

# Digital Design & Computer Arch.

## Lecture 8: Timing and Verification

Prof. Onur Mutlu

ETH Zürich

Spring 2022

18 March 2022

# Readings (This Week)

---

- Hardware Description Languages and Verilog
  - H&H Chapter 4 in full
- Timing and Verification
  - H&H Chapters 2.9 and 3.5 + (start Chapter 5)
- By tomorrow, make sure you are done with
  - **P&P Chapters 1-3 + H&H Chapters 1-4**

# Readings (Next Week)

---

## ■ Von Neumann Model, LC-3, and MIPS

- P&P, Chapter 4, 5
- H&H, Chapter 6
- P&P, Appendices A and C (ISA and microarchitecture of LC-3)
- H&H, Appendix B (MIPS instructions)

## ■ Programming

- P&P, Chapter 6

## ■ **Recommended:** Digital Building Blocks

- H&H, Chapter 5

# Assignment: Lecture Video (April 1)

---

- Why study computer architecture? Why is it important?
- Future Computing Platforms: Challenges & Opportunities
- **Required Assignment**
  - **Watch one of** Prof. Mutlu's lectures and analyze either (or both)
    - <https://www.youtube.com/watch?v=kgiZISOcGFM> (May 2017)
    - <https://www.youtube.com/watch?v=mskTeNnf-i0> (Feb 2021)
- **Optional Assignment – for 1% extra credit**
  - **Write a 1-page summary** of one of the lectures and email us
    - What are your key takeaways?
    - What did you learn?
    - What did you like or dislike?
    - Submit your summary to [Moodle](#) by April 1

# Extra Assignment: Moore's Law (I)

---

- **Paper review**
- G.E. Moore. "Cramming more components onto integrated circuits," Electronics magazine, 1965
  
- **Optional Assignment – for 1% extra credit**
  - **Write a 1-page review**
  - Upload PDF file to Moodle – Deadline: April 7
  
- I strongly recommend that you **follow my guidelines for (paper) review** (see next slide)

# Extra Assignment 2: Moore's Law (II)

---

## ■ Guidelines on how to review papers critically

- ❑ Guideline slides: [pdf](#) [ppt](#)
- ❑ Video: <https://www.youtube.com/watch?v=tOL6FANAj8c>
- ❑ Example reviews on "Main Memory Scaling: Challenges and Solution Directions" ([link to the paper](#))
  - [Review 1](#)
  - [Review 2](#)
- ❑ Example review on "Staged memory scheduling: Achieving high performance and scalability in heterogeneous systems" ([link to the paper](#))
  - [Review 1](#)

# Agenda

---

- Wrap up FSMs in Verilog
- Timing in **combinational circuits**
- Timing in **sequential circuits**
- **Circuit verification**

# Wrap Up: FSMs in Verilog



# Recall: Why Use (Non)-Blocking Statements

---

- Non-blocking statements allow operating on “old” values
    - Enable easy sequential logic descriptions
  - Blocking statements allow a sequence of operations
    - Allow operating on immediately updated values
    - More like a “software” programming language
  - If the sensitivity list is correct, blocks with non-blocking statements will always evaluate to the same result
    - This may require some additional iterations
-

# Rules for Signal Assignment

---

- Use `always @(posedge clk)` and `non-blocking` assignments (`<=`) to model `synchronous sequential logic`

```
always @ (posedge clk)
    q <= d; // non-blocking
```

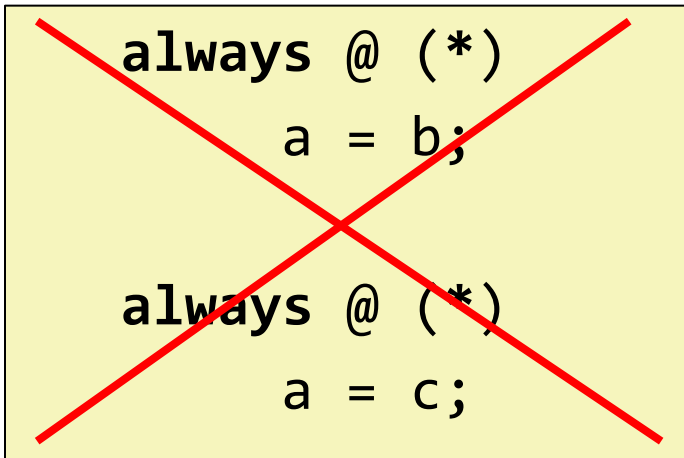
- Use continuous assignments (`assign`) to model simple combinational logic

```
assign y = a & b;
```

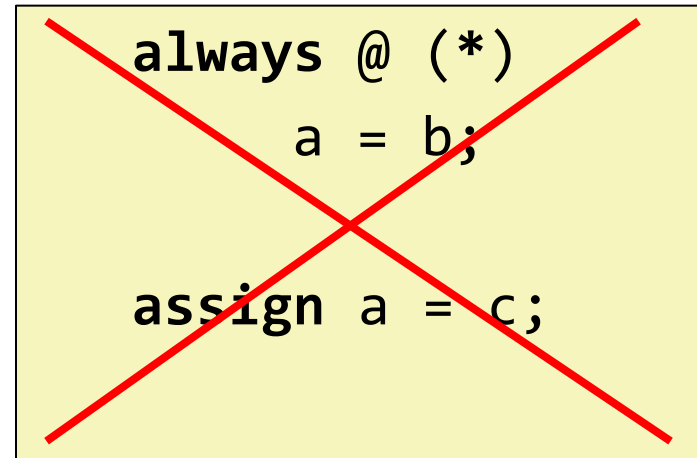
# Rules for Signal Assignment (Cont.)

---

- Use **always @ (\*)** and **blocking** assignments (=) to model more **complicated combinational logic**.
- You **cannot** make assignments to the **same** signal in more than one always block or in a *continuous assignment*



```
always @ (*)  
    a = b;  
  
always @ (*)  
    a = c;
```

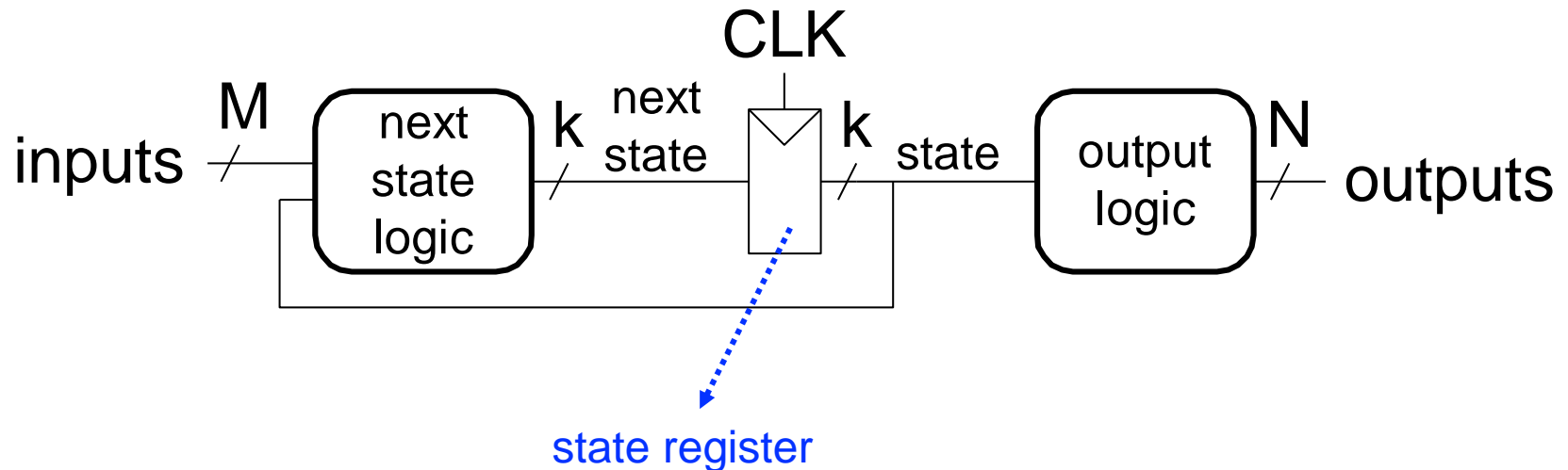


```
always @ (*)  
    a = b;  
  
assign a = c;
```

# Recall: Finite State Machines (FSMs)

---

- Each FSM consists of three separate parts:
  - next state logic
  - state register
  - output logic

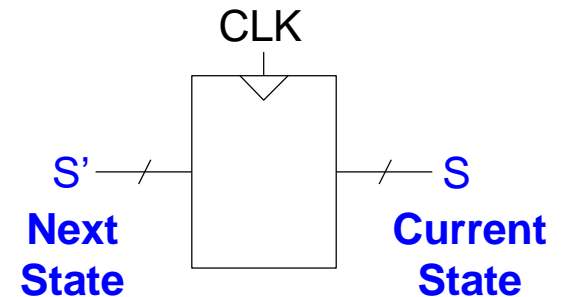


# Recall: Finite State Machines (FSMs) Comprise

---

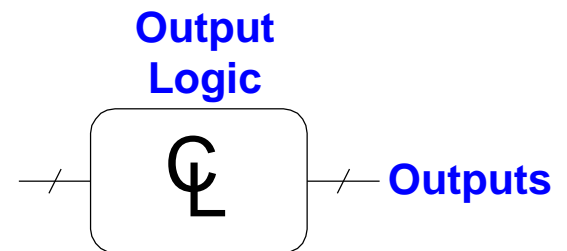
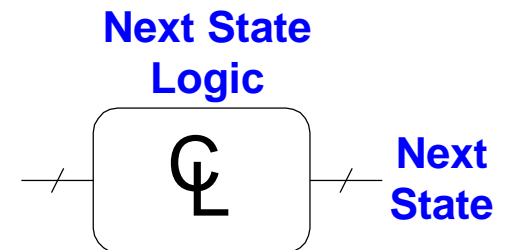
## ■ Sequential circuits

- State register(s)
  - Store the current state and
  - Load the next state at the clock edge



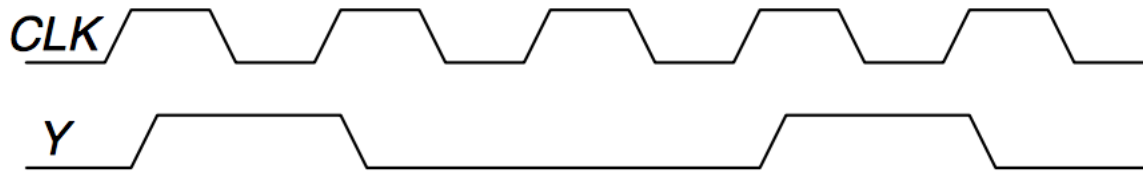
## ■ Combinational Circuits

- Next state logic
  - Determines what the next state will be
- Output logic
  - Generates the outputs

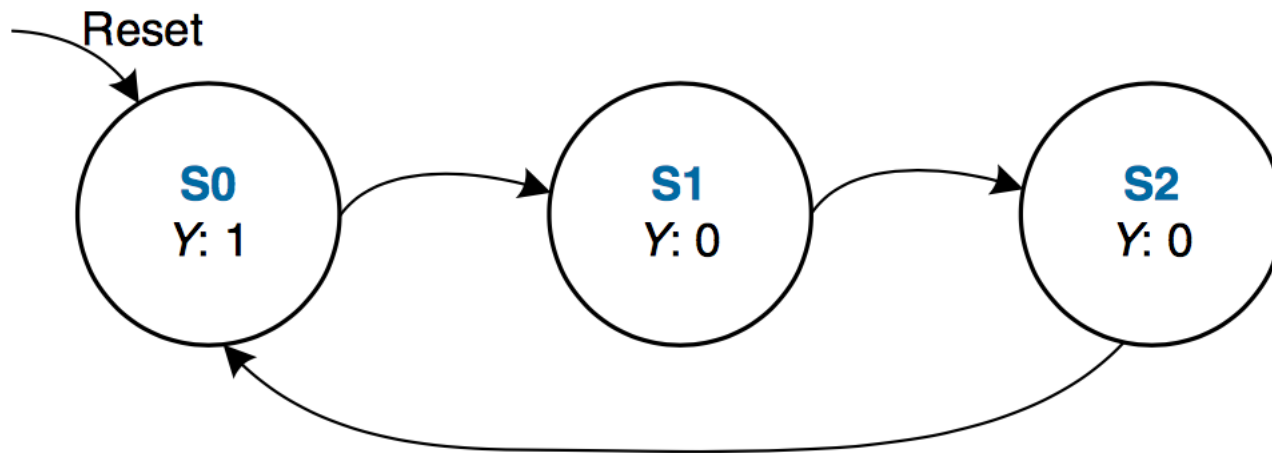


# FSM Example 1: Divide the Clock Frequency by 3

---



The output  $Y$  is HIGH for **one clock cycle out of every 3**. In other words, the output **divides the frequency of the clock by 3**.



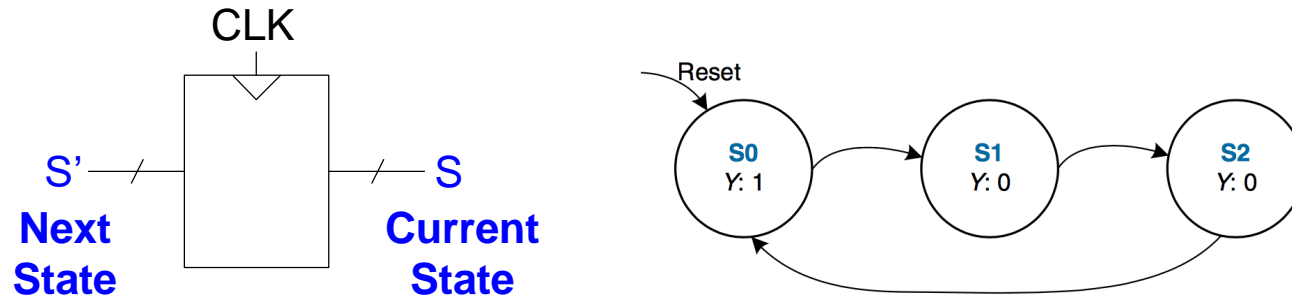
# Implementing FSM Example 1: Definitions

---

```
module divideby3FSM (input clk,  
                    input reset,  
                    output q);  
  
    reg [1:0] state, nextstate;  
  
    parameter S0 = 2'b00;  
    parameter S1 = 2'b01;  
    parameter S2 = 2'b10;
```

- We define **state** and **nextstate** as 2-bit reg
- The parameter descriptions are **optional**, it makes reading easier

# Implementing FSM Example 1: State Register

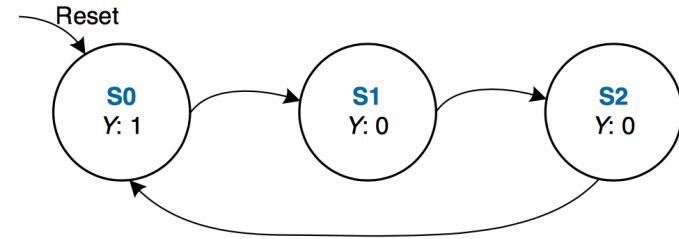
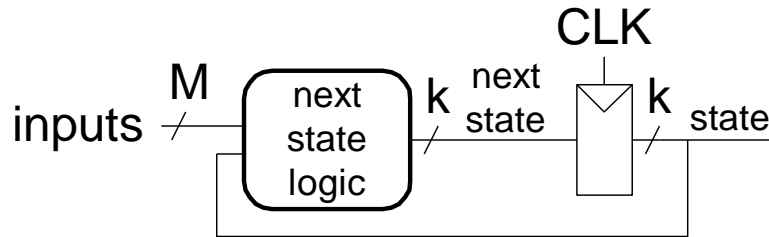


```
// state register
always @ (posedge clk, posedge reset)
  if (reset) state <= S0;
  else      state <= nextstate;
```

- This part defines the **state register** (memorizing process)
- Sensitive to only **clk**, **reset**
- In this example, **reset** is active when it is '1' (active-high)

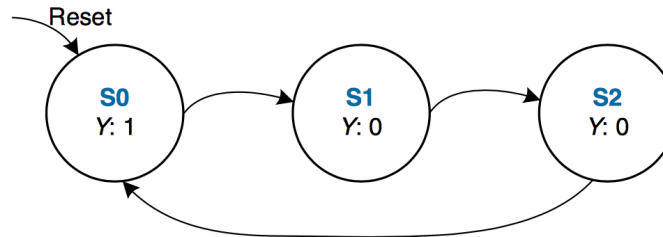
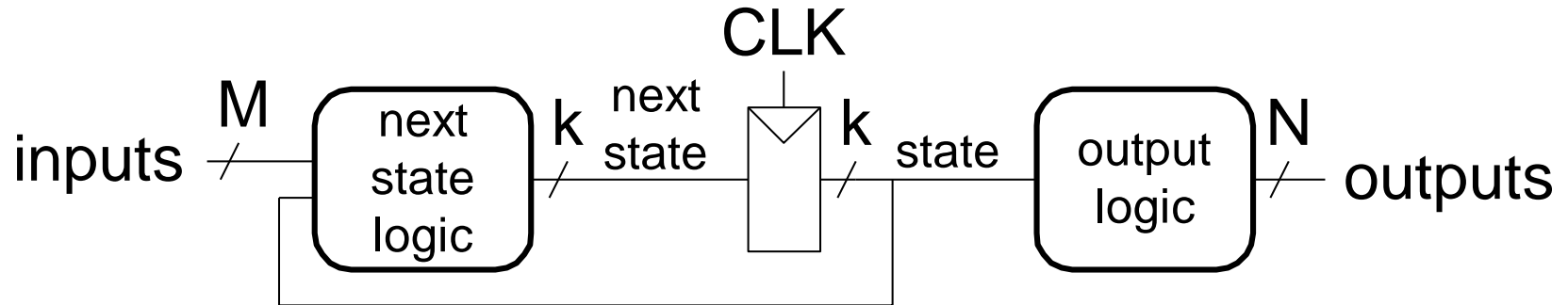


# Implementing FSM Example 1: Next State Logic



```
// next state logic
always @ (*)
  case (state)
    S0:      nextstate = S1;
    S1:      nextstate = S2;
    S2:      nextstate = S0;
    default: nextstate = S0;
  endcase
```

# Implementing FSM Example 1: Output Logic



```
// output logic
assign q = (state == S0);
```

- In this example, output depends only on state
  - **Moore type FSM**

# Implementation of FSM Example 1

---

```
module divideby3FSM (input clk, input reset, output q);
    reg [1:0] state, nextstate;

    parameter S0 = 2'b00; parameter S1 = 2'b01; parameter S2 = 2'b10;

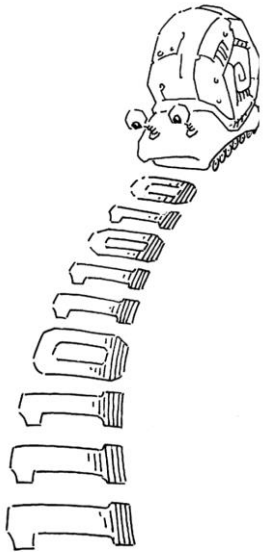
    always @ (posedge clk, posedge reset) // state register
        if (reset) state <= S0;
        else      state <= nextstate;

    always @ (*) // next state logic
        case (state)
            S0:      nextstate = S1;
            S1:      nextstate = S2;
            S2:      nextstate = S0;
            default: nextstate = S0;
        endcase

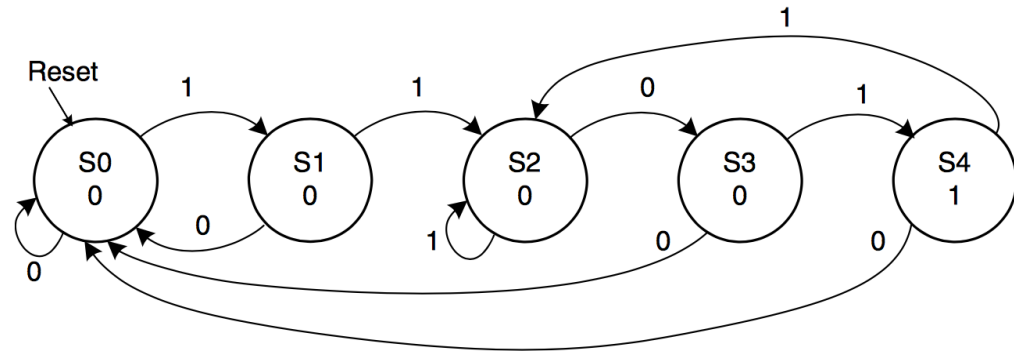
    assign q = (state == S0); // output logic
endmodule
```

# FSM Example 2: Smiling Snail

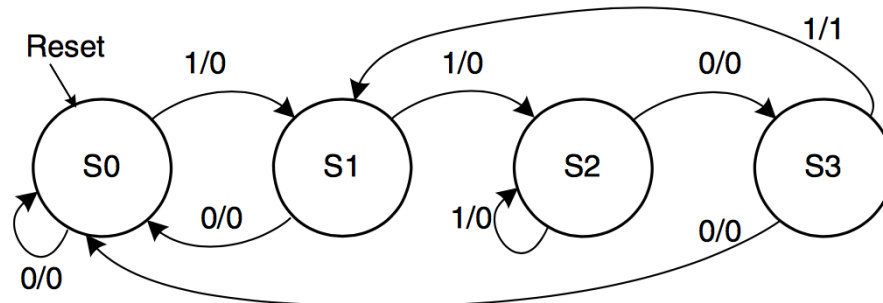
- Alyssa P. Hacker has a snail that crawls down a paper tape with 1's and 0's on it
- The snail smiles whenever the last four digits it has crawled over are **1101**
- Design Moore and Mealy FSMs of the snail's brain



**Moore**



**Mealy**



# Implementing FSM Example 2: Definitions

```
module SmilingSnail (input clk,  
                    input reset,  
                    input number,  
                    output smile);
```

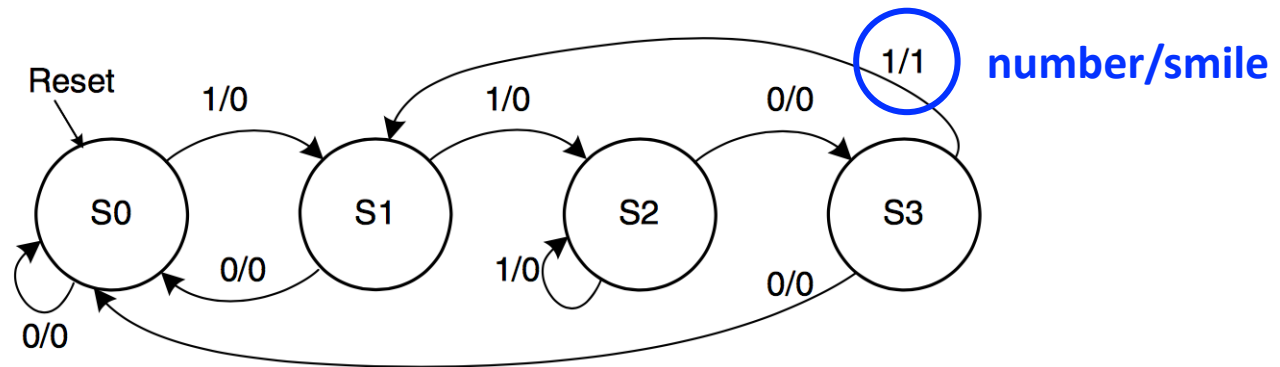
```
    reg [1:0] state, nextstate;
```

```
    parameter S0 = 2'b00;
```

```
    parameter S1 = 2'b01;
```

```
    parameter S2 = 2'b10;
```

```
    parameter S3 = 2'b11;
```



# Implementing FSM Example 2: State Register

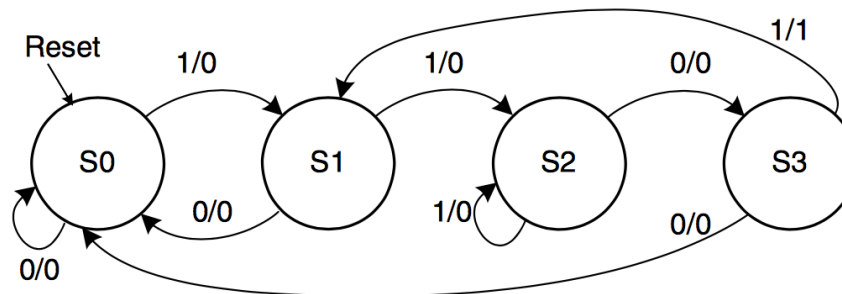
---

```
// state register
always @ (posedge clk, posedge reset)
    if (reset) state <= S0;
    else      state <= nextstate;
```

- This part defines the **state register** (memorizing process)
- Sensitive to only **clk**, **reset**
- In this example **reset** is active when '1' (active-high)

# Implementing FSM Example 2: Next State Logic

```
// next state logic
always @ (*)
  case (state)
    S0: if (number) nextstate = S1;
        else nextstate = S0;
    S1: if (number) nextstate = S2;
        else nextstate = S0;
    S2: if (number) nextstate = S2;
        else nextstate = S3;
    S3: if (number) nextstate = S1;
        else nextstate = S0;
    default: nextstate = S0;
  endcase
```



# Implementing FSM Example 2: Output Logic

---

```
// output logic  
assign smile = (number & state == S3);
```

- In this example, output depends on state and input
  - **Mealy type FSM**
- We used a simple combinational assignment



# Implementation of FSM Example 2

```
module SmilingSnail (input clk,
                    input reset,
                    input number,
                    output smile);

    reg [1:0] state, nextstate;

    parameter S0 = 2'b00;
    parameter S1 = 2'b01;
    parameter S2 = 2'b10;
    parameter S3 = 2'b11;

    // state register
    always @ (posedge clk, posedge
reset)
        if (reset) state <= S0;
        else      state <= nextstate;
```

```
    always @ (*) // next state logic
        case (state)
            S0: if (number)
                    nextstate = S1;
                else nextstate = S0;
            S1: if (number)
                    nextstate = S2;
                else nextstate = S0;
            S2: if (number)
                    nextstate = S2;
                else nextstate = S3;
            S3: if (number)
                    nextstate = S1;
                else nextstate = S0;
            default: nextstate = S0;
        endcase
    // output logic
    assign smile = (number & state==S3);

endmodule
```

# What Did We Learn?

---

- Basics of describing **sequential circuits** in Verilog
- The **always** statement
  - ❑ Needed for defining memorizing elements (**flip-flops, latches**)
  - ❑ Can also be used to define **combinational circuits**
- **Blocking** vs **Non-blocking** statements
  - ❑ **=** assigns the value **immediately**
  - ❑ **<=** assigns the value **at the end of the block**
- **Describing FSMs in Verilog**
  - ❑ Next state logic
  - ❑ State assignment
  - ❑ Output logic

**Now:**

Timing and Verification

# What Will We Learn Today?

---

- Timing in **combinational circuits**
  - Propagation delay and contamination delay
  - Glitches
- Timing in **sequential circuits**
  - Setup time and hold time
  - Determining how fast a circuit can operate
- **Circuit Verification**
  - How to make sure a circuit works correctly
  - Functional verification
  - Timing verification

# Tradeoffs in Circuit Design

# Circuit Design is a Tradeoff Between:

---

## ■ Area

- Circuit **area** is proportional to the **cost** of the device

## ■ Speed / Throughput

- We want **faster**, more **capable** circuits

## ■ Power / Energy

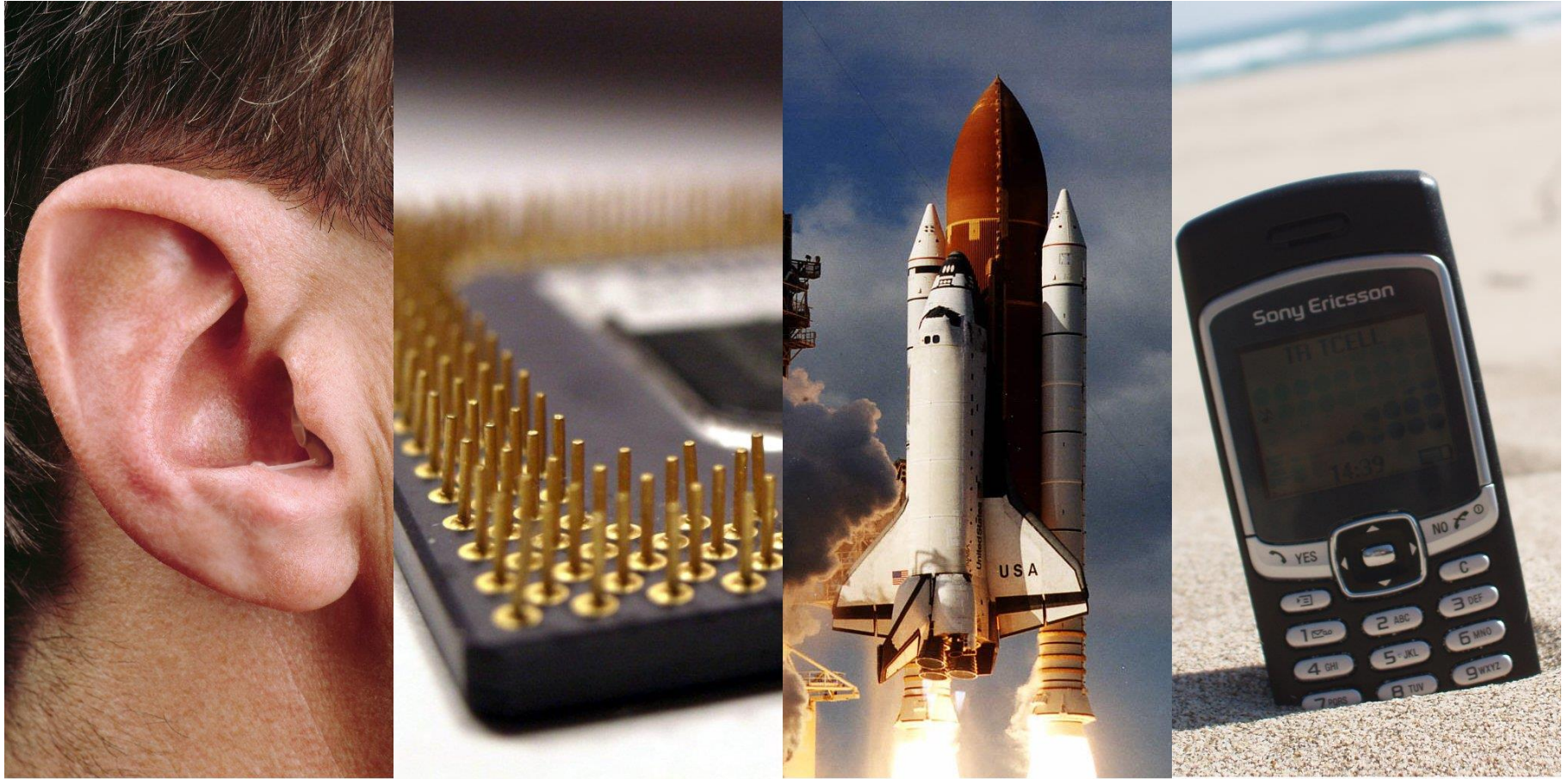
- Mobile devices need to work with a **limited** power supply
- High performance devices **dissipate** more than 100W/cm<sup>2</sup>

## ■ Design Time

- Designers are **expensive** in *time* and *money*
- The **competition** will not wait for you

# Requirements and Goals Depend On Application

---



# Circuit Timing

---

- Until now, we investigated **logical functionality**
- What about **timing**?
  - How **fast** is a circuit?
  - How can we make a circuit **faster**?
  - What happens if we run a circuit **too fast**?
- A design that is logically correct can still **fail** because of real-world **implementation issues**!



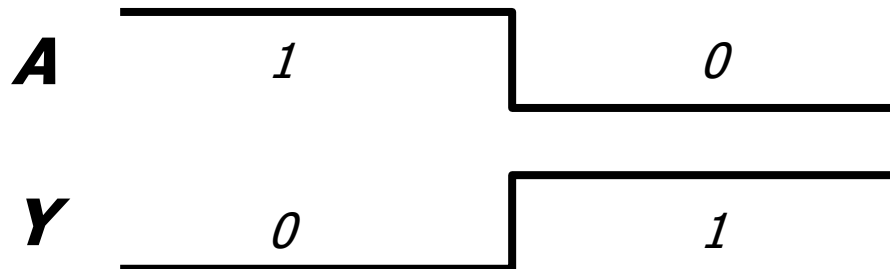
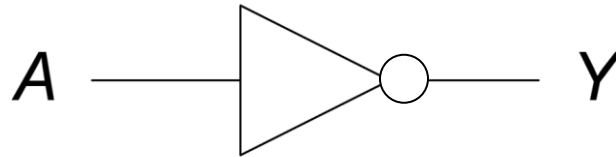
# Part 1:

# Combinational Circuit Timing

# Digital Logic Abstraction

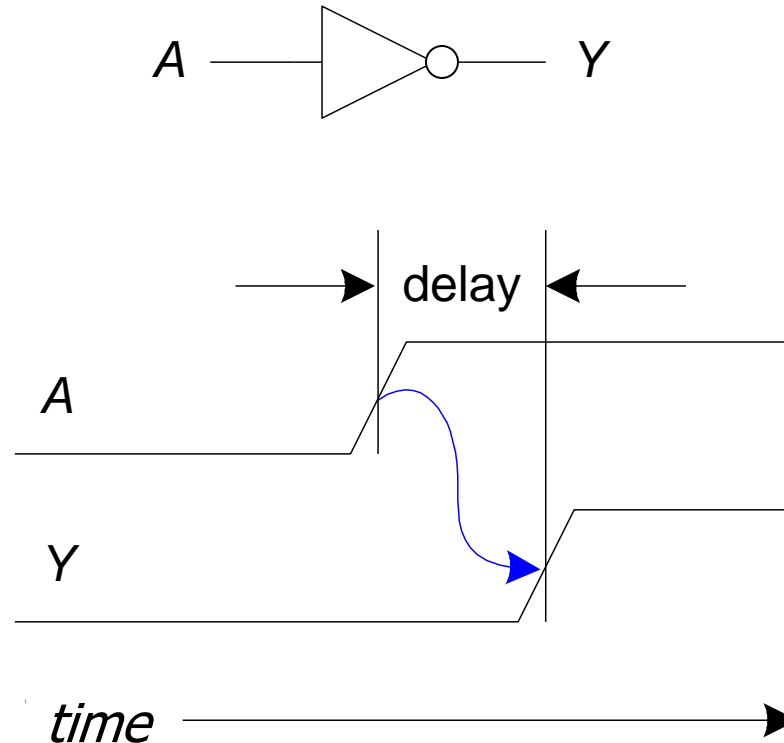
---

- “**Digital logic**” is a convenient **abstraction**
  - Output changes *immediately* with the input

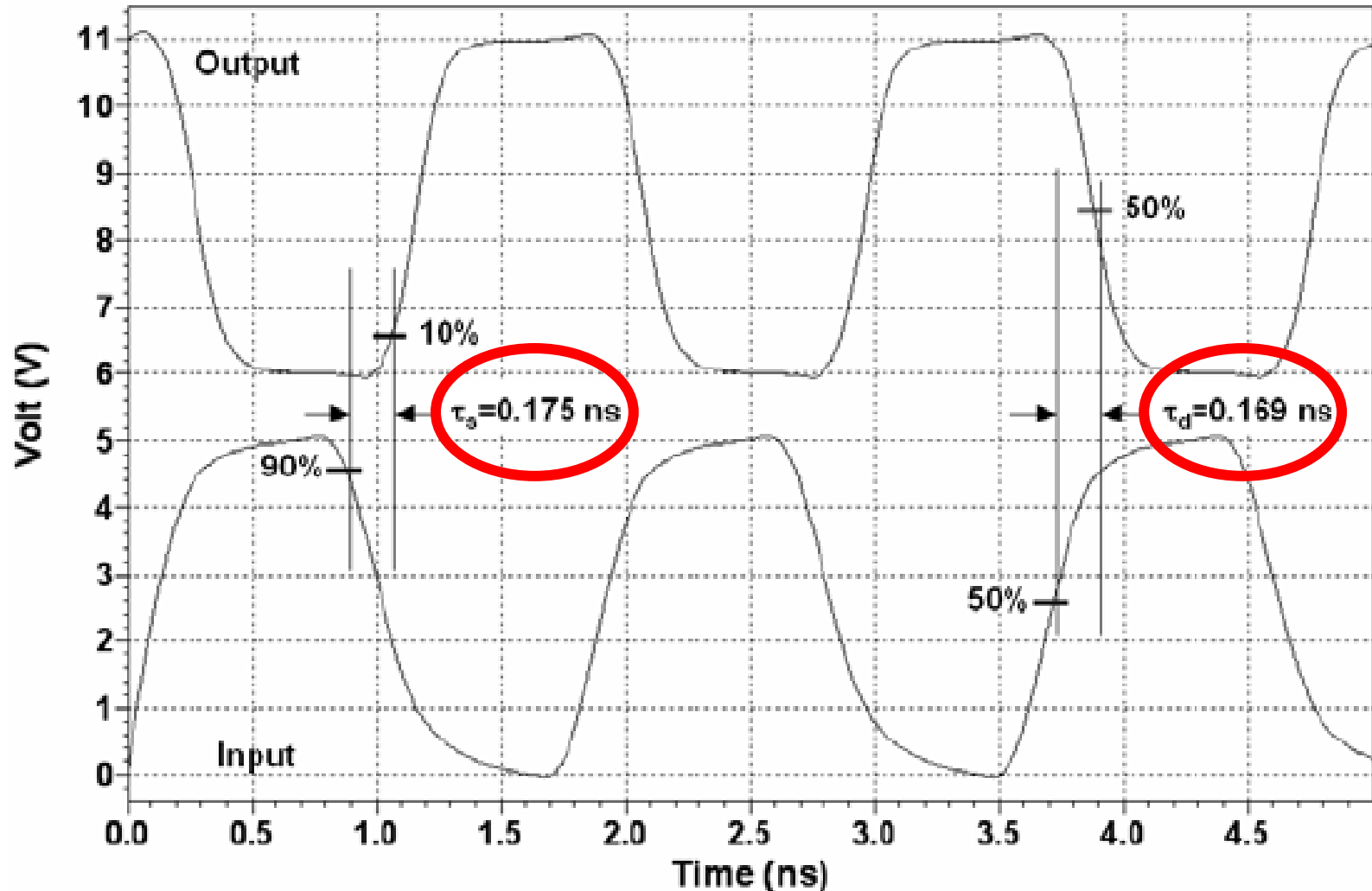


# Combinational Circuit Delay

- In reality, **outputs** are **delayed** from **inputs**
  - Transistors take a finite amount of time to switch



# Real Inverter Delay Example



# Circuit Delay and Its Variation

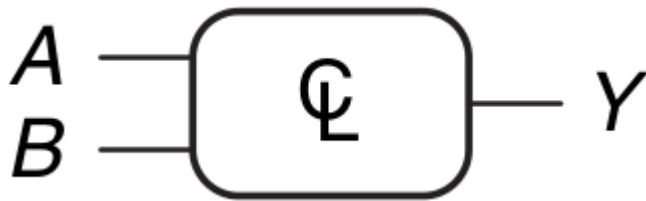
---

- Delay is fundamentally caused by
  - **Capacitance** and **resistance** in a circuit
  - Finite **speed of light** (not so fast on a nanosecond scale!)
- **Anything** affecting these quantities can change delay:
  - **Rising** (i.e., 0  $\rightarrow$  1) vs. **falling** (i.e., 1  $\rightarrow$  0) inputs
  - Different **inputs** have different **delays**
  - Changes in **environment** (e.g., temperature)
  - **Aging** of the circuit
- We have a **range of possible delays** from input to output

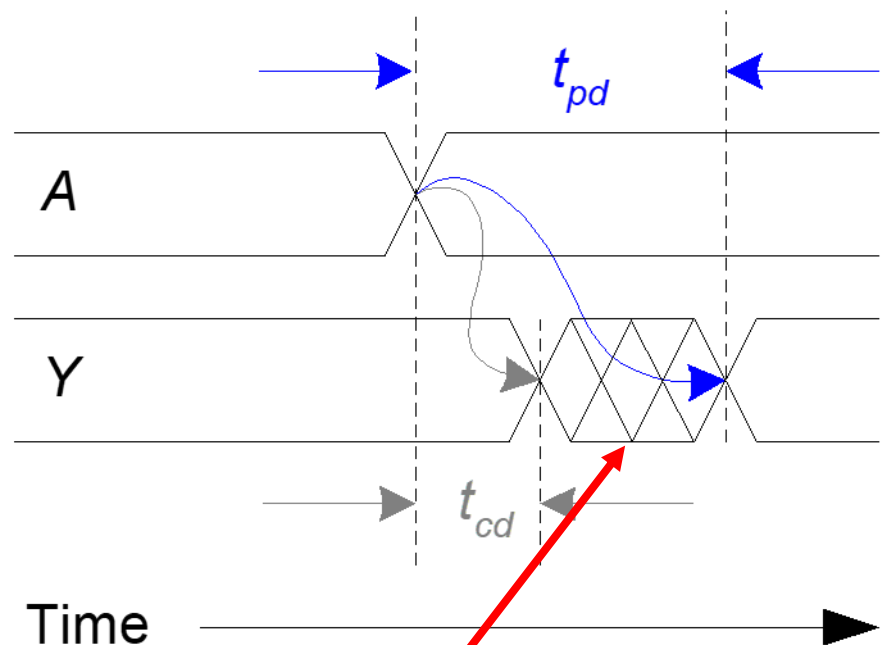
# Delays from Input to Output Y

- **Contamination delay ( $t_{cd}$ ):** delay until Y ***starts changing***
- **Propagation delay ( $t_{pd}$ ):** delay until Y ***finishes changing***

## Example Circuit



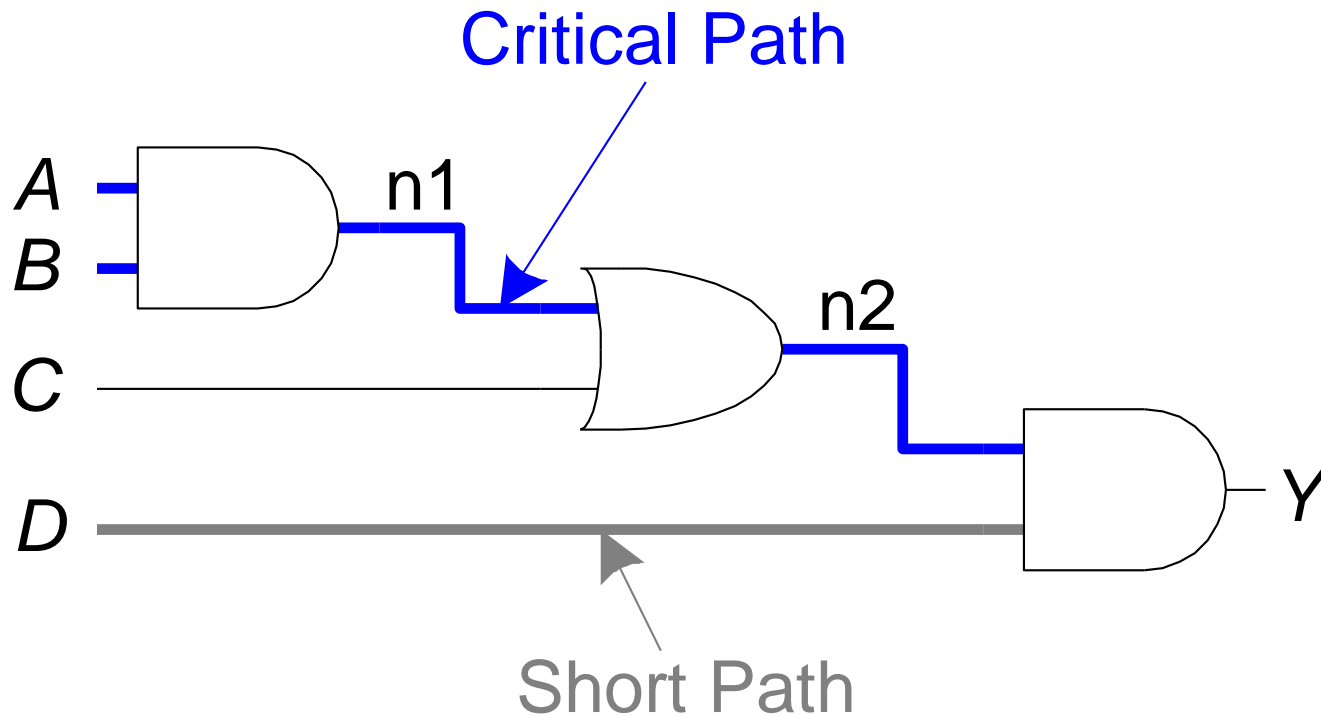
## Effect of Changing Input 'A'



**Cross-hatching  
means value is changing**

# Calculating Longest & Shortest Delay Paths

- We care about **both** the *longest* and *shortest* delay paths in a circuit (we will see why later in the lecture)



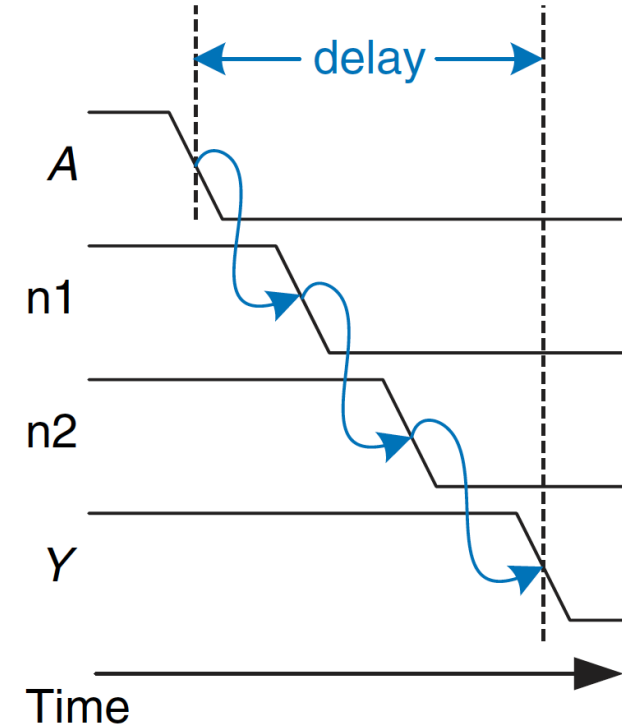
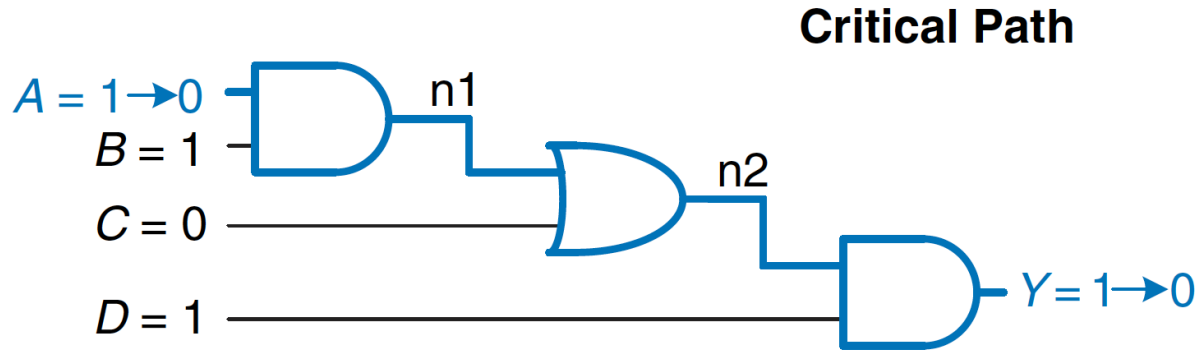
- **Critical (Longest) Path:**

$$t_{pd} = 2 t_{pd\_AND} + t_{pd\_OR}$$

- **Shortest Path:**

$$t_{cd} = t_{cd\_AND}$$

# Calculating Longest Delay Path (Critical Path)



- **Critical (Longest) Path:**
- **Shortest Path:**

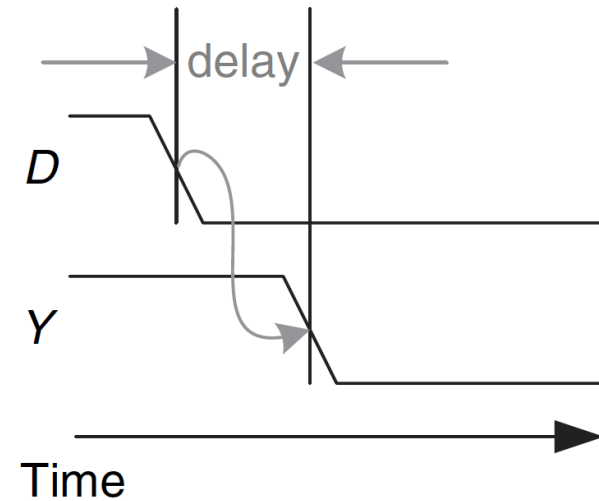
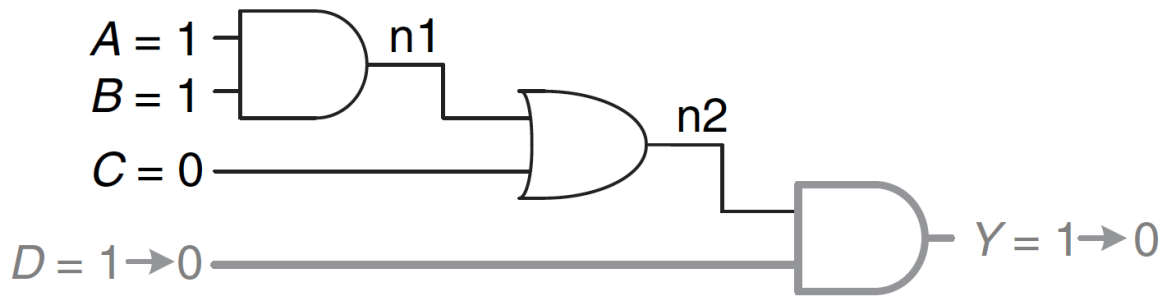
$$t_{pd} = 2 t_{pd\_AND} + t_{pd\_OR}$$

$$t_{cd} = t_{cd\_AND}$$



# Calculating Shortest Delay Path

## Short Path



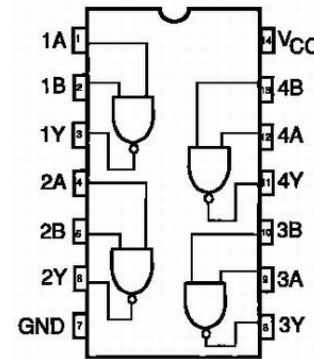
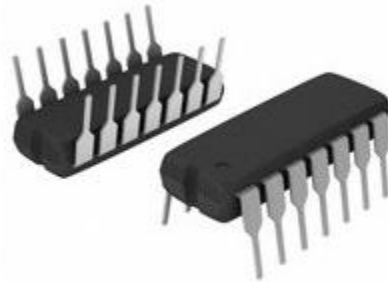
- **Critical (Longest) Path:**

$$t_{pd} = 2 t_{pd\_AND} + t_{pd\_OR}$$

- **Shortest Path:**

$$t_{cd} = t_{cd\_AND}$$

# Example $t_{pd}$ for a Real NAND-2 Gate



Symbol	Parameter	Conditions	25 °C			−40 °C to +125 °C		Unit
			Min	Typ	Max	Max (85 °C)	Max (125 °C)	
74HC00								
t <sub>pd</sub>	propagation delay	nA, nB to nY; see <a href="#">Figure 6</a> <a href="#">[1]</a>						
		V <sub>CC</sub> = 2.0 V	-	25	-	115	135	ns
		V <sub>CC</sub> = 4.5 V	-	9	-	23	27	ns
		V <sub>CC</sub> = 5.0 V; C <sub>L</sub> = 15 pF	-	7	-	-	-	ns
		V <sub>CC</sub> = 6.0 V	-	7	-	20	23	ns

- Heavy **dependence** on **voltage** and **temperature**!

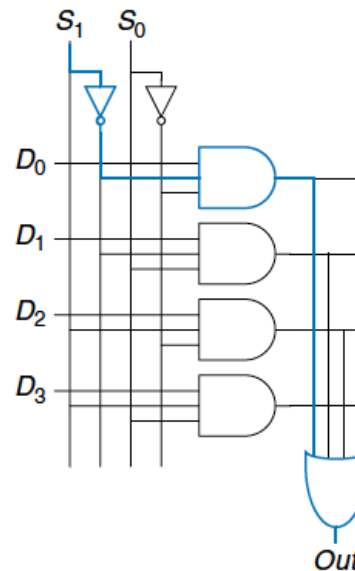
# Example Worst-Case $t_{pd}$

- Two different **implementations** of a **4:1 multiplexer**

## Gate Delays

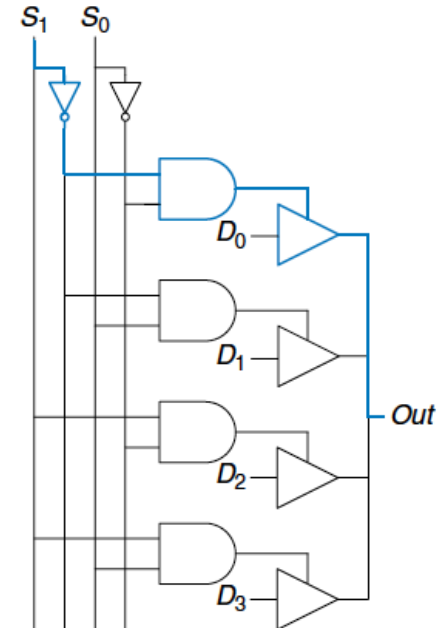
Gate	$t_{pd}$ (ps)
NOT	30
2-input AND	60
3-input AND	80
4-input OR	90
tristate (A to Y)	50
tristate (enable to Y)	35

## Implementation 1



$$\begin{aligned} t_{pd\_sy} &= t_{pd\_INV} + t_{pd\_AND3} + t_{pd\_OR4} \\ &= 30 \text{ ps} + 80 \text{ ps} + 90 \text{ ps} \\ &= 200 \text{ ps} \\ \text{(a)} \quad t_{pd\_dy} &= t_{pd\_AND3} + t_{pd\_OR4} \\ &= 170 \text{ ps} \end{aligned}$$

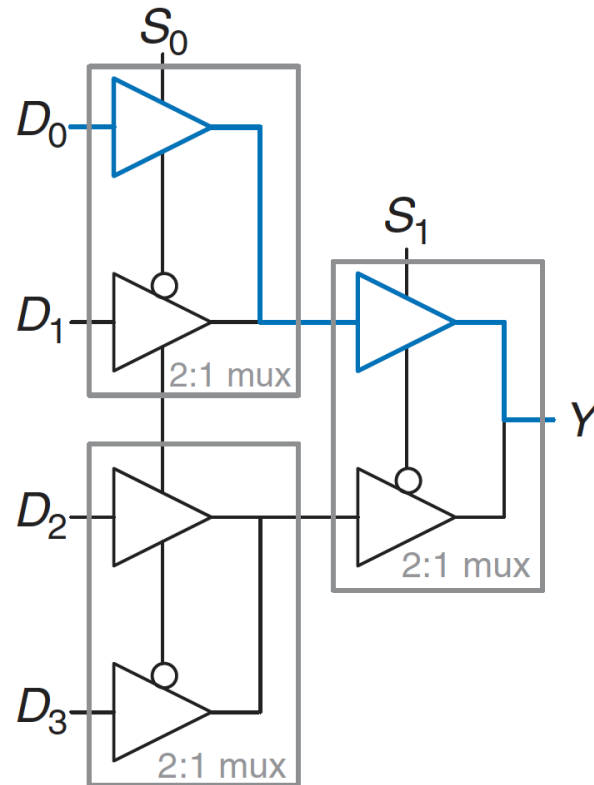
## Implementation 2



$$\begin{aligned} t_{pd\_sy} &= t_{pd\_INV} + t_{pd\_AND2} + t_{pd\_TRI\_SY} \\ &= 30 \text{ ps} + 60 \text{ ps} + 35 \text{ ps} \\ &= 125 \text{ ps} \\ \text{(b)} \quad t_{pd\_dy} &= t_{pd\_TRI\_AY} \\ &= 50 \text{ ps} \end{aligned}$$

- Different designs lead to very **different delays**

# Aside: A Third 4:1 MUX Implementation



$$t_{pd_{s0y}} = t_{pd\_TRLSY} + t_{pd\_TRI\_AY} = 85 \text{ ns}$$

$$t_{pd_{dy}} = 2 t_{pd\_TRI\_AY} = 100 \text{ ns}$$

**Figure 2.74** 4:1 multiplexer propagation delays: hierarchical using 2:1 multiplexers

# Disclaimer: Calculating Long/Short Paths

---

- It's **not** always this easy to determine the long/short paths!
  - Not all **input transitions** affect the **output**
  - Can have **multiple different paths** from an input to output
- In reality, circuits are **not** all built equally
  - Different instances of the **same gate** have **different delays**
  - **Wires** have **nonzero delay** (increasing with length)
  - Temperature/voltage affect circuit speeds
    - Not all circuit elements are affected the same way
    - Can even **change the critical path**!
- Designers assume “**worst-case**” **conditions** and run many **statistical simulations** to balance yield/performance

# Combinational Timing Summary

---

- Circuit outputs change some time **after** the inputs change
  - Caused by finite speed of light (not so fast on a ns scale!)
  - Delay is dependent on inputs, environmental state, etc.
- The range of possible delays is characterized by:
  - **Contamination delay ( $t_{cd}$ )**: *minimum possible* delay
  - **Propagation delay ( $t_{pd}$ )**: *maximum possible* delay
- Delays **change** with:
  - Circuit design (e.g., topology, materials)
  - Operating conditions

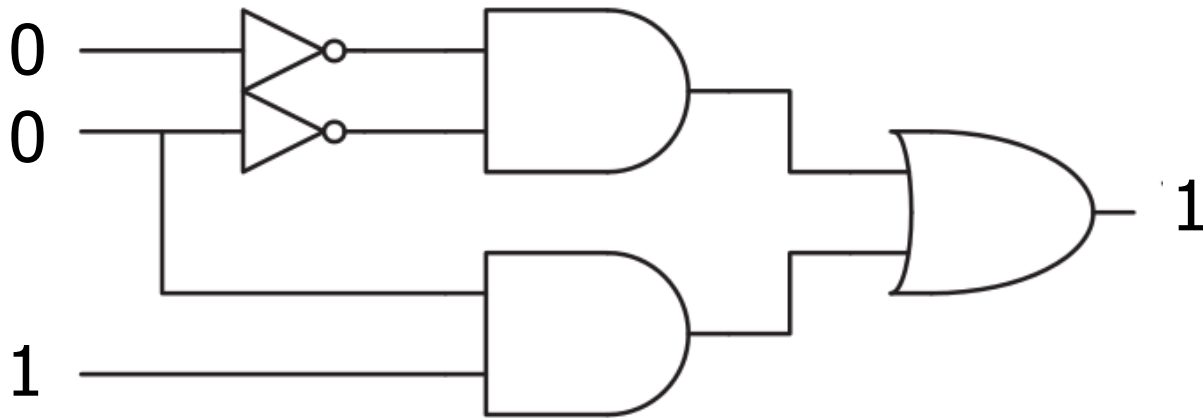
# Output Glitches

# Glitches

---

- **Glitch:** **one** input transition causes **multiple** output transitions

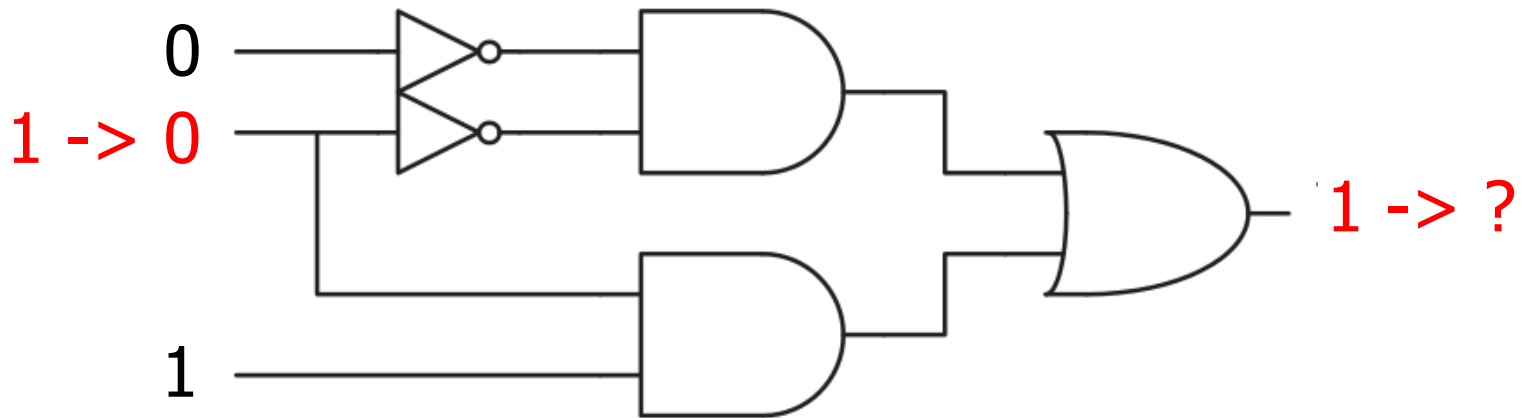
## Circuit initial state





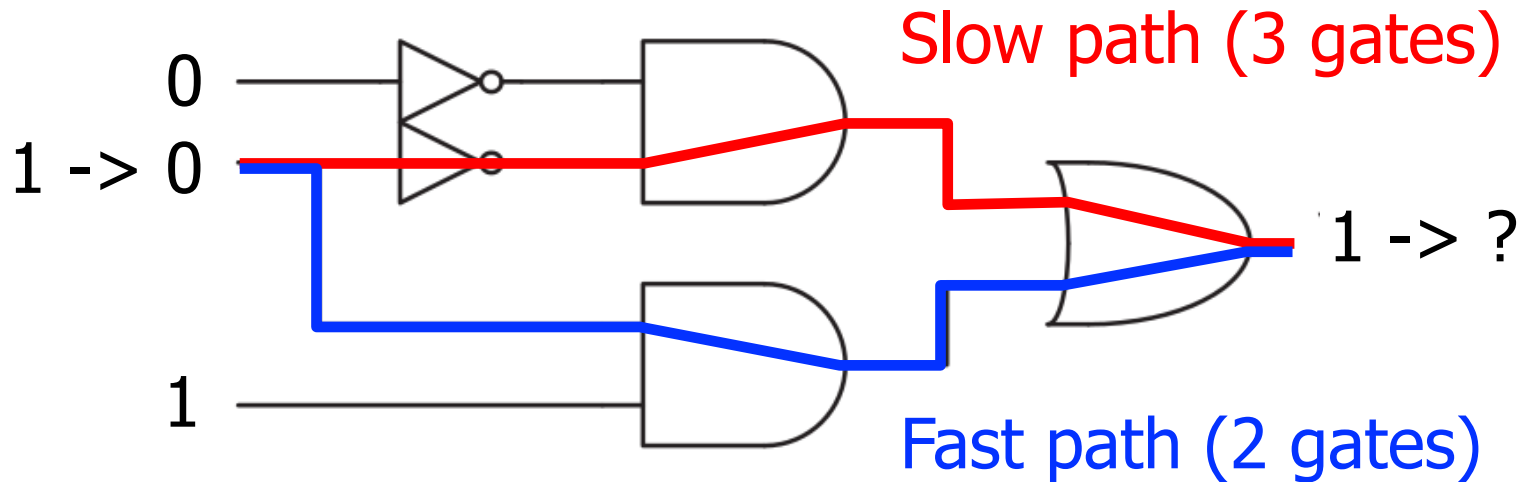
# Glitches

- **Glitch:** **one** input transition causes **multiple** output transitions



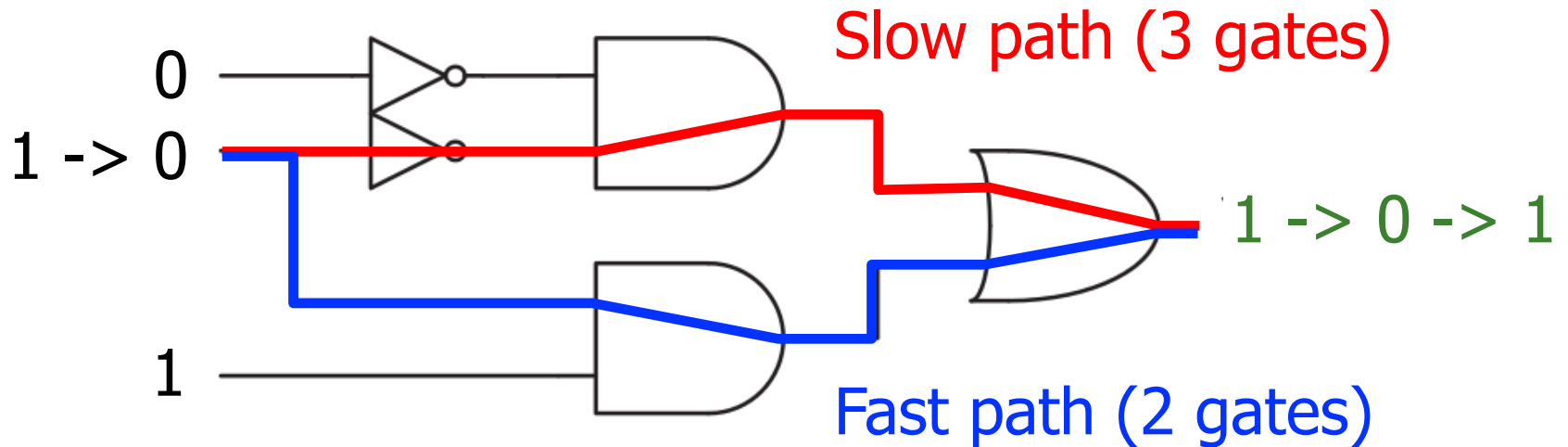
# Glitches

- **Glitch:** **one** input transition causes **multiple** output transitions



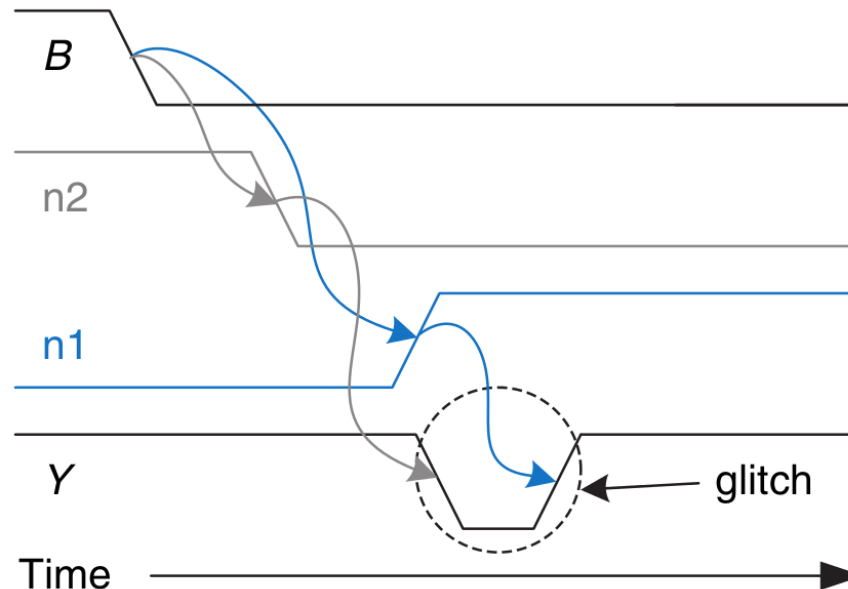
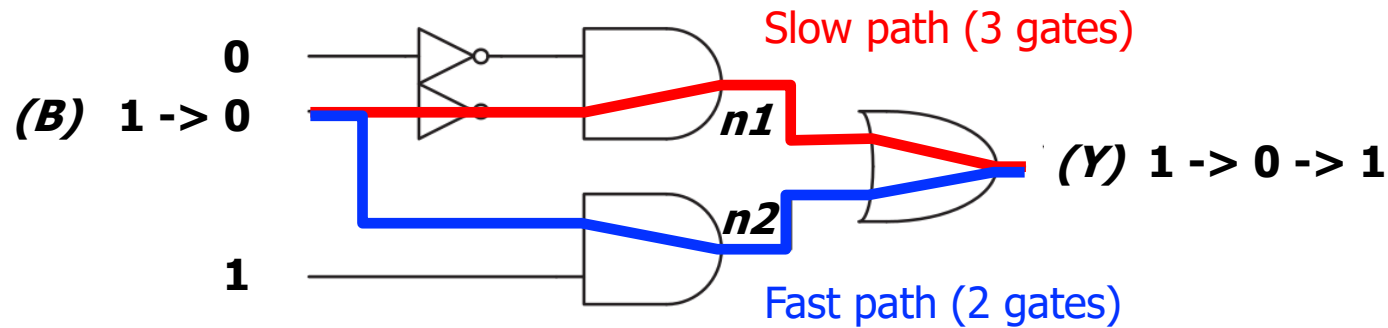
# Glitches

- **Glitch:** **one** input transition causes **multiple** output transitions



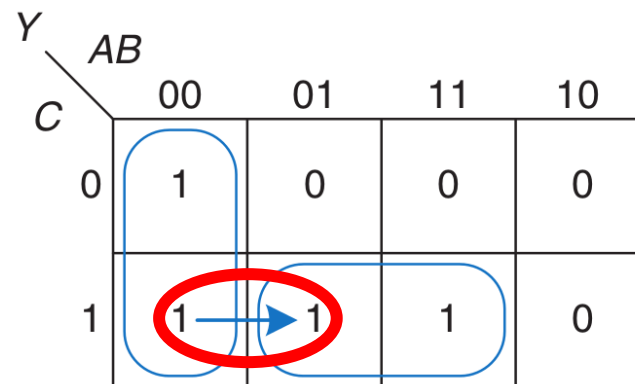
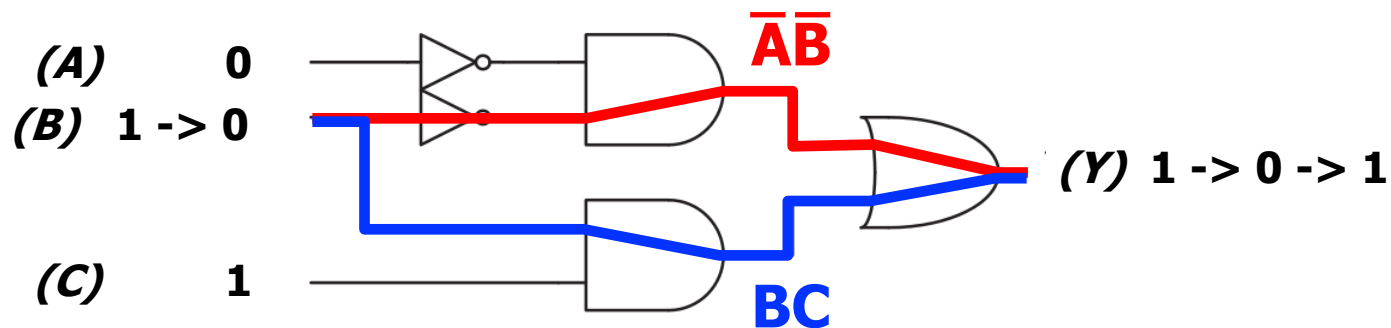
# Glitches

- **Glitch:** one input transition causes **multiple** output transitions



# Optional: Avoiding Glitches Using K-Maps

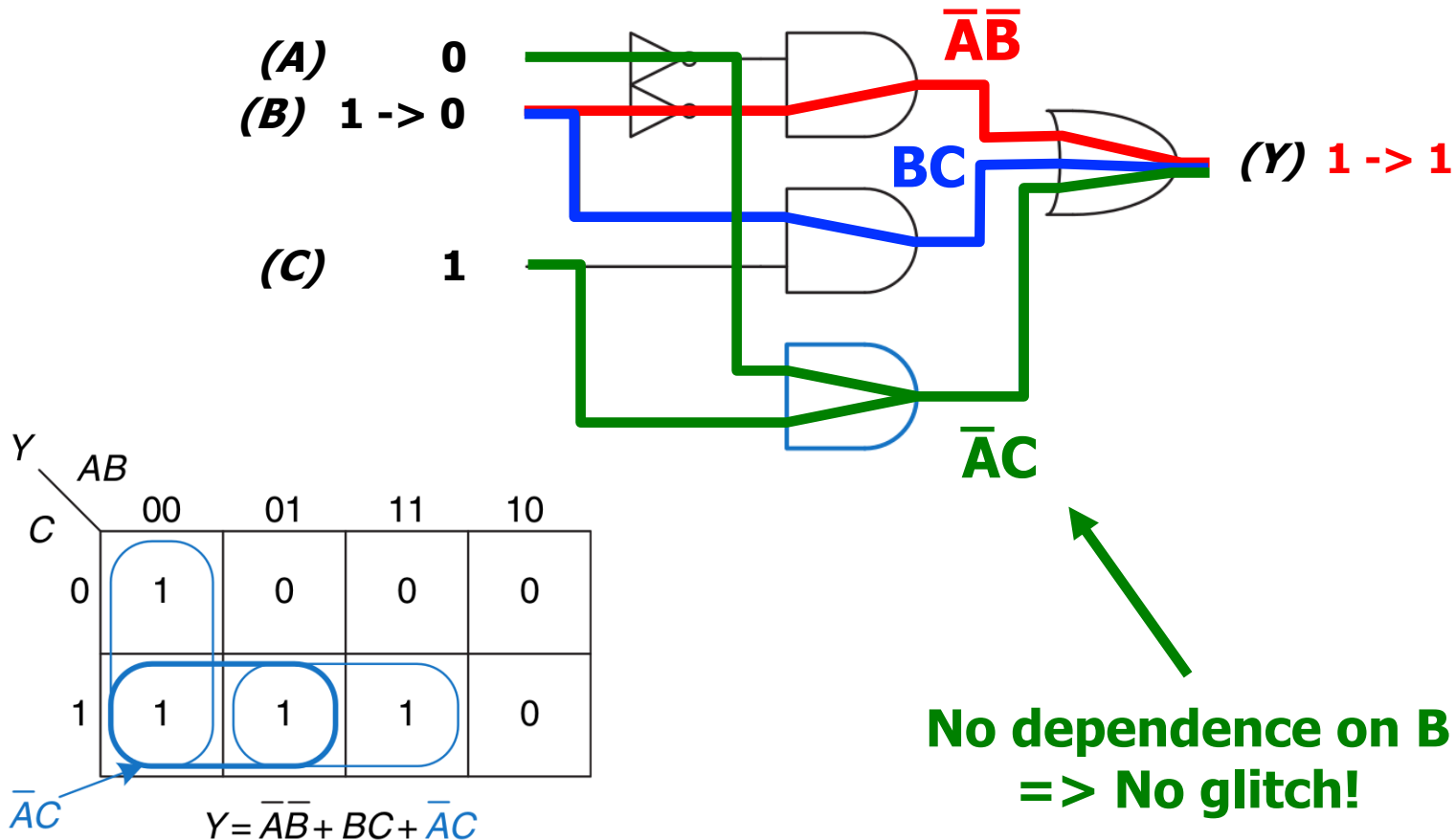
- Glitches are **visible** in **K-maps**
  - Recall: K-maps show the results of a change in a **single input**
  - A glitch occurs when **moving between prime implicants**



$$Y = \overline{AB} + BC$$

# Optional: Avoiding Glitches Using K-Maps

- We can **fix** the issue by adding in the **consensus** term
  - Ensures **no transition** between different **prime implicants**



# Avoiding Glitches

---

- **Q:** Do we **always** care about glitches?
  - **Fixing** glitches is **undesirable**
    - More chip **area**
    - More **power consumption**
    - More **design effort**
  - The circuit is **eventually** guaranteed to **converge** to the **right value** regardless of glitchiness
  
- **A:** No, not always!
  - If we only care about the **long-term steady state output**, we can **safely ignore** glitches
  - Up to the **designer to decide** if glitches matter in their application
    - When examining simulation output, important to recognize glitches

# Part 2:

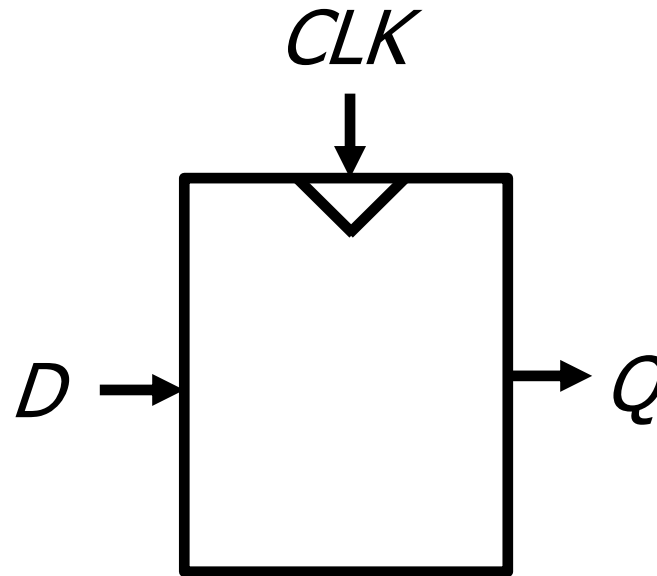
## Sequential Circuit Timing



# Recall: D Flip-Flop

---

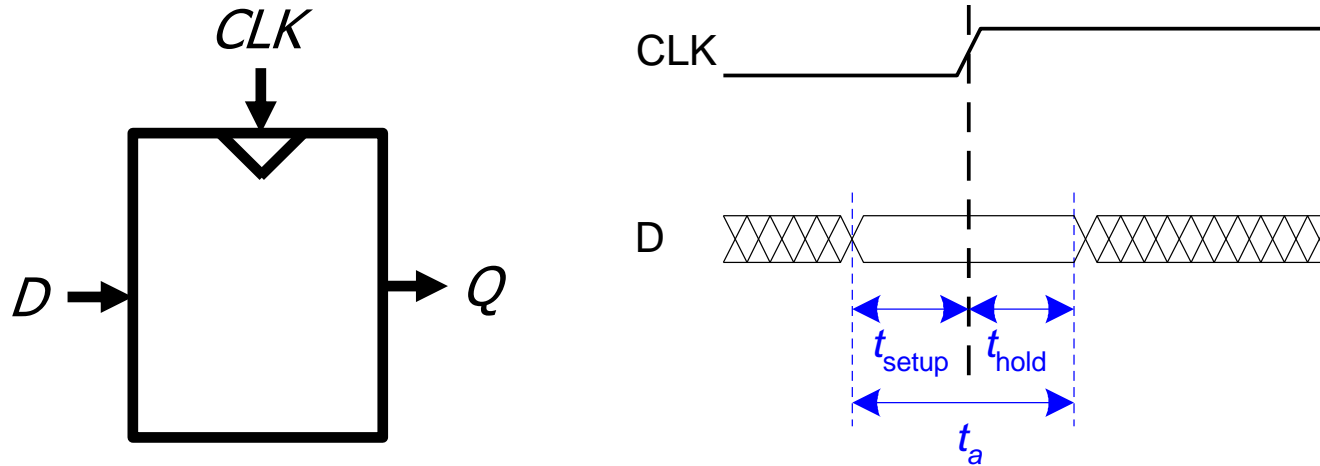
- Flip-flop **samples D** at the **active clock edge**
  - It outputs the **sampled value** to Q
  - It **"stores"** the **sampled value** until the next active clock edge



- The D flip-flop is **made** from **combinational** elements
- **D, Q, CLK all have timing requirements!**

# D Flip-Flop Input Timing Constraints

- **D** must be **stable** when **sampled** (i.e., at active clock edge)

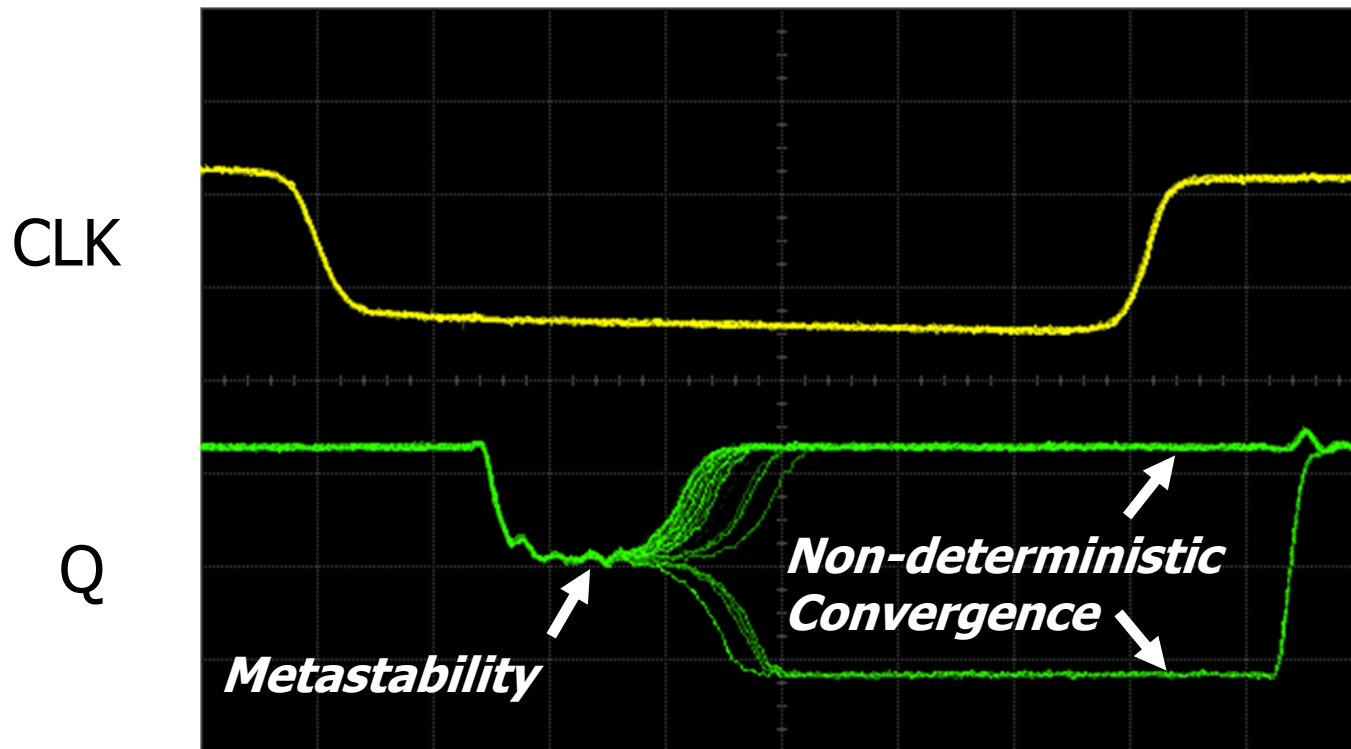


- **Setup time ( $t_{\text{setup}}$ )**: time **before** the clock edge that data must be stable (i.e. not changing)
- **Hold time ( $t_{\text{hold}}$ )**: time **after** the clock edge that data must be stable
- **Aperture time ( $t_a$ )**: time **around** clock edge that data must be stable ( $t_a = t_{\text{setup}} + t_{\text{hold}}$ )

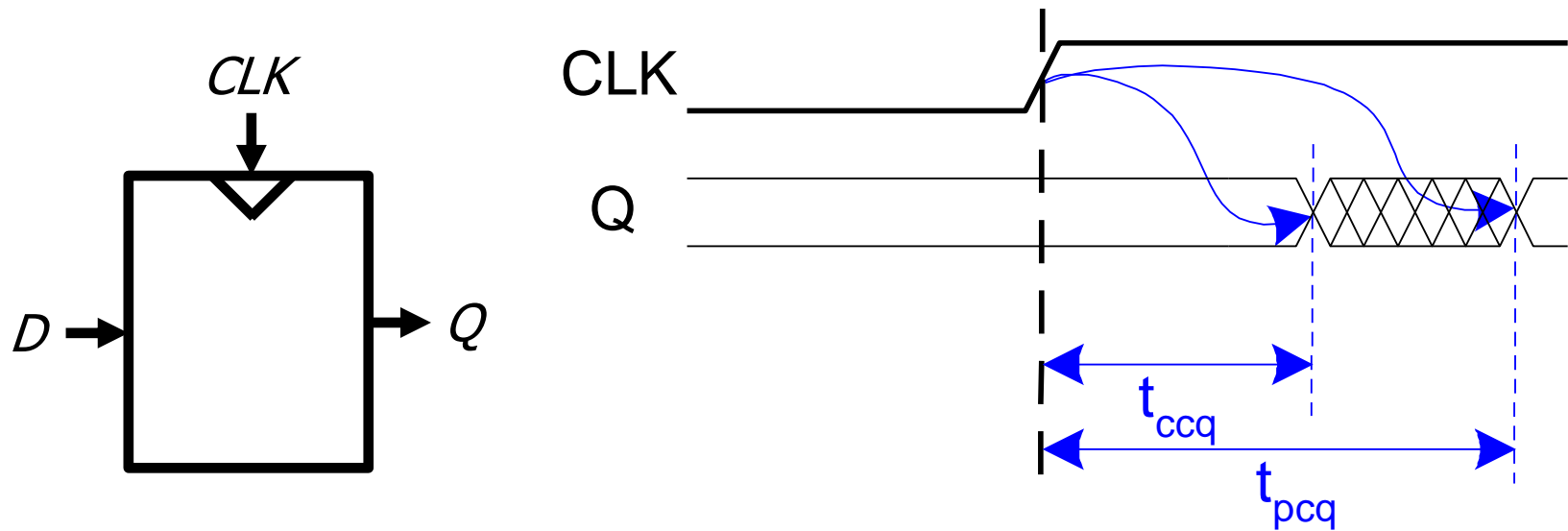
# Violating Input Timing: Metastability

- If D is **changing** when sampled, **metastability** can occur
  - ❑ Flip-flop output is **stuck** somewhere between '1' and '0'
  - ❑ Output eventually settles **non-deterministically**

## Example Timing Violations (NAND RS Latch)



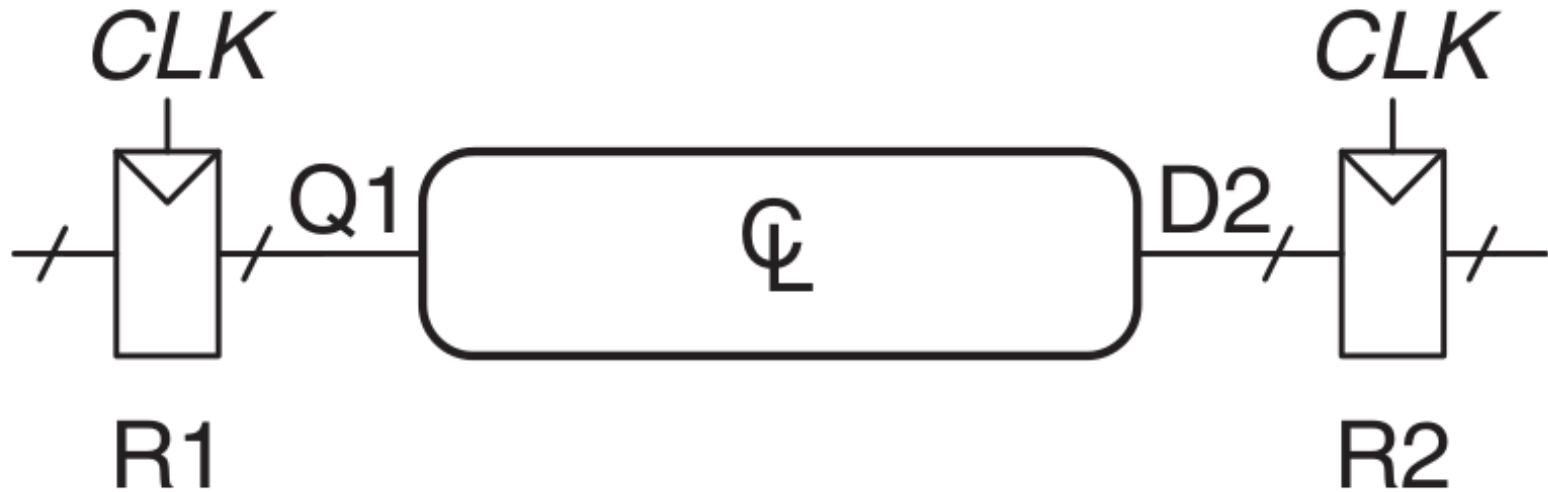
# Flip-Flop Output Timing



- **Contamination delay clock-to-q ( $t_{ccq}$ ):** earliest time after the clock edge that  $Q$  starts to change (i.e., is unstable)
- **Propagation delay clock-to-q ( $t_{pcq}$ ):** latest time after the clock edge that  $Q$  stops changing (i.e., is stable)

# Recall: Sequential System Design

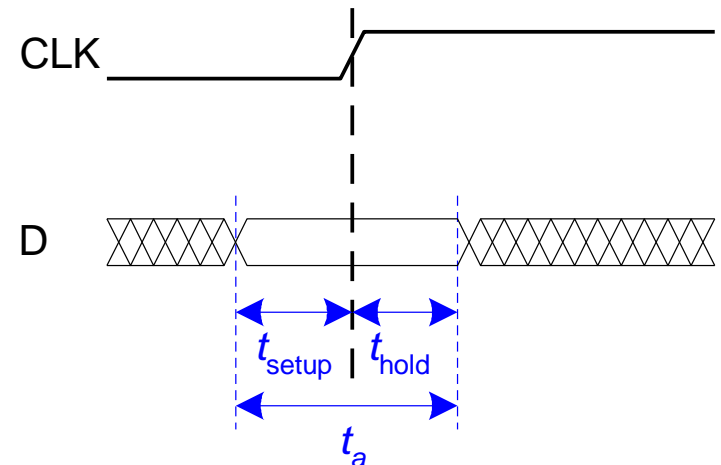
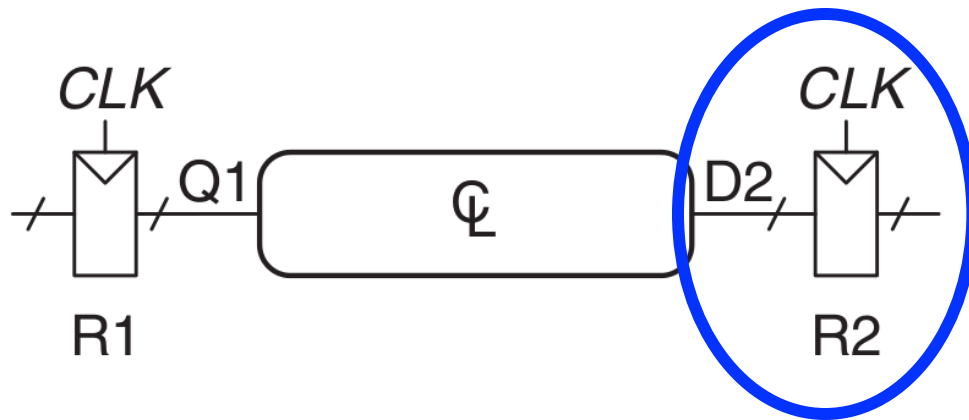
---



- Multiple **flip-flops** are connected with **combinational logic**
- **Clock** runs with period  $T_c$  (cycle time)
- **Must meet timing requirements for both R1 and R2!**

# Ensuring Correct Sequential Operation

- Need to ensure correct input timing on **R2**
- Specifically, **D2** must be **stable**:
  - at least  $t_{\text{setup}}$  **before** the clock edge
  - at least until  $t_{\text{hold}}$  **after** the clock edge

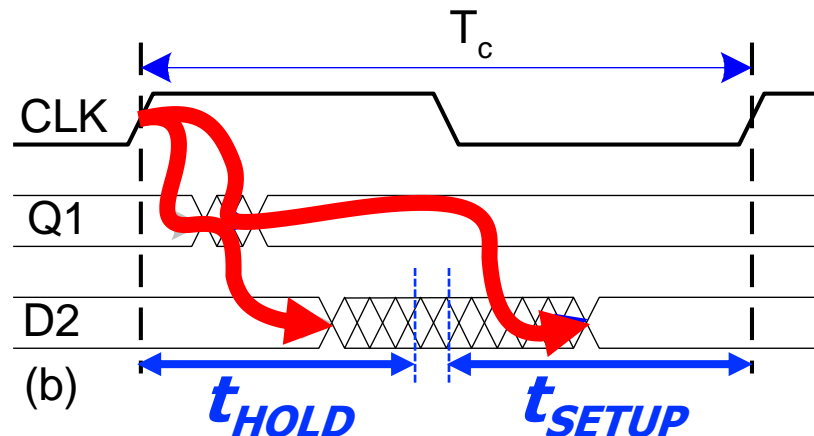
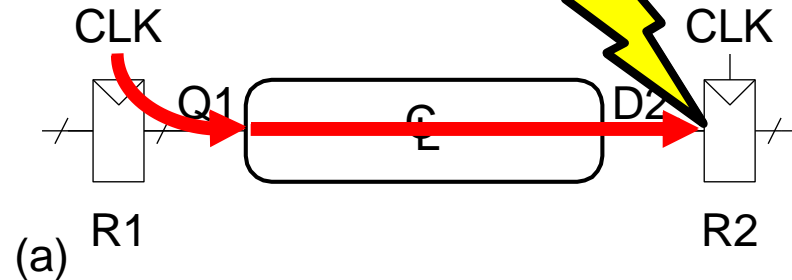


# Ensuring Correct Sequential Operation

- This means there is both a **minimum** and **maximum** delay between two flip-flops

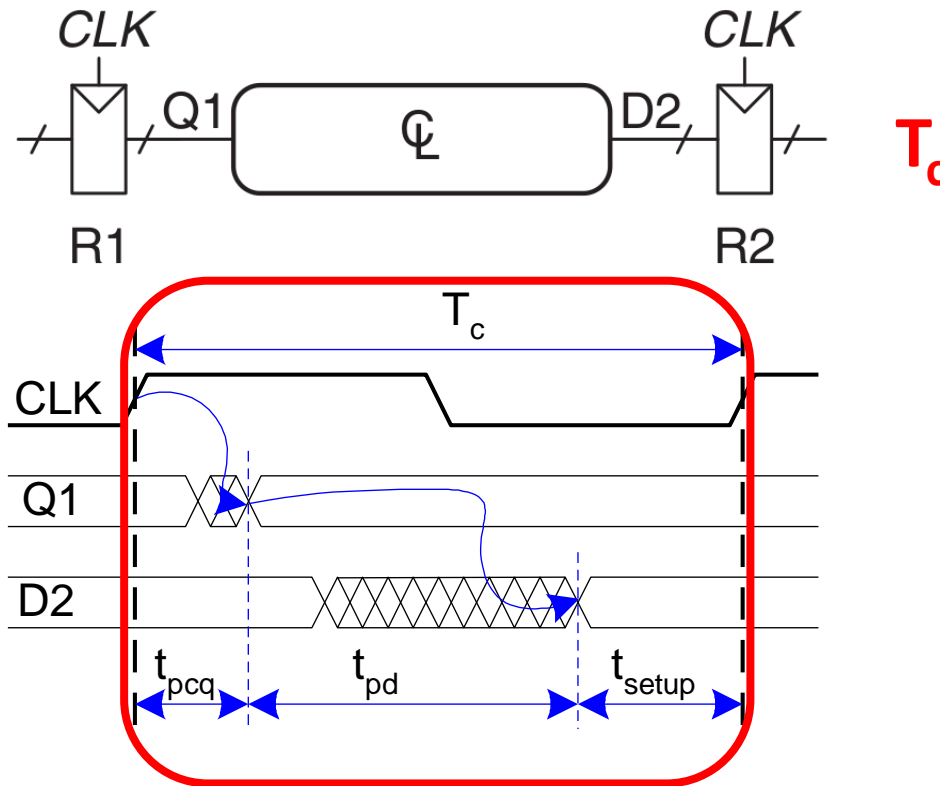
- CL **too fast** -> R2  $t_{\text{hold}}$  violation
- CL **too slow** -> R2  $t_{\text{setup}}$  violation

**Potential  
R2  $t_{\text{setup}}$   
VIOLATION!**



# Setup Time Constraint

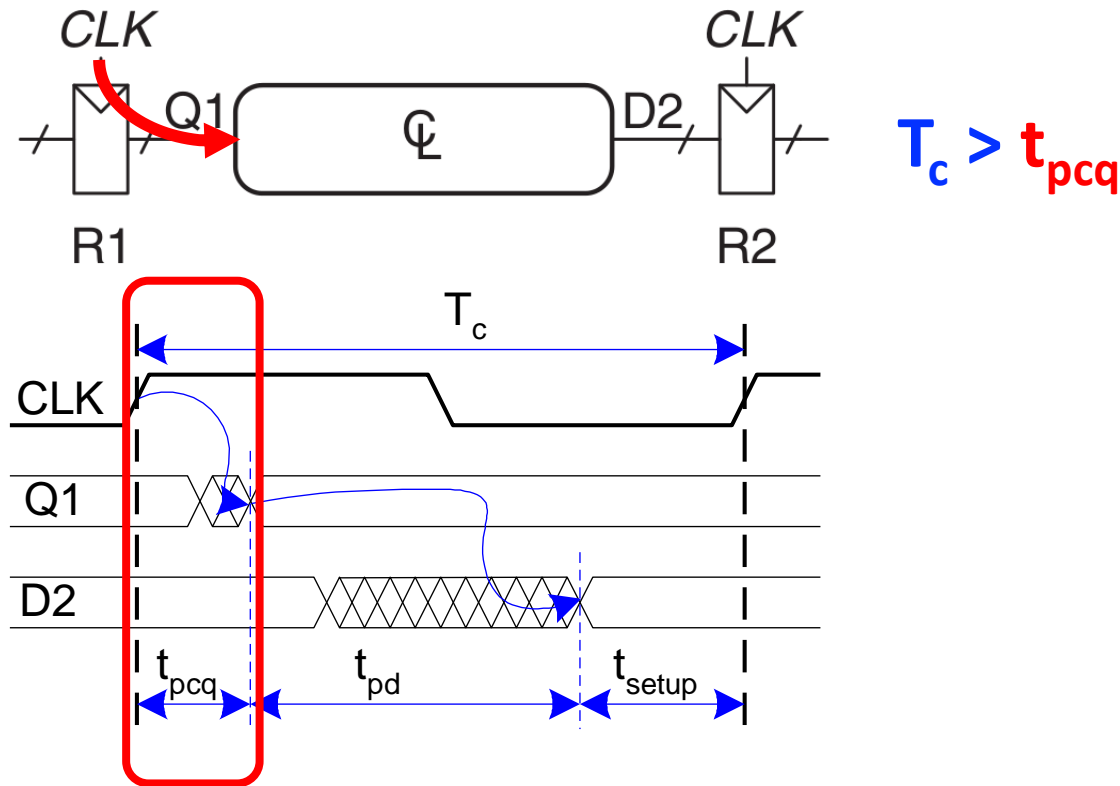
- **Safe timing** depends on the **maximum** delay from R1 to R2
- The input to R2 must be stable at least  $t_{\text{setup}}$  **before** the clock edge.





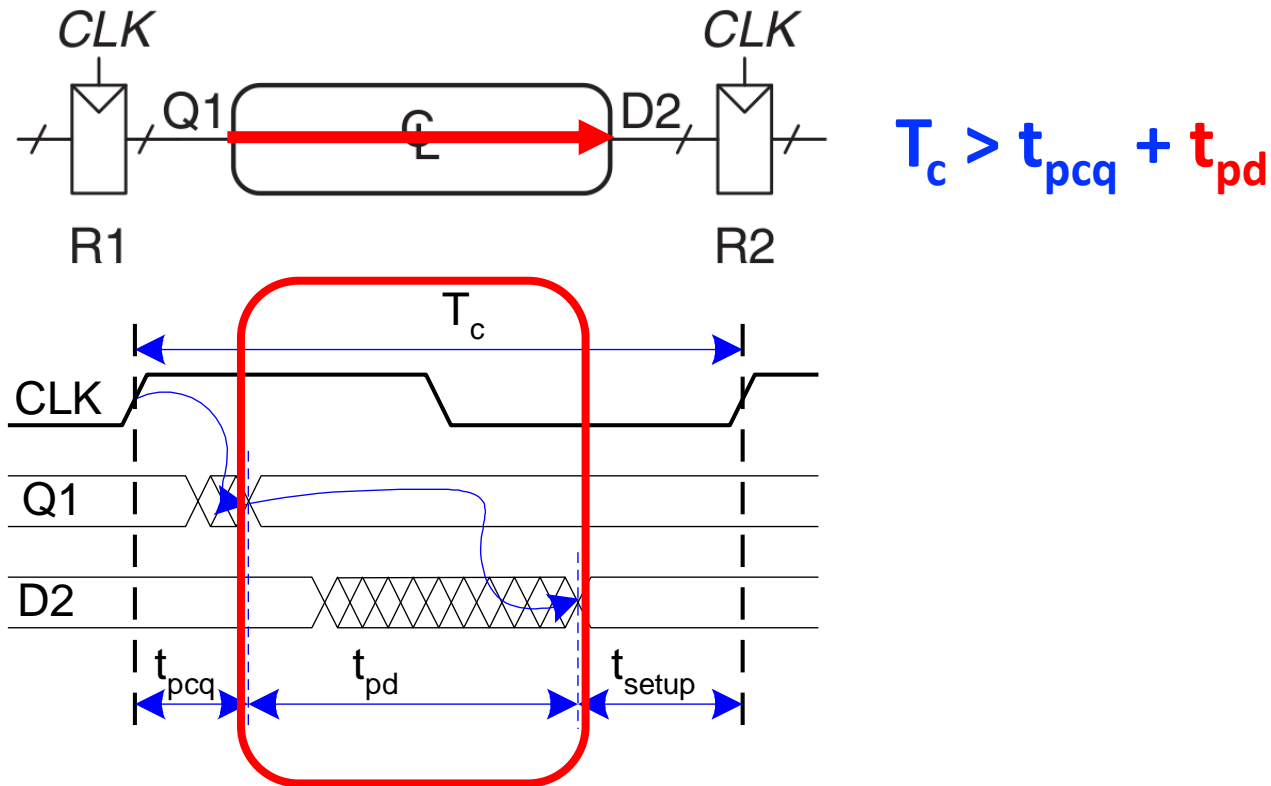
# Setup Time Constraint

- **Safe timing** depends on the **maximum** delay from R1 to R2
- The input to R2 must be stable at least  $t_{\text{setup}}$  **before** the clock edge.



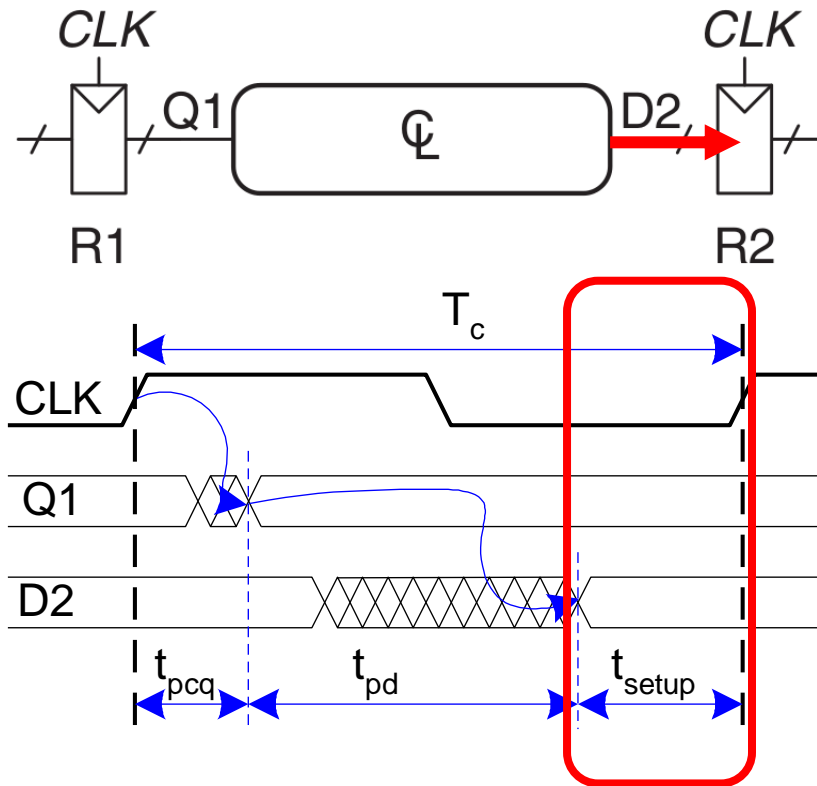
# Setup Time Constraint

- **Safe timing** depends on the **maximum** delay from R1 to R2
- The input to R2 must be stable at least  $t_{\text{setup}}$  **before** the clock edge.



# Setup Time Constraint

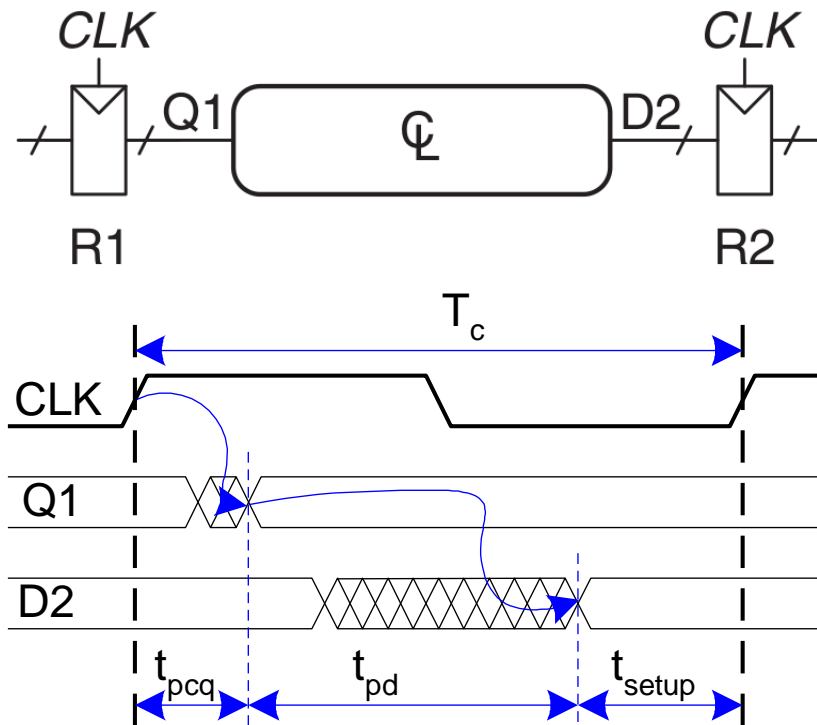
- **Safe timing** depends on the **maximum** delay from R1 to R2
- The input to R2 must be stable at least  $t_{\text{setup}}$  **before** the clock edge.



$$T_c > t_{\text{pcq}} + t_{\text{pd}} + t_{\text{setup}}$$

# Setup Time Constraint

- **Safe timing** depends on the **maximum** delay from R1 to R2
- The input to R2 must be stable at least  $t_{\text{setup}}$  **before** the clock edge.

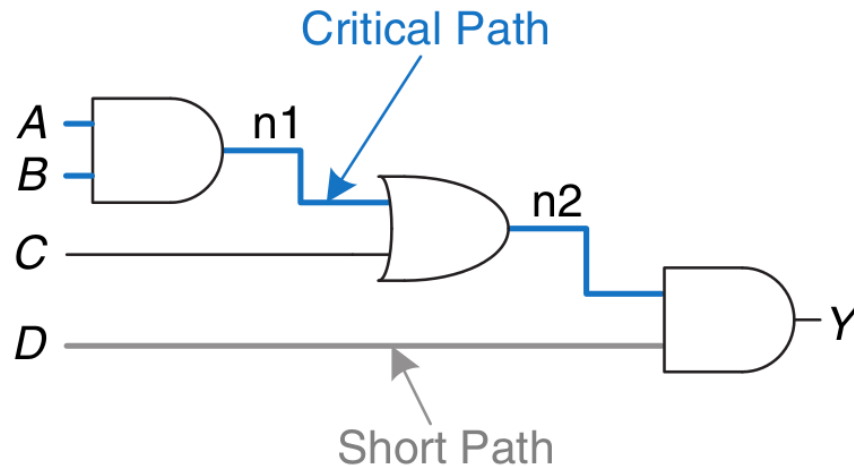


$$T_c > \underbrace{t_{\text{pcq}} + t_{\text{pd}}}_{\text{Useful work}} + t_{\text{setup}}$$

Wasted work

**Sequencing overhead:**  
amount of time **wasted**  
each cycle due to sequencing  
element timing requirements

# $t_{\text{setup}}$ Constraint and Design Performance



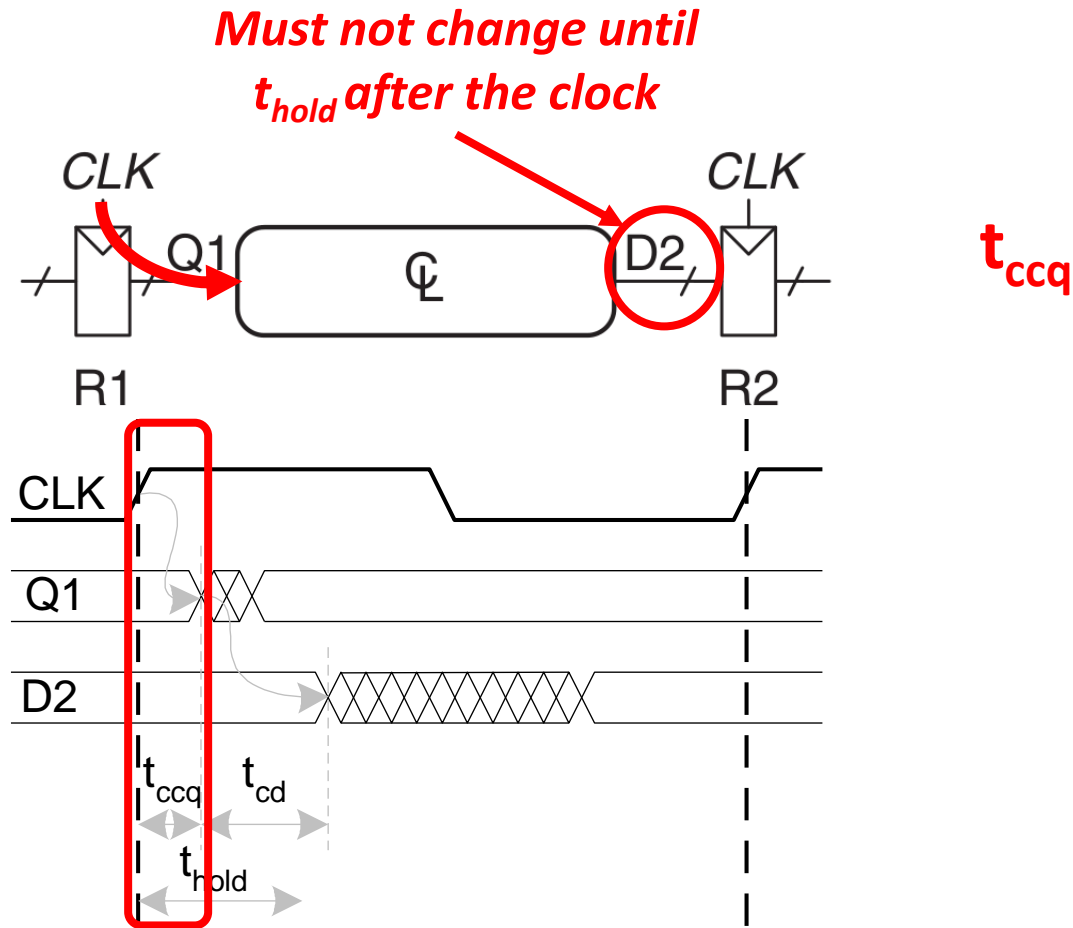
- **Critical path:** path with the longest  $t_{\text{pd}}$

$$T_c > t_{\text{pcq}} + t_{\text{pd}} + t_{\text{setup}}$$

- Overall design performance is determined by the critical path  $t_{\text{pd}}$ 
  - Determines the **minimum clock period** (i.e., **max operating frequency**)
  - If the critical path is too **long**, the design will run **slowly**
  - If critical path is too **short**, each cycle will do very **little useful work**
    - i.e., most of the cycle will be **wasted** in sequencing overhead

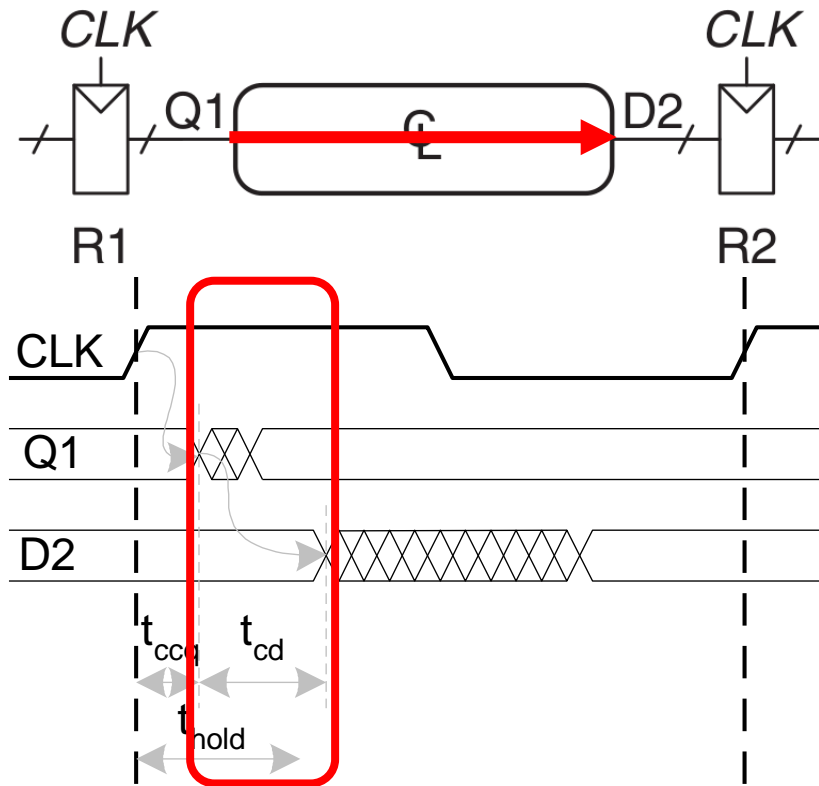
# Hold Time Constraint

- **Safe timing** depends on the **minimum** delay from R1 to R2
- **D2** (i.e., R2 input) must be stable for at least  $t_{hold}$  **after** the clock edge



# Hold Time Constraint

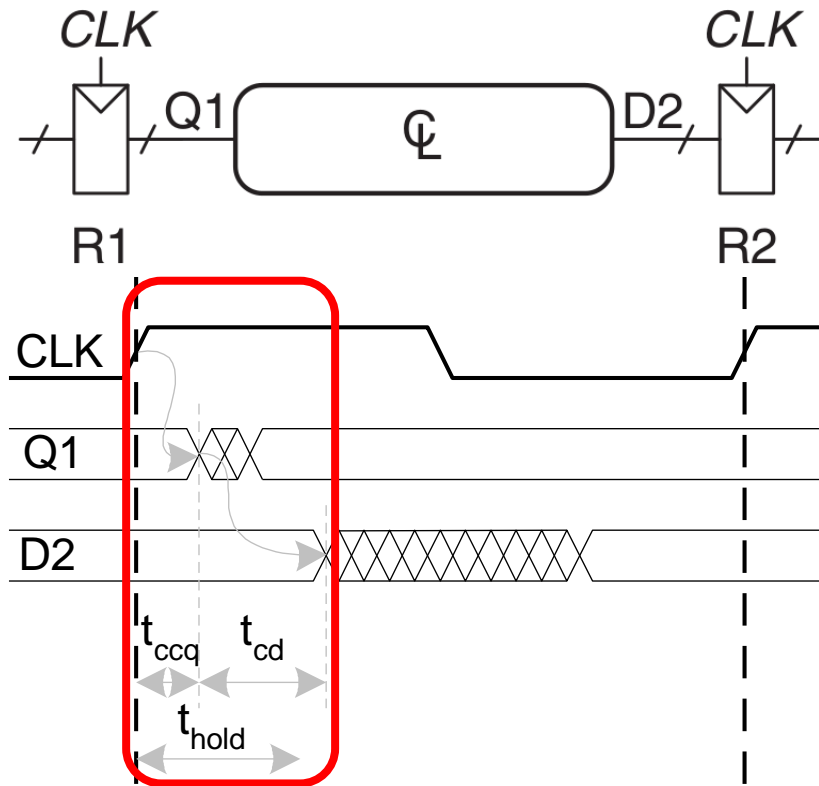
- **Safe timing** depends on the **minimum** delay from R1 to R2
- **D2** (i.e., R2 input) must be stable for at least  $t_{\text{hold}}$  **after** the clock edge



$$t_{\text{ccq}} + t_{\text{cd}}$$

# Hold Time Constraint

- **Safe timing** depends on the **minimum** delay from R1 to R2
- **D2** (i.e., R2 input) must be stable for at least  $t_{\text{hold}}$  **after** the clock edge

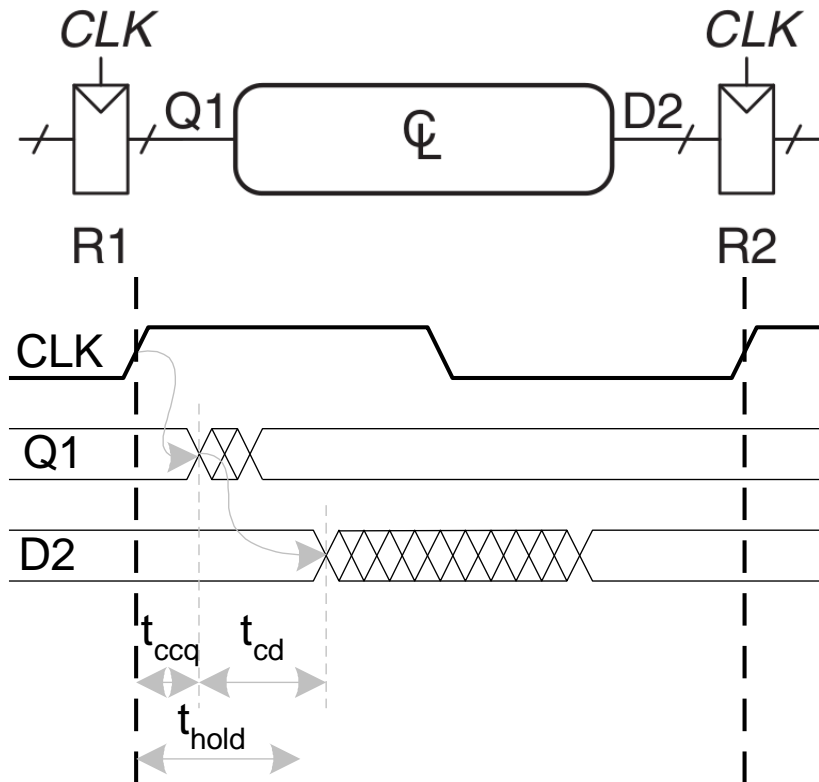


$$t_{\text{ccq}} + t_{\text{cd}} > t_{\text{hold}}$$



# Hold Time Constraint

- **Safe timing** depends on the **minimum** delay from R1 to R2
- **D2** (i.e., R2 input) must be stable for at least  $t_{\text{hold}}$  **after** the clock edge



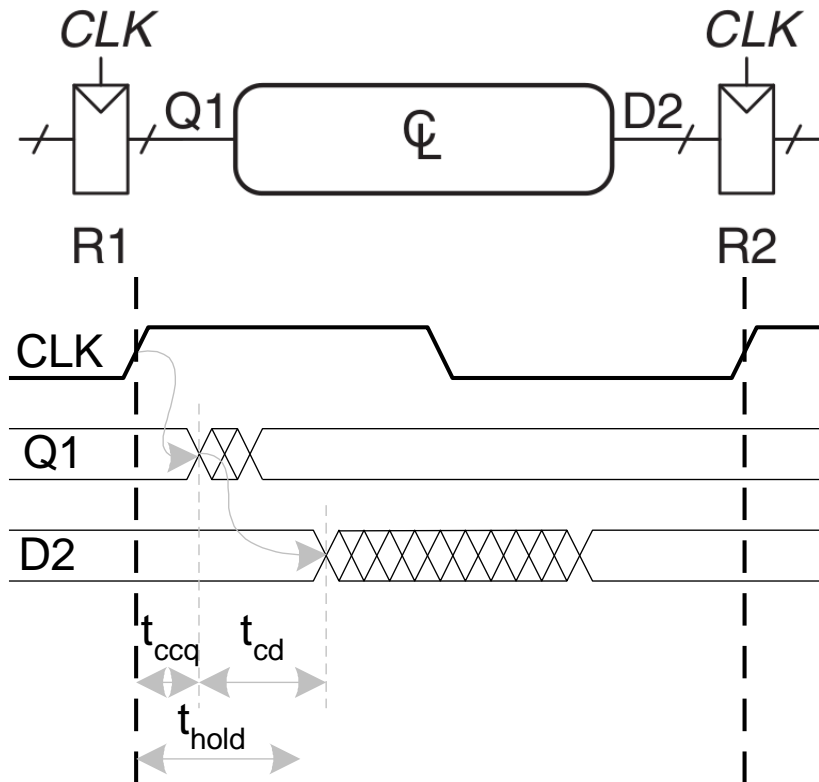
$$t_{\text{ccq}} + t_{\text{cd}} > t_{\text{hold}}$$

$$t_{\text{cd}} > t_{\text{hold}} - t_{\text{ccq}}$$

We need to have a **minimum** combinational delay!

# Hold Time Constraint

- **Safe timing** depends on the **minimum** delay from R1 to R2
- **D2** (i.e., R2 input) must be stable for at least  $t_{\text{hold}}$  **after** the clock edge



$$t_{\text{ccq}} + t_{\text{cd}} > t_{\text{hold}}$$

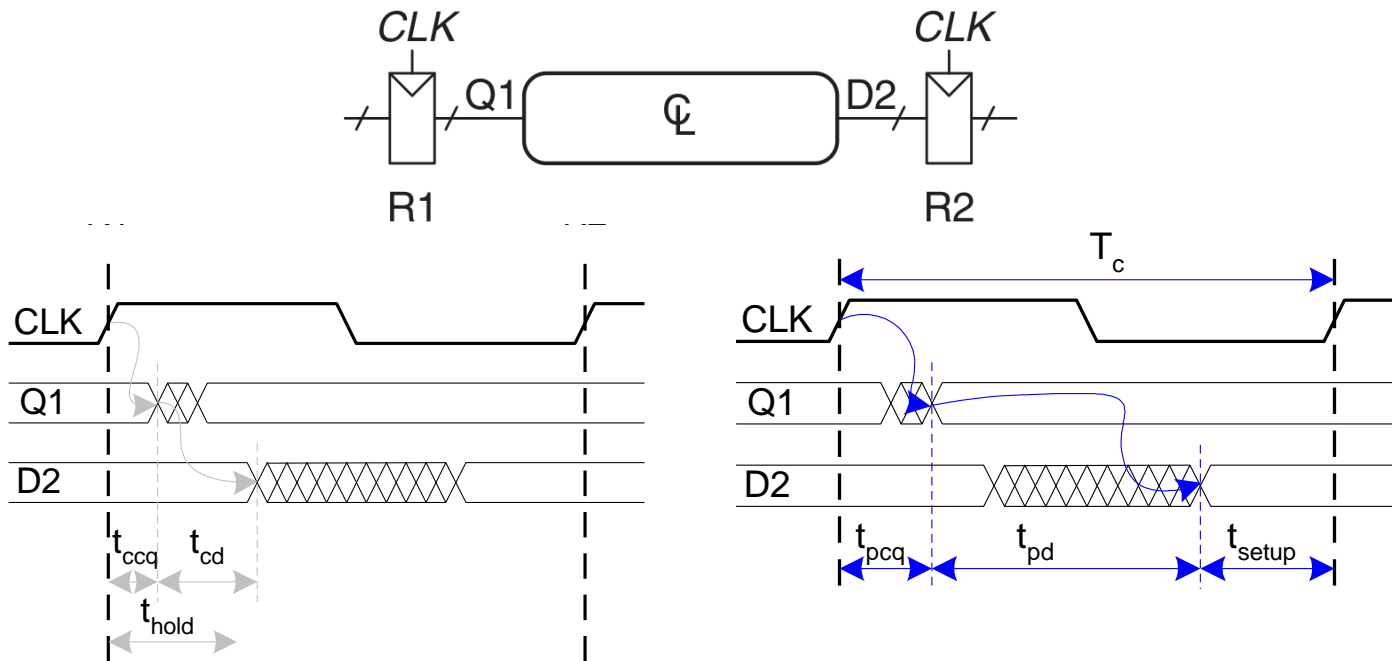
$$t_{\text{cd}} > t_{\text{hold}} - t_{\text{ccq}}$$

Does **NOT** depend on  $T_c$ !

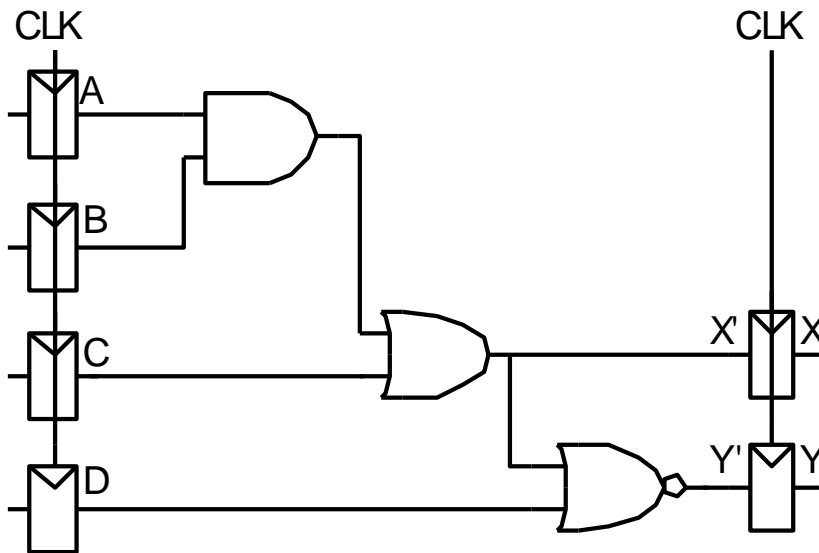
**Very hard** to fix  $t_{\text{hold}}$  violations after manufacturing- must modify circuits!

# Sequential Timing Summary

$t_{ccq} / t_{pcq}$	<b>clock-to-q</b> delay (contamination/propagation)
$t_{cd} / t_{pd}$	<b>combinational logic</b> delay (contamination/propagation)
$t_{setup}$	time that <b>FF inputs</b> must be stable <b>before</b> next clock edge
$t_{hold}$	time that <b>FF inputs</b> must be stable <b>after</b> a clock edge
$T_c$	clock period



## Example: Timing Analysis

 $t_{pd} =$  $t_{cd} =$ 

## Check setup time constraints:

$$T_c > t_{pcq} + t_{pd} + t_{setup}$$

$T_c >$

$$f_{max} = 1/T_c =$$

## Timing Characteristics

$$t_{ccq} = 30 \text{ ps}$$

$$t_{pcq} = 50 \text{ ps}$$

$$t_{\text{setup}} = 60 \text{ ps}$$

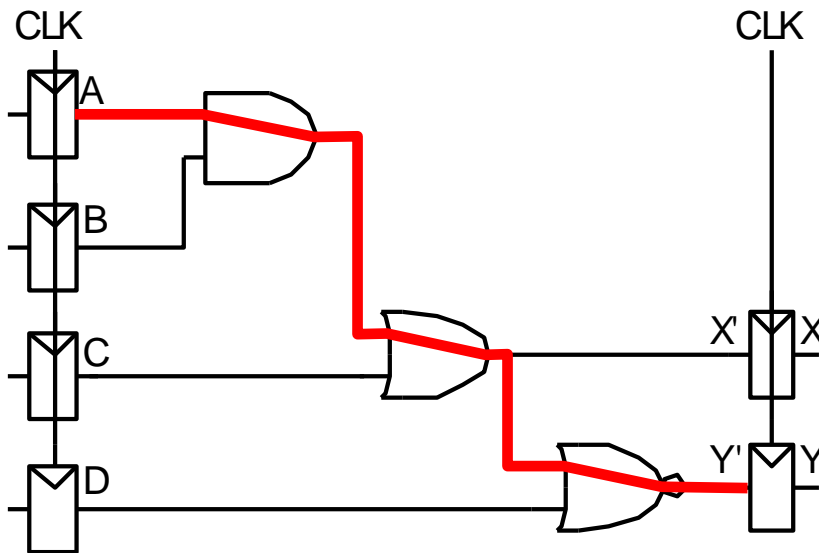
$$t_{\text{hold}} = 70 \text{ ps}$$

per gate  $\left[ \begin{array}{l} t_{pd} = 35 \text{ ps} \\ t_{cd} = 25 \text{ ps} \end{array} \right.$

### Check hold time constraints:

$$t_{ccq} + t_{cd} > t_{hold} ?$$

## Example: Timing Analysis



$$t_{pd} = 3 \times 35 \text{ ps} = 105 \text{ ps}$$

$t_{cd} =$

## Check setup time constraints:

$$T_c > t_{pcq} + t_{pd} + t_{setup}$$

$T_c >$

$$f_{max} = 1/T_c =$$

## Timing Characteristics

$$t_{ccq} = 30 \text{ ps}$$

$$t_{pcq} = 50 \text{ ps}$$

$$t_{\text{setup}} = 60 \text{ ps}$$

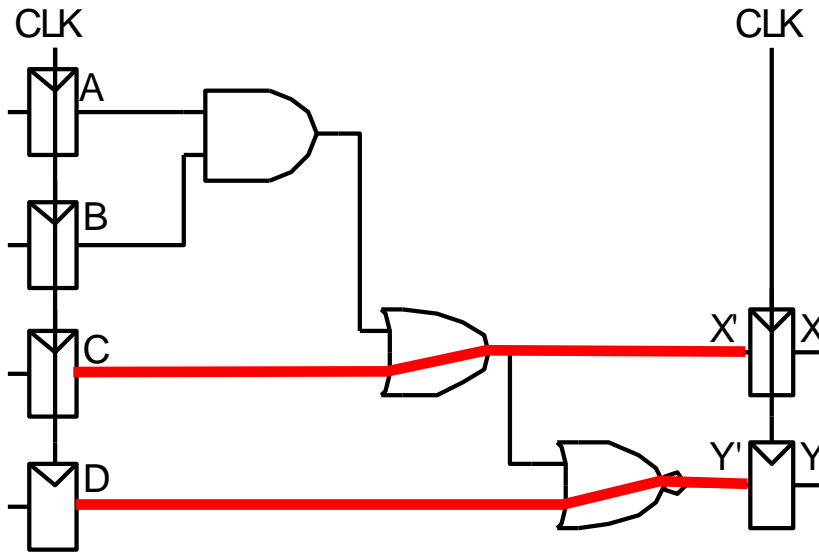
$$t_{\text{hold}} = 70 \text{ ps}$$

per gate  $\left[ \begin{array}{l} t_{pd} = 35 \text{ ps} \\ t_{cd} = 25 \text{ ps} \end{array} \right.$

### Check hold time constraints:

$$t_{ccq} + t_{cd} > t_{hold} ?$$

# Example: Timing Analysis



$$t_{pd} = 3 \times 35 \text{ ps} = 105 \text{ ps}$$

$$t_{cd} = 25 \text{ ps}$$

Check setup time constraints:

$$T_c > t_{pcq} + t_{pd} + t_{setup}$$

$$T_c >$$

$$f_{max} = 1/T_c =$$

## Timing Characteristics

$$t_{ccq} = 30 \text{ ps}$$

$$t_{pcq} = 50 \text{ ps}$$

$$t_{setup} = 60 \text{ ps}$$

$$t_{hold} = 70 \text{ ps}$$

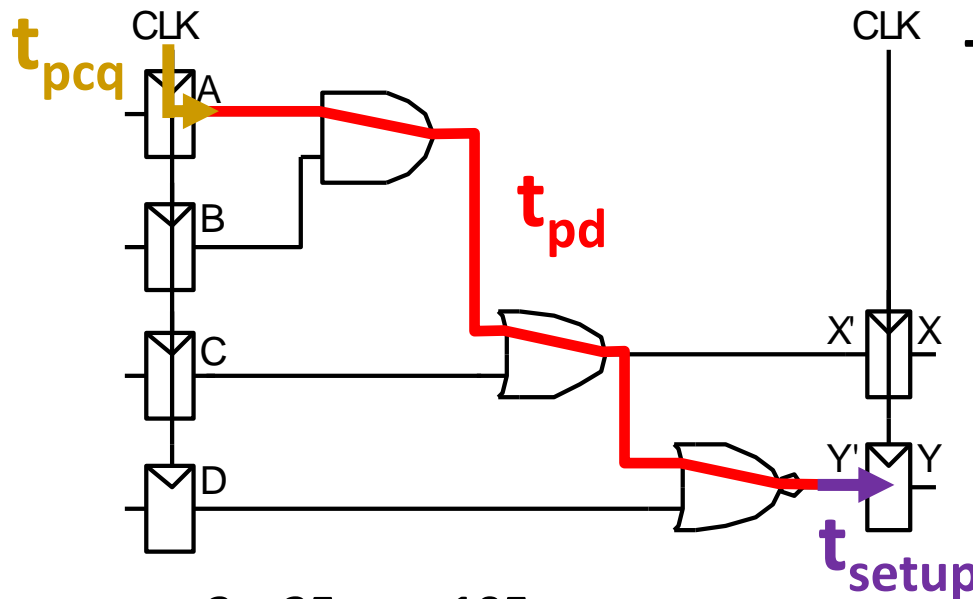
per gate

$$\left[ \begin{array}{l} t_{pd} \\ t_{cd} \end{array} \right] = \left[ \begin{array}{l} 35 \text{ ps} \\ 25 \text{ ps} \end{array} \right]$$

Check hold time constraints:

$$t_{ccq} + t_{cd} > t_{hold} ?$$

# Example: Timing Analysis



$$t_{pd} = 3 \times 35 \text{ ps} = 105 \text{ ps}$$

$$t_{cd} = 25 \text{ ps}$$

Check setup time constraints:

$$T_c > t_{pcq} + t_{pd} + t_{setup}$$

$$T_c > (50 + 105 + 60) \text{ ps} = 215 \text{ ps}$$

$$f_{max} = 1/T_c = 4.65 \text{ GHz}$$

## Timing Characteristics

$$t_{ccq} = 30 \text{ ps}$$

$$t_{pcq} = 50 \text{ ps}$$

$$t_{setup} = 60 \text{ ps}$$

$$t_{hold} = 70 \text{ ps}$$

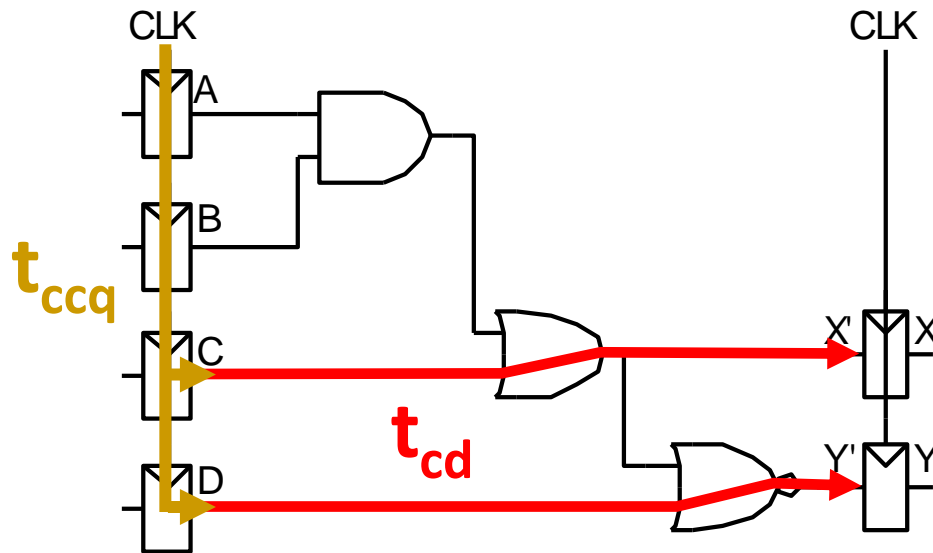
per gate

$$\begin{cases} t_{pd} = 35 \text{ ps} \\ t_{cd} = 25 \text{ ps} \end{cases}$$

Check hold time constraints:

$$t_{ccq} + t_{cd} > t_{hold} ?$$

# Example: Timing Analysis



$$t_{pd} = 3 \times 35 \text{ ps} = 105 \text{ ps}$$

$$t_{cd} = 25 \text{ ps}$$

Check setup time constraints:

$$T_c > t_{pcq} + t_{pd} + t_{setup}$$

$$T_c > (50 + 105 + 60) \text{ ps} = 215 \text{ ps}$$

$$f_{max} = 1/T_c = 4.65 \text{ GHz}$$

## Timing Characteristics

$$t_{ccq} = 30 \text{ ps}$$

$$t_{pcq} = 50 \text{ ps}$$

$$t_{setup} = 60 \text{ ps}$$

$$t_{hold} = 70 \text{ ps}$$

per gate

$$\begin{cases} t_{pd} = 35 \text{ ps} \\ t_{cd} = 25 \text{ ps} \end{cases}$$

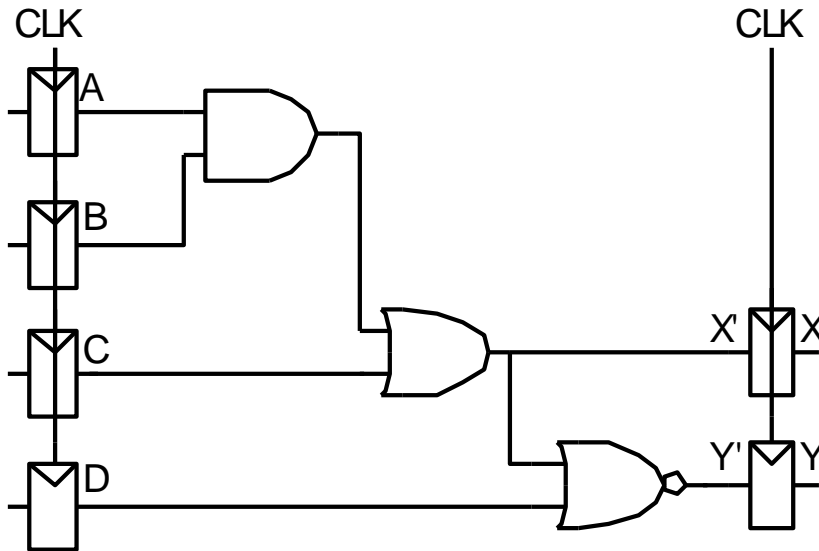
Check hold time constraints:

$$t_{ccq} + t_{cd} > t_{hold} ?$$

$$(30 + 25) \text{ ps} > 70 \text{ ps} ?$$



# Example: Timing Analysis



$$t_{pd} = 3 \times 35 \text{ ps} = 105 \text{ ps}$$

$$t_{cd} = 25 \text{ ps}$$

Check setup time constraints:

$$T_c > t_{pcq} + t_{pd} + t_{setup}$$

$$T_c > (50 + 105 + 60) \text{ ps} = 215 \text{ ps}$$

$$f_{max} = 1/T_c = 4.65 \text{ GHz}$$

## Timing Characteristics

$$t_{ccq} = 30 \text{ ps}$$

$$t_{pcq} = 50 \text{ ps}$$

$$t_{setup} = 60 \text{ ps}$$

$$t_{hold} = 70 \text{ ps}$$

per gate

$$\left[ \begin{array}{l} t_{pd} = 35 \text{ ps} \\ t_{cd} = 25 \text{ ps} \end{array} \right.$$

Check hold time constraints:

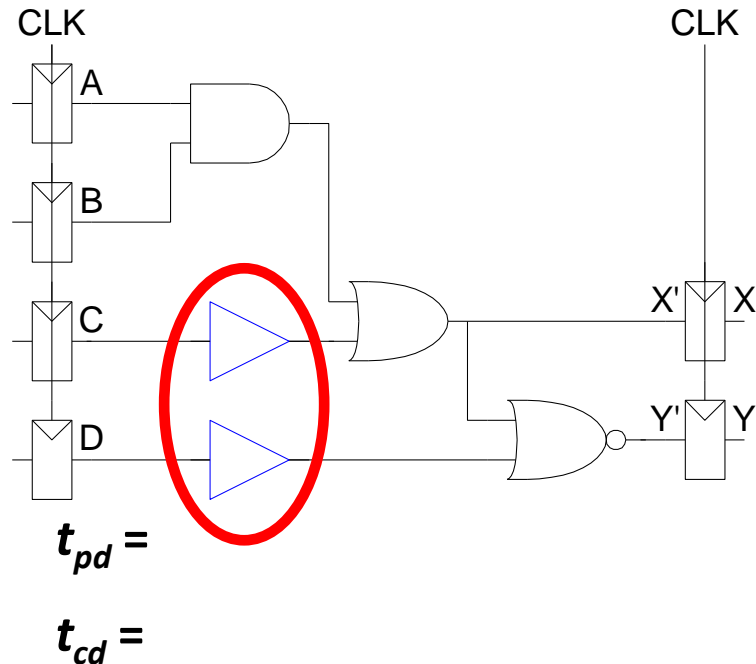
$$t_{ccq} + t_{cd} > t_{hold} ?$$

$$(30 + 25) \text{ ps} > 70 \text{ ps} ?$$

**FAIL**

# Example: Fixing Hold Time Violation

Add buffers to the short paths:



Check setup time constraints:

$$T_c > t_{pcq} + t_{pd} + t_{setup}$$

$$T_c >$$

$$f_c =$$

## Timing Characteristics

$$t_{ccq} = 30 \text{ ps}$$

$$t_{pcq} = 50 \text{ ps}$$

$$t_{setup} = 60 \text{ ps}$$

$$t_{hold} = 70 \text{ ps}$$

per gate

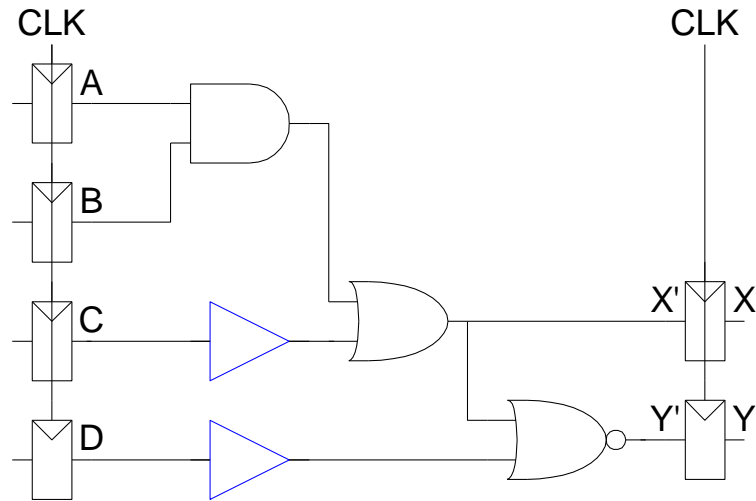
$$\begin{cases} t_{pd} = 35 \text{ ps} \\ t_{cd} = 25 \text{ ps} \end{cases}$$

Check hold time constraints:

$$t_{ccq} + t_{cd} > t_{hold} ?$$

# Example: Fixing Hold Time Violation

Add buffers to the short paths:



$$t_{pd} = 3 \times 35 \text{ ps} = 105 \text{ ps}$$

$$t_{cd} = 2 \times 25 \text{ ps} = 50 \text{ ps}$$

Check setup time constraints:

$$T_c > t_{pcq} + t_{pd} + t_{setup}$$

$$T_c >$$

$$f_c =$$

## Timing Characteristics

$$t_{ccq} = 30 \text{ ps}$$

$$t_{pcq} = 50 \text{ ps}$$

$$t_{setup} = 60 \text{ ps}$$

$$t_{hold} = 70 \text{ ps}$$

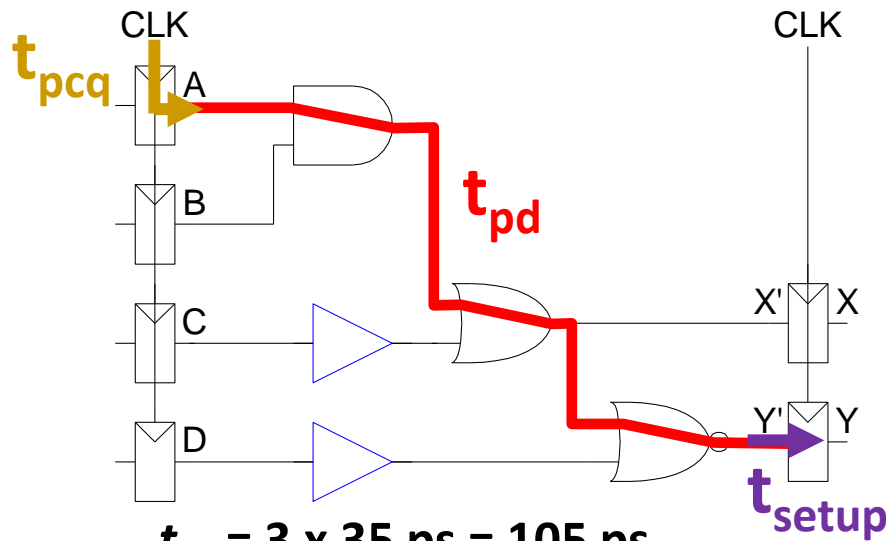
$$\text{per gate} \left[ \begin{array}{l} t_{pd} = 35 \text{ ps} \\ t_{cd} = 25 \text{ ps} \end{array} \right.$$

Check hold time constraints:

$$t_{ccq} + t_{cd} > t_{hold} ?$$

# Example: Fixing Hold Time Violation

Add buffers to the short paths:



Check setup time constraints:

$$T_c > t_{pcq} + t_{pd} + t_{setup}$$

$$T_c > (50 + 105 + 60) \text{ ps} = 215 \text{ ps}$$

$$f_c = 1/T_c = 4.65 \text{ GHz}$$

## Timing Characteristics

$$t_{ccq} = 30 \text{ ps}$$

$$t_{pcq} = 50 \text{ ps}$$

$$t_{setup} = 60 \text{ ps}$$

$$t_{hold} = 70 \text{ ps}$$

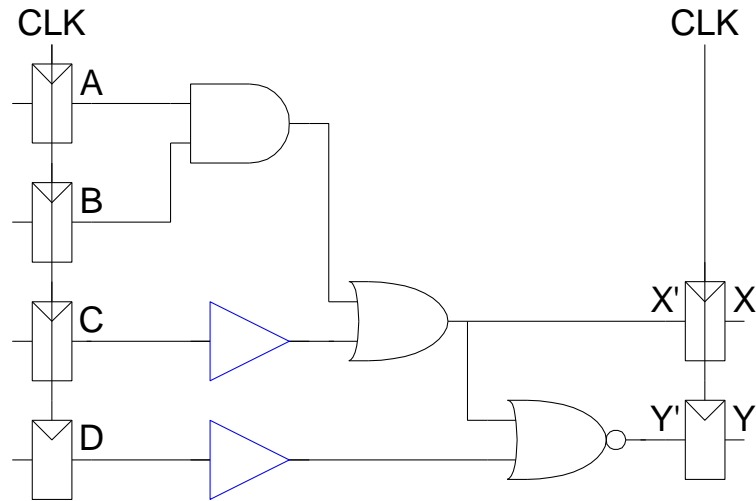
$$\text{per gate} \begin{cases} t_{pd} = 35 \text{ ps} \\ t_{cd} = 25 \text{ ps} \end{cases}$$

Check hold time constraints:

$$t_{ccq} + t_{cd} > t_{hold} ?$$

# Example: Fixing Hold Time Violation

Add buffers to the short paths:



$$t_{pd} = 3 \times 35 \text{ ps} = 105 \text{ ps}$$

$$t_{cd} = 2 \times 25 \text{ ps} = 50 \text{ ps}$$

Check setup time constraints:

$$T_c > t_{pcq} + t_{pd} + t_{setup}$$

$$T_c > (50 + 105 + 60) \text{ ps} = 215 \text{ ps}$$

$$f_c = 1/T_c = \mathbf{4.65 \text{ GHz}}$$

Note: no change to max frequency!

## Timing Characteristics

$$t_{ccq} = 30 \text{ ps}$$

$$t_{pcq} = 50 \text{ ps}$$

$$t_{setup} = 60 \text{ ps}$$

$$t_{hold} = 70 \text{ ps}$$

per gate

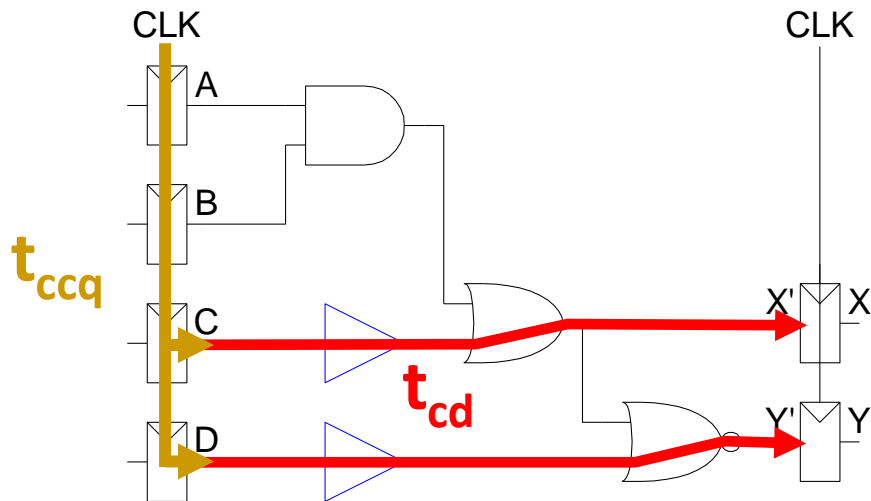
$$\left[ \begin{array}{l} t_{pd} = 35 \text{ ps} \\ t_{cd} = 25 \text{ ps} \end{array} \right]$$

Check hold time constraints:

$$t_{ccq} + t_{cd} > t_{hold} ?$$

# Example: Fixing Hold Time Violation

Add buffers to the short paths:



$$t_{pd} = 3 \times 35 \text{ ps} = 105 \text{ ps}$$

$$t_{cd} = 2 \times 25 \text{ ps} = 50 \text{ ps}$$

Check setup time constraints:

$$T_c > t_{pcq} + t_{pd} + t_{setup}$$

$$T_c > (50 + 105 + 60) \text{ ps} = 215 \text{ ps}$$

$$f_c = 1/T_c = 4.65 \text{ GHz}$$

## Timing Characteristics

$$t_{ccq} = 30 \text{ ps}$$

$$t_{pcq} = 50 \text{ ps}$$

$$t_{setup} = 60 \text{ ps}$$

$$t_{hold} = 70 \text{ ps}$$

$$\text{per gate} \begin{cases} t_{pd} = 35 \text{ ps} \\ t_{cd} = 25 \text{ ps} \end{cases}$$

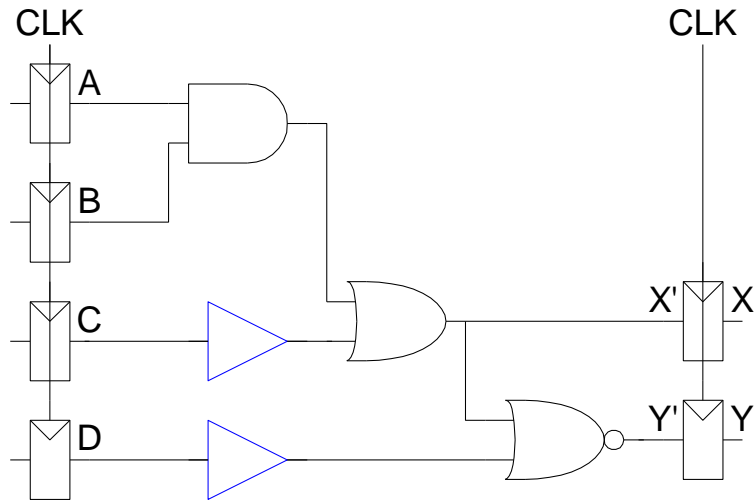
Check hold time constraints:

$$t_{ccq} + t_{cd} > t_{hold} ?$$

$$(30 + 50) \text{ ps} > 70 \text{ ps} ?$$

# Example: Fixing Hold Time Violation

Add buffers to the short paths:



$$t_{pd} = 3 \times 35 \text{ ps} = 105 \text{ ps}$$

$$t_{cd} = 2 \times 25 \text{ ps} = 50 \text{ ps}$$

Check setup time constraints:

$$T_c > t_{pcq} + t_{pd} + t_{setup}$$

$$T_c > (50 + 105 + 60) \text{ ps} = 215 \text{ ps}$$

$$f_c = 1/T_c = 4.65 \text{ GHz}$$

## Timing Characteristics

$$t_{ccq} = 30 \text{ ps}$$

$$t_{pcq} = 50 \text{ ps}$$

$$t_{setup} = 60 \text{ ps}$$

$$t_{hold} = 70 \text{ ps}$$

$$\text{per gate} \left[ \begin{array}{l} t_{pd} = 35 \text{ ps} \\ t_{cd} = 25 \text{ ps} \end{array} \right]$$

Check hold time constraints:

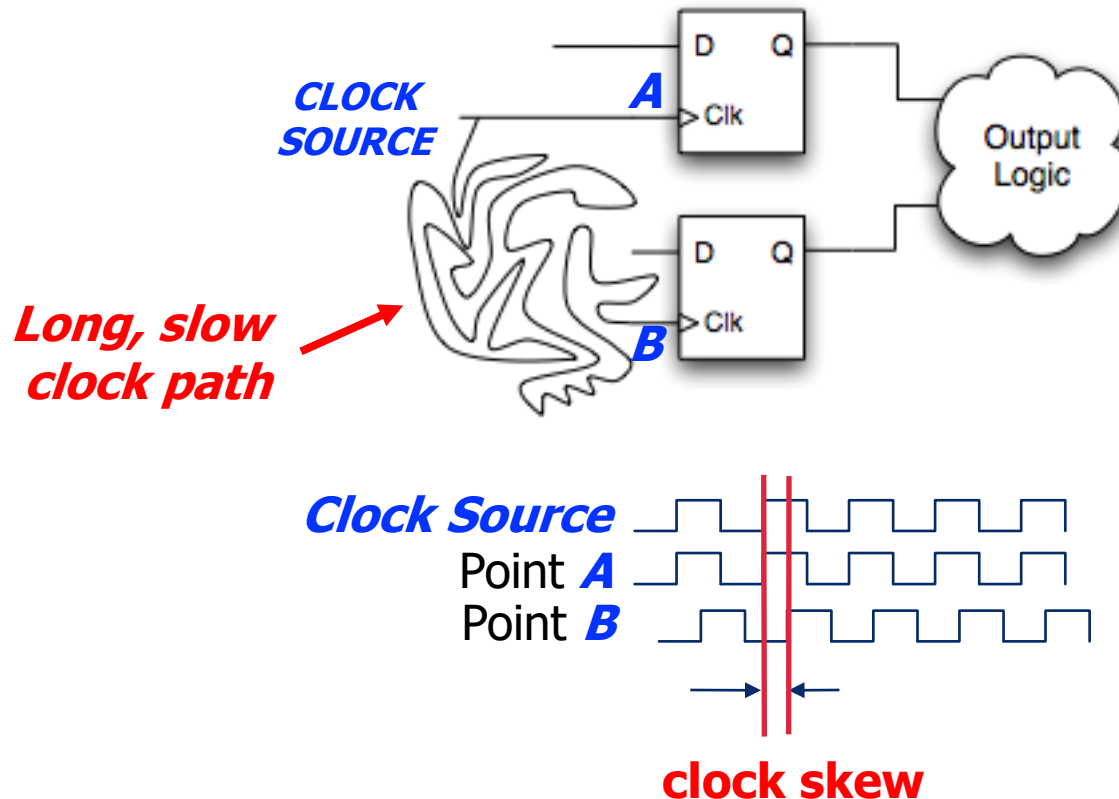
$$t_{ccq} + t_{cd} > t_{hold} ?$$

$$(30 + 50) \text{ ps} > 70 \text{ ps} ?$$

**PASS**

# Clock Skew

- To make matters worse, **clocks have delay** too!
  - The clock does **not** reach all parts of the chip at the same time!
- **Clock skew:** time difference between two clock edges

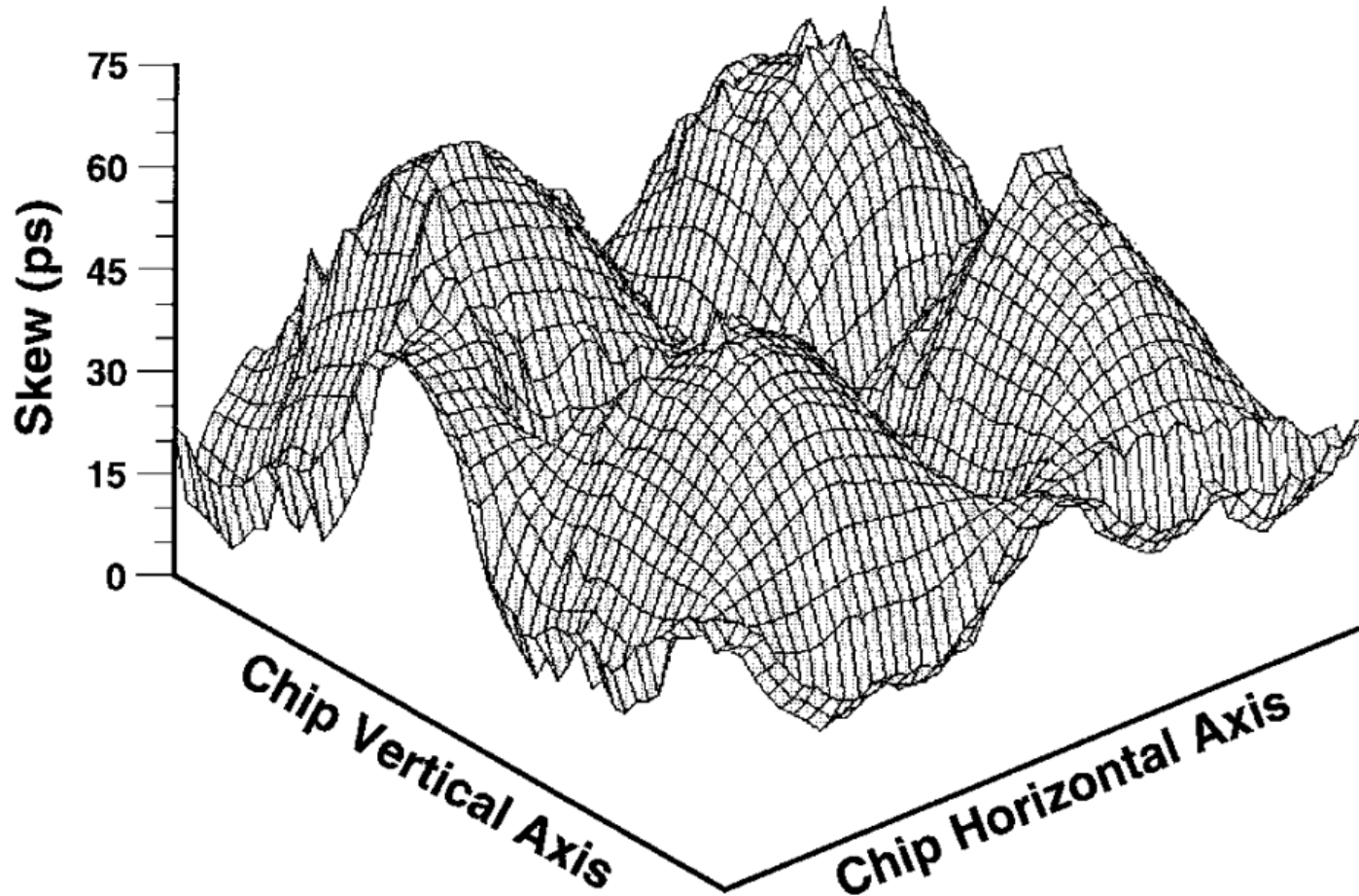




# Clock Skew Example

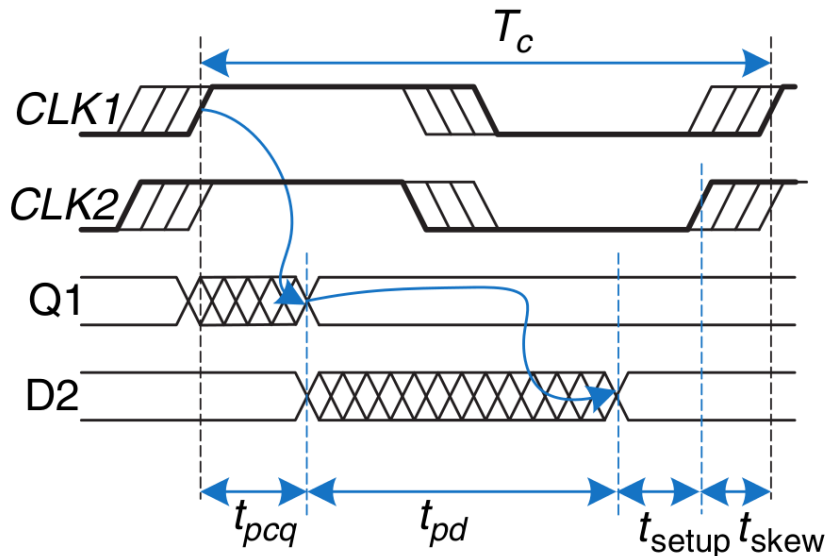
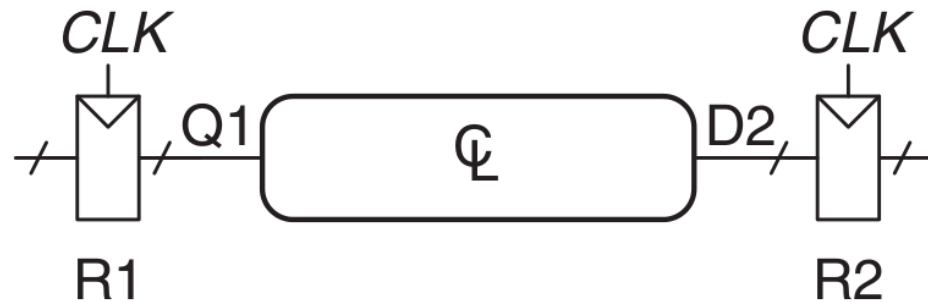
---

- Example of the **Alpha 21264** clock skew spatial distribution



# Clock Skew: Setup Time Revisited

- **Safe timing** requires considering the **worst-case skew**
  - Clock arrives at **R2 before R1**
  - Leaves **as little time as possible** for the **combinational logic**



Signal must arrive at D2 **earlier!**

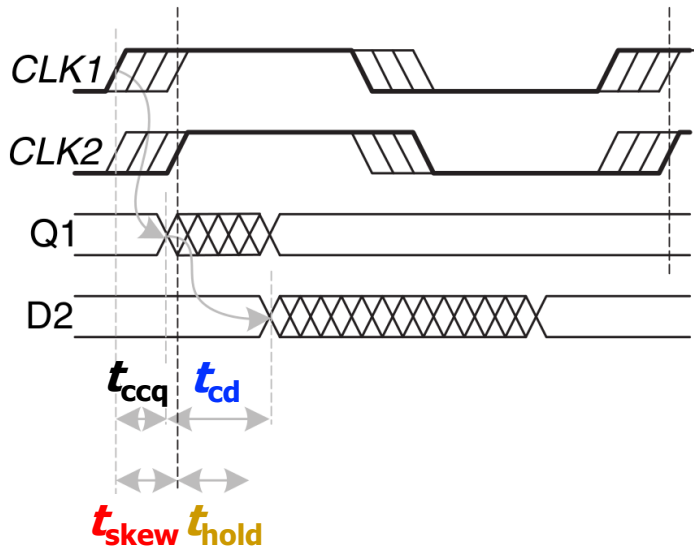
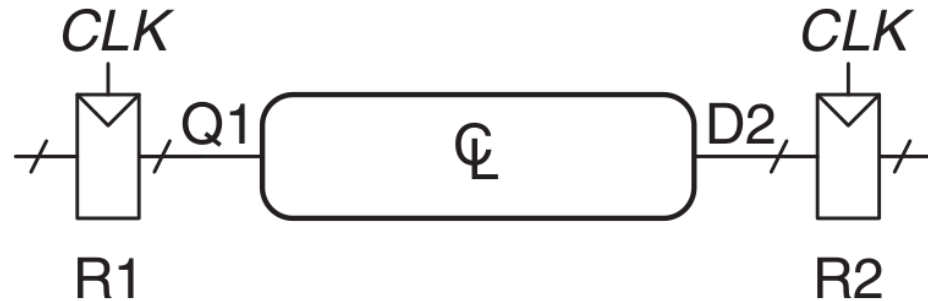
This effectively **increases**  $t_{\text{setup}}$ :

$$T_c > t_{\text{pcq}} + t_{\text{pd}} + t_{\text{setup}} + t_{\text{skew}}$$

$$T_c > t_{\text{pcq}} + t_{\text{pd}} + t_{\text{setup, effective}}$$

# Clock Skew: Hold Time Revisited

- **Safe timing** requires considering the **worst-case skew**
  - Clock arrives at **R2** *after* **R1**
  - Increases the **minimum required delay** for the **combinational logic**



Signal must arrive at D2 *later*!

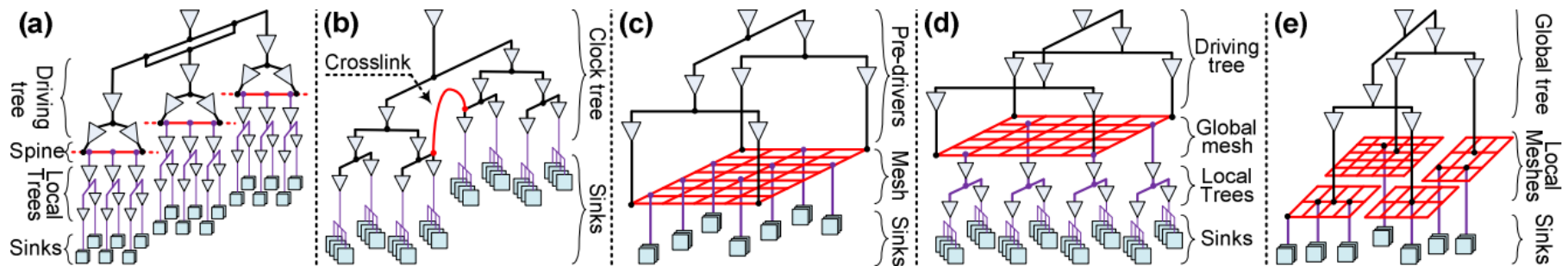
This effectively *increases*  $t_{hold}$ :

$$t_{cd} + t_{ccq} > t_{hold} + t_{skew}$$

$$t_{cd} + t_{ccq} > t_{hold, effective}$$

# Clock Skew: Summary

- **Skew** effectively **increases** both  $t_{\text{setup}}$  and  $t_{\text{hold}}$ 
  - Increased **sequencing overhead**
  - i.e., less useful work done per cycle
- Designers must keep skew to a **minimum**
  - Requires intelligent **"clock network"** across a chip
  - **Goal: clock** arrives at all locations at roughly the **same time**



Source: Abdelhadi, Ameer, et al. "Timing-driven variation-aware nonuniform clock mesh synthesis." GLSVLSI'10.

# Part 3:

## Circuit Verification

# How Do You Know That A Circuit Works?

---

- You have designed a circuit
  - Is it **functionally** correct?
  - Even if it is logically correct, does the hardware meet all **timing** constraints?
- How can you **test** for:
  - Functionality?
  - Timing?
- Answer: **simulation tools!**
  - Formal verification tools (e.g., SAT solvers)
  - HDL timing simulation (e.g., Vivado)
  - Circuit simulation (e.g., SPICE)

# Testing Large Digital Designs

---

- Testing can be the **most time consuming** design stage
  - Functional correctness of **all logic paths**
  - Timing, power, etc. of **all circuit elements**
- Unfortunately, **low-level** (e.g., circuit) simulation is **much slower** than **high-level** (e.g., HDL, C) simulation
- **Solution:** we split responsibilities:
  - 1) Check **only functionality** at a **high level** (e.g., C, HDL)
    - (Relatively) **fast** simulation time allows **high code coverage**
    - **Easy** to write and run tests
  - 2) Check **only timing, power**, etc. at **low level** (e.g., circuit)
    - **No functional testing** of low-level model
    - Instead, test **functional equivalence** to high-level model
      - **Hard**, but **easier** than testing logical functionality at this level

# Testing Large Digital Designs

---

- We have **tools** to handle different levels of verification
  - *Logic synthesis tools* guarantee equivalence of high-level logic and synthesized circuit-level description
  - *Timing verification tools* check all **circuit timings**
  - *Design rule checks* ensure that **physical circuits** are buildable
- **The task of a logic designer** is to:
  - Provide **functional tests** for logical correctness of the design
  - Provide **timing constraints** (e.g., desired operating frequency)
- Tools and/or circuit engineers will decide if it can be built!



# Part 4:

# Functional Verification

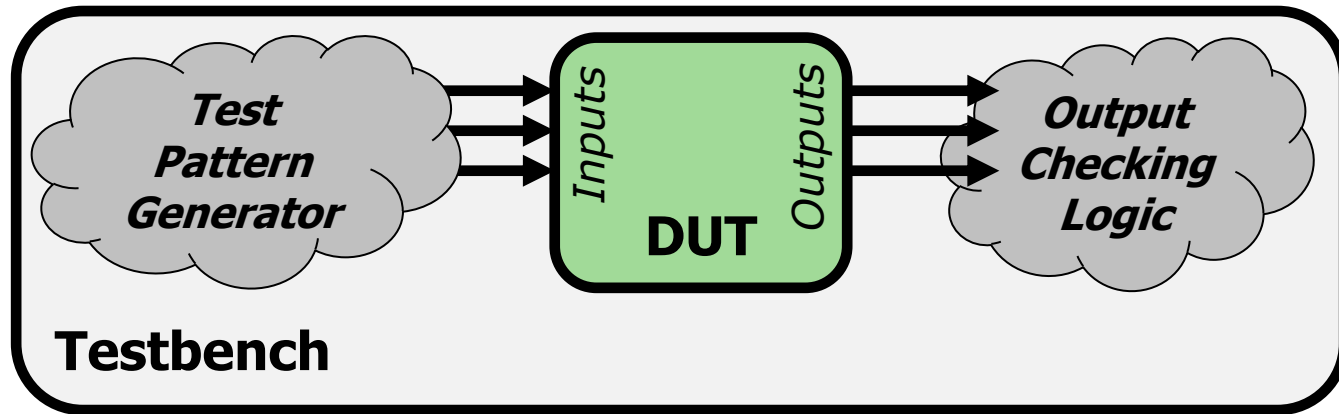
# Functional Verification

---

- Goal: check **logical correctness** of the design
- Physical circuit timing (e.g.,  $t_{\text{setup}}/t_{\text{hold}}$ ) is typically ignored
  - May implement simple checks to catch obvious bugs
  - We'll discuss timing verification later in this lecture
- There are two primary approaches
  - Logic simulation (e.g., C/C++/Verilog test routines)
  - Formal verification techniques
- In this course, we will use Verilog for functional verification

# Testbench-Based Functional Testing

- **Testbench:** a module created specifically to test a design
  - Tested design is called the “**device under test (DUT)**”



- Testbench **provides inputs (test patterns)** to the DUT
  - Hand-crafted values
  - Automatically generated (e.g., sequential or random values)
- Testbench **checks outputs** of the DUT against:
  - Hand-crafted values
  - A “golden design” that is known to be bug-free

# Testbench-Based Functional Testing

---

- A testbench can be:
  - **HDL code** written to test other HDL modules
  - **Circuit schematic** used to test other circuit designs
- The testbench is **not** designed for **hardware synthesis!**
  - Runs in **simulation** only
    - HDL simulator (e.g., Vivado simulator)
    - SPICE circuit simulation
  - Testbench uses **simulation-only** constructs
    - E.g., “wait 10ns”
    - E.g., ideal voltage/current source
    - Not suitable to be physically built!

# Common Verilog Testbench Types

---

Testbench	Input/Output Generation	Error Checking
Simple	Manual	Manual
Self-Checking	Manual	Automatic
Automatic	Automatic	Automatic

# Example DUT

---

- We will walk through different types of testbenches to test a module that implements the logic function:

$$y = (\bar{b} \cdot \bar{c}) + (a \cdot \bar{b})$$

```
// performs y = ~b & ~c | a & ~b
module sillyfunction(input  a, b, c,
                    output y);

    wire b_n, c_n;
    wire m1, m2;

    not not_b(b_n, b);
    not not_c(c_n, c);

    and minterm1(m1, b_n, c_n);
    and minterm2(m2, a, b_n);
    or  out_func(y, m1, m2);

endmodule
```

# Useful Verilog Syntax for Testbenching

---

```
module example_syntax();  
    reg    a;  
  
    // like "always" block, but runs only once at sim start  
    initial  
    begin  
        a = 0; // set value of reg: use blocking assignments  
        #10;    // wait (do nothing) for 10 ns  
        a = 1;  
        $display("printf() style message!"); // print message  
    end  
endmodule
```

# Simple Testbench



# Simple Testbench

---

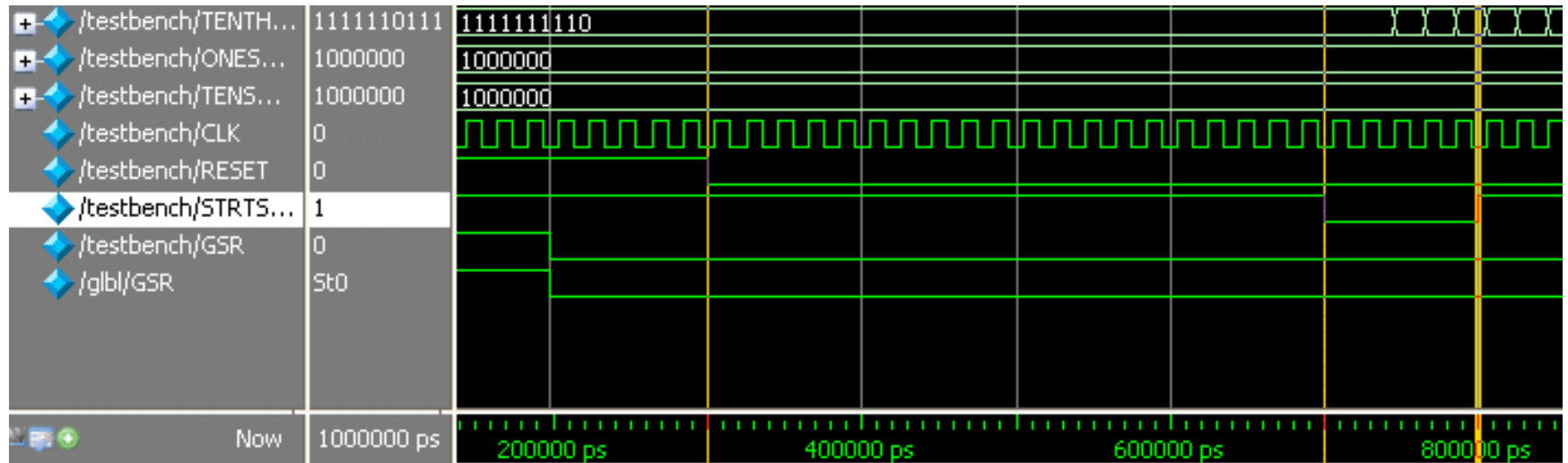
```
module testbench1(); // No inputs, outputs
    reg    a, b, c;      // Manually assigned
    wire  y;            // Manually checked

    // instantiate device under test
    sillyfunction dut (.a(a), .b(b), .c(c), .y(y) );

    // apply hardcoded inputs one at a time
    initial begin
        a = 0; b = 0; c = 0; #10; // apply inputs, wait 10ns
        c = 1; #10;                // apply inputs, wait 10ns
        b = 1; c = 0; #10;         // etc .. etc..
        c = 1; #10;
        a = 1; b = 0; c = 0; #10;
    end
endmodule
```

# Simple Testbench: Output Checking

- Most common method is to look at **waveform diagrams**
  - ❑ ***Thousands*** of signals over ***millions*** of clock cycles
  - ❑ Too many to just printf()!



***time*** →

- **Manually check** that output is correct **at all times**

# Simple Testbench

---

## ■ **Pros:**

- ❑ Easy to design
- ❑ Can easily test a few, specific inputs (e.g., corner cases)

## ■ **Cons:**

- ❑ **Not scalable** to many test cases
- ❑ Outputs must be checked **manually** outside of the simulation
  - E.g., inspecting dumped waveform signals
  - E.g., printf() style debugging

# Self-Checking Testbench

# Self-Checking Testbench

```
module testbench2();  
    reg a, b, c;  
    wire y;  
  
    sillyfunction dut(.a(a), .b(b), .c(c), .y(y));  
  
    initial begin  
        a = 0; b = 0; c = 0; #10; // apply input, wait 10ns  
        if (y !== 1) $display("000 failed."); // check result  
        c = 1; #10;  
        if (y !== 0) $display("001 failed.");  
        b = 1; c = 0; #10;  
        if (y !== 0) $display("010 failed.");  
    end  
endmodule
```

# Self-Checking Testbench

---

## ■ Pros:

- ❑ Still easy to design
- ❑ Still easy to test a few, specific inputs (e.g., corner cases)
- ❑ **Simulator will print** whenever an error occurs

## ■ Cons:

- ❑ **Still not scalable** to millions of test cases
- ❑ Easy to make an **error** in **hardcoded** values
  - You make just as many **errors** writing a testbench as actual code
  - **Hard to debug** whether an issue is in the testbench or in the DUT

# Self-Checking Testbench using Testvectors

---

- Write *testvector file*
  - List of inputs and expected outputs
  - Can create vectors **manually** *or* **automatically** using an already verified, simpler **"golden model"** (more on this later)
- Example file:

```
$ cat testvectors.tv
```

```
000_1
```

```
001_0
```

```
010_0
```

```
011_0
```

```
100_1
```

```
101_1
```

```
110_0
```

```
111_0
```

```
...
```

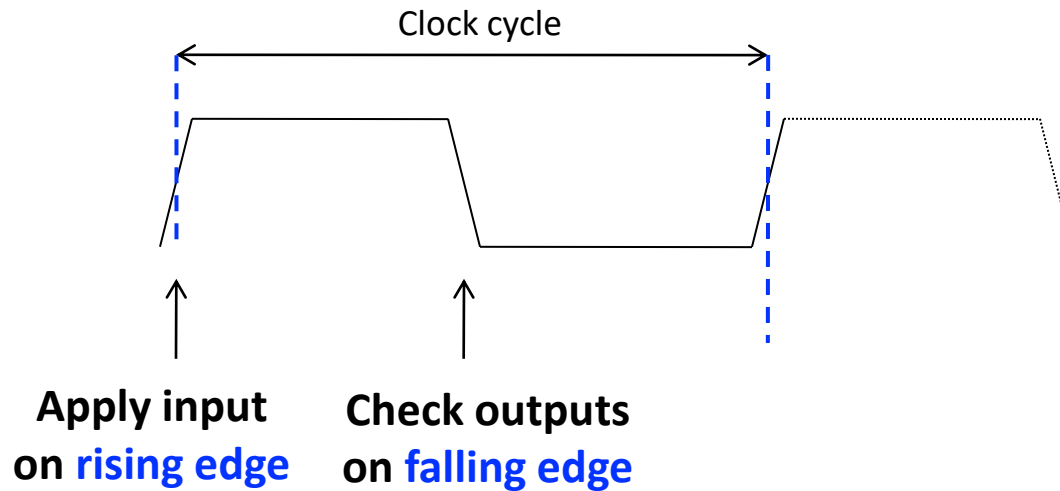
**Format:**  
**input\_output**



# Testbench with Testvectors Design

---

- Use a “**clock signal**” for assigning inputs, reading outputs
  - Test one **testvector** each “**clock cycle**”



- Note: “clock signal” simply separates **inputs** from **outputs**
  - Allows us to *observe* the inputs/outputs in waveform diagrams
  - Not used for checking physical circuit timing (e.g.,  $t_{\text{setup}}$ / $t_{\text{hold}}$ )
  - We’ll discuss *circuit timing verification* later in this lecture



# Testbench Example (1 / 5): Signal Declarations

---

- Declare signals to hold internal state

```
module testbench3();  
    reg          clk, reset;      // clock and reset are internal  
    reg          a, b, c, yexpected; // values from testvectors  
    wire         y;              // output of circuit  
    reg [31:0] vectornum, errors;  // bookkeeping variables  
    reg [3:0]  testvectors[10000:0]; // array of testvectors  
  
    // instantiate device under test  
    sillyfunction dut(.a(a), .b(b), .c(c), .y(y) );
```

**H&H Section 4.9, Example 4.39**

# Testbench Example (2/5): Clock Generation

---

```
// generate clock
always      // no sensitivity list, so it always executes
begin
    clk = 1; #5; clk = 0; #5;      // 10ns period
end
```

# Testbench Example (3/5): Read Testvectors into Array

```
// at start of test, load vectors and pulse reset
initial    // Only executes once
begin
    $readmemb("example.tv", testvectors); // Read vectors
    vectornum = 0; errors = 0;             // Initialize
    reset = 1; #27; reset = 0;            // Apply reset wait
end

// Note: $readmemb reads testvector files written in
// hexadecimal
```

# Testbench Example (4/5): Assign Inputs/Outputs

---

```
// apply test vectors on rising edge of clk
always @(posedge clk)
begin
    {a, b, c, yexpected} = testvectors[vectornum];
end
```

- Apply {a, b, c} inputs on the *rising edge* of the clock
- Get **yexpected** for checking the output on the *falling edge*
- Rising/falling edges are chosen only by convention
  - You can use any part of the clock signal
  - Your H+H textbook uses this convention

# Testbench Example (5/5): Check Outputs

```
always @(negedge clk)
begin
    if (~reset) // don't test during reset
    begin
        if (y !== yexpected)
        begin
            $display("Error: inputs = %b", {a, b, c});
            $display("  outputs = %b (%b exp)", y, yexpected);
            errors = errors + 1;
        end

        // increment array index and read next testvector
        vectornum = vectornum + 1;

        if (testvectors[vectornum] === 4'bx)
        begin
            $display("%d tests completed with %d errors",
                    vectornum, errors);
            $finish;                // End simulation
        end
    end
end
end
```

# Self-Checking Testbench with Testvectors

---

## ■ Pros:

- ❑ Still easy to design
- ❑ Still easy to test a few, specific inputs (e.g., corner cases)
- ❑ Simulator will print whenever an error occurs
- ❑ **No need** to change hardcoded values for **different tests**

## ■ Cons:

- ❑ May be **error-prone** depending on source of testvectors
- ❑ More scalable, but still **limited** by reading a file
  - Might have many more combinational paths to test than will fit in memory

# Automatic Testbench

# Golden Models

---

- A **golden model** represents the ideal circuit behavior
  - ❑ Must be developed, and might be **difficult** to write
  - ❑ Can be done in C, Perl, Python, Matlab or even in Verilog
- For our example circuit:

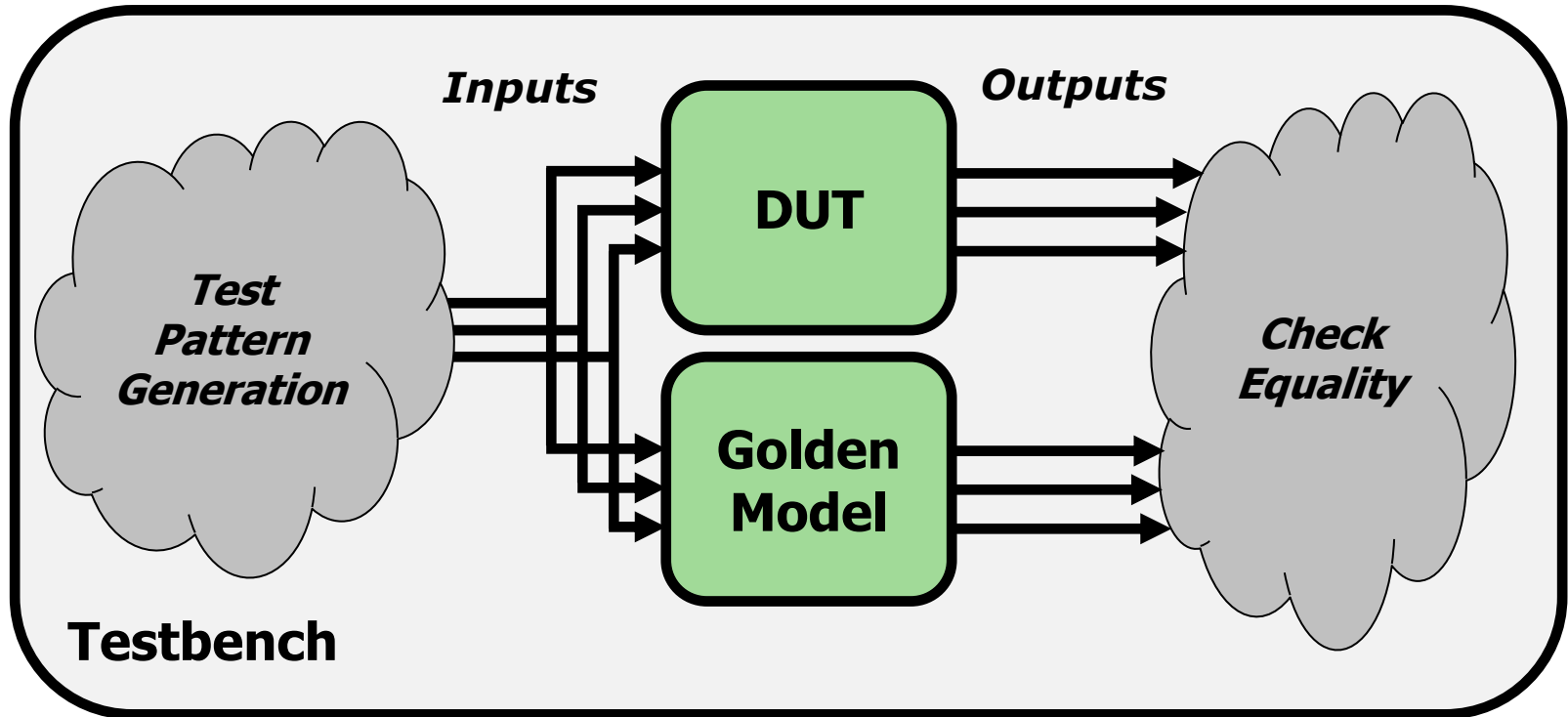
```
module golden_model(input  a, b, c,  
                    output y);  
    assign y = ~b & ~c | a & ~b; // high-level abstraction  
endmodule
```

- **Simpler** than our earlier gate-level description
  - ❑ Golden model is usually **easier to design and understand**
  - ❑ Golden model is much **easier to verify**



# Automatic Testbench

- The DUT **output** is compared against the **golden model**



- **Challenge:** need to **generate inputs** to the designs
  - ❑ Sequential values to cover the entire input space?
  - ❑ Random values?

# Automatic Testbench: Code

```
module testbench1();  
    ... // variable declarations, clock, etc.  
  
    // instantiate device under test  
    sillyfunction dut (a, b, c, y_dut);  
    golden_model gold (a, b, c, y_gold);  
  
    // instantiate test pattern generator  
    test_pattern_generator tgen (a, b, c, clk);  
  
    // check if y_dut is ever not equal to y_gold  
    always @(negedge clk)  
    begin  
        if(y_dut !== y_gold)  
            $display(...)  
    end  
endmodule
```

# Automatic Testbench

---

## ■ Pros:

- ❑ Output checking is **fully automated**
- ❑ Could even compare **timing** using a **golden timing model**
- ❑ **Highly scalable** to as much simulation time as is feasible
  - Leads to **high coverage** of the input space
- ❑ Better **separation of roles**
  - Separate designers can work on the DUT and the golden model
  - DUT testing engineer can focus on **important test cases** instead of output checking

## ■ Cons:

- ❑ Creating a correct golden model may be (very) **difficult**
- ❑ Coming up with **good testing inputs** may be **difficult**

# However, Even with Automatic Testing...

---

- How long would it take to test a **32-bit adder**?
  - In such an adder there are **64** inputs =  $2^{64}$  possible inputs
  - If you test **one input in 1ns**, you can test  $10^9$  inputs per second
    - or  $8.64 \times 10^{14}$  inputs per day
    - or  $3.15 \times 10^{17}$  inputs per year
  - we would still need **58.5 years** to test all possibilities
- Brute force testing is **not feasible** for most circuits!
  - Need to prune the overall testing space
  - E.g., formal verification methods, choosing 'important cases'
- **Verification is a hard problem**

# Part 5:

# Timing Verification

# Timing Verification Approaches

---

- High-level simulation (e.g., C, Verilog)
  - Can **model timing** using “#x” statements in the DUT
  - Useful for hierarchical modeling
    - Insert delays in FF's, basic gates, memories, etc.
    - High level design will have some notion of timing
  - Usually **not as accurate** as real circuit timing
  
- Circuit-level timing verification
  - Need to first **synthesize** your design to actual circuits
    - No one general approach- very **design flow specific**
    - Your FPGA/ASIC/etc. technology has **special tool(s)** for this
      - E.g., Xilinx Vivado (what you're using in lab)
      - E.g., Synopsys/Cadence Tools (for VLSI design)

# The Good News

---

- Tools will try to meet timing for you!
  - Setup times, hold times
  - Clock skews
  - ...
- They usually provide a '**timing report**' or '**timing summary**'
  - **Worst-case** delay paths
  - Maximum operation **frequency**
  - Any timing **errors** that were found

# The Bad News

---

- The **tool can fail** to find a solution
  - Desired clock frequency is too **aggressive**
    - Can result in **setup time violation** on a particularly long path
  - **Too much logic** on clock paths
    - Introduces excessive **clock skew**
  - Timing issues with asynchronous logic
- The tool will provide (hopefully) **helpful errors**
  - Reports will contain paths that failed to meet timing
  - Gives a place from where to start debugging
- **Q:** How can we **fix timing errors**?



# Meeting Timing Constraints

---

- Unfortunately, this is often a **manual, iterative** process
  - Meeting strict timing constraints (e.g., high performance designs) can be **tedious**
- Can try **synthesis/place-and-route** with different options
  - Different **random seeds**
  - Manually provided **hints** for place-and-route
- Can **manually optimize** the reported **problem paths**
  - Simplify **complicated logic**
  - Split up **long combinational logic paths**
  - Recall: fix hold time violations by adding **more** logic!

# Meeting Timing Constraints: Principles

---

- Let's go back to the fundamentals
- Clock cycle time is determined by the maximum logic delay we can accommodate without violating timing constraints
- Good design principles
  - **Critical path design:** Minimize the maximum logic delay  
→ Maximizes performance
  - **Balanced design:** Balance maximum logic delays across different parts of a system (i.e., between different pairs of flip flops)  
→ No bottlenecks + minimizes wasted time
  - **Bread and butter design:** Optimize for the common case, but make sure non-common-cases do not overwhelm the design  
→ Maximizes performance for realistic cases

# Lecture Summary

---

- Timing in **combinational circuits**
  - Propagation delay and contamination delay
  - Glitches
- Timing in **sequential circuits**
  - Setup time and hold time
  - Determining how fast a circuit can operate
- **Circuit Verification**
  - How to make sure a circuit works correctly
  - Functional verification
  - Timing verification

# Digital Design & Computer Arch.

## Lecture 8: Timing and Verification

Prof. Onur Mutlu

ETH Zürich

Spring 2022

18 March 2022