

Intro to Digital Design

Sequential Logic

Instructor: Chris Thachuk

Teaching Assistants:

Jiuyang Lyu

Stephanie Osorio-Tristan

Nandini Talukdar

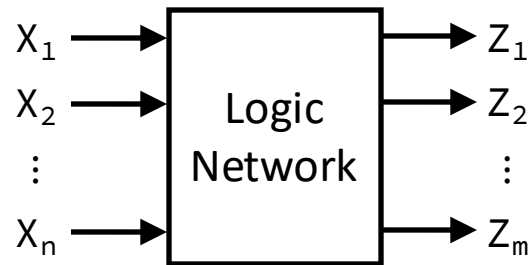
Wen Li

Relevant Course Information

- ❖ Lab 3 Demos due during your assigned demo slots
 - Don't forget to submit your lab materials *before* Wednesday at 2:30 pm, regardless of your demo time
- ❖ Lab 4 – Extension of Lab 3 using 7-seg displays
- ❖ Quiz 1 is next week in lecture
 - Last 20 minutes, worth 10% of your course grade
 - On Lectures 1-3: CL, K-maps, Waveforms, and Verilog
 - Past Quiz 1 (+ solutions) on website: Course Info → Quizzes

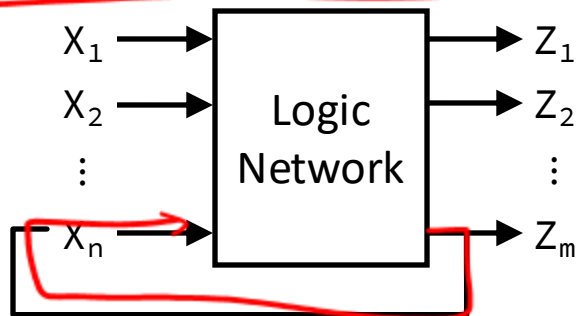
Synchronous Digital Systems (SDS)

❖ Combinational Logic (CL)



- Network of logic gates without feedback
- Outputs are functions only of inputs

❖ Sequential Logic (SL)



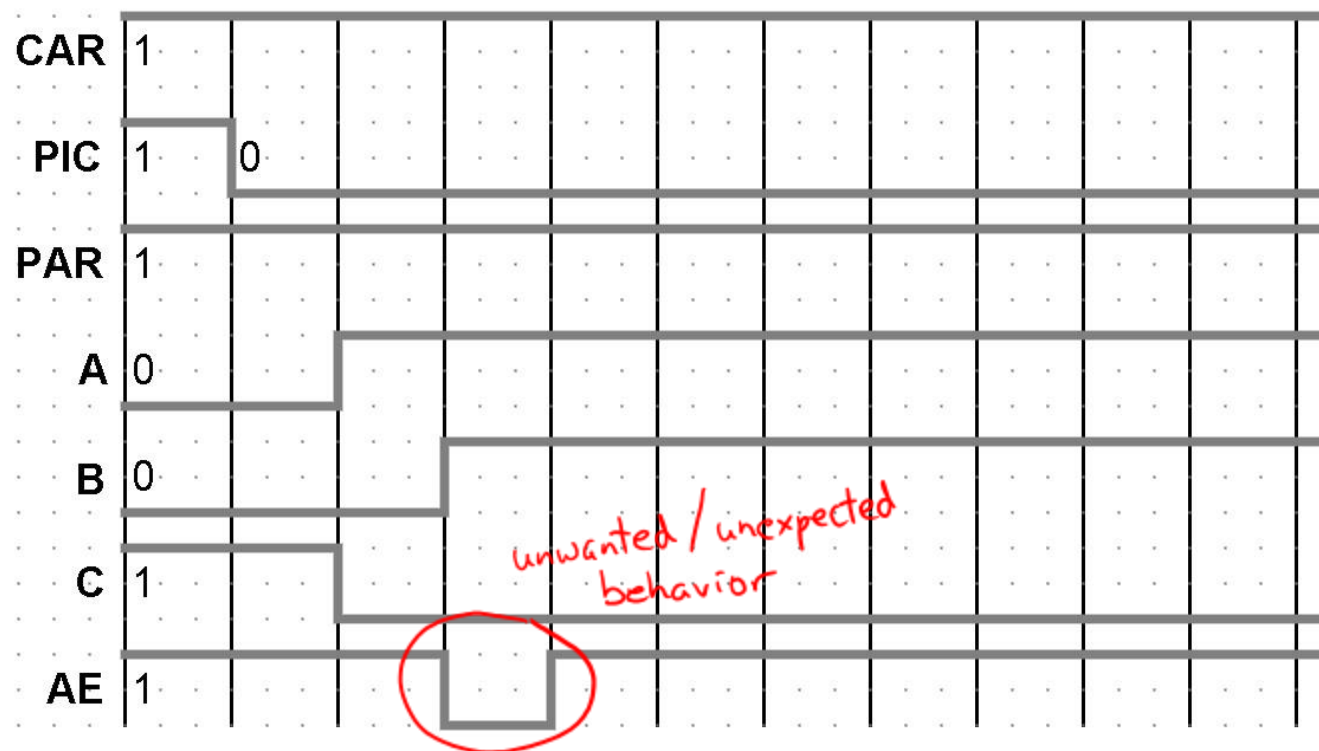
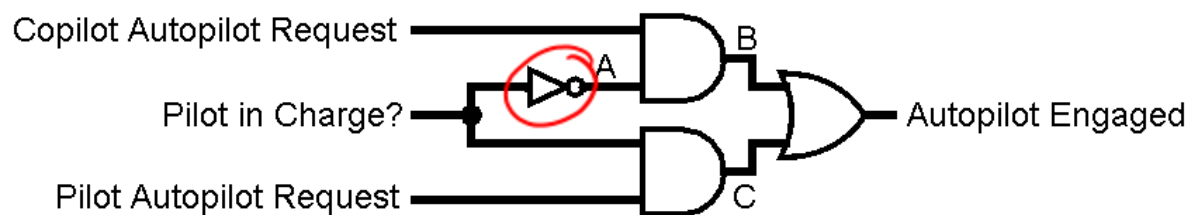
- The presence of feedback introduces the notion of “state”
- Circuits that can “remember” or store information

Uses for Sequential Logic

- ❖ Place to store values for some amount of time:
 - Registers
 - Memory
- ❖ *timing!* *Help control flow of information between combinational logic blocks*
 - Hold up the movement of information to allow for orderly passage through CL

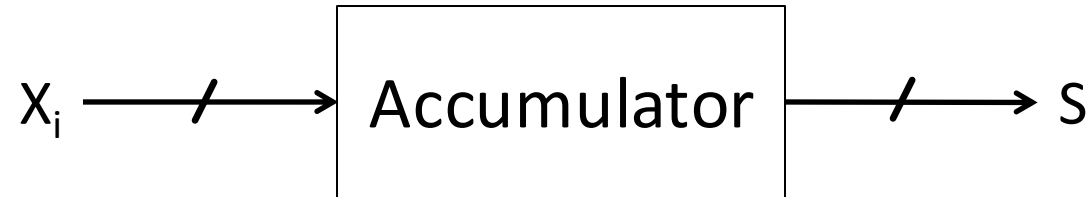
Control Flow of Information?

- ❖ Circuits can temporarily go to incorrect states!



Accumulator Example

- ❖ An example of why we would need to control the flow of information.



- ❖ Want:

```
S = 0; initialize?  
for (i = 0; i < n; i++)  
    S = S + X_i;
```

stop condition?

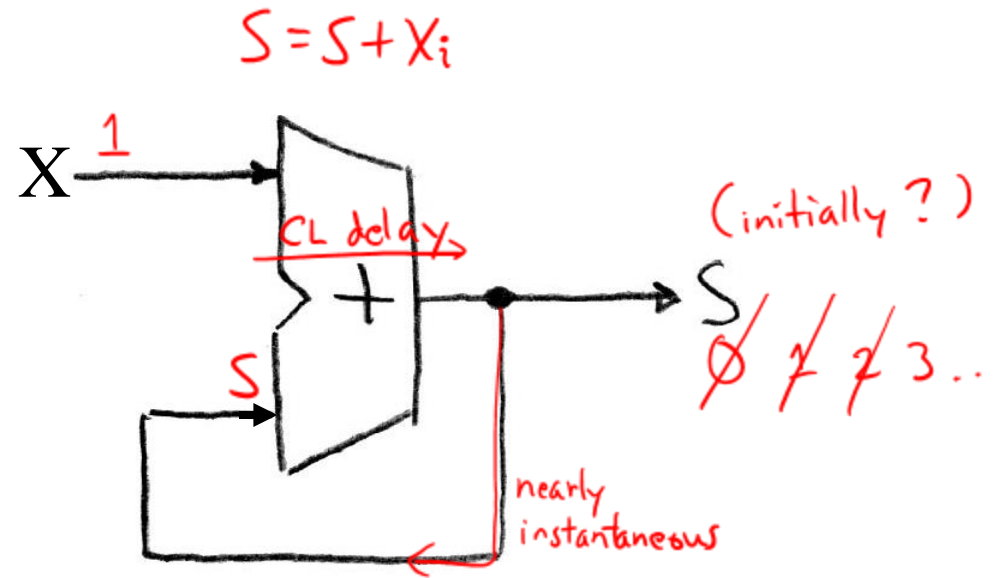
sequence of inputs

- ❖ Assume:
 - Each X value is applied in succession, one per cycle
 - The sum since cycle 0 is present on S

Accumulator: First Try

❖ Does this work?

■ No

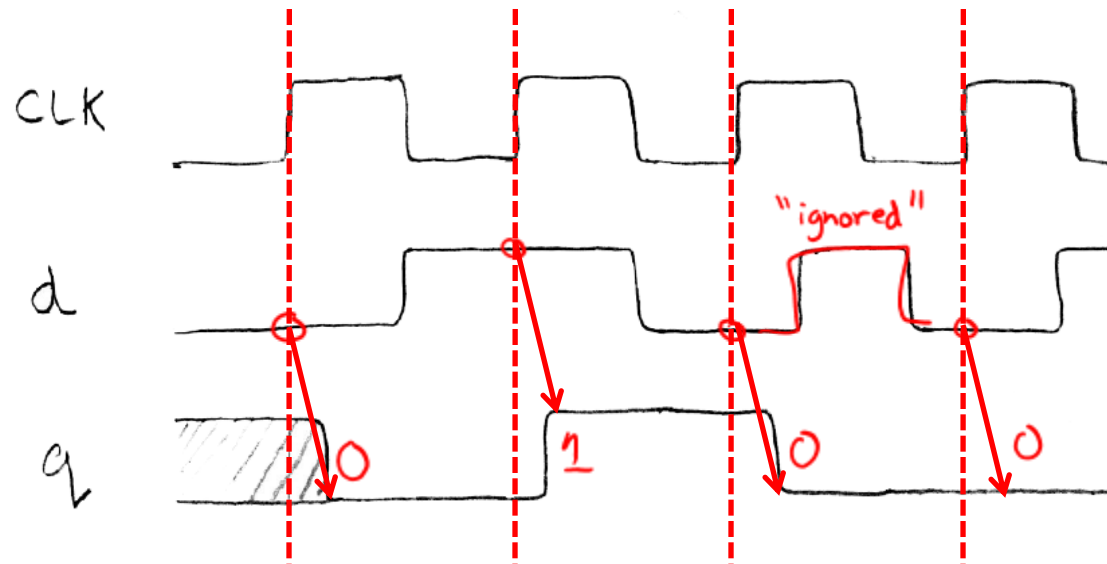
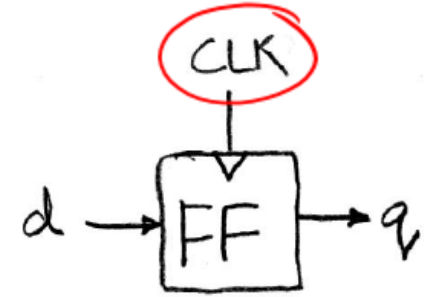


- 1) How to control the next iteration of the 'for' loop?
- 2) How do we accomplish ' $S = 0$ '?

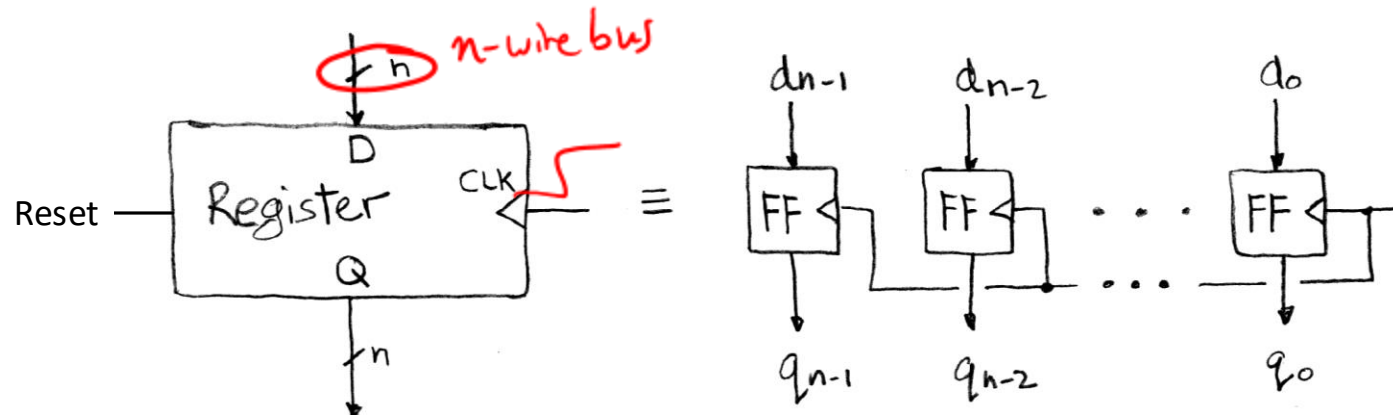
State Element: Flip-Flop

❖ Positive edge-triggered D-type flip flop

- On the rising edge of the clock ($0 \rightarrow 1$), input d is sampled and transferred to the output q
- At all other times, the input d is ignored and the previously sampled value is retained

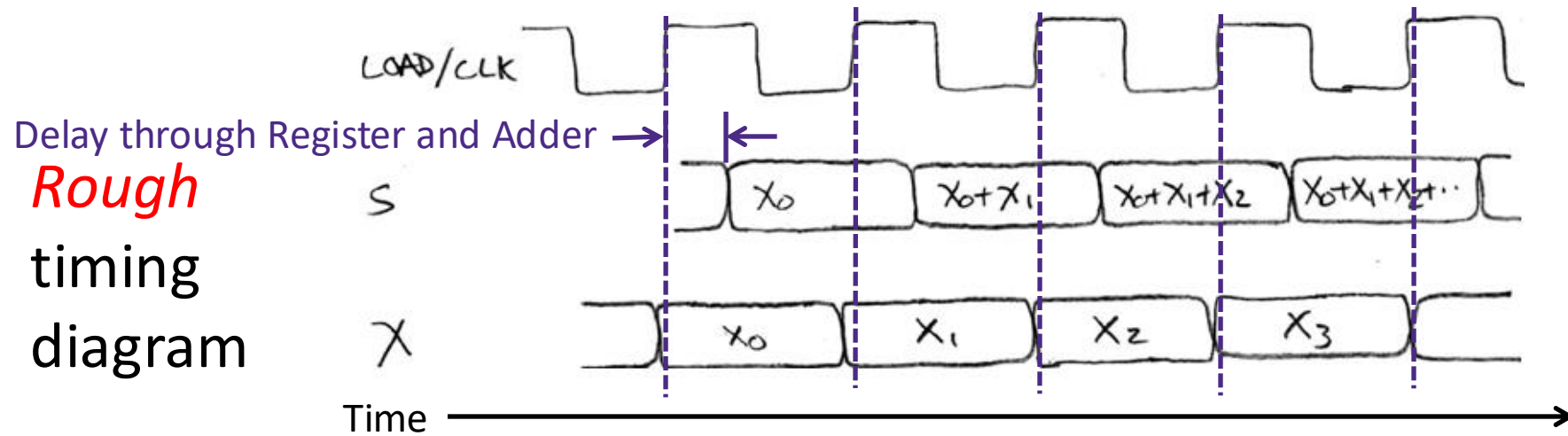
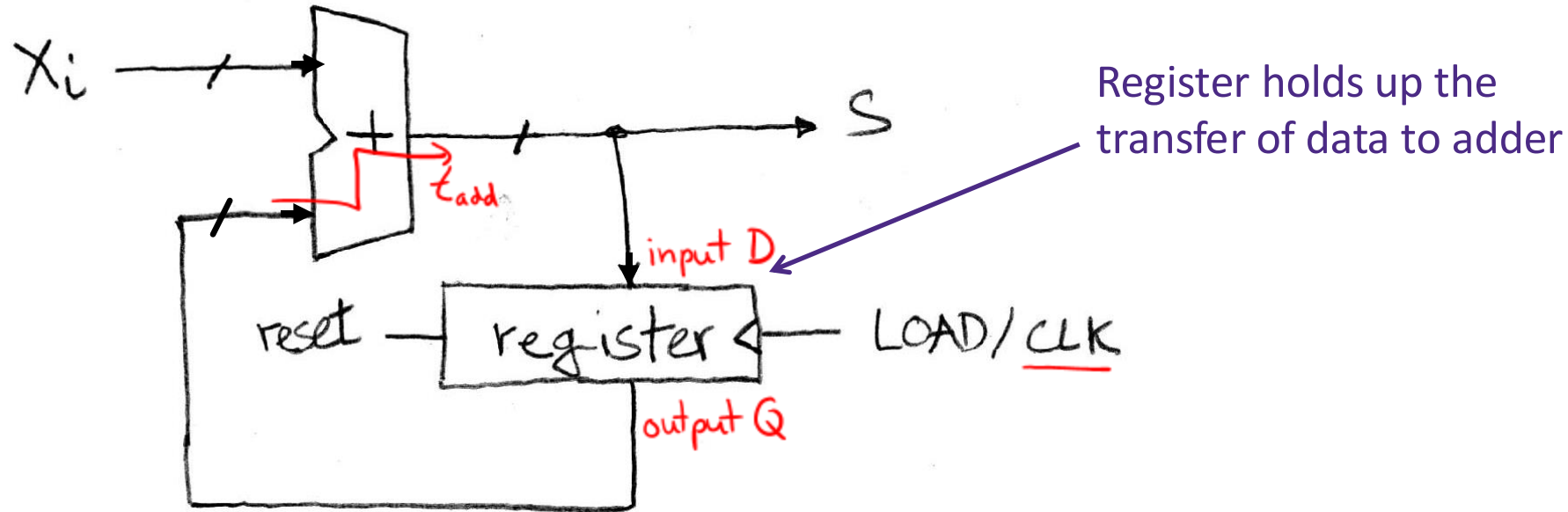


State Element: Register



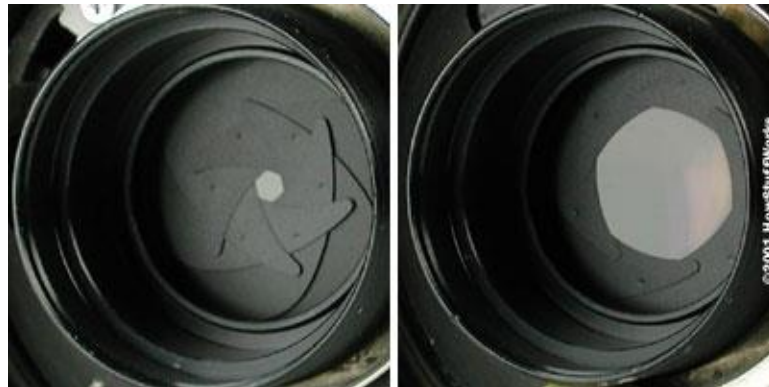
- ❖ n instances of flip-flops together
 - One for every bit in input/output bus width
- ❖ Output Q resets to zero when Reset signal is high *during* clock trigger
 - Some extra circuitry required for this

Accumulator: Second Try



Flip-Flop Timing Terminology (1/2)

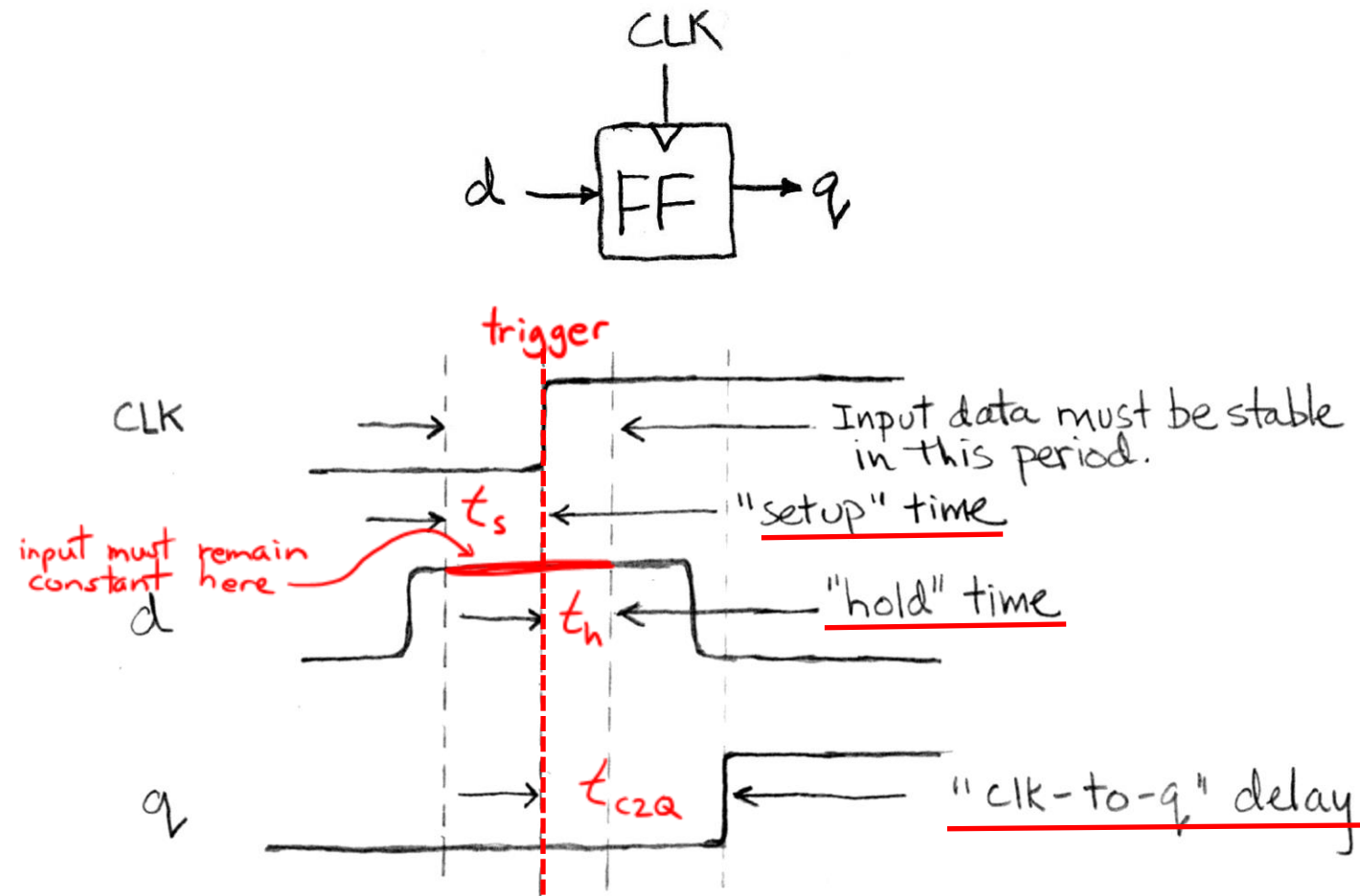
- ❖ Camera Analogy: non-blurry digital photo
 - *Don't move* while camera shutter is opening
 - *Don't move* while camera shutter is closing
 - *Check for blurriness* once image appears on the display



Flip-Flop Timing Terminology (2/2)

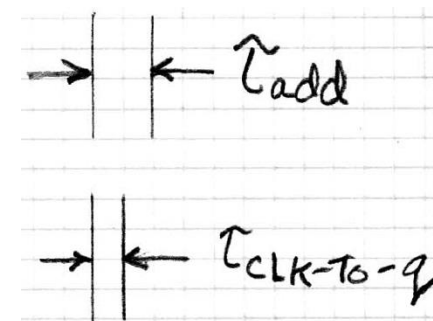
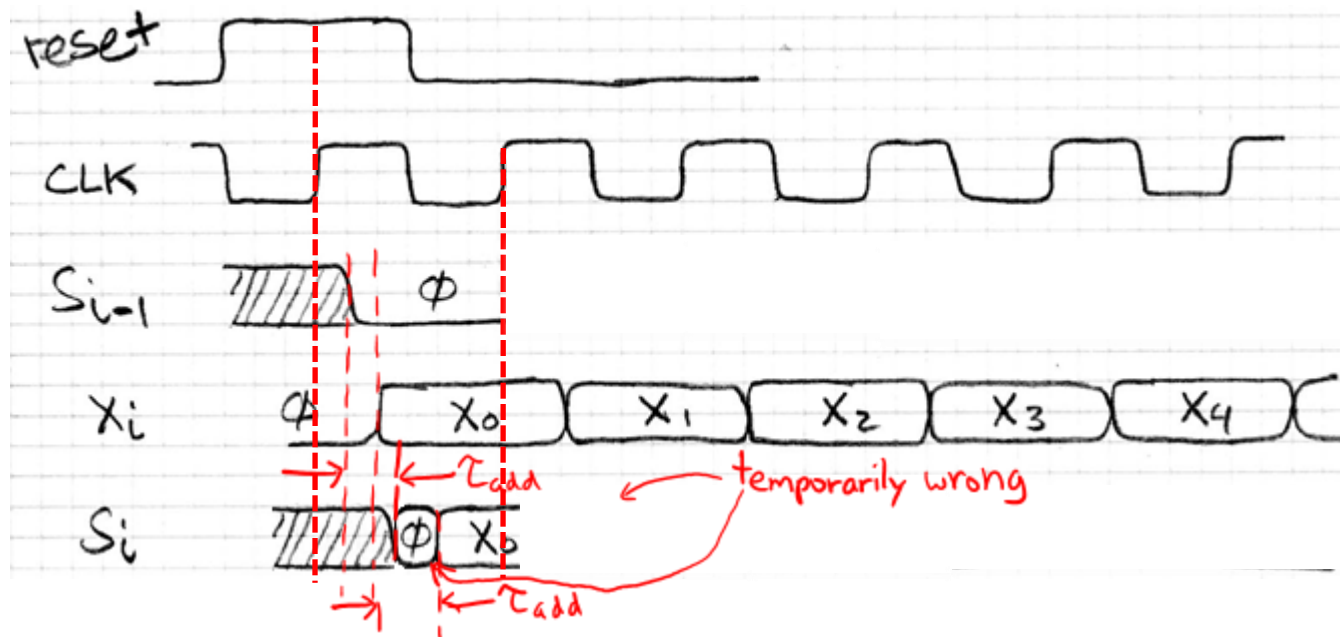
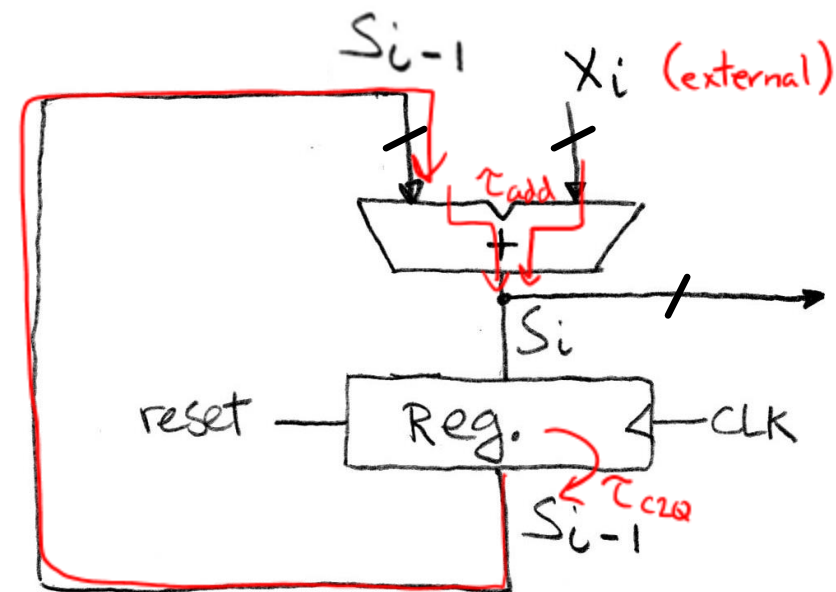
- ❖ Now applied to sequential logic elements:
 - **Setup Time:** how long the input must be stable *before* the CLK trigger for proper input read
 - **Hold Time:** how long the input must be stable *after* the CLK trigger for proper input read
 - **“CLK-to-Q” Delay:** how long it takes the output to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

Flip-Flop Timing Behavior



Accumulator: Proper Timing

- ❖ reset signal shown
- ❖ X_i and S_{i-1} arrive at adder at different times
 - S_i becomes "wrong" temporarily but corrects before register captures its value
- ❖ Avoid input instability around rising edge of CLK



Review Question

❖ Which of the following statements is TRUE?

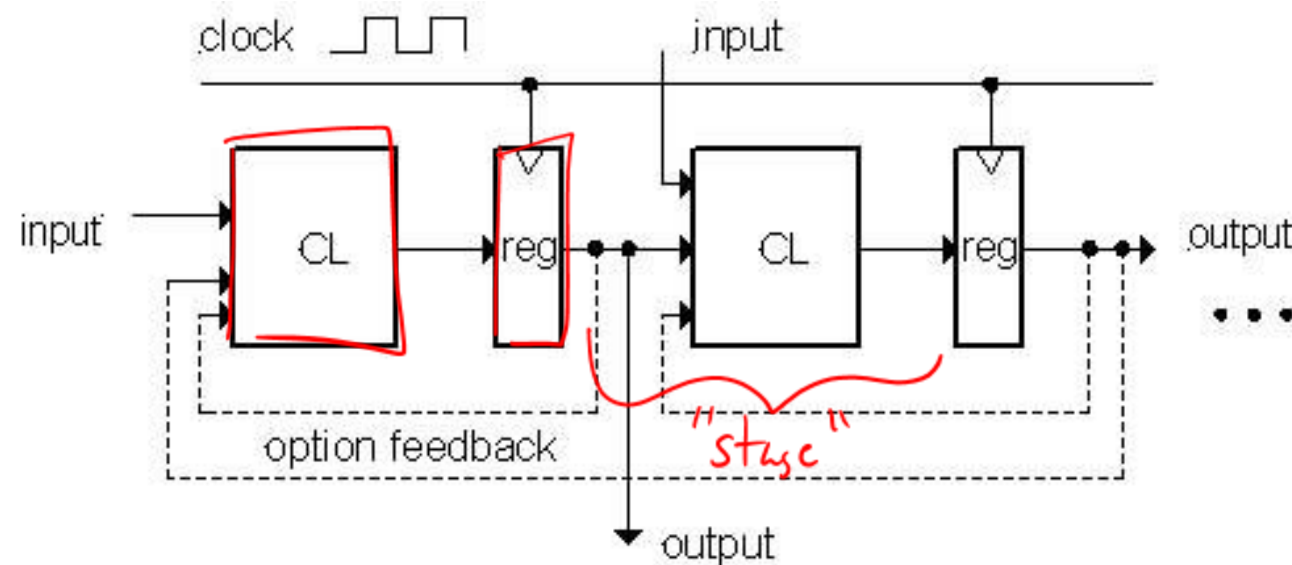
(A) The input to a flip-flop must remain stable throughout the ~~CLK to-Q delay~~. *setup & hold times*

(B) A flip-flop ~~switches between 0 and 1~~ on each trigger. *input D → output Q*

(C) In a SDS, we only need to know setup time, hold time, and clk-to-q delay constants to ensure correct behavior. *also need CL delays, clock period, external input timing, etc.*

(D) **None of the above.**

Model for Synchronous Digital Systems



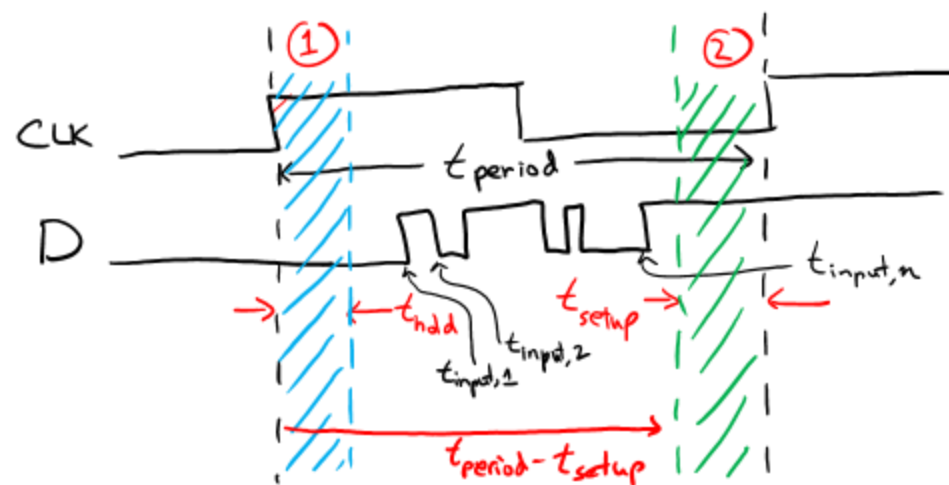
- ❖ Combinational logic blocks separated by registers
 - Clock signal connects only to sequential logic elements
 - Feedback is optional depending on application
- ❖ How do we ensure proper behavior?
 - How fast can we run our clock?

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 $t_{hold} \stackrel{\textcircled{1}}{\leq} t_{input,i} \stackrel{\textcircled{2}}{\leq} t_{period} - t_{setup}$ for all i
 - Two separate constraints!

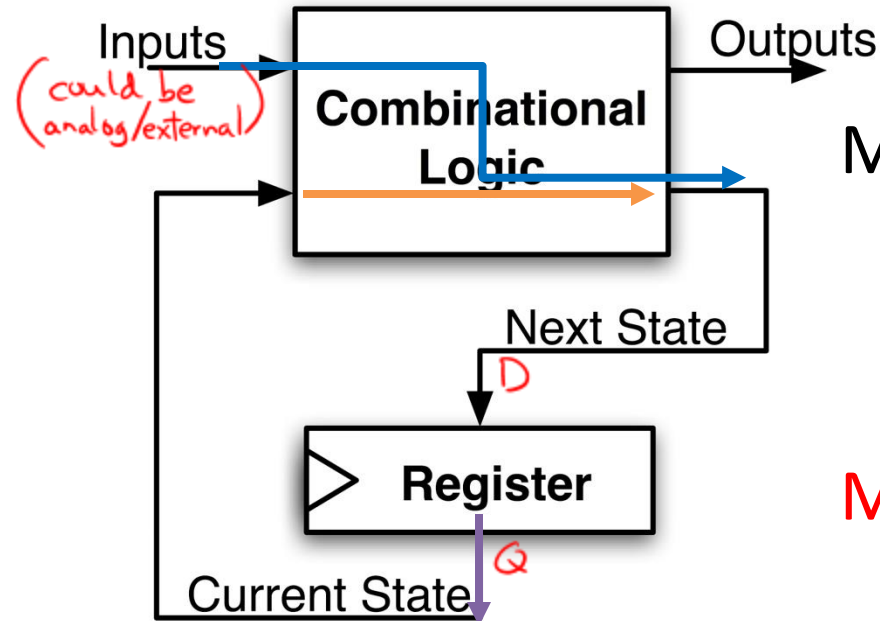
$$\textcircled{1} \quad t_{input,1} \geq t_{hold}$$

$$\textcircled{2} \quad t_{input,n} \leq t_{period} - t_{setup}$$



Minimum Delay

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

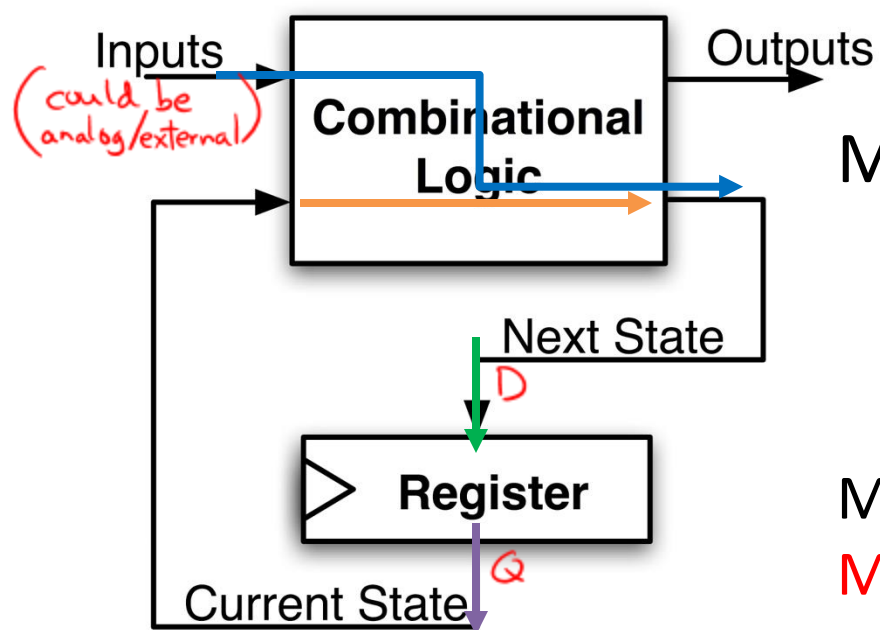


$$\text{Min Delay} = \min(\text{CLK-to-Q Delay} + \text{Min CL Delay}, \text{Min CL Delay})$$

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

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)

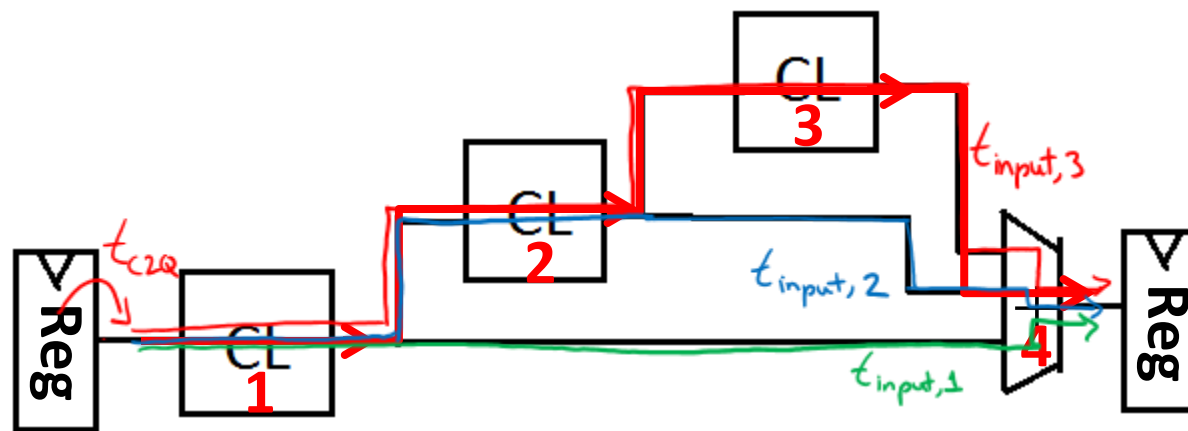


$$\text{Max Delay} = \max(\text{CLK-to-Q Delay} \\ + \text{Max CL Delay}, \\ + \text{Max CL Delay})$$

$$\text{Min Period} = \text{Max Delay} + \text{Setup Time}$$
$$\text{Max Freq} = 1/\text{Min Period}$$

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

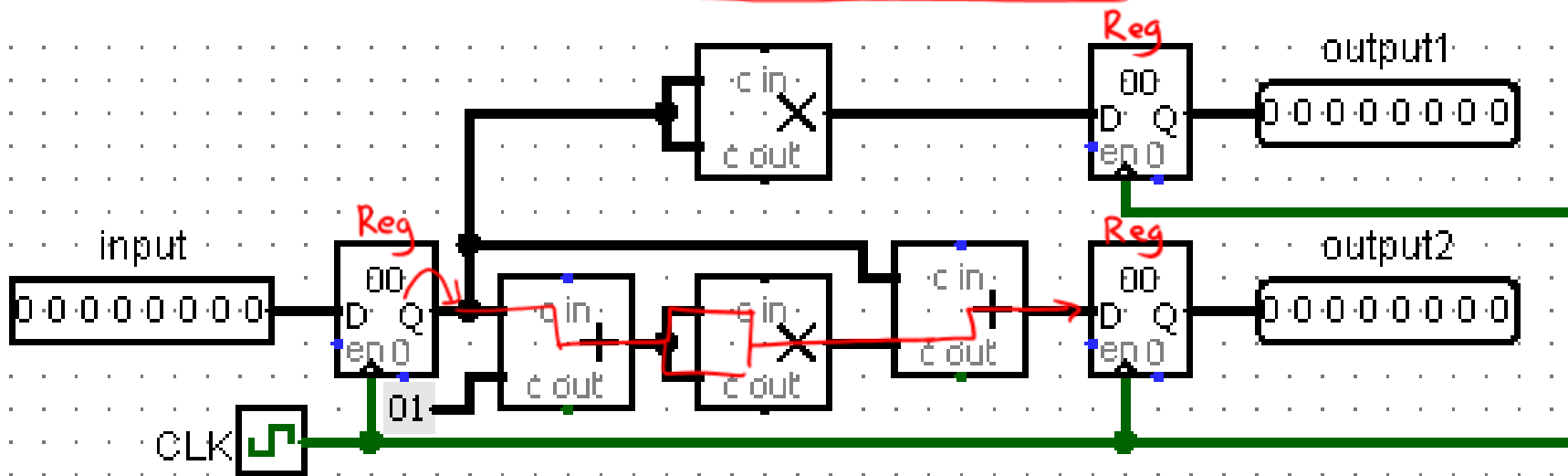
Practice Question

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

$f_{req} = 10^9 s^{-1} \rightarrow period = 10^{-9} s = 1 ns = 1000 ps$

milli 10^{-3}
 micro 10^{-6}
 nano 10^{-9}
 pico 10^{-12}

$t_{hold} \leq t_{input,i} \leq t_{period} - t_{setup}$



$t_{input,n} = t_{cq} + t_{add} + t_{mult} + t_{add} \leq t_{period} - t_{setup} \Rightarrow t_{cq} \leq 550 ps$

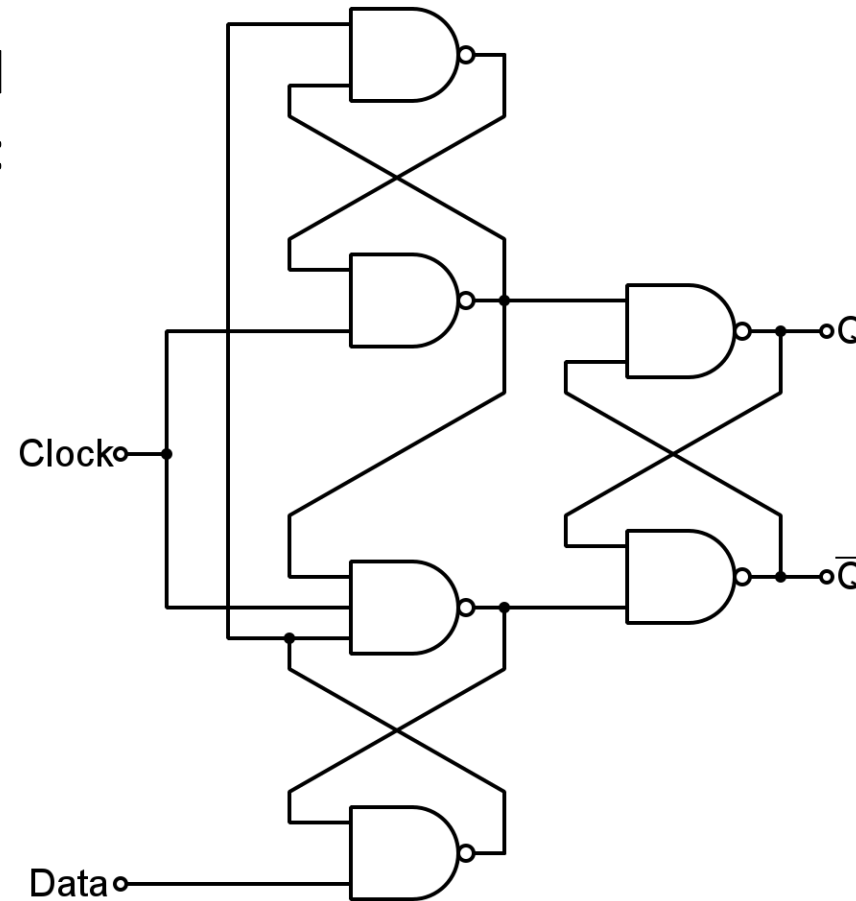
$\begin{matrix} 100 & 200 & 100 & 1000 & 50 \end{matrix}$

- (A) **550 ps** (B) 750 ps (C) 500 ps (D) 700 ps

Technology Break

Where Do Timing Terms Come From?

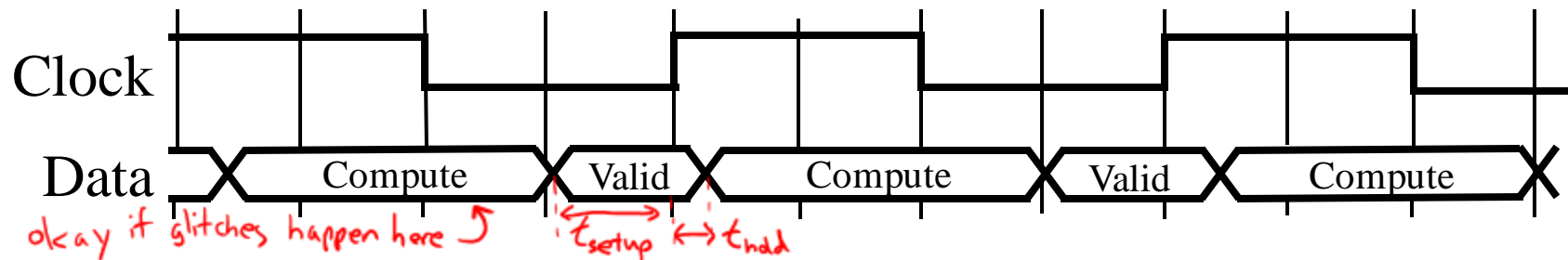
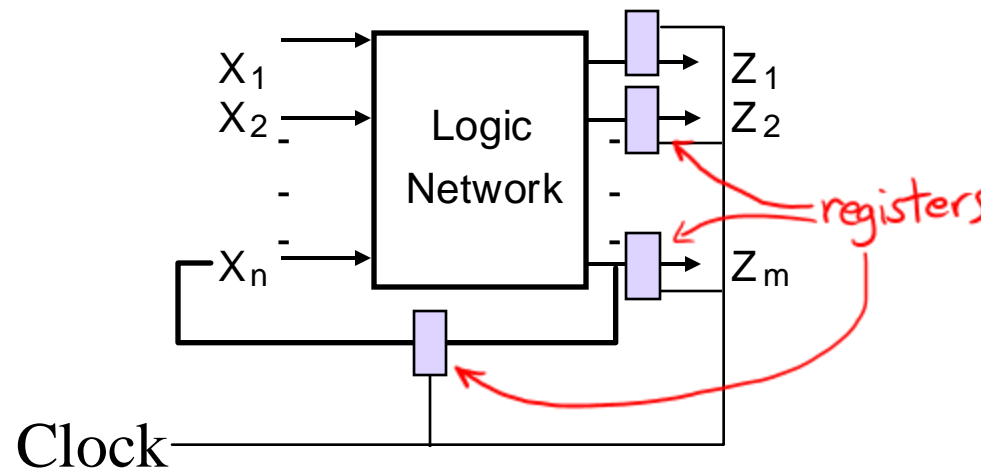
Edge-triggered
D flip-flop:



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

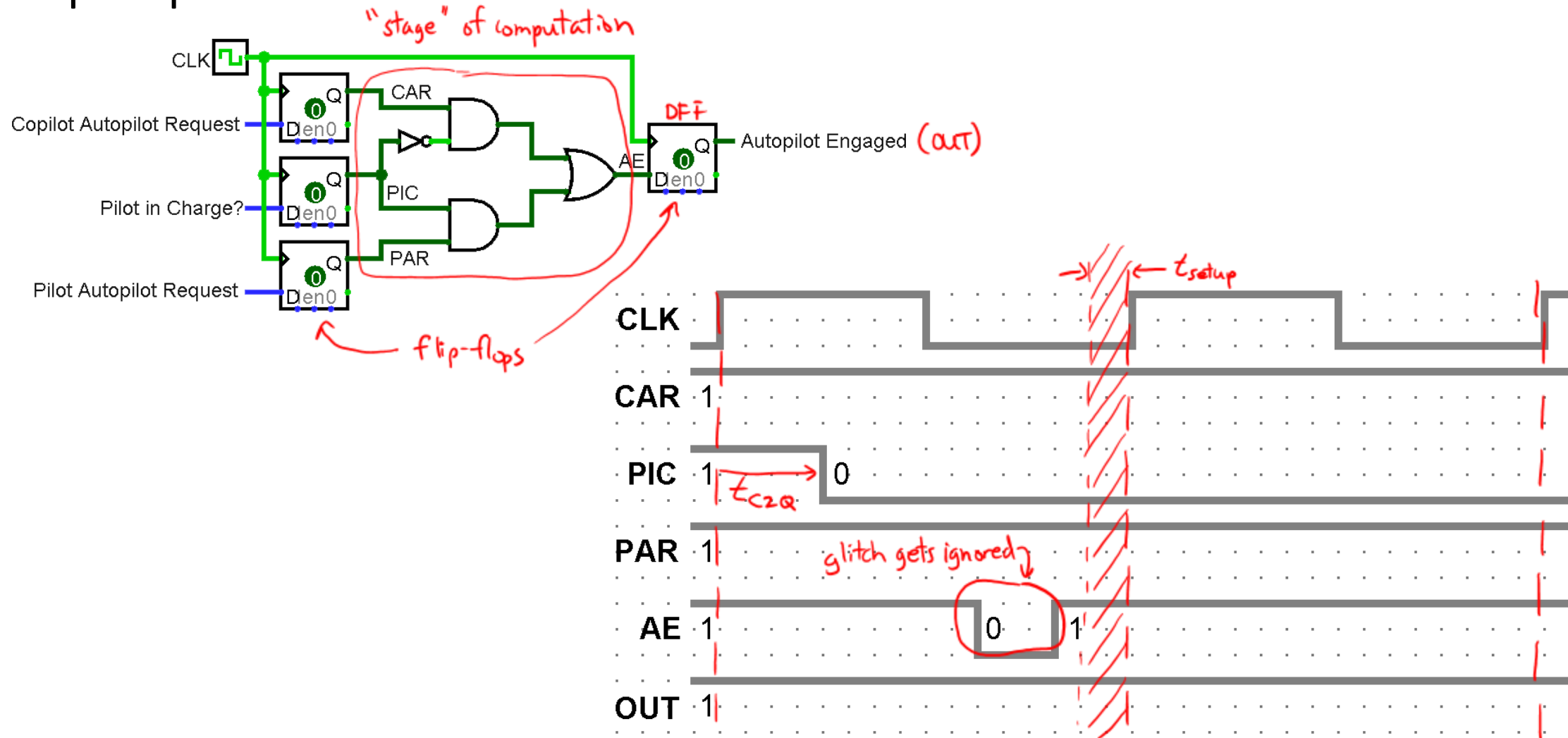
Safe Sequential Circuits

- ❖ Clocked elements on feedback, perhaps outputs
 - Clock signal synchronizes operation
 - Clocked elements hide glitches/hazards



Autopilot Revisited

- ❖ Flip-flops can “filter out” unintended behavior:

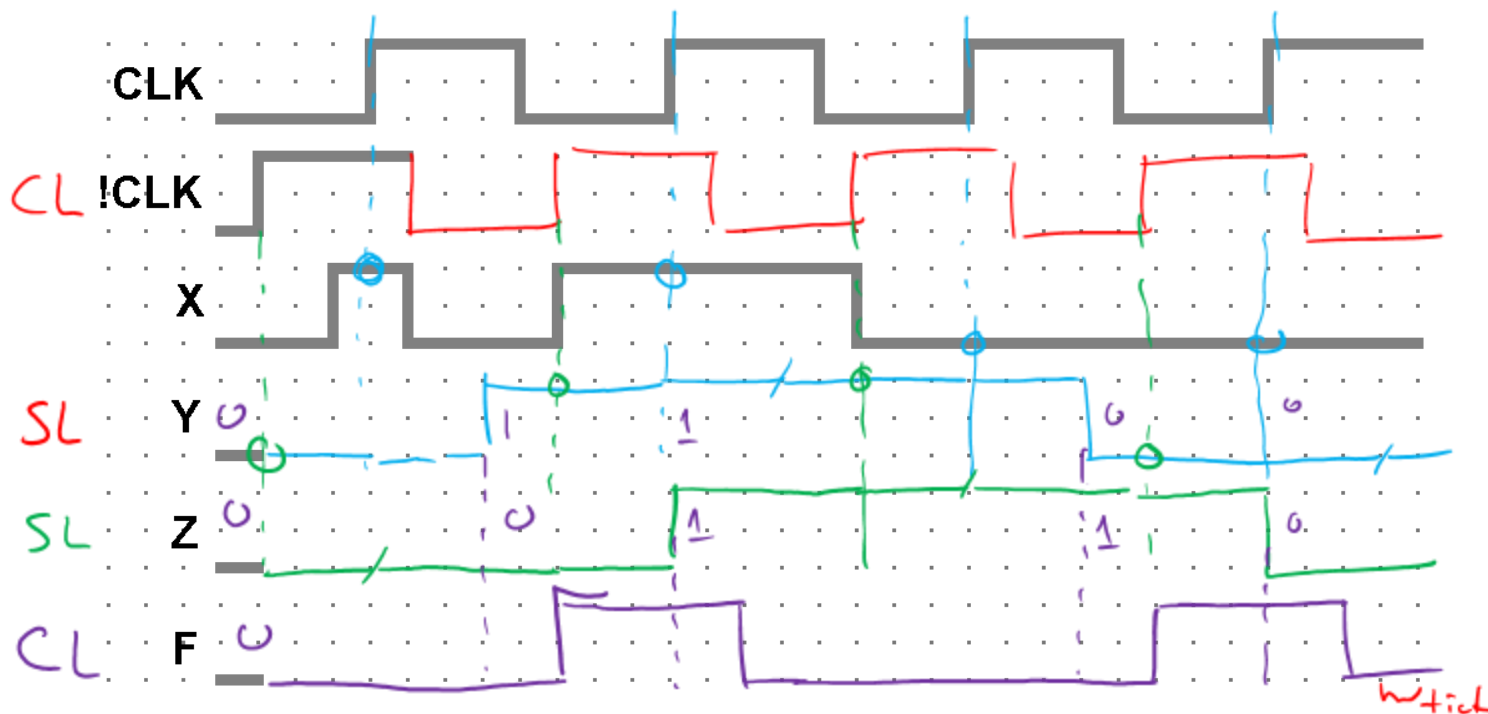
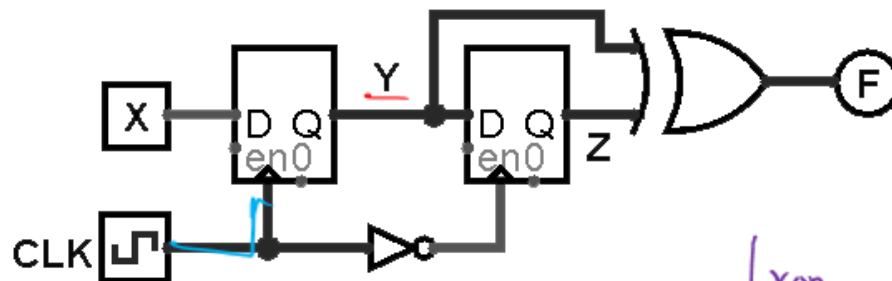


Waveform Diagrams Revisited

- ❖ Easiest to start with CLK on top
 - Solve signal by signal, from inputs to outputs
 - Can only draw the waveform for a signal if *all* of its input waveforms are drawn
- ❖ When does a signal update?
 - A state element^(SL) updates based on CLK triggers
 - A combinational element^(CL) updates ANY time ANY of its inputs changes

Example: SDS Waveform Diagram

- ❖ Assume: $t_{C2Q} = 3$ ticks, $t_{XOR} = 2$ ticks, $t_{NOT} = 1$ tick; $t_s = t_h = 0$
 - Note: clocking the gate is a *terrible* idea



		X or
0	0	0
0	1	1
1	0	1
1	1	0

Verilog: Basic D Flip-Flop, Register

```

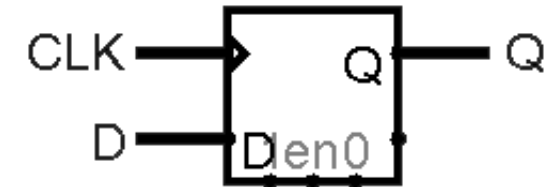
module basic_D_FF (q, d, clk);
  output logic q; // q is state-holding
  input logic d, clk;

  always_ff @(posedge clk)
    q <= d; // use <= for clocked elements
endmodule

```

react to rising edges of clk signal

use <= for clocked elements



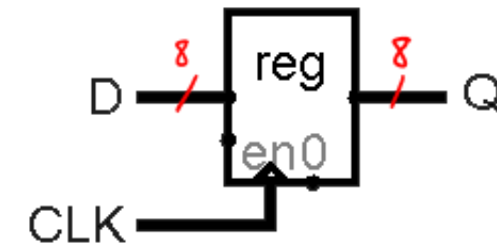
```

module basic_reg (q, d, clk);
  output logic [7:0] q;
  input logic [7:0] d;
  input logic clk;

  always_ff @(posedge clk)
    q <= d;
endmodule

```

bus widths of 8



Procedural Blocks

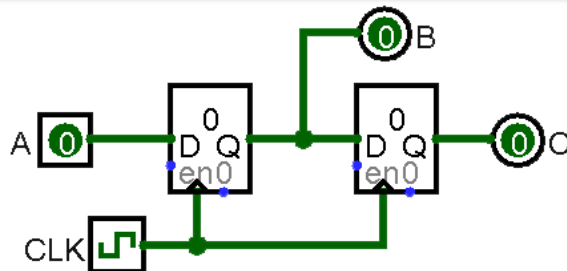
- ❖ `always`: loop to execute over and over again
 - Block gets triggered by a *sensitivity list*
 - Any object that is assigned a value in an `always` statement must be declared as a variable (`logic` or `reg`).
 - Example:
 - `always @ (posedge clk)`
- ❖ `always_ff`: special SystemVerilog for SL
 - *Only for use with sequential logic – signal intent that you want flip-flops*
 - Example:
 - `always_ff @ (posedge clk)`

Blocking vs. Nonblocking

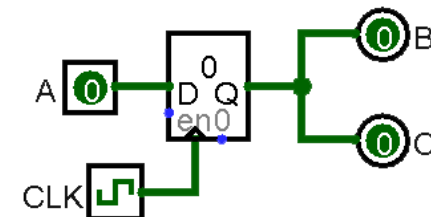
- ❖ **Blocking** statement (=): statement effects evaluated sequentially
 - Resembles programming languages
- ❖ **Nonblocking** statement (<=): statement effects evaluated “in parallel”
 - Resembles hardware

- ❖ Example:

```
always_ff @ (posedge clk)
begin
    b = a;
    c = b;
end
```



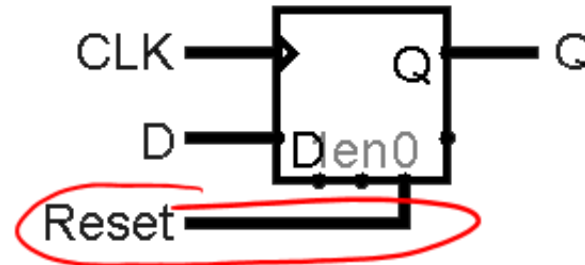
```
always_ff @ (posedge clk)
begin
    b <= a;
    c <= b;
end
```



SystemVerilog Coding Guidelines

- 1) When modeling sequential logic, use *nonblocking* assignments
- 2) When modeling combinational logic with an `always_comb` block, use *blocking* assignments
- 3) When modeling both sequential and combinational logic within the same `always_ff` block, use *nonblocking* assignments
- 4) Do not mix *blocking* and *nonblocking* assignments in the same `always_*` block
- 5) Do not make assignments to the same variable from more than one `always_*` block

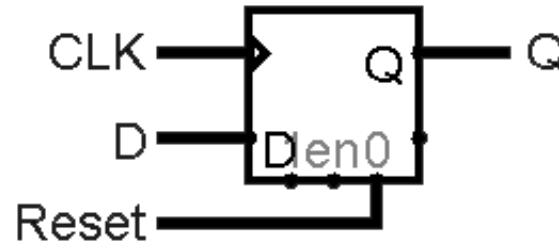
Verilog: Reset Functionality



❖ Option 1: synchronous reset

```
module D_FF1 (q, d, reset, clk);  
  output logic q; // q is state-holding  
  input  logic d, reset, clk;  
  
  always_ff @(posedge clk)  
    if (reset) ← reset can only occur on clock trigger  
      q <= 0; // on reset, set to 0  
    else  
      q <= d; // otherwise pass d to q  
  
endmodule
```


Verilog: Reset Functionality



❖ Option 2: asynchronous reset

```
module D_FF1 (q, d, reset, clk);  
  output logic q; // q is state-holding  
  input  logic d, reset, clk;  
  
  always_ff @(posedge clk or posedge reset)  
    if (reset)  
      q <= 0; // on reset, set to 0  
    else  
      q <= d; // otherwise pass d to q  
  
endmodule
```

any reset posedge, no matter where in the clock cycle

Verilog: Simulated Clock

- ❖ For simulation, you need to generate a clock signal:
 - For entirety of simulation/program, so use an `always` block

Explicit Edges:

```

initial
    clk = 0;
always begin
    #50  clk <= 1;
    #50  clk <= 0;
end
  
```

Toggle:

```

initial
    clk = 0;
always
    #50  clk <= ~clk;
  
```

- ❖ Define clock period:
 - Define **parameter**

like #define macro substitution in C

```

parameter period = 100;
initial
    clk = 0;
always
    #(period/2)  clk <= ~clk;
  
```

Verilog Testbench with Clock

```

module D_FF_testbench;
  logic CLK, reset, d;
  logic q;

  parameter PERIOD = 100;

  D_FF dut (.q, .d, .reset, .CLK); // Instantiate the D_FF

  initial CLK <= 0; // Set up clock
  always #(PERIOD/2) CLK <= ~CLK;

  initial begin
    d <= 0; reset <= 1;
    @(posedge CLK); reset <= 0;
    @(posedge CLK); d <= 1;
    @(posedge CLK); d <= 0;
    @(posedge CLK); #(PERIOD/4) d <= 1;
    @(posedge CLK);
    $stop(); // end the simulation
  end
endmodule

```

← simulated inputs
← DUT output

toggle form of clock

these occur just after clock triggers

no statement here by choice

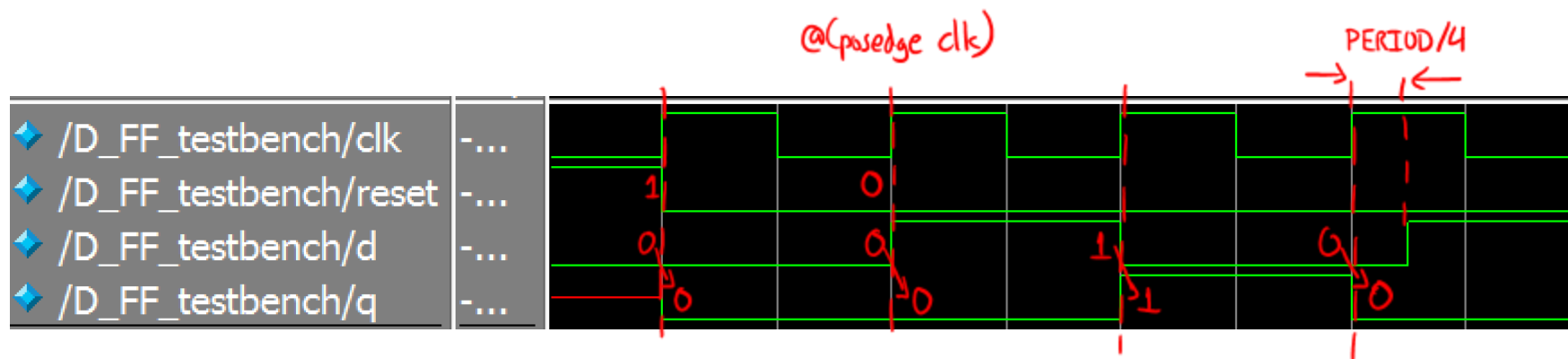
t=0 →
t=100 →
t=200 →
t=300 →
t=400 →
t=500 →

// Set up signals

Timing Controls

- ❖ Delay: `#<time>`
 - Delays by a specific amount of simulation time
 - Can do calculations in `<time>`
 - Examples: `#(PERIOD/4)`, `#50`
- ❖ Edge-sensitive: `@(<pos/negedge> signal)`
 - Delays next statement until specified transition on signal
 - Example: `@(posedge CLK)`
- ❖ Level-sensitive Event: `wait(<expression>)`
 - Delays next statement until `<expression>` evaluates to TRUE
 - Example: `wait(enable == 1)`

ModelSim Waveforms



```

initial begin
    d <= 0; reset <= 1;
    @(posedge CLK);      reset <= 0;
    @(posedge CLK); d <= 1;
    @(posedge CLK); d <= 0;
    @(posedge CLK); #(PERIOD/4) d <= 1;
    @(posedge CLK);
    $stop();
end
    
```

happens just after posedge

wait after posedge

no statement occurs exactly at posedge

Summary (1/2)

- ❖ State elements controlled by clock
 - Store information
 - Control the flow of information between other state elements and combinational logic
- ❖ Registers implemented from flip-flops
 - Triggered by CLK, pass input to output, can reset
- ❖ Critical path constrains clock rate
 - Timing constants: setup time, hold time, clk-to-q delay, propagation delays

Summary (2/2)

- ❖ Generating a clock
 - Manually create using a `always` block
 - Need to decide on period
- ❖ Blocking vs. Non-blocking
 - Blocking: Statements executed “in series”
 - Non-blocking: Statements executed “in parallel”
 - Always use non-blocking for clocked elements
- ❖ Synchronous vs. Asynchronous
 - Whether signals are controlled by clock or not