



ÇUKUROVA  
ÜNİVERSİTESİ

Department  
of  
Computer Engineering

# CEN 263

# Digital Design

Autumn 2024

Lecture 5

# Week 5 Outlines

- Sequential building blocks
  - Latches
  - Flip-Flop
  - Counters
- SystemVerilog modelling  
for combinational & sequential building blocks



## Processor (CPU)

- Datapath
- Control Logic
- Pipelining

## Synchronous Digital Systems

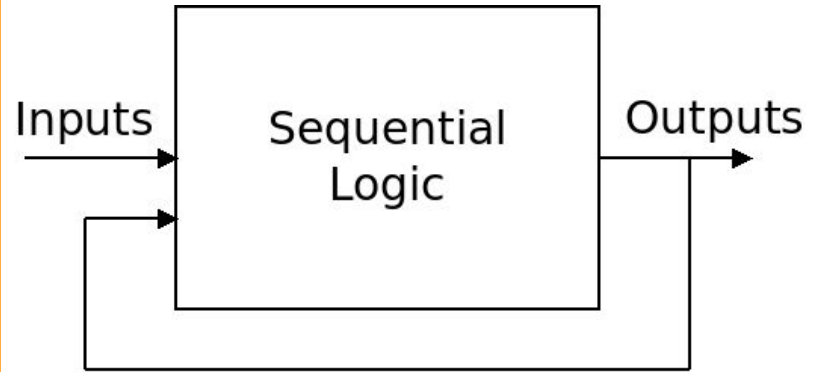
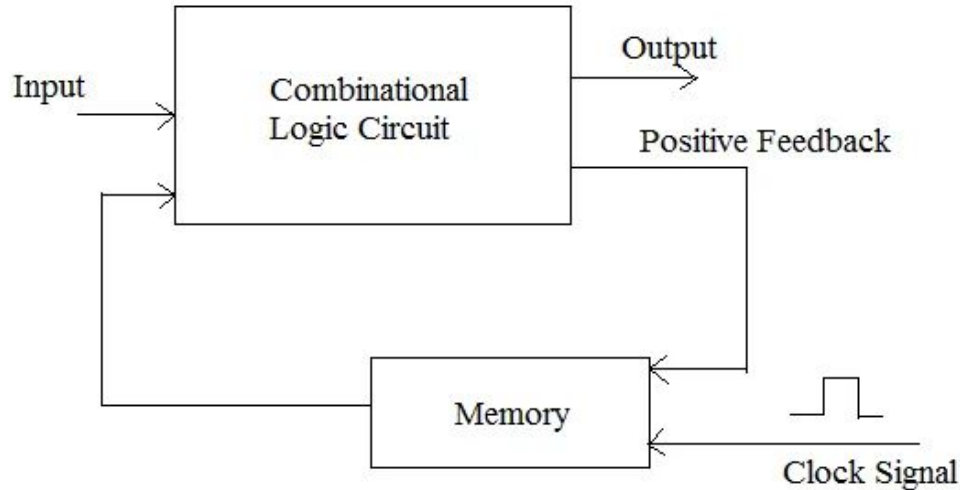
- **Sequential Logic**
- Combinational Logic

## Transistors

# Sequential Logic: Clock and Registers

# Synchronous Digital Systems

The outputs of sequential logic depend on both **current** and **prior** input values.



# Synchronous Digital Systems

Hence, **sequential logic has memory**.

Sequential logic might explicitly remember certain previous inputs, or it might distill the prior inputs into a smaller amount of information called the state of the system.

# Synchronous Digital Systems

The state of a digital sequential circuit is a set of bits called **state** variables that contain all the information about the past necessary to explain the future behavior of the circuit.

# Combinational vs Sequential

Sequential logic uses memory elements to store data, with outputs depending on both current inputs and previous states.

**Contrast with Combinational Logic:** Combinational logic has no memory, output depends only on the current input.

**Applications:** Widely used in digital systems like registers, counters, memory, and processors.



# Asynchronous vs. Synchronous Sequential Logic

## Asynchronous Sequential Logic

- Outputs can change immediately with input changes.
- ***No clock signal is required***; relies on signal changes.
- Faster but more prone to timing issues and glitches.

## Synchronous Sequential Logic

- Outputs change **in response to a clock signal**.
- *State transitions occur only at defined intervals (e.g., clock edges).*
- More predictable and stable, commonly used in modern digital systems.

# Universal Gates

- Nand
- Nor

# Universal Gates - Nand

A	B	C	D	F
0	0	0	0	0
0	0	0	1	0
0	0	1	0	0
0	0	1	1	1
0	1	0	0	0
0	1	0	1	0
0	1	1	0	0
0	1	1	1	1
1	0	0	0	0
1	0	0	1	0
1	0	1	0	0
1	0	1	1	1
1	1	0	0	1
1	1	0	1	1
1	1	1	0	1
1	1	1	1	1

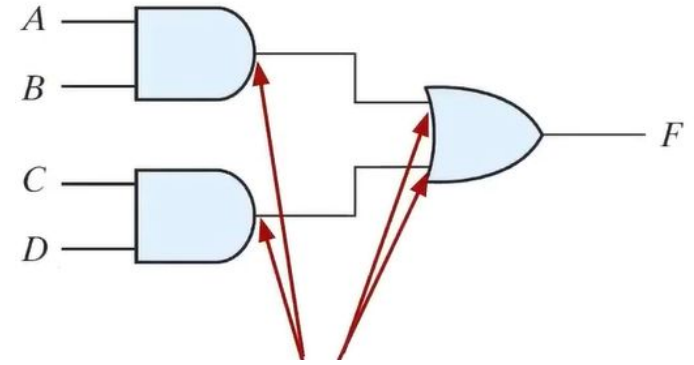
		D			
		00	01	11	10
B	AB	00	01	11	10
	00	0	0	1	0
	01	0	0	1	0
	11	1	1	1	1
	10	0	0	1	0
		C			
		00	01	11	10

# Universal Gates - Nand

A	B	C	D	F
0	0	0	0	0
0	0	0	1	0
0	0	1	0	0
0	0	1	1	1
0	1	0	0	0
0	1	0	1	0
0	1	1	0	0
0	1	1	1	1
1	0	0	0	0
1	0	0	1	0
1	0	1	0	0
1	0	1	1	1
1	1	0	0	1
1	1	0	1	1
1	1	1	0	1
1	1	1	1	1

		D			
		00	01	11	10
B	00	0	0	1	0
	01	0	0	1	0
	11	1	1	1	1
	10	0	0	1	0
		C			
		A			

$$F = AB + CD$$



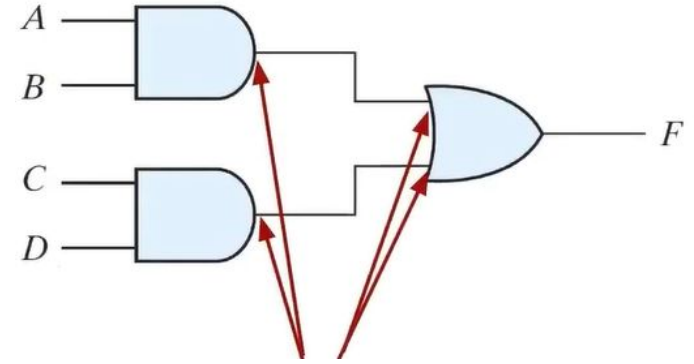
Add bubbles (De Morgan)

# Universal Gates - Nand

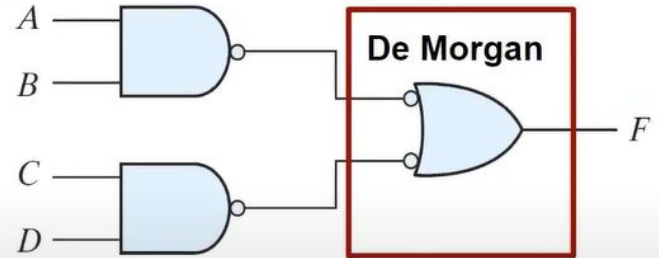
A	B	C	D	F
0	0	0	0	0
0	0	0	1	0
0	0	1	0	0
0	0	1	1	1
0	1	0	0	0
0	1	0	1	0
0	1	1	0	0
0	1	1	1	1
1	0	0	0	0
1	0	0	1	0
1	0	1	0	0
1	0	1	1	1
1	1	0	0	1
1	1	0	1	1
1	1	1	0	1
1	1	1	1	1

		D			
		00	01	11	10
B	AB \ CD	00	01	11	10
	00	0	0	1	0
	01	0	0	1	0
	11	1	1	1	1
	10	0	0	1	0

$$F = AB + CD$$



Add bubbles (De Morgan)

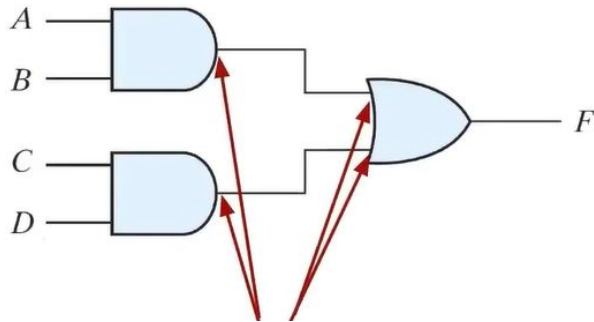


# Universal Gates - Nand

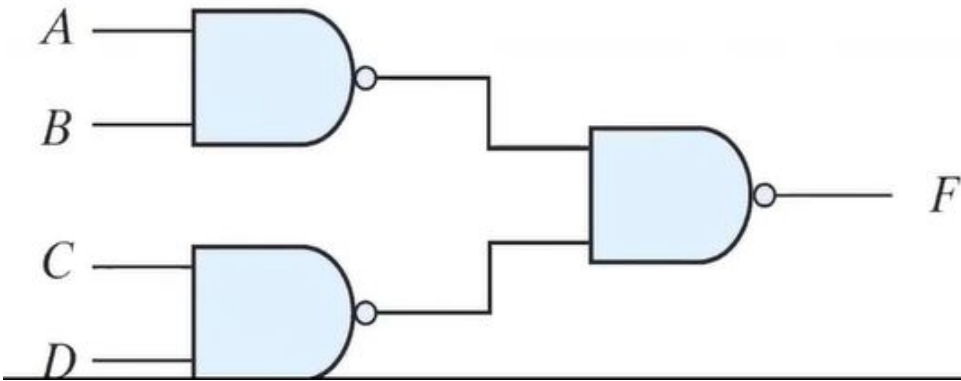
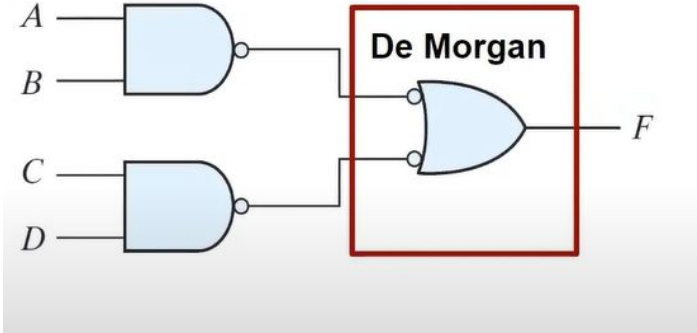
A	B	C	D	F
0	0	0	0	0
0	0	0	1	0
0	0	1	0	0
0	0	1	1	1
0	1	0	0	0
0	1	0	1	0
0	1	1	0	0
0	1	1	1	1
1	0	0	0	0
1	0	0	1	0
1	0	1	0	0
1	0	1	1	1
1	1	0	0	1
1	1	0	1	1
1	1	1	0	1
1	1	1	1	1

AB \ CD		D			
		00	01	11	10
B	00	0	0	1	0
	01	0	0	1	0
	11	1	1	1	1
	10	0	0	1	0

$$F = AB + CD$$

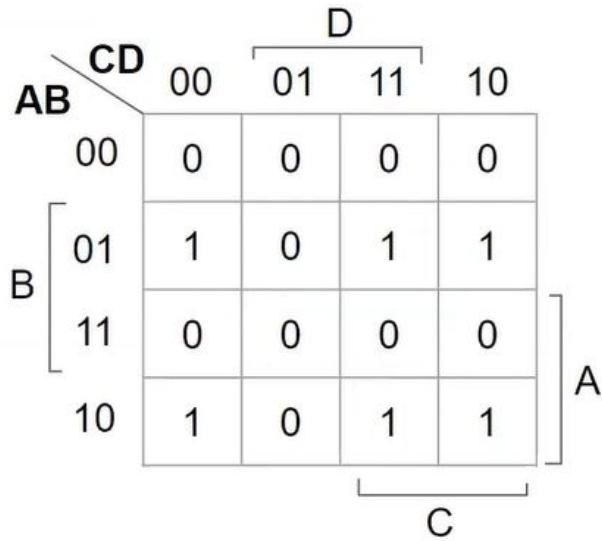


Add bubbles (De Morgan)



# Universal Gates - Nor

A	B	C	D	F
0	0	0	0	0
0	0	0	1	0
0	0	1	0	0
0	0	1	1	0
0	1	0	0	1
0	1	0	1	0
0	1	1	0	1
0	1	1	1	1
1	0	0	0	1
1	0	0	1	0
1	0	1	0	1
1	0	1	1	1
1	1	0	0	0
1	1	0	1	0
1	1	1	0	0
1	1	1	1	0



# Universal Gates - Nor

A	B	C	D	F
0	0	0	0	0
0	0	0	1	0
0	0	1	0	0
0	0	1	1	0
0	1	0	0	1
0	1	0	1	0
0	1	1	0	1
0	1	1	1	1
1	0	0	0	1
1	0	0	1	0
1	0	1	0	1
1	0	1	1	1
1	1	0	0	0
1	1	0	1	0
1	1	1	0	0
1	1	1	1	0

		D			
		00	01	11	10
B	00	0	0	0	0
	01	1	0	1	1
	11	0	0	0	0
	10	1	0	1	1

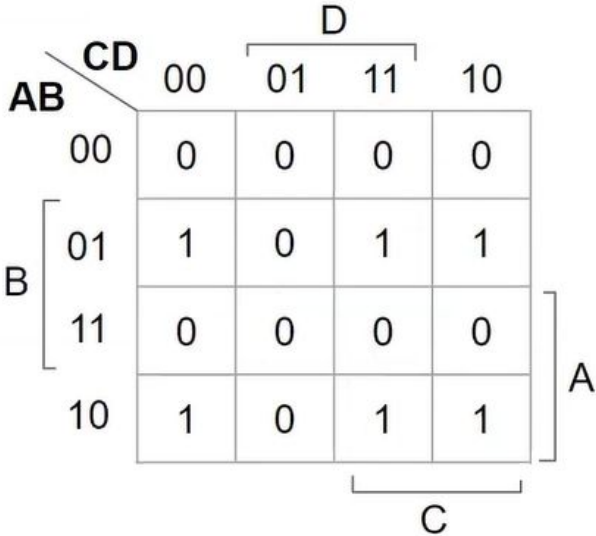
$$F = A'BC + A'BD' + AB'C + AB'D'$$

$$F = (A'B + AB')(C + D')$$



# Universal Gates - Nor

A	B	C	D	F
0	0	0	0	0
0	0	0	1	0
0	0	1	0	0
0	0	1	1	0
0	1	0	0	1
0	1	0	1	0
0	1	1	0	1
0	1	1	1	1
1	0	0	0	1
1	0	0	1	0
1	0	1	0	1
1	0	1	1	1
1	1	0	0	0
1	1	0	1	0
1	1	1	0	0
1	1	1	1	0



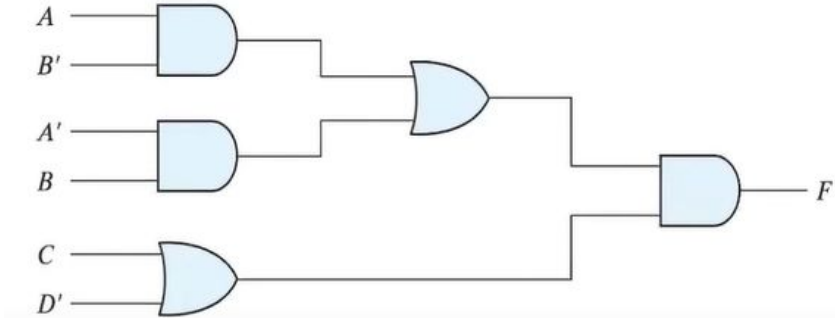
$$F = A'BC + A'BD' + AB'C + AB'D'$$

$$F = (A'B + AB')(C + D')$$

# Universal Gates - Nor

A	B	C	D	F
0	0	0	0	0
0	0	0	1	0
0	0	1	0	0
0	0	1	1	0
0	1	0	0	1
0	1	0	1	0
0	1	1	0	1
0	1	1	1	1
1	0	0	0	1
1	0	0	1	0
1	0	1	0	1
1	0	1	1	1
1	1	0	0	0
1	1	0	1	0
1	1	1	0	0
1	1	1	1	0

		D			
		00	01	11	10
B	AB	00	01	11	10
	00	0	0	0	0
	01	1	0	1	1
	11	0	0	0	0
	10	1	0	1	1

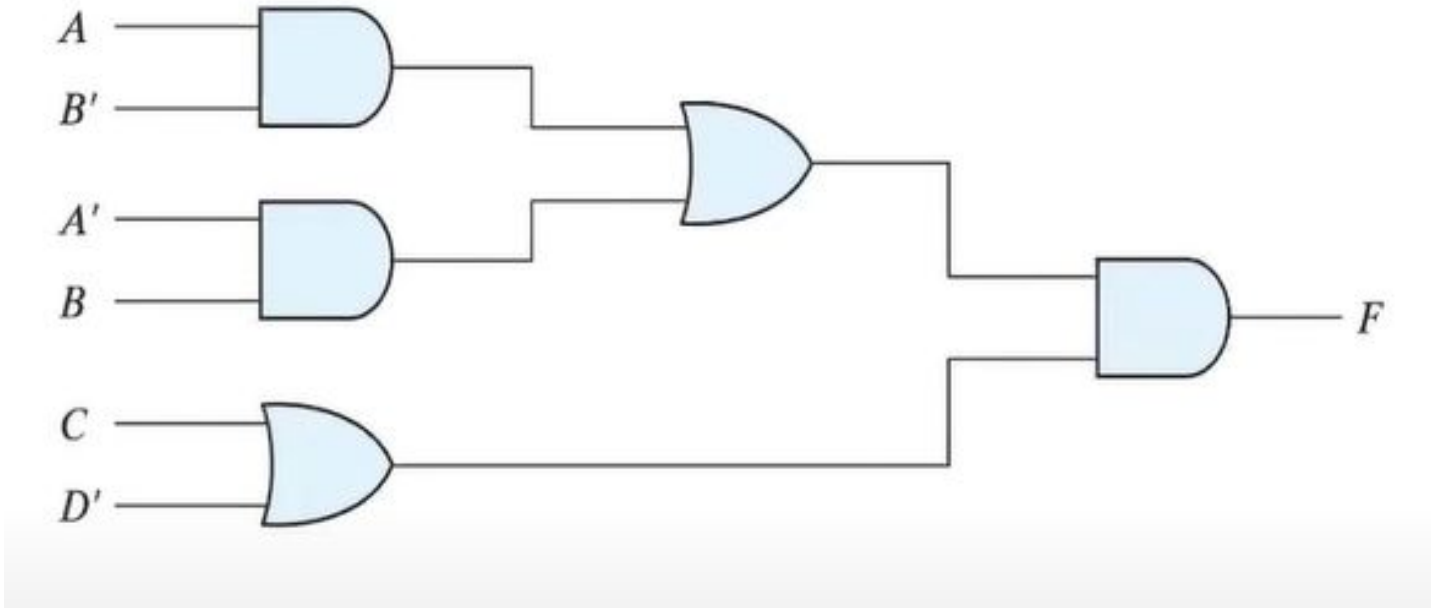


$$F = A'BC + A'BD' + AB'C + AB'D'$$

$$F = (A'B + AB')(C + D')$$

# Universal Gates - Nor - Assignment!

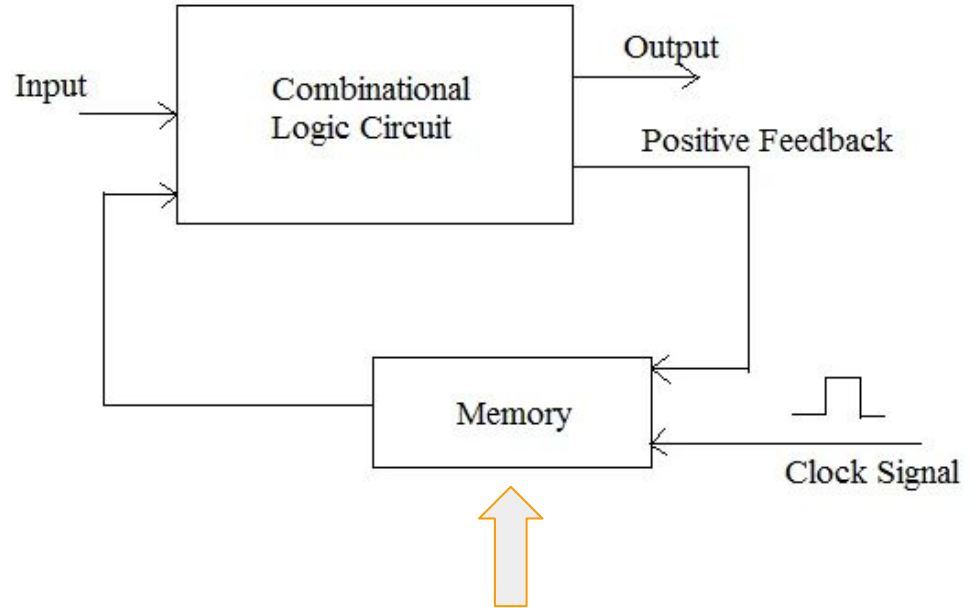
- Implement the following circuit only nor gate?



# Storage Elements: Latches and Flip-Flops

## General Definition:

- **Storage Elements** are devices that "hold" or "store" data in digital circuits.
- They serve as the memory units within sequential logic systems.
- Capable of retaining a bit of data, even as inputs change, until triggered to update.



# Storage Elements: Latches and Flip-Flops

## Latches:

- Level-sensitive devices; they store data based on **input levels (0 or 1), not clock edges**.
- Used in systems where data must be retained while an input signal is active.

## Flip-Flops:

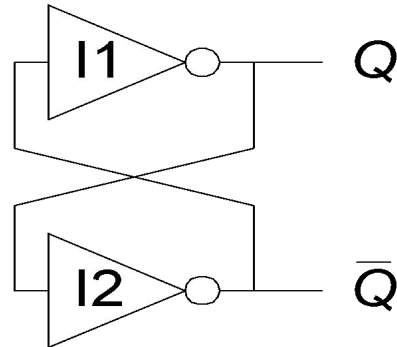
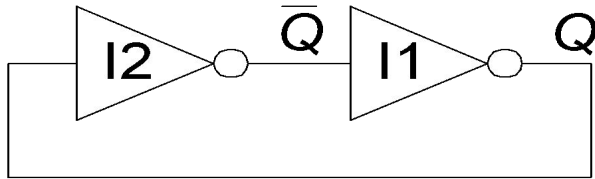
- Edge-triggered devices; **they capture data based on clock edges (either rising or falling)**.
- Preferred for synchronous designs where controlled, predictable data changes are critical.

# Latches: Basics

- **Definition:** Level-sensitive memory element that changes state while enabled.
- **Types:** SR Latch, D Latch
- **Behavior:**
  - **SR Latch:** Set-Reset functionality; can cause invalid states if both inputs are 1.
  - **D Latch:** Data input with enable; resolves SR's invalid state.

# Latches: Basics

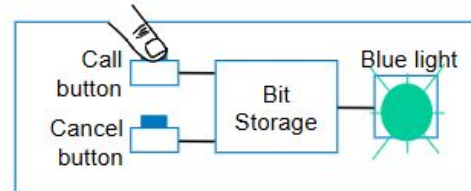
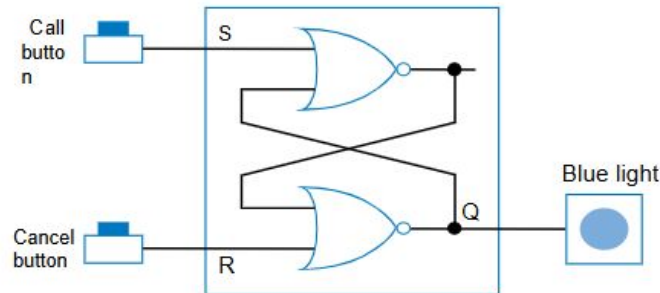
- Fundamental building block of sequential circuits
- Two outputs:  $Q$ ,  $Q'$
- There is a feedback loop!
  - In a typical combinational logic, there is no feedback loop.



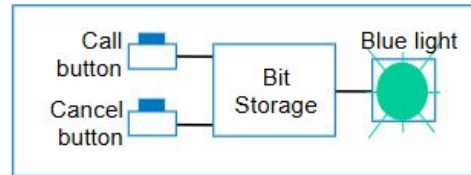
# Latches: Basics

- Flight attendant call button
  - Press call: light turns on
    - *Stays on* after button released
  - Press cancel: light turns off
  - Logic gate circuit to implement this?

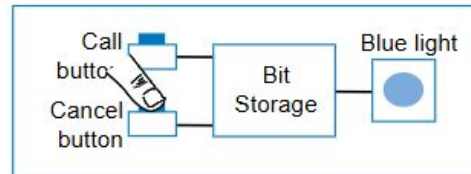
- SR latch implementation
  - Call=1 : sets Q to 1 and keeps it at 1
  - Cancel=1 : resets Q to 0



1. Call button pressed – light turns on



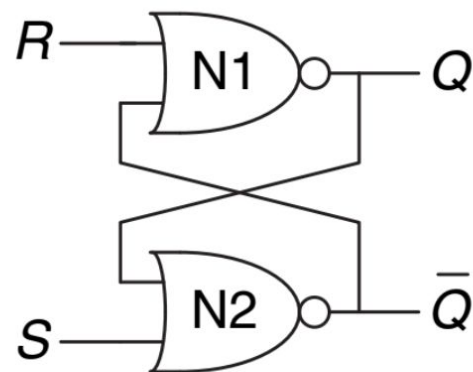
2. Call button released – light *stays on*



3. Cancel button pressed – light turns off



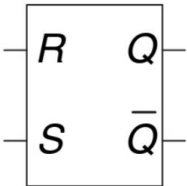
# SR Latch with NOR gates



**Figure 3.3** SR latch schematic

Case	$S$	$R$	$Q$	$\bar{Q}$
IV	0	0	$Q_{prev}$	$\bar{Q}_{prev}$
I	0	1	0	1
II	1	0	1	0
III	1	1	0	0

**Figure 3.5** SR latch truth table



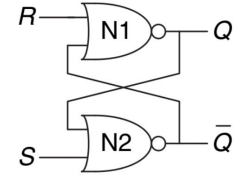
**Figure 3.6** SR latch symbol

# SR Latch with NOR gates

Truth Table for SR Latch with NOR Gates:

S (Set)	R (Reset)	Q (Output)	Q' (Complement)	Description
0	0	Previous Q	Previous Q'	No Change
0	1	0	1	Reset (Q = 0)
1	0	1	0	Set (Q = 1)
1	1	Invalid	Invalid	Invalid (not allowed)

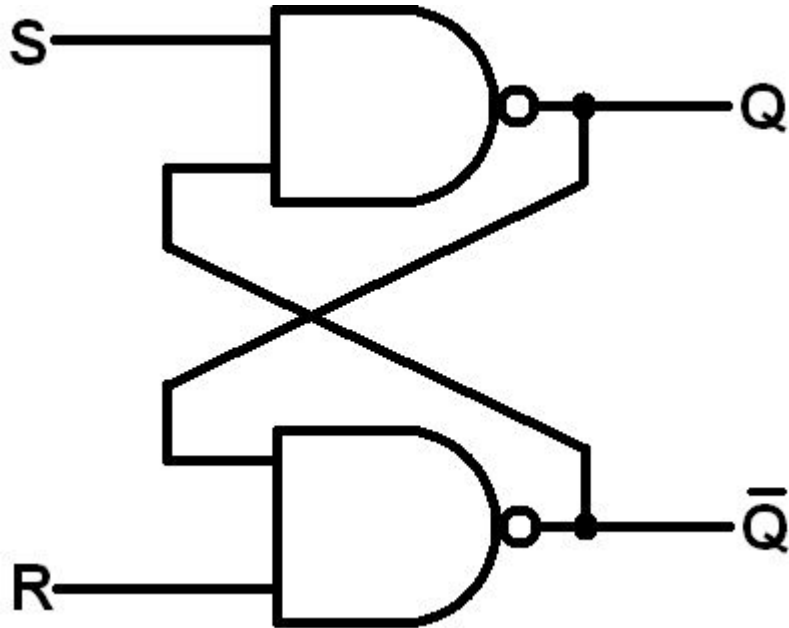
- **Explanation:**
  - When  $S = 1$  and  $R = 0$ , Q is set to 1 (Set state).
  - When  $S = 0$  and  $R = 1$ , Q is reset to 0 (Reset state).
  - $S = 0$  and  $R = 0$  holds the current state (no change).
  - $S = 1$  and  $R = 1$  creates an invalid state, as it causes both Q and Q' to be 0, violating complementarity.



**Figure 3.3** SR latch schematic

## SR Latch with NAND gates

p.s. Nand/Nor gates are the universal gates.

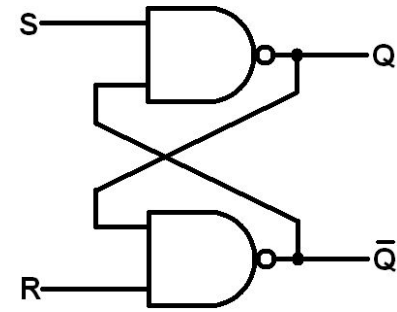


# SR Latch with NAND gates

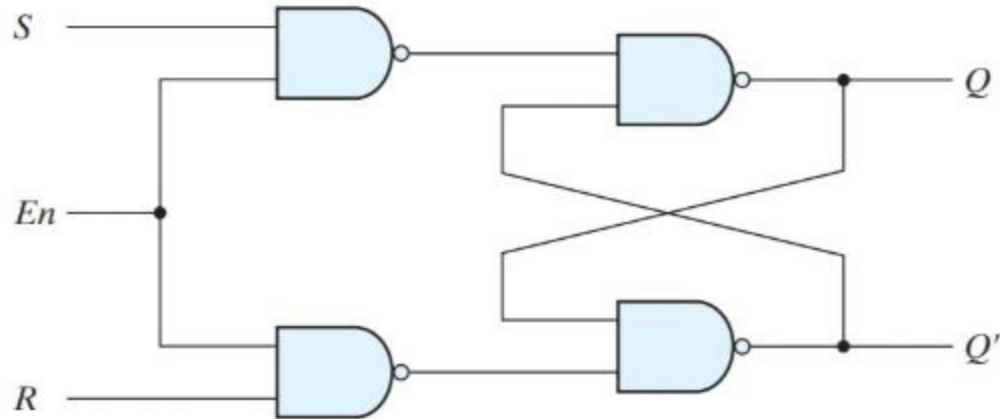
Truth Table for SR Latch with NAND Gates:

S (Set)	R (Reset)	Q (Output)	Q' (Complement)	Description
1	1	Previous Q	Previous Q'	No Change
1	0	1	0	Reset (Q = 0)
0	1	0	1	Set (Q = 1)
0	0	Invalid	Invalid	Invalid (not allowed)

- **Explanation:**
  - When  $S = 0$  and  $R = 1$ , Q is set to 1 (Set state).
  - When  $S = 1$  and  $R = 0$ , Q is reset to 0 (Reset state).
  - $S = 1$  and  $R = 1$  holds the current state (no change).
  - $S = 0$  and  $R = 0$  creates an invalid state, as it forces both Q and Q' to be 1, which contradicts the complement relationship.



# SR Latch with NAND gates with Enable (E)



(a) Logic diagram

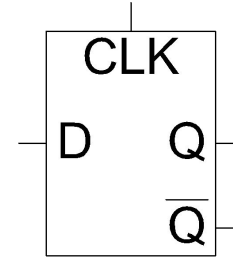
$En$	$S$	$R$	Next state of $Q$
0	X	X	No change
1	0	0	No change
1	0	1	$Q = 0$ ; reset state
1	1	0	$Q = 1$ ; set state
1	1	1	Indeterminate

(b) Function table

# D Latch

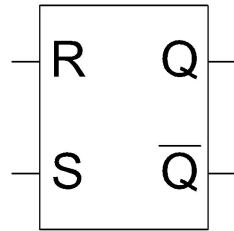
- Two inputs:  $CLK$ ,  $D$ 
  - $CLK$ : controls *when* the output changes
  - $D$  (the data input): controls *what* the output changes to
- Function
  - When  $CLK = 1$ ,  $D$  passes through to  $Q$  (the latch is *transparent*)
  - When  $CLK = 0$ ,  $Q$  holds its previous value (the latch is *opaque*)
- Avoids invalid case when  $Q \neq \text{NOT } Q$

D Latch  
Symbol

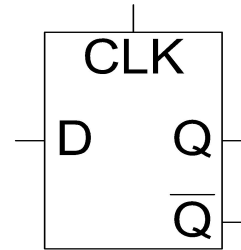


# D Latch Internal Circuit

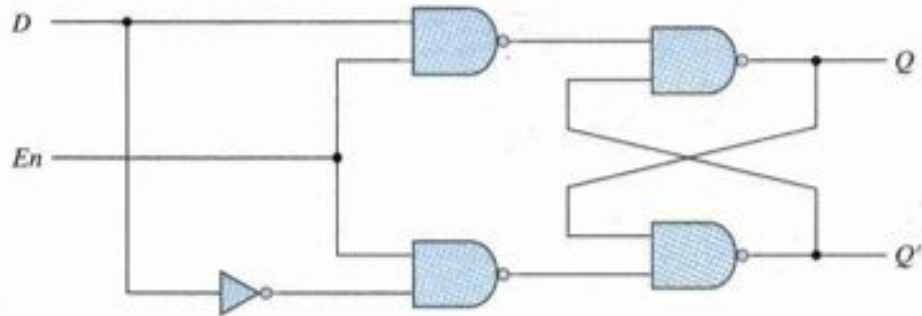
SR Latch  
Symbol



D Latch Symbol



# D Latch with NAND gates with Enable (E)



(a) Logic diagram

$En$	$D$	Next state of $Q$
0	X	No change
1	0	$Q = 0$ ; reset state
1	1	$Q = 1$ ; set state

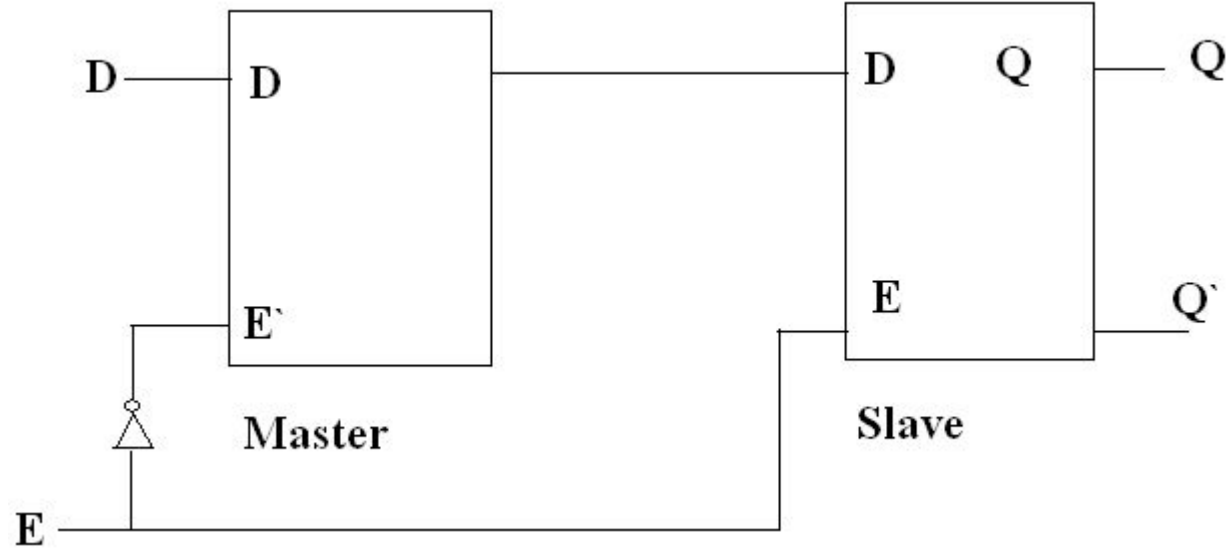
(b) Function table

**FIGURE 5.6**  
D latch



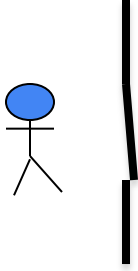
# D Latch with NAND gates with Enable (E)

POSEDGE CLK D\_FF

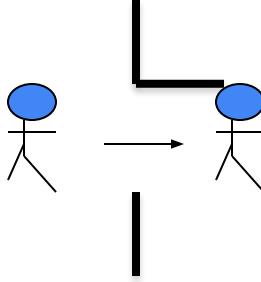


# Latch and Flip-flop (two latches)

A latch can be considered as a door

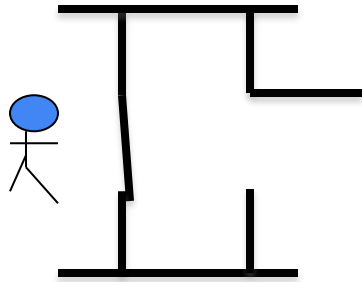


CLK = 0, door is shut

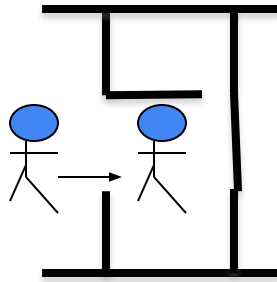


CLK = 1, door is unlocked

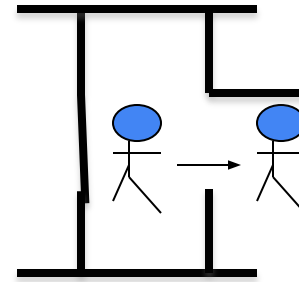
A flip-flop is a two door entrance



CLK = 1



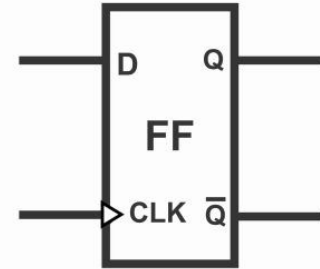
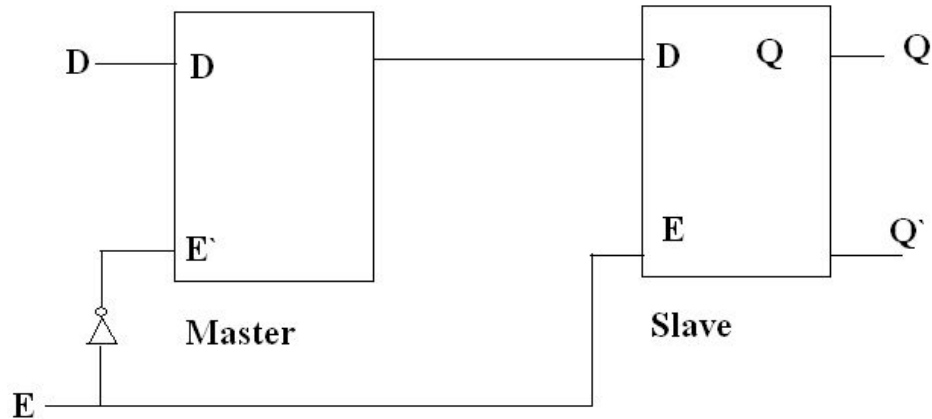
CLK = 0



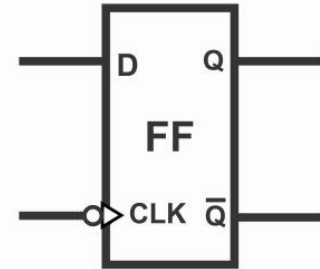
CLK = 1

# D Latch with NAND gates with Enable (E)

POSEDGE and NEGEDGE CLK



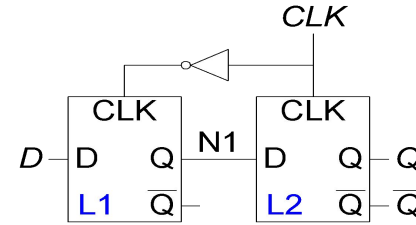
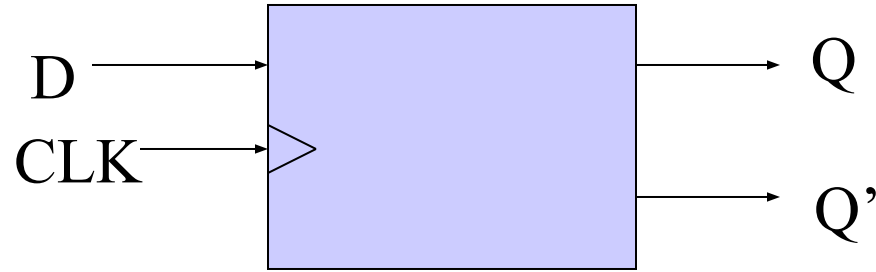
(a)



(b)

**Fig 5.17 (a) Logic symbol for positive-edge-triggered D flip-flop .(b) Logic symbol for negative-edge-triggered D flip-flop**

# D Flip-Flop (Delay)



Id	D	Q(t)	Q(t+1)
0	0	0	0
1	0	1	0
2	1	0	1
3	1	1	1

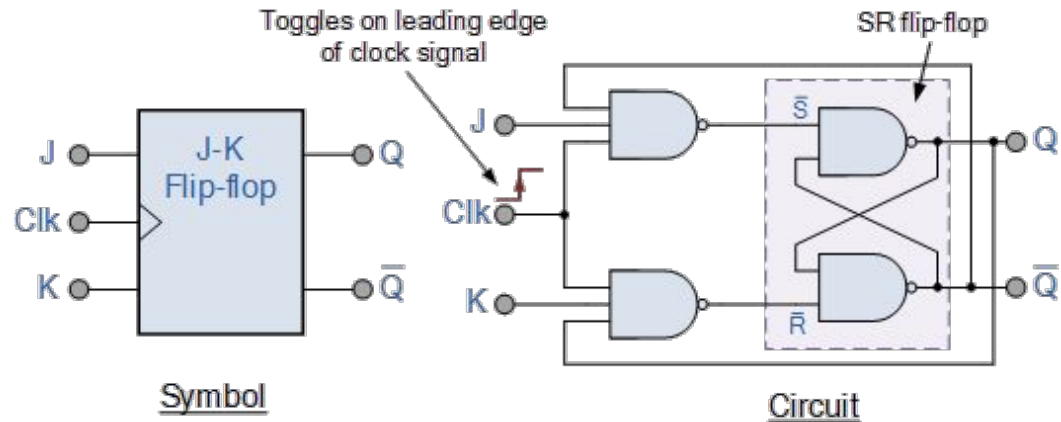
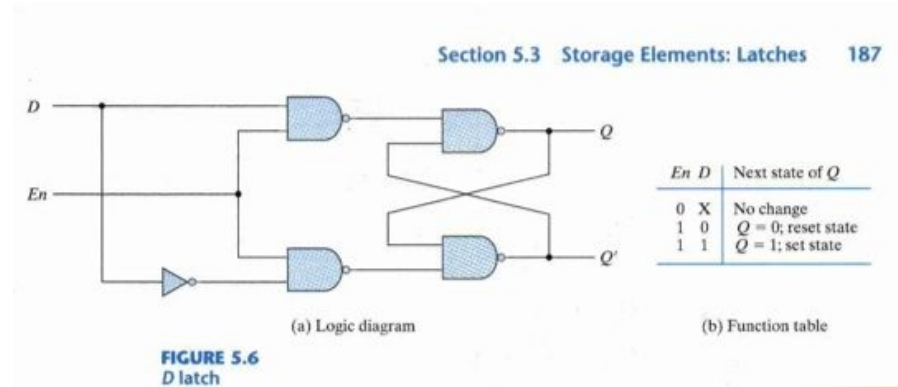
State table

PS \ D	0	1
0	0	1
1	0	1

NS = Q(t+1)

Characteristic Expression:  $Q(t+1) = D(t)$

# JK FF

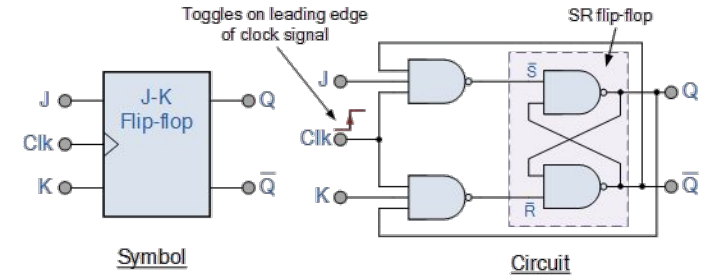


# JK FF

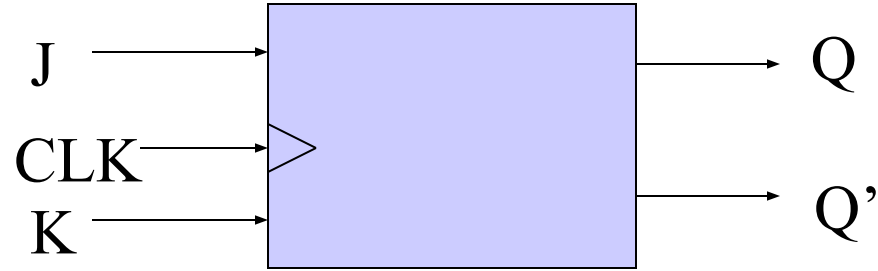
Clock Edge	J	K	Q (Previous)	Q (Next)	Description
Rising	0	0	0	0	No Change
Rising	0	0	1	1	No Change
Rising	0	1	0	0	Reset (0)
Rising	0	1	1	0	Reset (0)
Rising	1	0	0	1	Set (1)
Rising	1	0	1	1	Set (1)
Rising	1	1	0	1	Toggle (Q = 1)
Rising	1	1	1	0	Toggle (Q = 0)

## Explanation

- $J = 0, K = 0$ : No change. The output  $Q$  remains in its previous state.
- $J = 0, K = 1$ : Reset. The output  $Q$  is set to 0.
- $J = 1, K = 0$ : Set. The output  $Q$  is set to 1.
- $J = 1, K = 1$ : Toggle. The output  $Q$  switches to the opposite of its previous state (from 0 to 1, or from 1 to 0).



# JK F-F

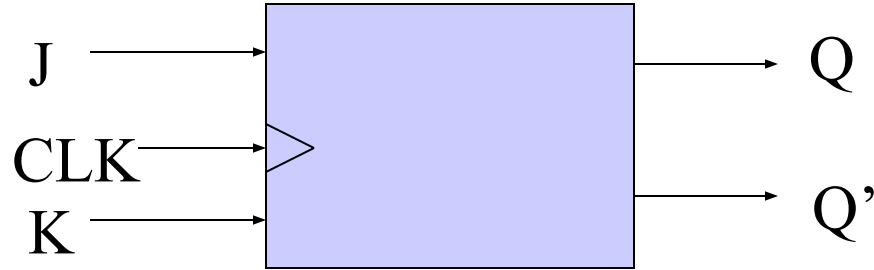


State table

PS \ JK	00	01	10	11
0	0	0	1	?
1	1	0	1	?

↗  
 $Q(t+1)$

# JK F-F



State table

PS \ JK	00	01	10	11
0	0	0	1	1
1	1	0	1	0

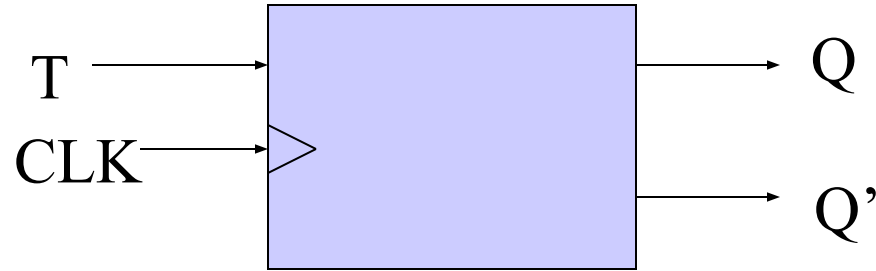
Arrows point from the text "Q(t+1)" and "Toggle" to the cells in the state table. The "Q(t+1)" arrow points to the cell (JK=01, PS=1) which contains 0. The "Toggle" arrow points to the cell (JK=11, PS=1) which contains 0.

Characteristic Expression

$$Q(t+1) = Q(t)K'(t) + Q'(t)J(t)$$



# T Flip-Flop (Toggle)



State table

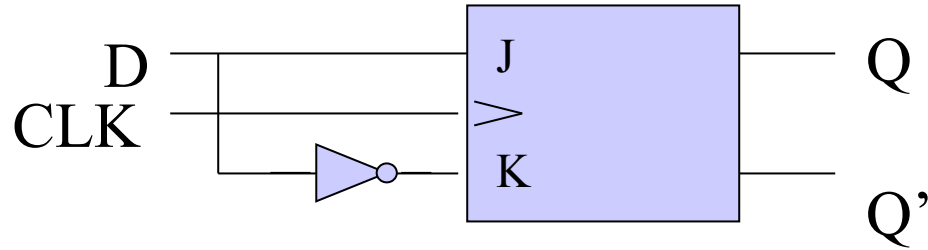
PS \ T	0	1
0	0	1
1	1	0

←  $Q(t+1)$

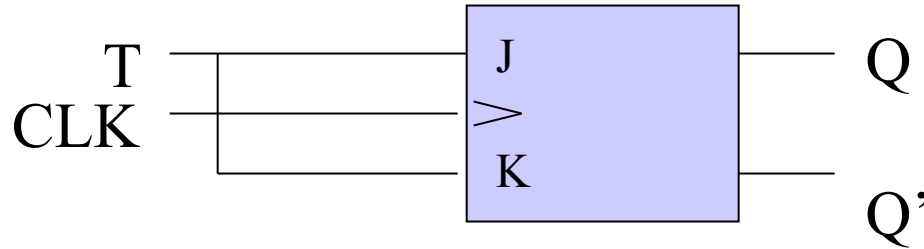
Characteristic Expression

$$Q(t+1) = Q'(t)T(t) + Q(t)T'(t)$$

# Using a JK F-F to implement a D and T F-F

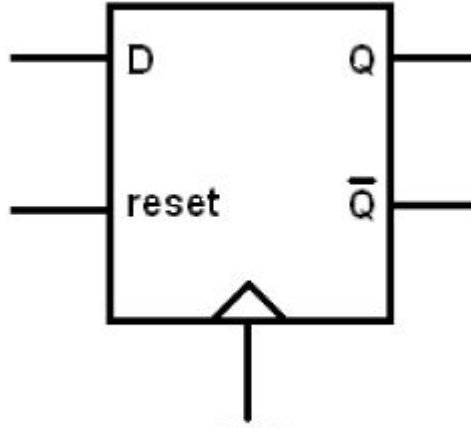


D flip flop



T flip flop

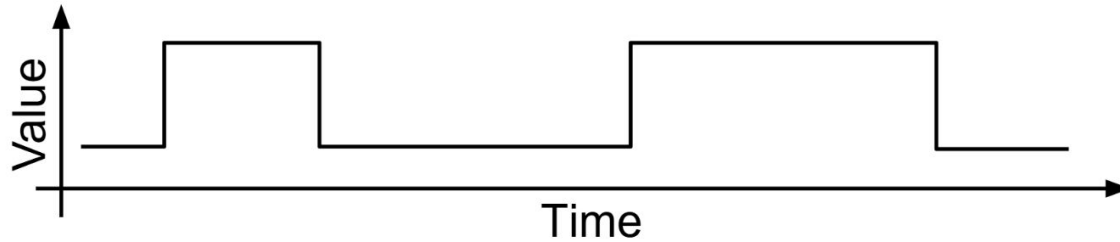
## D FF async reset



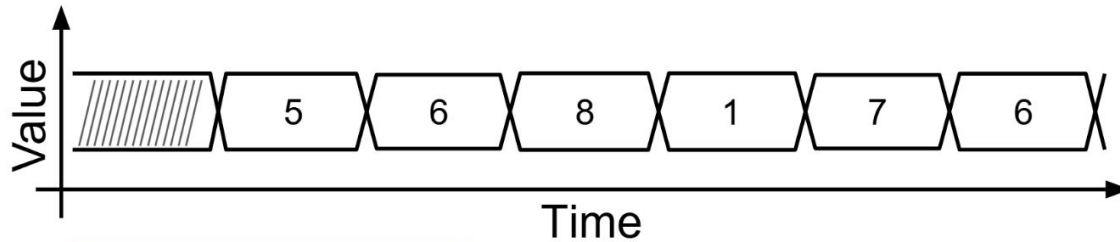
Truth Table for D Flip-Flop with Asynchronous Reset

Reset	Clock Edge	D	Q (Next)	Description
1	-	X	0	Asynchronous Reset ( $Q = 0$ )
0	Rising	0	0	Load 0 on Rising Edge
0	Rising	1	1	Load 1 on Rising Edge
0	-	X	Q (Prev)	Hold (No Clock Edge)

- Graph of a signal's value over time



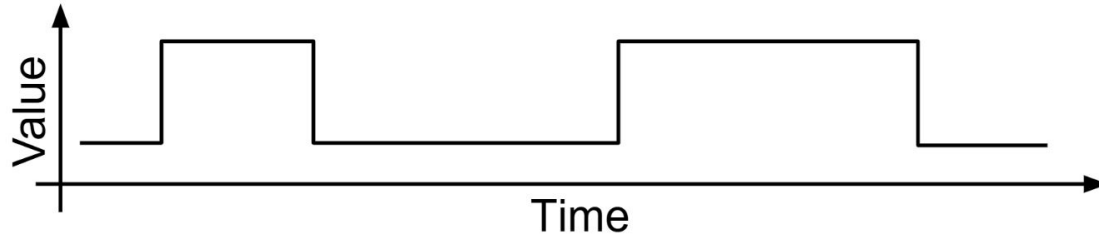
For 1-bit signals, high=1, low=0.



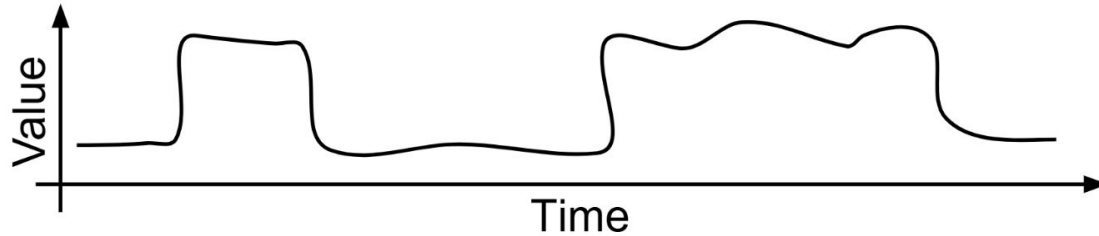
Grayed out = unknown or garbage value.

For  $n$ -bit signals, we can write the value on the waveform.

- Graph of a signal's value over time

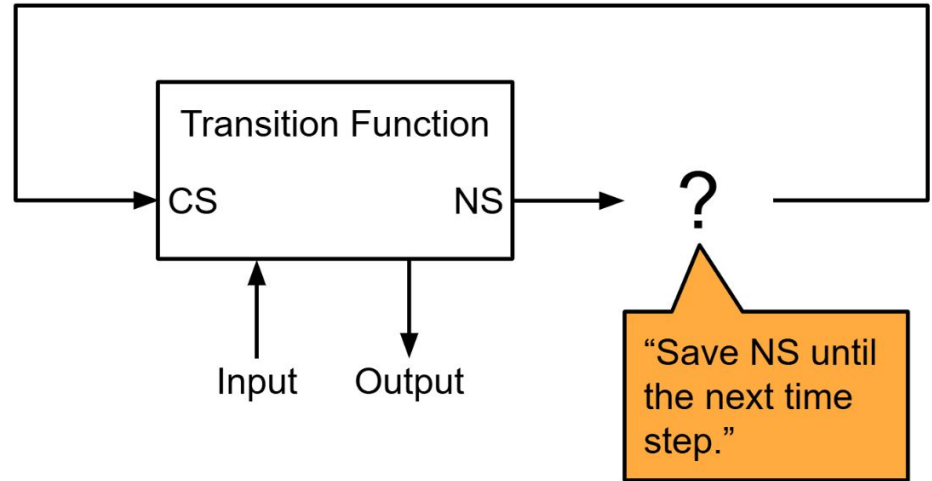


An ideal signal  
looks like this, but...



...in reality, signals  
are noisy!

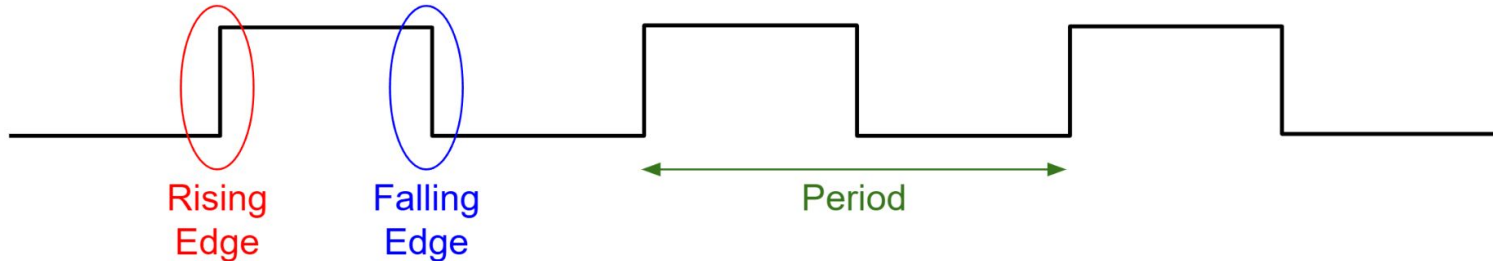
How do we differentiate  
between timesteps?  
How do we “save” which  
state we’re in?



# Synchronous Digital Systems

## Clock

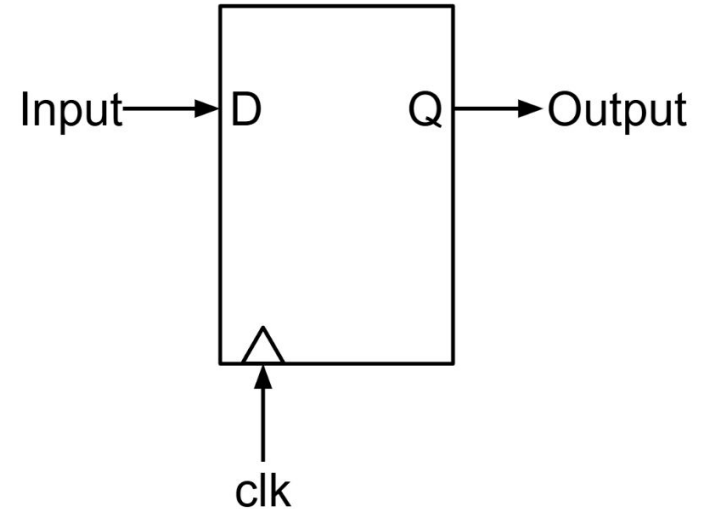
- Clock: Signal that alternates between 0 and 1
- Rising edge: Time when the clock switches from 0 to 1
- Falling edge: Time when the clock switches from 1 to 0
- Clock period: Time between rising edges
- Clock frequency: Number of rising edges per second
  - $\text{Frequency} = 1 / \text{period}$



# Synchronous Digital Systems

## Flip-Flop

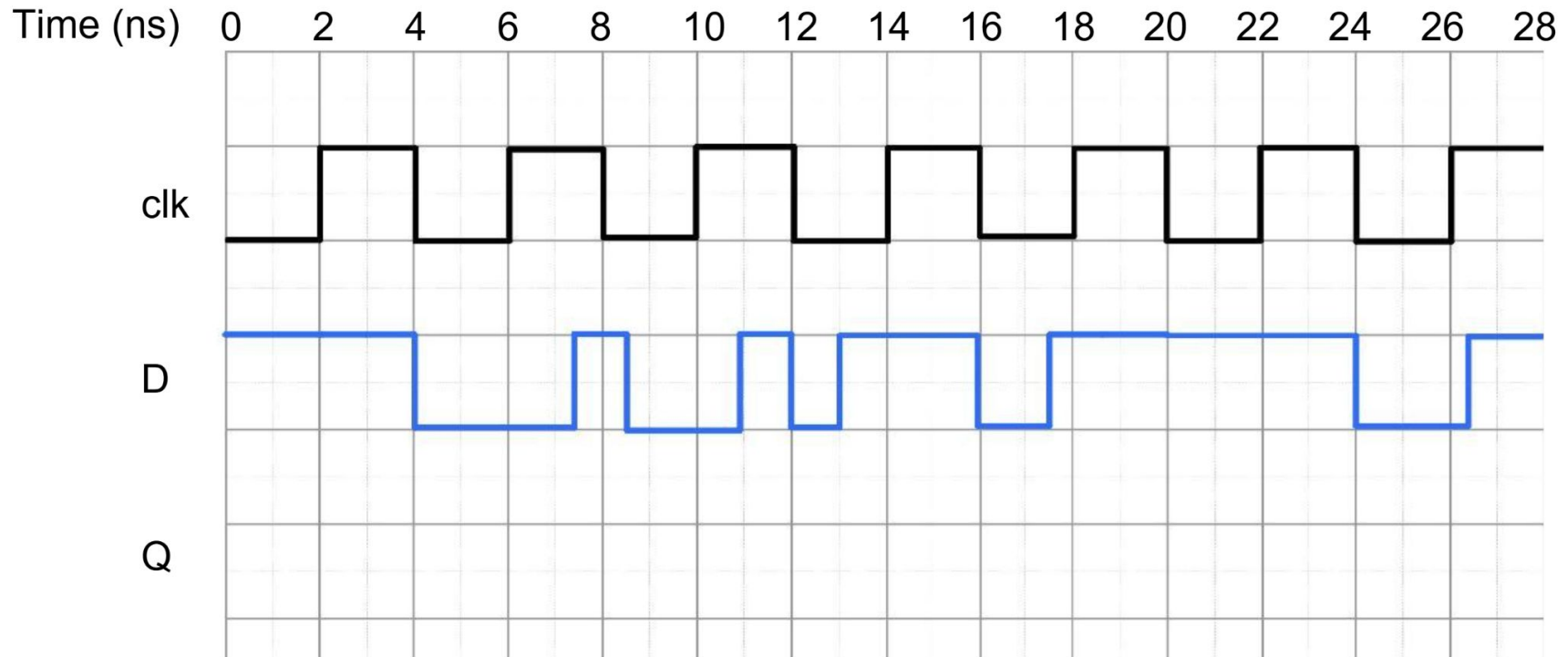
- Inputs:
  - One-bit value D
  - One-bit clock value (often drawn as triangle)
- Outputs:
  - One-bit value Q
- Behavior:
  - On the rising edge of the clock, set  $Q=D$
  - At all other times, do nothing
- Usage:
  - Store data: Between rising edges, the output value at Q will stay steady
  - Control the flow of data: Hold data at the D input until the next rising edge





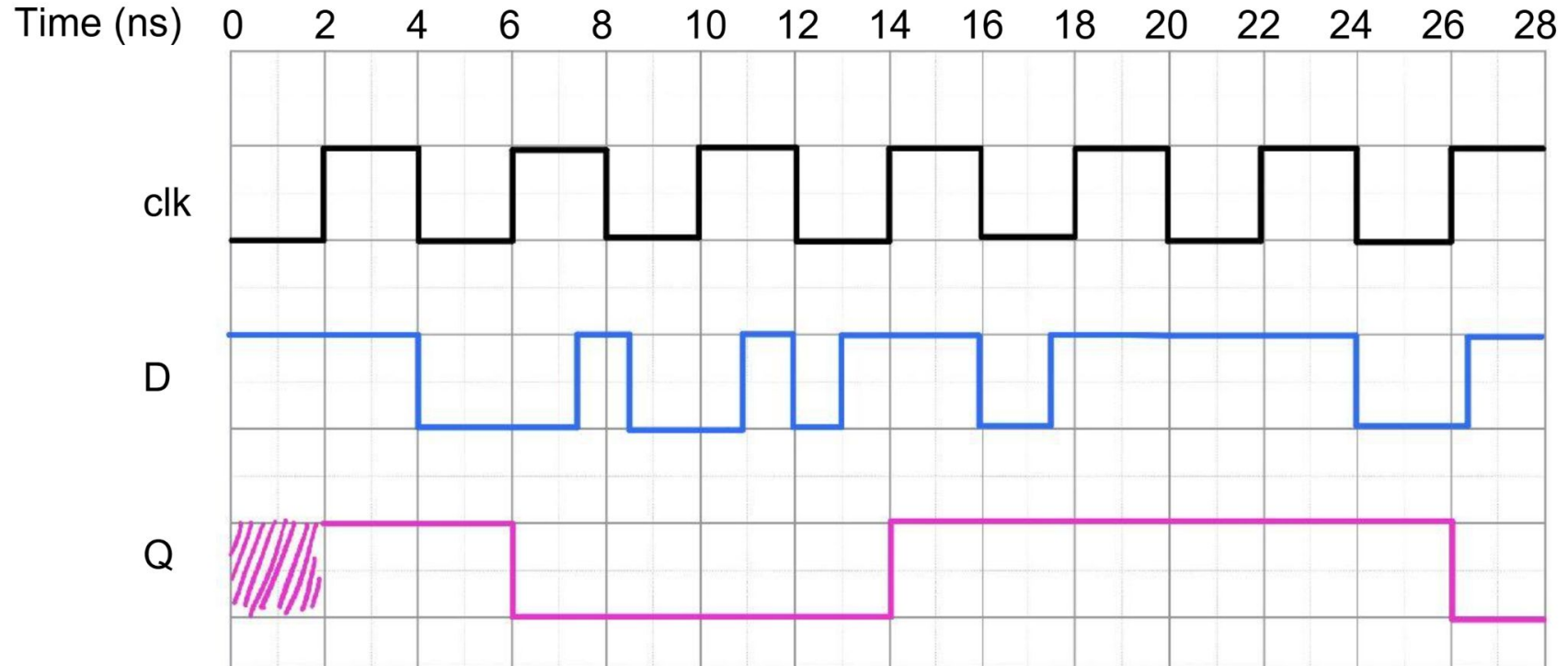
# Synchronous Digital Systems

## Flip-Flop Timing Example



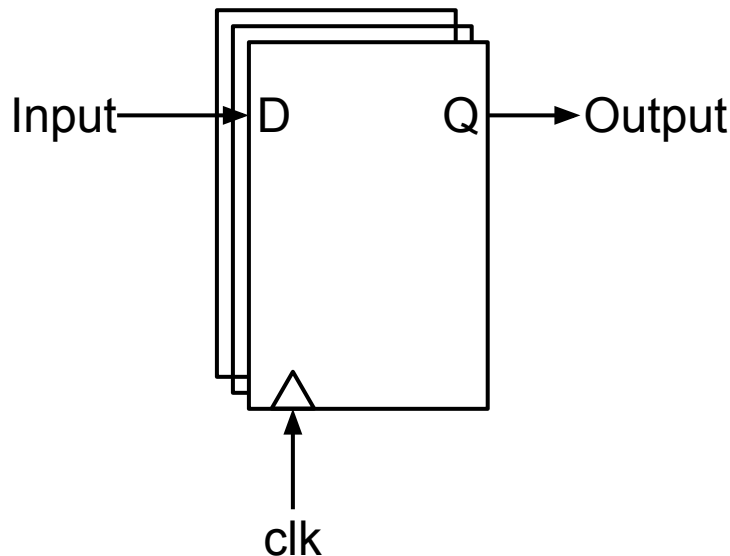
# Synchronous Digital Systems

## Flip-Flop Timing Example



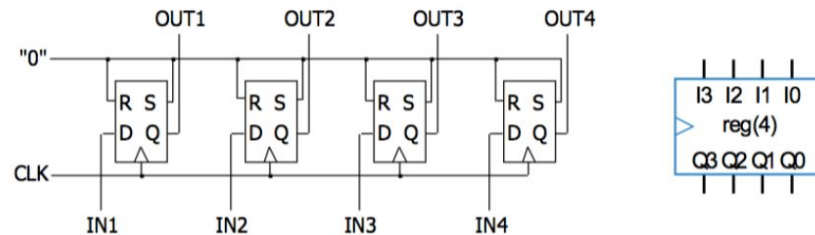
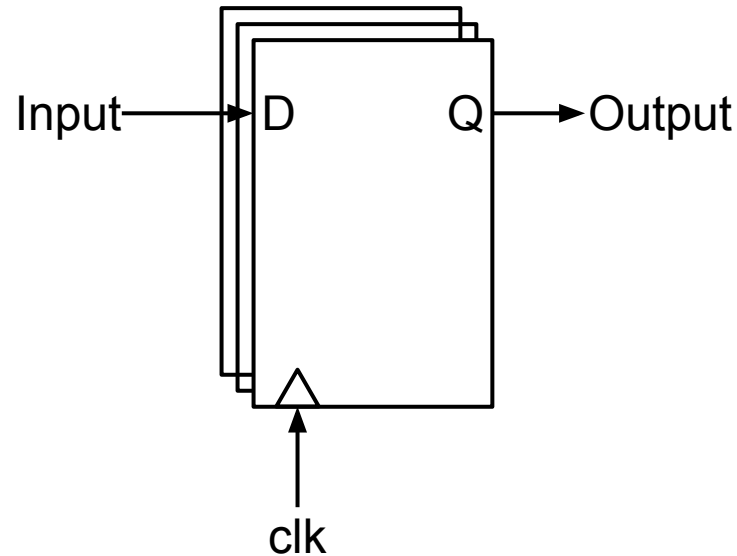
# Registers

- *n*-bit register: *n* flip-flops bundled together
  - Each flip-flop stores 1 bit of data
  - The register stores *n* bits of data in total
- Inputs:
  - *n*-bit value D
  - One-bit clock value
- Outputs:
  - *n*-bit value Q
- Behavior:
  - On the rising edge of the clock, set  $Q=D$
  - At all other times, do nothing



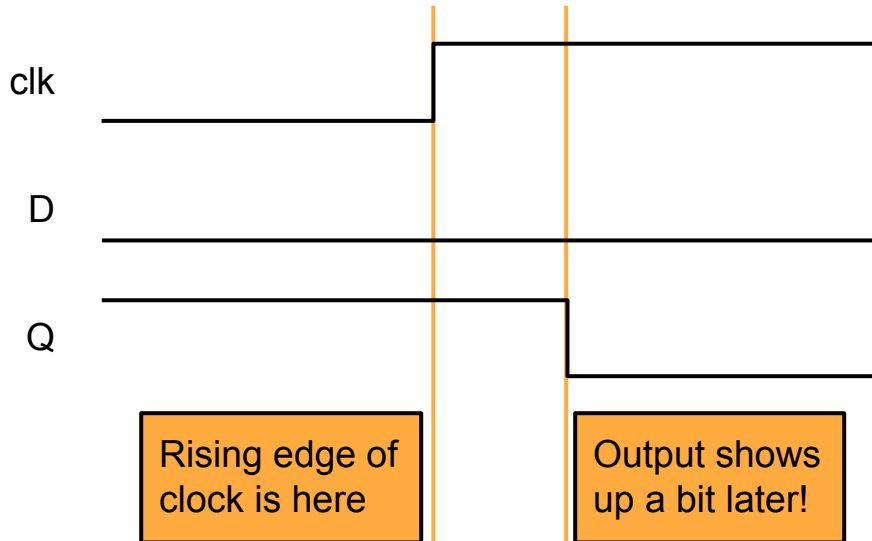
# Registers

- *n*-bit register: *n* flip-flops bundled together
  - Each flip-flop stores 1 bit of data
  - The register stores *n* bits of data in total
- Inputs:
  - *n*-bit value D
  - One-bit clock value
- Outputs:
  - *n*-bit value Q
- Behavior:
  - On the rising edge of the clock, set  $Q=D$
  - At all other times, do nothing

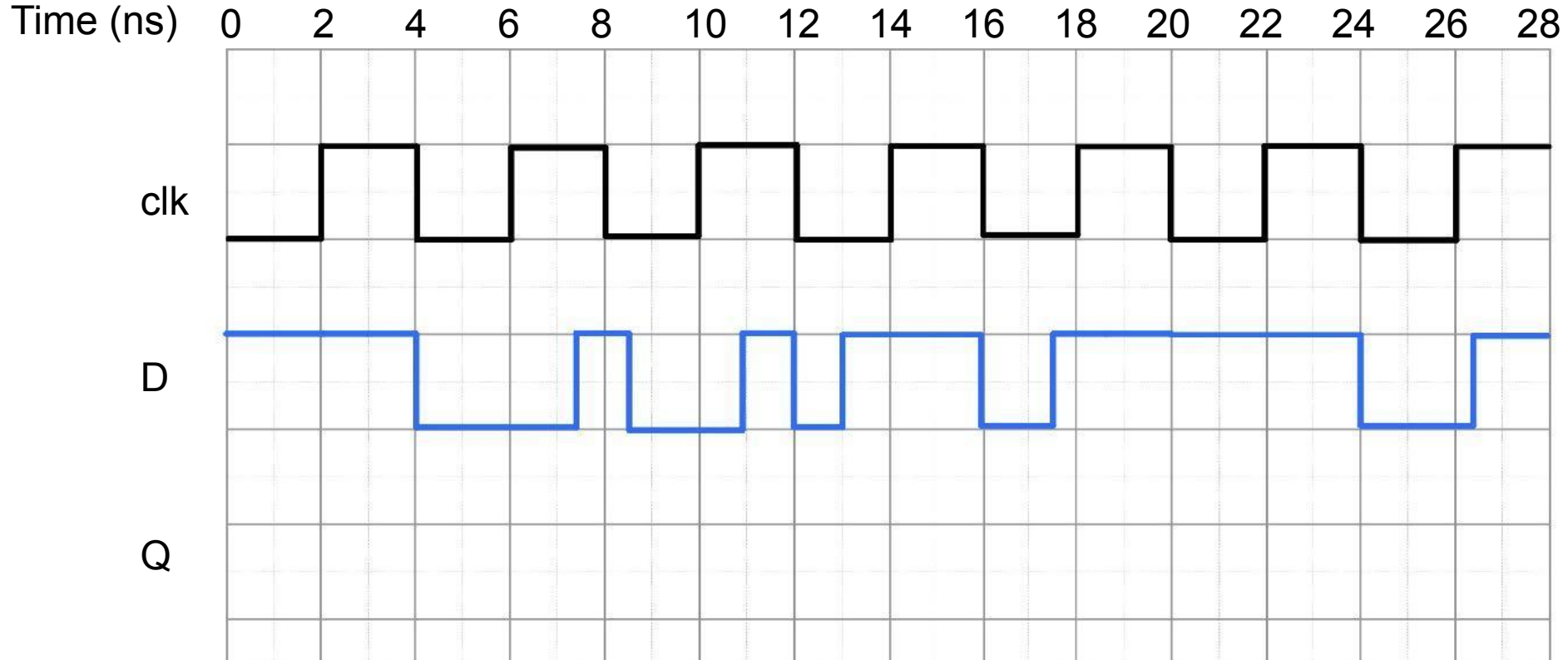


# Register/Flip Flop Delay: clk-to-q delay

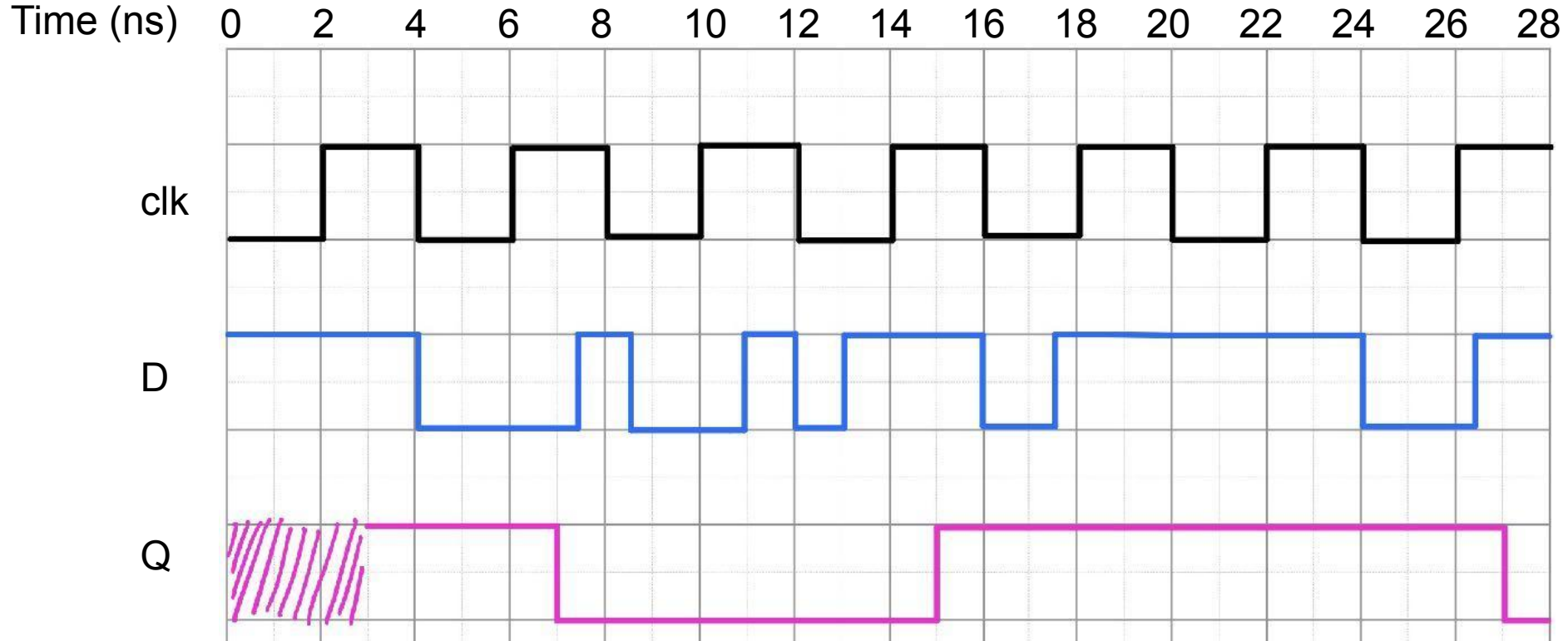
- Registers/Flip Flops can't transfer the D input to Q output instantly
- clk-to-q delay: Time it takes after the rising edge for the Q output to change



# Flip-Flop Timing with 1 ns clk-to-q Delay



# Flip-Flop Timing with 1 ns clk-to-q Delay



## Flip-Flop Timing with 1 ns clk-to-q Delay

In the context of a flip-flop, the **"1 ns clk-to-Q delay"** refers to the **propagation delay** of the flip-flop. This is the time it takes for a change at the clock input to be reflected at the output Q after the clock edge triggers the flip-flop.

So, if a flip-flop has a 1 ns clk-to-Q delay, it means that after a clock edge occurs, it will take 1 ns before the new value is reflected at the output Q. This is an inherent characteristic of the flip-flop and is an essential parameter for designing reliable synchronous digital circuits.





*Break!*

## Counters

Design a 2-bit counter with enable

# Counters

## Design a 2-bit counter with enable

Truth Table for 2-Bit Counter with Enable

Enable	Q1 (Previous)	Q0 (Previous)	Q1 (Next)	Q0 (Next)	Description
0	0	0	0	0	Hold (No Change)
0	0	1	0	1	Hold (No Change)
0	1	0	1	0	Hold (No Change)
0	1	1	1	1	Hold (No Change)
1	0	0	0	1	Count 0 to 1
1	0	1	1	0	Count 1 to 2
1	1	0	1	1	Count 2 to 3
1	1	1	0	0	Count 3 to 0 (Reset)

# Counters

## Design a 2-bit counter with enable

Truth Table for 2-Bit Counter with Enable

Enable	Q1 (Previous)	Q0 (Previous)	Q1 (Next)	Q0 (Next)	Description
0	0	0	0	0	Hold (No Change)
0	0	1	0	1	Hold (No Change)
0	1	0	1	0	Hold (No Change)
0	1	1	1	1	Hold (No Change)
1	0	0	0	1	Count 0 to 1
1	0	1	1	0	Count 1 to 2
1	1	0	1	1	Count 2 to 3
1	1	1	0	0	Count 3 to 0 (Reset)

$$Q1_{next} = \sum(2,3,5,6)$$

$$Q0_{next} = \sum(1,3,4,6)$$

# Counters

## Design a 2-bit counter with enable

$$Q1_{next} = \sum(2,3,5,6)$$

0 0	1 0	3 1	2 1
4 0	5 1	7 0	6 1

$$Q0_{next} = \sum(1,3,4,6)$$

0 0	1 1	3 1	2 0
4 1	5 0	7 0	6 1

## Counters

Design a 2-bit counter with enable

0 0	1 0	3 1	2 1
4 0	5 1	7 0	6 1

$$Q1_{\text{next}} = \sum(2,3,5,6)$$

$$Y1 = \text{En}' \cdot Q1 + Q1 \cdot Q0' + \text{En} \cdot Q0 \cdot Q1'$$

## Counters

Design a 2-bit counter with enable

$$Q0_{next} = \sum(1,3,4,6)$$

0 0	1 1	3 1	2 0
4 1	5 0	7 0	6 1

$$Y0 = E_n'.Q0 + Q0'.E_n$$

## Counters

Design a 2-bit counter with enable

$$Y1 = En'.Q1 + Q1.Q0' + En.Q0.Q1'$$

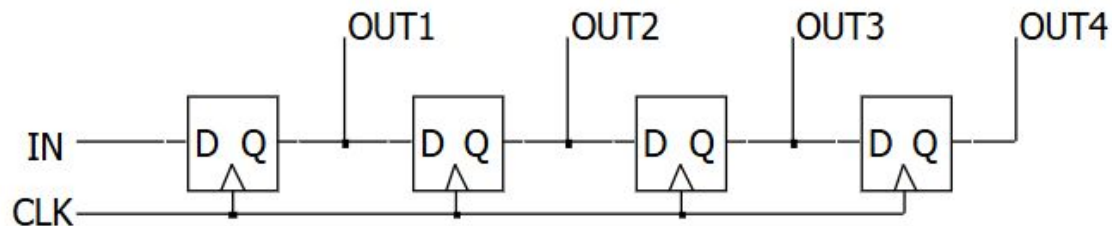
$$Y0 = En'.Q0 + Q0'.En$$

**Assignment:** Draw the 2-bit counter with enable circuit diagram with D FF



# Shift register

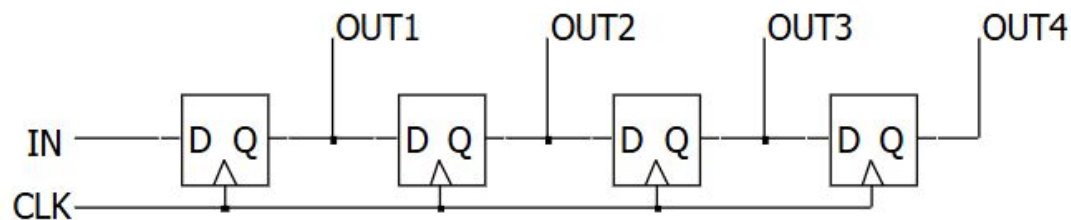
- Holds & shifts samples of input



Time	Input	OUT1	OUT2	OUT3	OUT4
0	1	0	0	0	0
1	0				
2	1				
3	1				
4	0				
5	0				
6	0				

# Shift register

- Holds & shifts samples of input



Time	Input	OUT1	OUT2	OUT3	OUT4
0	1	0	0	0	0
1	0	1	0	0	0
2	1	0	1	0	0
3	1	1	0	1	0
4	0	1	1	0	1
5	0	0	1	1	0
6	0	0	0	1	1

## Chapter 3

Next week we will continue with Finite  
State Machines

# Chapter 4 - COMBINATIONAL LOGIC

## Bitwise Operators

### 1. Bitwise AND (&)

The `&` operator performs a logical AND operation on each bit of its operands. The result is `1` only if both bits in the corresponding positions are `1`.

```
module bitwise_and_example;
    logic [3:0] a = 4'b1101; // Input 1
    logic [3:0] b = 4'b1011; // Input 2
    logic [3:0] result;

    initial begin
        result = a & b;
        $display("Bitwise AND: %b & %b = %b", a, b, result);
        // Output: Bitwise AND: 1101 & 1011 = 1001
    end
endmodule
```

# Chapter 4 - COMBINATIONAL LOGIC

## Bitwise Operators

### 2. Bitwise OR (|)

The `|` operator performs a logical OR operation on each bit. The result is `1` if at least one of the corresponding bits is `1`.

```
module bitwise_or_example;
    logic [3:0] a = 4'b1101;
    logic [3:0] b = 4'b1011;
    logic [3:0] result;

    initial begin
        result = a | b;
        $display("Bitwise OR: %b | %b = %b", a, b, result);
        // Output: Bitwise OR: 1101 | 1011 = 1111
    end
endmodule
```

# Chapter 4 - COMBINATIONAL LOGIC

## Bitwise Operators

### 3. Bitwise XOR (^)

The ^ operator performs an exclusive OR (XOR) operation on each bit. The result is 1 if the bits in the corresponding positions are different.

```
module bitwise_xor_example;
    logic [3:0] a = 4'b1101;
    logic [3:0] b = 4'b1011;
    logic [3:0] result;

    initial begin
        result = a ^ b;
        $display("Bitwise XOR: %b ^ %b = %b", a, b, result);
        // Output: Bitwise XOR: 1101 ^ 1011 = 0110
    end
endmodule
```

# Chapter 4 - COMBINATIONAL LOGIC

## Bitwise Operators

### 4. Bitwise NOT (~)

The `~` operator performs a bitwise negation (NOT). It inverts each bit: `0` becomes `1`, and `1` becomes `0`.

```
module bitwise_not_example;
    logic [3:0] a = 4'b1101;
    logic [3:0] result;

    initial begin
        result = ~a;
        $display("Bitwise NOT: ~%b = %b", a, result);
        // Output: Bitwise NOT: ~1101 = 0010
    end
endmodule
```

# Chapter 4 - COMBINATIONAL LOGIC

## Bitwise Operators

### 5. Bitwise NAND ( ~& )

The `~&` operator performs a bitwise AND and then negates the result. It is the complement of the AND operation.

```
module bitwise_nand_example;
    logic [3:0] a = 4'b1101;
    logic [3:0] b = 4'b1011;
    logic [3:0] result;

    initial begin
        result = ~(a & b);
        $display("Bitwise NAND: ~( %b & %b ) = %b", a, b, result);
        // Output: Bitwise NAND: ~(1101 & 1011) = 0110
    end
endmodule
```



# Chapter 4 - COMBINATIONAL LOGIC

## Bitwise Operators

### 6. Bitwise NOR ( $\sim|$ )

The  $\sim|$  operator performs a bitwise OR and then negates the result. It is the complement of the OR operation.

```
module bitwise_nor_example;
    logic [3:0] a = 4'b1101;
    logic [3:0] b = 4'b1011;
    logic [3:0] result;

    initial begin
        result = ~(a | b);
        $display("Bitwise NOR: ~( %b | %b ) = %b", a, b, result);
        // Output: Bitwise NOR: ~(1101 | 1011) = 0000
    end
endmodule
```

# Chapter 4 - COMBINATIONAL LOGIC

## Bitwise Operators

### 7. Bitwise XNOR ( $\sim^{\wedge}$ or $\wedge^{\sim}$ )

The  $\sim^{\wedge}$  or  $\wedge^{\sim}$  operator performs a bitwise XOR and then negates the result. It is the complement of the XOR operation.

```
module bitwise_xnor_example;
    logic [3:0] a = 4'b1101;
    logic [3:0] b = 4'b1011;
    logic [3:0] result;

    initial begin
        result = ~(a ^ b);
        $display("Bitwise XNOR: ~( %b ^ %b ) = %b", a, b, result);
        // Output: Bitwise XNOR: ~(1101 ^ 1011) = 1001
    end
endmodule
```

# Chapter 4 - COMBINATIONAL LOGIC

## Bitwise Operators

### **Applications of Bitwise Operators in SystemVerilog**

1. **Masking:** Using AND to clear specific bits.
2. **Setting Bits:** Using OR to set specific bits to 1.
3. **Toggling Bits:** Using XOR to flip specific bits.
4. **Inversion:** Using NOT to complement a binary value.
5. **Logic Operations in Modules:** Efficiently implement logic functions in combinational circuits.

# Chapter 4 - COMBINATIONAL LOGIC

## Bitwise Operators

```
module masking_setting_example;
    logic [7:0] data = 8'b11011010;
    logic [7:0] mask = 8'b11110000;
    logic [7:0] set_bits = 8'b00001111;
    logic [7:0] masked_data, updated_data;

    initial begin
        masked_data = data & mask; // Masking
        updated_data = data | set_bits; // Setting bits
        $display("Masked Data: %b & %b = %b", data, mask, masked_data);
        $display("Updated Data: %b | %b = %b", data, set_bits, updated_data);
        // Output: Masked Data: 11011010 & 11110000 = 11010000
        //           Updated Data: 11011010 | 00001111 = 11011111
    end
endmodule
```

## Chapter 4 - COMBINATIONAL LOGIC

### Conditional Assignment

Conditional assignments select the output from among alternatives based on an input called the condition.

The conditional operator `?:` chooses, based on a first expression, between a second and third expression. The first expression is called the condition. If the condition is 1, the operator chooses the second expression. If the condition is 0, the operator chooses the third expression.

# Chapter 4 - COMBINATIONAL LOGIC

## Conditional Assignment

?: is especially useful for describing a multiplexer because, based on the first input, it selects between two others.

2:1 multiplexer with 4-bit inputs and outputs using the conditional operator.

```
module mux_2to1 (  
    input logic [3:0] in0,    // 4-bit input 0  
    input logic [3:0] in1,    // 4-bit input 1  
    input logic sel,          // Selection line  
    output logic [3:0] out     // 4-bit output  
);  
  
    // Conditional operator for the multiplexer  
    always_comb begin  
        out = sel ? in1 : in0;  
    end  
  
endmodule
```

# Chapter 4 - COMBINATIONAL LOGIC

## Verification of 2:1 mux?

```
// Testbench for the 2:1 multiplexer
module tb_mux_2to1;
    logic [3:0] in0, in1; // Inputs
    logic sel;           // Selection line
    logic [3:0] out;     // Output

    // Instantiate the 2:1 multiplexer
    mux_2to1 uut (
        .in0(in0),
        .in1(in1),
        .sel(sel),
        .out(out)
    );

    initial begin
        // Test case 1: sel = 0, select in0
        in0 = 4'b1010;
        in1 = 4'b0101;
        sel = 0;
        #10;
        $display("Test Case 1: sel=%b, in0=%b, in1=%b, out=%b", sel, in0, in1, out);

        // Test case 2: sel = 1, select in1
        sel = 1;
        #10;
        $display("Test Case 2: sel=%b, in0=%b, in1=%b, out=%b", sel, in0, in1, out);
    end
endmodule
```

```
mux_2to1 uut (
    .in0(in0), // Connect testbench signal `in0` to module port `in0`
    .in1(in1), // Connect testbench signal `in1` to module port `in1`
    .sel(sel), // Connect testbench signal `sel` to module port `sel`
    .out(out)  // Connect module port `out` to testbench signal `out`
);
```

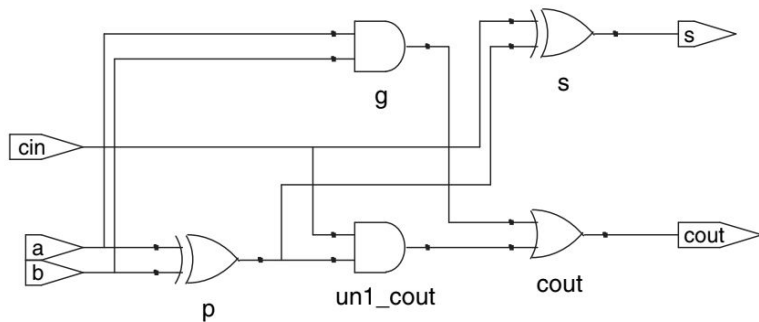
```
// Test case 3: Different inputs
in0 = 4'b1111;
in1 = 4'b0000;
sel = 0;
#10;
$display("Test Case 3: sel=%b, in0=%b, in1=%b, out=%b", sel, in0, in1, out);

sel = 1;
#10;
$display("Test Case 4: sel=%b, in0=%b, in1=%b, out=%b", sel, in0, in1, out);

// End simulation
$finish;

end
endmodule
```

# Chapter 4 - COMBINATIONAL LOGIC



**Figure 4.8** fulladder synthesized circuit

## SystemVerilog

In SystemVerilog, internal signals are usually declared as `logic`.

```
module fulladder(input  logic a, b, cin,
                output logic s, cout);
```

```
logic p, g;
```

```
assign p = a ^ b;
```

```
assign g = a & b;
```

```
assign s = p ^ cin;
```

```
assign cout=g | (p & cin);
```

endmodule



# Chapter 4 - COMBINATIONAL LOGIC

## SystemVerilog

**Table 4.1** SystemVerilog operator precedence

	Op	Meaning
H i g h e s t	~	NOT
	*, /, %	MUL, DIV, MOD
	+, -	PLUS, MINUS
	<<, >>	Logical Left/Right Shift
	<<<, >>>	Arithmetic Left/Right Shift
	<, <=, >, >=	Relative Comparison
	==, !=	Equality Comparison
L o w e s t	&, ~&	AND, NAND
	^, ~^	XOR, XNOR
	, ~	OR, NOR
	?:	Conditional

Logical and arithmetic shifts are operations that shift the bits of a number to the left or right. The key difference between them lies in how they handle the **sign bit** (most significant bit) and the type of data they are designed for.

# Chapter 4 - COMBINATIONAL LOGIC

## 1. Logical Shift

- **Definition:** Bits are shifted left or right, and vacant bit positions are filled with 0.
- **Application:** Used for unsigned numbers because the sign bit is not preserved.
- **Operation:**
  - **Logical Left Shift (LSL):**
    - Moves bits to the left.
    - New bits on the right are set to 0.
    - Example:  
1010 (10) Logical Left Shift by 1 → 10100 (20)
  - **Logical Right Shift (LSR):**
    - Moves bits to the right.
    - New bits on the left are set to 0.
    - Example:  
1010 (10) Logical Right Shift by 1 → 0101 (5)

## 2. Arithmetic Shift

- **Definition:** Bits are shifted left or right, but the **sign bit** is preserved during right shifts (only for signed numbers).
- **Application:** Used for signed numbers in two's complement representation.
- **Operation:**
  - **Arithmetic Left Shift (ASL):**
    - Same as logical left shift.
    - New bits on the right are set to 0.
    - The sign bit is not preserved because the value might change.
    - Example:  
1010 (-6 in 4-bit two's complement) Arithmetic Left Shift by 1 → 0100 (8)
  - **Arithmetic Right Shift (ASR):**
    - Moves bits to the right.
    - The **sign bit** (leftmost bit) is copied to the new leftmost positions, preserving the number's sign.
    - Example:  
1010 (-6 in 4-bit two's complement) Arithmetic Right Shift by 1 → 1101 (-3)

# Chapter 4 - COMBINATIONAL LOGIC

## Key Differences

Aspect	Logical Shift	Arithmetic Shift
Purpose	Used for unsigned numbers	Used for signed numbers
Left Shift Behavior	Same for both	Same for both
Right Shift Behavior	Vacant bits filled with 0	Sign bit is preserved for right shifts
Preservation of Sign	Does not preserve the sign bit	Preserves the sign bit (right shift only)
Typical Use Cases	Bit masking, multiplying/dividing by powers of 2 for unsigned values	Arithmetic operations on signed numbers

# Chapter 4 - SEQUENTIAL LOGIC

## A 4-bit flip-flop in SystemVerilog

```
module flop(  
    input logic clk,           // Clock signal  
    input logic [3:0] d,       // 4-bit input data  
    output logic [3:0] q       // 4-bit output data  
);  
    always_ff @(posedge clk)    // Trigger on positive edge of the clock  
        q <= d;                // Non-blocking assignment: q gets the value of d  
endmodule
```

This code defines a **4-bit flip-flop** in SystemVerilog. It takes a clock signal ( `clk` ) and a 4-bit input ( `d` ), and stores the value of `d` in the output ( `q` ) on every **positive edge of the clock**.

# Chapter 4 - SEQUENTIAL LOGIC

## A 4-bit flip-flop in SystemVerilog

### Key Concepts in Detail

#### 1. `always_ff` Block

- The `always_ff` block is a special kind of `always` block that **exclusively models flip-flops**.
- **Syntax:** `always_ff @(posedge clk)`
  - The block is triggered on the **positive edge** of the `clk` signal ( `posedge clk` ).
  - Inside the block, assignments represent behavior that happens when the clock signal transitions from low (0) to high (1).
- **Purpose of `always_ff`:**
  - It explicitly specifies that the block is intended to model flip-flops.
  - Helps avoid unintended behavior (e.g., combinational logic or latches) and allows synthesis tools to warn the user if the block's behavior does not correspond to flip-flops.


# Chapter 4 - SEQUENTIAL LOGIC

## A 4-bit flip-flop in SystemVerilog

### 2. Sensitivity List

- In general, the `always` block monitors the **sensitivity list**:
- The code inside the block is executed when the event in the sensitivity list occurs.
- In this example:
  - `@(posedge clk)` means the block runs **only** on the rising edge of the clock signal.


`always @(sensitivity list)`



# Chapter 4 - SEQUENTIAL LOGIC

## A 4-bit flip-flop in SystemVerilog

`always @(sensitivity list)`



- **Sensitivity List Types:**

- `@(posedge clk)` → Triggers on the rising edge of `clk`.
- `@(negedge clk)` → Triggers on the falling edge of `clk`.
- `@(*)` → Automatically infers all signals used in the block (typically for combinational logic).

# Chapter 4 - SEQUENTIAL LOGIC

## A 4-bit flip-flop in SystemVerilog

### 3. Flip-Flop Behavior

- Flip-flops are **edge-triggered storage elements** that update their output only on specific clock edges.
- Behavior in the Code:
  - `q <= d;`
    - When the clock signal ( `clk` ) rises (positive edge), the value of `d` is **copied into** `q` .
    - Otherwise, `q` retains its previous value, which mimics the "memory" behavior of flip-flops.



# Chapter 4 - SEQUENTIAL LOGIC

## A 4-bit flip-flop in SystemVerilog

### 4. Nonblocking Assignment ( `<=` )

- Purpose:
  - Nonblocking assignments ( `<=` ) are used in sequential logic (like flip-flops) to ensure the right-hand side (RHS) of the assignment is evaluated first, and the result is assigned to the left-hand side (LHS) after the current time step.
- Key Features of Nonblocking Assignment:
  - Allows multiple sequential blocks to run in parallel.
  - Avoids race conditions in flip-flop chains.

# Chapter 4 - SEQUENTIAL LOGIC

## A 4-bit flip-flop in SystemVerilog

```
always_ff @(posedge clk) begin
    q1 <= d;    // Assigns d to q1
    q2 <= q1;   // Uses the previous value of q1
end
```

In this example, `q2` will get the old value of `q1`, not the newly updated value. This ensures predictable and correct sequential behavior.

# Chapter 4 - SEQUENTIAL LOGIC

## A 4-bit flip-flop in SystemVerilog

5. `always_ff`, `always_latch`, and `always_comb`

- Problem with Generic `always` :
  - A generic `always` block can model flip-flops, latches, or combinational logic, depending on how it is written.
  - This flexibility can lead to accidental hardware mismatches or bugs.

# Chapter 4 - SEQUENTIAL LOGIC

## A 4-bit flip-flop in SystemVerilog

- **Solution: Specialized Constructs:**
  - `always_ff` : Used to model **flip-flops**.
  - `always_latch` : Used to model **latches**.
  - `always_comb` : Used to model **combinational logic**.
- **Advantages of Specialized Constructs:**
  - Helps tools detect mismatches between the intent and implementation.
  - Ensures better readability and maintainability of the code.
  - Avoids unintended hardware generation.

# Chapter 4 - SEQUENTIAL LOGIC

## A 4-bit flip-flop in SystemVerilog

### How Flip-Flops Work in the Example

#### 1. Triggering Event:

- The block is executed only when a rising edge of `clk` occurs.

#### 2. Assignment ( `q <= d` ):

- The value of `d` is stored in `q`.
- `q` holds this value until the next rising edge of the clock.

#### 3. State Retention:

- If no rising edge occurs, `q` retains its previous value, demonstrating the "memory" behavior of flip-flops.

```
module flop(  
    input logic clk,           // Clock signal  
    input logic [3:0] d,       // 4-bit input data  
    output logic [3:0] q       // 4-bit output data  
);  
    always_ff @(posedge clk) // Trigger on positive edge of the clock  
        q <= d;             // Non-blocking assignment: q gets the value of d  
endmodule
```

# Chapter 4 - SEQUENTIAL LOGIC

## A 4-bit flip-flop in SystemVerilog

### Summary of Terminology

Term	Explanation
Sensitivity List	Specifies when the <code>always</code> block executes (e.g., <code>@(posedge clk)</code> ).
<code>always_ff</code>	Explicitly models flip-flops, ensuring clear intent and avoiding synthesis errors.
Nonblocking Assignment ( <code>&lt;=</code> )	Used in sequential logic to avoid race conditions and ensure predictable behavior.
Flip-Flop	Edge-triggered storage element that updates its output on clock edges.

# Chapter 4 - SEQUENTIAL LOGIC

## A 4-bit flip-flop in SystemVerilog

### Why Use `always_ff`?

- **Error Prevention:** Ensures that only flip-flop behavior is modeled.
- **Tool Warnings:** Synthesis tools can detect errors if the code does not match flip-flop semantics.
- **Code Clarity:** Explicitly states the designer's intent to model flip-flops, reducing ambiguity.



# Assignment:

Look at full adder using *always/process*  
in your textbook (pg. 200)





*End of Week 5!*