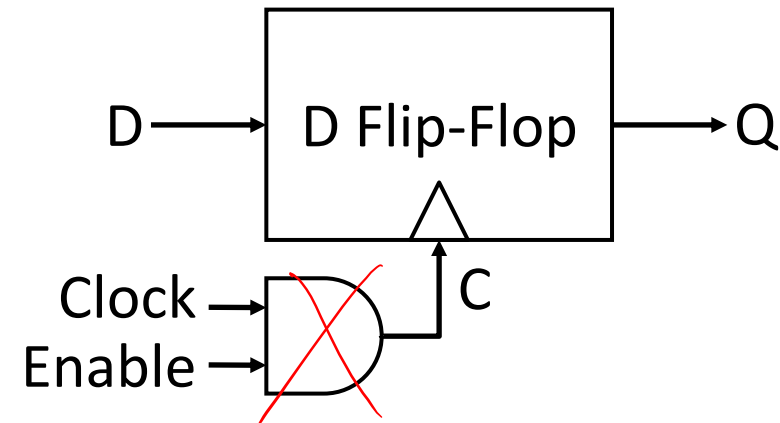
# Extra: Clock Divider (not for simulation)

## ❖ Why/how does this work?

```
// divided_clocks[0]=25MHz, [1]=12.5Mhz, ...  
module clock_divider (clock, divided_clocks);  
    input logic        clock;  
    output logic [31:0] divided_clocks;  
  
    initial  
        divided_clocks = 0;  
  
    always_ff @(posedge clock)  
        divided_clocks <= divided_clocks + 1;  
  
endmodule // clock_divider
```

## Extra: Flip-Flop Realities: Gating the Clock

- ❖ Delay can cause part of circuit to get out of sync with rest
  - More timing headaches!
  - Adds to *clock skew*
- ❖ Hard to track non-uniform triggers



- ❖ **NEVER GATE THE CLOCK!!!**