

---

# Chapter 3 – Boolean Algebra and Digital Logic

# Objectives

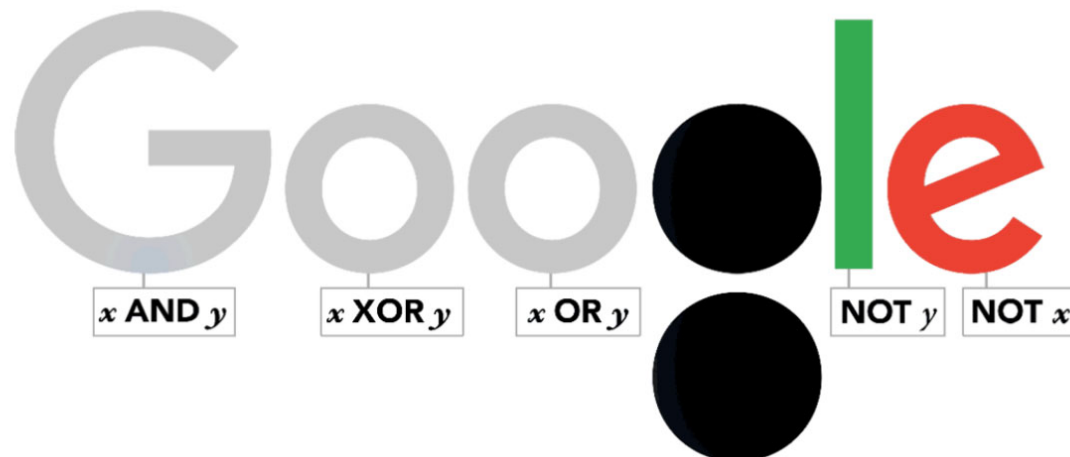
---

- ❑ Understand the relationship between **Boolean logic** and digital computer **circuits**.
- ❑ Learn how to design **simple** logic circuits.
- ❑ Understand how digital circuits work together to form **complex** computer systems.

# Introduction

---

- In the latter 19<sup>th</sup> century, George Boole suggested that *logical thought could be represented through mathematical equations.*
- Boolean algebra is **everywhere**
- <https://www.google.com/doodles/george-booles-200th-birthday>



# Application of Boolean Algebra

---

- ☐ Digital circuit
- ☐ Google search
- ☐ Database (SQL)
- ☐ Programming
- ☐ .....

# An Example on Programming

---

```
while ((A && B) || (A && !B)) || !A)
{
    // do something
}
```

$$\begin{aligned} &= AB + AB' + A' \\ &= A(B + B') + A' \\ &= A(1) + A' \\ &= 1 \end{aligned}$$

## 3.2 Boolean Algebra

---

- Boolean algebra is a mathematical system for manipulating variables that can have one of two values.
  - In formal logic, these values are “true” and “false”
  - In digital systems, these values are “on”/“off,” “high”/“low,” or “1”/“0”.
  - So, it is perfect for binary number systems
- Boolean expressions are created to operate Boolean variables.
  - Common Boolean operators include AND, OR, and NOT.

# Boolean Algebra

- The function of Boolean operator can be **completely** described using a **Truth Table**.
- The truth tables of the Boolean operators AND and OR are shown on the right.
- The AND operator is also known as the **Boolean product “.”**. The OR operator is the **Boolean sum “+”**.

X AND Y

X	Y	XY
0	0	0
0	1	0
1	0	0
1	1	1

X OR Y

X	Y	X+Y
0	0	0
0	1	1
1	0	1
1	1	1

# Boolean NOT

---

- The truth table of the Boolean NOT operator is shown on the right.
- The NOT operation is most often designated by an **overbar** “ $\overline{\phantom{x}}$ ”.
- Some books use the prime mark (  $'$  ) or the “elbow” (  $\neg$  ), for instead.

NOT x	
x	$\overline{x}$
0	1
1	0



# Boolean Function

---

- A Boolean function has:
  - At least one Boolean variable,
  - At least one Boolean operator, and
  - At least one input from the set of  $\{0,1\}$ .
- It produces an output that is a member of the set  $\{0,1\}$  – Either 0 or 1.

Now you know why the binary numbering system is so handy for digital systems.

# Boolean Algebra

- Let's look at a truth table for the following Boolean function shown on the right. :

$$F(x, y, z) = x\bar{z} + y$$

- To evaluate the Boolean function easier, the truth table contains a **extra columns (shaded)** to hold the evaluations of partial function.

$$F(x, y, z) = x\bar{z} + y$$


x	y	z	$\bar{z}$	$x\bar{z}$	$x\bar{z} + y$
0	0	0	1	0	0
0	0	1	0	0	0
0	1	0	1	0	1
0	1	1	0	0	1
1	0	0	1	1	1
1	0	1	0	0	0
1	1	0	1	1	1
1	1	1	0	0	1

# Rules Of Precedence

- Arithmetic has its rules of precedence
  - Like arithmetic, Boolean operations follow the **rules of precedence** (priority):
  - **NOT** operator > **AND** operator > **OR** operator .
- This explains why we chose the shaded partial function in that order in the table.

$F(x, y, z) = x\bar{z} + y$

x	y	z	$\bar{z}$	$x\bar{z}$	$x\bar{z} + y$
0	0	0	1	0	0
0	0	1	0	0	0
0	1	0	1	0	1
0	1	1	0	0	1
1	0	0	1	1	1
1	0	1	0	0	0
1	1	0	1	1	1
1	1	1	0	0	1



Rules Of Precedence

# Use Boolean Algebra in Circuit Design

---

- Digital circuit designer always like achieve the following goals:
  - ***Cheaper*** to produce
  - Consume ***less power***
  - run ***faster***
- How to do it? -- We know that:
  - Computers contain circuits that implement Boolean functions → Boolean functions can express circuits
  - If we can simplify a Boolean function, that express a circuit, we can archive the above goals
- We always can reduce a Boolean function to its ***simplest*** form by using a number of Boolean laws can help us do so.

# Boolean Algebra Laws

---

- Most Boolean algebra laws have either an **AND (product)** form or an **OR (sum)** form. We give the laws with both forms.
  - Since the laws are always true, so X (and Y) could be either 0 or 1

Identity Name	AND Form	OR Form
Identity Law	$1x = x$	$0 + x = x$
Null Law	$0x = 0$	$1 + x = 1$
Idempotent Law	$xx = x$	$x + x = x$
Inverse Law	$x\bar{x} = 0$	$x + \bar{x} = 1$

# Boolean Algebra Laws (`Cont)

- The second group of Boolean laws should be familiar to you from your study of algebra:

Identity Name	AND Form	OR Form
Commutative Law	$xy = yx$	$x+y = y+x$
Associative Law	$(xy)z = x(yz)$	$(x+y)+z = x+(y+z)$
Distributive Law	$x+yz = (x+y)(x+z)$	$x(y+z) = xy+xz$

$$\begin{aligned}(x+y)(x+z) &= xx+xz+xy+yz \\ &= x+xz+xy+yz = x(1+z)+xy+yz \\ &= x+xy+yz = x(1+y)+yz \\ &= x+yz\end{aligned}$$

# Boolean Algebra Laws (`Cont)

- The last group of Boolean laws are perhaps the most useful.
  - If you have studied set theory or formal logic, these laws should be familiar to you.

Identity Name	AND Form	OR Form
Absorption Law	$x(x+y) = x$	$x + xy = x$
DeMorgan's Law	$\overline{(xy)} = \bar{x} + \bar{y}$	$\overline{(x+y)} = \bar{x}\bar{y}$
Double Complement Law	$\overline{(\bar{x})} = x$	

# DeMorgan's law

- DeMorgan's law provides an easy way of finding the negation (complement) of a Boolean function.
- DeMorgan's law states:

$$\overline{(xy)} = \bar{x} + \bar{y} \quad \text{and} \quad \overline{(x+y)} = \bar{x}\bar{y}$$



- Example

$z = xy = \text{coming to school}$   
 $x = \text{car is working}$   
 $y = \text{will not be snowing}$

More Examples?

- I **will** come to school tomorrow if

- (A) my car is working, **and**
- (B) it won't be snowing

- =** ■ I **won't** come to school tomorrow if

- (A) my car **is not** working, **or**
- (B) it **will** snowing





# DeMorgan's Law

---

- DeMorgan's law can be extended to any number of variables.
  - Replace each variable by its negation (complement)
  - Change all ANDs to ORs and all ORs to ANDs.
- Let's say  $F(X, Y, Z)$  is the following, what is  $\bar{F}$  ?

$$F(X, Y, Z) = (XY) + (\bar{X}Y) + (X\bar{Z})$$

$$\begin{aligned} F' &= (X'+Y')(X+Y')(X'+Z) \\ &= (XX'+Y')(X'+Z) \\ &= X'Y'+Y'Z \end{aligned}$$

# Simplify Boolean function

□ Let's use Boolean laws to simplify:

as follows:  $F(X, Y, Z) = (X+Y) (X+\bar{Y}) (\bar{X}\bar{Z})$

$(X + Y) (X + \bar{Y}) (\bar{X}\bar{Z})$	
$(X + Y) (X + \bar{Y}) (\bar{X} + Z)$	<u>DeMorgan's Law</u>
$(XX + X\bar{Y} + YX + Y\bar{Y}) (\bar{X} + Z)$	<u>Double complement Law</u>
$((X + Y\bar{Y}) + X(Y + \bar{Y})) (\bar{X} + Z)$	<u>Distributive Law</u>
$((X + 0) + X(1)) (\bar{X} + Z)$	Commutative and Distributive Laws
$X(\bar{X} + Z)$	Inverse Law
$X\bar{X} + XZ$	Idempotent and Identity Laws
$0 + XZ$	Distributive Law
$XZ$	Inverse Law
	Identity Law

Identity Name	AND Form	OR Form
Identity Law	$1x = x$	$0 + x = x$
Null (or Dominance) Law	$0x = 0$	$1 + x = 1$
Idempotent Law	$xx = x$	$x + x = x$
Inverse Law	$xx' = 0$	$x + x' = 1$
Commutative Law	$xy = yx$	$x + y = y + x$
Associative Law	$(xy)z = x(yz)$	$(x + y) + z = x + (y + z)$
Distributive Law	$x + (yz) = (x + y)(x + z)$	$x(y + z) = xy + xz$
Absorption Law	$x(x + y) = x$	$x + xy = x$
DeMorgan's Law	$(xy)' = x' + y'$	$(x + y)' = x'y'$
Double Complement Law	$x'' = x$	

$$X + X' Y = X + Y$$

$$\begin{aligned}
 (x+y)(x+z) &= xx+xz+xy+yz \\
 &= x+xz+xy+yz = x(1+z)+xy+yz \\
 &= x+xy+yz = x(1+y)+yz \\
 &= x+yz
 \end{aligned}$$

# Logic simplification steps

---

- Apply De Morgan's theorems
- Expanding out parenthesis
- Find the common factors
- Popular rules used:

$$X + XY = X$$

$$XY + X\bar{Y} = X$$

$$X + \bar{X}Y = X + Y$$

$$X + X = X, \quad XX = X$$

$$X + 0 = X, \quad X + 1 = 1$$

$$X0 = 0 \quad X1 = X$$

$$\overline{(\overline{xy})} = \overline{x} + \overline{y} \quad | \quad \overline{(x+y)} = \overline{x}\overline{y}$$

## Example (1)

□ Apply De Morgan's theorem

$$\overline{WXYZ} = W' + X' + Y' + Z' \quad \overline{W+X+Y+Z} = W' X' Y' Z'$$

$$\overline{(A+B+C)D} = D' + A'B'C'$$

$$\overline{\overline{A}B + \overline{C}D + EF} = (A'+B)(C+D')(E'+F')$$

## Example (2)

- $(\overline{A}\overline{B}(C+BD)+\overline{A}\overline{B})C$

$$\begin{aligned} &= ( \overline{A}\overline{B}'C + \overline{A}\overline{B}'BD + \overline{A}\overline{B}' ) C \\ &= \overline{A}\overline{B}'CC + 0 + \overline{A}\overline{B}'C \\ &= (\overline{A}+\overline{A}') \overline{B}'C \\ &= \overline{B}'C \end{aligned}$$

## Example (3)

■  $\bar{A}BC + A\bar{B}\bar{C} + \bar{A}\bar{B}\bar{C} + A\bar{B}C + ABC$

$$= A'BC + B'C'(A+A') + AC(B+B')$$

$$= A'BC + B'C' + AC$$

$$= (A'B + A)C + B'C'$$

$$= (A' + A)(B + A)C + B'C' \quad [\text{Distributive Law}]$$

$$= AC + BC + B'C'$$

## Example (4)

■  $\overline{(AB+AC)} + \bar{A}\bar{B}C$

$$\begin{aligned} &= (A'+B')(A'+C') + A'B'C \\ &= A' + A'C' + A'B' + A'B'C + B'C' \\ &= A' (1+C') + A'B' (1+C) + B'C' \\ &= A' + A'B' + B'C' \quad \text{Null law} \\ &= A' + B'C' \quad \text{Null law} \\ &= (A(B+C))' \end{aligned}$$



## Example (5)

$$X + \overline{X}Y = X + Y$$

■  $A\overline{C} + A\overline{B}C + ABCD + AB\overline{D}$

$$\begin{aligned} &= A[ (C' + C''B') + B(D' + D''C) ] \\ &= A[ (C' + B') + B(D' + C) ] \\ &= A[ C' + CB + B' + BD' ] \\ &= A[ C' + B + B' + D' ] \\ &= A[ C' + 1 + D' ] \\ &= A[ 1 + D' ] \\ &= A1 \\ &= A \end{aligned}$$

## Example (6)

$$X + \overline{X}Y = X + Y$$

- $(\overline{A} + \overline{B} + \overline{C})(\overline{B} + C)(A + \overline{B})$

$$\begin{aligned} &= (A' + B' + C') (AB' + AC + B' + B'C) \\ &= (A' + B' + C') (AC + AB' + B') \\ &= (A' + B' + C') (AC + B') \text{ [Distributive Law]} \\ &= 0 + A'B' + AB'C + B' + 0 + B'C' \\ &= B' (A' + AC + 1 + C') \\ &= B' (A' + C + 1) \\ &= B' (A' + 1) \\ &= B' \end{aligned}$$

# An Example on Programming

---

```
while (((A && B) || (A && !B)) || !A)
{
    // do something
}
```

=

```
while (1)
{
    // do something
}
```

# Boolean Algebra

---

- Through our exercises in simplifying Boolean expressions, we see that there are 1+ ways of stating the same Boolean expression.
  - These “synonymous” forms are *logically equivalent*.
  - Logically equivalent expressions could produce confusions

$$(X+Y) (X+\overline{Y}) (\overline{X}\overline{Z}) = XZ$$

- In order to eliminate the confusion, designers express Boolean expression in a unified and *standardized* form, called ***canonical form***.

# Boolean Algebra: Minterm and Maxterm

---

- Some books uses sum-of-minterms form and product-of-maxterms form
  - A minterm is a logical expression of  $n$  variables that employs only the **complement** operator and the **product** operator.
    - For example,  $abc$ ,  $ab'c$  and  $abc'$  are 3 minterms for a Boolean function of the three variables  $a$ ,  $b$ , and  $c$ .
  - A maxterm is a logical expression of  $n$  variables that employs only the **complement** operator and the **sum** operator.

# Logic Reduction: Minterm / Maxterm

## - Example using Karnaugh (K) map

Out = ABC  
Minterm = ABC  
Numeric = 1 1 1

BC	00	01	11	10
A				
0	0	0	0	0
1	0	0	1	0

Out = ABC

Out =  $\bar{A}\bar{B}\bar{C}$   
Minterm =  $\bar{A}\bar{B}\bar{C}$   
Numeric = 0 1 0

BC	00	01	11	10
A				
0	0	0	0	1
1	0	0	0	0

Out =  $\bar{A}\bar{B}\bar{C}$

Out =  $\bar{A}\bar{B}\bar{C} + ABC$

BC	00	01	11	10
A				
0	0	0	0	1
1	0	0	1	0

Numeric = 0 1 0      1 1 1  
Minterm =  $\bar{A}\bar{B}\bar{C}$       ABC  
Out =  $\bar{A}\bar{B}\bar{C} + ABC$

Out = (A + B + C)  
Maxterm = A + B + C  
Numeric = 1 1 1  
Complement = 0 0 0

BC	00	01	11	10
A				
0	0	1	1	1
1	1	1	1	1

Out = (A + B + C)(A + B +  $\bar{C}$ )  
Maxterm = (A + B + C)      Maxterm = (A + B +  $\bar{C}$ )  
Numeric = 1 1 1      Numeric = 1 1 0  
Complement = 0 0 0      Complement = 0 0 1

BC	00	01	11	10
A				
0	0	0	1	1
1	1	1	1	1

Out = ( $\bar{A} + \bar{B} + \bar{C}$ )  
Maxterm =  $\bar{A} + \bar{B} + \bar{C}$   
Numeric = 0 0 0  
Complement = 1 1 1

BC	00	01	11	10
A				
0	1	1	1	1
1	1	1	0	1

Out = (A + B + C)(A + B +  $\bar{C}$ )

BC	00	01	11	10
A				
0	0	0	1	1
1	1	1	1	1

A B C = 0 0 x  
Complement = 1 1 x  
Sum-term = (A + B)  
Out = (A + B)

# Boolean Algebra

---

- There are two canonical forms for Boolean expressions: *sum-of-products* and *product-of-sums*.
  - Boolean product ( $\times$ )  $\rightarrow$  **AND**  $\rightarrow$  logical conjunction operator
  - Boolean sum ( $+$ )  $\rightarrow$  **OR**  $\rightarrow$  logical disjunction operator
- In the *sum-of-products form*, ANDed variables are *ORed together*.
  - For example:  $F(x, y, z) = xy + xz + yz$
- In the *product-of-sums form*, ORed variables are *ANDed together*.
  - For example:  $F(x, y, z) = (x+y)(x+z)(y+z)$

# Create Canonical Form Via Truth Table

- It is easy to convert a function to **sum-of-products** form from its truth table.
- We only interested in the production of the inputs which yields **TRUE** (=1).
  - We first **highlight** the lines that result in 1.
  - Then, we **group** them together with OR.

$$F(x, y, z) = x\bar{z} + y$$

x	y	z	$x\bar{z} + y$
0	0	0	0
0	0	1	0
0	1	0	1
0	1	1	1
1	0	0	1
1	0	1	0
1	1	0	1
1	1	1	1



# Create Canonical Form Via Truth Table ('Cont)

- Look at this example:

$$F(x, y, z) = x\bar{z} + y$$

$$= (\bar{x}\bar{y}\bar{z}) + (\bar{x}yz) + (x\bar{y}\bar{z}) + (x\bar{y}z)$$

- It may not be the simplest form. But, it is the standard **sum-of-products canonical form**

$$F(x, y, z) = x\bar{z} + y$$

	x	y	z	$x\bar{z} + y$
	0	0	0	0
	0	0	1	0
$(\bar{x}\bar{y}\bar{z})$	0	1	0	1
$(\bar{x}yz)$	0	1	1	1
$(x\bar{y}\bar{z})$	1	0	0	1
	1	0	1	0
$(x\bar{y}z)$	1	1	0	1
$(xyz)$	1	1	1	1

$$X + \overline{X}Y = X + Y$$

## Exercise

---

□ **Convert**  $ABC + A'BC + AB'C + A'B'C + ABC'$   
**to** its **simplest form**

$$\begin{aligned} ABC + A'BC + AB'C + A'B'C + ABC' &= BC(A + A') + B'C(A + A') + ABC' \\ &= BC1 + B'C1 + ABC' \\ &= C(B + B') + ABC' \\ &= C + ABC' \\ &= C + AB \end{aligned}$$

Distributivity

$$\begin{aligned} C + AB &= (C + AB)(C + C') \\ &= C + 0 + ABC + ABC' = C(1 + AB) + ABC' \\ &= C1 + ABC' = C + ABC' \end{aligned}$$

# Exercise

□ **Convert**  $AB + C$  **to** the **sum-of-products form**

$$\begin{aligned}
 AB &= AB \cdot 1 && \text{By Th4} \\
 &= AB \cdot (C + C') && \text{By Th 15} \\
 &= ABC + ABC' && \text{By distributive law} \\
 &= CBA + C'BA && \text{By associative law}
 \end{aligned}$$

$$\begin{aligned}
 C &= C \cdot 1 && \text{By Th4} \\
 &= C \cdot (A + A') && \text{By Th15} \\
 &= CA + CA' && \text{By distributive law} \\
 &= CA \cdot 1 + CA' \cdot 1 && \text{By Th4} \\
 &= CA \cdot (B + B') + CA' \cdot (B + B') && \text{By Th15} \\
 &= CAB + CAB' + CA'B + CA'B' && \text{By distributive law} \\
 &= CBA + CBA' + CB'A + CB'A' && \text{By associative law}
 \end{aligned}$$

$$\begin{aligned}
 AB+C &= (CBA + C'BA) + (CBA + CBA' + CB'A + CB'A') \\
 &= CBA + CBA' + CB'A + CB'A' + C'BA
 \end{aligned}$$

$$ABC + A'BC + AB'C + A'B'C + ABC'$$

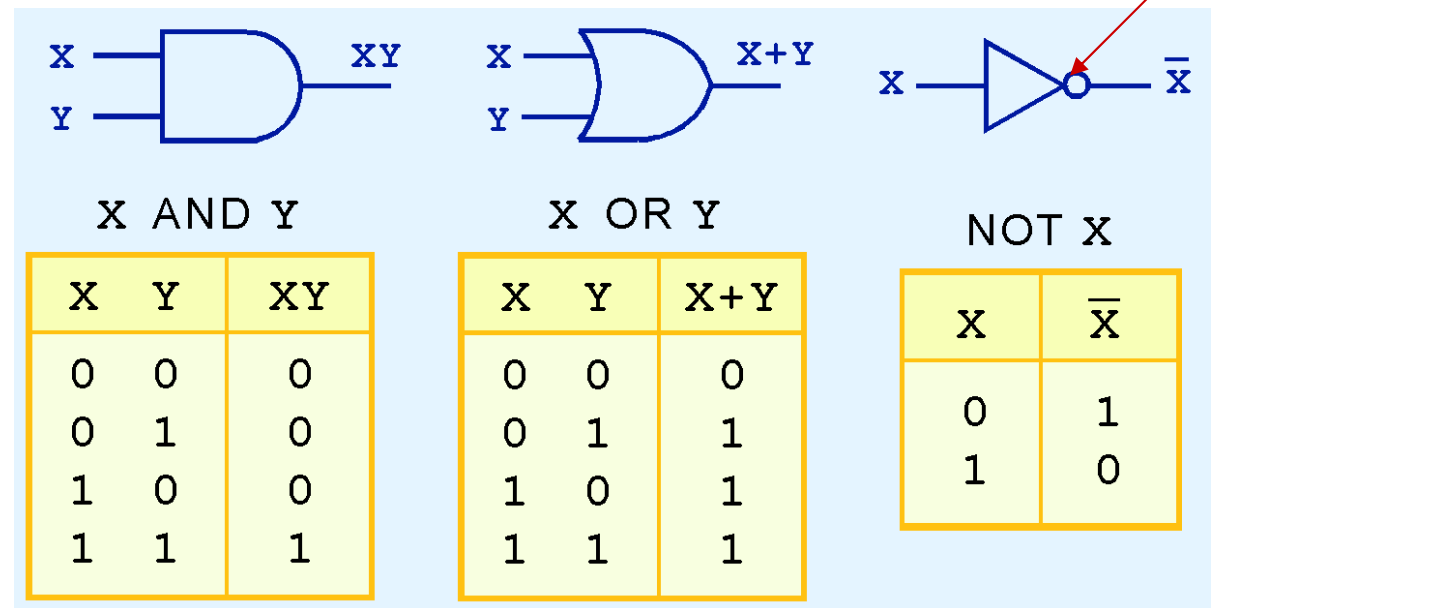
## 3.3 Logic Gates

---

- We've seen Boolean functions in abstract terms.
- You may still ask:
  - *How could Boolean function be used in computer?*
- In reality, Boolean functions are implemented as digital circuits, which called *Logic Gates*.
- A logic gate is an **electronic device** that produces a result based on input values.
  - A logic gate may contain multiple transistors, but, we think them as **one integrated unit**.
  - **Integrated circuits** (IC) contain collections of gates, for a particular purpose.

# AND, OR, and NOT Gates

- Three simplest gates are the **AND**, **OR**, and **NOT** gates.



- Their symbol and their truth tables are listed above.

# NAND and NOR Gates

□ **NAND** and **NOR** are two additional gates.

■ Their symbols and truth tables are shown on the right.

□ **NAND = NOT AND**

□ **NOR = NOT OR**

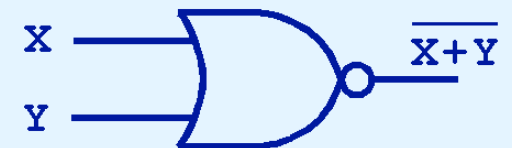
X NAND Y

X	Y	X NAND Y
0	0	1
0	1	1
1	0	1
1	1	0



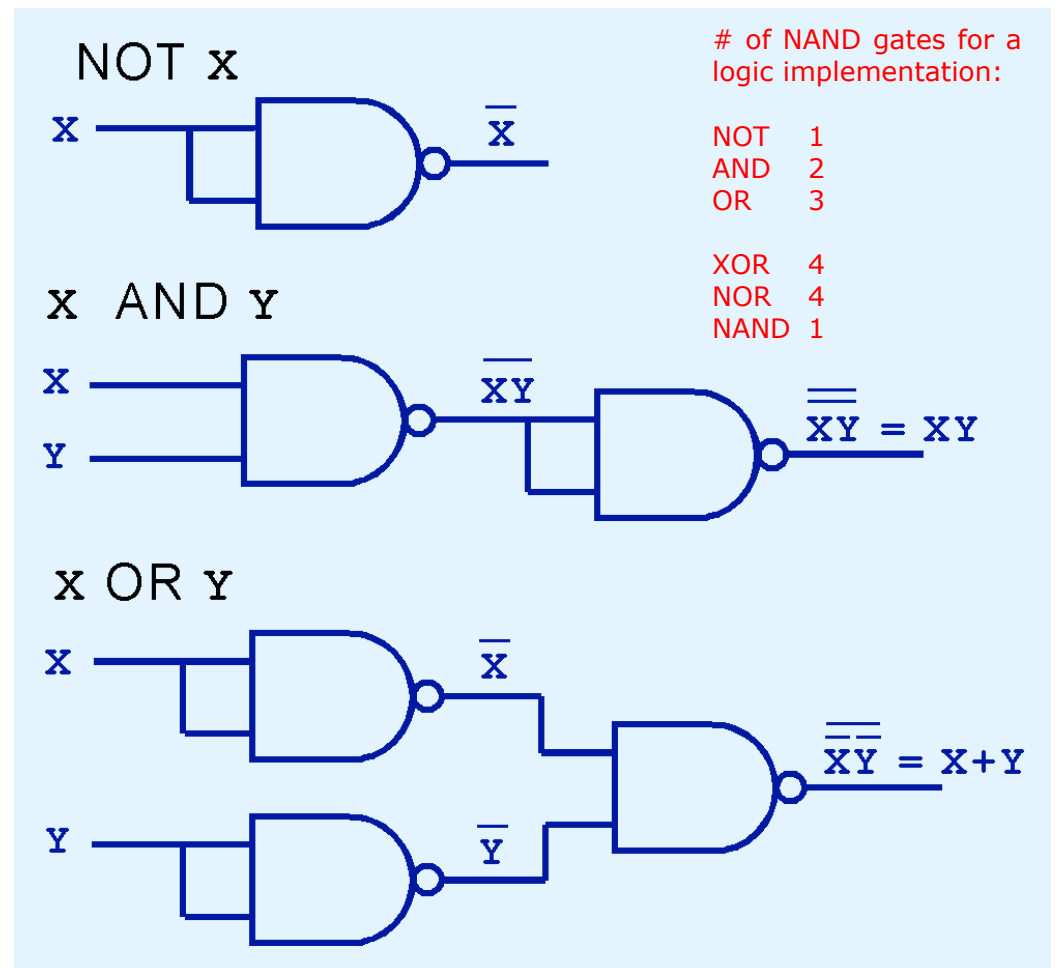
X NOR Y

X	Y	X NOR Y
0	0	1
0	1	0
1	0	0
1	1	0



# The Application of NAND and NOR Gates

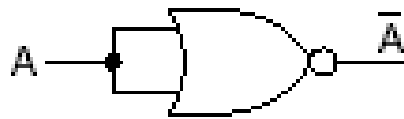
- NAND and NOR are known as universal gates! – gates of all gates
  - They are inexpensive to produce
- More important: Any Boolean function can be constructed using only **NAND** or only **NOR** gates.



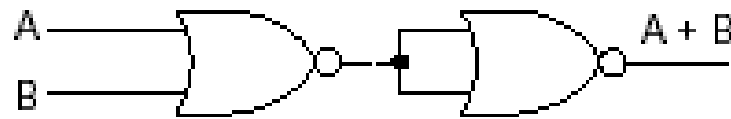
# The Universal Gates: NOR Gate

---

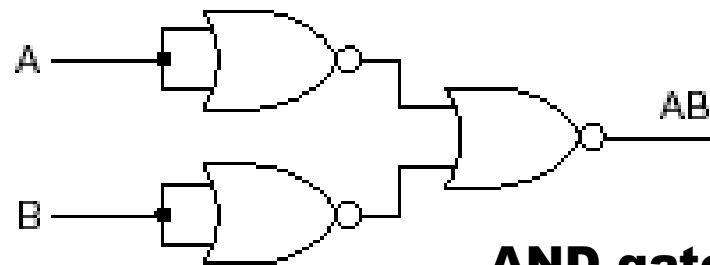
- Using **NOR** gate to construct **AND**, **OR**, and **NOT** gates



**Not gate**



**OR gate**

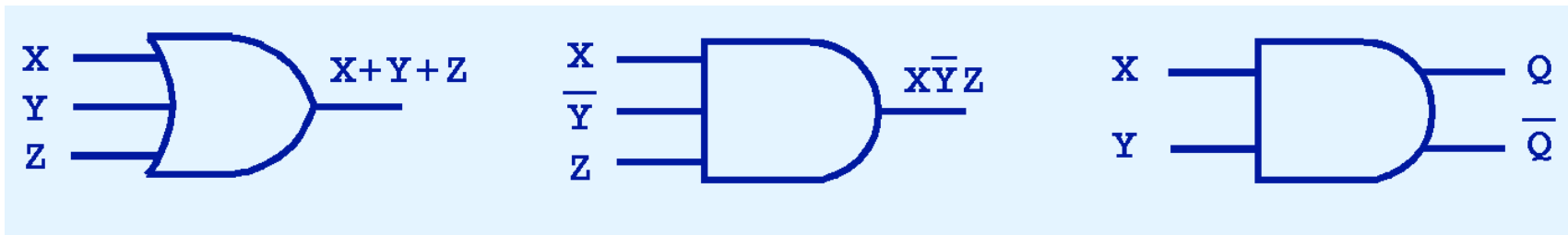


**AND gate**



# Multiple Inputs and Outputs of Gates

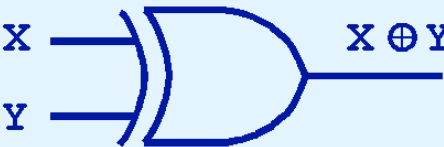
- The gates could have **multiple inputs** and/or **multiple outputs**.
  - The second output can be provided as the **complement** of the first output.
  - We'll see more integrated circuits, which have multiple inputs/outputs.



# XOR Gates

- Another very useful gate is the **Exclusive OR (XOR)** gate.
- The output of the XOR operation is true (**1**) only when the values of inputs are different.

X XOR Y		
X	Y	$X \oplus Y$
0	0	0
0	1	1
1	0	1
1	1	0



$$\begin{aligned}x \text{ XOR } y \\&= x' y + x y' \\&= (x+y) (x'+y')\end{aligned}$$

- The symbol for XOR is  $\oplus$

# Parity generator / checker

---

- ❑ Electrical noise in the transmission of binary information can cause errors
  - ❑ Parity can detect these types of errors
  - ❑ Parity systems
    - Odd parity
    - Even parity
  - ❑ Add a bit to the binary information
-

# Even parity check

---

## □ Even parity check

## □ Example: input: A(7...0), Output: even\_parity bit

- If there are even numbers of 1 in A, even\_parity = '0',
- If there are odd numbers of 1 in A, even\_parity = '1'

e.g., A = "10100001",

even\_parity = '1'

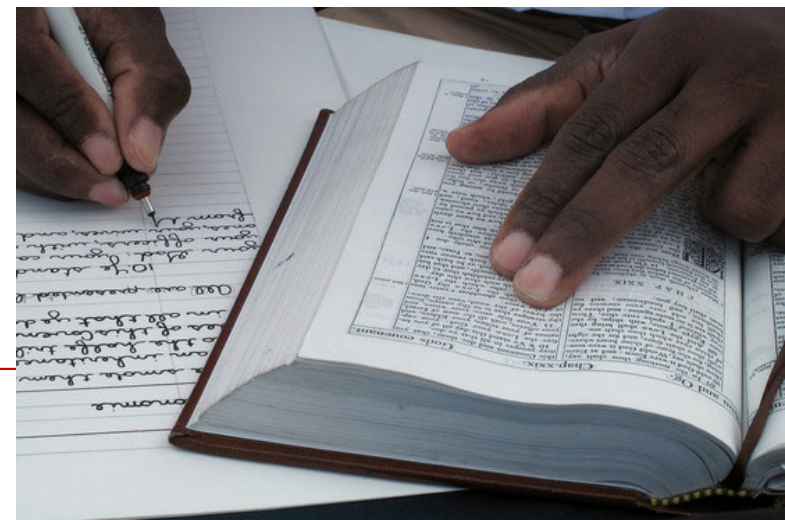
A = "10100011",

even\_parity = '0'

---

# Odd parity check

---



❑ Odd parity check

❑ Example: input:  $A(7..0)$ , Output: odd\_parity bit

- If there are odd numbers of 1 in  $A$ ,  
odd\_parity = '0',
- If there are even numbers of 1 in  $A$ ,  
odd\_parity = '1'

e.g.,  $A = "10100001"$ ,

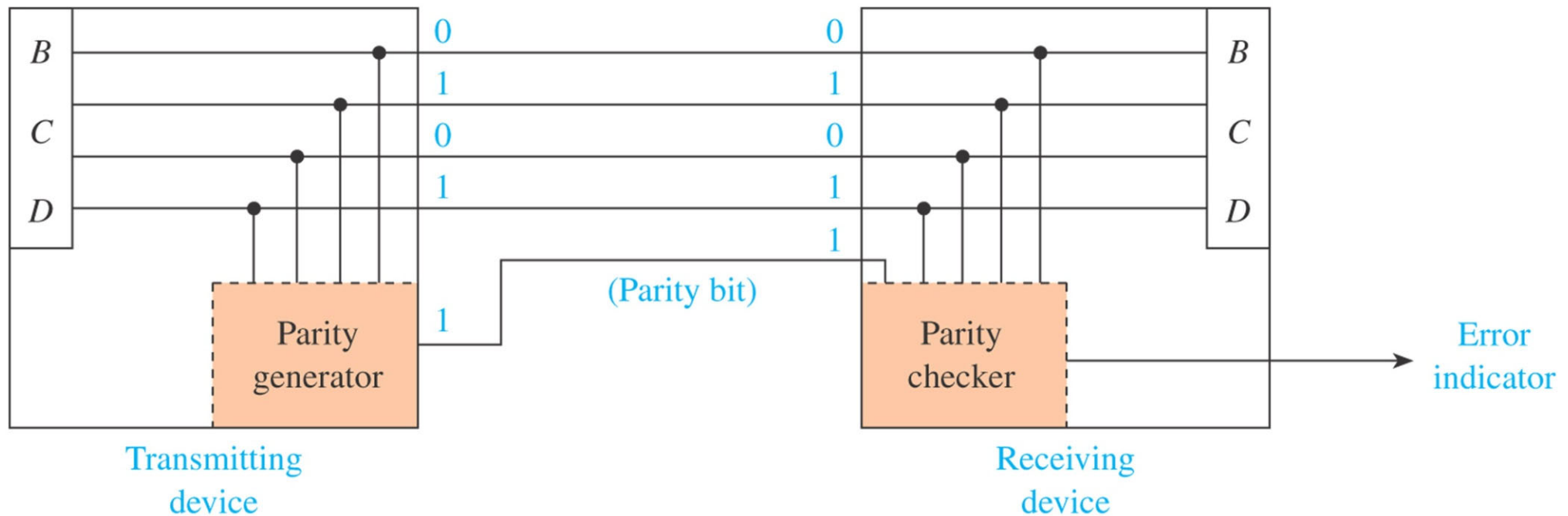
odd\_parity = '0'

$A = "10100011"$ ,

odd\_parity = '1'

# Odd-parity generator/checker system

---



# Error detection

---

- ❑ **Transmitting end:** The parity generator creates the parity bit.
- ❑ **Receiving end:** The parity checker determines if the parity is correct.
- ❑ e.g., odd-parity check of 8-bit data
  - Data send: 10111101 + 1
  - Data received: 101011011odd-parity check: The number of 1 is even → *error*

# Discussion point

---

- ❑ What are disadvantages of even parity (or odd parity) check to detect transmission errors? Consider the following case:
  - Protocol: 8-bit plus one even parity bit
  - Information sent: 11011100 + 1
  - Information received: 10010100 + 1
- ❑ The parity generator/checker system detects only errors that occur to 1 bit.



# Parity check using XOR

---

- $N-1$  XOR gates can be cascaded to form a circuit with  $N$  inputs and a single output
  - *even-parity circuit.*
  - Example:  $N=8$ , Inputs=10111101, *even-parity output*  
$$=((1\oplus 0)\oplus (1\oplus 1))\oplus ((1\oplus 1)\oplus (0\oplus 1))=0$$
- Odd-parity check circuit: *even-parity check circuit*  $\rightarrow$  Inverted  $\rightarrow$  Odd-parity check
  - Example:  $N=8$ , Inputs=10111101, *odd-parity output*  
$$=\text{NOT}(((1\oplus 0)\oplus (1\oplus 1))\oplus ((1\oplus 1)\oplus (0\oplus 1)))=1$$

# Two Types of Logic Circuits

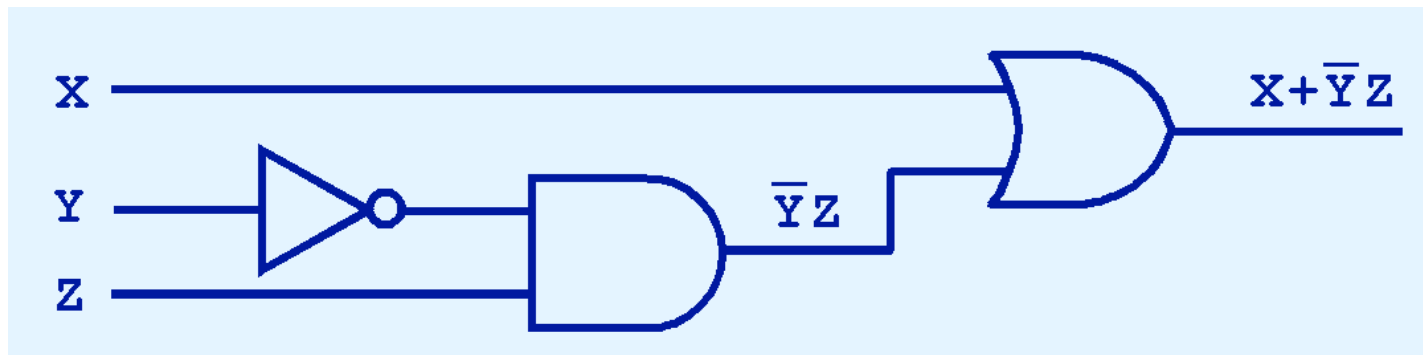
---

- Combinational Logic Circuit (*CLC*)
  - *Good at designing computational components in the CPU, such as ALU*
- Sequential Logic Circuit (*SLC*)
  - *Good at designing memory components, such as registers and memory*

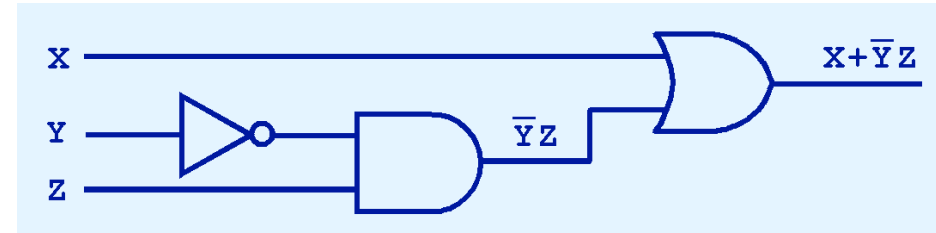
# Logic Gates

- We use the combination of gates to implement Boolean functions.
- The circuit below implements the Boolean function:

$$F(X, Y, Z) = X + \bar{Y}Z$$



# 3.5 Combinational Circuits



- The circuit implements the Boolean function:

$$F(X, Y, Z) = X + \bar{Y}Z$$

- The major characteristics of this kind of circuits:

- The circuit produces an output almost immediately after the inputs are given.

- This kind of circuits are called **combinational logic circuit (CLC)**.

- In a later section, we will explore circuits where this is not the case.

# Simplify *CLC via* Boolean Algebra

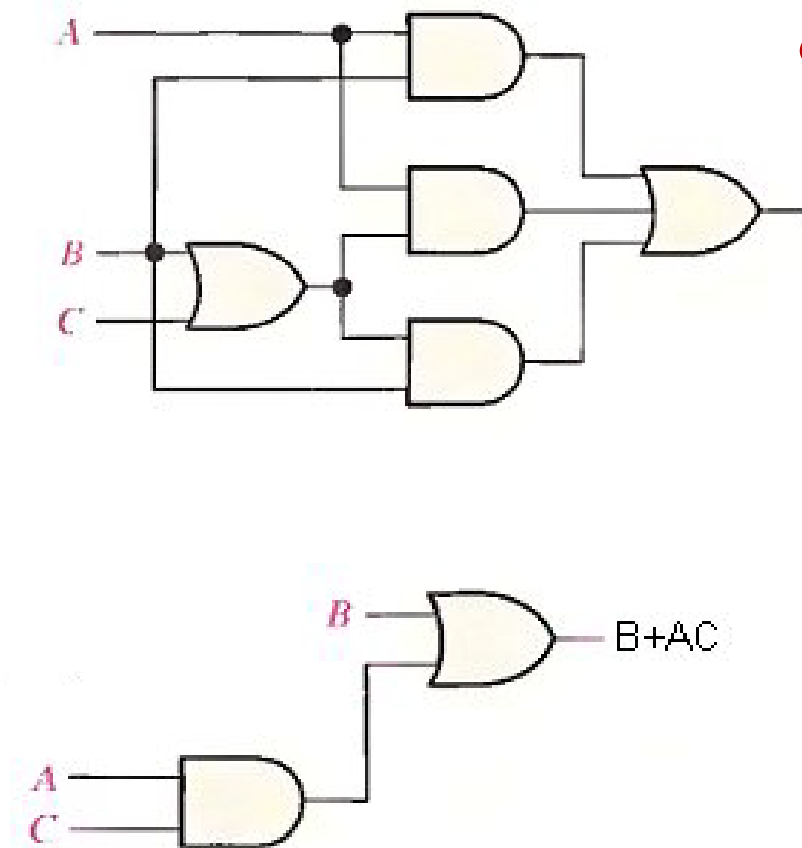
---

- As I have mentioned previously:
  - The **simpler** that we can express a Boolean function, the **smaller** the circuit will be constructed.
  - Simpler circuits are ***cheaper*** → consume ***less power*** → run ***faster*** than complex circuits.
- We always want to reduce a Boolean function to its ***simplest*** form.
- It is important to simplify combinational logic circuit via Boolean algebra laws

# Simplify *CLC* via Boolean Algebra

Can we simplify this circuit? If yes, then how?

□ Look at this example



$$\begin{aligned} & AB + A(B + C) + B(B + C) \\ &= AB + AB + AC + BB + BC \\ &= AB + AB + AC + B + BC \\ &= AB + AC + B + BC \\ &= AB + AC + B \\ &= B + AC \end{aligned}$$

# Steps to Simplify a Complex Circuit

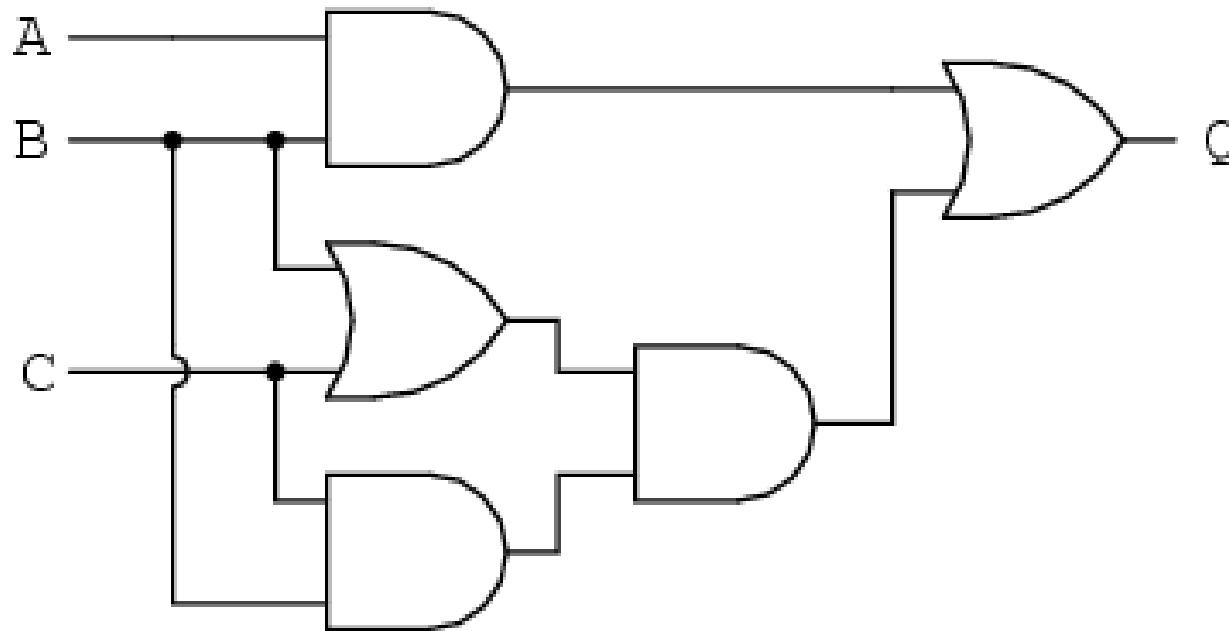
---

- From this example, we know that the basic steps to simplify a complex circuit is the following:
  - **Step1: Express** a logical circuit into a Boolean expression
  - **Step2: Simplify** the Boolean expression as much as possible
  - **Step3: Re-express** the simplified expression back to a circuit.

# Example of Simplify a Logical Circuit

---

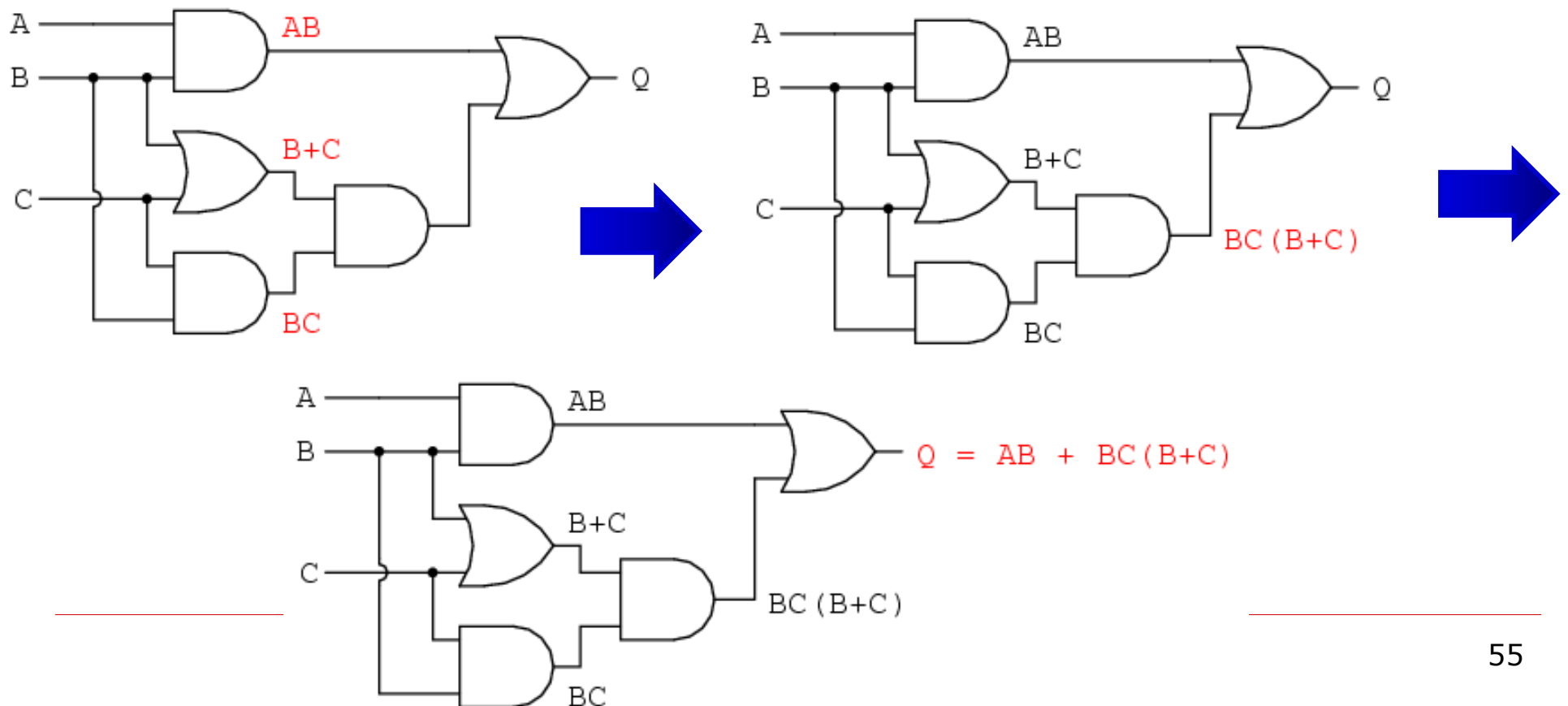
- Simplify the following circuit





# Example of Simplify a Logical Circuit

- **Step1:** Express a logical circuit into a Boolean expression



# Example of Simplify a Logical Circuit

---

- **Step2:** Simplify the Boolean expression as much as possible

$$AB + BC(B + C)$$



Distributing terms

$$AB + BBC + BCC$$



Applying identity  $AA = A$   
to 2nd and 3rd terms

$$AB + BC + BC$$



Applying identity  $A + A = A$   
to 2nd and 3rd terms

$$AB + BC$$



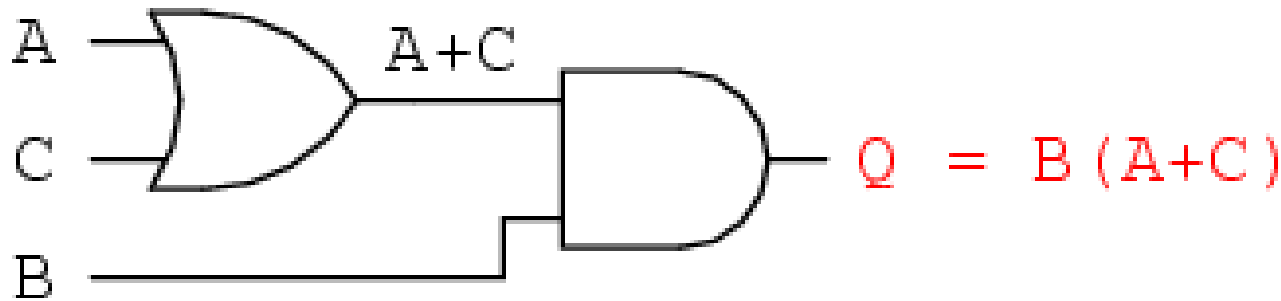
Factoring **B** out of terms

$$B(A + C)$$

# Example of Simplify a Logical Circuit

---

- **Step3:** Re-express the simplified expression back to a circuit



Obviously, the simplified circuit is much simpler than the original one

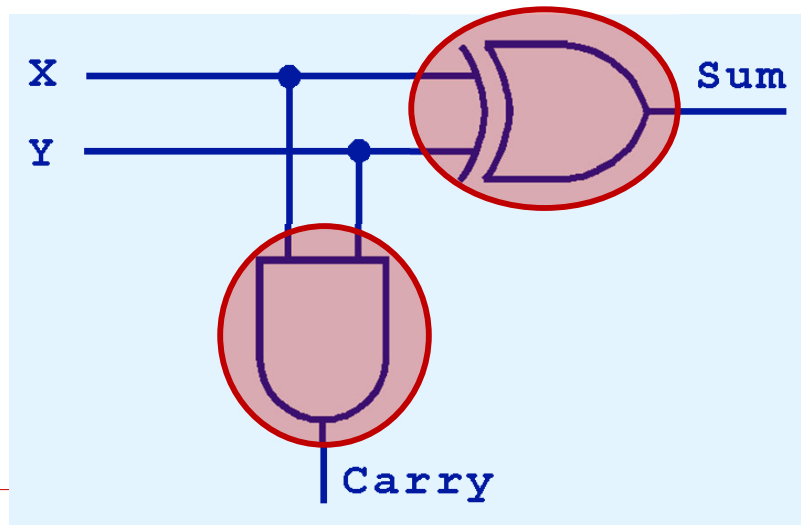
# Combinational Circuits: *Half Adder*

- Combinational logic circuits can be used to create many useful devices.
- *Half Adder*: Compute the sum of two bits.
- Let's gain some insight of how to construct a half adder by looking at its truth table on the right.

Inputs		Outputs	
X	Y	Sum	Carry
0	0	0	0
0	1	1	0
1	0	1	0
1	1	0	1

# Combinational Circuits: *Half Adder* ('Cont)

- It consists two gates:
  - a **XOR** gate -- the sum bit
  - a **AND** gate -- the carry bit



Inputs		Outputs	
X	Y	Sum	Carry
0	0	0	0
0	1	1	0
1	0	1	0
1	1	0	1

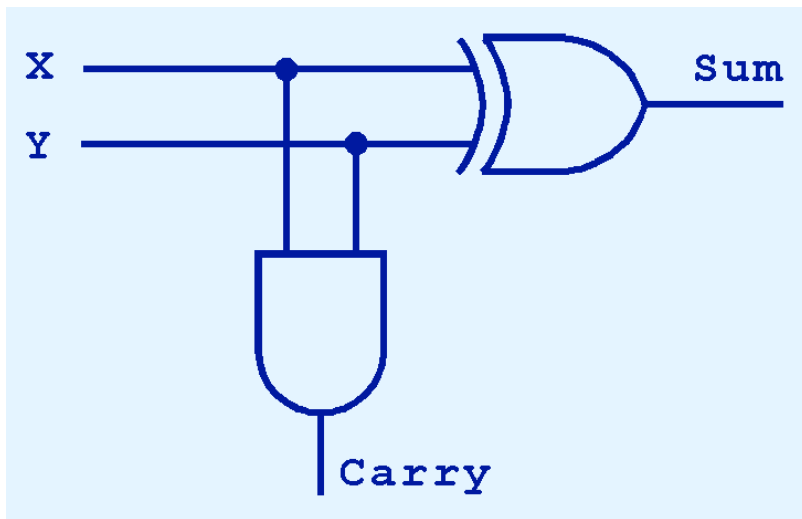
# Combinational Circuits: *Full Adder*

- We can extend the half adder to a full adder, which includes an **additional carry bit (Carry In)**
- The truth table for a full adder is shown on the right.

Inputs			Outputs	
X	Y	Carry In	Sum	Carry Out
0	0	0	0	0
0	0	1	1	0
0	1	0	1	0
0	1	1	0	1
1	0	0	1	0
1	0	1	0	1
1	1	0	0	1
1	1	1	1	1

# Half Adder → Full Adder ?

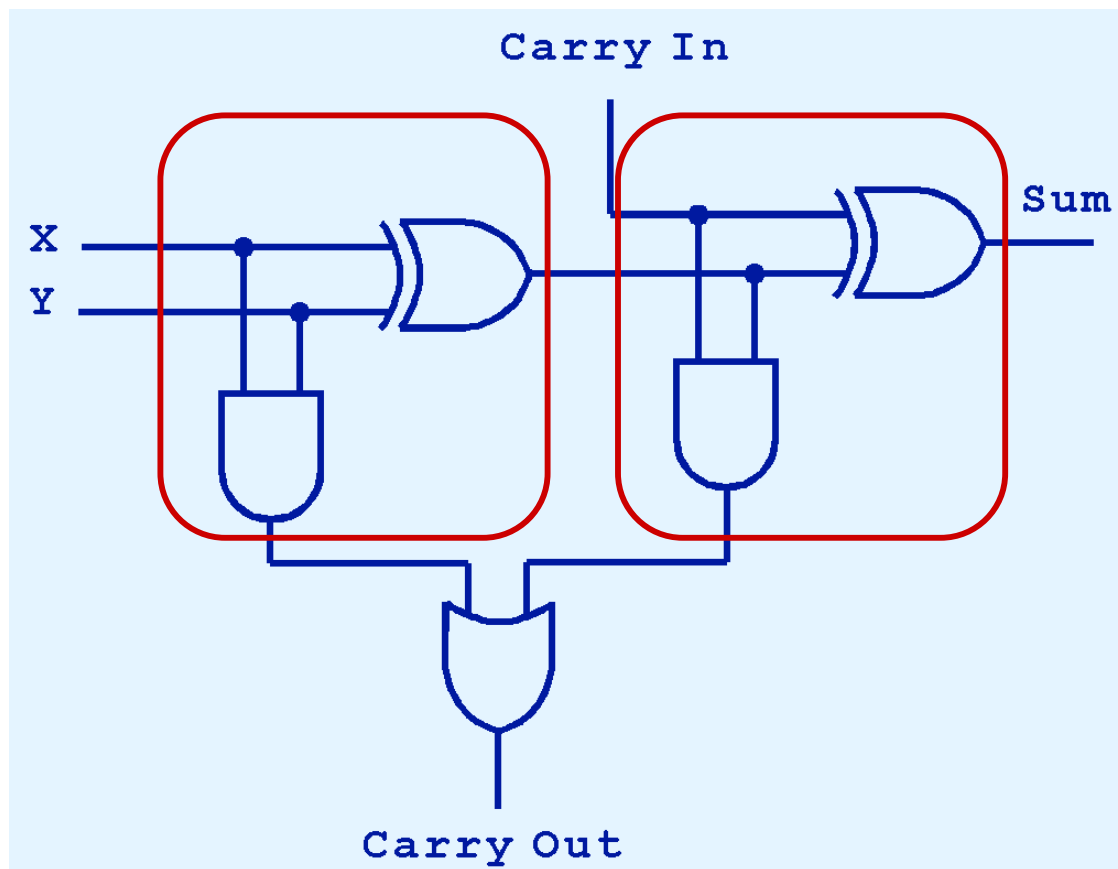
- How can we extend the **half adder** to a **full adder**?



Inputs			Outputs	
X	Y	Carry In	Sum	Carry Out
0	0	0	0	0
0	0	1	1	0
0	1	0	1	0
0	1	1	0	1
1	0	0	1	0
1	0	1	0	1
1	1	0	0	1
1	1	1	1	1

- Hint: First calculate  $X + Y$  by a half adder, then the sum adds the carry in bit, then.....

# The Full Adder

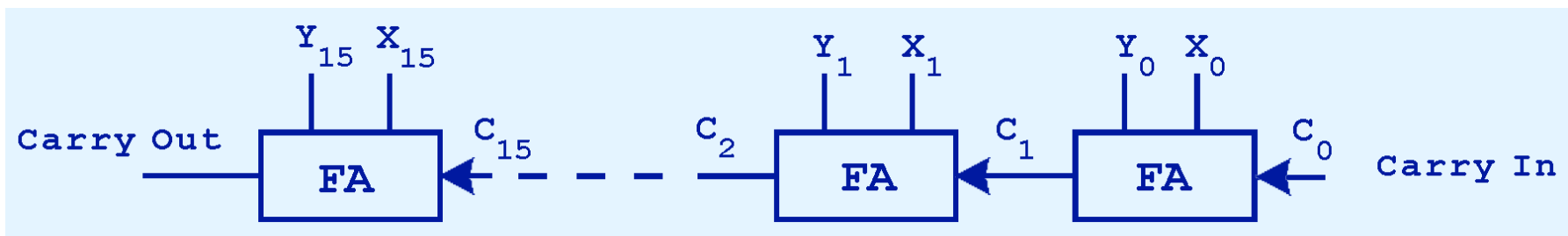


Inputs			Outputs	
X	Y	Carry In	Sum	Carry Out
0	0	0	0	0
0	0	1	1	0
0	1	0	1	0
0	1	1	0	1
1	0	0	1	0
1	0	1	0	1
1	1	0	0	1
1	1	1	1	1



# Ripple-carry Adder

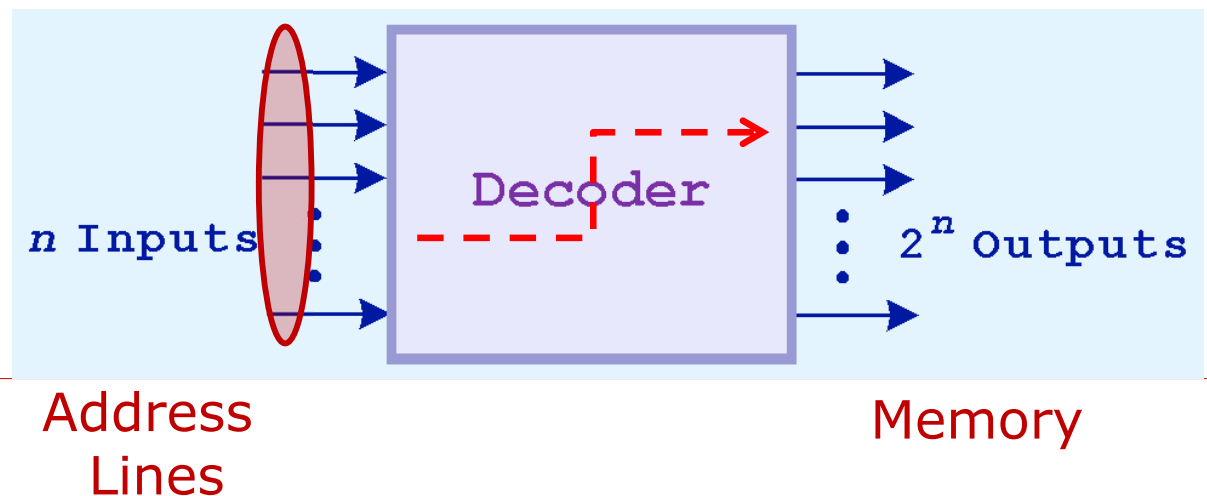
- Just as we combined half adders to construct a full adder, full adders can be connected in series.
- The carry bit “ripples” from one adder to the next. This configuration is called a **ripple-carry adder**.



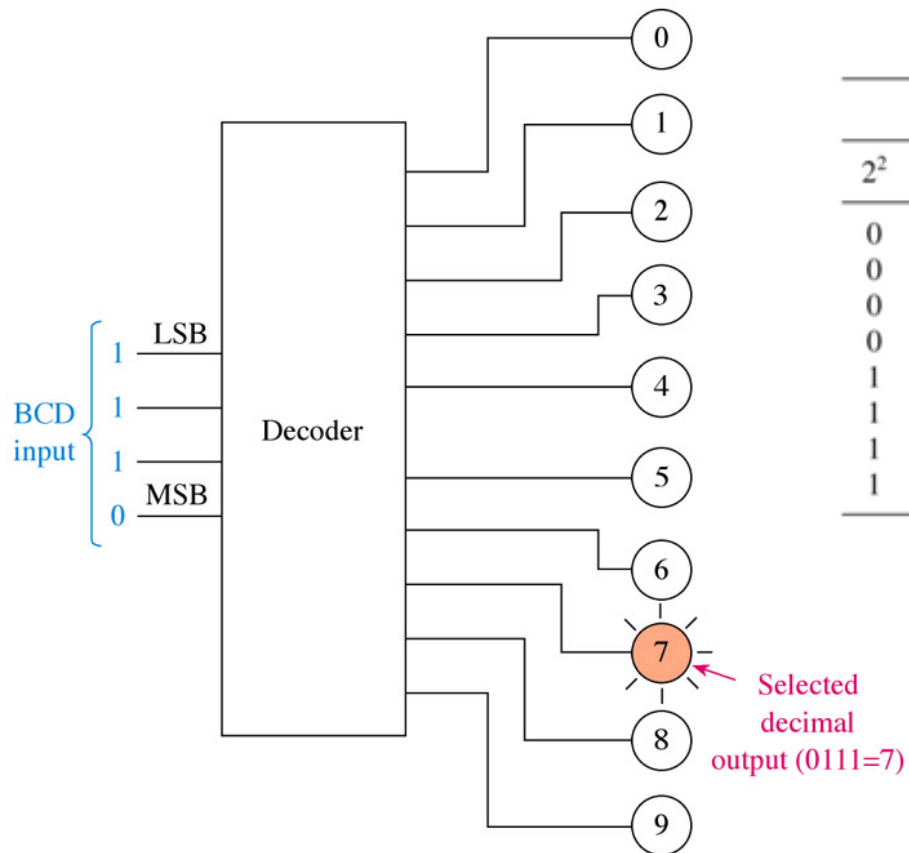
- This is the full adder for two 16 bits!

# Decoder

- ❑ **Decoder** is another important combinational circuit.
- ❑ It is used to **select a memory location** according a address in binary form
  - Application: given a memory address → Obtain its memory content.
- ❑ Address decoder with  $n$  inputs can select one out of  $2^n$  locations.



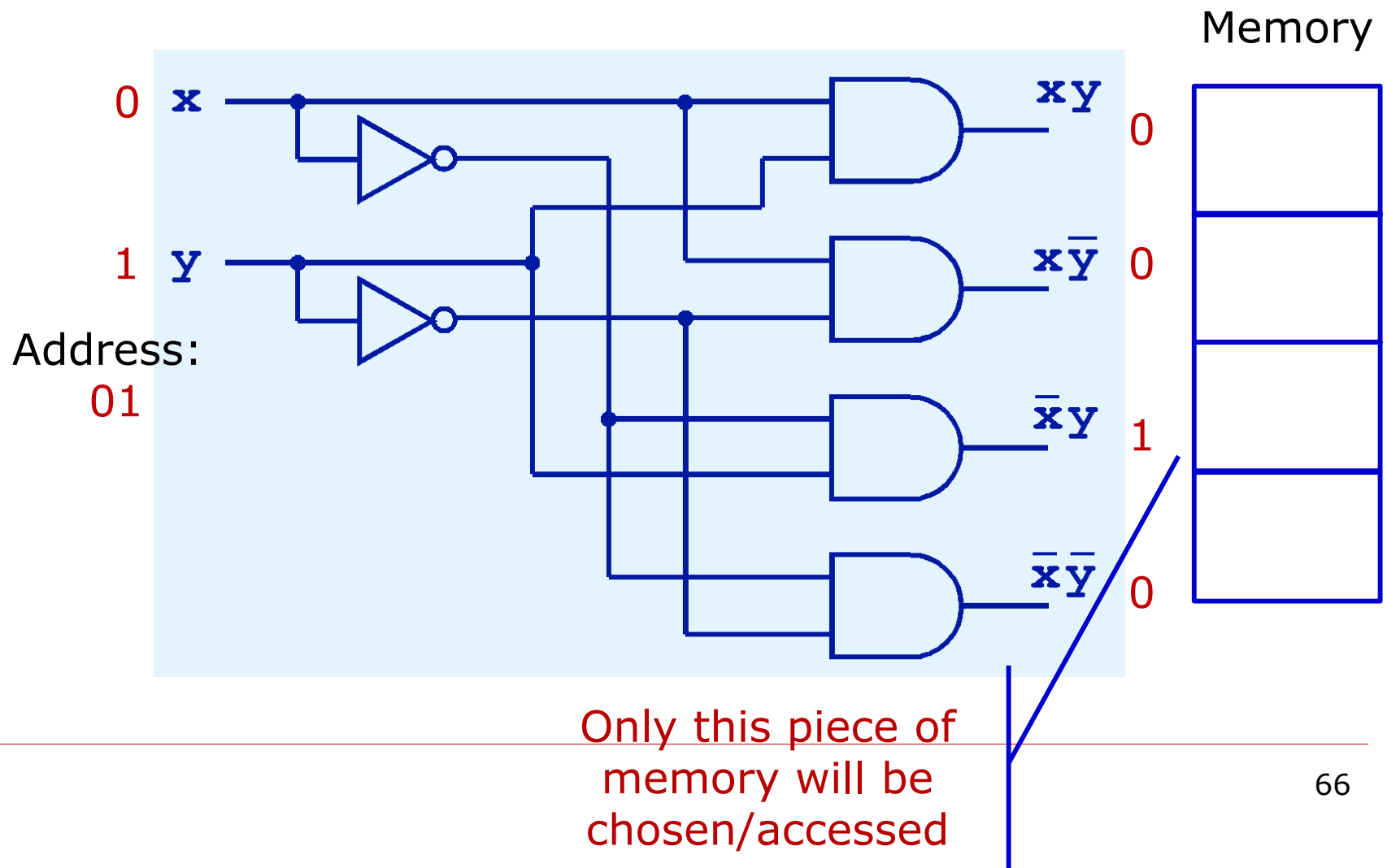
# Decoder

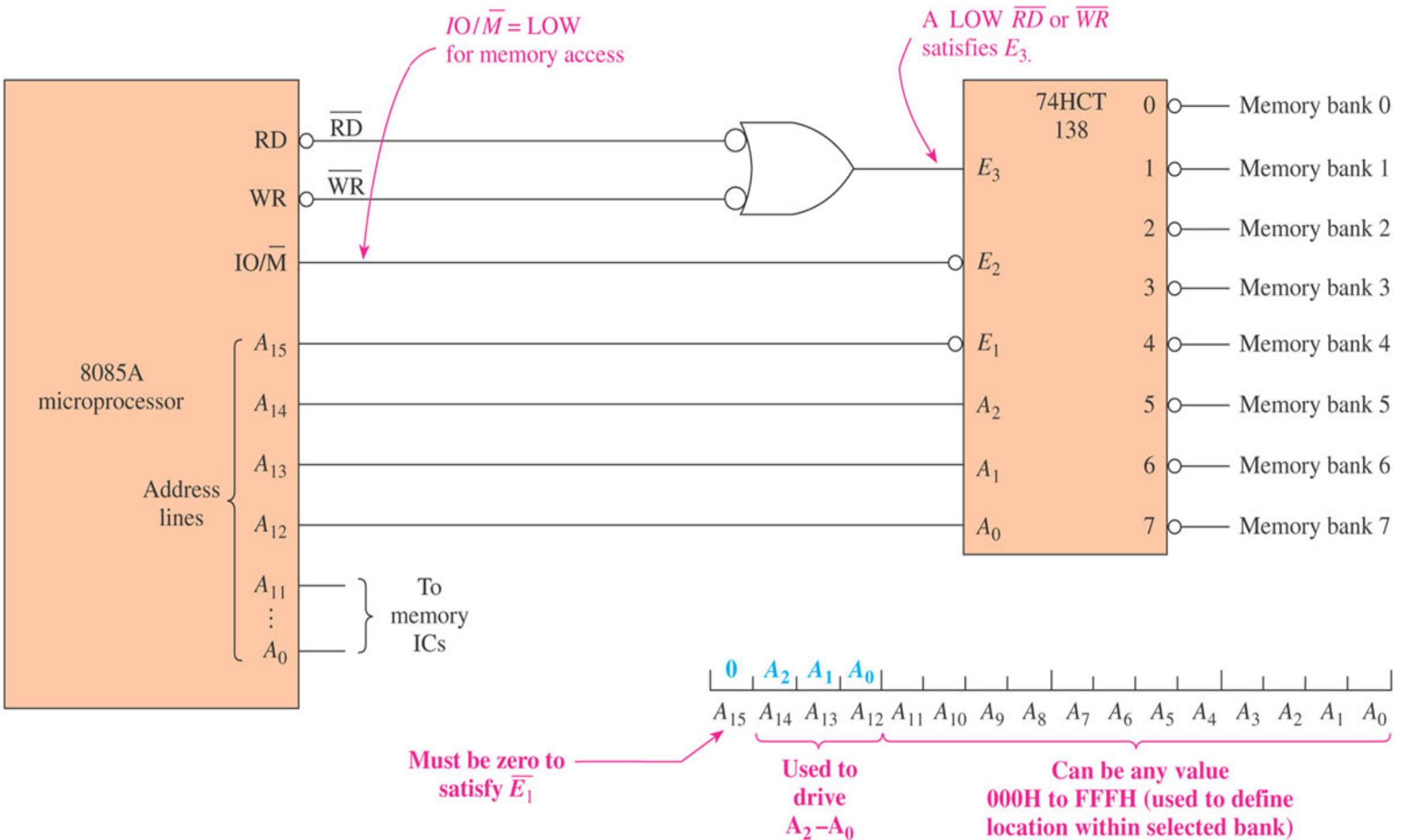


Input			Output							
$2^2$	$2^1$	$2^0$	0	1	2	3	4	5	6	7
0	0	0	1	0	0	0	0	0	0	0
0	0	1	0	1	0	0	0	0	0	0
0	1	0	0	0	1	0	0	0	0	0
0	1	1	0	0	0	1	0	0	0	0
1	0	0	0	0	0	0	1	0	0	0
1	0	1	0	0	0	0	0	1	0	0
1	1	0	0	0	0	0	0	0	1	0
1	1	1	0	0	0	0	0	0	0	1

# A 2-to-4 Decoder

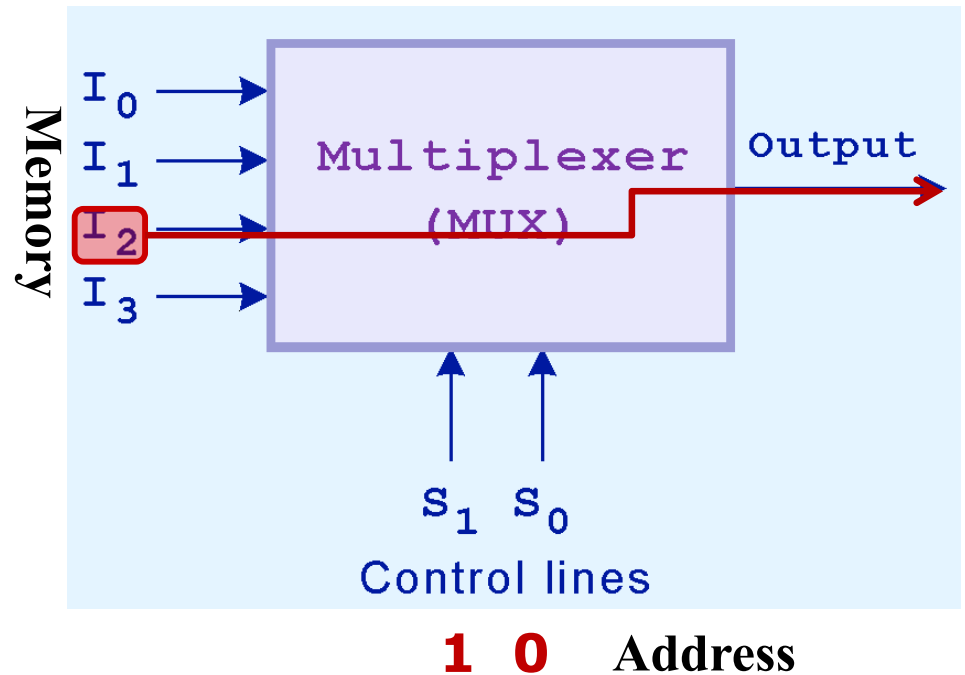
□ This is a 2-to-4 decoder :



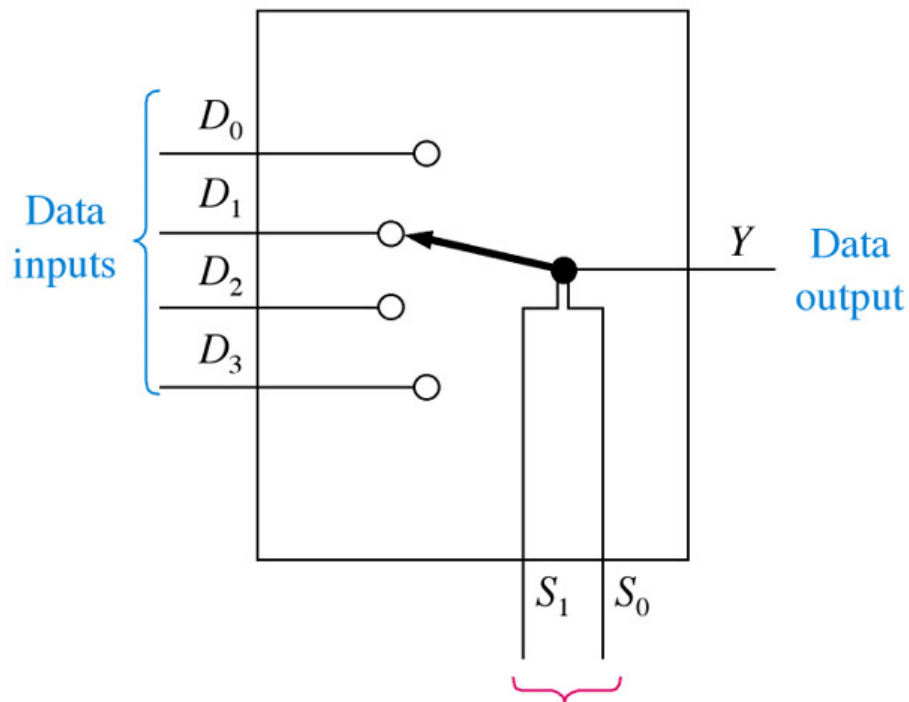


# Multiplexer

- ❑ A multiplexer works just **the opposite** to a decoder.
- ❑ It selects a single value from multiple inputs.
- ❑ The chosen input for output is determined by the value of the multiplexer's control lines.
- ❑ To select from  **$n$**  inputs,  **$\log_2 n$**  control lines are required.



# A four-line multiplexer



Data select control input determines which data input is connected to the output

**Table 8–5** Data Select Input Codes for Figure 8–30

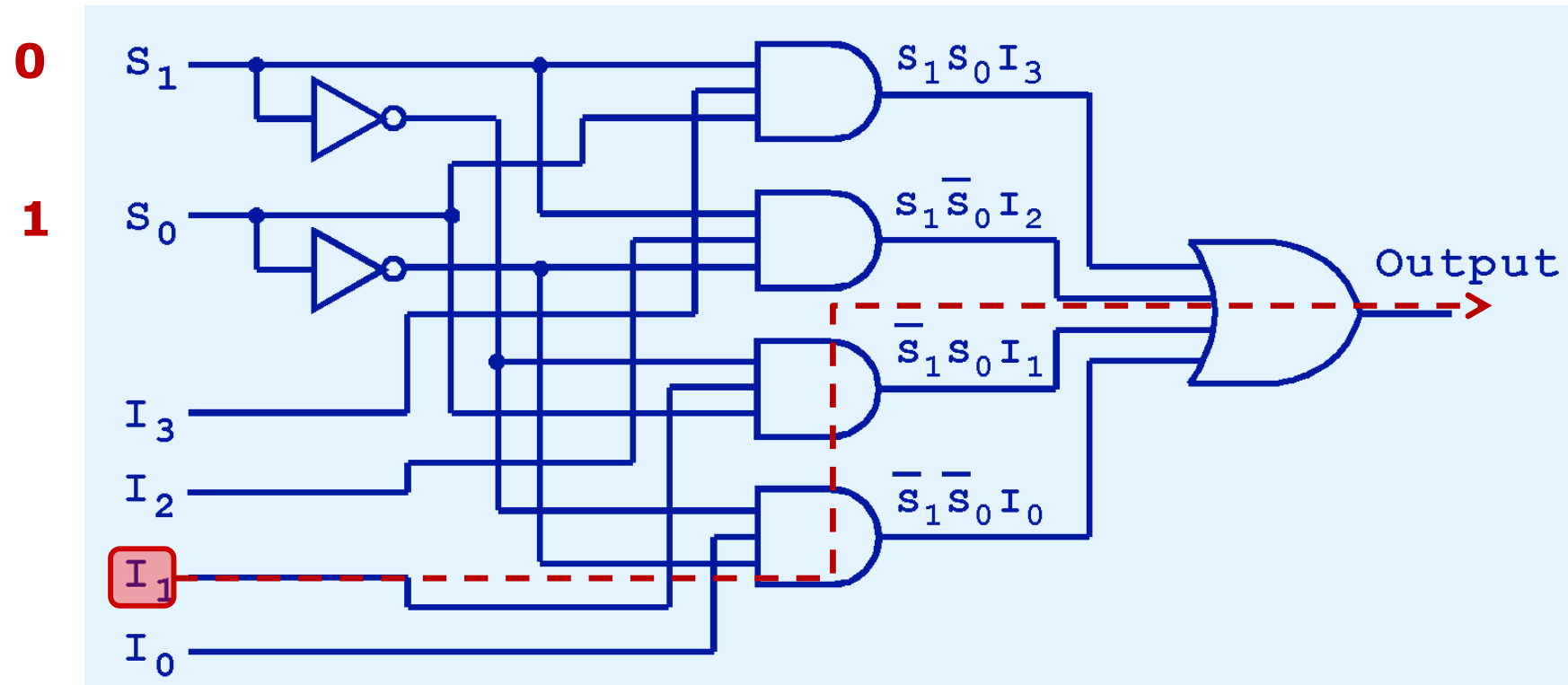
Data Select Control Inputs		Data Input Selected
$S_1$	$S_0$	
0	0	$D_0$
0	1	$D_1$
1	0	$D_2$
1	1	$D_3$

What is the logic equation for the output  $Y$  =?

$$Y = S_1' S_0$$

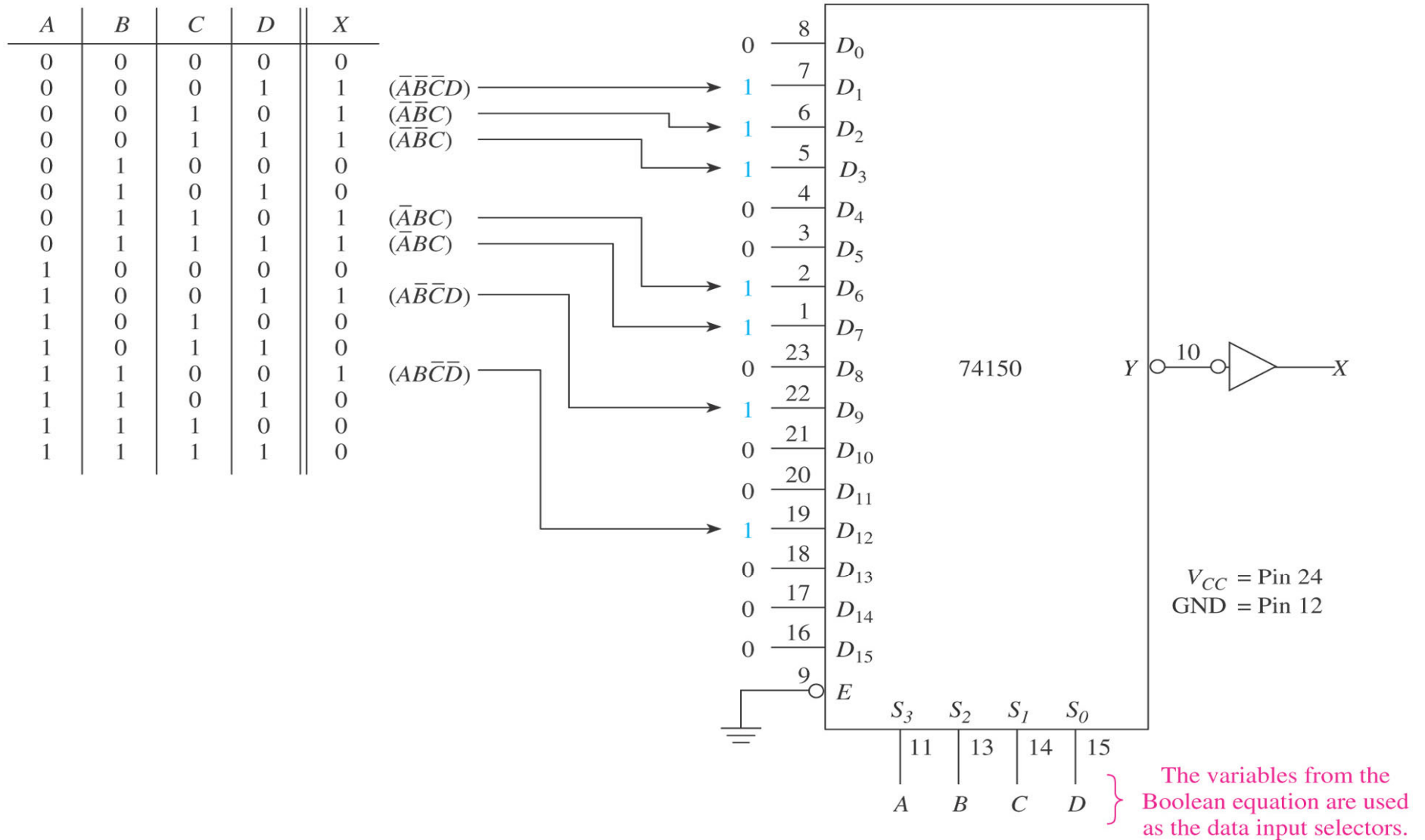
# Combinational Circuits

- This is a 4-to-1 multiplexer.



which input is transferred to the output?





Using a multiplexer to implement the Boolean equation

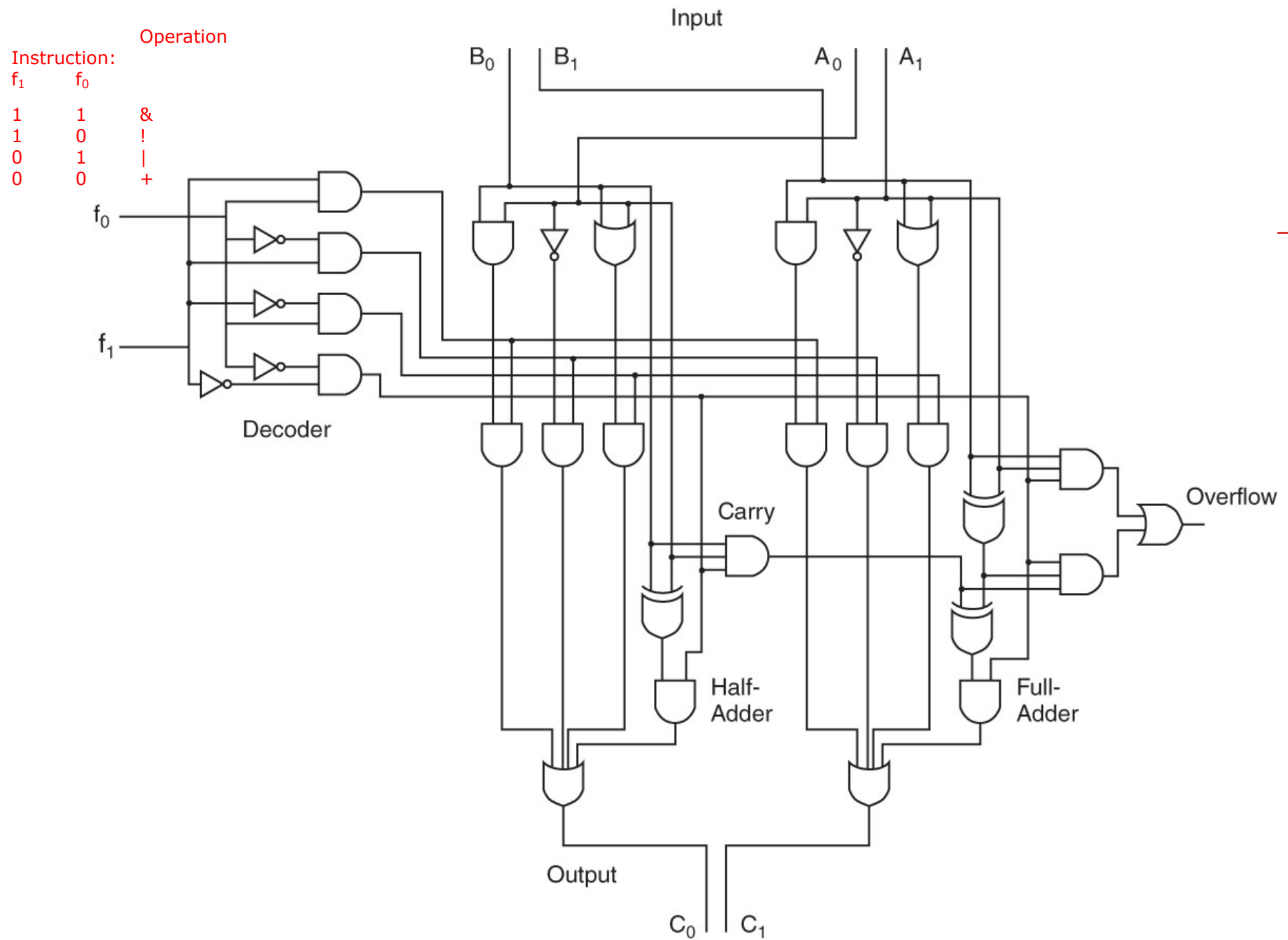
$$X = \bar{A} \bar{B} \bar{C} D + A \bar{B} \bar{C} D + A B \bar{C} \bar{D} + \bar{A} B C + \bar{A} \bar{B} C$$

# Practice

---

- Using a 8-to-1 multiplexer to implement a 3-bit prime detector

Find minterms based prime numbers first, and then use them to construct the multiplexer.



A Simple Two-Bit ALU

## 3.6 Sequential Logic Circuits (*SLC*)

---

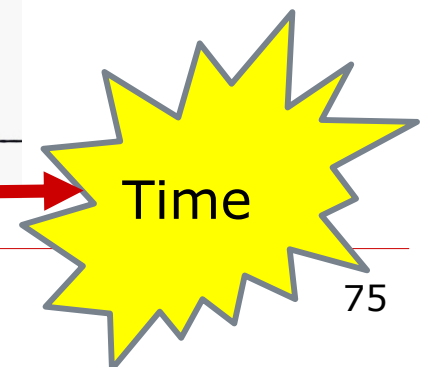
- Combinational logic circuits are perfect for those applications when a Boolean function be **immediately evaluated**, given the current inputs.
    - Examples: multiplexer, ripple-carry adder, shifter, etc
  - However, sometimes, we need a kind of circuits that change value by considering **the current inputs** and **its current state**.
    - **Memory** is such an example that requires to remember the current state
    - The circuits need to “**remember**” their states.
  - *Sequential logic circuits* (*SLC*) provide this functionality.
-

# How to “*remember*”?

---

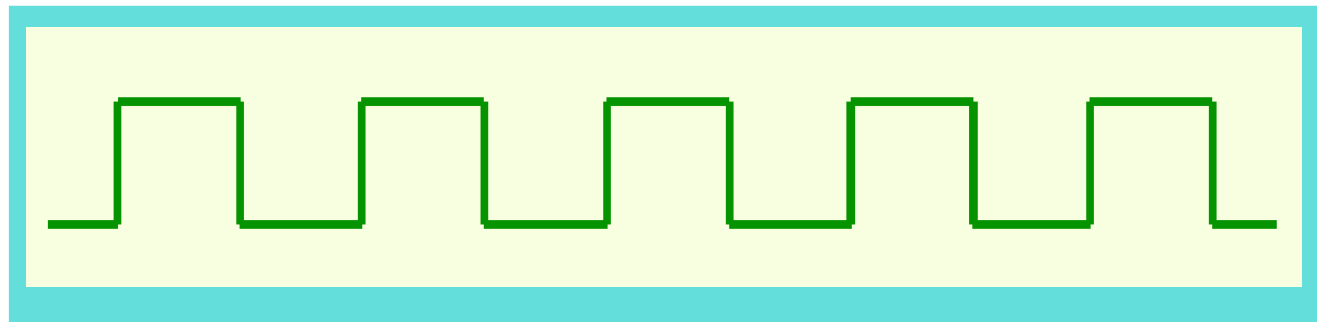
□ Think about the states in your own life-time

- 1 years old, blabla...
- 2 years old, blabla...
- 3 years old, blabla...



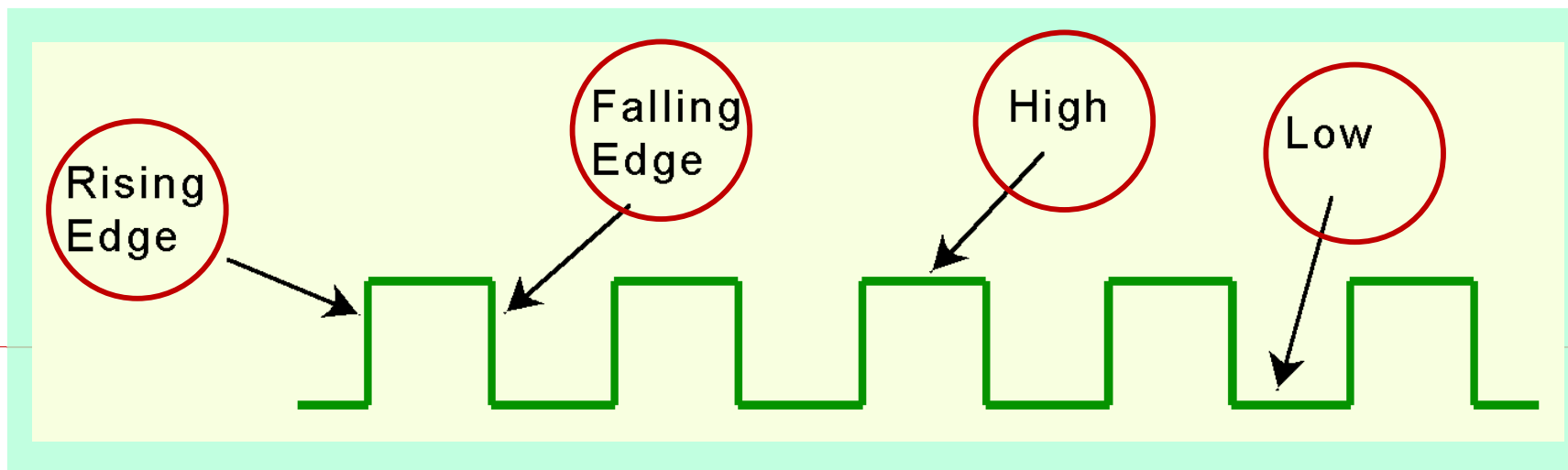
# Essential Component of Sequential Circuits: **Clocks**

- As the name implies, sequential logic circuits require a means by which events can be sequenced.
- The change of states is triggered by the clock.
  - The “***clock***” is ***a special circuit*** that sends electrical pulses to a *sequential logic circuit*.
- Clocks produce electrical waveforms constantly, such as the one shown below.



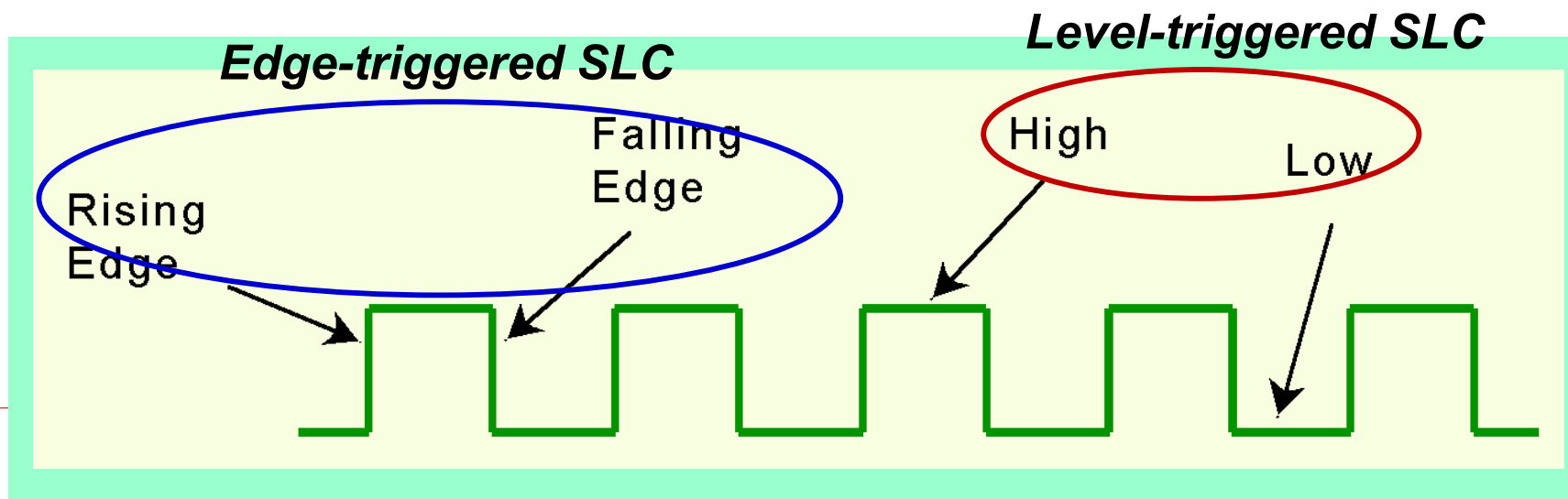
# When Change Its State?

- ❑ State changes occur in sequential circuits, *only when the clock ticks*.
- ❑ A sequential logic circuits could changes it state
  - Either, at the rising/falling edge of the clock pulse ,
  - Or, when the clock pulse reaches its highest/lowest level.



# Edge-triggered Or Level-triggered?

- ❑ SLC that changes its state at the rising edge, or the falling edge of the clock pulse is called **Edge-triggered SLC**.
- ❑ SLC that changes its state when the clock voltage reaches to its highest or lowest level are called **Level-triggered SLC**.





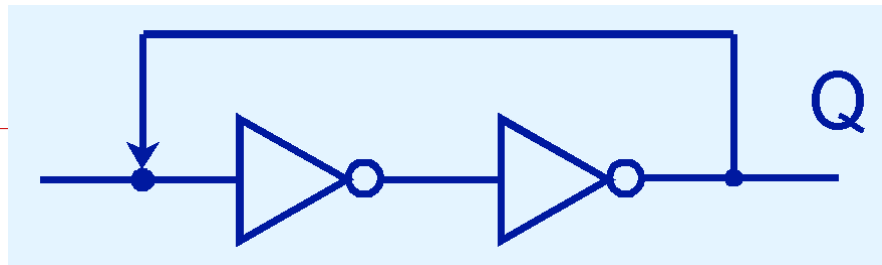
# Latch And Flip-flop

---

- latch and flip-flop are two kinds of SLCs, which are used to construct memory
  - A **latch** is *level-triggered*
  - A **flip-flop** is *edge-triggered*
- Which one depends on the length of the clock pulse?
  - Latch, or
  - flip-flop?

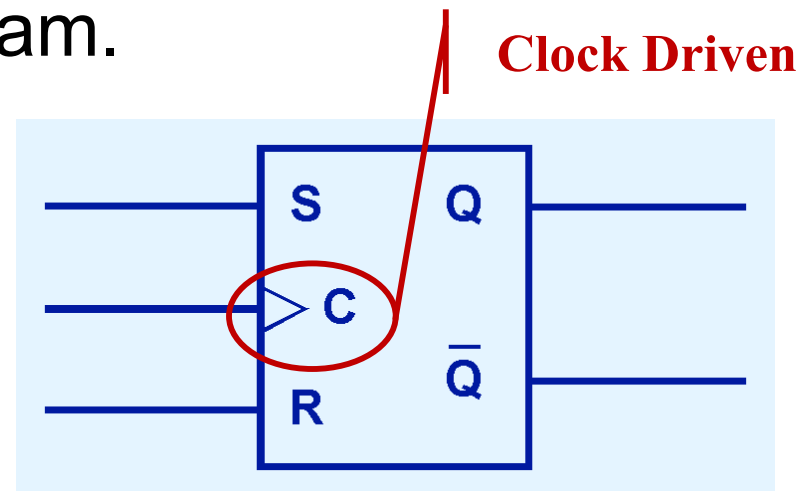
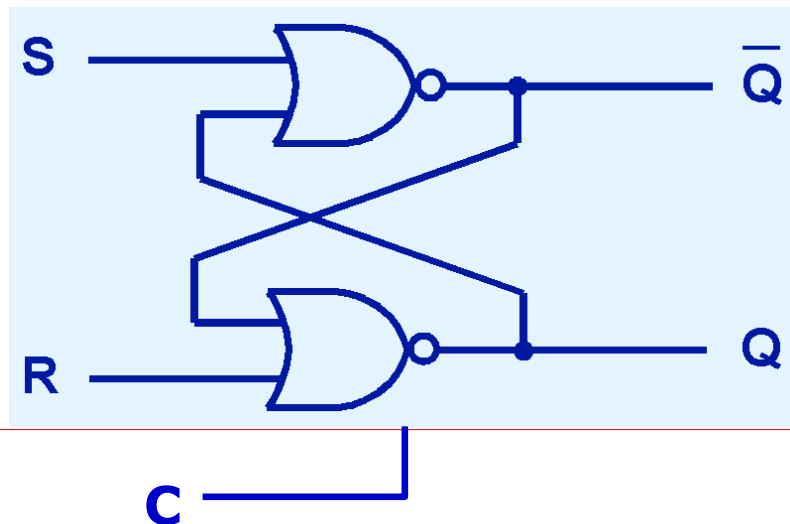
# Essential Component Of Sequential Circuits: *Feedback*

- The most important design mechanism of SLC is *Feedback*
  - *Feedback* can retain the state of sequential circuits
- Feedback in digital circuits occurs when an output is looped back as an input.
- A simple example of this concept is shown below.
  - If Q is 0 it will always be 0, if it is 1, it will always be 1. --- *The motivation of Memory!*



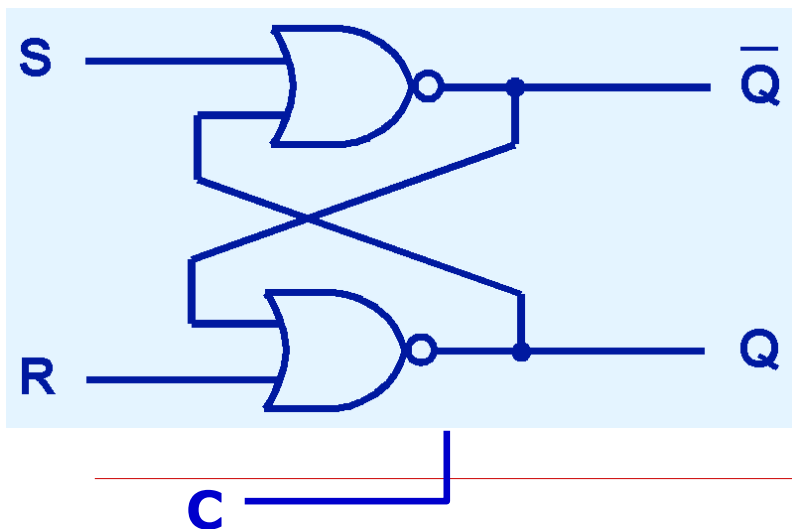
# SR Flip-flop

- You can see how feedback works by examining the most basic sequential logic components, the **SR flip-flop**.
  - The “**SR**” stands for *set/reset*.
- The internals of an SR flip-flop are shown below, along with its block diagram.



# Behavior Of An SR Flip-flop

- The behavior of an SR flip-flop is illustrated in the following truth table.
- Let's denote  $Q(t)$  as the value of the output at time  $t$ , and
- Denote  $Q(t+1)$  is the value of  $Q$  at time  $t+1$ .



S	R	$Q(t+1)$
0	0	$Q(t)$ (no change)
0	1	0 (reset to 0)
1	0	1 (set to 1)
1	1	undefined

# SR Flip-flop Truth Table

- We consider  $Q(t)$ , its current output, as the third input for SR flip-flop, besides S and R.
- The truth table for this circuit, as shown on the right.
- When both S and R are 1, the SR flip-flop is in forbidden state

Retain its  
original value

Change its  
value

Present State			Next State
S	R	$Q(t)$	$Q(t+1)$
0	0	0	0
0	0	1	1
0	1	0	0
0	1	1	0
1	0	0	1
1	0	1	1
1	1	0	undefined
1	1	1	undefined

$Q(t+1) = Q(t)$

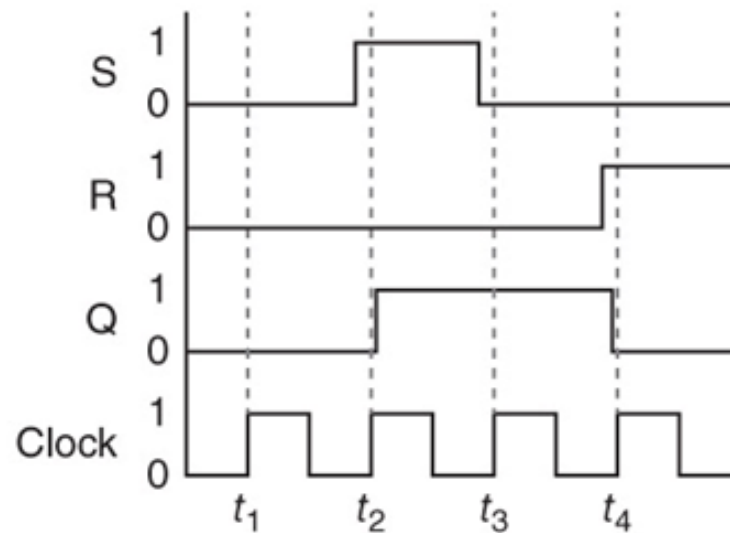
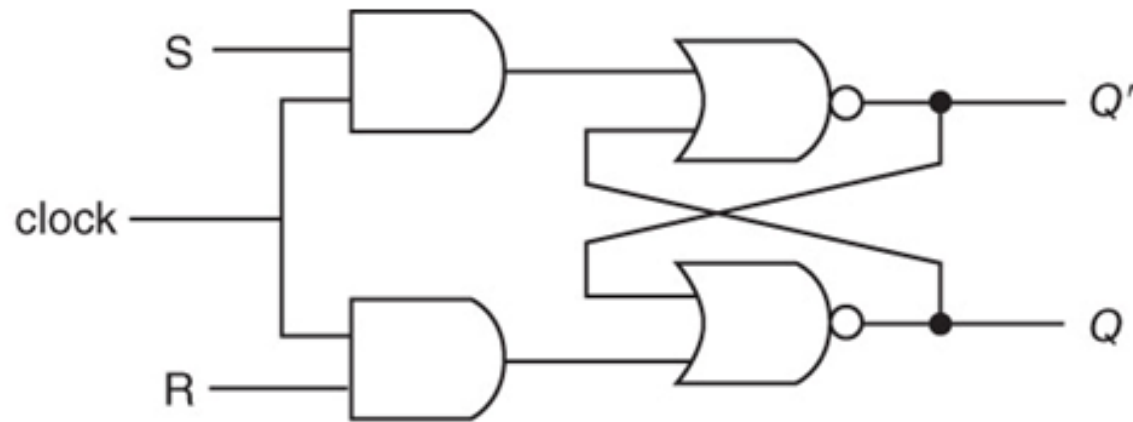
0

1

forbidden state

# Clocked SR Flip-flop

---

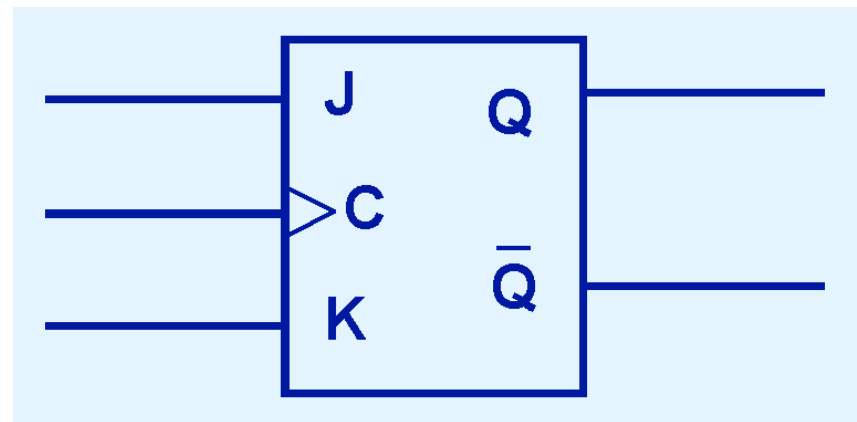


clock enables S or R!

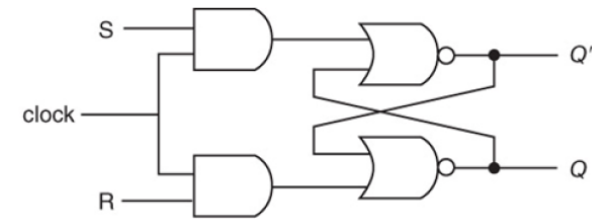
# JK Flip-flop

---

- One limitation of SR flip-flop is that, when S and R are both 1, the output is *undefined*.
    - This is not nice because it wastes a state
  - Therefore, SR flip-flop can be modified to provide a stable state when both S and R inputs are 1.
- 
- This modified flip-flop is called a **JK flip-flop**, shown on the right.
    - The “JK” is in honor of Jack Kilby.



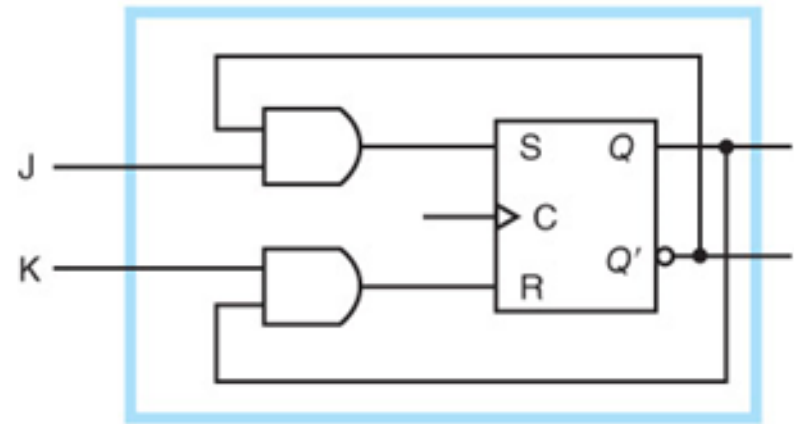
## 3.6 Sequential Circuits



□ On the right, we see how an SR flip-flop can be modified to create a JK flip-flop.

□ The truth table indicates that the flip-flop is stable for all inputs.

■ When J and K are both 1,  $Q(t+1) = \neg Q(t)$

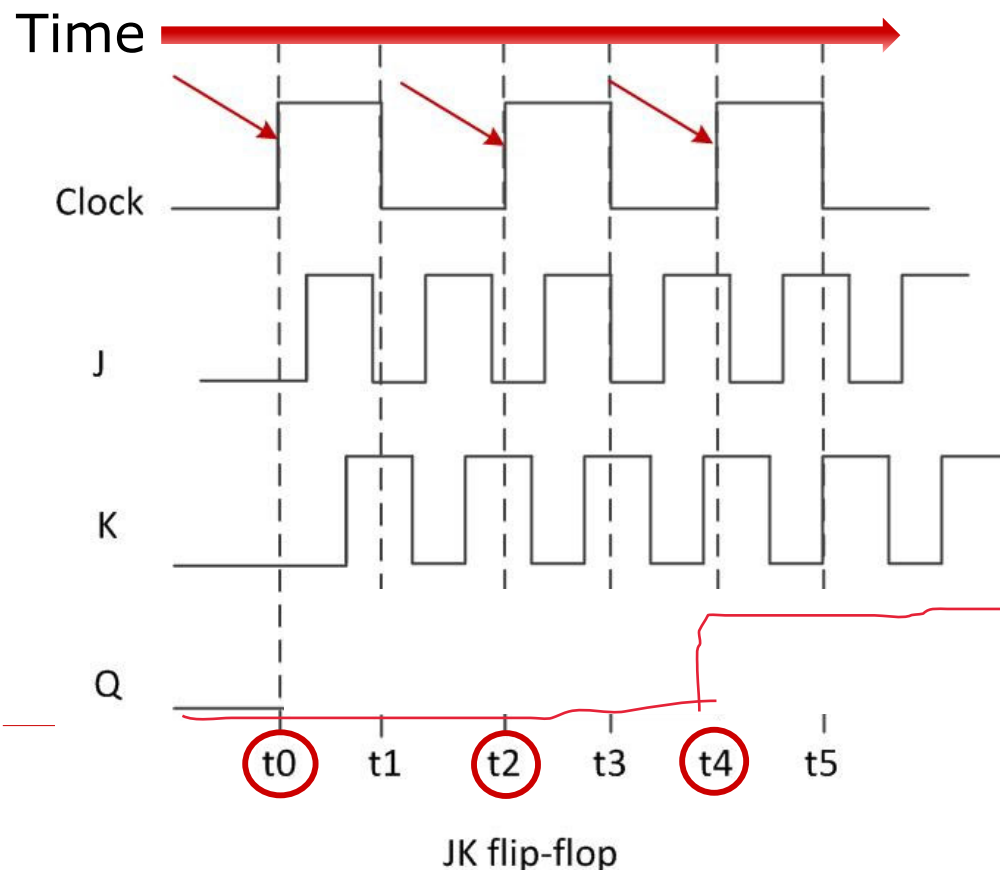


J	K	$Q(t+1)$
0	0	$Q(t)$ (no change)
0	1	0 (reset to 0)
1	0	1 (set to 1)
1	1	$\bar{Q}(t)$



# An Example

- Let's say a JK flip-flop is rising-edge triggered
- At  $t_0$ ,  $Q(t) = 0$ . What will be the changes of the value of  $Q$  over time?



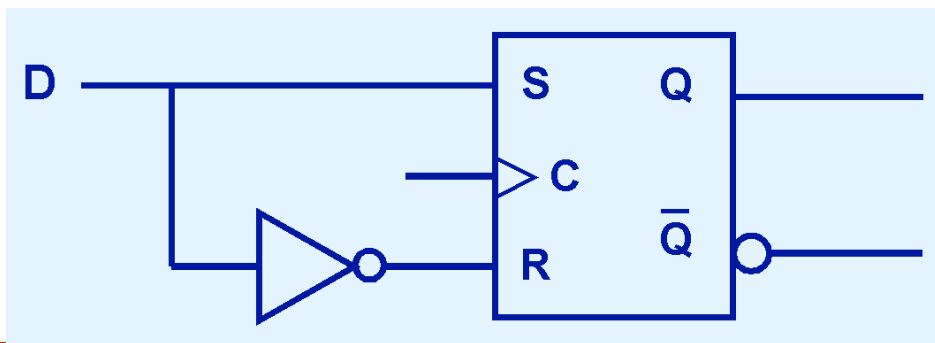
- Any time other than the rising edge won't trigger this JK flip-flop to change its state

J	K	$Q(t+1)$
0	0	$Q(t)$ (no change)
0	1	0 (reset to 0)
1	0	1 (set to 1)
1	1	$\bar{Q}(t)$

87

# D Flip-flop

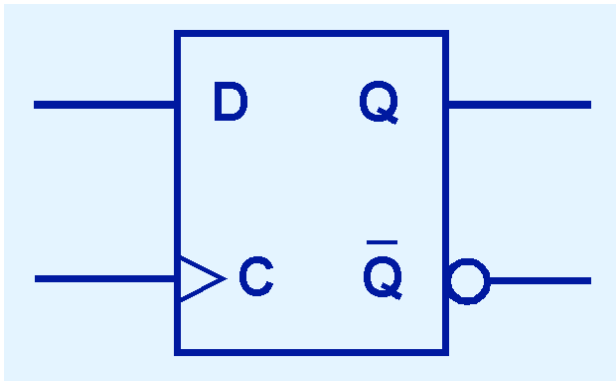
- Another modification of the SR flip-flop is the D flip-flop, shown below with its truth table.
- You will notice that the output of the flip-flop **remains the same** during subsequent clock pulses. The output changes **only when** the value of D changes.



D	$Q(t+1)$
0	0
1	1

# D Flip-flop

- The D flip-flop is the **fundamental** circuit of computer **memory**.
  - D flip-flop and its truth table are illustrated as below.



D	$Q(t+1)$
0	0
1	1

## 3.6 Sequential Circuits

---

- Sequential circuits are used anytime that we need to design a “**stateful**” application.
  - A stateful application is one where the next state of the machine depends on the current state of the machine and the input.
- A stateful application **requires both combinational and sequential logic.**
- The following slides provide several examples of circuits that fall into this category.

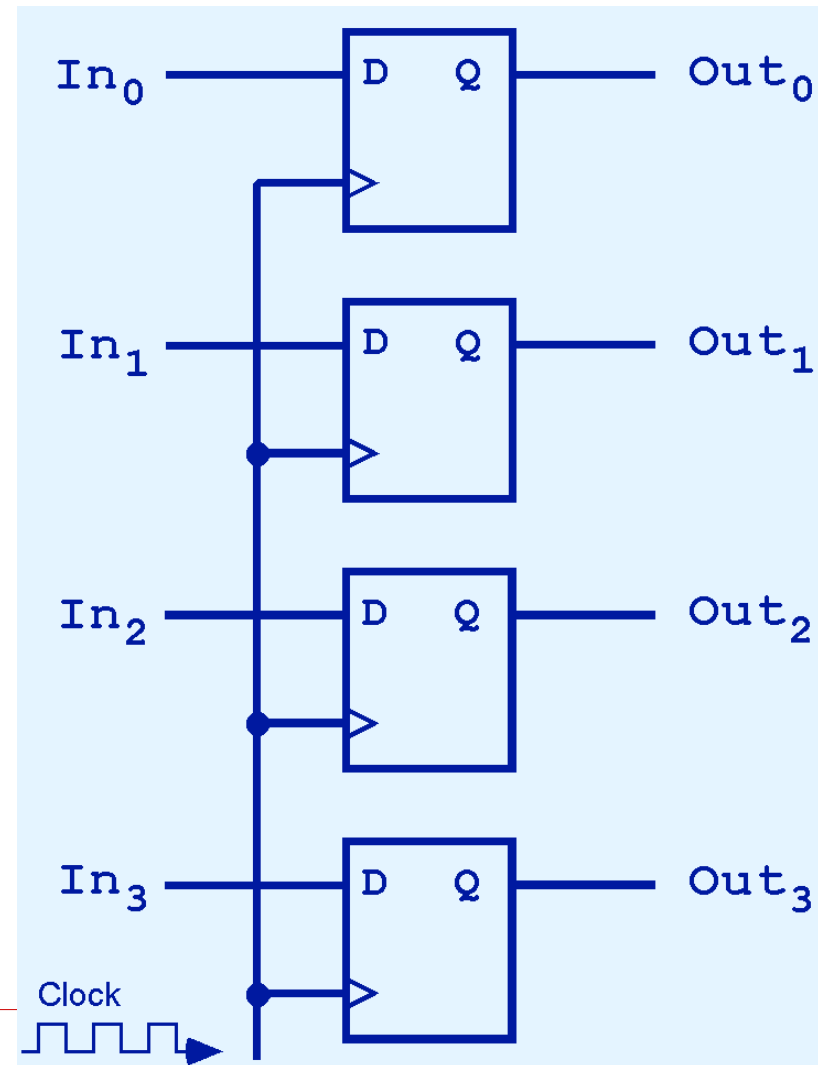
Can you think of others?

## 3.6 Sequential Circuits

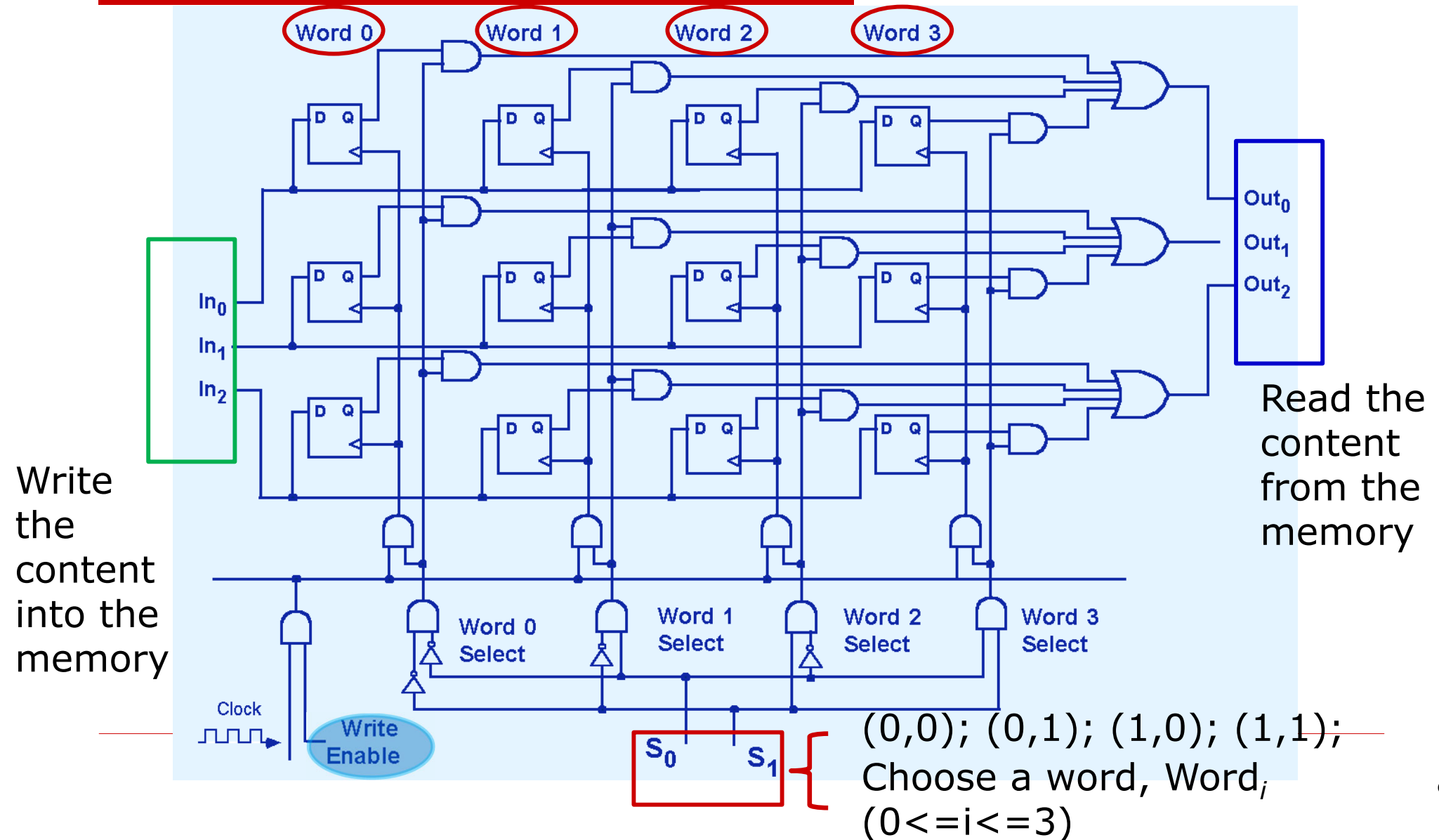
- This illustration shows a 4-bit **register** consisting of D flip-flops. You will usually see its block diagram (below) instead.



A larger memory configuration is shown on the next slide.

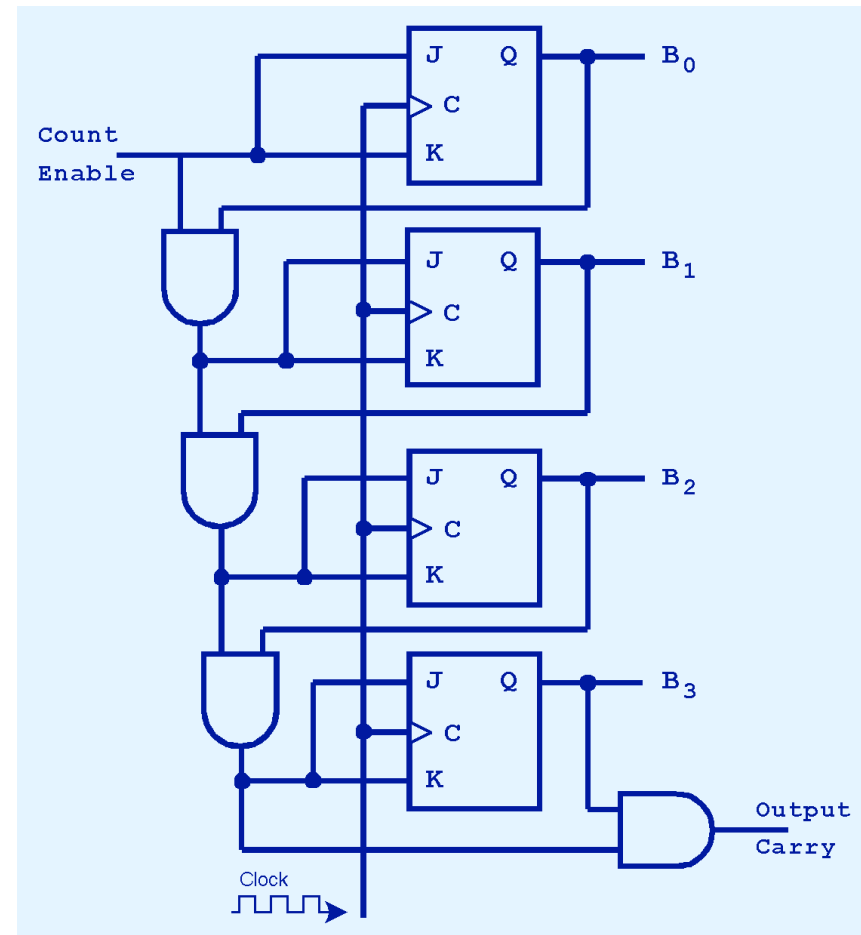


# 3.6 4X3 Memory



## 3.6 Sequential Circuits

- A binary **counter** is another example of a sequential circuit.
- The low-order bit is **complemented** at each clock pulse.
- Whenever it changes from 0 to 1, the next bit is complemented, and so on through the other flip-flops.



## 3.7 Designing Circuits

---

- Digital designers rely on **specialized software to create efficient circuits.**
  - Thus, software is an enabler for the construction of better hardware.
- Of course, **software** is in reality a collection of algorithms that could just as well be **implemented in hardware.**
  - Recall the **Principle of Equivalence of Hardware and Software.**



# Designing Circuits

---

- ❑ When we need to implement a **simple, specialized algorithm** and its execution **speed** must be as **fast** as possible, a **hardware solution is often preferred.**
- ❑ This is the idea behind ***embedded systems***, which are small special-purpose computers that we find in many everyday things.
- ❑ Embedded systems **require special programming** that demands an **understanding of the operation of digital circuits**, the basics of which you have learned in this chapter.
- ❑ Assembly programming for performance. FPGA (hardware) design using HDL (e.g. VHDL, Verilog) languages!

# Chapter 3 Conclusion

---

- ❑ Computers are implementations of Boolean logic.
- ❑ Boolean functions are completely described by truth tables.
- ❑ Logic gates are small circuits that implement Boolean operators.
- ❑ The basic gates are AND, OR, and NOT.
  - The XOR gate is very useful in parity checkers and adders.
- ❑ The “universal gates” are NOR and NAND.

# Chapter 3 Conclusion

---

- ❑ **Computer circuits** consist of **combinational** logic circuits and **sequential** logic circuits.
- ❑ **Combinational** circuits **produce outputs almost immediately** when their inputs change.
- ❑ **Sequential** circuits **require clocks** to control their changes of state.
- ❑ The basic sequential circuit unit is the **flip-flop**:  
The behaviors of the **SR, JK, and D** flip-flops are the most important to know.

---

# End of Chapter 3