# EE101: Basic Electronics
# Session: 2021-22

Co-Instructor: Prof. Rohit Sinha

Dept of EEE, IIT Guwahati

# Digital Electronics

Syllabus Outline:

- Number Systems and Binary codes
- Boolean Algebra and Logic Gates
- Combinational Logic Circuits
- Sequential Circuits

Texts and Refs:

- N. S. Widmer, G. L. Moss, and R. J. Tocci, Digital Systems, 12th edition. Pearson, 2017.
- D. P. Leach, A. P. Malvino, and G. Saha, Digital Principles And Applications, 8th edition. McGraw-Hill, 2014.

# Introduction to Digital Circuits

# Types of Electronic Circuits

- There are two ways of representing the numerical value of the quantities:
  - Analog representation
  - Digital representation

- Based on the nature of signal being processed, the electronic circuits can be categorized as:
  - Analog circuits
  - Digital circuits
  - Mixed-signal (both Analog and Digital) circuits

In analog representation, a quantity is represented by a voltage, current or meter movement that is proportional to the value of that quantity

Examples:

- Analog watch
- Automobile speedometer
- Analog voltmeter or ammeter





Analog quantities can vary over a continuous range of values

In digital representation, the quantity is represented by symbols called digits which are not proportional to the value of that quantity

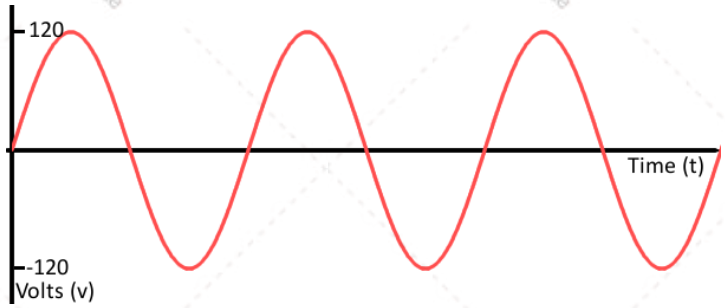
Example: A digital watch

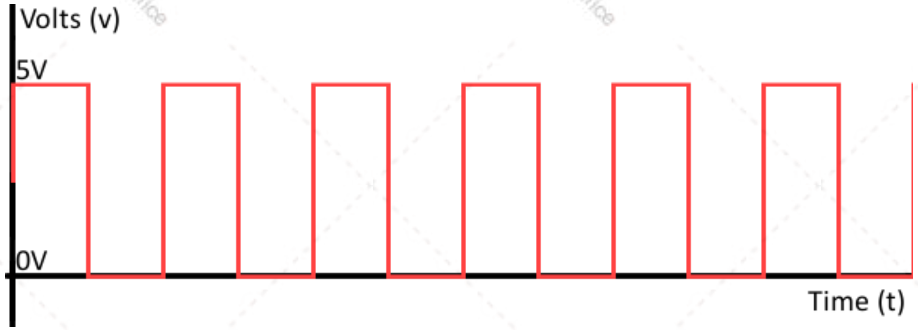

It provides the time of day in the form of decimal digits which represent hours and minutes (and sometimes seconds)

This digital representation of the time of day changes in discrete steps

Digital quantities vary in discrete steps over a range of values

# Analog vs Digital Systems



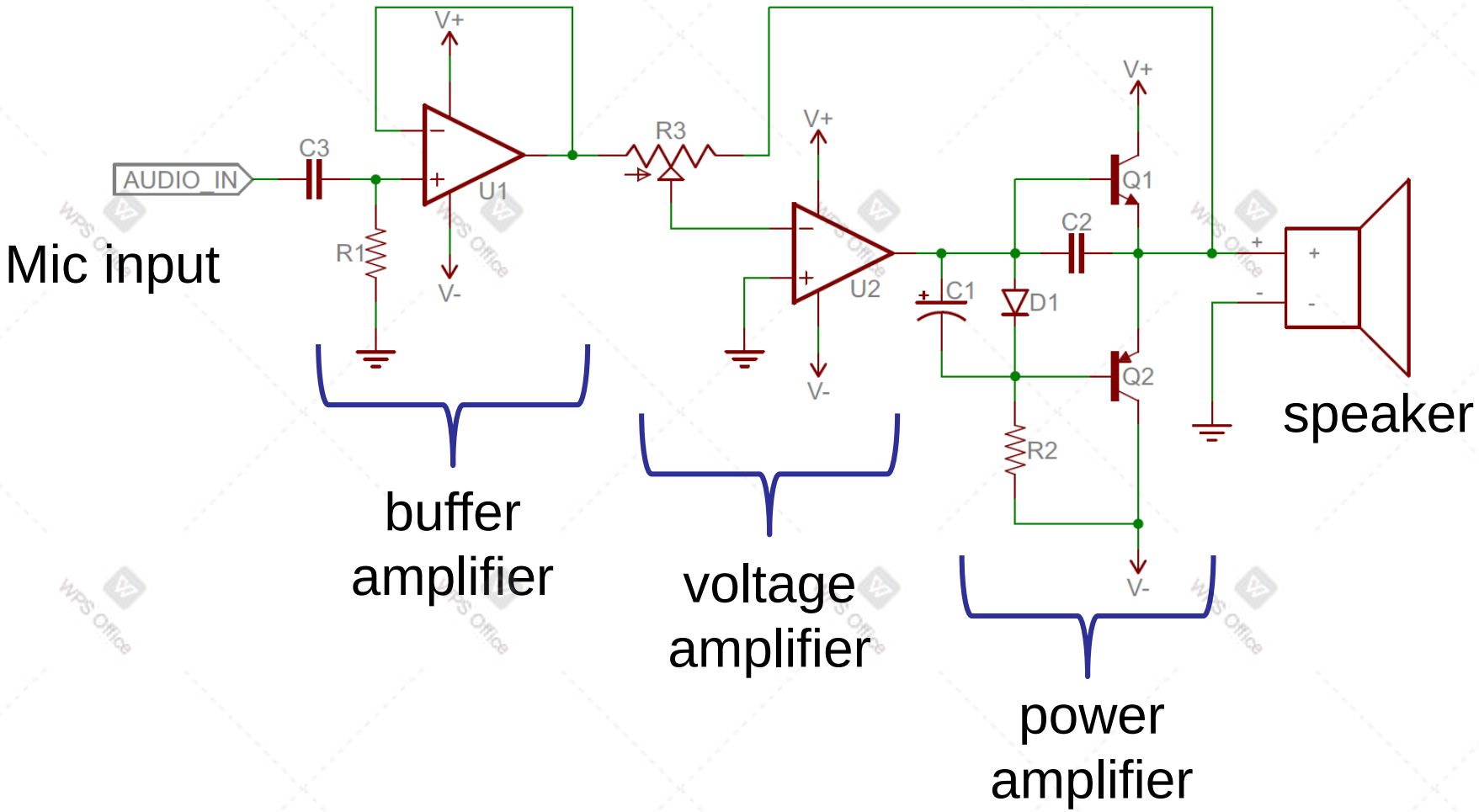




- Analog signal
  - Continuous time-varying voltages and/or currents
  - Infinite possibilities
- Analog circuit elements:
  - Resistor, Capacitor, Inductor, Diode, Transistor, OPAMP

- Digital signal
  - Discrete or sampled in time and quantized
  - Two possible values:
    - 0 V, low, false (logic 0)
    - 5 V, high, true (logic 1)
- Digital circuit elements:
  - Logic gates: AND, OR, NOT; Combinational circuits and Flip-flops

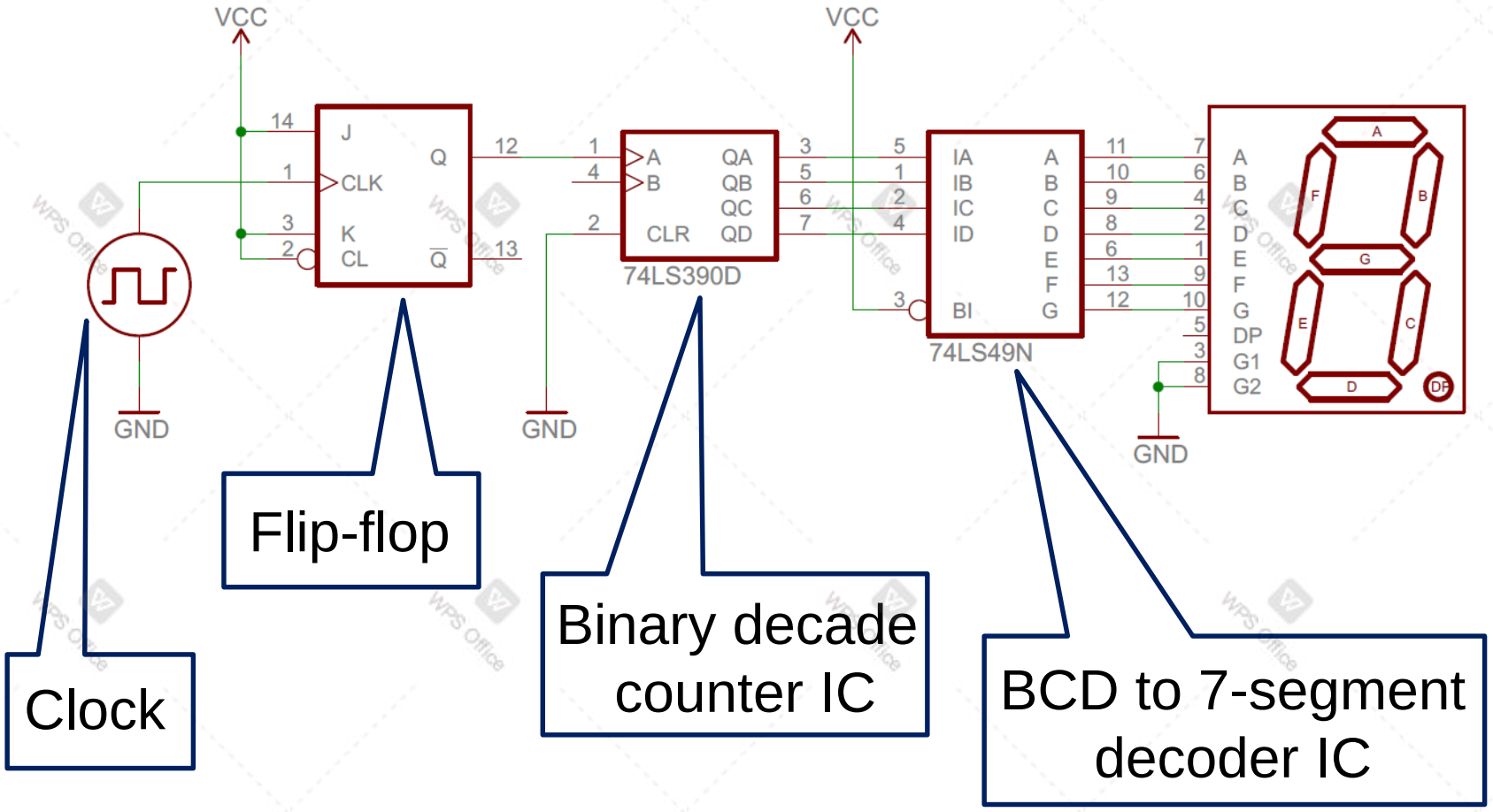
# An Example of Analog System

An analog audio amplifier



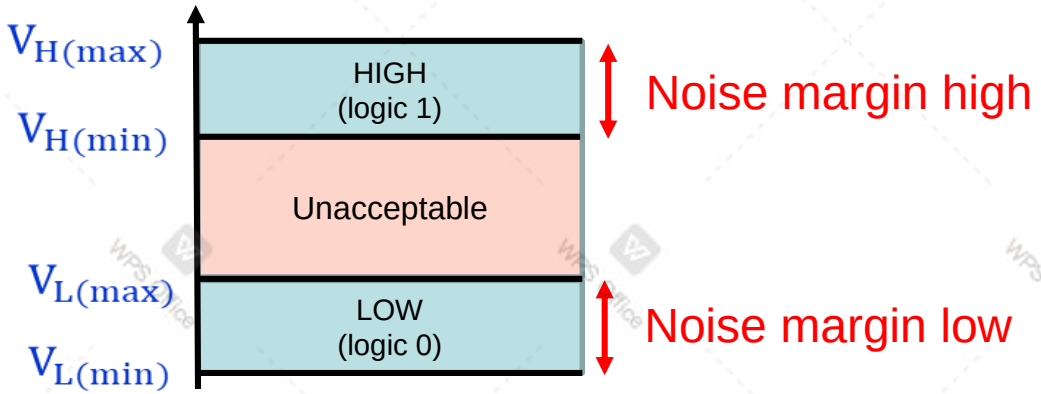
# An Example of Digital System

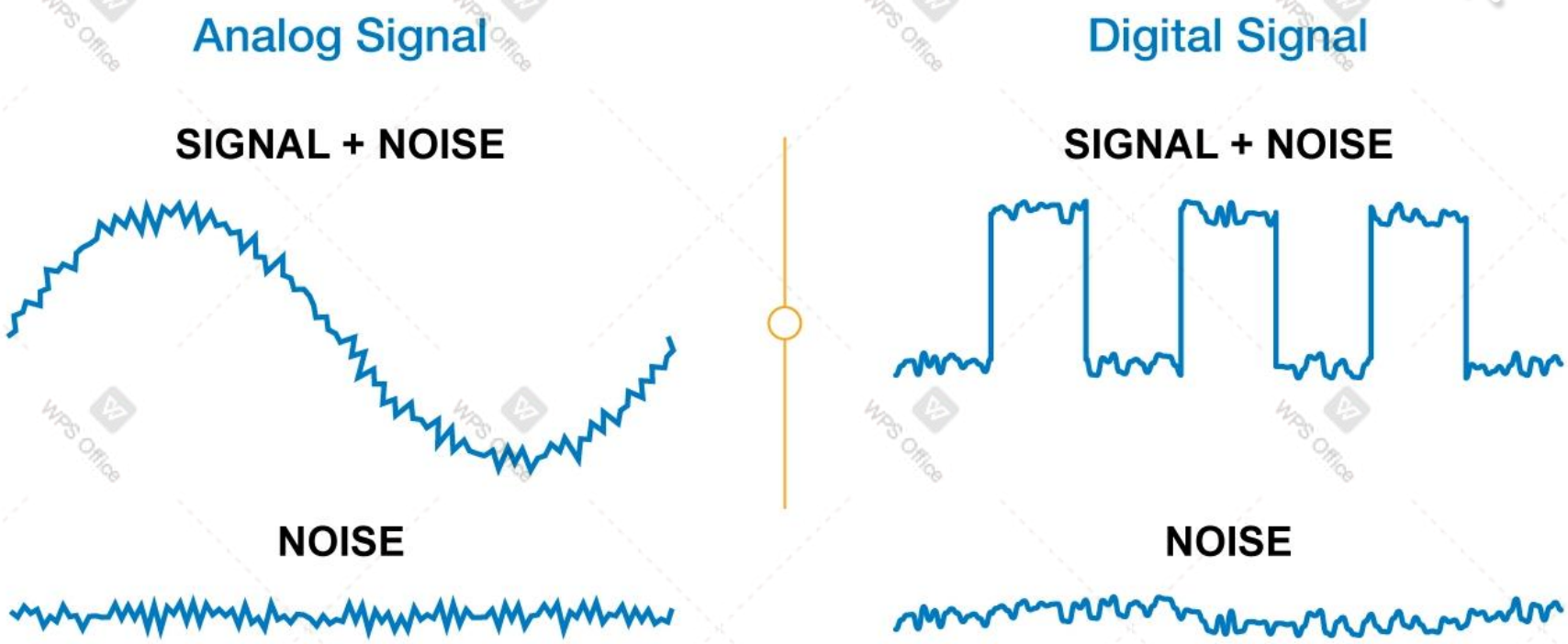
Single digit 7-segment display decade counter



# Advantages of Digital Paradigm

1. In all electronic circuits, the signals get affected by the spurious voltage fluctuations (noise). In digital circuits, the exact value of a voltage is not important as long as it falls within the defined logic level ranges of voltage.






*In contrast to analog circuit, the digital circuit can mitigate the affect of noise level not exceeding the noise margin*

2. Digital systems are easier to design as they use only switching circuits with only HIGH and LOW ranges

3. Accuracy and precision are greater

4. Operation can be programmed so offer greater flexibility

5. It is easy to transmit and store the information in digital form

6. More digital circuitry can be fabricated on IC chips

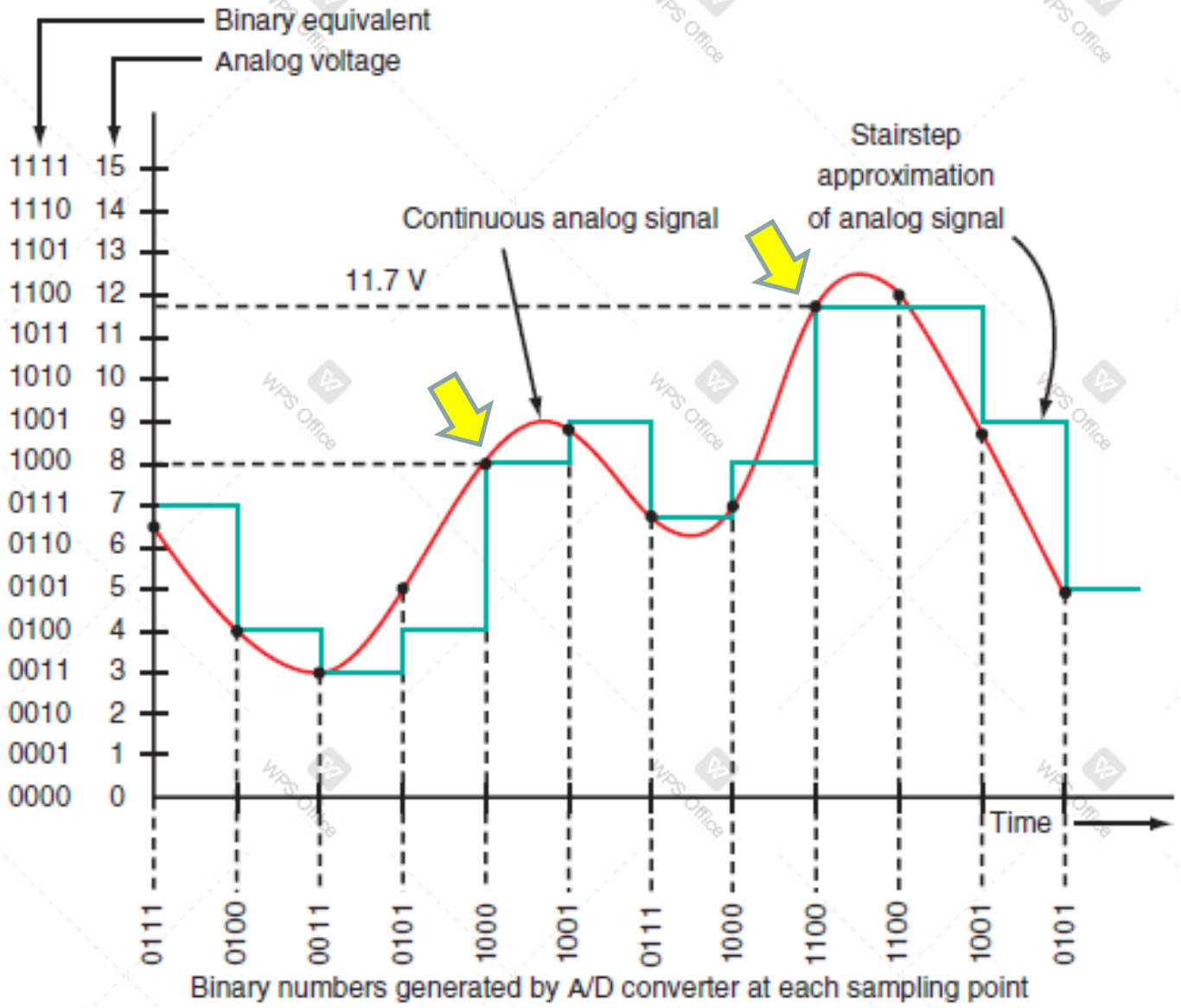
# Limitations of Digital Paradigm

1. Real world signals are analogue in nature. For processing such signals by digital systems, it is required to translate a continuous signal to a digital signal through A/D conversion which leads to small *quantization errors*.

   A/D conversion involves sampling followed by quantization which is explained graphically on the next slide.

   To reduce quantization errors, more voltage levels need to be defined, thus increasing the size of resulting digital data

# Analog-to-Digital (A/D) Conversion



2. Digital circuits operate only with digital signals hence, *encoders* and *decoders* are required for the processing. This increases the cost of digital equipment
3. For smaller circuits, the digital circuit is comparatively more expensive than analog one
4. Energy consumption in digital circuit is more than analog circuit for same calculation or signal processing
5. Production of heat is more due to higher energy consumption
6. Portability of digital circuit is difficult

# Number Systems

# Number Systems

| | |
|---|---|
| Decimal | Base 10, using 0,1,….,8,9 |
| Binary | Base 2, using only 0 & 1 |
| Octal | Base 8, using 0,1,….,6,7 |
| Hexadecimal | Base 16, using 0,1,….,9, A,…,F |

*Used in computers and digital systems*

Any base can be used and the numbers can be changed from one base to another to change representation. The Mayans used a Base-20 system, the Babylonians used Base-60!!!

# Decimal Number System – Base 10

It uses ten numerals or symbols: 0,1,2,3,4,5,6,7,8 and 9

A positional value system is used where each digit has its own place value or weight expressed as a power of 10

Example: A generic decimal number with (N+1) digits in the integer part and K digits in the fractional part

$$a_N, a_{N-1}, \ldots, a_1, a_0 \,.\, a_{-1}, a_{-2}, \ldots, a_{-K}$$

decimal point

represents the following value

$$a_N \times 10^N + a_{N-1} \times 10^{N-1} + \ldots + a_1 \times 10^1 + a_0 10^0 + a_{-1} \times 10^{-1} + \ldots + a_{-K} \times 10^{-K}$$

# Binary Number System – Base 2

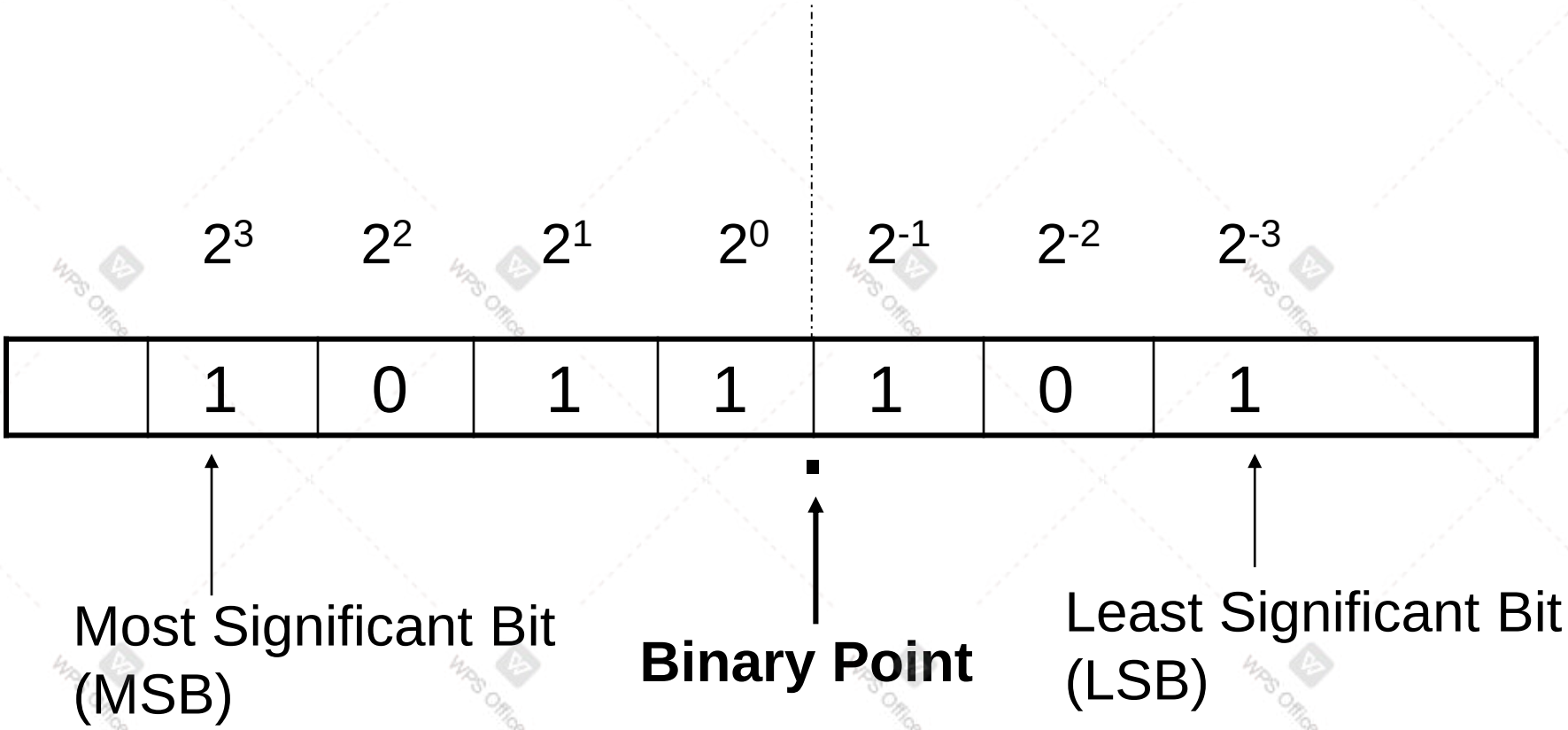
Uses only two numerals (0 and 1). In this system, the digit is referred to as bit (a shortened form of binary digit)

It also uses a positional value system where each bit has its own place value or weight expressed as a power of 2

- Places to the left of the binary point have positive powers of 2
- Places to the right of the binary point have negative powers of 2

*This is exactly what we did in the decimal system except that there we used powers of 10 instead of powers of 2!*

# Example : The number $1011.101_2$



# BINARY TO DECIMAL CONVERSION

Example: Convert the number $1011.101_2$ to its decimal equivalent

$$1011.101_2 = 1 \times 2^3 + 0 \times 2^2 + 1 \times 2^1 + 1 \times 2^0 + 1 \times 2^{-1} + 0 \times 2^{-2} + 1 \times 2^{-3}$$

$$= 8 + 0 + 2 + 1 + 0.5 + 0 + 0.125$$

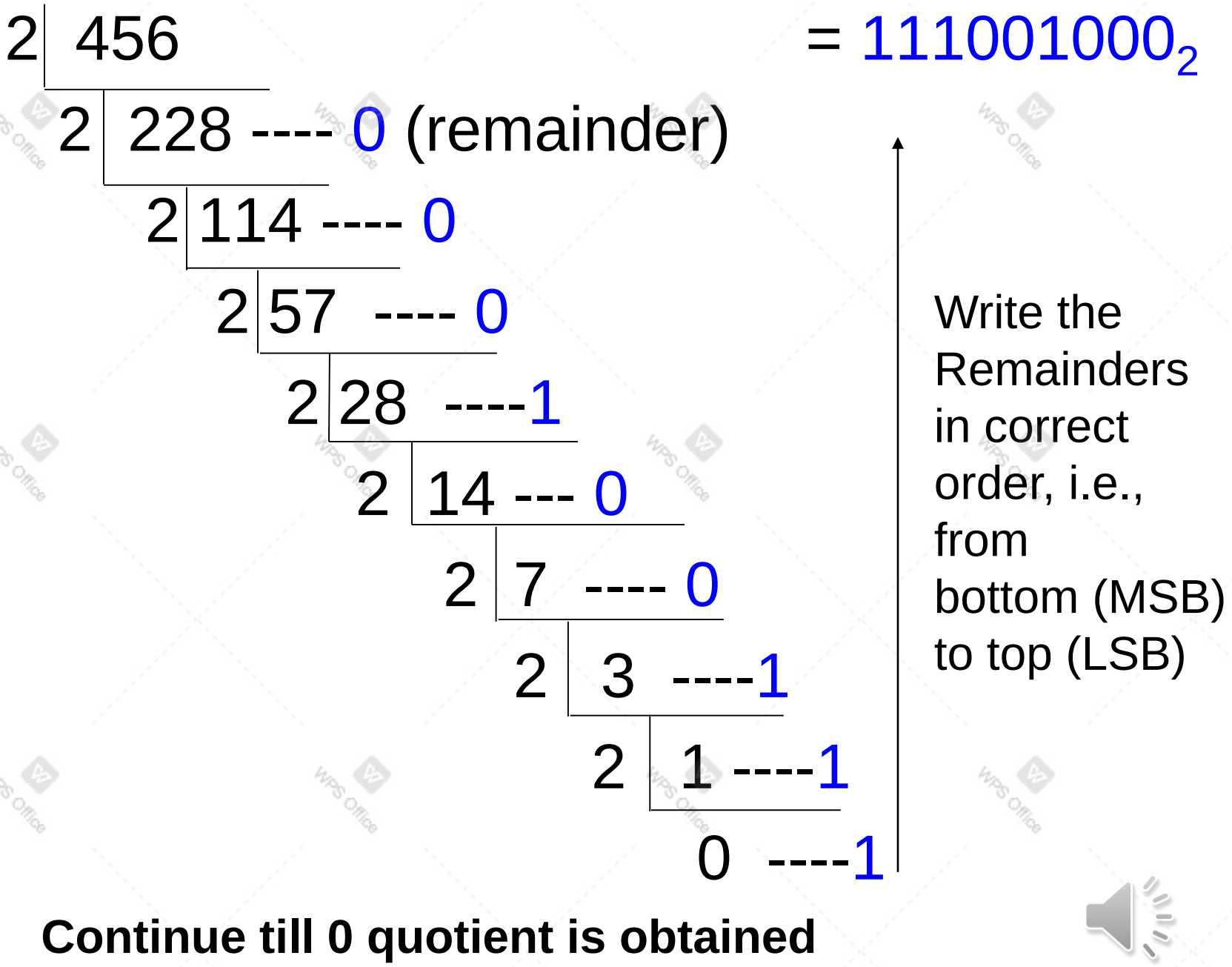
$$= 11.625_{10}$$

# DECIMAL TO BINARY CONVERSION

Example: Convert $456_{10}$ into binary equivalent

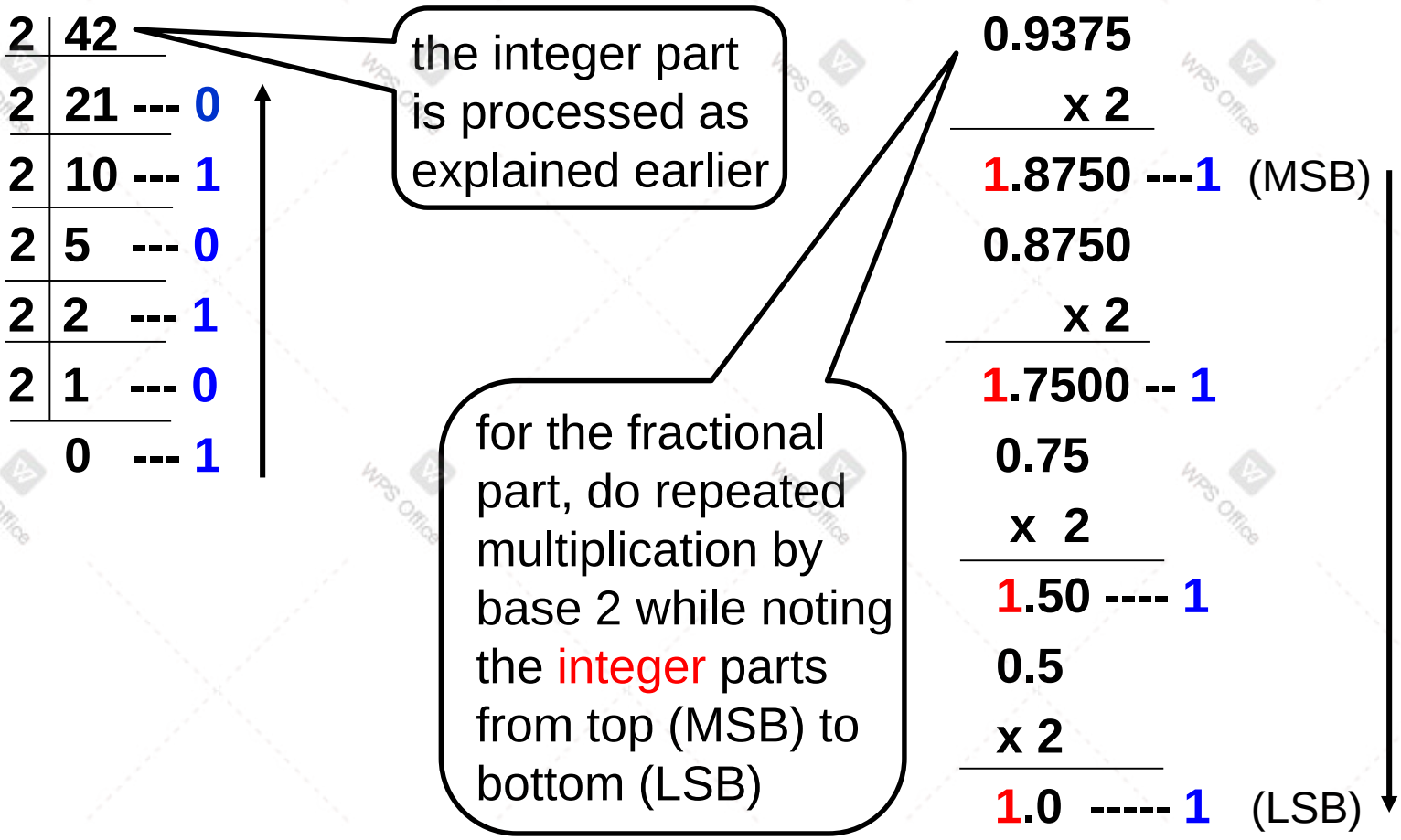
For the same, the given decimal number is repeated divided by 2 and the remainders are noted (shown on the next slide)

The answer is $111001000_2$




**Continue till 0 quotient is obtained**

## Convert $42.9375_{10}$ to binary equivalent




**Continue till 0 is obtained in fraction part**

$$42.9375_{10} = 101010.1111_2$$

**Example: Convert $25.012_{10}$ into binary equivalent**

| | | |
|---|---|---|
| 2 | 25 | |
| 2 | 12 | ---- 1 |
| 2 | 6 | ---- 0 |
| 2 | 3 | ---- 0 |
| 2 | 1 | ---- 1 |
| | 0 | ---- 1 |

0.012
x 2
0.024 --- 0
x 2
0.048 --- 0
x 2
0.096 --- 0
x2
0.192----0

0.192
x 2
0.384 --- 0
x 2
0.768 --- 0
x 2
1.536 --- 1
0.536
x 2
1.072 --- 1
0.072
x 2
0.144 --- 0
x 2
0.288 --- 0

$25.012_{10} = 11001.0000001100_2$

(approximately)

# Octal Number System

It uses base-8 positional value system and 8 possible numerals are 0,1,2,3,4,5,6,7

## Octal to decimal conversion

$24.6_8 = 2 \times 8^1 + 4 \times 8^0 + 6 \times 8^{-1} = 16 + 4 + 0.75 = 20.75_{10}$

## Decimal to octal conversion

same as decimal-to-binary conversion but with base 8

Example: Convert $49.21875_{10}$ into octal equivalent

| | | |
|---|---|---|
| 8 | 49 | |
| 8 | 6 | --- 1 ↑ |
| | 0 | --- 6 |

| | |
|---|---|
| 0.21875 | |
| × 8 | |
| 1.75000 | ----- 1 |
| .75 | |
| × 8 | |
| 6.00 | ----- 6 ↓ |

**$49.21875_{10} = 61.16_8$**

# Octal to Binary Conversion

Conversion from octal to binary is performed by converting each octal digit to its 3-bit binary equivalent

$0_8 = 000_2$, $1_8 = 001_2$, …… , $6_8 = 110_2$, $7_8 = 111_2$

Ex : $472_8 = 100\ 111\ 010_2$

Ex : $642.71_8 = 110\ 100\ 010.111\ 001_2$

# Binary to Octal Conversion

The bits of the given binary number are grouped into groups of 3 bits starting from the LSB with inserting 0s to complete the last group, if required. Then each group is converted into its octal equivalent.

Ex : $1100011001_2 = \underline{001}\ \underline{100}\ \underline{011}\ \underline{001} = 1431_8$

1 4 3 1

In case of the mixed binary number, the grouping is done with respect to the binary point. Moving left for the integer part and moving right for the fractional part.

Ex : $1100.1011_2 = \underline{001}\ \underline{100}.\underline{101}\ \underline{100} = 14.54_8$

1 4 5 4

# Hexadecimal Number System

It uses base 16 and 16 possible symbols are 0 to 9 plus the letters A,B,C,D,E,F

$A=10_D$, $B=11_D$, $C=12_D$, $D=13_D$, $E=14_D$, $F=15_D$

## Hex to Decimal Conversion

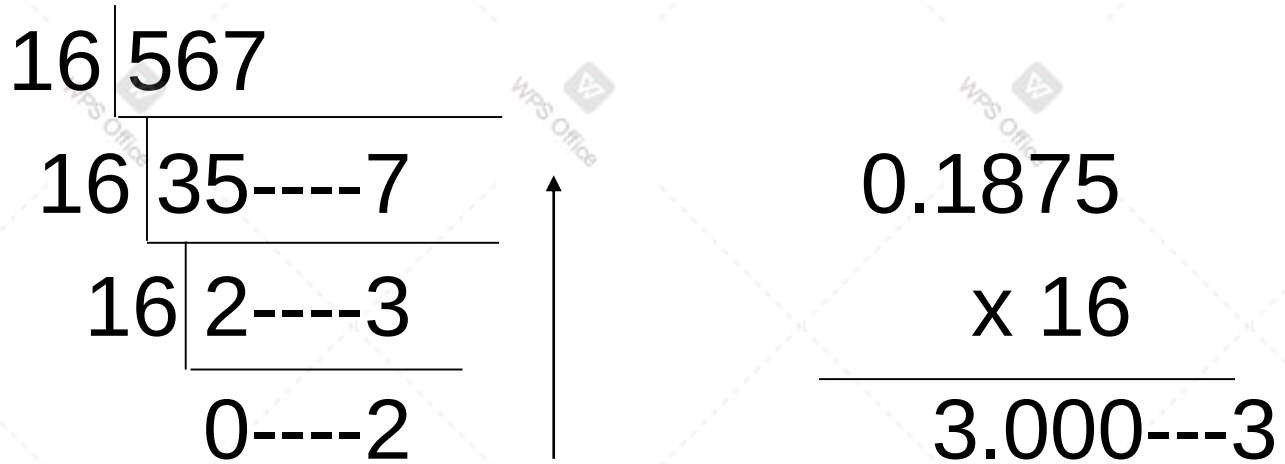
Use positional value system

Example: $3E6_{16} = 3 \times 16^2 + 14 \times 16^1 + 6 \times 16^0$

$= 998_{10}$

# Decimal to Hex Conversion

Example: Covert $567.1875_{10}$ to its hex equivalent




$567.1875_{10} = 237.3_{16}$

# Hex to Binary Conversion

Conversion from hex to binary is performed by converting each hex digit to its 4-bit binary equivalent

$0_{16} = 0000_2$, … , $9_{16} = 1001_2$, $A_{16} = 1010_2$, … , $F_{16} = 1111_2$

Example: $F9_{16} = 1111\ 1001_2$

Example: $20E.CA_{16} = 0010\ 0000\ 1110.1100\ 1010_2$

# Binary to Hex Conversion

Identical to binary-to-octal conversion except the groups of 4 bits are required to be formed

Example: $1110100110_2 = \underline{0011}\ \underline{1010}\ \underline{0110} = 3A6_{16}$
Example: $1101.00111100_2 = D.3C_{16}$

Example : Convert $B2F_{16}$ to octal equivalent

First convert hex to binary, then convert the resultant binary to octal

$B2F_{16} = 1011\ 0010\ 1111_2 = \underline{101}\ \underline{100}\ \underline{101}\ \underline{111} = 5457_8$

# Binary Arithmetic

Arithmetic operations are possible on binary numbers just as they are on decimal numbers. In fact the procedures are quite similar in both systems.

| Case | Addition | Subtraction | Multiplication | Division |
|---|---|---|---|---|
| i) | 0 + 0 = 0 | 0 – 0 = 0 | 0 × 0 = 0 | 0 / 1 = 0 |
| ii) | 0 + 1 = 1 | 1 – 0 = 1 | 0 × 1 = 0 | 1 / 1 = 1 |
| iii) | 1 + 0 = 1 | 1 – 1 = 0 | 1 × 0 = 0 | 0 / 0 (not valid) |
| iv) | 1 + 1 = 10 | 0 – 1 = 10 – 1 (with borrow 1) = 1 | 1 × 1 = 1 | 1 / 0 (not valid) |

# Binary Arithmetic Examples

| Addition | Subtraction | Multiplication | Division |
|---|---|---|---|
| 1 1<br>+ 1 1<br>1 1 0<br><br>Here, 1+1 (right most) = 0 and its carry 1 is added to left column as 1+1+1=11. | 1 0<br>– 0 1<br>0 1<br><br>Here, 0 – 1 (right most) = 1 because we take 2 borrow from left column, so the left entry becomes 0 and the top right entry becomes 10. | 1 1<br>× 1 0<br>0 0<br>1 1 ↓<br>1 1 0 | 11) 1 1 0 (10<br>1 1<br>× 0 0 |

# Octal Arithmetic

- Octal addition table

$A_8 + B_8 = Sum_8$

| + | 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 |
|---|---|---|---|---|---|---|---|---|
| 0 | 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 |
| 1 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 10 |
| 2 | 2 | 3 | 4 | 5 | 6 | 7 | 10 | 11 |
| 3 | 3 | 4 | 5 | 6 | 7 | 10 | 11 | 12 |
| 4 | 4 | 5 | 6 | 7 | 10 | 11 | 12 | 13 |
| 5 | 5 | 6 | 7 | 10 | 11 | 12 | 13 | 14 |
| 6 | 6 | 7 | 10 | 11 | 12 | 13 | 14 | 15 |
| 7 | 7 | 10 | 11 | 12 | 13 | 14 | 15 | 16 |

A

Sum

B

- Example – Addition

$456_8 + 173_8 = 651_8$

| | | | |
|---|---|---|---|
| **Carry** | **1 1** | | |
| | 4 5 6 | → | $302_{10}$ |
| + | 1 7 3 | → | $123_{10}$ |
| | 6 5 1 | → | $425_{10}$ |

- Example – Subtraction

$456_8 - 173_8 = 263_8$

| | | | |
|---|---|---|---|
| **Borrow** | **8** | | |
| | $^{3}\not{4}\,5\,6$ | → | $302_{10}$ |
| – | 1 7 3 | → | $123_{10}$ |
| | 2 6 3 | → | $179_{10}$ |

# Hexadecimal Arithmetic

- Hex addition table

$X_{16} + Y_{16} = Sum_{16}$

| + | 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | A | B | C | D | E | F |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| 0 | 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | A | B | C | D | E | F |
| 1 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | A | B | C | D | E | F | 10 |
| 2 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | A | B | C | D | E | F | 10 | 11 |
| 3 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | A | B | C | D | E | F | 10 | 11 | 12 |
| 4 | 4 | 5 | 6 | 7 | 8 | 9 | A | B | C | D | E | F | 10 | 11 | 12 | 13 |
| 5 | 5 | 6 | 7 | 8 | 9 | A | B | C | D | E | F | 10 | 11 | 12 | 13 | 14 |
| 6 | 6 | 7 | 8 | 9 | A | B | C | D | E | F | 10 | 11 | 12 | 13 | 14 | 15 |
| 7 | 7 | 8 | 9 | A | B | C | D | E | F | 10 | 11 | 12 | 13 | 14 | 15 | 16 |
| 8 | 8 | 9 | A | B | C | D | E | F | 10 | 11 | 12 | 13 | 14 | 15 | 16 | 17 |
| 9 | 9 | A | B | C | D | E | F | 10 | 11 | 12 | 13 | 14 | 15 | 16 | 17 | 18 |
| A | A | B | C | D | E | F | 10 | 11 | 12 | 13 | 14 | 15 | 16 | 17 | 18 | 19 |
| B | B | C | D | E | F | 10 | 11 | 12 | 13 | 14 | 15 | 16 | 17 | 18 | 19 | 1A |
| C | C | D | E | F | 10 | 11 | 12 | 13 | 14 | 15 | 16 | 17 | 18 | 19 | 1A | 1B |
| D | D | E | F | 10 | 11 | 12 | 13 | 14 | 15 | 16 | 17 | 18 | 19 | 1A | 1B | 1C |
| E | E | F | 10 | 11 | 12 | 13 | 14 | 15 | 16 | 17 | 18 | 19 | 1A | 1B | 1C | 1D |
| F | F | 10 | 11 | 12 | 13 | 14 | 15 | 16 | 17 | 18 | 19 | 1A | 1B | 1C | 1D | 1E |

X

Sum

Y

- Example – Addition

$6AB_{16} + 2CA_{16} = 975_{16}$

| | | | |
|---|---|---|---|
| **Carry** | **1 1** | | |
| | 6 A B | → | $1707_{10}$ |
| + | 2 C A | → | $714_{10}$ |
| | 9 7 5 | → | $2421_{10}$ |

- Example – Subtraction

$6AB_{16} – 2CA_{16} = 3E1_{16}$

| | | | |
|---|---|---|---|
| **Borrow** | **16** | | |
| | $^{5}\cancel{6}$ A B | → | $1707_{10}$ |
| – | 2 C A | → | $714_{10}$ |
| | 3 E 1 | → | $993_{10}$ |

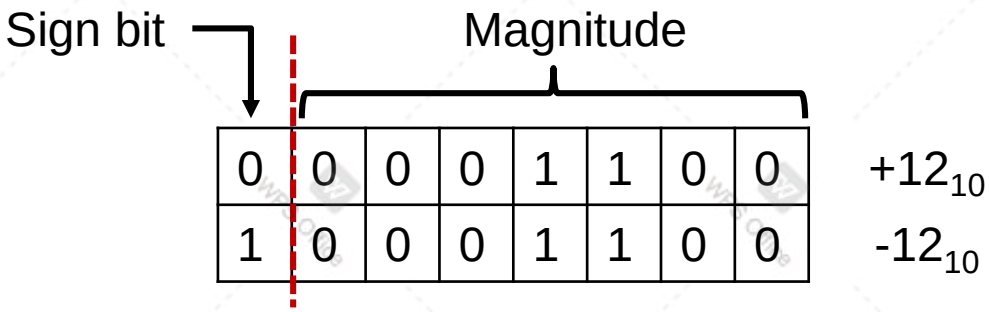
# Binary Information and Representation

- The smallest unit of information in digital systems is the bit, a single fact or value.
- Apart from that different groupings of bits are also defined and used to store large amounts of information and more complex data types.

| Number of Bits | Common Representation Terms |
|---|---|
| 1 | Bit / Flag |
| 4 | Nibble / Nybble |
| 8 | Byte / Octet / Character |
| 16 | Double Byte / Word |
| 32 | Double Word / Long Word |
| 64 | Very Long Word |

# Signed Binary Numbers

- Two common ways to represent negative numbers within the digital system are:
  - Signed magnitude representation
  - Radix complement representation
- In signed magnitude representation, the MSB of the binary numbers is used to indicate the sign




*Though the signed magnitude approach is easy to implement but leads to complications in doing arithmetic*

# Complement Arithmetic

- For each radix-$r$ system (radix represents base of number system), two complements are defined:

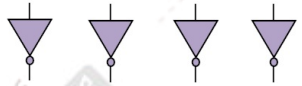
  1) Diminished radix complement or $(r-1)$'s complement
  2) Radix complement or $r$'s complement

- In binary system, we have 1's complement and 2's complement

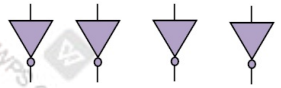
- In the digital systems, the complements are used in order to simplify the subtraction operation and for the logical manipulations.

# Binary Number System Complements

- The 1's complement of a binary number is obtained by bit inversion i.e., changing all 1's to 0's and all 0's to 1's

| | | | | |
|---|---|---|---|---|
| Binary number → | 1 | 0 | 1 | 0 |
| Bit inversion → |  | | | |
| 1's complement → | 0 | 1 | 0 | 1 |

- The 2's complement of a binary number is derived by adding 1 to the LSB of 1's complement of that number

| | | | | |
|---|---|---|---|---|
| Binary number → | 1 | 0 | 1 | 0 |
| Bit inversion → |  | | | |
| 1's complement → | 0 | 1 | 0 | 1 |
| Add 1 to LSB → | | | | 1 |
| 2's complement → | 0 | 1 | 1 | 0 |

# Subtraction using 2’s Complement

- Procedure for subtracting two binary numbers using 2's complement is as follows:
  - First, find the 2's complement of the subtrahend.
  - Now, add the resulting complement to the minuend.
  - If the above addition generates the carry then discard that and the result is a positive number; else take 2's complement of the result which will be negative number.

# Subtraction using 2's Complement (Contd.)

Ex: $1101_2 - 0111_2 = ?$

2's comp of 0111 = 1000 + 1 = 1001

Add minuend and 2's comp: 1101 + 1001 = 1 0110

By discarding the carry, we get the result as a positive number $0110_2$

Ex: $1010_2 - 1101_2 = ?$

2's comp of 1101 = 0010 + 1 = 0011

Adding minuend and 2's comp: 1010 + 0011 = 1101

No carry has generated so find 2's comp of 1101 → 0011

We get the result as a negative number $-0011_2$

# Addition using 2's Complement

- **Case-1: $A_2$ + (-$B_2$) where |$A_2$| > |$B_2$|**
  - Same as **$A_2$ - $B_2$**, so apply 2's complement subtraction
- **Case-2: $A_2$ + (-$B_2$) where |$A_2$| < |$B_2$|**
  - Same as **$A_2$ - $B_2$**, so apply 2's complement subtraction
- **Case-3: (-$A_2$) + (-$B_2$)**
  - Find the 2's complement of both the negative numbers, and then add both these complemented numbers.
  - In this case, the carry will always be generated
  - Discard the carry and take the 2's complement of the remaining bits to yield the result as negative number.

# Addition using 2's Complement (Contd.)

Example: Add $-1011_2$ and $-1110_2$ in five-bit register

1. First express, the magnitudes of the negative numbers into 5-bits, i.e., 01011 and 01110

2. Now, find the 2's complement of both the numbers

   2's complement of 01011 = 10100+1 = 10101

   2's complement of 01110 = 10001+1 = 10010

3. Add the complemented numbers

   10101 + 10010 = 1 00111

4. Discard the carry and take 2's complement of 00111 to get 11001. On declaring it as a negative number, we get the desired result as $-11001_2$

# Binary Codes

- When numbers, letters or words are represented by a special group of symbols, it is said that they are being encoded and the group of symbols is called a code.
- The digital data involves group of binary bits (for representation, transmission and storage) which are also referred to as binary code.
- The binary code is represented by the number as well as alphanumeric characters.
- Advantages of the binary code:
  - Suited for computer applications & digital communication
  - Ease the analysis and designing of digital circuits

## Commonly used Binary Codes

- Weighted codes:
  - BCD or 8421 code
  - 2421 code
  - 5211 code

- Non-weighted codes:
  - Excess-3 code
  - Gray code
  - Alphanumeric codes
  - Error detection & correction code

# Binary Coded Decimal (BCD) or 8421 code

If each digit of a decimal number is represented by its binary equivalent, the result is a code called binary coded decimal or BCD. Since a decimal digit can be as large as 9, 4 bits are required to code each digit with place value from left to right as 8421 ($2^3$ $2^2$ $2^1$ $2^0$).

Example: $874_D$ = 1000 0111 0100 (BCD)

BCD is a sequential code as each succeeding code entry is one binary number greater than its preceding one

# 2421 and 5211 codes

These are also 4-bit codes and used for representing decimal number with bits carry the weight (from left to right) as indicated in their names

These codes possess the self-complementing property and therefore called as *reflective codes*

We can obtain 9’s complement (for subtraction) of a decimal number encoded in these codes by simply by inverting all bits

These codes are non-sequential codes

| Decimal Number | BCD (8421) Code | 2421 Code | 5211 Code |
|---|---|---|---|
| 0 | 0000 | 0000 | 0000 |
| 1 | 0001 | 0001 | 0001 |
| 2 | 0010 | 0010 | 0011 |
| 3 | 0011 | 0011 | 0101 |
| 4 | 0100 | 0100 | 0111 |
| 5 | 0101 | 1011 | 1000 |
| 6 | 0110 | 1100 | 1010 |
| 7 | 0111 | 1101 | 1100 |
| 8 | 1000 | 1110 | 1110 |
| 9 | 1001 | 1111 | 1111 |

# Excess-3 (XS-3) Code

- XS-3 code of a decimal number is derived by adding 3 to each decimal digit
- XS-3 is a non-weighted code but both reflective and sequential
- XS-3 addition and subtraction is more straightforward to implement than other decimal codes, especially BCD

| Decimal Number | BCD Code 8421 | Excess-3 Code (BCD + 0011) |
|---|---|---|
| 0 | 0000 | 0011 |
| 1 | 0001 | 0100 |
| 2 | 0010 | 0101 |
| 3 | 0011 | 0110 |
| 4 | 0100 | 0111 |
| 5 | 0101 | 1000 |
| 6 | 0110 | 1001 |
| 7 | 0111 | 1010 |
| 8 | 1000 | 1011 |
| 9 | 1001 | 1100 |

# Gray Code

This belongs to a class of codes called minimum-change codes, in which only one bit in the code groups changes when going from one stage to the next.

The Gray code is an unweighted code.

This code is not suited for arithmetic operation but finds application in input/output devices and some types of analog to digital converters.

Gray code is used in rotary and optical encoders, error detection in digit communication and the minimization of switching circuits or Karnaugh maps.

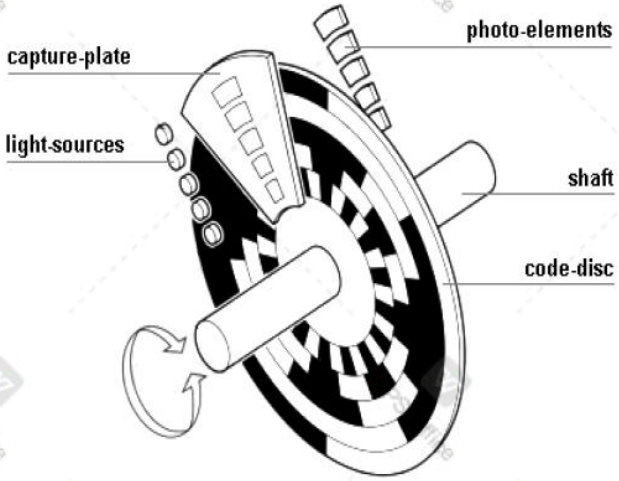
| Decimal Number | Binary equivalent | Gray Code |
|---|---|---|
| 0 | 0000 | 0000 |
| 1 | 0001 | 0001 |
| 2 | 0010 | 0011 |
| 3 | 0011 | 0010 |
| 4 | 0100 | 0110 |
| 5 | 0101 | 0111 |
| 6 | 0110 | 0101 |
| 7 | 0111 | 0100 |
| 8 | 1000 | 1100 |
| 9 | 1001 | 1101 |
| 10 | 1010 | 1111 |
| 11 | 1011 | 1110 |
| 12 | 1100 | 1010 |
| 13 | 1101 | 1011 |
| 14 | 1110 | 1001 |
| 15 | 1111 | 1000 |

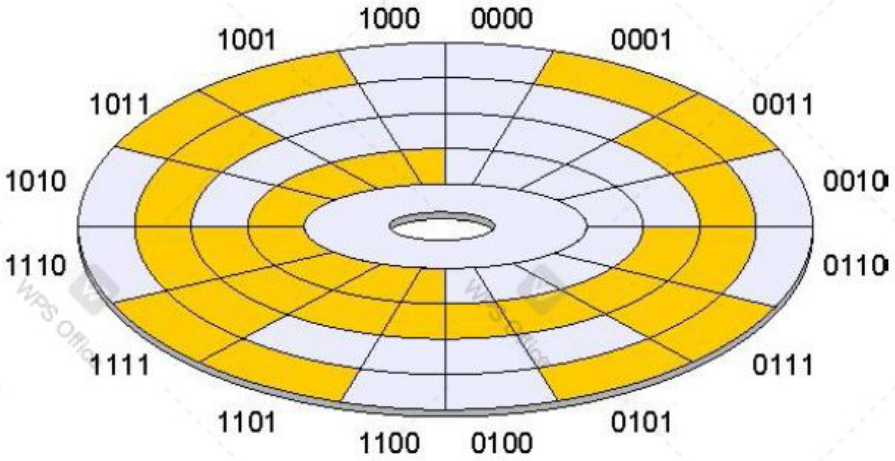
3 bits undergo change

All 4 bits undergo change

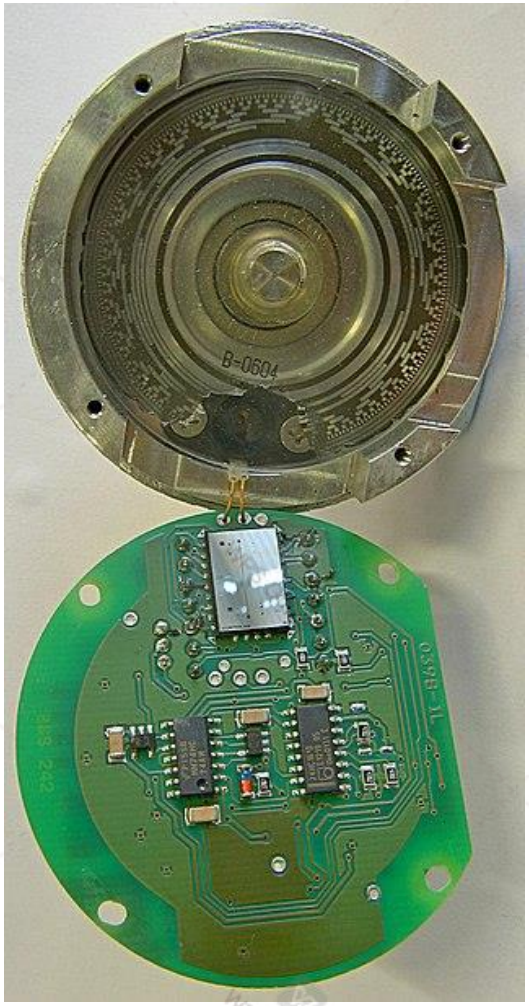
3 bits undergo change

# Shaft Angle Encoding




4-bit Gray coded disc used for shaft angle encoding



A Gray coded absolute rotary encoder with 13 tracks. Housing, interrupter disk, and light source are in the top; sensing element and support components are in the bottom.
Courtesy:
https://en.wikipedia.org/wiki/Gray_code

# Constructing *n*-bit Gray Code: Mirror Technique

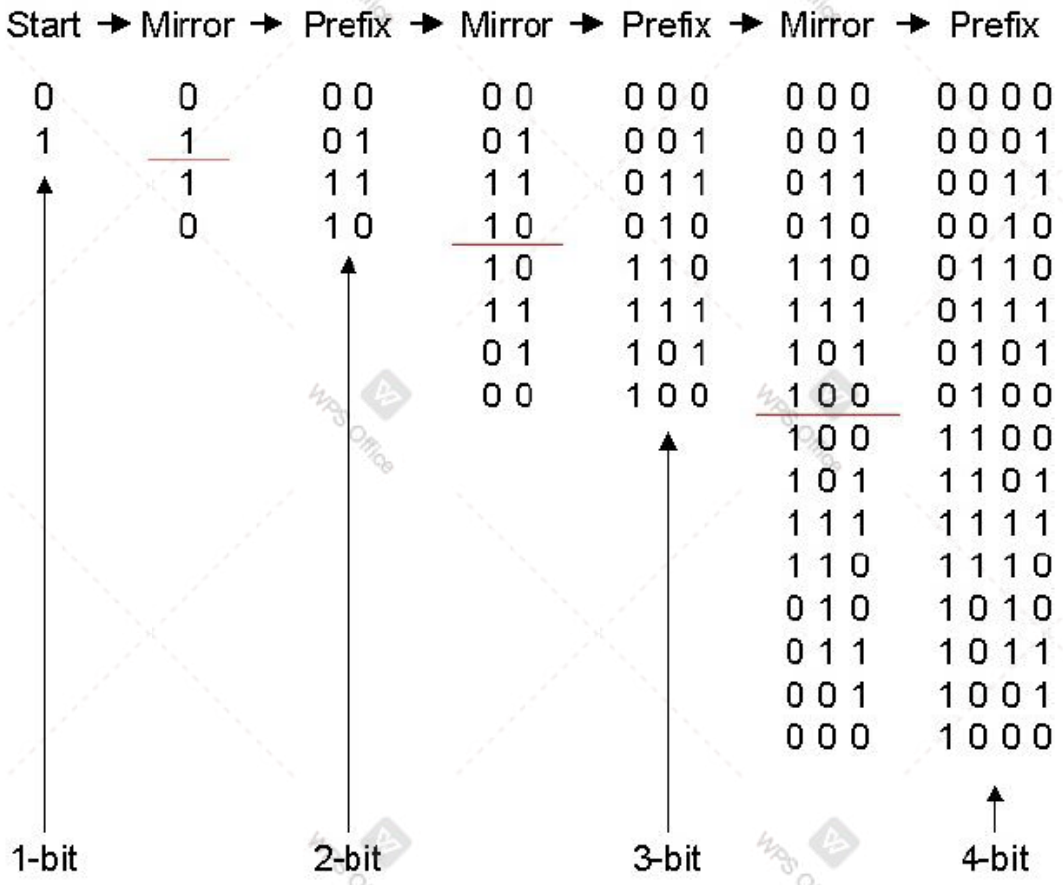
**Procedure:**

1) Commence with the simplest gray code possible; that is, for a single bit.

2) Create a mirror image of the existing gray code below the original values.

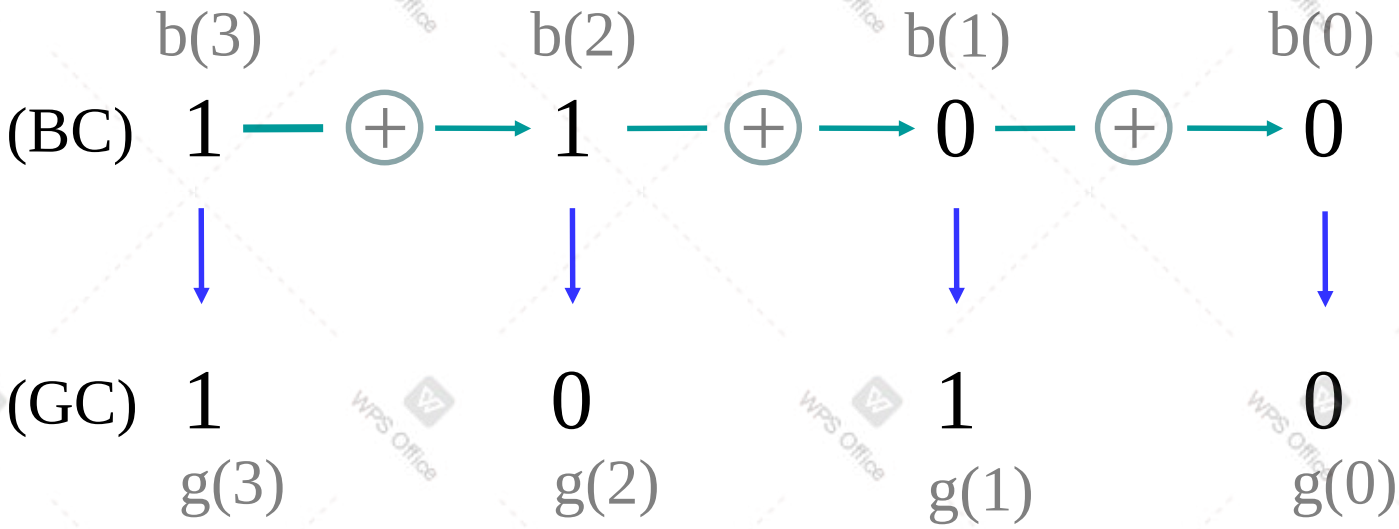
3) Prefix the original values with 0s and the mirrored values with 1s.

4) Repeat steps (2) and (3) until the desired code size is achieved.



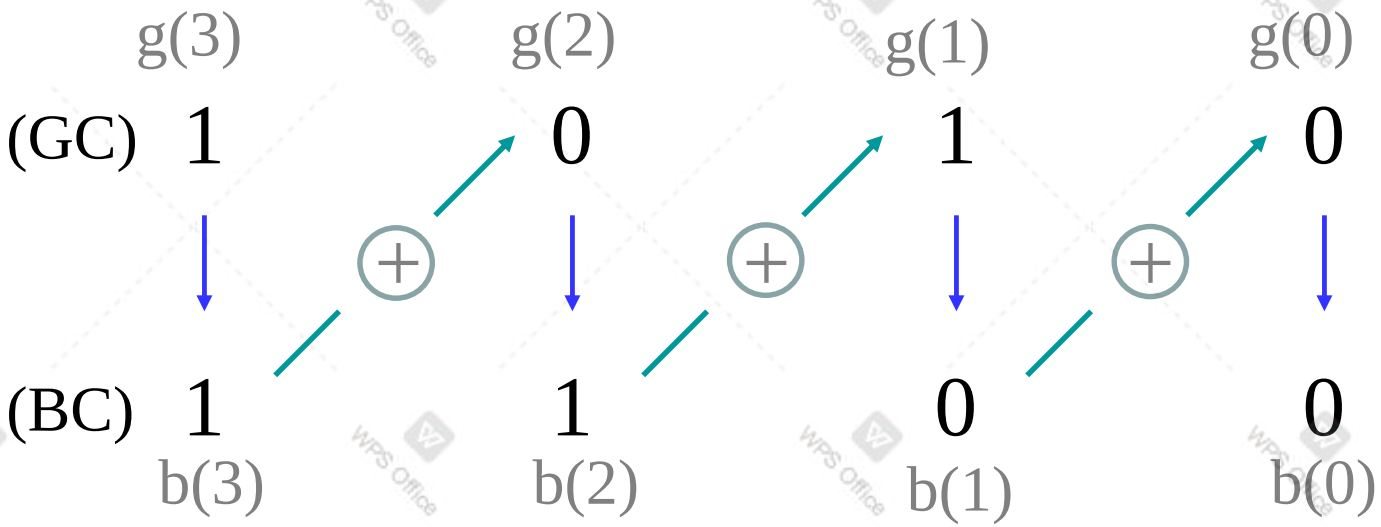

For this reason, the Gray code is also known as reflected binary code

# Binary to Gray Code Conversion




- Symbol $\oplus$ denotes XOR logic operation
- Note $g(3) = b(3)$,
  $g(2) = b(3) \oplus b(2)$,
  $g(1) = b(2) \oplus b(1)$,
  $g(0) = b(1) \oplus b(0)$

# Gray to Binary Code Conversion




- Symbol $\oplus$ denotes XOR logic operation
- Note $b(3) = g(3)$,

$$b(2) = b(3) \oplus g(2),$$
$$b(1) = b(2) \oplus g(1),$$
$$b(0) = b(1) \oplus g(0)$$

# Alphanumeric Codes

An alphanumeric code represents all of the various characters, functions and special symbols that are found in a standard typewriter or computer keyboard or displayed on Web.

## ASCII Code:

- The most widely used alphanumeric code, the American Standard Code for Information Interchange used in computers.
- The original ASCII code is a 7-bit code and so it represents $2^7$=128 unique characters. This code is made up of a 3-bit group, which is followed by a 4-bit code.
- The ASCII code starts from 00h to 7Fh. In this, the code from 00h to 1Fh is used for control characters, and the code from 20h to 7Fh is used for graphic symbols.

# ASCII Table

| | _0 | _1 | _2 | _3 | _4 | _5 | _6 | _7 | _8 | _9 | _A | _B | _C | _D | _E | _F |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| 0_ | U+0000 NUL 0 | U+0001 SOH 1 | U+0002 STX 2 | U+0003 ETX 3 | U+0004 EOT 4 | U+0005 ENQ 5 | U+0006 ACK 6 | U+0007 BEL 7 | U+0008 BS 8 | U+0009 HT 9 | U+000A LF 10 | U+000B VT 11 | U+000C FF 12 | U+000D CR 13 | U+000E SO 14 | U+000F SI 15 |
| 1_ | U+0010 DLE 16 | U+0011 DC1 17 | U+0012 DC2 18 | U+0013 DC3 19 | U+0014 DC4 20 | U+0015 NAK 21 | U+0016 SYN 22 | U+0017 ETB 23 | U+0018 CAN 24 | U+0019 EM 25 | U+001A SUB 26 | U+001B ESC 27 | U+001C FS 28 | U+001D GS 29 | U+001E RS 30 | U+001F US 31 |
| 2_ | U+0020 SP 32 | U+0021 ! 33 | U+0022 " 34 | U+0023 # 35 | U+0024 $ 36 | U+0025 % 37 | U+0026 & 38 | U+0027 ' 39 | U+0028 ( 40 | U+0029 ) 41 | U+002A * 42 | U+002B + 43 | U+002C , 44 | U+002D - 45 | U+002E . 46 | U+002F / 47 |
| 3_ | U+0030 0 48 | U+0031 1 49 | U+0032 2 50 | U+0033 3 51 | U+0034 4 52 | U+0035 5 53 | U+0036 6 54 | U+0037 7 55 | U+0038 8 56 | U+0039 9 57 | U+003A : 58 | U+003B ; 59 | U+003C < 60 | U+003D = 61 | U+003E > 62 | U+003F ? 63 |
| 4_ | U+0040 @ 64 | U+0041 A 65 | U+0042 B 66 | U+0043 C 67 | U+0044 D 68 | U+0045 E 69 | U+0046 F 70 | U+0047 G 71 | U+0048 H 72 | U+0049 I 73 | U+004A J 74 | U+004B K 75 | U+004C L 76 | U+004D M 77 | U+004E N 78 | U+004F O 79 |
| 5_ | U+0050 P 80 | U+0051 Q 81 | U+0052 R 82 | U+0053 S 83 | U+0054 T 84 | U+0055 U 85 | U+0056 V 86 | U+0057 W 87 | U+0058 X 88 | U+0059 Y 89 | U+005A Z 90 | U+005B [ 91 | U+005C \ 92 | U+005D ] 93 | U+005E ^ 94 | U+005F _ 95 |
| 6_ | U+0060 ` 96 | U+0061 a 97 | U+0062 b 98 | U+0063 c 99 | U+0064 d 100 | U+0065 e 101 | U+0066 f 102 | U+0067 g 103 | U+0068 h 104 | U+0069 i 105 | U+006A j 106 | U+006B k 107 | U+006C l 108 | U+006D m 109 | U+006E n 110 | U+006F o 111 |
| 7_ | U+0070 p 112 | U+0071 q 113 | U+0072 r 114 | U+0073 s 115 | U+0074 t 116 | U+0075 u 117 | U+0076 v 118 | U+0077 w 119 | U+0078 x 120 | U+0079 y 121 | U+007A z 122 | U+007B { 123 | U+007C \| 124 | U+007D } 125 | U+007E ~ 126 | U+007F DEL 127 |

ASCII characters are classified in four groups: control characters, special characters, numbers and letters

# Alphanumeric Codes (contd.)

## Extended ASCII Code:

- The extended ASCII is 8-bit code and supports 256 symbols where math and graphic symbols are added.
- The range of the extended ASCII is 80h to FFh.

## Unicode:

- Unicode is an industry standard for encoding of written text. Its aim is to provide a unique number to identify every character for every language, on any platform.
- It maps every character to a specific code, called **code point**. A code point takes the form of U+<hex-code> ranging from U+0000 to U+10FFFF.
- Unicode defines different characters encodings: UTF-8/16/32. The most used ones being UTF-8 especially on the Web.

# Error Detection and Correction Codes

- Error detection and correction code plays an important role in the transmission of data from one source to another.
- Due to noise, the bits of the data may change (either 0 to 1 or 1 to 0) during transmission.

## Error Detecting Codes:

- The error detection codes are the code used for detecting the error in the received data bitstream. In these codes, some bits are included appended to the original bitstream.
- Popular error detection techniques are: Parity check, Checksum, and Cyclic redundancy check (CRC).
- In parity codes, we add one parity bit either to the right of the LSB or left to the MSB to the original bitstream. Two types of parity codes are possible, i.e., even parity code and odd parity code.

# Error Correcting Codes:

- The error-correcting codes find the correct number of corrupted bits and their positions in the message.
- There are two types of error correction codes: block code and convolution code.
- In block codes, the message is contained in fixed-size blocks of bits. In this, the redundant bits are added for correcting and detecting errors.
- Hamming code is an example of a block code. The two simultaneous bit errors are detected, and single-bit errors are corrected by this code.
- In convolution code, the message consists of data streams of random length, and parity symbols are generated by the sliding application of the Boolean function to the data stream. The Hamming code is used for error correction.

# Addition using XS-3 Code

## Steps for addition:

1. Convert numbers to XS-3 form by adding 0011 to BCD code of each decimal digit

2. Add the converted numbers using the laws of basic binary addition

3. Now, add 0011 to those groups which produced a carry and subtract 0011 from those which did not produce a carry during the addition

4. The obtained result is XS-3 form of the desired sum

# Addition using XS-3 Code (Contd.)

Example: Add $597_{10}$ and $365_{10}$ using XS-3 code

**1st Step:** Obtain the XS-3 forms of the given decimal numbers as: 1000 1100 1010 and 0110 1001 1000

**2nd Step:** Perform the addition of the converted numbers

Group-3 Group-2 Group-1

$$\begin{array}{r} 1000\ 1100\ 1010 \\ +\ 0110\ 1001\ 1000 \\ \hline 1111\ 0110\ 0010 \end{array}$$

**3rd Step:** Groups-1&2 produced a carry so add 0011 to them while Group-3 didn't so subtract 0011 from it. This yields

$$1100\ 1001\ 0101$$

**4th Step:** $11001001010 1 \xrightarrow{\text{XS}-3 \text{ to BCD}} 10010110001 0 \rightarrow 962_{10}$

# Subtraction using XS-3 Code

## Steps for subtraction:

1. Convert numbers to XS-3 form by adding 0011 to BCD code of each decimal digit

2. Subtract the converted numbers using the laws of binary subtraction

3. Now, subtract 0011 to those groups which involved a borrow from the next higher group and add 0011 to those which did not involve borrow during the subtraction

4. The obtained result is XS-3 form of the desired sum

# Subtraction using XS-3 Code (Contd.)

Example: Subtract $365_{10}$ from $554_{10}$ using XS-3 code

**1st Step:** Obtain the XS-3 forms of the minuend and subtrahend as: 1000 1000 0111 and 0110 1001 1000

**2nd Step:** Perform the subtraction of the converted numbers

Group-3 Group-2 Group-1

$$\begin{array}{r} 1000\ 1000\ 0111 \\ -\ 0110\ 1001\ 1000 \\ \hline 0001\ 1110\ 1111 \end{array}$$

**3rd Step:** Groups-1&2 involved a borrow so subtract 0011 from them while Group-3 didn't so add 0011 to it. This yields

0100 1011 1100

**4th Step:** $010010111100 \xrightarrow{\text{XS}-3 \text{ to BCD}} 000110001001 \rightarrow 189_{10}$

# Higher Bit BCD Codes & Error Detection

- A salient example of 5-bit BCD code is “2-out of-5” code which has exactly two 1s in each group.
- A salient example of 7-bit BCD code is “bi-quinary” code which comprises a two-state (*bi*) and a five-state (*quinary*) components with each having exactly one 1
- The biquinary is a weighted code while the 2-out of-5 is an unweighted code
- For having even parity, both the code aid error detection

| Decimal digit | 2-out of-5 | Biquinary |
|---|---|---|
| | Unweighted | 50 43210 |
| 0 | 00011 | 01 00001 |
| 1 | 00101 | 01 00010 |
| 2 | 00110 | 01 00100 |
| 3 | 01001 | 01 01000 |
| 4 | 01010 | 01 10000 |
| 5 | 01100 | 10 00001 |
| 6 | 10001 | 10 00010 |
| 7 | 10010 | 10 00100 |
| 8 | 10100 | 10 01000 |
| 9 | 11000 | 10 10000 |

# LOGIC GATES

# Logic Gates

Digital circuits called logic gates can be constructed from diodes, transistors, and resistors in such a way that the circuit output is the result of a basic logic operation (OR, AND, NOT) performed on the circuits.

Logic gates which do more complicated logical operations are also widely available, e.g. NOR, NAND, Ex-OR, etc.

# Boolean or Switching Algebra

Boolean or switching algebra is used to analyze and simplify the digital (logic) circuits

The logic gates being digital circuits can be algebraically described using Boolean algebra

In Boolean algebra, the variables can take on only logical values: 0 (False) and 1 (True)

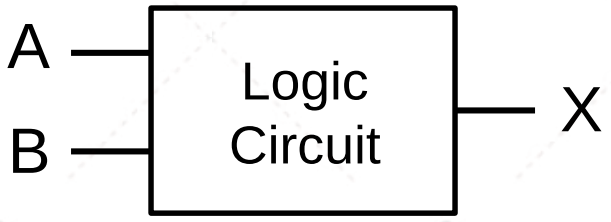
*Note, 1 and 0 are NOT NUMBERS. They merely represent the logical variables TRUE and FALSE*

The basic operations in Boolean algebra are **AND**, **OR** and **NOT**

# Logic Circuit and Truth Table

The logic circuits comprise one more logic gates interconnect in specific manner to achieve the desired logic

A truth table is a tabular representation for describing how a logic circuit's output depends on the logic levels present at the circuit's inputs




Truth table

| A | B | X |
|---|---|---|
| 0 | 0 | 0 |
| 0 | 1 | 1 |
| 1 | 0 | 1 |
| 1 | 1 | 1 |

Binary counting sequence

For $n$-input logic circuit, the number of input combinations is $2^n$

# Function of Single Binary Variable

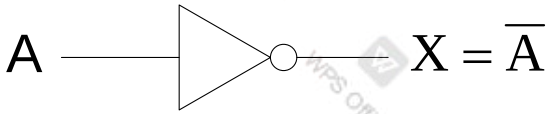
X = f(A)

Examples:  NOT (Inversion or Complement)

Buffer or Double-inversion

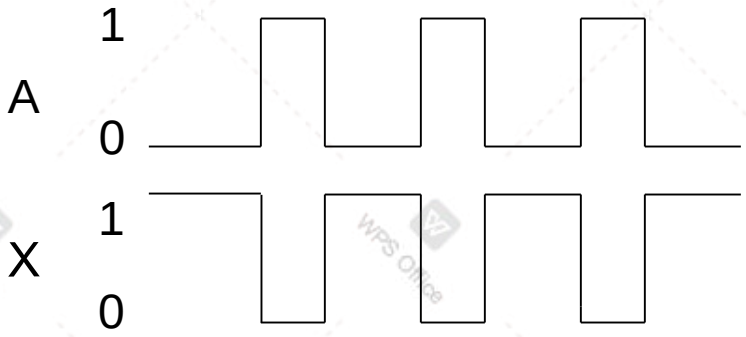
# NOT Operation $X = \overline{A}$

NOT gate or Inverter




| A | $X = \overline{A}$ |
|---|---|
| 0 | 1 |
| 1 | 0 |

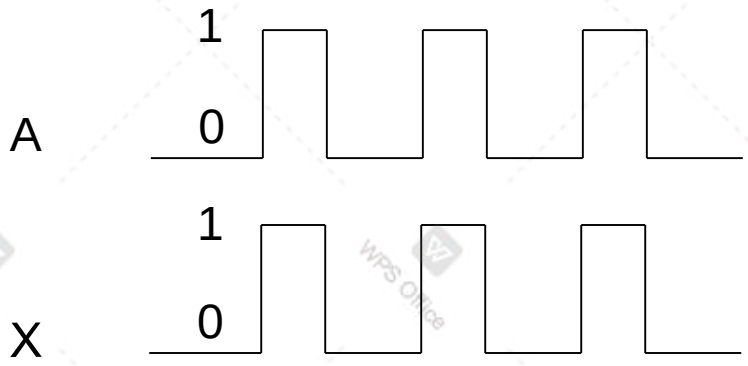
Truth Table




Time Diagram of NOT Gate

# Buffer Operation $X = A$

Buffer gate or Double Inverter


Time Diagram of Buffer Gate

| A | $X = A$ |
|---|---|
| 0 | 0 |
| 1 | 1 |

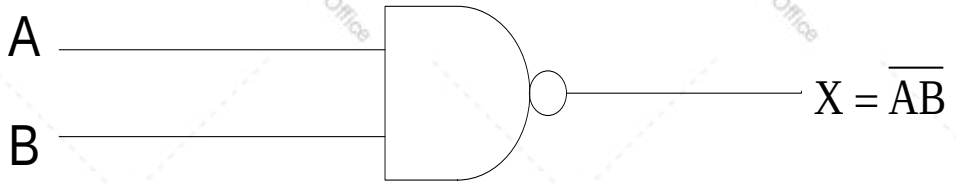
Truth Table

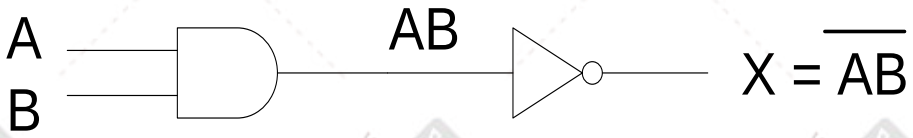
# Function of Two or More Binary Variables

$X = f(A,B,\ldots)$

Examples:
- NAND (Not-AND)
- NOR (Not-OR)
- AND
- OR
- Ex-OR
- Ex-NOR

# NAND gate

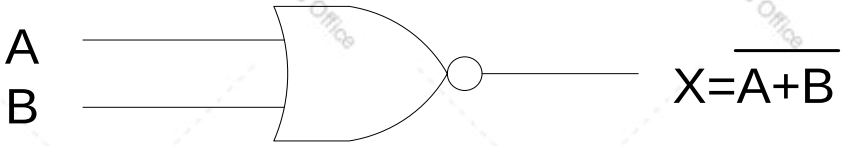


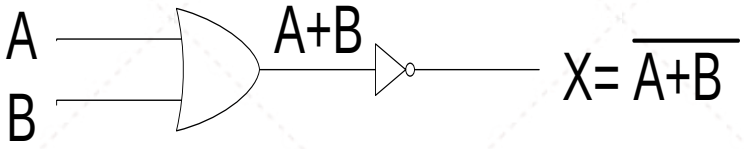




***Truth Table for NAND Gate***

| A | B | AB | $\overline{AB}$ |
|---|---|---|---|
| 0 | 0 | 0 | 1 |
| 0 | 1 | 0 | 1 |
| 1 | 0 | 0 | 1 |
| 1 | 1 | 1 | 0 |

# NOR gate



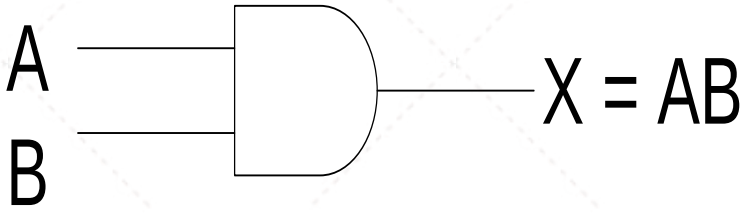




***Truth Table for NOR Gate***

| A | B | A+B | $\overline{A+B}$ |
|---|---|---|---|
| 0 | 0 | 0 | 1 |
| 0 | 1 | 1 | 0 |
| 1 | 0 | 1 | 0 |
| 1 | 1 | 1 | 0 |

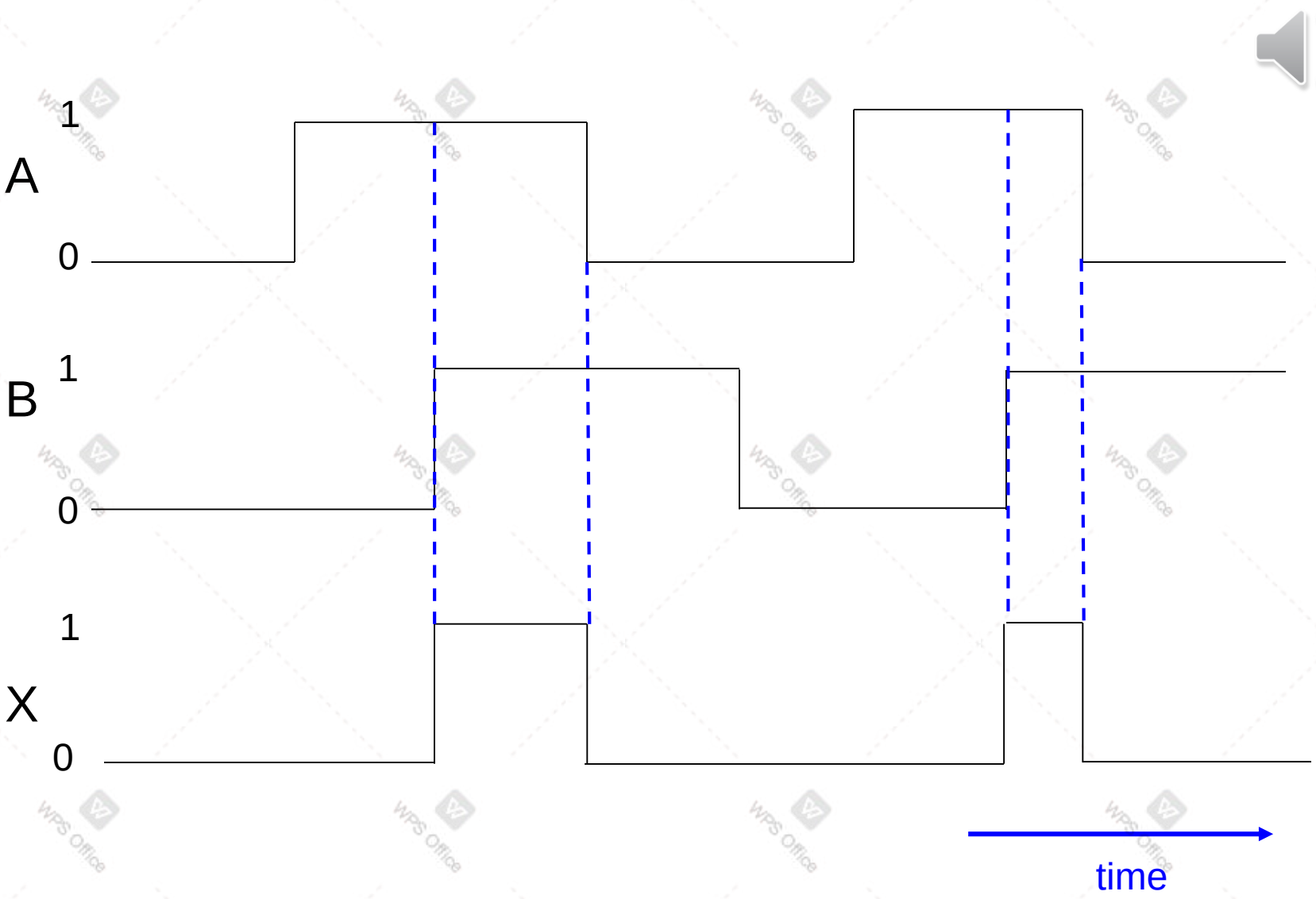
# AND Operation X = AB

AND gate




**Truth Table**

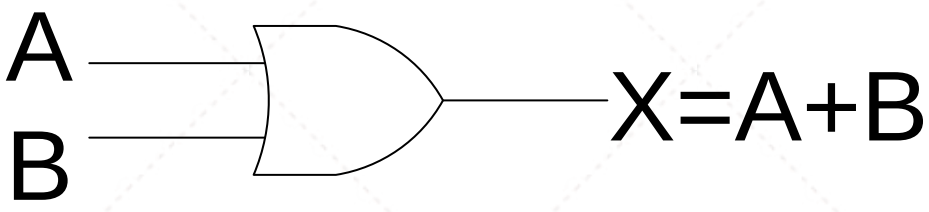
| A | B | X |
|---|---|---|
| 0 | 0 | 0 |
| 0 | 1 | 0 |
| 1 | 0 | 0 |
| 1 | 1 | 1 |




**Time Diagram of 2-input AND Gate**

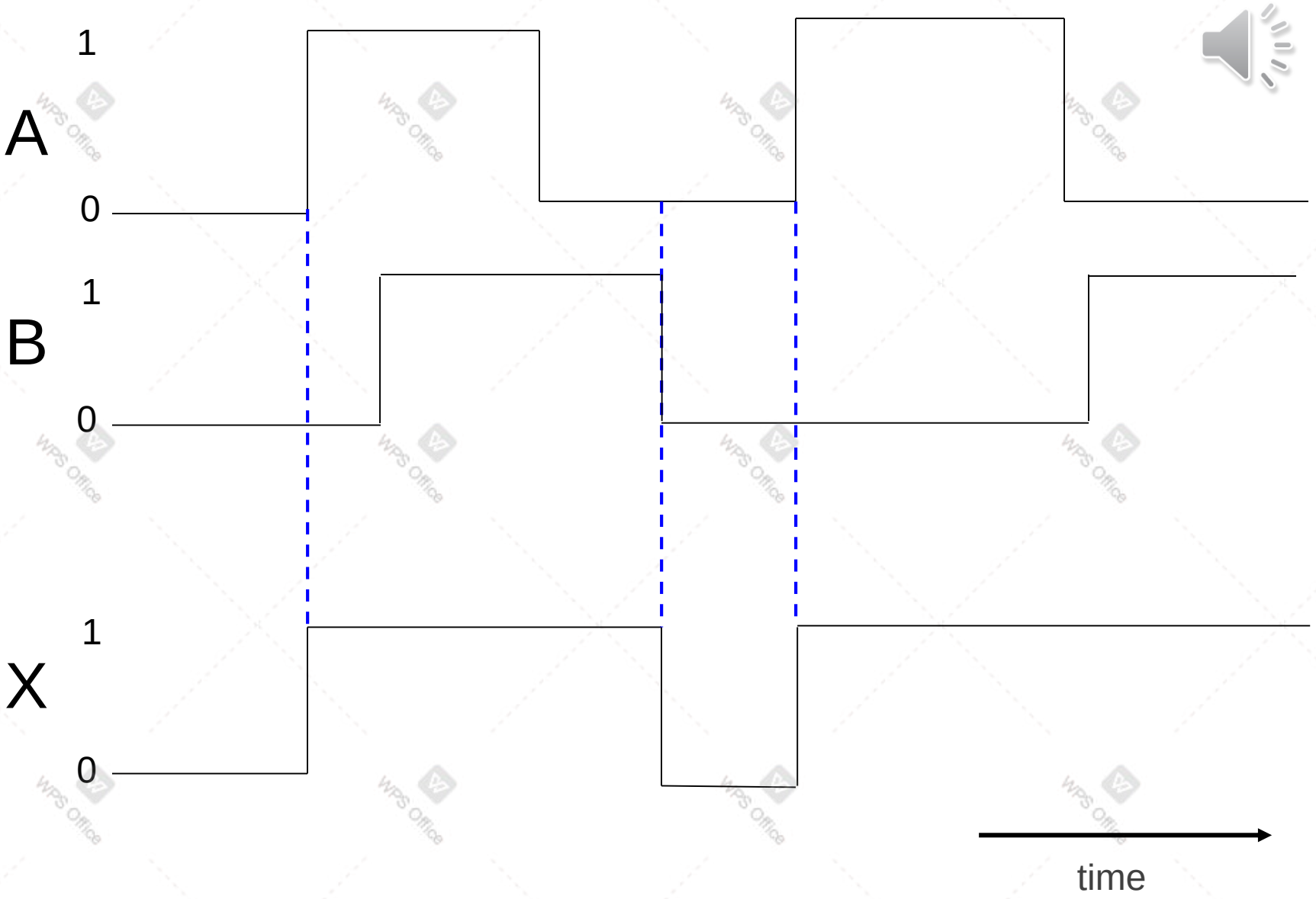
# OR Operation X = A+B

OR gate



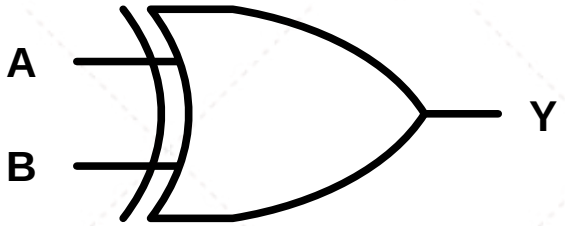

**Truth Table**

| A | B | X |
|---|---|---|
| 0 | 0 | 0 |
| 0 | 1 | 1 |
| 1 | 0 | 1 |
| 1 | 1 | 1 |




Time diagram of 2-input OR Gate

# Exclusive-OR (XOR) Gate




$$Y = A\overline{B} + \overline{A}B = A \oplus B$$

**symbol $\oplus$ denotes XOR operation**

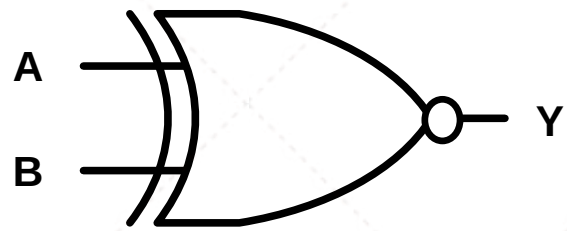
**Comparator:** Output is LOW (0) if both the inputs are identical; else output is HIGH (1)

**Half-Adder:** Y is the sum of A and B (in binary) *ignoring the carry out that will be generated by* 1+1

Truth Table

| A | B | Y |
|---|---|---|
| 0 | 0 | 0 |
| 0 | 1 | 1 |
| 1 | 0 | 1 |
| 1 | 1 | 0 |

# Exclusive-NOR (XNOR) Gate




$$\mathbf{Y} = \boldsymbol{AB} + \overline{\mathbf{A}}\,\overline{\mathbf{B}} = \mathbf{A} \odot \mathbf{B}$$

**Symbol ⊙ denotes XNOR operation**

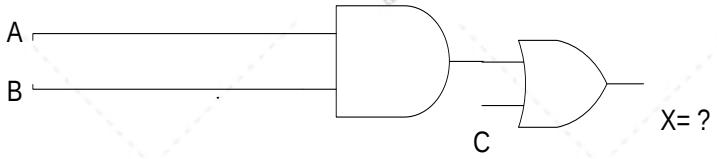
**Output inverted XOR gate:**
Output is HIGH (1) if both the inputs are identical; else output is LOW (0)

XNOR gates are used in parity checking, arithmetic and encryption circuits

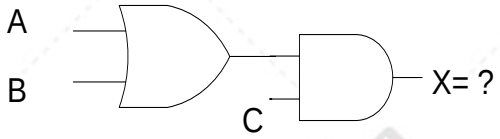
Truth Table

| A | B | Y |
|---|---|---|
| 0 | 0 | 1 |
| 0 | 1 | 0 |
| 1 | 0 | 0 |
| 1 | 1 | 1 |

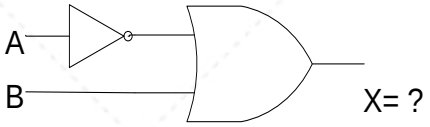
# Describing Logic Circuits Algebraically



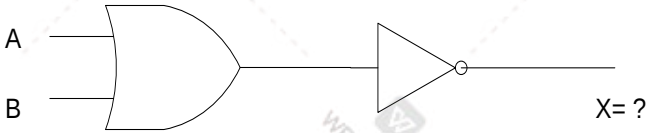

$$X = AB + C$$




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

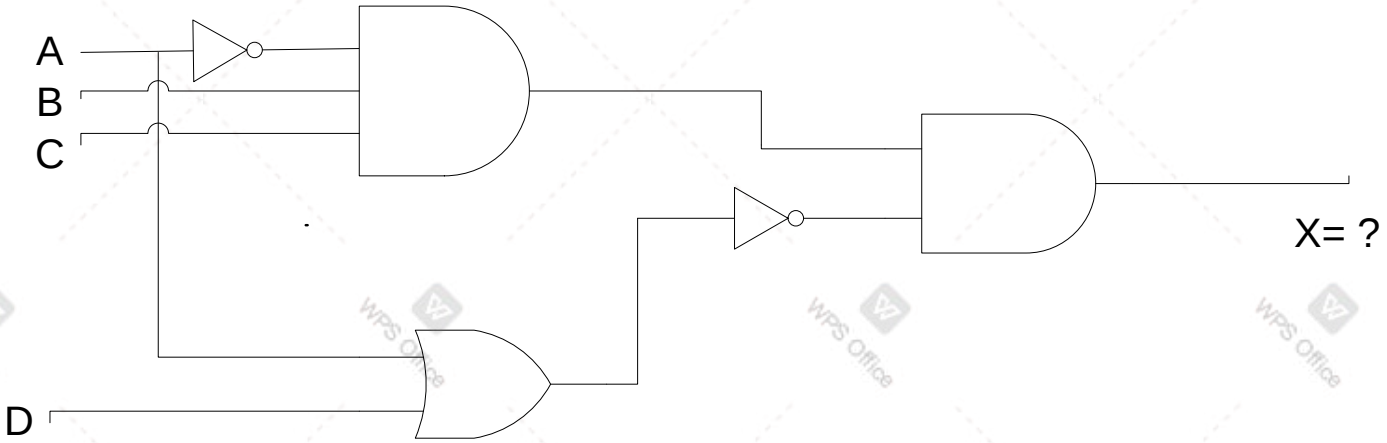



$$X = \overline{A} + B$$



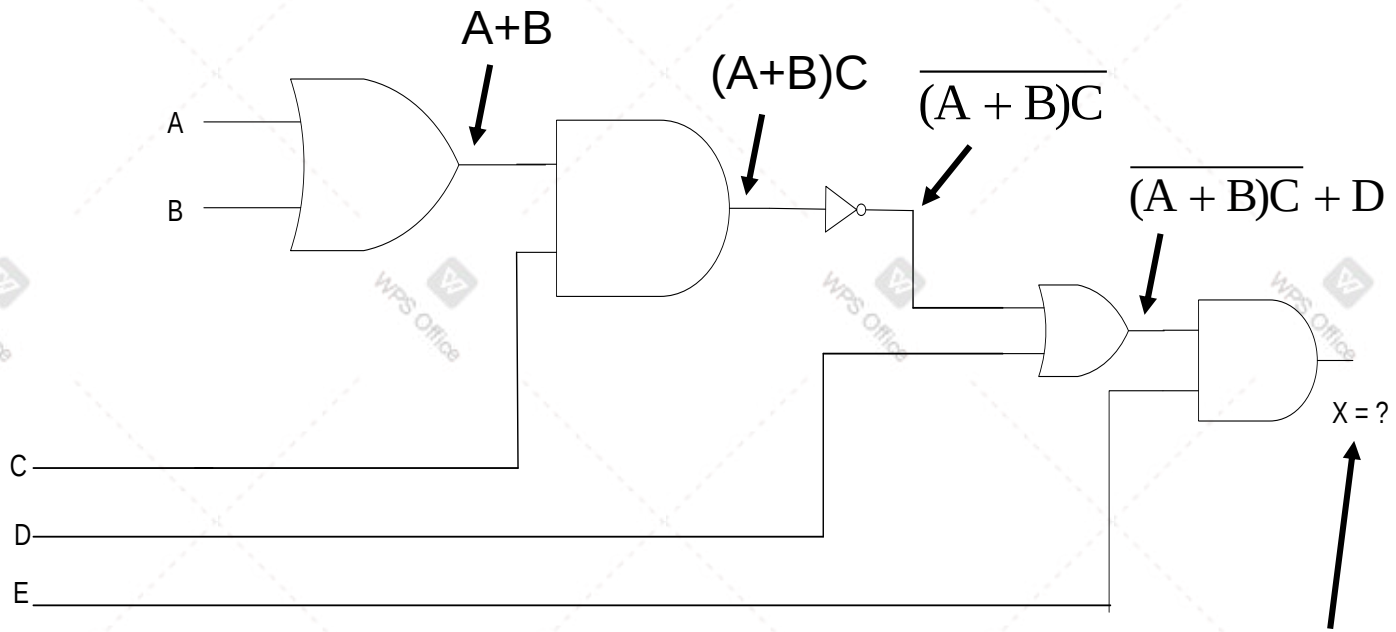

$$X = (\overline{A + B})$$

# Find the Output Expression of Logic Circuit



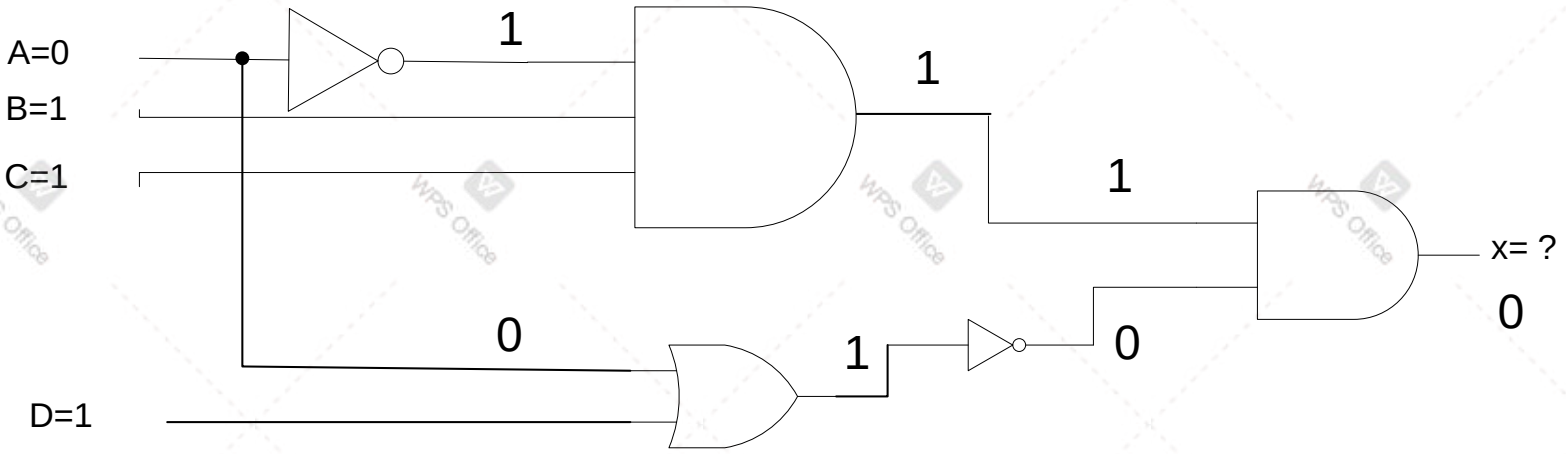

$$X = \overline{A}BC(\overline{A + D})$$

# Find the Output Expression of Logic Circuit




$$X = \left(\overline{(A+B)C} + D\right)E$$

# Determining output level from a time diagram



# Boolean (Switching) Algebra

# Boolean (Switching) Algebra Postulates

- Assume **A**, **B**, and **C** are logic variables that can have the values **0** (false) and **1** (true)
- Operators and their symbols: **OR** denoted by "**+**", **AND** denoted by "**·**", **NOT A** denoted by either " $\overline{\mathbf{A}}$ " or " $\mathbf{A}'$ "
- **Duality principle**: any Boolean algebraic equality remains valid whenever the **OR** and **AND** operators, and identity elements **0** and **1**, have been interchanged

| Indx. | Identity | Dual Identity | Name of law |
|---|---|---|---|
| (1) | $\mathbf{A} + \mathbf{0} = \mathbf{A}$ | $\mathbf{A} \cdot \mathbf{1} = \mathbf{A}$ | identity |
| (2) | $\mathbf{A} + \overline{\mathbf{A}} = \mathbf{1}$ | $\mathbf{A} \cdot \overline{\mathbf{A}} = \mathbf{0}$ | complementarity |
| (3) | $\mathbf{A} + \mathbf{B} = \mathbf{B} + \mathbf{A}$ | $\mathbf{A} \cdot \mathbf{B} = \mathbf{B} \cdot \mathbf{A}$ | commutative |
| (4) | $\mathbf{A} + (\mathbf{B} + \mathbf{C}) = (\mathbf{A} + \mathbf{B}) + \mathbf{C}$ | $\mathbf{A} \cdot (\mathbf{B} \cdot \mathbf{C}) = (\mathbf{A} \cdot \mathbf{B}) \cdot \mathbf{C}$ | associative |
| (5) | $\mathbf{A} + (\mathbf{B} \cdot \mathbf{C}) = (\mathbf{A} + \mathbf{B}) \cdot (\mathbf{A} + \mathbf{C})$ | $\mathbf{A} \cdot (\mathbf{B} + \mathbf{C}) = (\mathbf{A} \cdot \mathbf{B}) + (\mathbf{A} \cdot \mathbf{C})$ | distributive |

# Proof of Distributive Law

For simplicity, often the dot "·" is omitted in writing

Dual form: $A + BC = (A + B)(A + C)$

Proof:

$$\begin{aligned} \text{RHS} &= (A + B)(A + C) \\ &= AA + AB + AC + BC \\ &= A + AB + AC + BC \\ &= A(1 + B) + AC + BC \\ &= A + AC + BC \\ &= A(1 + C) + BC \\ &= A + BC \\ &= \text{LHS} \end{aligned}$$

# Boolean Algebra Theorems

| Indx. | Identity | Dual Identity | Name of law |
|---|---|---|---|
| (6) | $\mathbf{A + A = A}$ | $\mathbf{A \cdot A = A}$ | idempotent |
| (7) | $\mathbf{A + 1 = 1}$ | $\mathbf{A \cdot 0 = 0}$ | annulment |
| (8) | $\mathbf{A + AB = A}$ | $\mathbf{A(A + B) = A}$ | absorption |
| (9) | $\mathbf{AB + A\overline{B} = A}$ | $\mathbf{(A + B)(A + \overline{B}) = A}$ | redundancy |
| (10) | $\mathbf{A + \overline{A}B = A + B}$ | $\mathbf{A(\overline{A} + B) = AB}$ | redundancy |
| (11) | $\mathbf{AB + \overline{A}C + BC = AB + \overline{A}C}$ | $\mathbf{(A + B)(\overline{A} + C)(B + C)} = \mathbf{(A + B)(\overline{A} + C)}$ | consensus |

# Some sample proofs

(8) $\mathbf{A + AB = A(1 + B) = A \cdot 1 = A}$

(9) $\mathbf{AB + A\bar{B} = A(B + \bar{B}) = A \cdot 1 = A}$

(10) $\mathbf{A + \bar{A}B = A + B}$ **Try proving this**

$$\mathbf{LHS = A(1 + B) + \bar{A}B}$$
$$\mathbf{= A + AB + \bar{A}B = A + (A + \bar{A})B = A + B = RHS}$$

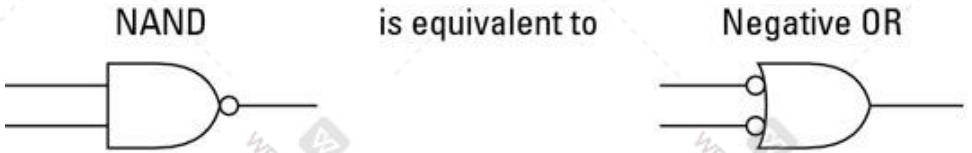
(11) $\mathbf{AB + \bar{A}C + BC = AB + \bar{A}C}$ **Try proving this**

$$\mathbf{LHS = AB + \bar{A}C + BC(A + \bar{A})}$$
$$\mathbf{= AB + \bar{A}C + ABC + \bar{A}BC}$$
$$\mathbf{= (AB + ABC) + (\bar{A}C + \bar{A}BC)}$$
$$\mathbf{= AB(1 + C) + \bar{A}C(1 + B)}$$
$$\mathbf{= AB + \bar{A}C = RHS}$$

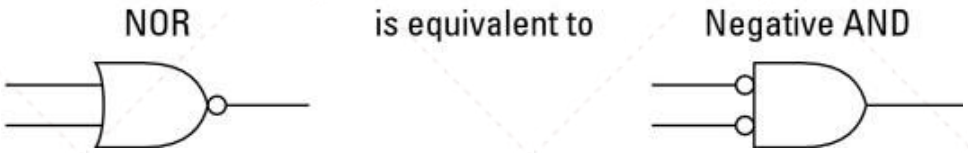
# Boolean Algebra Theorems (Contd.)

- **De Morgan's theorem**: an AND gate with inverted output behaves the same as an OR gate with inverted inputs and vice versa

$$\overline{\mathbf{AB}} = \overline{\mathbf{A}} + \overline{\mathbf{B}}$$




$$\overline{\mathbf{A} + \mathbf{B}} = \overline{\mathbf{A}}\,\overline{\mathbf{B}}$$




- Generalized De Morgan's theorem:

$$\overline{\mathbf{A} + \mathbf{B} + \mathbf{C} \cdots} = \overline{\mathbf{A}}\,\overline{\mathbf{B}}\,\overline{\mathbf{C}} \cdots$$

$$\overline{\mathbf{ABC} \cdots} = \overline{\mathbf{A}} + \overline{\mathbf{B}} + \overline{\mathbf{C}} \cdots$$

# Truth Table Proof of De Morgan's Laws

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

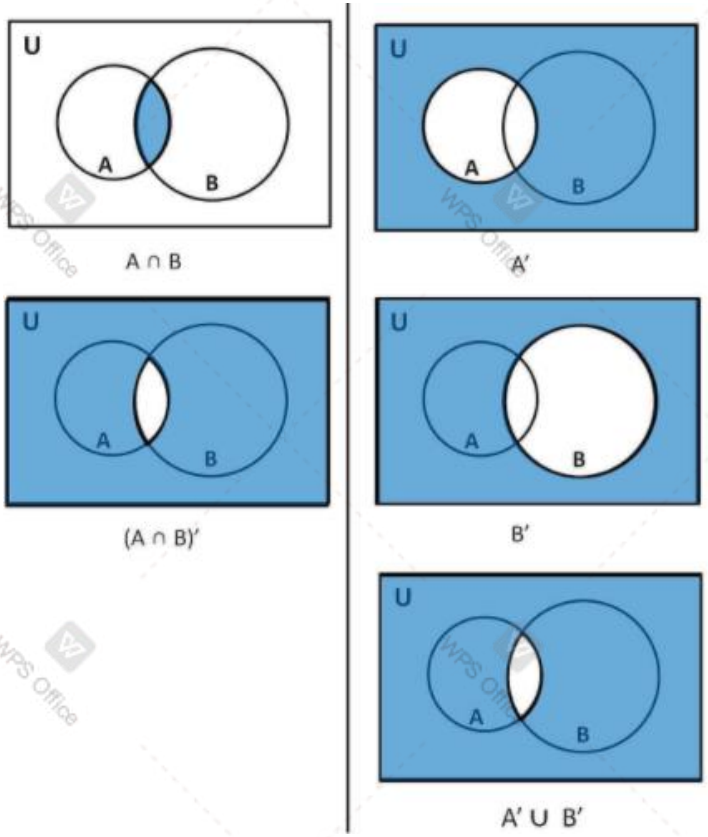
| $A$ | $B$ | $A'$ | $B'$ | $AB$ | $(AB)'$ | $A' + B'$ |
|---|---|---|---|---|---|---|
| 0 | 0 | 1 | 1 | 0 | 1 | 1 |
| 0 | 1 | 1 | 0 | 0 | 1 | 1 |
| 1 | 0 | 0 | 1 | 0 | 1 | 1 |
| 1 | 1 | 0 | 0 | 1 | 0 | 0 |

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

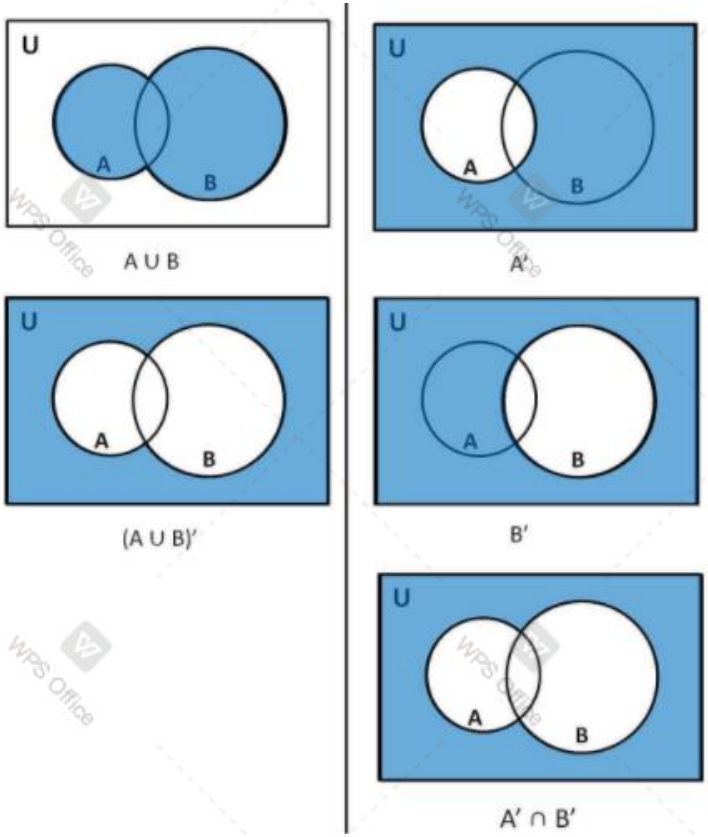
| $A$ | $B$ | $A'$ | $B'$ | $A + B$ | $(A + B)'$ | $A'B'$ |
|---|---|---|---|---|---|---|
| 0 | 0 | 1 | 1 | 0 | 1 | 1 |
| 0 | 1 | 1 | 0 | 1 | 0 | 0 |
| 1 | 0 | 0 | 1 | 1 | 0 | 0 |
| 1 | 1 | 0 | 0 | 1 | 0 | 0 |

# Venn Diagram Proof of De Morgan's Laws

$(A \cap B)' = A' \cup B'$




$(A \cup B)' = A' \cap B'$



# Algebraic Proof of De Morgan's Laws

- Consider $(A + B)' = A'B'$, which essentially states that $A'B'$ is complement to $(A + B)$
- Thus, we need to prove $(A + B) + A'B' = 1$ and $(A + B)(A'B') = 0$

**Case-1:** $(A + B) + A'B' = 1$

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

$$= AA + AB + AA' + A'B + A'B'$$

$$= A(1 + B) + A'(B + B') = A + A' = 1$$

**Case-2:** $(A + B)(A'B') = 0$

$$LHS = (A + B)A'B' = AA'B' + A'BB' = 0$$

# Boolean Algebra Theorems (Contd.)

- **Boole's / Shannon's expansion**: It expresses a logic function in terms of two functions with one fewer variable each

$$f(x_1, x_2, \cdots, x_n) = x_1 \cdot f(1, x_2, \cdots, x_n) + x_1' \cdot f(0, x_2, \cdots, x_n)$$

Dual form:

$$f(x_1, x_2, \cdots, x_n) = [x_1 + f(0, x_2, \cdots, x_n)] \cdot [x_1' + f(1, x_2, \cdots, x_n)]$$

Note, only $\cdot$, $+$, and $'$ operators are used above

- Recursively applying, it allows to synthesize any Boolean function using multiplexers only

# Examples

- Simplify $Z = \overline{(\bar{A} + C) \cdot (B + \bar{D})}$

$$= \overline{(\bar{A} + C)} + \overline{(B + \bar{D})} \qquad \text{De Morgan's law}$$

$$= (\bar{\bar{A}} \cdot \bar{C}) + (\bar{B} \cdot \bar{\bar{D}}) \qquad \text{De Morgan's law}$$

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

- Simplify $Z = \bar{A}C\overline{(\bar{A}BD)} + \bar{A}B\bar{C}\bar{D} + A\bar{B}C$

$$= \bar{A}C(\bar{\bar{A}} + \bar{B} + \bar{D}) + \bar{A}B\bar{C}\bar{D} + A\bar{B}C \qquad \text{De Morgan's law}$$

$$= \bar{A}C(A + \bar{B} + \bar{D}) + \bar{A}B\bar{C}\bar{D} + A\bar{B}C \qquad \text{Double negation}$$

$$= \bar{A}CA + \bar{A}C\bar{B} + \bar{A}C\bar{D} + \bar{A}B\bar{C}\bar{D} + A\bar{B}C \qquad \text{Distributive law}$$

$$= 0 + \bar{B}C(A + \bar{A}) + \bar{A}C\bar{D} + \bar{A}B\bar{C}\bar{D} \qquad \text{Compl+Assoc law}$$

$$= \bar{B}C + \bar{A}\,\bar{D}(C + B\bar{C}) \qquad \text{Compl+Assoc law}$$

$$= \bar{B}C + \bar{A}\,\bar{D}(C + B) \qquad \text{Redundancy law}$$

# Universality of NAND and NOR Gates

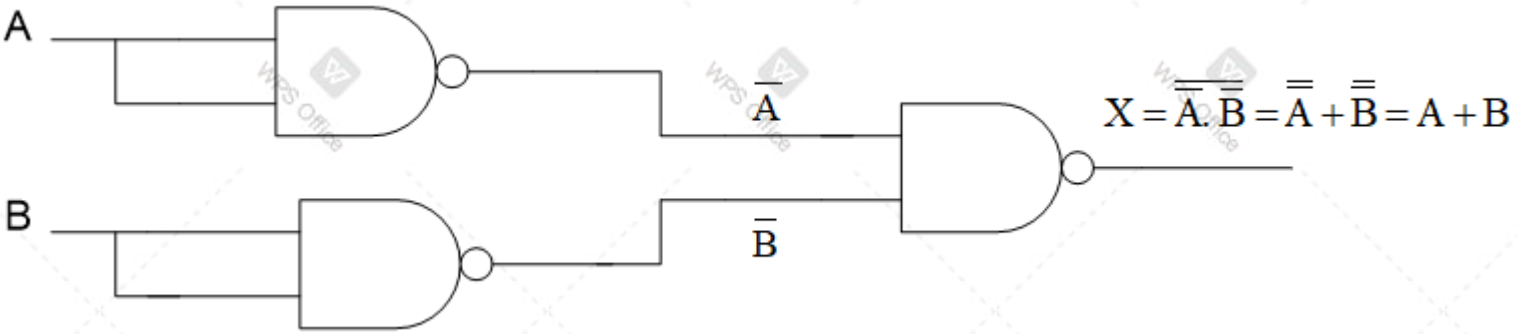
- All the Boolean expressions can be implemented using various combinations of OR, AND and NOT gates.
- Again NAND or NOR gates in proper combination can be used to perform each of the Boolean operations OR, AND, and NOT.
- Any logic expression can be implemented using **ONLY** NAND or NOR gates.
- Therefore, NAND and NOR gates are known as Universal Gates

# Realizing Basic Gates Using NAND Gates

- NOT gate

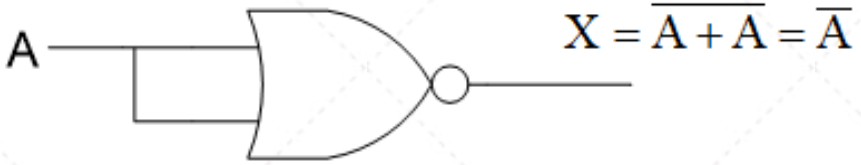

- AND gate


- OR gate

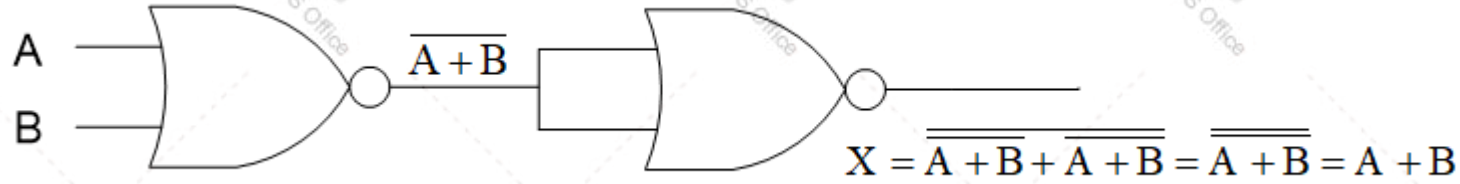


# Realizing Basic Gates Using NOR Gates

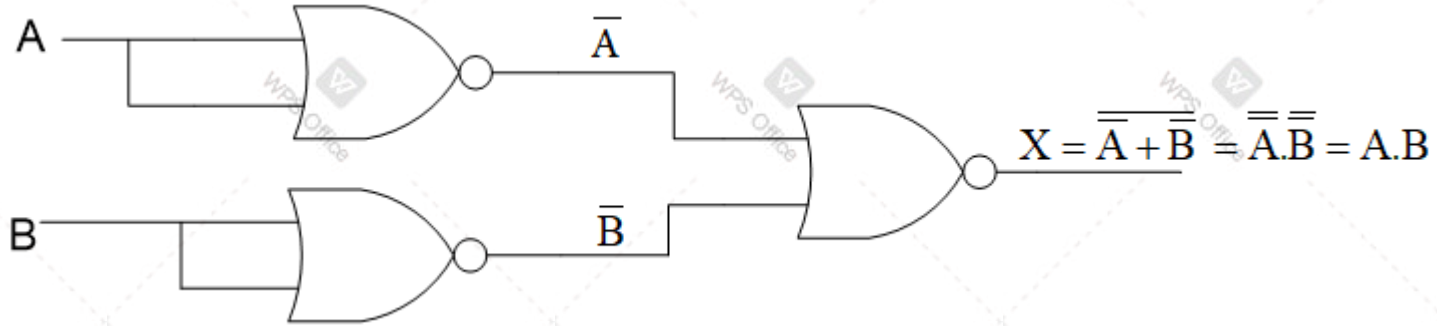
- NOT gate




- OR gate

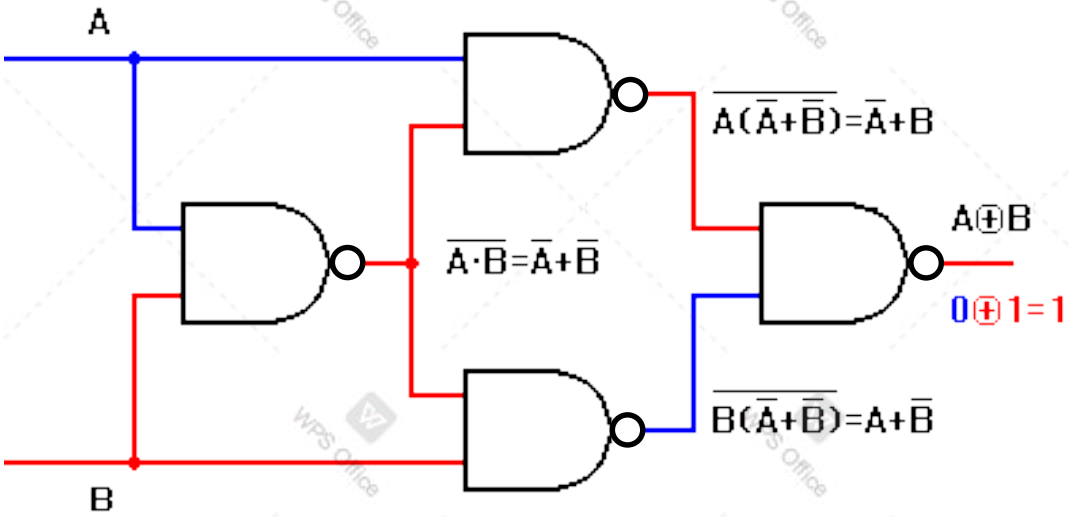



- AND gate



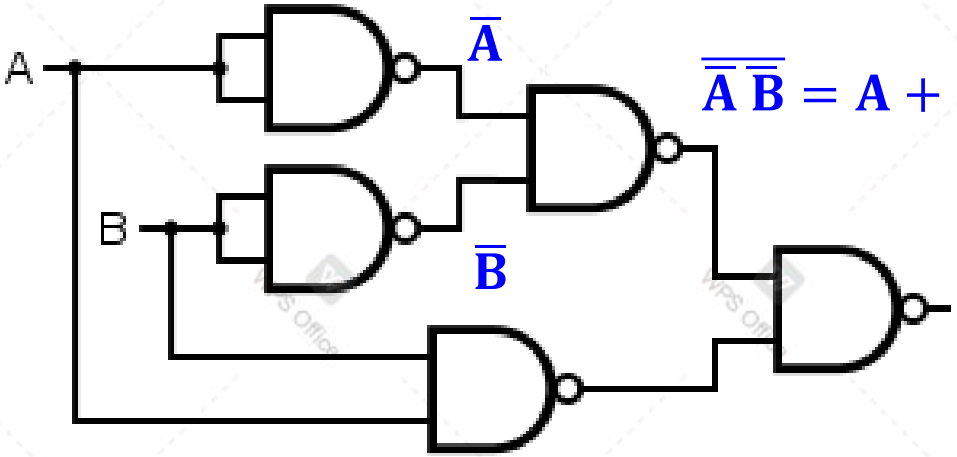
# Realizing XOR & XNOR using NAND Gates Only

XOR




$$Y = \overline{(\bar{A}+B)(A+\bar{B})}$$
$$= A\bar{B} + \bar{A}B$$
$$= A \oplus B$$

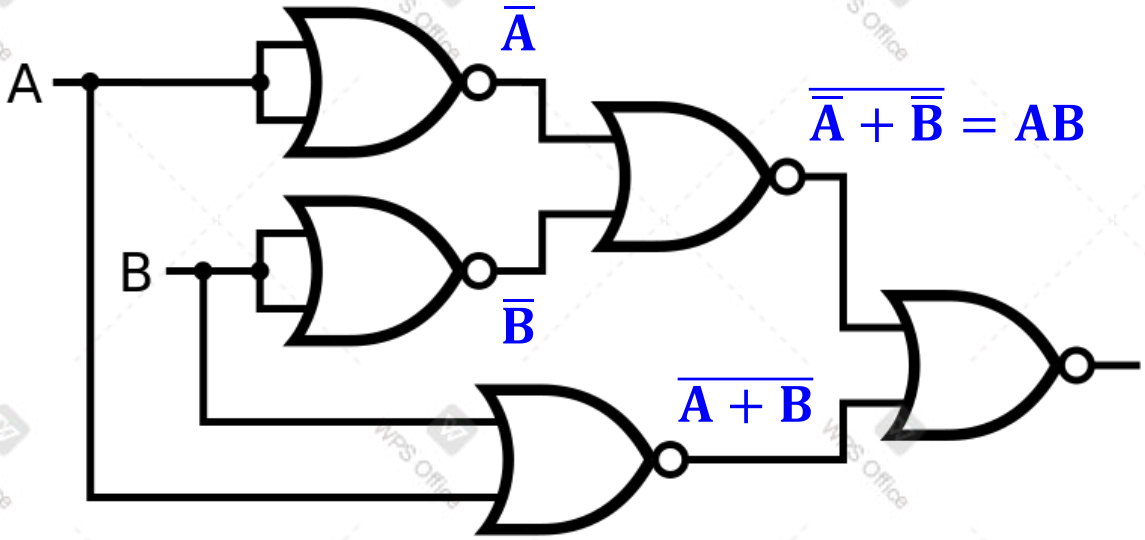
XNOR




$$Y = \overline{\overline{AB}\,(A+B)}$$
$$= \overline{\overline{AB}} + \overline{(A+B)}$$
$$= AB + \bar{A}\,\bar{B} = A \odot B$$

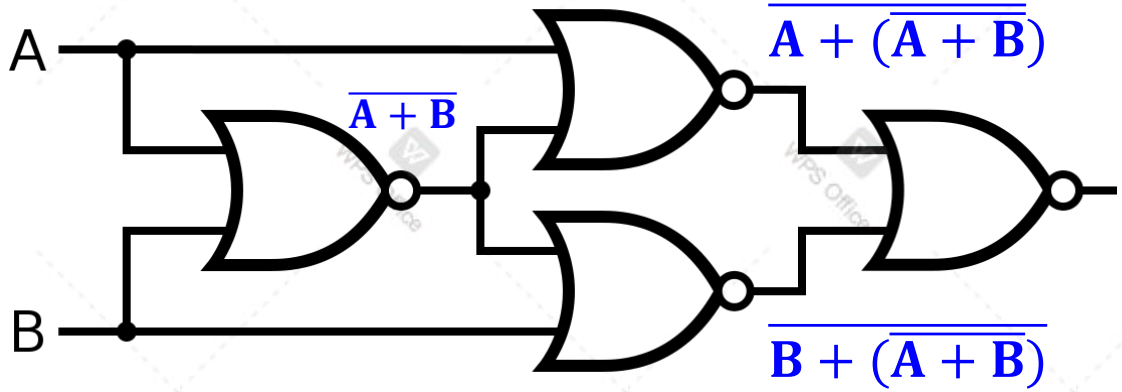
# Realizing XOR & XNOR using NOR Gates Only

XOR




$$Y = \overline{AB + (\overline{A+B})}$$
$$= (\overline{A}+\overline{B})(A+B)$$
$$= A\overline{B} + \overline{A}B$$
$$= A \oplus B$$

XNOR




$$Y = A \odot B$$

## Combinational Logic Circuits
## or **Combinational Circuits**

The digital circuits made up of combination of logic gates discussed earlier can be classified as combinational logic circuits because at any time, the logic level at the output depends on the combination of logic levels present at the inputs.

A combinational circuit has no memory, so its output depends only on the current value of its inputs.

## Standard Forms For Logical Expression

For the purposes of simplification and design of the logic circuits, we try to express their logical expressions in one of two standard forms:

Sum-of-Products (SOP) Form

or

Product-of-Sums (POS) Form

# Sum-of-Products (SOP) Form

Examples: (i) $ABC+A\bar{B}\bar{C}$ (ii) $AB+A\bar{B}\bar{C}+C\bar{D}+\bar{D}$

Note that each expression consists of two or more **AND** terms (products) that are **ORed** (summed) together such that:

(i) The same variable never appears twice in a product. (Because $A.A = A$ and $A.\bar{A} = 0$)

(ii) One complement sign can not cover more than one variable in a term (e.g., we can not have $\overline{ABC}$ as we can then use De Morgan's law to simplify

## Example-1

Express the following Boolean function as sum of products, i.e., SOP form

$$f(A, B, C, D) = (\overline{A} + BC)(B + CD)$$

**Solution:**

$$f(A, B, C, D) = (\overline{A} + BC)(B + CD)$$

$$= \overline{A}B + \overline{A}CD + BBC + BCCD$$

$$= \overline{A}B + \overline{A}CD + BC + BCD$$

SOP form

## Example-2

Express the following Boolean function as sum of products

$$f(A, B, C, D, E) = (A + \overline{BC})(\overline{D + BE})$$

**Solution:**

$$\begin{aligned} f(A, \cdots, E) &= (A + \overline{BC})(\overline{D + BE}) \\ &= (A + \overline{B} + \overline{C})[\overline{D}(\overline{B} + \overline{E})] \\ &= (A + \overline{B} + \overline{C})(\overline{B}\,\overline{D} + \overline{D}\,\overline{E}) \\ &= A\overline{B}\,\overline{D} + \overline{B}\,\overline{B}\,\overline{D} + \overline{C}\,\overline{B}\,\overline{D} + A\overline{D}\,\overline{E} + \overline{B}\,\overline{D}\,\overline{E} + \overline{C}\,\overline{D}\,\overline{E} \\ &= A\overline{B}\,\overline{D} + A\overline{D}\,\overline{E} + \overline{B}\,\overline{C}\,\overline{D} + \overline{B}\,\overline{D} + \overline{B}\,\overline{D}\,\overline{E} + \overline{C}\,\overline{D}\,\overline{E} \end{aligned}$$

# Canonical SOP Form

A SOP form in which each product term involves all the variables (complemented or uncomplemented) is referred to canonical SOP form

In canonical SOP form, each individual product term is referred to as a **minterm**

Example: $f(A, B, C) = \overline{A}BC + A\overline{B}\overline{C} + A\overline{B}C$

Canonical SOP form

On simplifying we have, $f(A, B, C) = \overline{A}BC + A\overline{B}$

As 2nd prod no longer has all literals, canonical form is lost

## Product of Sums (POS) Form

In this form a Boolean function consists of two or more **OR** terms (sums) that are **ANDed** together. Each OR term contains two or more variables in complemented or uncomplemented form such that:

(i) The same variable used never appears twice in a sum. (Because $A + A = A$ and $A + \overline{A} = 1$)

(ii) One complement sign can not cover more than one variable in a term

E.g. $f(A, B, C) = (A + \overline{B} + C)(A + C)$

# Canonical POS Form

When a Boolean function is expressed as a POS, where each term consists of a sum, the sum involving all of the variables in either complemented or uncomplemented form

In canonical POS form, each individual sum term is referred to as a **maxterm**

E.g. $f(A, B, C) = (A + \overline{B} + C)(\overline{A} + B + C)$

# Minterms and Maxterms of 3 Variables

| A | B | C | Minterms | | Maxterms | |
|---|---|---|---|---|---|---|
| | | | Representing term | Designation | Representing term | Designation |
| 0 | 0 | 0 | $\bar{A} \cdot \bar{B} \cdot \bar{C}$ | $m_0$ | $A + B + C$ | $M_0$ |
| 0 | 0 | 1 | $\bar{A} \cdot \bar{B} \cdot C$ | $m_1$ | $A + B + \bar{C}$ | $M_1$ |
| 0 | 1 | 0 | $\bar{A} \cdot B \cdot \bar{C}$ | $m_2$ | $A + \bar{B} + C$ | $M_2$ |
| 0 | 1 | 1 | $\bar{A} \cdot B \cdot C$ | $m_3$ | $A + \bar{B} + \bar{C}$ | $M_3$ |
| 1 | 0 | 0 | $A \cdot \bar{B} \cdot \bar{C}$ | $m_4$ | $\bar{A} + B + C$ | $M_4$ |
| 1 | 0 | 1 | $A \cdot \bar{B} \cdot C$ | $m_5$ | $\bar{A} + B + \bar{C}$ | $M_5$ |
| 1 | 1 | 0 | $A \cdot B \cdot \bar{C}$ | $m_6$ | $\bar{A} + \bar{B} + C$ | $M_6$ |
| 1 | 1 | 1 | $A \cdot B \cdot C$ | $m_7$ | $\bar{A} + \bar{B} + \bar{C}$ | $M_7$ |

# Example

Express the following Boolean function in the canonical SOP Form

$$f(A, B, C) = A + \overline{B}C$$

**Solution:** In each product term, insert the missing variables using identity $(X \cdot 1 = X)$ and complement $(X + \overline{X} = 1)$ laws

$$f(A, B, C) = A + \overline{B}C$$

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

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

$$= ABC + AB\overline{C} + A\overline{B}C + A\overline{B}\overline{C} + A\overline{B}C + \overline{A}\overline{B}C$$

$$= ABC + AB\overline{C} + A\overline{B}C + A\overline{B}\overline{C} + \overline{A}\overline{B}C$$

$$= m_7 + m_6 + m_5 + m_4 + m_1 = \Sigma m(1,4,5,6,7)$$

# Example

Express the following Boolean function in the canonical POS Form

$$f(A, B, C) = (A + B)(B + C)(C + A)$$

**Solution:** In each product term, insert the missing variables using identity $(X + 0 = X)$ and complement $(X \cdot \overline{X} = 0)$ laws

$$f(A, B, C) = (A + B)(B + C)(C + A)$$

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

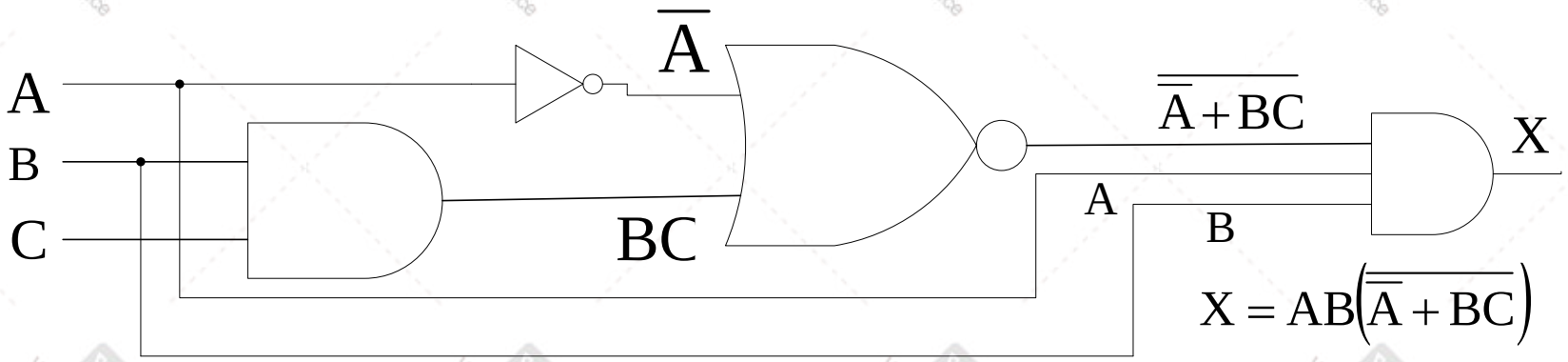
$$= (A + B + C)(A + B + \overline{C})(A + B + C) \ldots$$

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

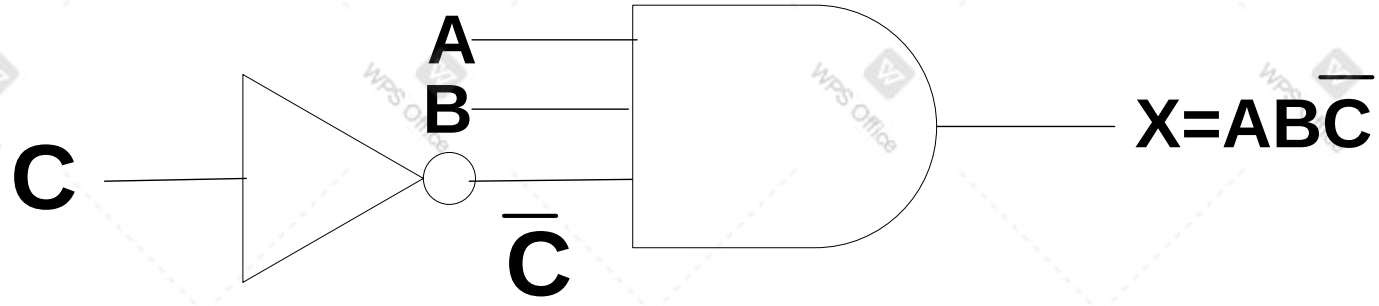
$$= (A + B + C)(A + B + \overline{C})(A + \overline{B} + C)(\overline{A} + B + C)$$

$$= M_0 \cdot M_1 \cdot M_2 \cdot M_4 = \Pi M(0,1,2,4)$$

# Simplifying Logic Circuits




$$X = AB\left(\overline{\overline{A}+BC}\right) = AB\left(\overline{\overline{A}}\cdot\overline{BC}\right) = AB[A\cdot(\overline{B}+\overline{C})] = AB(\overline{B}+\overline{C}) = AB\overline{C}$$



# Logic Simplification

## Karnaugh Map

## Quine-McCluskey Method

# Karnaugh Map (K-Map)

- Karnaugh map is a powerful, widely used, **graphical method of logic simplification** in order to derive the simplest SOP or POS form
- Karnaugh maps allow expressions with up to four variables to be handled in a straightforward manner, and can be used with expressions of **up to six Boolean variables**
- **Each cell** in the Karnaugh map corresponds to **one line** in the **truth table**

# K-Map based Minimization Procedure

Steps to simplify a logic function using K-map are:

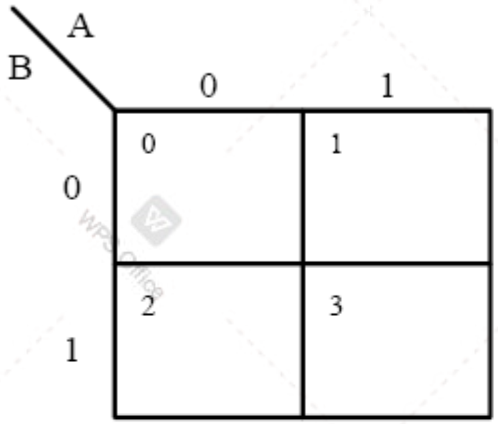
1. Draw an empty K-map
2. Fill in 1’s and 0’s into the corresponding cells as per the given logic function or it’s truth table
3. Identify and draw loops on the K-map to group together adjacent 1’s (for SOP form) or adjacent 0’s (for POS form)
4. Create the expression for each group and combine to give the simplified logic expression

# Creating an empty Karnaugh Map

- The **number of inputs** $(N)$ **in logic function** defines the **number of cells** $(2^N)$ in the K-map to be drawn.
- For two variables, A and B, there are four possible input states. Hence, the K-map will have 4 cells, **each cell corresponding to one particular input state.**
- Often the top left corner of **each cell** show the **decimal equivalent** of the binary input code. Notice that the cells are ordered in Gray code such that **only 1-bit changes** when we go to an **adjacent cell.**
- Each cell corresponds to one particular **minterm** $m_i$ or **maxterm** $M_i$, where $i$ is the **decimal equivalent** denoting **the cell.**

# Multivariable Karnaugh Maps

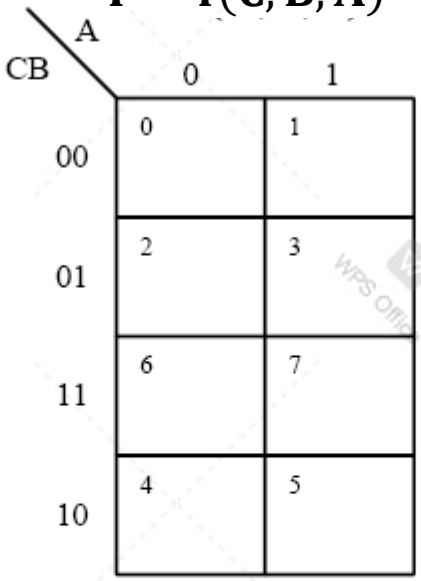
## Two-variable K-map
$F = f(B, A)$

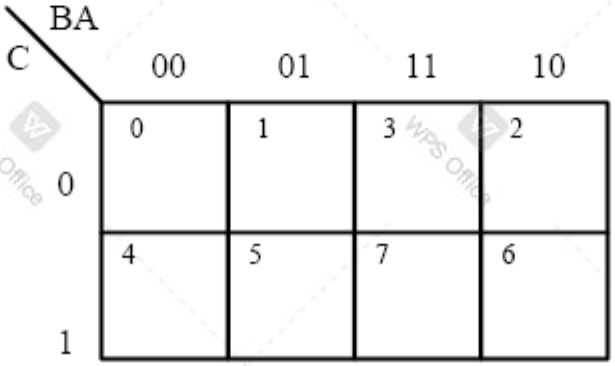



### Truth table for two variables

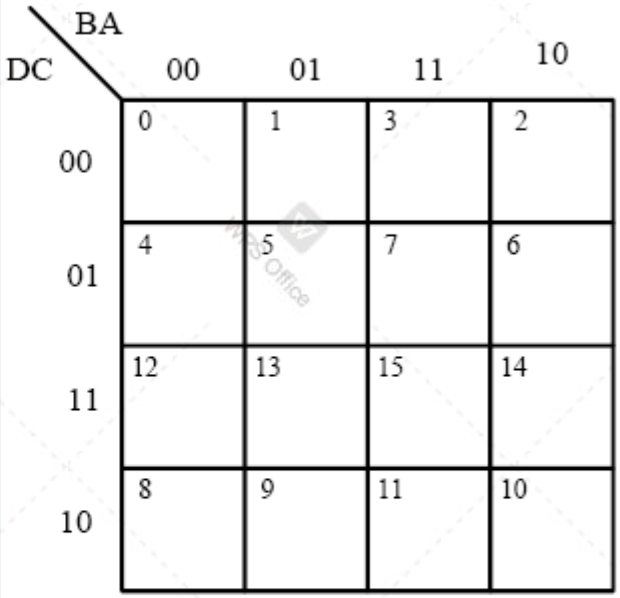
| B | A | Output F | Binary code/ Decimal eq. |
|---|---|---|---|
| 0 | 0 | | $00_2 = 0_{10}$ |
| 0 | 1 | | $01_2 = 1_{10}$ |
| 1 | 0 | | $10_2 = 2_{10}$ |
| 1 | 1 | | $11_2 = 3_{10}$ |

## Three-variable K-map
$F = f(C, B, A)$




## Four-variable K-map
$F = f(D, C, B, A)$



# Filling in the Karnaugh Map

- **Each cell** of the Karnaugh map is filled with **either a 1 or a 0** signifying the **output** for the input corresponding to that cell.
- The **original binary function** which needs to be simplified may be **provided** in any one of the following forms:
  - by specifying the binary/decimal numbers that correspond to an output of 1 (i.e. **sum of products**) or output of 0 (i.e. **product of sums**);
  - a **general logical/Boolean expression**;
  - a **truth table**.

# K-Map Filling Example

**Map the following canonical SOP expression on a K-map:**

$$\boldsymbol{F(A,B,C,D) = ABCD + AB\overline{C}D + AB\overline{C}\overline{D} + A\overline{B}C\overline{D} + \overline{A}B\overline{C}\overline{D} + \overline{A}\overline{B}CD + \overline{A}\overline{B}\overline{C}D}$$

Given expression is a sum of minterms:

$$F = \Sigma m(15, 13, 12, 10, 8, 3, 1)$$

Fill in 1's into the cells corresponding to the above inputs
(rest of the cells have output 0 so they are left empty)

| AB \ CD | 00 | 01 | 11 | 10 |
|---|---|---|---|---|
| 00 | | 1 | 1 | |
| 01 | 1 | | | |
| 11 | 1 | 1 | 1 | |
| 10 | | | | 1 |

# K-Map Filling Example

## Map the following SOP expression on the K-map:

$$\boldsymbol{F(A,B,C) = \overline{A} + A\overline{B} + AB\overline{C}}$$

The SOP expression is not in canonical form because each product term does not have three variables. The first term is missing two variables, the second term is missing one variable, and the third term is standard.

First term is expanded as

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

Second term is expanded as

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

Thus, the canonical SOP form of the given expression:

$$F = \bar{A}BC + \bar{A}B\bar{C} + \bar{A}\bar{B}C + \bar{A}\bar{B}\bar{C} + A\bar{B}C + A\bar{B}\bar{C} + AB\bar{C}$$

$$F = \Sigma m(0, 1, 2, 3, 4, 5, 6)$$

| AB \ C | 0 | 1 |
|---|---|---|
| 00 | 1 | 1 |
| 01 | 1 | 1 |
| 11 | 1 | |
| 10 | 1 | 1 |

# K-Map Filling Example

**Map the following canonical POS expression on a K-map:**

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

Given expression is a sum of maxterms:

$$F = \Pi M(15, 11, 12, 3, 2)$$

Fill in 0's into the cells corresponding to the above inputs
(rest of the cells have output 1 so are left empty)

| AB \ CD | 00 | 01 | 11 | 10 |
|---|---|---|---|---|
| 00 | | | 0 | 0 |
| 01 | | | | |
| 11 | 0 | | 0 | |
| 10 | | | 0 | |

# K-Map Filling Example

## Map the following POS expression on the K-map:

$$\boldsymbol{F(A,B,C) = (AB + \bar{A})(AB + C)}$$

The POS expression is not in canonical form to first doing that conversion

$$F = (AB + \bar{A})(AB + C)$$
$$= (A + \bar{A})(B + \bar{A})(A + C)(B + C)$$
$$= (B + \bar{A})(A + C)(B + C)$$
$$= (\bar{A} + B + C\bar{C})(A + B\bar{B} + C)(A\bar{A} + B + C)$$
$$= (\bar{A} + B + C)(\bar{A} + B + \bar{C})(A + B + C) \ldots$$
$$(A + \bar{B} + \mathrm{C})(A + B + C)(\bar{A} + B + C)$$
$$= (A + B + C)(A + \bar{B} + \mathrm{C})(\bar{A} + B + C)(\bar{A} + B + \bar{C})$$
$$= \Pi M(0, 2, 4, 5)$$

| AB \ C | 0 | 1 |
|---|---|---|
| 00 | 0 | |
| 01 | 0 | |
| 11 | | |
| 10 | 0 | 0 |

# Creating groups in Karnaugh Map

- When all the output values are entered, all of the 1's (or all of the 0's, depending on if SOP or POS form is finally required, respectively) must be **grouped together** according to the following rules:
  - Each **group** must contain $\mathbf{2^n}$ **adjacent cells**, where $n$ is an +ve integer.
  - Each **group** should be made **as large as possible**, while satisfying the constraint of having cells numbering to a power of 2.
  - The **larger the groups** formed, **the simpler the final function.**

# Creating groups in Karnaugh Map (contd.)

- **Each cell** with output 1 for SOP (or each cell with output 0 for POS) must be included in **at least one group**.
- The **groups may overlap**, so one cell may be included in several groups, but any group that has all its elements included in other groups can be ignored.
- There may be several correct minimal forms for a given logic function, **dependent upon the particular groupings** that are chosen.
- Once the groups have been assigned, a logic expression can be created. **For each group**, write down the **common inputs** (AND/product terms if using 1s; OR/sum terms if using 0s) that describe that specific group and combine accordingly (SOP or POS).

# Solved Example

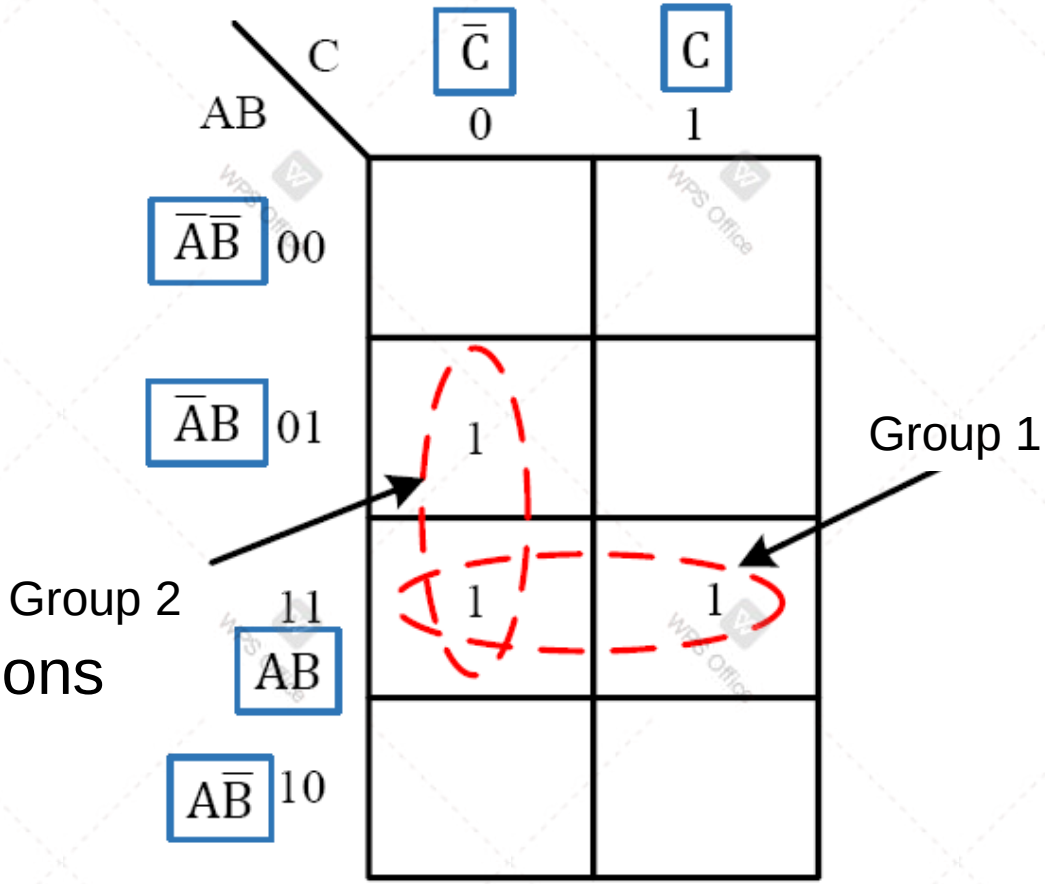
**Simplify $F(A,B,C) = \bar{A}B\bar{C} + AB\bar{C} + ABC$ into simpler SOP form using K-map**

Group 1 $\Rightarrow AB$

Group 2 $\Rightarrow B\bar{C}$

The simplified SOP form is obtained by ORing the group expressions

$$F(A,B,C) = AB + B\bar{C}$$



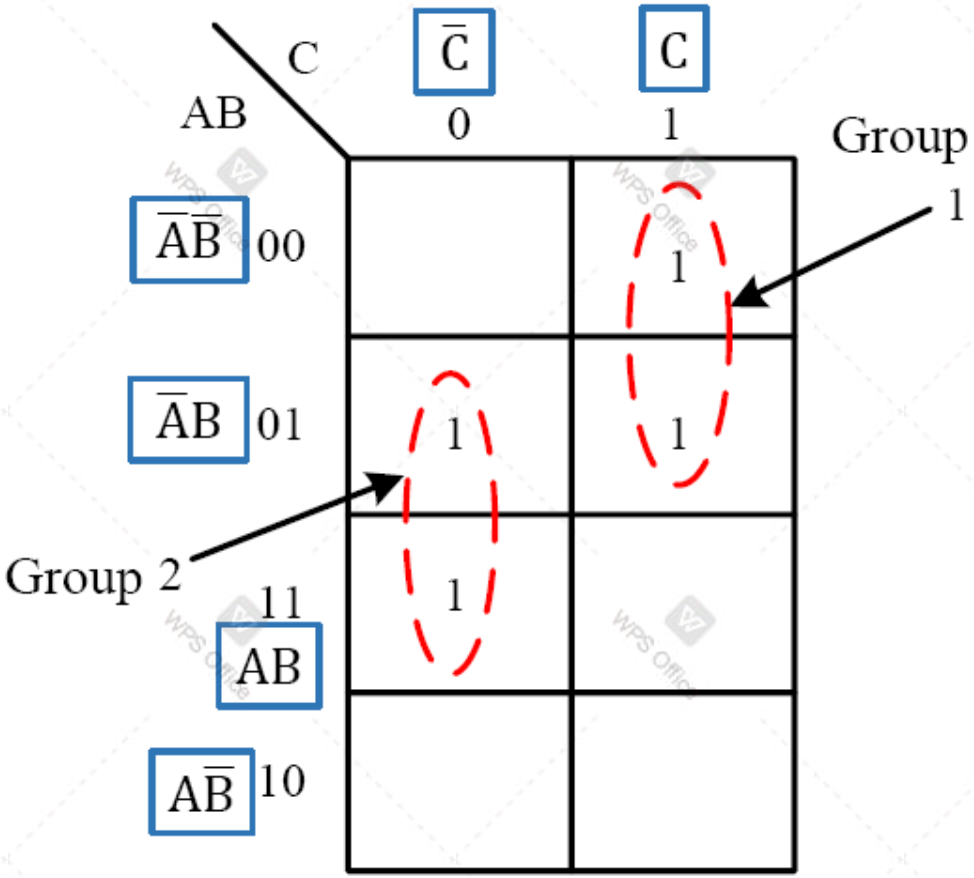
# Solved Example

**Simplify $F(A,B,C) = \Sigma m(1,2,3,6)$ into minimum SOP form using K-map**

Group 1 $\Rightarrow \bar{A}C$

Group 2 $\Rightarrow B\bar{C}$

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



# Solved Example

**Simplify $F(A,B,C,D) = \Sigma m(2,3,4,5,7,8,10,13,15)$ into minimum SOP form using K-map**

Groupings:

$1 \Rightarrow BD$

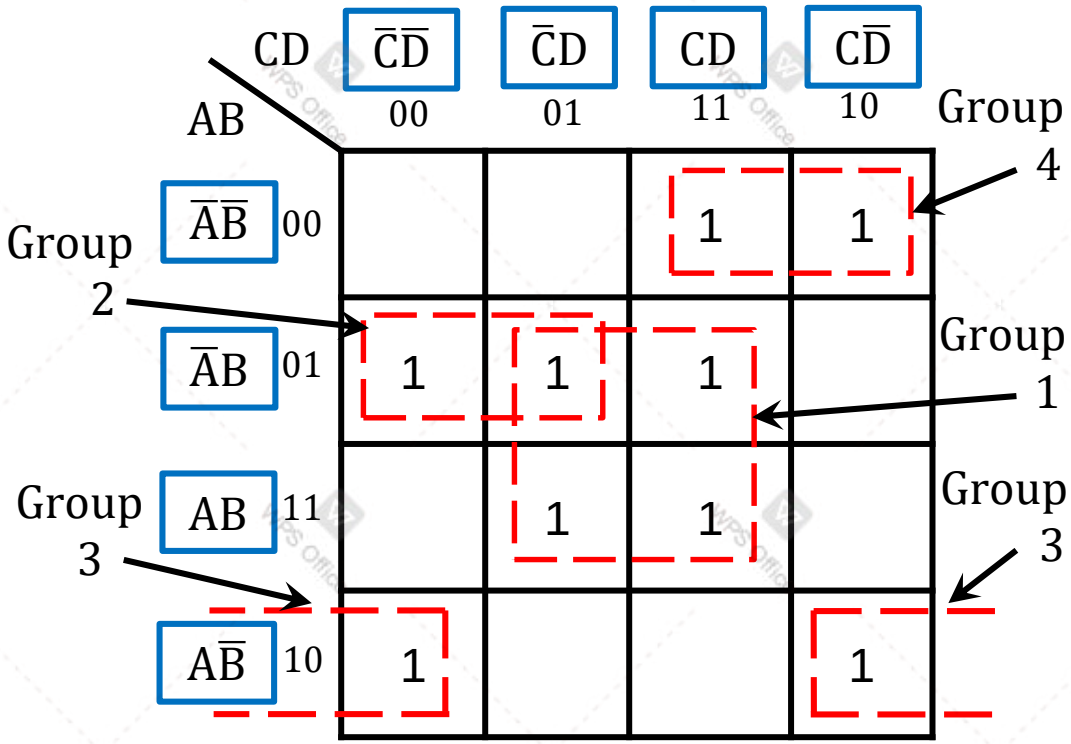
$2 \Rightarrow \bar{A}B\bar{C}$

$3 \Rightarrow A\bar{B}\bar{D}$

$4 \Rightarrow \bar{A}\bar{B}C$

Minimum SOP form:

$$F = BD + \bar{A}B\bar{C} + \cdots A\bar{B}\bar{D} + \bar{A}\bar{B}C$$



# Solved Example

**Simplify $F(A,B,C,D) = \Sigma m(0,5,7,8,10,12,14,15)$ into minimum SOP form using K-map**

Groupings:

1 ⇒ $\bar{A}BD$

2 ⇒ $ABC$

3 ⇒ $AC\bar{D}$

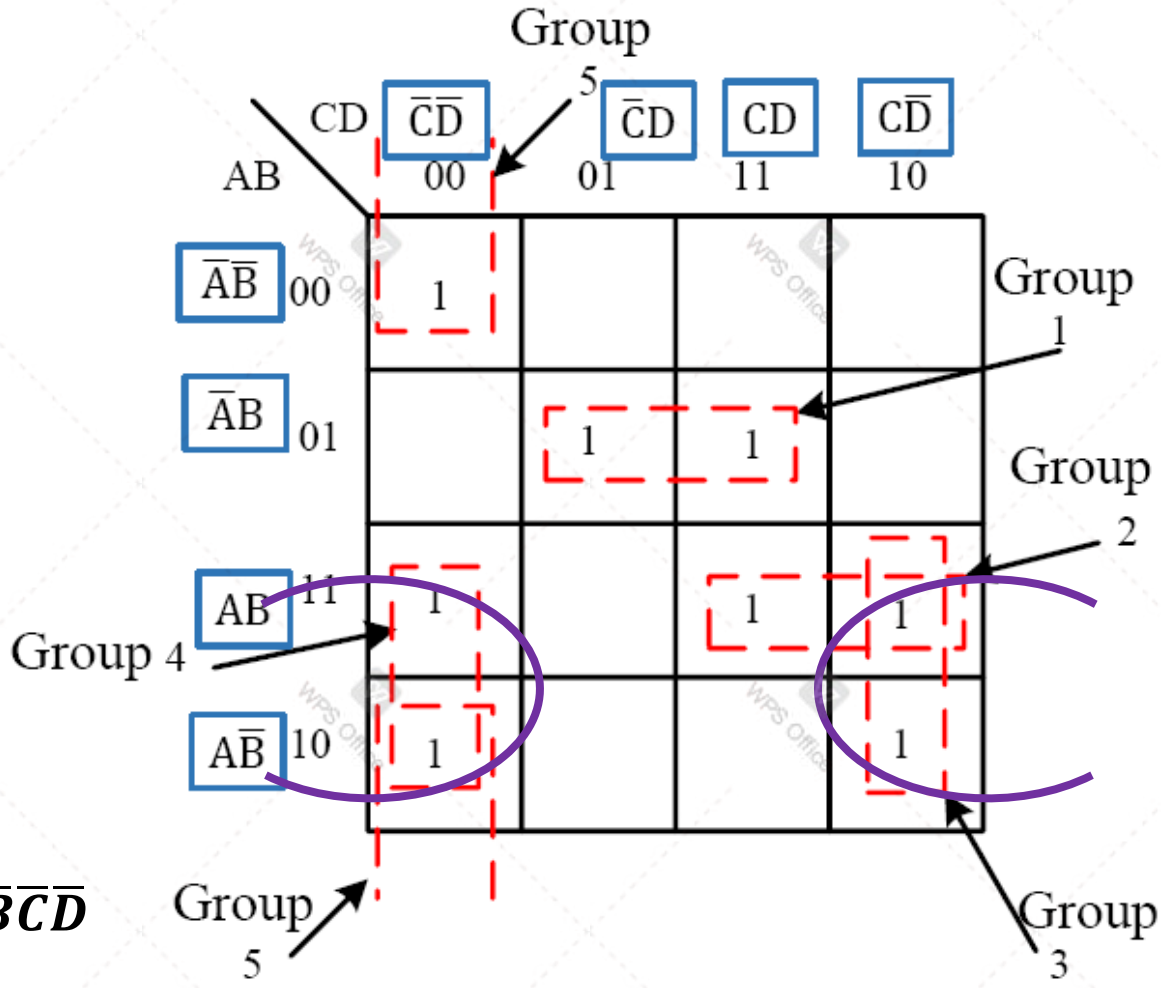
4 ⇒ $A\bar{C}\bar{D}$

5 ⇒ $\bar{B}\bar{C}\bar{D}$

$F = \bar{A}BD + ABC + AC\bar{D} + A\bar{C}\bar{D} + \bar{B}\bar{C}\bar{D}$

Optimally minimized form:

$\boldsymbol{F = A\bar{D} + \bar{A}BD + ABC + \bar{B}\bar{C}\bar{D}}$



# Solved Example

**Use a K-map find the minimum SOP form of**

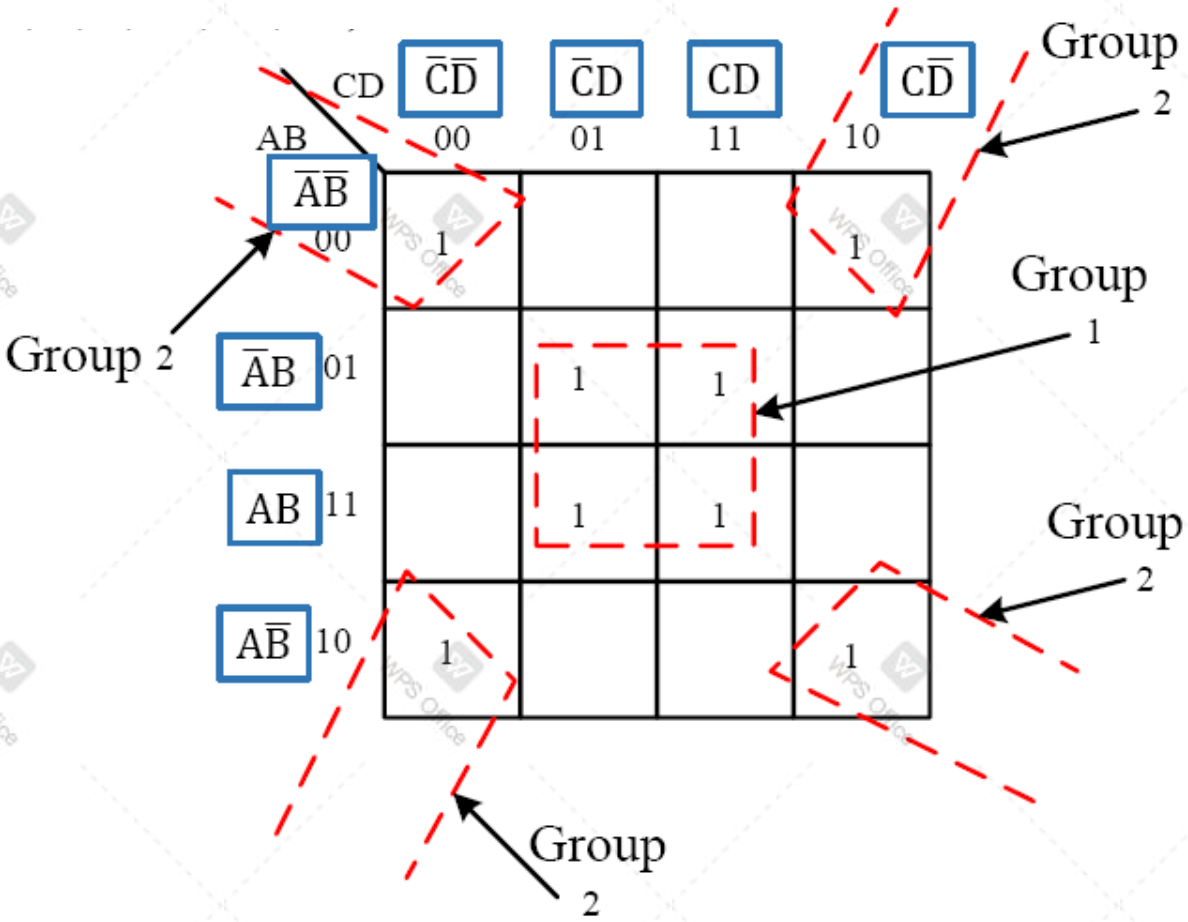
$$F(A,B,C,D) = \Sigma m(0,2,5,7,8,10,13,15)$$

Groupings:

$1 \Rightarrow BD$

$2 \Rightarrow \bar{B}\bar{D}$

$$F = BD + \bar{B}\bar{D}$$



# Solved Example

**Use a K-map to minimize the following standard POS expression:**

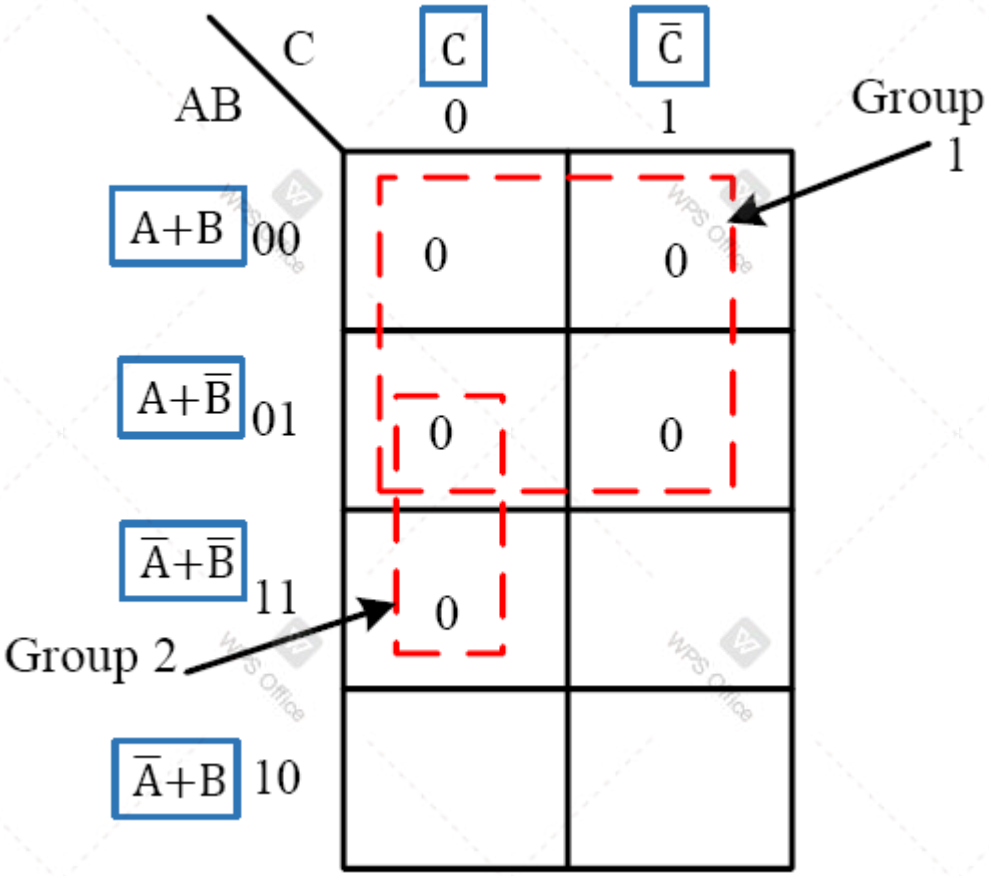
$$F = (A + B + C)(A + B + \overline{C})(A + \overline{B} + C)(A + \overline{B} + \overline{C})(\overline{A} + \overline{B} + C)$$

Groupings:

$1 \Rightarrow A$

$2 \Rightarrow (\overline{B} + C)$

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



# Solved Example

**Use K-map to convert the following standard POS expression into: (a) a minimum POS expression, (b) a standard SOP expression, and c) a minimum SOP expression.**

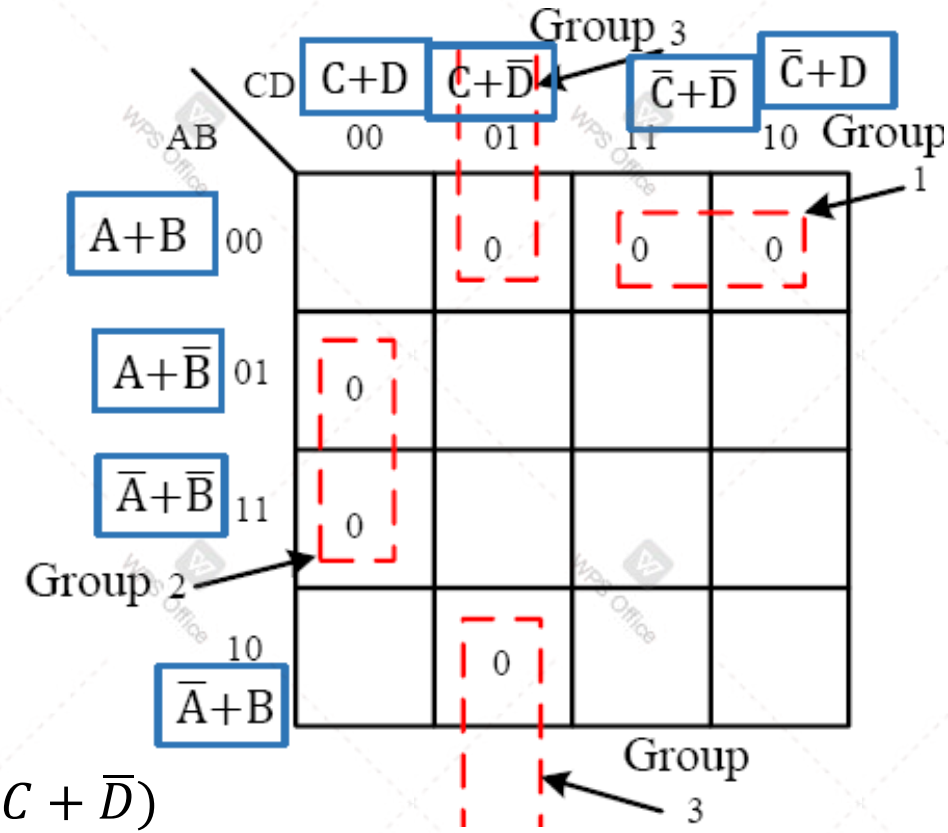
$$F(A,B,C,D) = \Pi M(1,2,3,4,9,12)$$

a) Groupings:

$1 \Rightarrow (A + B + \bar{C})$

$2 \Rightarrow (\bar{B} + C + D)$

$3 \Rightarrow (B + C + \bar{D})$




Minimum POS form:

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

b) The remaining cells are filled with 1s.

Now, sum all minterns corresponding to these cells with output 1 to get the standard SOP expression as:

$$F = \Sigma m(0, 5, 6, 7, 8, 10, 11, 13, 14, 15)$$

| AB \ CD | $\bar{C}\bar{D}$ 00 | $\bar{C}D$ 01 | $CD$ 11 | $C\bar{D}$ 10 |
|---|---|---|---|---|
| $\bar{A}\bar{B}$ 00 | 1 | 0 | 0 | 0 |
| $\bar{A}B$ 01 | 0 | 1 | 1 | 1 |
| $AB$ 11 | 0 | 1 | 1 | 1 |
| $A\bar{B}$ 10 | 1 | 0 | 1 | 1 |

$$F = \bar{A}\bar{B}\bar{C}\bar{D} + \bar{A}B\bar{C}\bar{D} + \bar{A}BCD + \bar{A}BC\bar{D} + AB\bar{C}D + ABCD + ABC\bar{D} + A\bar{B}CD + A\bar{B}\bar{C}\bar{D} + A\bar{B}C\bar{D}$$

c)

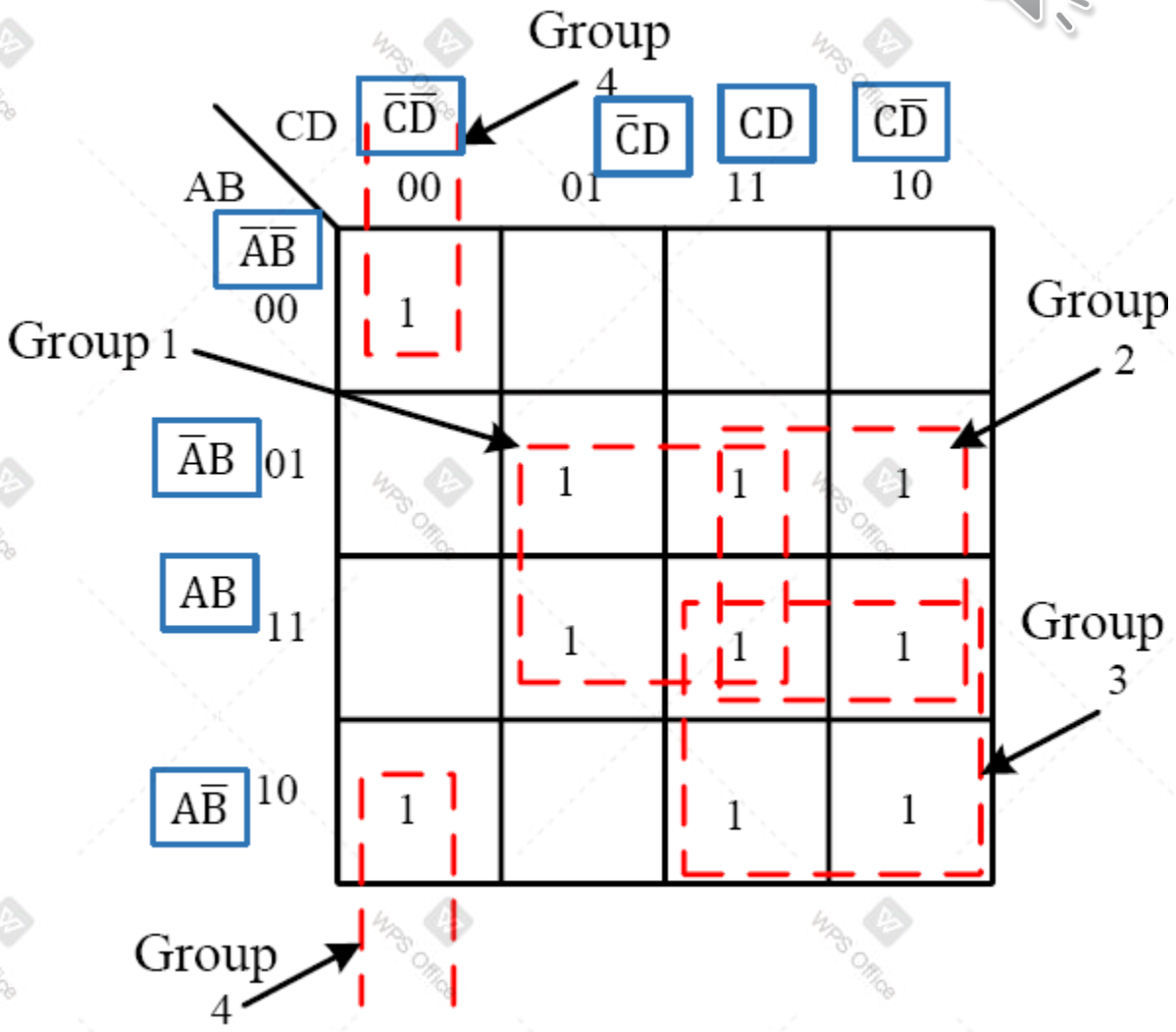
Groupings:

$1 \Rightarrow BD$

$2 \Rightarrow BC$

$3 \Rightarrow AC$

$4 \Rightarrow \bar{B}\bar{C}\bar{D}$




Minimum SOP:

$F = BD + BC + AC + \bar{B}\bar{C}\bar{D}$

# Prime Implicants, Essential and Non-essential Prime Implicants

- An **implicant** is a minterm in SOP form or a maxterm in POS form of a Boolean function
- Each possible group on the K-map representation of a Boolean function is referred to as **prime implicant (PI)**
- A PI which cover at least one minterm that can’t be covered by any other PI is called an **essential prime implicant (EPI)**
- **Non-essential prime implicant** is a PI which does not cover any 1 which cannot covered by some other PI
- EPIs are always included in the minimized expression along with some non-EPIs, if required

# Example

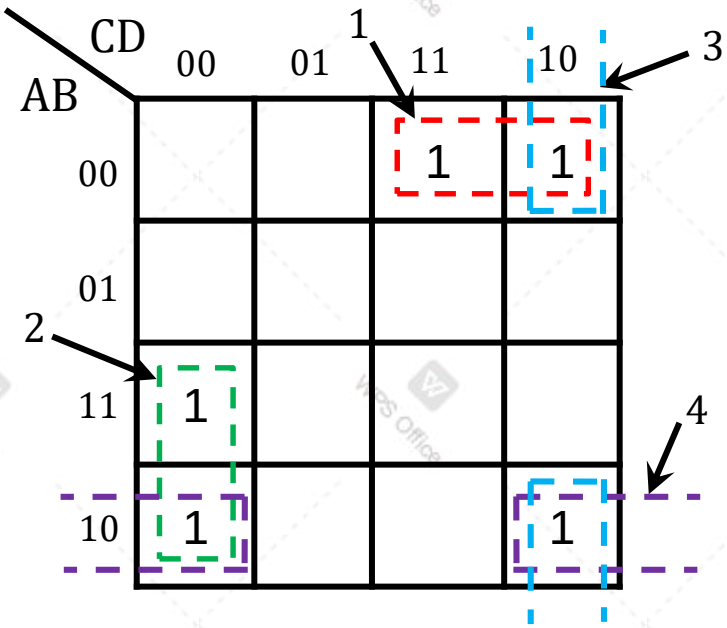
- Minimize Boolean function

  $F(A, B, C, D) = \Sigma m(2, 3, 8, 10, 12)$

- From the K-Map representation, four possible PIs are:

  $p_1 = (m_2 + m_3)$, $p_2 = (m_8 + m_{12})$,

  $p_3 = (m_8 + m_{10})$, $p_4 = (m_3 + m_{10})$



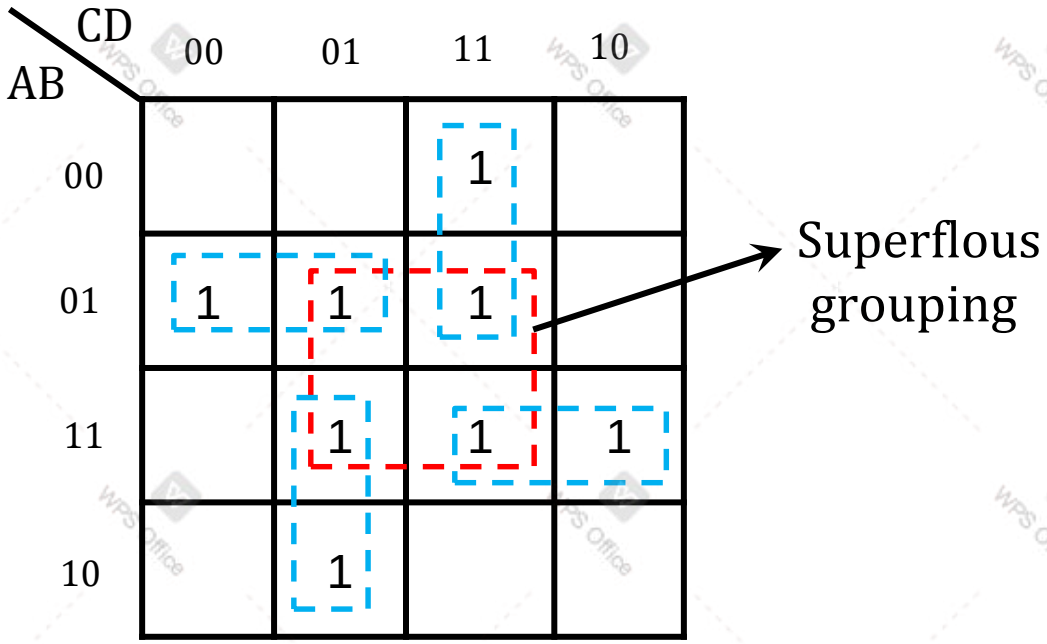

- Two equivalently minimized Boolean expressions that can be formed are:

  $F = p_1 + p_2 + p_3$ or $F = p_1 + p_2 + p_4$

- Thus, $p_1$ and $p_2$ are the EPIs of the given function and they will be present in all minimized solutions

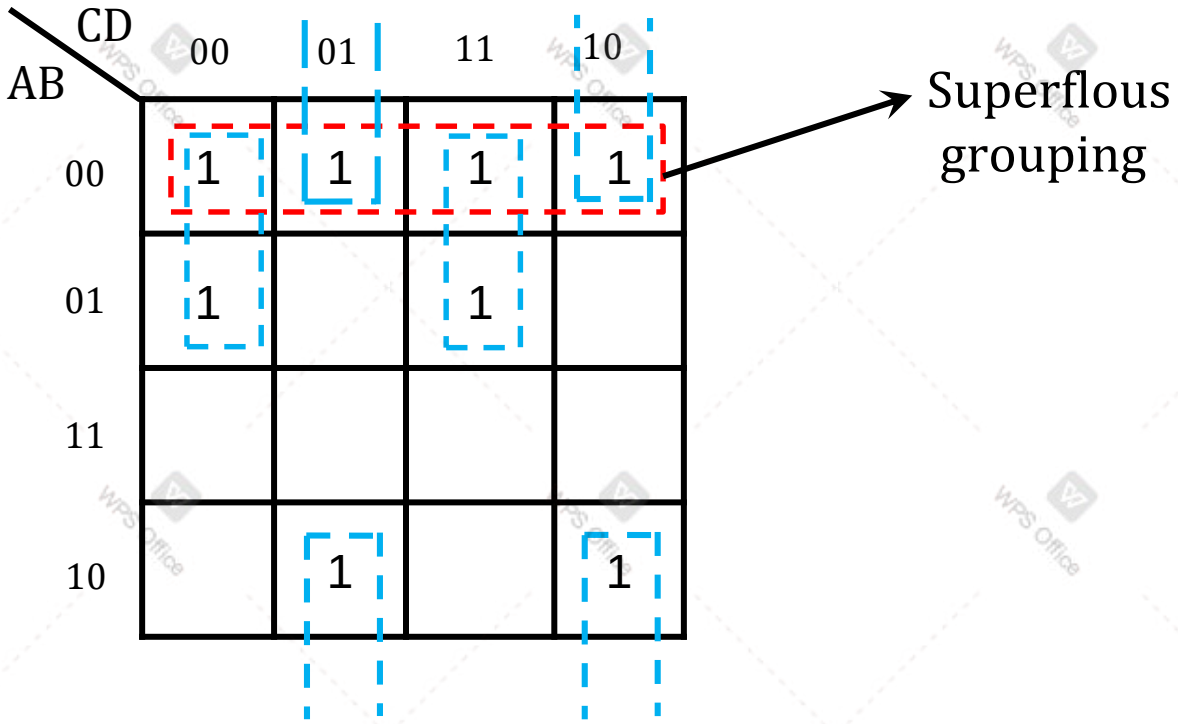
# Hazard with Preoccupation for Finding the Largest Prime Implicants

**Ex:** Minimize $F(A, B, C, D) = \Sigma m(3, 4, 5, 7, 9, 13, 14, 15)$



# Hazard with Preoccupation for Finding the Largest Prime Implicants

**Ex:** Minimize $F(A,B,C,D) = \Sigma m(0,1,2,3,4,7,9,10)$



# Rules of  Making Combination in K-Map to avoid such hazards

1. Encircle and accept as essential prime implicants any cell or cells that cannot be combined with any other

2. Identify the cells that can be combined with a single other cell in only one way

   Encircle such two-cell combinations. A cell which can be combined into a two-grouping but can be so done in more than one way is to be temporarily bypassed

# Rules of  Making Combination in K-Map to avoid such hazards

3. Identify the cells that can be combined with three other cells in only one way

   If not all of the four cells so involved are already covered in groupings of two, encircle these four cells. Again, a cell which can be grouped in groups of four in more than one way is to be temporarily bypassed

4. Repeat the procedure for groups of eight etc.

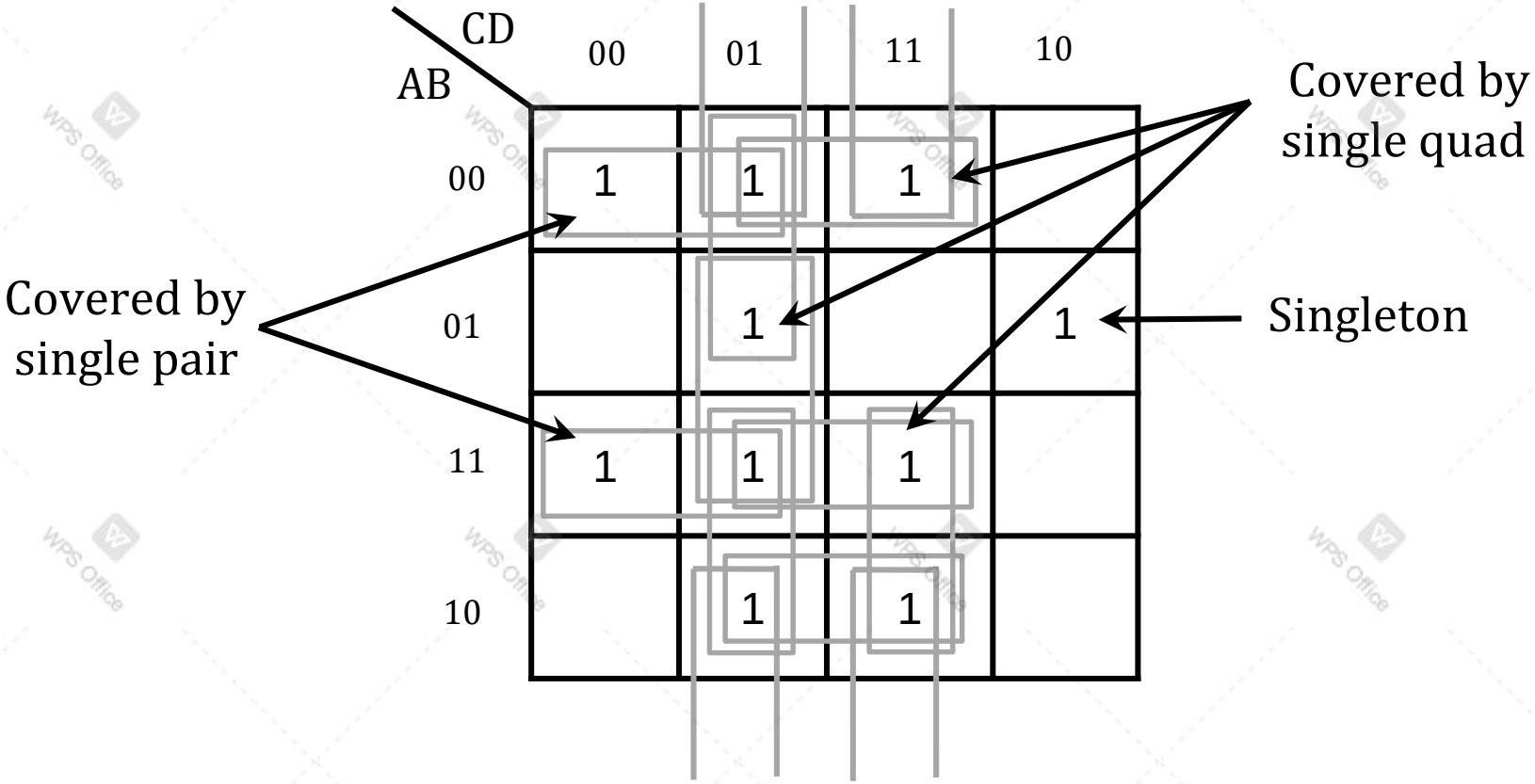
# Rules of  Making Combination in K-Map to avoid such hazards

5. After the above procedure, if there still remains some uncovered cells, they may be combined with each other or with already covered cells in any manner.

Of course, we would want to include these left-over cells in as few groupings as possible.

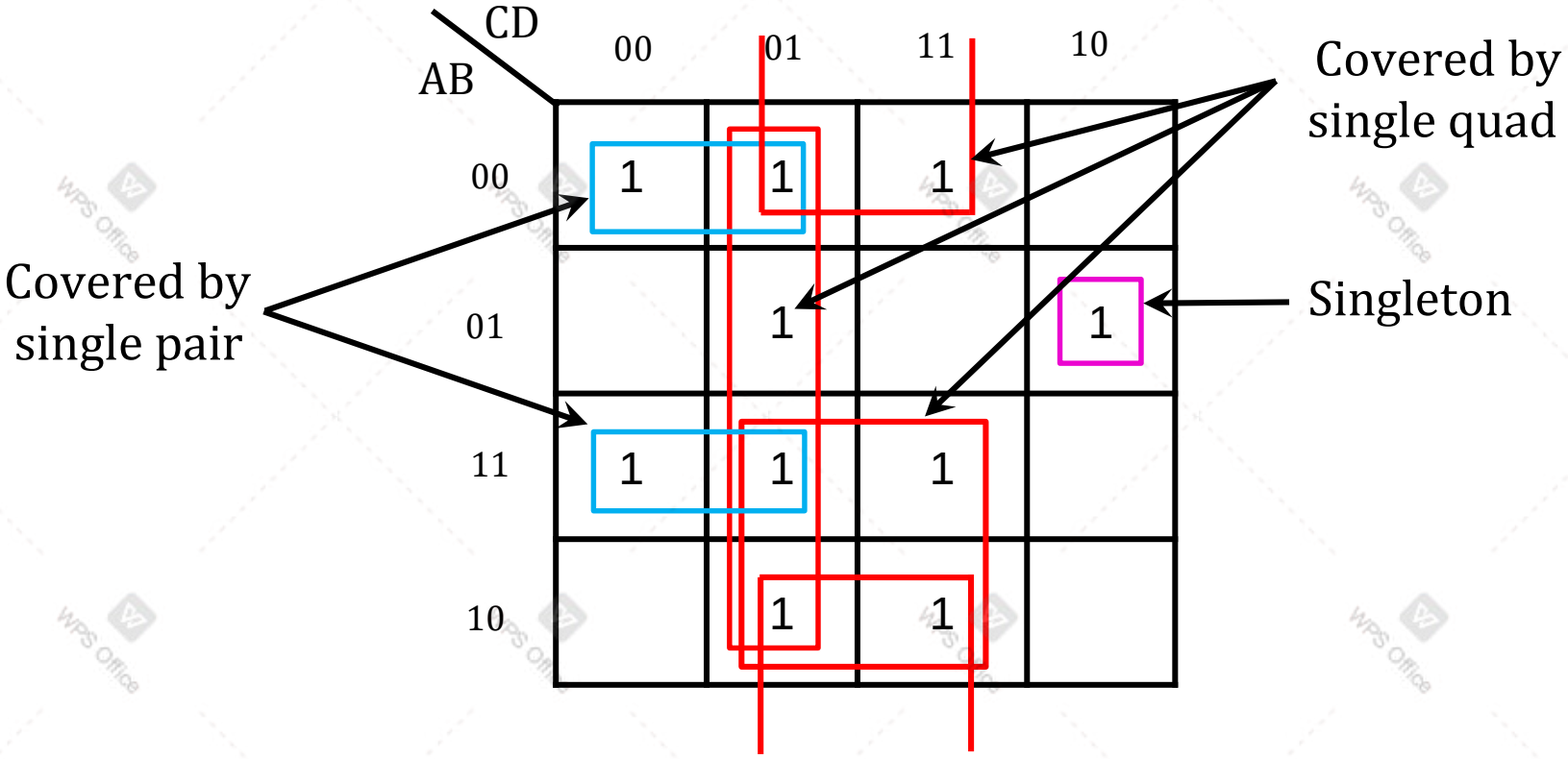
# Example-1

**Ex:** Minimize $F(A, B, C, D) = \Sigma m(0, 1, 3, 5, 6, 9, 11, 12, 13, 15)$



# Example-1 (contd.)

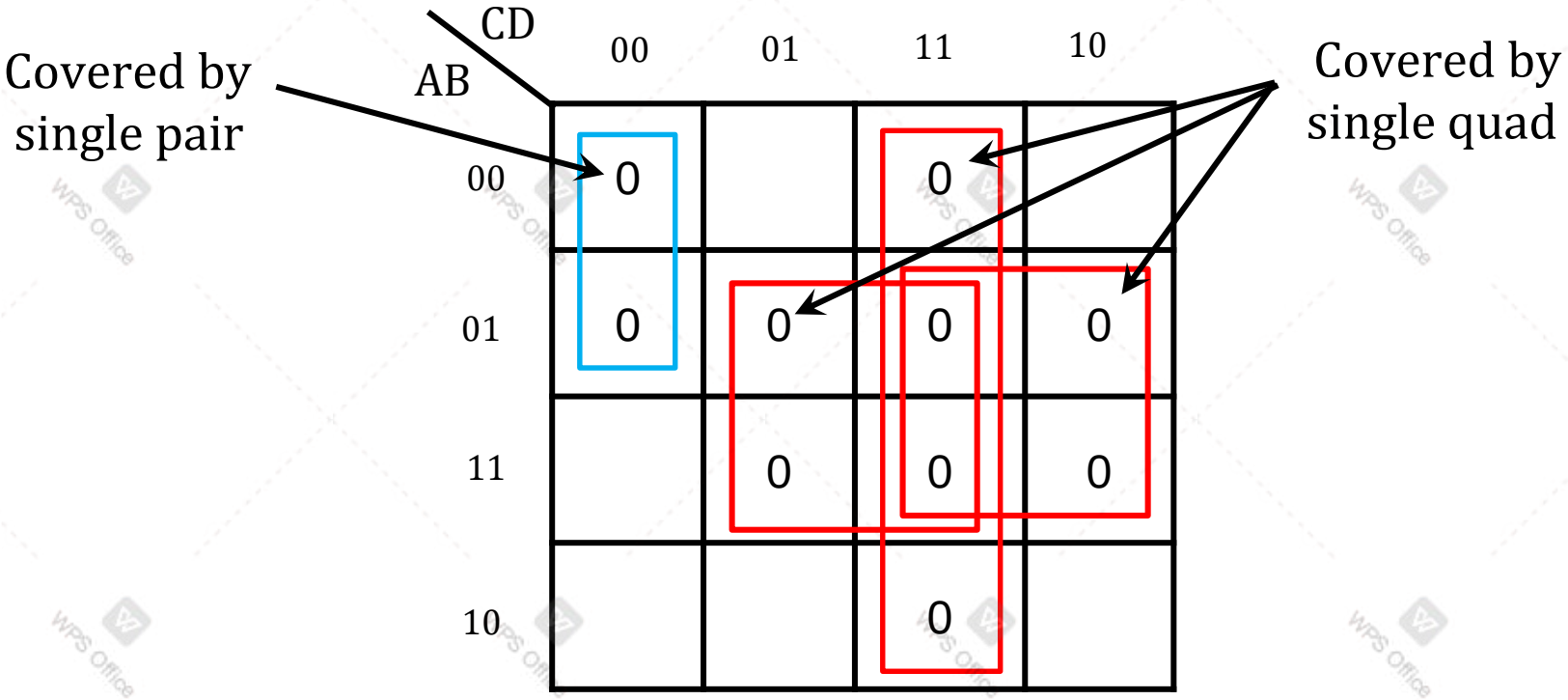
**Example:** $F(A,B,C,D) = \Sigma m(0,1,3,5,6,9,11,12,13,15)$




$$F = \bar{A}BC\bar{D} + \bar{A}\bar{B}\bar{C} + AB\bar{C} + AD + \bar{B}D + \bar{C}D$$

# Example-2

**Ex:** Minimize $F(A,B,C,D) = \Pi M(0,3,4,5,6,7,11,13,14,15)$




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

# Incompletely Specified Functions

## (Functions with Don’t Care Conditions)

# K-Map with “Don’t Care” Conditions

- Sometimes a situation arises in which some input variable combinations are not allowed.

  **E.g.** For BCD input, there are six invalid input combinations: 1010, 1011, 1100, 1101, 1110, and 1111.

- Since these unallowed states will never occur in for that particular application, they can be treated as **"don't care" terms** with respect to their effect on the output.

- For these "don't care" terms, **either 1 or 0 may be assigned to the output**; it really does not matter since they will never occur.

# K-Map with “Don’t Care” Conditions

- The **"don't care" terms** can be used to our advantage on the Karnaugh map to **further simplify the Boolean expression** for a particular application.

- For **each "don't care" term**, an **X** is placed in the cell. When grouping the 1s, some of the **Xs can be treated as 1s** to make a larger grouping, while others **can be treated as 0s** if they cannot be used (and vice-versa if we are grouping 0s).

- The larger a group, the simpler the resulting term will be.

# Example

Design a logic circuit which produces high (1) output when the BCD input is greater than 6.

- For 6 invalid combinations of BCD input, the logic circuit output is unspecified so is denoted by X (either 0 or 1)
- Draw the truth table and note for two forms of the output logic expression which is required to be minimized:

SOP: $Y = \Sigma m(7, \ldots, 9) + \Sigma d(10, \ldots, 15)$

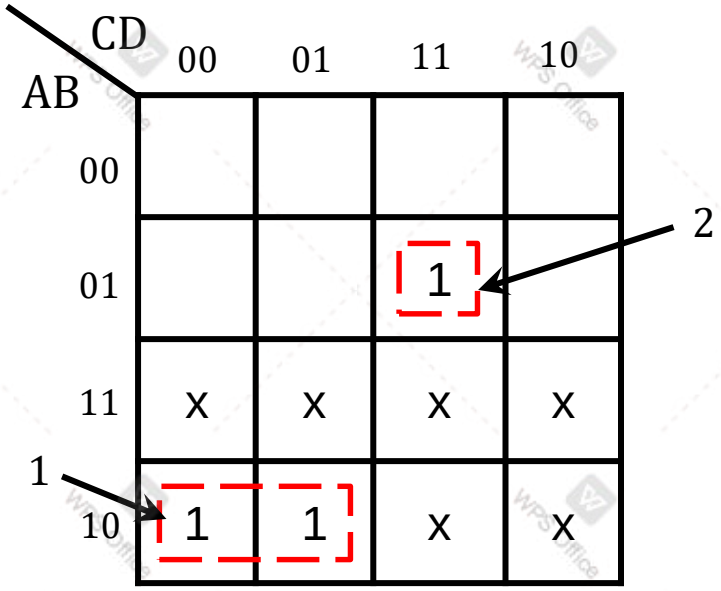
POS: $Y = \Pi M(0, \ldots, 6).\Pi d(10, \ldots, 15)$

Truth Table

| Input ABCD | Output Y |
|---|---|
| 0000 | 0 |
| 0001 | 0 |
| 0010 | 0 |
| 0011 | 0 |
| 0100 | 0 |
| 0101 | 0 |
| 0110 | 0 |
| 0111 | 1 |
| 1000 | 1 |
| 1001 | 1 |
| 1010 | X |
| 1011 | X |
| 1100 | X |
| 1101 | X |
| 1110 | X |
| 1111 | X |

# “Don’t Care” Example: SOP Form

| Input ABCD | Output Y |
|---|---|
| 0000 | 0 |
| 0001 | 0 |
| 0010 | 0 |
| 0011 | 0 |
| 0100 | 0 |
| 0101 | 0 |
| 0110 | 0 |
| 0111 | 1 |
| 1000 | 1 |
| 1001 | 1 |
| 1010 | X |
| 1011 | X |
| 1100 | X |
| 1101 | X |
| 1110 | X |
| 1111 | X |

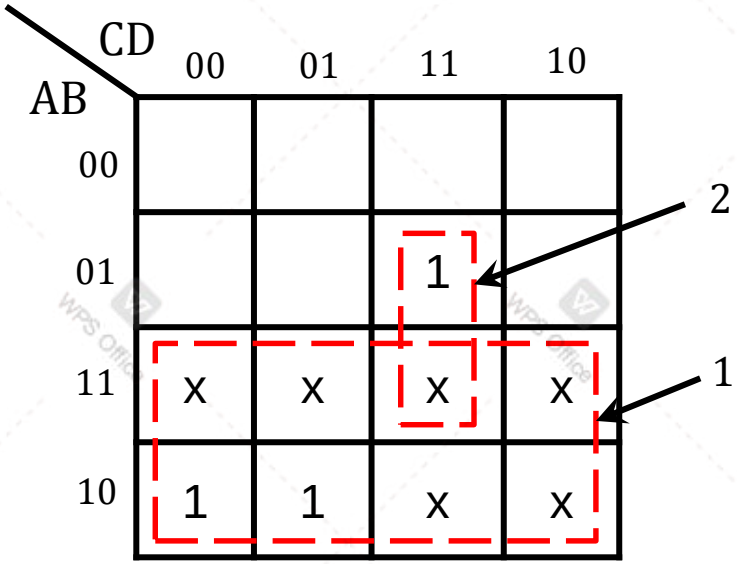



Without “don’t cares”

$1 \Rightarrow A\bar{B}\bar{C}$

$2 \Rightarrow \bar{A}BCD$

$$Y = A\bar{B}\bar{C} + \bar{A}BCD$$




With “don’t cares”

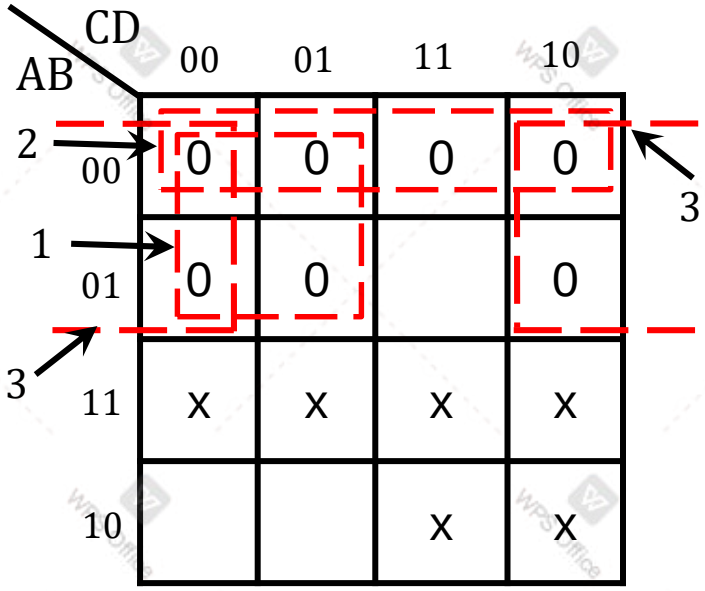
$1 \Rightarrow A$

$2 \Rightarrow BCD$

$$Y = A + BCD$$

# “Don’t Care” Example: POS Form

| Input ABCD | Output Y |
|---|---|
| 0000 | 0 |
| 0001 | 0 |
| 0010 | 0 |
| 0011 | 0 |
| 0100 | 0 |
| 0101 | 0 |
| 0110 | 0 |
| 0111 | 1 |
| 1000 | 1 |
| 1001 | 1 |
| 1010 | X |
| 1011 | X |
| 1100 | X |
| 1101 | X |
| 1110 | X |
| 1111 | X |



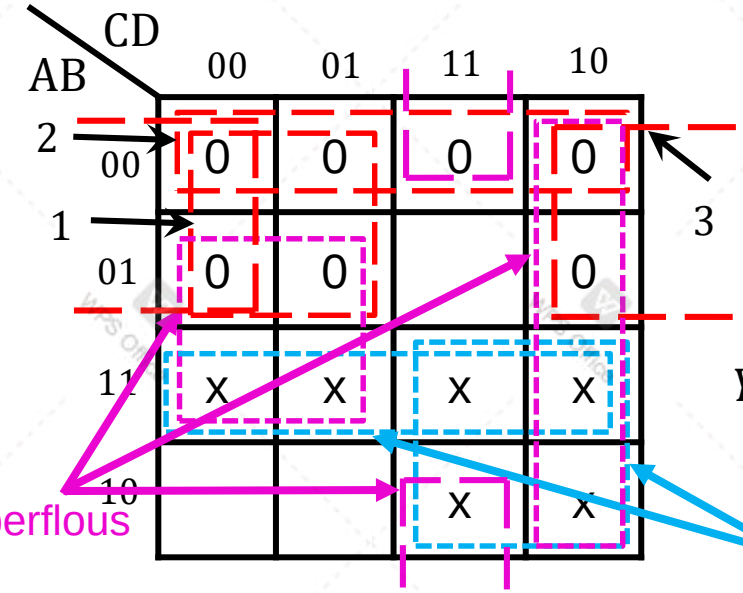

Without “don’t cares”

$1 \Rightarrow A + C$

$2 \Rightarrow A + B$

$3 \Rightarrow A + \bar{C} + D$

$Y = (A + C)(A + B)(A + D)$




With “don’t cares”

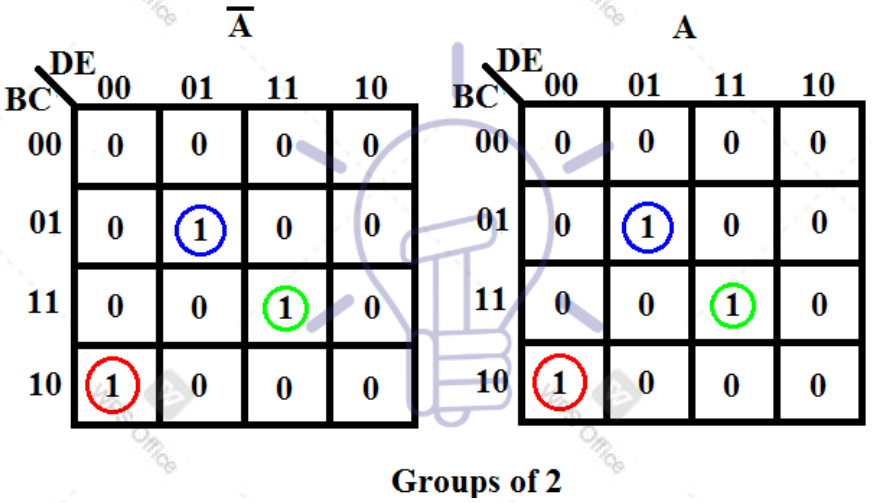
$Y = (A + C)(A + B)(A + D)$

# Quine-McCluskey Method

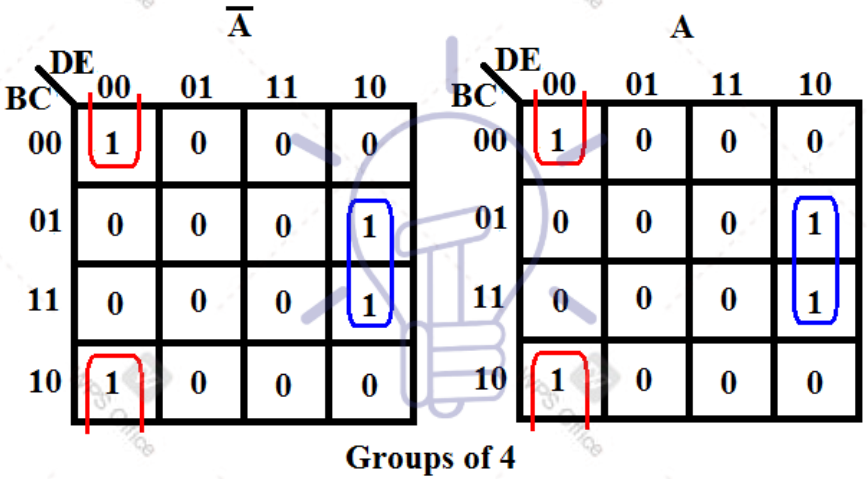
# or

# Tabular Method of Boolean Function Minimization

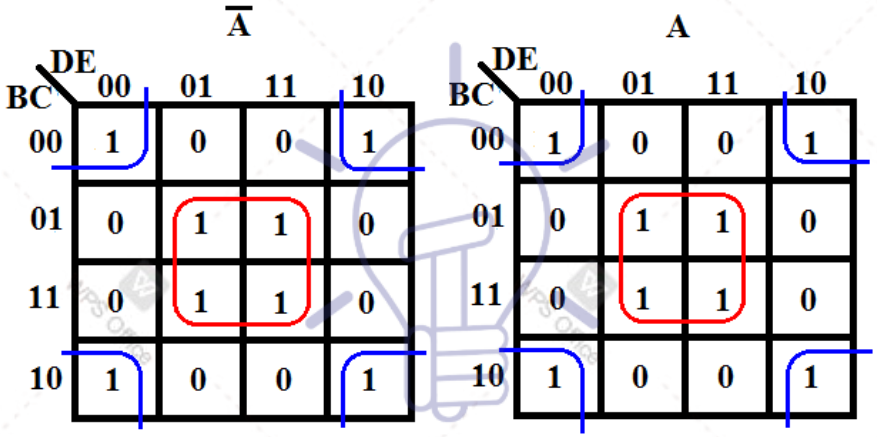
# 5 Variable K-Map & Sample Groupings



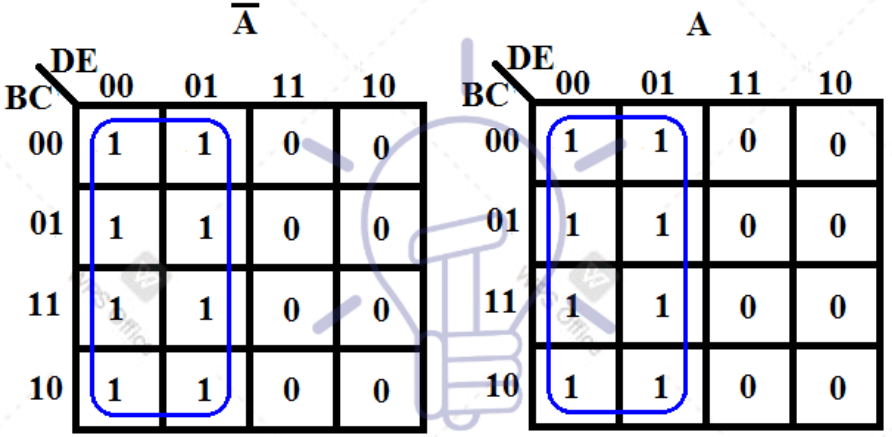

Groups of 2




Groups of 4



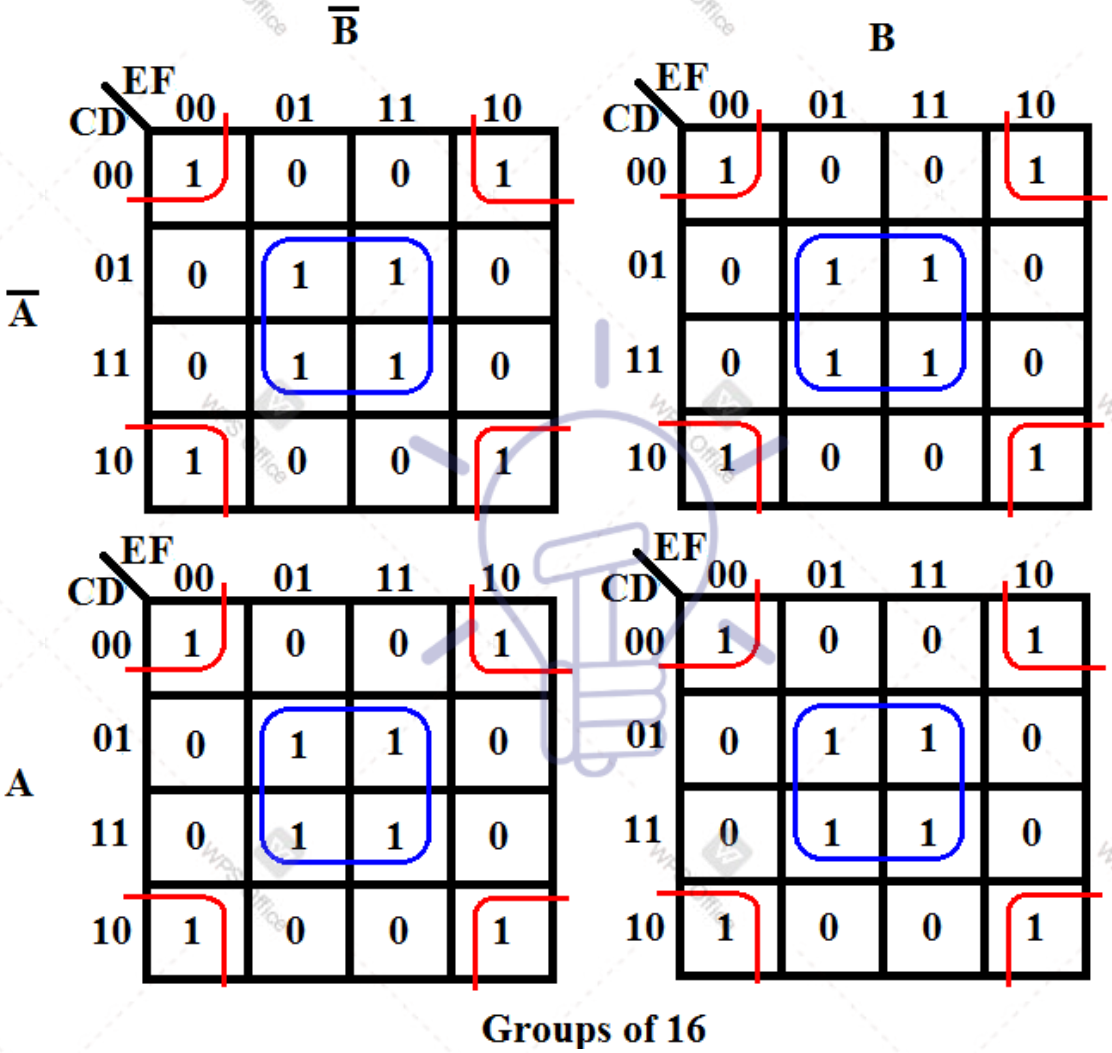

Groups of 8




Group of 16

Courtesy: https://www.electricaltechnology.org/2018/05/karnaugh-map-k-map.html

# 6 Variable K-Map and Sample Grouping




Courtesy: https://www.electricaltechnology.org/2018/05/karnaugh-map-k-map.html

# Quine-McCluskey Method

- Tabular approach to logic function minimization
  - Compute all prime implicants by combining the minterms in a bottom up manner
  - Identify the essential prime implicants
  - Find a minimum expression for Boolean functions
- No visualization of prime implicants
- Can be programmed and implemented in a computer

# QM Method Example 1

Minimize the given Boolean function using QM method

$$F(A,B,C,D) = \Sigma m(0,3,5,6,7,10,12,13) + \Sigma d(2,9,15)$$

- **Step 1:** Divide all the minterms including the don't cares (if present) into groups

**True minterms**

| Minterm | ABCD |
|---|---|
| 0 | 0000 |
| 3 | 0011 |
| 5 | 0101 |
| 6 | 0110 |
| 7 | 0111 |
| 12 | 1100 |
| 10 | 1010 |
| 13 | 1101 |

**Don't cares**

| Minterm | ABCD |
|---|---|
| 2 | 0010 |
| 9 | 1001 |
| 15 | 1111 |

# QM Method Example 1

- **Step 1:** Divide all the minterms including the don't cares (if present) into groups and list then inform of a implicant table

| Group | Minterm | ABCD | |
|---|---|---|---|
| G0 | 0 | 0000 | 0 |
| G1 | 2 | 0010 | 1 |
| G2 | 3 | 0011 | 2 |
| | 5 | 0101 | |
| | 6 | 0110 | |
| | 9 | 1001 | |
| | 10 | 1010 | |
| | 12 | 1100 | |
| G3 | 7 | 0111 | 3 |
| | 13 | 1101 | |
| G4 | 15 | 1111 | 4 |

# of ones in the binary code

# QM Method Example 1

- **Step 2:** Merge minterms from adjacent groups to form a new implicant table

| Group | Minterm | ABCD | Merge Mark |
|---|---|---|---|
| G0 | 0 | 0000 | √ |
| G1 | 2 | 0010 | √ |
| G2 | 3 | 0011 | √ |
| | 5 | 0101 | √ |
| | 6 | 0110 | √ |
| | 9 | 1001 | √ |
| | 10 | 1010 | √ |
| | 12 | 1100 | √ |
| G3 | 7 | 0111 | √ |
| | 13 | 1101 | √ |
| G4 | 15 | 1111 | √ |

| Group | Merged Minterms | ABCD |
|---|---|---|
| G0’ | 0, 2 | 00-0 |
| G1’ | 2, 3 | 001- |
| | 2, 6 | 0-10 |
| | 2, 10 | -010 |
| G2’ | 3, 7 | 0-11 |
| | 5, 7 | 01-1 |
| | 6, 7 | 011- |
| | 5, 13 | -101 |
| | 9, 13 | 1-01 |
| | 12, 13 | 110- |
| G3’ | 7,15 | -111 |
| | 13,15 | 11-1 |

# QM Method Example 1

- **Step 3:** Repeat step 2 until no more merging is possible

| Groups | Merged Minterms | ABCD | Merge Mark |
|---|---|---|---|
| G0’ | 0, 2 | 00-0 | |
| G1’ | 2, 3 | 001- | √ |
| | 2, 6 | 0-10 | √ |
| | 2, 10 | -010 | |
| G2’ | 3, 7 | 0-11 | √ |
| | 5, 7 | 01-1 | √ |
| | 6, 7 | 011- | √ |
| | 5, 13 | -101 | √ |
| | 9, 13 | 1-01 | |
| | 12, 13 | 110- | |
| G3’ | 7,15 | -111 | √ |
| | 13,15 | 11-1 | √ |

| Group | Merged Minterms | ABCD |
|---|---|---|
| G1” | 2, 3, 6, 7 | 0-1- |
| | 2, 6, 3, 7 | 0-1- |
| G2” | 5, 7, 13, 15 | -1-1 |
| | 5, 13, 7, 15 | -1-1 |

Check for all unticked groupings of step 2 and the newly formed groupings in this step for further possible simplification

# QM Method Example 1

- **Step 3:** Repeat step 2 until no more merging is possible

| Group | Merged Minterm | ABCD | Merge Mark |
|---|---|---|---|
| G0” | 0, 2 | 00-0 | |
| G1” | 2, 3, 6, 7 | 0-1- | |
| | 2, 10 | -010 | |
| G2” | 5, 7, 13, 15 | -1-1 | |
| | 9, 13 | 1-01 | |
| | 12, 13 | 110- | |

No more merging is possible

- Thus the identified groupings are the prime implicants for the given Boolean function

# QM Method Example 1

- **Step 4:** Form a cover table listing all prime implicants against the involved minterms excluding the don't cares

$$F(A,B,C,D) = \Sigma m(0,3,5,6,7,10,12,13) + \Sigma d(2,9,15)$$

| Minterm ID | ABCD |
|---|---|
| 0,2 | 00-0 |
| 2,3,6,7 | 0-1- |
| 2,10 | -010 |
| 5,7,13,15 | -1-1 |
| 9,13 | 1-01 |
| 12,13 | 110- |

| Minterm ID | $\bar{A}\bar{B}\bar{D}$ | $\bar{A}C$ | $\bar{B}C\bar{D}$ | $BD$ | $AB\bar{C}$ | $A\bar{C}D$ |
|---|---|---|---|---|---|---|
| 0 | x | | | | | |
| 3 | | x | | | | |
| 5 | | | | x | | |
| 6 | | x | | | | |
| 7 | | x | | x | | |
| 10 | | | x | | | |
| 12 | | | | | x | |
| 13 | | | | x | x | x |

# QM Method Example 1

- **Step 5:** Identify the essential minterms (EMT), and hence the essential prime implicants (EPI)

EMT

EPI

| Minterm ID | $\bar{A}\bar{B}\bar{D}$ | $\bar{A}C$ | $\bar{B}C\bar{D}$ | $BD$ | $AB\bar{C}$ | $A\bar{C}D$ |
|---|---|---|---|---|---|---|
| 0 | x | | | | | |
| 3 | | x | | | | |
| 5 | | | | x | | |
| 6 | | x | | | | |
| 7 | | x | | x | | |
| 10 | | | x | | | |
| 12 | | | | | x | |
| 13 | | | | x | x | x |

- Minimized SOP expression:

$$\boldsymbol{F(A,B,C,D) = \bar{A}\bar{B}\bar{D} + \bar{A}C + \bar{B}C\bar{D} + BD + AB\bar{C}}$$

# QM Method Example 2 (compact manner)

Minimize $F(A,B,C,D) = \Sigma m(2,3,6,7,8,10,11,12,14,15)$

| | Step 1 | | | Step 2 | | | Step 3 | | | Step 4 | |
|---|---|---|---|---|---|---|---|---|---|---|---|
| 1 | 2 | √ | 0010 | 2,3 | √ | 001- | 2,3,6,7 | √ | 0-1- | 2,3,6,7,10,14,11,15 | - -1- |
| | 8 | √ | 1000 | 2,6 | √ | 0-10 | 2,6,3,7 | √ | 0-1- | 2,3,10,11, 6,7,14,15 | - -1- |
| | | | | 2,10 | √ | -010 | 2,3,10,11 | √ | -01- | 2,6,3,7, 10,11,14,15 | - -1- |
| 2 | 3 | √ | 0011 | 8,10 | √ | 10-0 | 2,10,3,11 | √ | -01- | 2,6,10,14, 3,7,11,15 | - -1- |
| | 6 | √ | 0110 | 8,12 | √ | 1-00 | 2,6,10,14 | √ | - -10 | 2,10,3,11,6, 4,7,15 | - -1- |
| | 10 | √ | 1010 | | | | 2,10,6,14 | √ | - -10 | 2,10,6,14, 3,11,7,15 | - -1- |
| | 12 | √ | 1100 | 3,7 | √ | 0-11 | 8,10,12,14 | | 1- -0 | | |
| | | | | 3,11 | √ | -011 | 8,12,10,14 | | 1- -0 | | |
| 3 | 7 | √ | 0111 | 6,7 | √ | 011- | | | | | |
| | 11 | √ | 1011 | 6,14 | √ | -110 | 3,7,11,15 | √ | - -11 | | |
| | 14 | √ | 1110 | 10,11 | √ | 101- | 3,11,7,15 | √ | - - 11 | | |
| | | | | 10,14 | √ | 1-10 | 6,7,14,15 | √ | -11- | | |
| 4 | 15 | √ | 1111 | 12,14 | √ | 11-0 | 6,14,7,15 | √ | -11- | | |
| | | | | | | | 10,11,14,15 | √ | 1-1- | | |
| | | | | 7,15 | √ | -111 | 10,14,11,15 | √ | 1-1- | | |
| | | | | 11,15 | √ | 1-11 | | | | | |
| | | | | 14,15 | √ | 111- | | | | | |

Prime implicants

# QM Method Example 2

- Finding the essential prime implicants

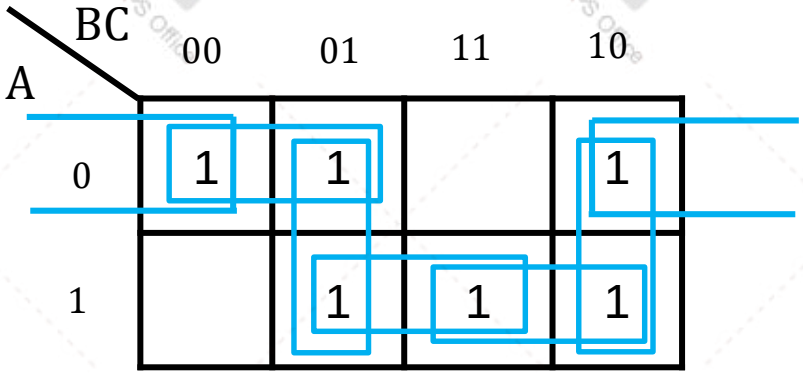
| Prime implicant | Covered minterms | Minterms | | | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|---|---|
| | | 2 | 3 | 6 | 7 | 8 | 10 | 11 | 12 | 14 | 15 |
| $A\overline{D}$ | 8,12,10,14 | | | | | x | x | | x | x | |
| $C$ | 2,3,6,7,10,11,14,15 | x | x | x | x | | x | x | | x | x |
| | | | | | | | | | | | |

$$\boldsymbol{F(A,B,C,D) = A\overline{D} + C}$$

# Example-3

- Minimize $F(A, B, C) = \Sigma m(0, 1, 2, 5, 6, 7)$

- Represent the given function on K-Map and perform the grouping to achieve the minimized SOP form




- How to choose the minimal grouping?

# Example-3 (contd.)

- What if we minimize $F(A, B, C) = \Sigma m(0, 1, 2, 5, 6, 7)$ using QM method?
- List the minterms of the given Boolean function in an implicant table and determine the prime implicants

| Step 1 | | | Step 2 | | Step 3 |
|---|---|---|---|---|---|
| 0 | √ | 000 | 0, 1 | 00- | |
| | | | 0, 2 | 0-0 | |
| 1 | √ | 001 | | | |
| 2 | √ | 010 | 1, 5 | -01 | |
| | | | 2, 6 | -10 | |
| 5 | √ | 101 | | | |
| 6 | √ | 110 | 5, 7 | 1-1 | |
| | | | 6, 7 | 11- | |
| 7 | √ | 111 | | | |

Note the code structures of the groupings.

No further merging is possible so all these are prime implicants

# Example-3 (contd.)

- Draw the cover table

| PI | Covered minterms | Minterms 0 | 1 | 2 | 5 | 6 | 7 |
|---|---|---|---|---|---|---|---|
| $p_1 = \bar{A}\bar{B}$ | 0, 1 | x | x | | | | |
| $p_2 = \bar{A}\bar{C}$ | 0, 2 | x | | x | | | |
| $p_3 = \bar{B}C$ | 1, 5 | | x | | x | | |
| $p_4 = B\bar{C}$ | 2, 6 | | | x | | x | |
| $p_5 = AC$ | 5, 6 | | | | x | | x |
| $p_6 = AB$ | 6, 7 | | | | | x | x |

- Note, each minterm is covered by at least two PIs so there is no EPI
- Such functions are known as cyclic Boolean function

# Example-3 (contd.)

- **Solution:** Break the deadlock by arbitrarily choosing one of the PIs as EPI and draw a **reduced cover table**
- For that, delete the row corresponding to chosen PI and the columns corresponding to its minterms in the original cover table as shown below (for **choice of $p_1$ as EPI**)

| PI | Covered minterms | Minterms 0 | 1 | 2 | 5 | 6 | 7 |
|---|---|---|---|---|---|---|---|
| $p_1$ √ | 0, 1 | x | x | | | | |
| $p_2$ | 0, 2 | x | | x | | | |
| $p_3$ | 1, 5 | | x | | x | | |
| $p_4$ | 2, 6 | | | x | | x | |
| $p_5$ | 5, 6 | | | | x | | x |
| $p_6$ | 6, 7 | | | | | x | x |

Reduced Cover Table

| PI | Covered minterms | Minterms 2 | 5 | 6 | 7 |
|---|---|---|---|---|---|
| $p_2$ | 0, 2 | x | | | |
| $p_3$ | 1, 5 | | x | | |
| $p_4$ | 2, 6 | x | | x | |
| $p_5$ | 5, 6 | | x | | x |
| $p_6$ | 6, 7 | | | x | x |

# Example-3 (contd.)

- Now, delete all PIs which get covered by other PIs in the reduced cover table.

| PI | Covered minterms | Minterms 2 | 5 | 6 | 7 |
|---|---|---|---|---|---|
| $p_2$ | 0, 2 | x | | | |
| $p_3$ | 1, 5 | | x | | |
| $p_4$ √ | 2, 6 | x | | x | |
| $p_5$ √ | 5, 6 | | x | | x |
| $p_6$ | 6, 7 | | | x | x |

- Break the deadlock in the further reduced cover table as done previously and produce a cover for all the minterms

$$F(A, B, C) = \overline{A}\,\overline{B} + B\overline{C} + AC$$

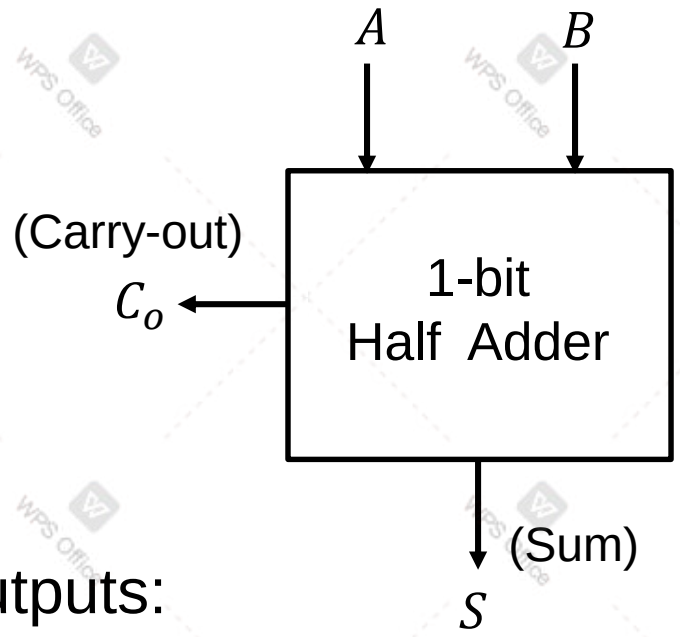
# Combinational Logic Circuits

# Half Adder Circuit

- It takes two single-bit inputs ($A$ and $B$) and produces two single-bit outputs denoted as sum ($S$) and carry-out ($C_o$)
- Truth table

| Input | | Output | |
|---|---|---|---|
| $A$ | $B$ | $C_o$ | $S$ |
| 0 | 0 | 0 | 0 |
| 0 | 1 | 0 | 1 |
| 1 | 0 | 0 | 1 |
| 1 | 1 | 1 | 0 |

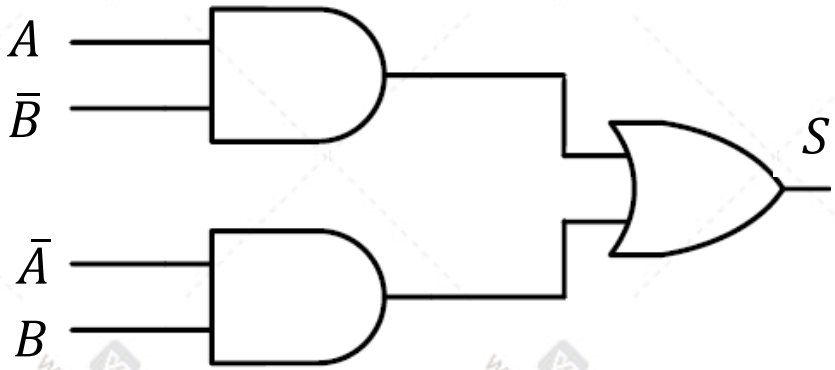
Block diagram



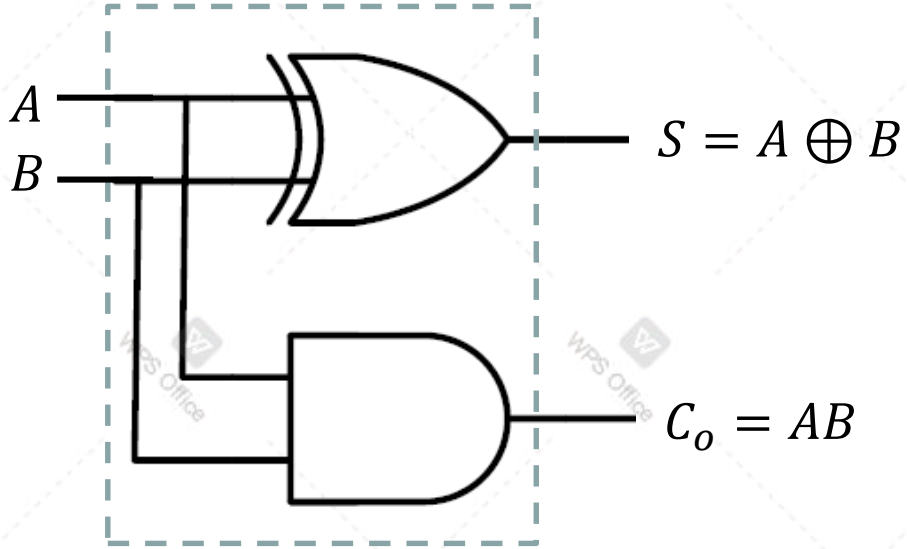

- The expression for the two outputs:

$$C_o = AB \text{ and } S = \bar{A}B + A\bar{B} = A \oplus B$$

# Half Adder Circuit





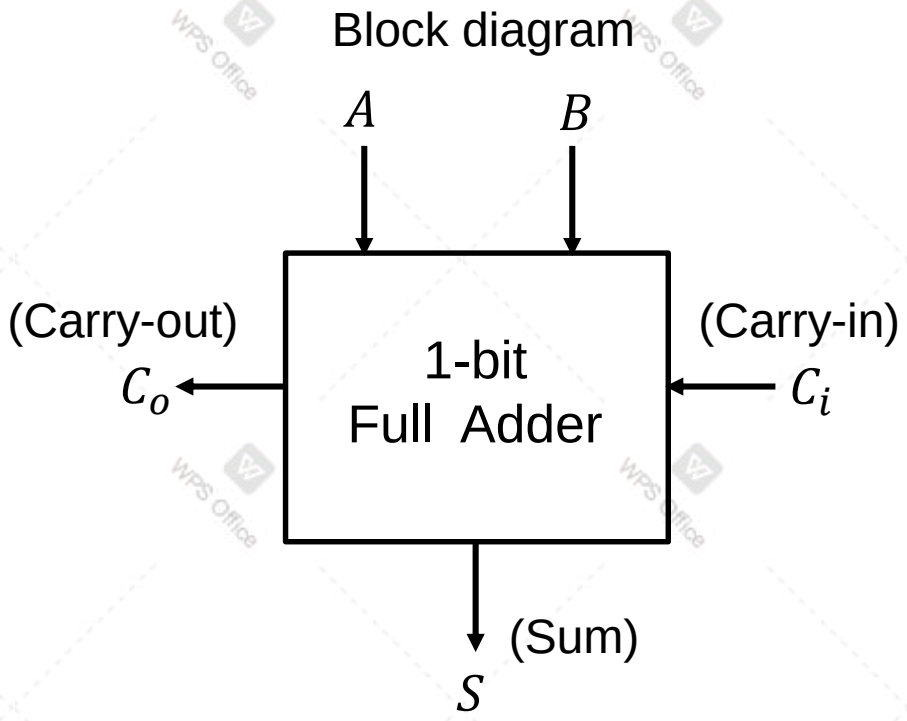



Half Adder

# Full Adder Circuit

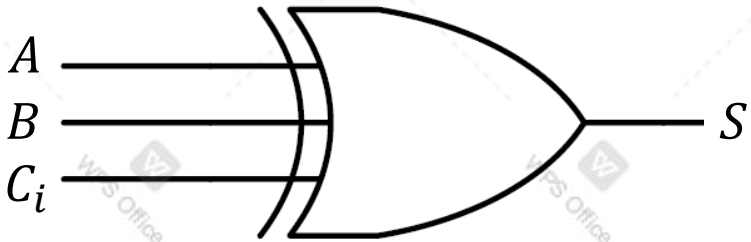
- Half adder accepts only 2 input bits. If two $n$-bit binary numbers are to be added, there can be a carry bit in various places
- Full adder takes 3 inputs, i.e. a carry-in ($C_i$) bit also

| | $A$ | $B$ | $C_i$ | $S$ | $C_o$ |
|---|---|---|---|---|---|
| | 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 |



# Full Adder Circuit: Sum

| $A$ | $B$ | $C_i$ | $S$ | $C_o$ |
|---|---|---|---|---|
| 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 |



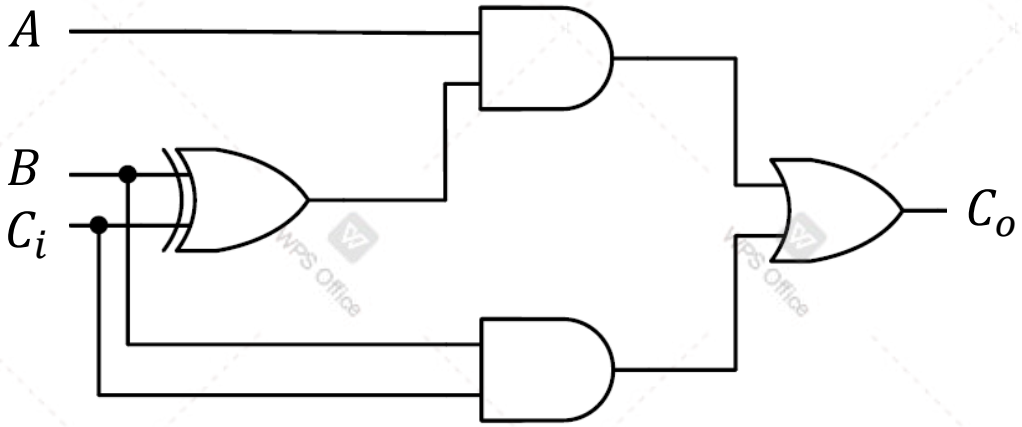

$$S = \bar{A}\bar{B}C_i + \bar{A}B\bar{C}_i + A\bar{B}\bar{C}_i + ABC_i$$

$$S = \bar{A}(\bar{B}C_i + B\bar{C}_i) + A(\bar{B}\bar{C}_i + BC_i)$$

$$S = \bar{A}(B \oplus C_i) + A\overline{(B \oplus C_i)} = A \oplus B \oplus C_i$$

# Full Adder Circuit: Output Carry

| $A$ | $B$ | $C_i$ | $S$ | $C_o$ |
|---|---|---|---|---|
| 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 |



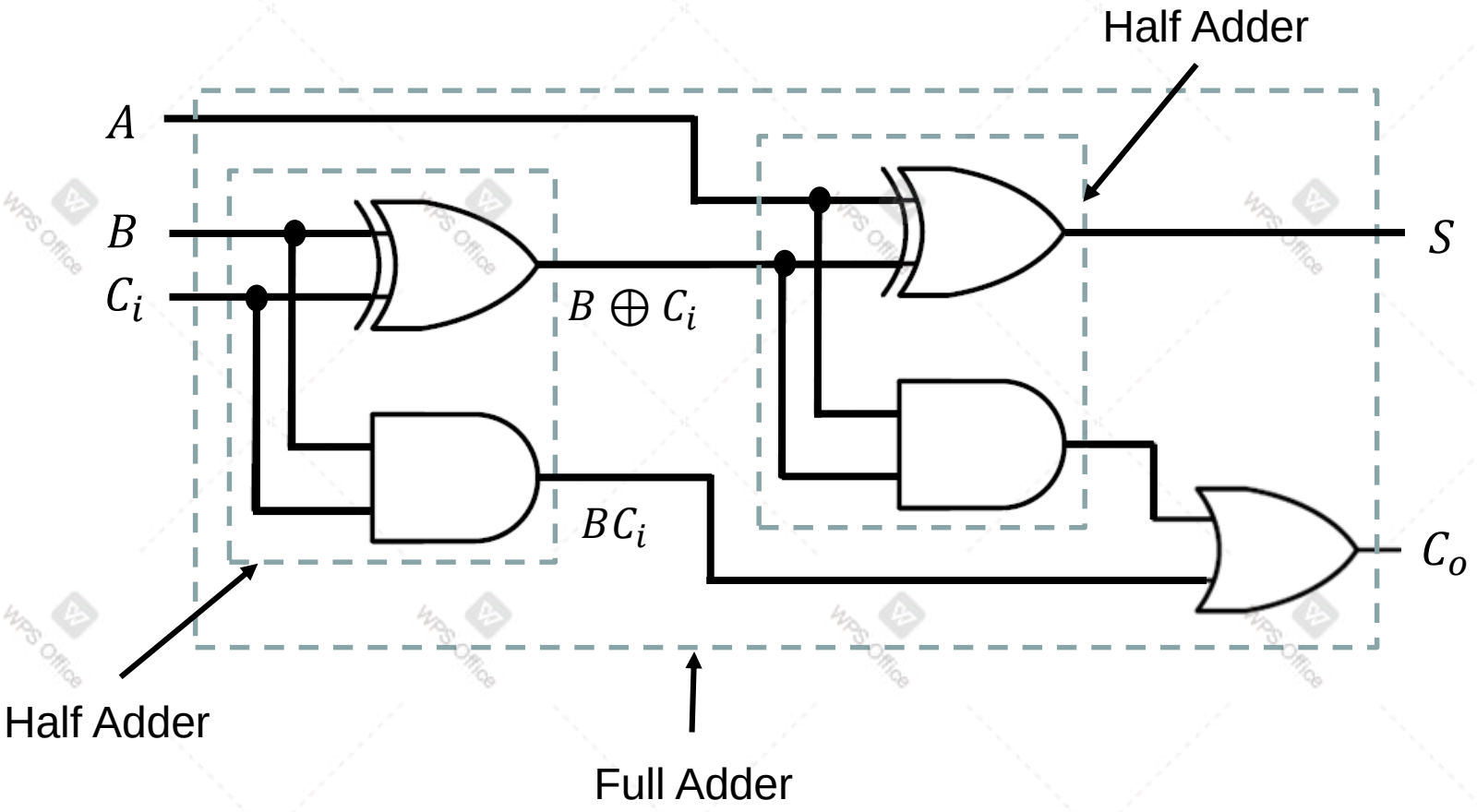

$$C_o = \bar{A}BC_i + A\bar{B}C_i + AB\bar{C}_i + ABC_i$$

$$C_o = BC_i(\bar{A} + A) + A(\bar{B}C_i + B\bar{C}_i)$$

$$C_o = BC_i + A(B \oplus C_i)$$

# Realizing Full Adder using two HAs

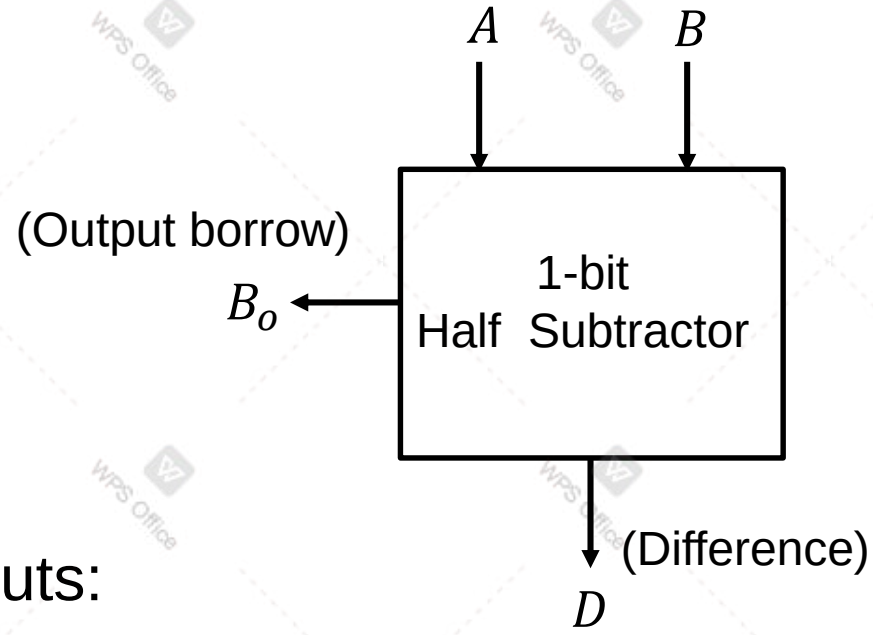
$$S = A \oplus B \oplus C_i \qquad C_o = BC_i + A(B \oplus C_i)$$



# Half Subtractor Circuit

- Performs subtraction of two single bits, one is minuend $A$ and the other is subtranhed $B$
- It takes 2 input bits and produces two output bits called difference ($D$) and output borrow ($B_o$)

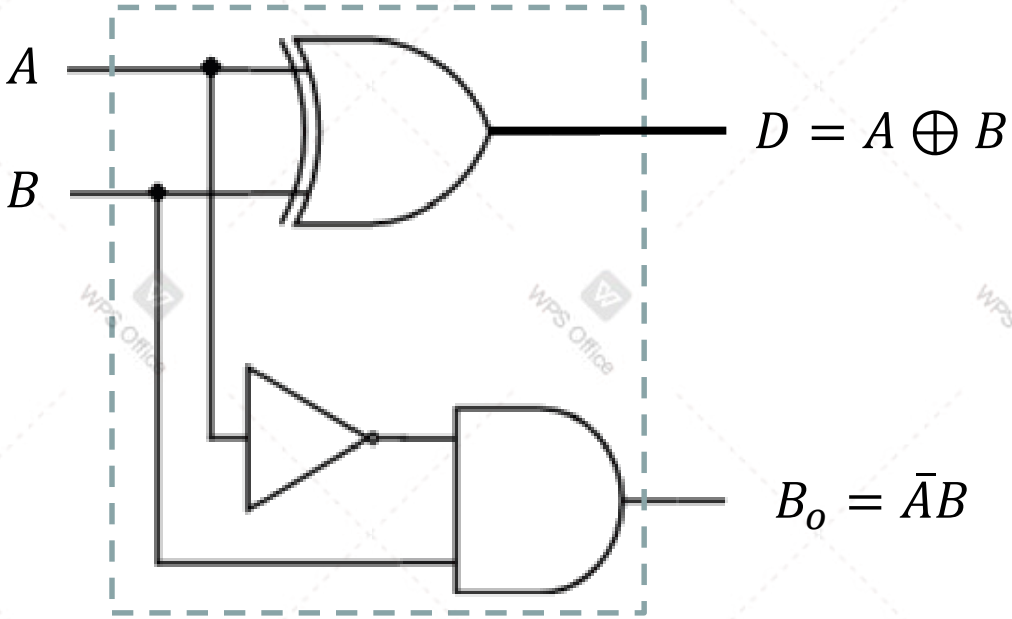
| Input | | Output | |
|---|---|---|---|
| $A$ | $B$ | $D$ | $B_o$ |
| 0 | 0 | 0 | 0 |
| 0 | 1 | 1 | 1 |
| 1 | 0 | 1 | 0 |
| 1 | 1 | 0 | 0 |




- The expression of the outputs:

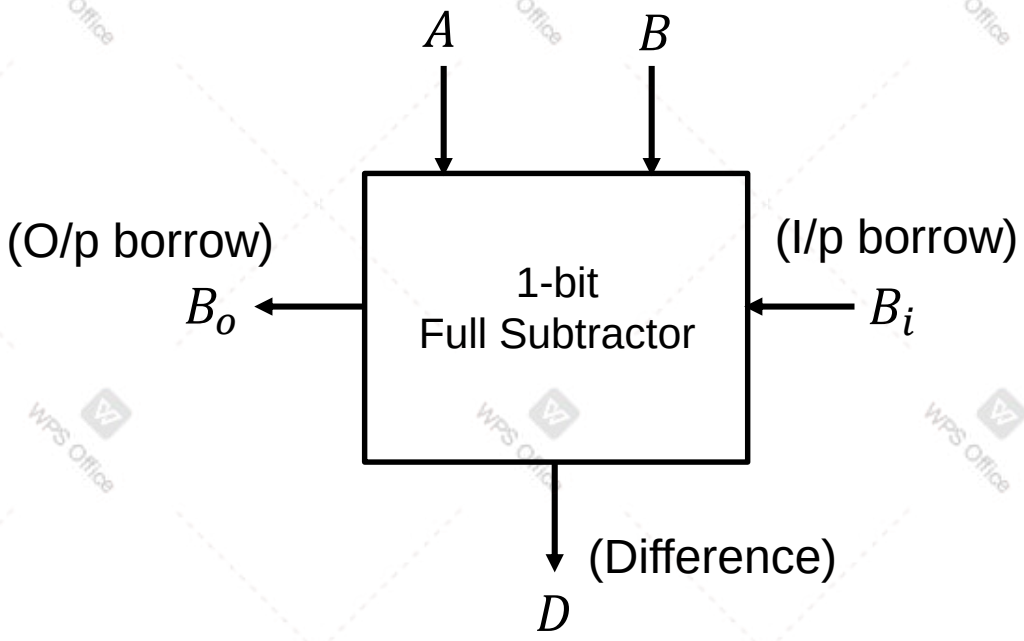
$$B_o = \bar{A}B \text{ and } D = \bar{A}B + A\bar{B} = A \oplus B$$

# Half Subtractor Circuit



# Full Subtractor Circuit

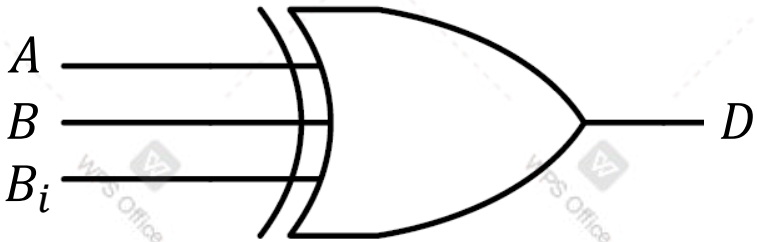
- Performs subtraction of two bits by taking into account the borrow of the previous adjacent lower bit place
- It has three inputs $A$, $B$ and $B_i$ denoting the minuend, subtrahend, and previous borrow, respectively




| $A$ | $B$ | $B_i$ | $D$ | $B_o$ |
|---|---|---|---|---|
| 0 | 0 | 0 | 0 | 0 |
| 0 | 0 | 1 | 1 | 1 |
| 0 | 1 | 0 | 1 | 1 |
| 0 | 1 | 1 | 0 | 1 |
| 1 | 0 | 0 | 1 | 0 |
| 1 | 0 | 1 | 0 | 0 |
| 1 | 1 | 0 | 0 | 0 |
| 1 | 1 | 1 | 1 | 1 |

# Full Subtractor Circuit: Difference

| $A$ | $B$ | $B_i$ | $D$ | $B_o$ |
|---|---|---|---|---|
| 0 | 0 | 0 | 0 | 0 |
| 0 | 0 | 1 | 1 | 1 |
| 0 | 1 | 0 | 1 | 1 |
| 0 | 1 | 1 | 0 | 1 |
| 1 | 0 | 0 | 1 | 0 |
| 1 | 0 | 1 | 0 | 0 |
| 1 | 1 | 0 | 0 | 0 |
| 1 | 1 | 1 | 1 | 1 |

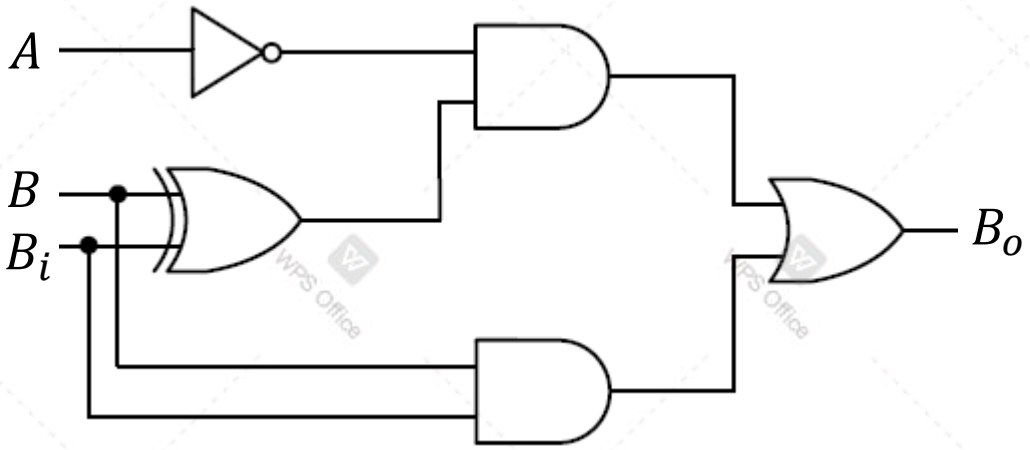



$$D = \bar{A}\bar{B}B_i + \bar{A}B\bar{B_i} + A\bar{B}\bar{B_i} + ABB_i = \bar{A}(\bar{B}B_i + B\bar{B_i}) + A(\bar{B}\bar{B_i} + BB_i)$$

$$D = \bar{A}(B \oplus B_i) + A\overline{(B \oplus B_i)} = A \oplus B \oplus B_i$$

# Full Subtractor Circuit: Output Barrow

| $A$ | $B$ | $B_i$ | $D$ | $B_o$ |
|---|---|---|---|---|
| 0 | 0 | 0 | 0 | 0 |
| 0 | 0 | 1 | 1 | 1 |
| 0 | 1 | 0 | 1 | 1 |
| 0 | 1 | 1 | 0 | 1 |
| 1 | 0 | 0 | 1 | 0 |
| 1 | 0 | 1 | 0 | 0 |
| 1 | 1 | 0 | 0 | 0 |
| 1 | 1 | 1 | 1 | 1 |




$$B_o = \bar{A}\bar{B}B_i + \bar{A}B\bar{B_i} + A\bar{B}B_i + ABB_i = BB_i(\bar{A} + A) + \bar{A}(\bar{B}B_i + B\bar{B_i})$$

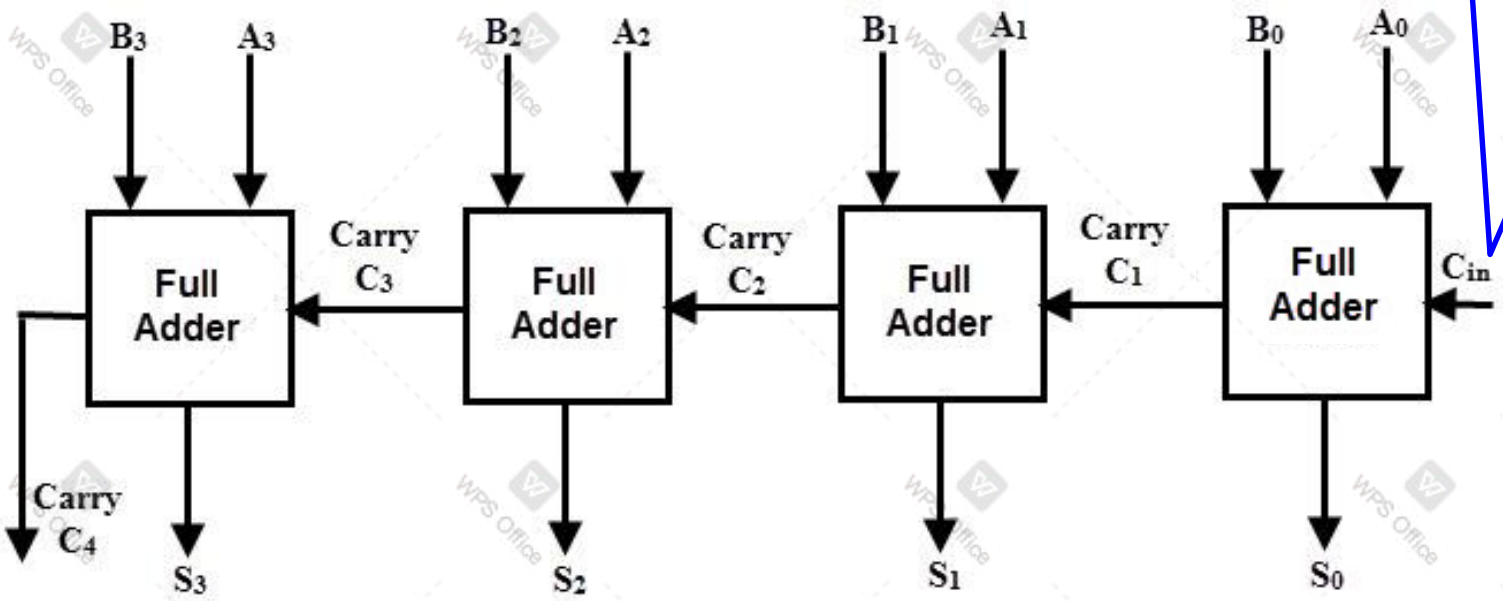
$$B_o = BB_i + \bar{A}(B \oplus B_i)$$

# Parallel Binary Adder

Addition of two 4-bit binary numbers:

$$\begin{array}{r} A_3\ A_2\ A_1\ A_0 \\ +\ \ B_3\ B_2\ B_1\ B_0 \\ \hline C_4\ S_3\ S_2\ S_1\ S_0 \end{array}$$

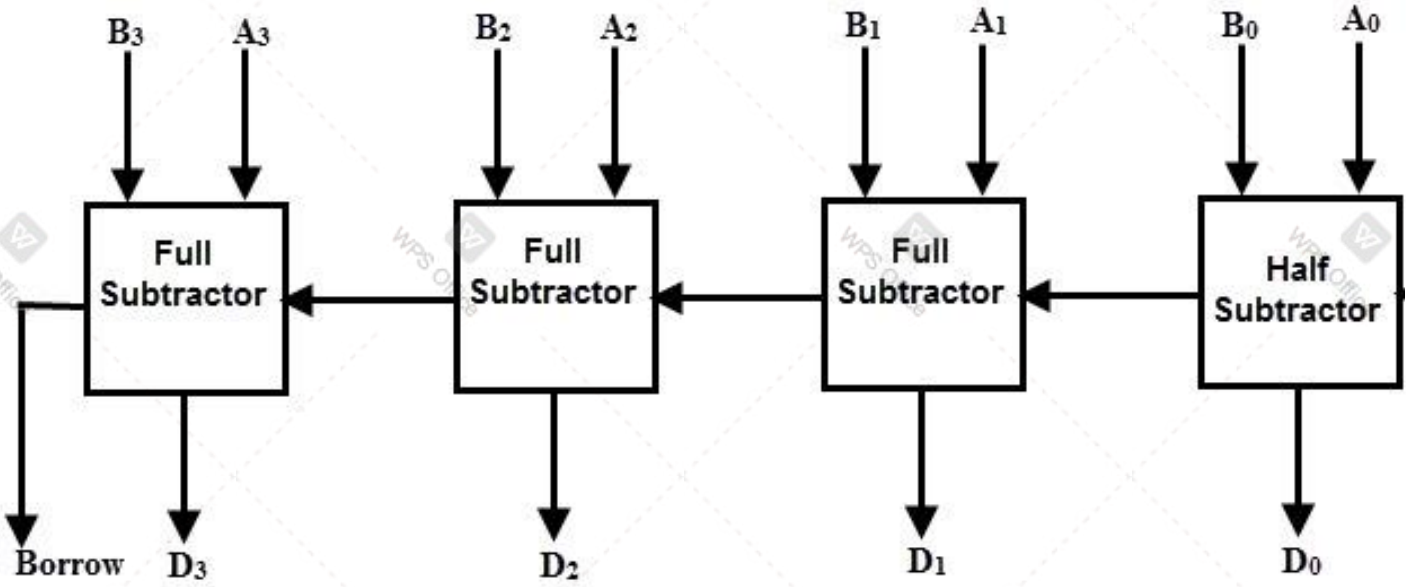
In LSB full-adder module $C_{in}$ is set to zero or we can also use a half-adder




Can be extended to add two n-bit binary numbers

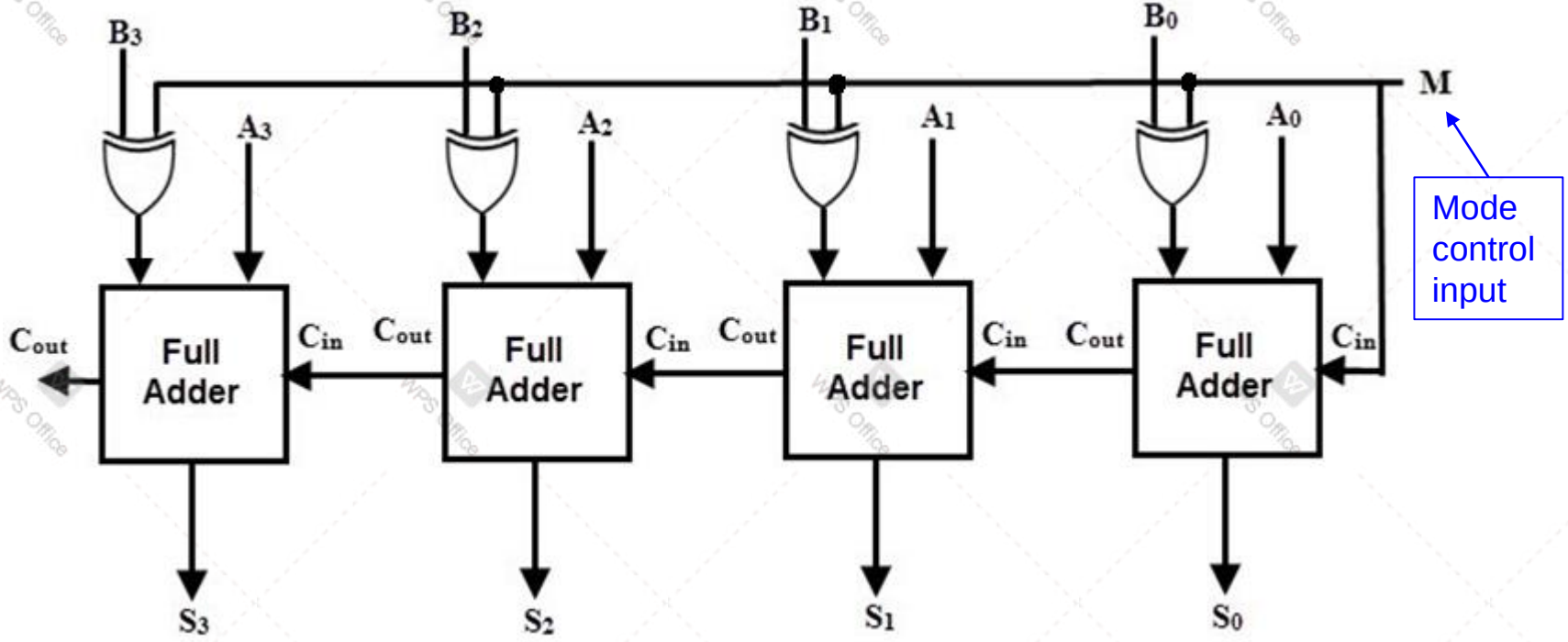
# Parallel Binary Subtractor

Subtraction of two 4-bit binary numbers:




Can be extended to subtract two n-bit binary numbers

# Parallel Adder / Subtractor




- Recall, the subtraction can also be achieved by adding 2's complement of subtrahend to minuend
- The input $\mathrm{M}$ controls the mode, for $\mathrm{M} = 0$ the circuit acts as adder and for $\mathrm{M} = 1$ it acts as subtractor
- Note, for $\mathrm{M} = 1$, each $\mathrm{B_i}$ bit get inverted with $1$ added to the LSB

# Code Converters

# Binary-to-BCD Code Converter

Input (Binary code): 4-bit

Output (BCD code): 5-bit (minimum)

$$D_4 = \Sigma m(10,11,12,13,14,15)$$

$$D_3 = \Sigma m(8,9)$$

$$D_2 = \Sigma m(4,5,6,7,14,15)$$

$$D_1 = \Sigma m(2,3,6,7,14,15)$$

$$D_0 = \Sigma m(1,3,5,7,9,11,13,15)$$

| Decimal value | Binary code $B_3B_2B_1B_0$ | BCD code $D_4D_3D_2D_1D_0$ |
|---|---|---|
| 0 | 0 0 0 0 | 0 0 0 0 0 |
| 1 | 0 0 0 1 | 0 0 0 0 1 |
| 2 | 0 0 1 0 | 0 0 0 1 0 |
| 3 | 0 0 1 1 | 0 0 0 1 1 |
| 4 | 0 1 0 0 | 0 0 1 0 0 |
| 5 | 0 1 0 1 | 0 0 1 0 1 |
| 6 | 0 1 1 0 | 0 0 1 1 0 |
| 7 | 0 1 1 1 | 0 0 1 1 1 |
| 8 | 1 0 0 0 | 0 1 0 0 0 |
| 9 | 1 0 0 1 | 0 1 0 0 1 |
| 10 | 1 0 1 0 | 1 0 0 0 0 |
| 11 | 1 0 1 1 | 1 0 0 0 1 |
| 12 | 1 1 0 0 | 1 0 0 1 0 |
| 13 | 1 1 0 1 | 1 0 0 1 1 |
| 14 | 1 1 1 0 | 1 0 1 0 0 |
| 15 | 1 1 1 1 | 1 0 1 0 1 |

# Binary-to-BCD Code Converter

For $D_4$ output

| $B_3B_2$ \ $B_1B_0$ | 00 | 01 | 11 | 10 |
|---|---|---|---|---|
| 00 | 0 | 0 | 0 | 0 |
| 01 | 0 | 0 | 0 | 0 |
| 11 | 1 | 1 | 1 | 1 |
| 10 | 0 | 0 | 1 | 1 |

$$D_4 = B_3B_2 + B_3B_1$$

For $D_3$ output

| $B_3B_2$ \ $B_1B_0$ | 00 | 01 | 11 | 10 |
|---|---|---|---|---|
| 00 | 0 | 0 | 0 | 0 |
| 01 | 0 | 0 | 0 | 0 |
| 11 | 0 | 0 | 0 | 0 |
| 10 | 1 | 1 | 0 | 0 |

$$D_3 = B_3\overline{B}_2\overline{B}_1$$

For $D_2$ output

| $B_3B_2$ \ $B_1B_0$ | 00 | 01 | 11 | 10 |
|---|---|---|---|---|
| 00 | 0 | 0 | 0 | 0 |
| 01 | 1 | 1 | 1 | 1 |
| 11 | 0 | 0 | 1 | 1 |
| 10 | 0 | 0 | 0 | 0 |

$$D_2 = \overline{B}_3B_2 + B_2B_1$$

For $D_1$ output

| $B_3B_2$ \ $B_1B_0$ | 00 | 01 | 11 | 10 |
|---|---|---|---|---|
| 00 | 0 | 0 | 1 | 1 |
| 01 | 0 | 0 | 1 | 1 |
| 11 | 1 | 1 | 0 | 0 |
| 10 | 0 | 0 | 0 | 0 |

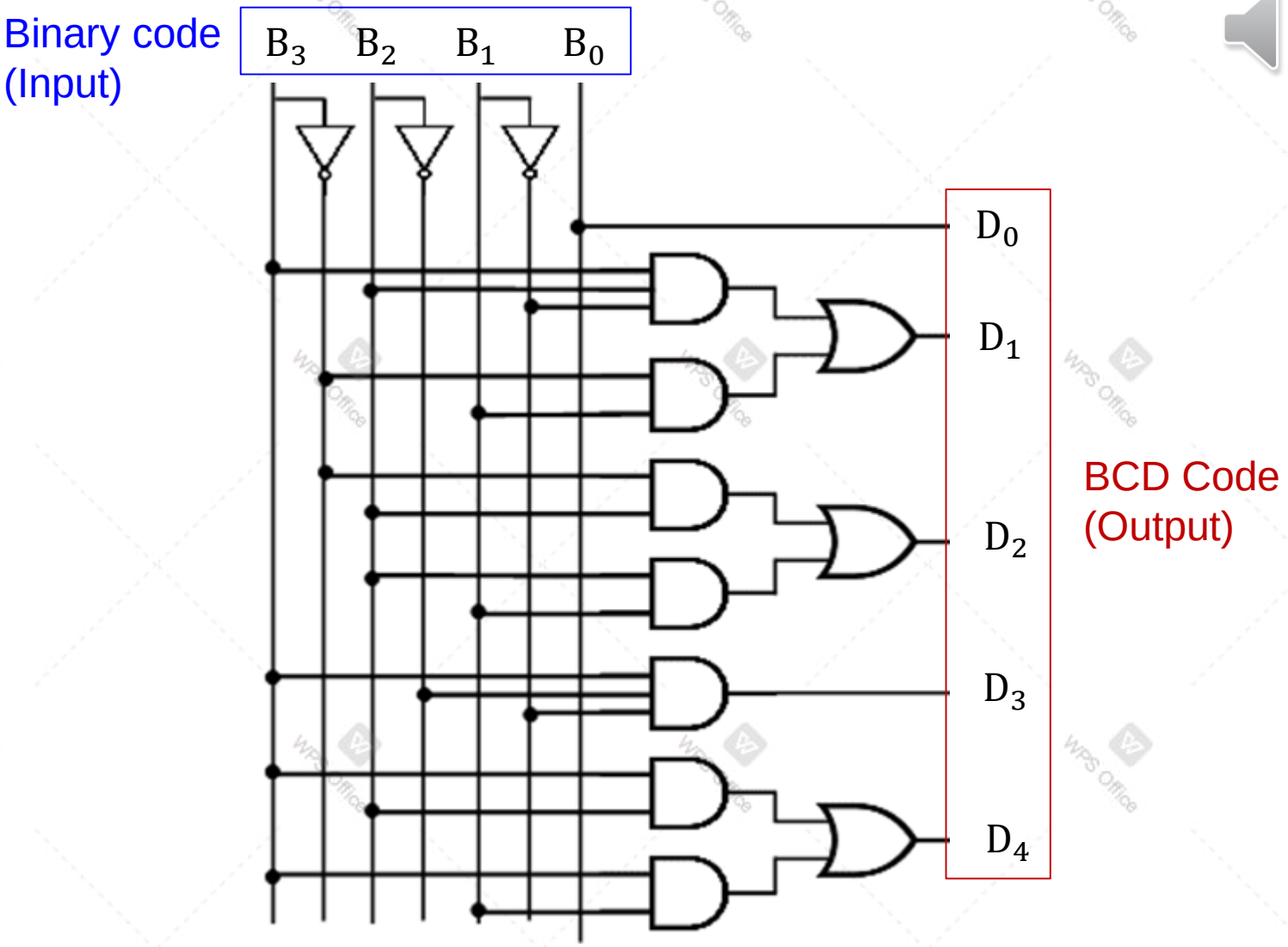
$$D_1 = B_3B_2\overline{B}_1 + \overline{B}_3B_1$$

For $D_0$ output

| $B_3B_2$ \ $B_1B_0$ | 00 | 01 | 11 | 10 |
|---|---|---|---|---|
| 00 | 0 | 1 | 1 | 0 |
| 01 | 0 | 1 | 1 | 0 |
| 11 | 0 | 1 | 1 | 0 |
| 10 | 0 | 1 | 1 | 0 |

$$D_0 = B_0$$

# Binary-to-BCD Code Converter Circuit



# BCD-to-Gray Code Converter

BCD code input: 4-bit,  Gray code output: 4 bit

| Decimal value | BCD code $D_3D_2D_1D_0$ | Gray code $G_3G_2G_1G_0$ |
|---|---|---|
| 0 | 0 0 0 0 | 0 0 0 0 |
| 1 | 0 0 0 1 | 0 0 0 1 |
| 2 | 0 0 1 0 | 0 0 1 1 |
| 3 | 0 0 1 1 | 0 0 1 0 |
| 4 | 0 1 0 0 | 0 1 1 0 |
| 5 | 0 1 0 1 | 0 1 1 1 |
| 6 | 0 1 1 0 | 0 1 0 1 |
| 7 | 0 1 1 1 | 0 1 0 0 |
| 8 | 1 0 0 0 | 1 1 0 0 |
| 9 | 1 0 0 1 | 1 1 0 1 |

$$G_3 = \Sigma m(8,9) + \Sigma d(10,\cdots,15)$$

$$G_2 = \Sigma m(4,5,6,7,8,9) + \Sigma d(10,\cdots,15)$$

$$G_1 = \Sigma m(2,3,4,5) + \Sigma d(10,\cdots,15)$$

$$G_0 = \Sigma m(1,2,5,6,9) + \Sigma d(10,\cdots,15)$$

# BCD-to-Gray Code Converter

For $G_3$ output

| $D_3D_2$ \ $D_1D_0$ | 00 | 01 | 11 | 10 |
|---|---|---|---|---|
| 00 | 0 | 0 | 0 | 0 |
| 01 | 0 | 0 | 0 | 0 |
| 11 | X | X | X | X |
| 10 | 1 | 1 | X | X |

$G_3 = D_3$

For $G_2$ output

| $D_3D_2$ \ $D_1D_0$ | 00 | 01 | 11 | 10 |
|---|---|---|---|---|
| 00 | 0 | 0 | 0 | 0 |
| 01 | 1 | 1 | 1 | 1 |
| 11 | X | X | X | X |
| 10 | 1 | 1 | X | X |

$G_2 = D_2 + D_3$

For $G_1$ output

| $D_3D_2$ \ $D_1D_0$ | 00 | 01 | 11 | 10 |
|---|---|---|---|---|
| 00 | 0 | 0 | 1 | 1 |
| 01 | 1 | 1 | 0 | 0 |
| 11 | X | X | X | X |
| 10 | 0 | 0 | X | X |

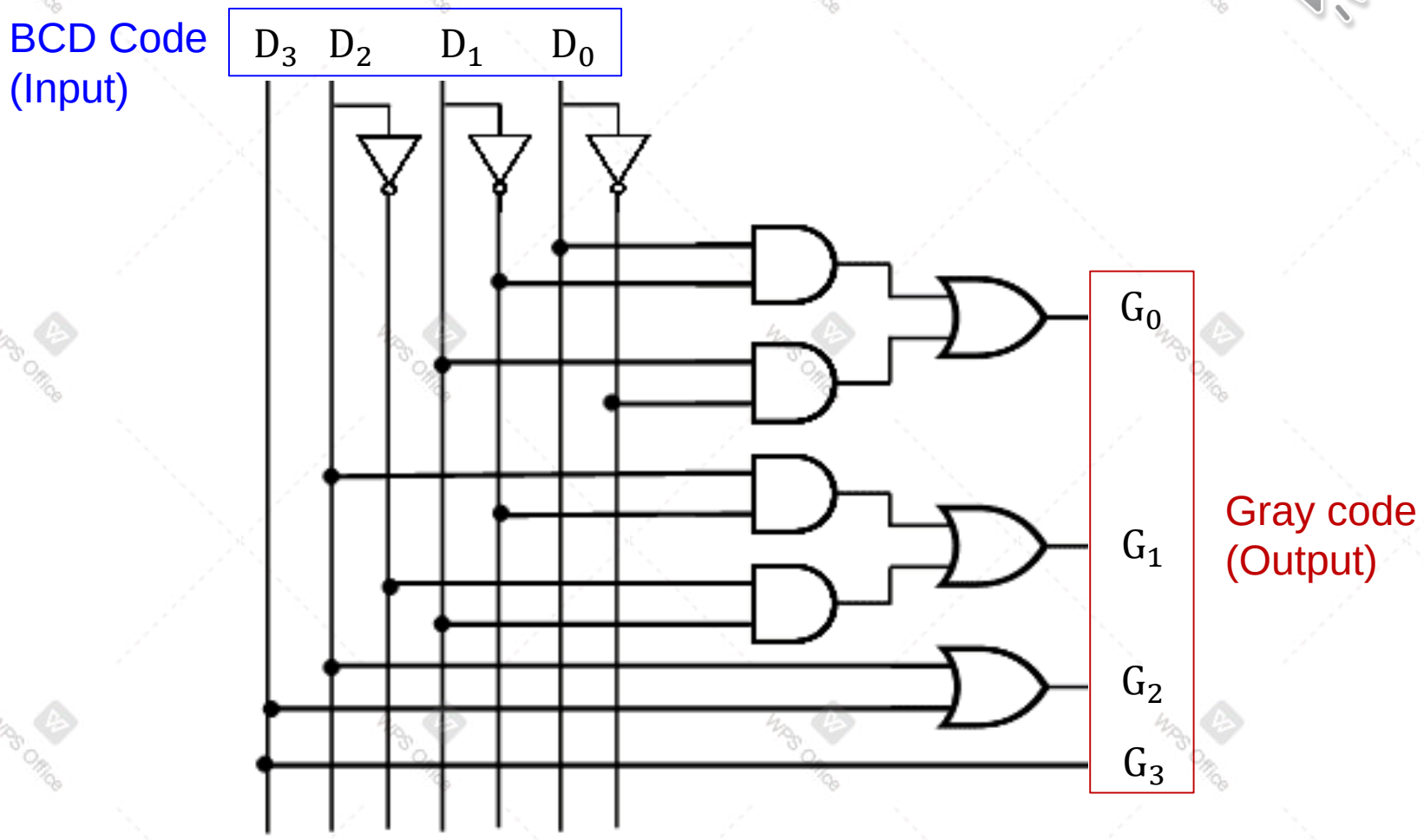
$G_1 = D_2\overline{D}_1 + \overline{D}_2D_1$

For $G_0$ output

| $D_3D_2$ \ $D_1D_0$ | 00 | 01 | 11 | 10 |
|---|---|---|---|---|
| 00 | 0 | 1 | 0 | 1 |
| 01 | 0 | 1 | 0 | 1 |
| 11 | X | X | X | X |
| 10 | 0 | 1 | X | X |

$G_0 = \overline{D}_1D_0 + D_1\overline{D}_0$

# BCD-to-Gray Code Converter Circuit



# BCD to Excess-3 Code Converter

BCD code input: 4-bit,  Excess-3 code output: 4 bit

| Decimal value | BCD code $D_3D_2D_1D_0$ | Excess-3 code $E_3E_2E_1E_0$ |
|---|---|---|
| 0 | 0 0 0 0 | 0 0 1 1 |
| 1 | 0 0 0 1 | 0 1 0 0 |
| 2 | 0 0 1 0 | 0 1 0 1 |
| 3 | 0 0 1 1 | 0 1 1 0 |
| 4 | 0 1 0 0 | 0 1 1 1 |
| 5 | 0 1 0 1 | 1 0 0 0 |
| 6 | 0 1 1 0 | 1 0 0 1 |
| 7 | 0 1 1 1 | 1 0 1 0 |
| 8 | 1 0 0 0 | 1 0 1 1 |
| 9 | 1 0 0 1 | 1 1 0 0 |

$$E_3 = \Sigma m(5,6,7,8,9) + \Sigma d(10,\cdots,15)$$

$$E_2 = \Sigma m(1,2,3,4,9) + \Sigma d(10,\cdots,15)$$

$$E_1 = \Sigma m(0,3,4,7,8) + \Sigma d(10,\cdots,15)$$

$$E_0 = \Sigma m(0,2,4,6,8) + \Sigma d(10,\cdots,15)$$

# BCD to Excess-3 Code Converter

For $E_3$ output

| $D_3D_2$ \ $D_1D_0$ | 00 | 01 | 11 | 10 |
|---|---|---|---|---|
| 00 | 0 | 0 | 0 | 0 |
| 01 | 0 | 1 | 1 | 1 |
| 11 | X | X | X | X |
| 10 | 1 | 1 | X | X |

$$E_3 = D_3 + D_2D_0 + D_2D_1$$

For $E_2$ output

| $D_3D_2$ \ $D_1D_0$ | 00 | 01 | 11 | 10 |
|---|---|---|---|---|
| 00 | 0 | 1 | 1 | 1 |
| 01 | 1 | 0 | 0 | 0 |
| 11 | X | X | X | X |
| 10 | 0 | 1 | X | X |

$$E_2 = D_2\overline{D}_1\overline{D}_0 + \overline{D}_2D_0 + \overline{D}_2D_1$$

For $E_1$ output

| $D_3D_2$ \ $D_1D_0$ | 00 | 01 | 11 | 10 |
|---|---|---|---|---|
| 00 | 1 | 0 | 1 | 0 |
| 01 | 1 | 0 | 1 | 0 |
| 11 | X | X | X | X |
| 10 | 1 | 0 | X | X |

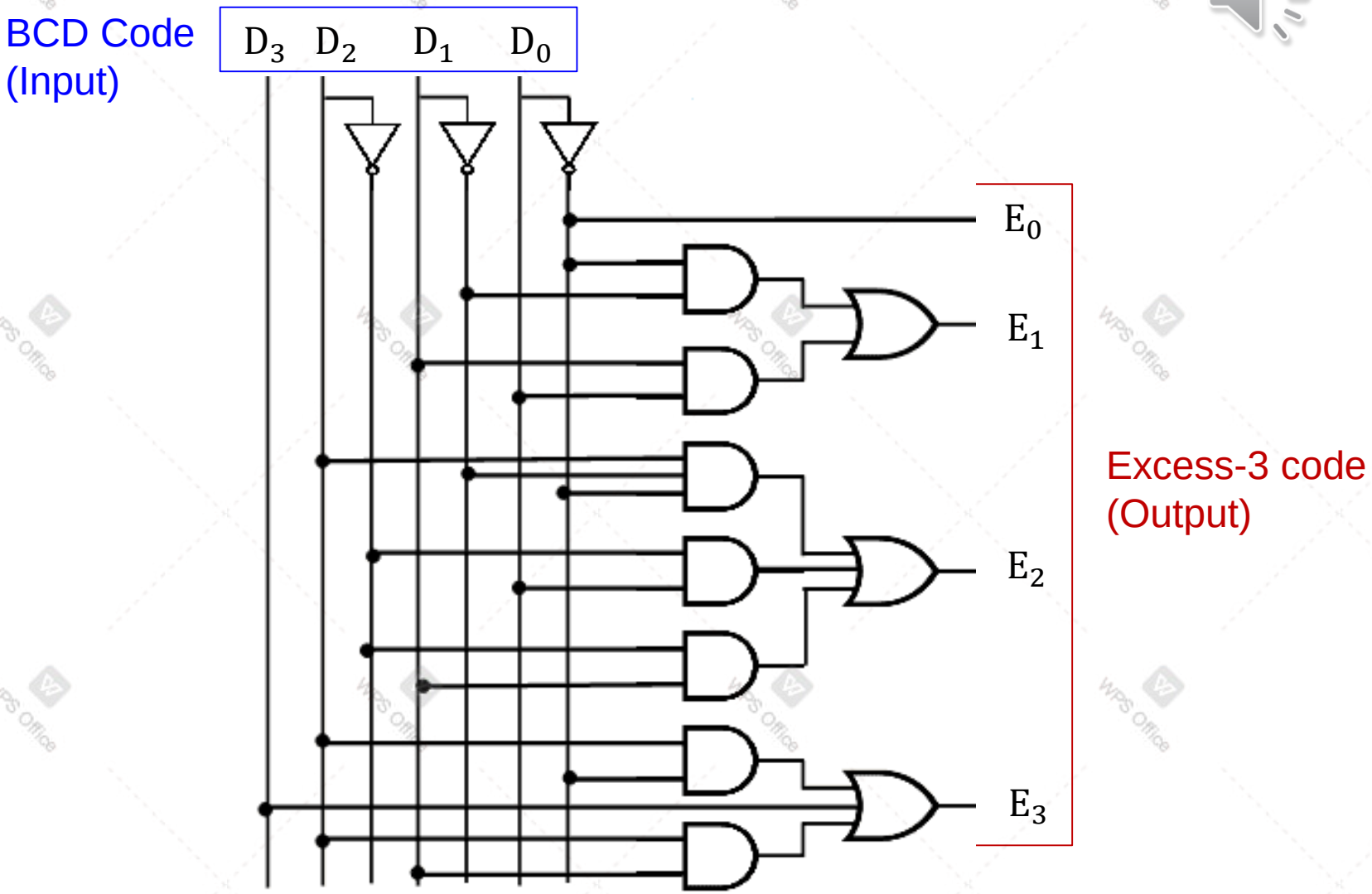
$$E_1 = \overline{D}_1\overline{D}_0 + D_1D_0$$

For $E_0$ output

| $D_3D_2$ \ $D_1D_0$ | 00 | 01 | 11 | 10 |
|---|---|---|---|---|
| 00 | 1 | 0 | 0 | 1 |
| 01 | 1 | 0 | 0 | 1 |
| 11 | X | X | X | X |
| 10 | 1 | 0 | X | X |

$$E_0 = \overline{D}_0$$

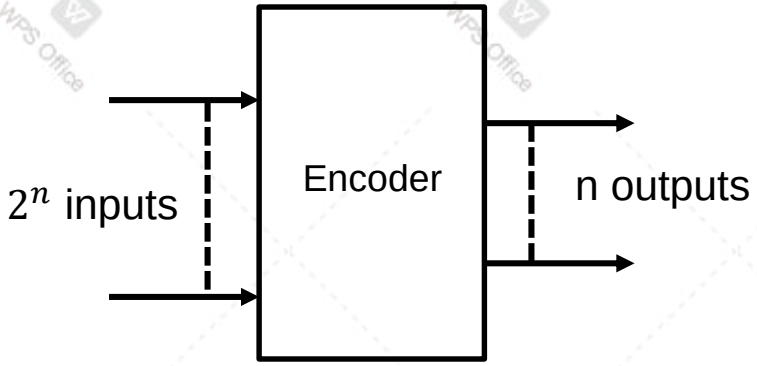
# BCD to Excess-3 Code Converter Circuit



# Encoder and Decoder Circuits

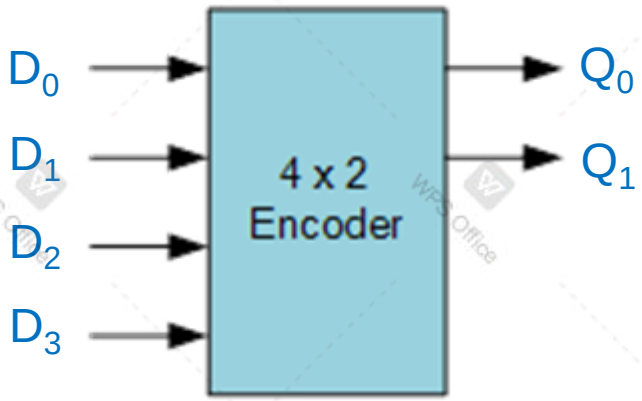
# Encoder Circuits

- Encoder circuit converts the applied information signal into a coded digital bit stream
- It takes $2^n$ inputs and generates $n$ outputs. Only one of inputs is activated at a time




- Common types of binary encoders include 4-to-2, 8-to-3 and 16-to-4 line configurations
- **Applications:** keyboard encoder, positional encoder, priority encoder etc.

# 4-to-2 Bit Encoder




| Inputs | | | | Outputs | |
|---|---|---|---|---|---|
| $D_3$ | $D_2$ | $D_1$ | $D_0$ | $Q_1$ | $Q_0$ |
| 0 | 0 | 0 | 1 | 0 | 0 |
| 0 | 0 | 1 | 0 | 0 | 1 |
| 0 | 1 | 0 | 0 | 1 | 0 |
| 1 | 0 | 0 | 0 | 1 | 1 |

$$Q_0 = D_1 + D_3$$

$$Q_1 = D_2 + D_3$$

# 8-to-3 Bit Encoder

| Inputs | | | | | | | | Outputs | | |
|---|---|---|---|---|---|---|---|---|---|---|
| $D_7$ | $D_6$ | $D_5$ | $D_4$ | $D_3$ | $D_2$ | $D_1$ | $D_0$ | $Q_2$ | $Q_1$ | $Q_0$ |
| 0 | 0 | 0 | 0 | 0 | 0 | 0 | 1 | 0 | 0 | 0 |
| 0 | 0 | 0 | 0 | 0 | 0 | 1 | 0 | 0 | 0 | 1 |
| 0 | 0 | 0 | 0 | 0 | 1 | 0 | 0 | 0 | 1 | 0 |
| 0 | 0 | 0 | 0 | 1 | 0 | 0 | 0 | 0 | 1 | 1 |
| 0 | 0 | 0 | 1 | 0 | 0 | 0 | 0 | 1 | 0 | 0 |
| 0 | 0 | 1 | 0 | 0 | 0 | 0 | 0 | 1 | 0 | 1 |
| 0 | 1 | 0 | 0 | 0 | 0 | 0 | 0 | 1 | 1 | 0 |
| 1 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 1 | 1 | 1 |

$$Q_0 = D_1 + D_3 + D_5 + D_7$$

$$Q_1 = D_2 + D_3 + D_6 + D_7$$

$$Q_2 = D_4 + D_5 + D_6 + D_7$$

- Binary encoders can’t handle the simultaneous presence of multiple inputs
- Solution lies in assigning priority to the inputs

# Priority Encoder

- The priority encoders output the code corresponds to the currently active input which has the highest priority
- Truth table of a priority encoder

| Inputs | | | | Outputs | | |
|---|---|---|---|---|---|---|
| $D_3$ | $D_2$ | $D_1$ | $D_0$ | $Q_1$ | $Q_0$ | V |
| 0 | 0 | 0 | 0 | x | x | 0 |
| 0 | 0 | 0 | 1 | 0 | 0 | 1 |
| 0 | 0 | 1 | x | 0 | 1 | 1 |
| 0 | 1 | x | x | 1 | 0 | 1 |
| 1 | x | x | x | 1 | 1 | 1 |

Validity output

- Note that the input $D_3$ has the highest priority; the output code for $D_0$ is generated only if higher priority inputs are zero

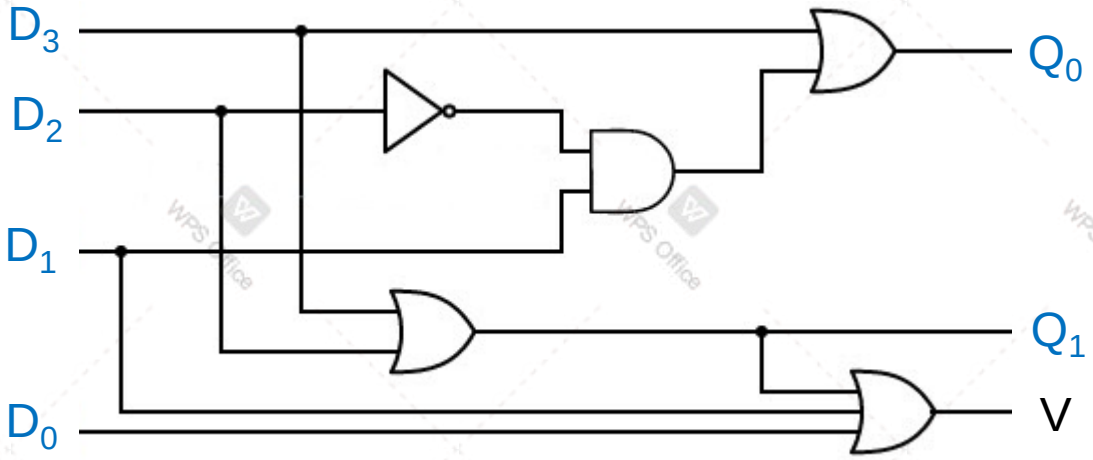
# Priority Encoder

- From the truth table, the expressions for the outputs can be written as

$$Q_1 = D_3 + \overline{D}_3 D_2 = D_3 + D_2$$

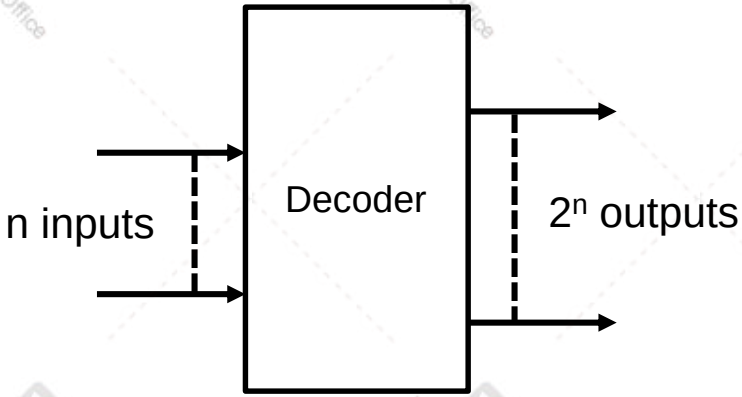
$$Q_0 = D_3 + \overline{D}_3 \overline{D}_2 D_1 = D_3 + \overline{D}_2 D_1$$

$$V = D_0 + D_1 + D_2 + D_3$$



# Decoder Circuits

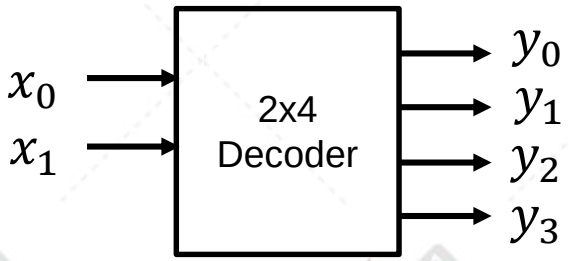
- Suppose we have n input bits which can represent up to $2^n$ distinct elements of coded information
- We need a device that allows us to select which of the $2^n$ elements, devices, memory locations, etc. is being selected
- In general:
  - A decoder has n input bits
  - A decoder has $2^n$ (or less) output bits
  - As a rule, all but one of the outputs is zero (deselected) at any time (called one-hot encoded)



# 2-to-4 Decoder

- The 2-to-4 decoder is a block which decodes the 2-bit binary inputs and produces four outputs
- One output corresponding to the input combination is a one
- Two inputs and four outputs are shown in the figure
- The output equations are:

$$y_0 = \bar{x}_1 \cdot \bar{x}_0$$
$$y_1 = \bar{x}_1 \cdot x_0$$
$$y_2 = x_1 \cdot \bar{x}_0$$
$$y_3 = x_1 \cdot x_0$$

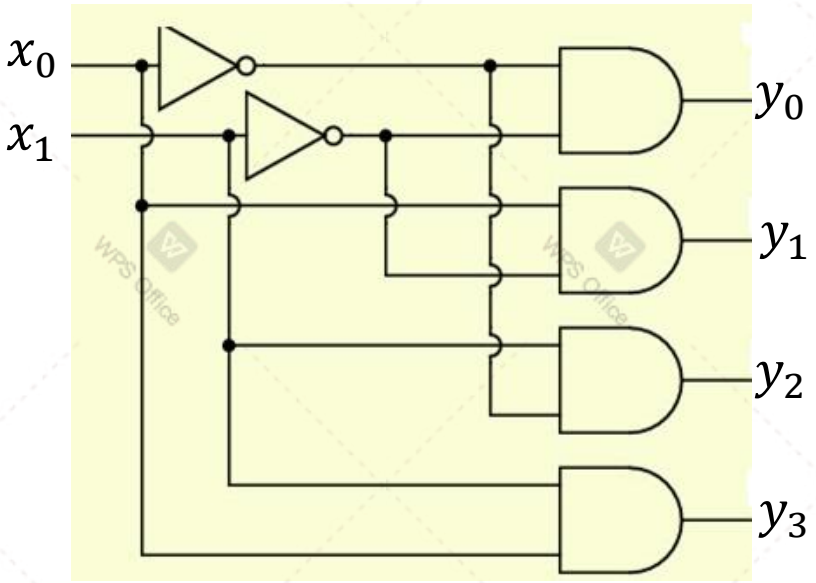


# 2-to-4 Decoder

- Truth table

| $x_1$ | $x_0$ | $y_3$ | $y_2$ | $y_1$ | $y_0$ |
|---|---|---|---|---|---|
| 0 | 0 | 0 | 0 | 0 | 1 |
| 0 | 1 | 0 | 0 | 1 | 0 |
| 1 | 0 | 0 | 1 | 0 | 0 |
| 1 | 1 | 1 | 0 | 0 | 0 |

Logic diagram



# 2-to-4 Decoder with Enable

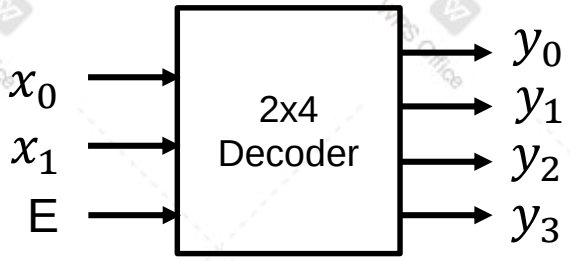
- A 2-to-4 decoder can be designed with an enable signal
- If enable is 0, all outputs are zero
- If enable is 1, then an output corresponding to two inputs is a one, all others are still zero
- The output equations are:

$$y_0 = \bar{x}_1 \cdot \bar{x}_0 \cdot E$$

$$y_1 = \bar{x}_1 \cdot x_0 \cdot E$$

$$y_2 = x_1 \cdot \bar{x}_0 \cdot E$$

$$y_3 = x_1 \cdot x_0 \cdot E$$

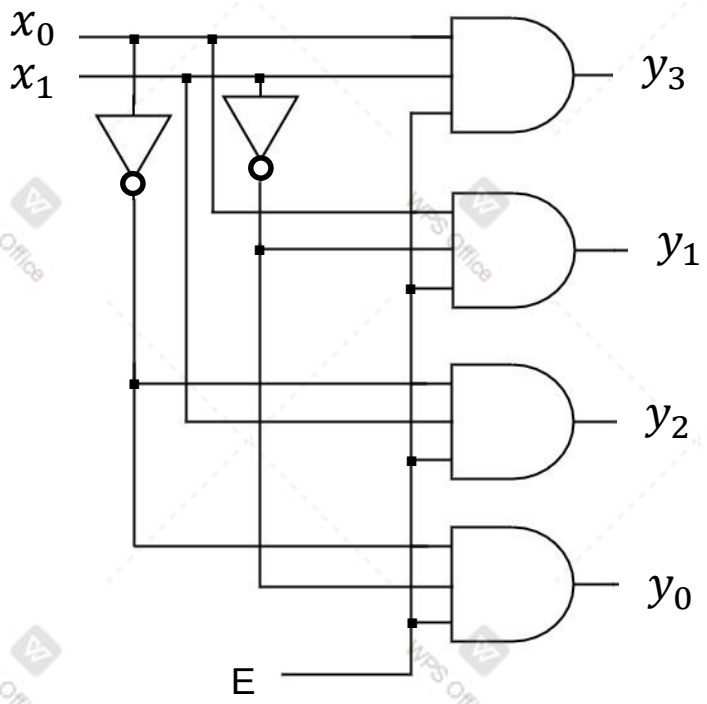


# 2-to-4 Decoder with Enable

Truth table

| $x_1$ | $x_0$ | E | $y_3$ | $y_2$ | $y_1$ | $y_0$ |
|---|---|---|---|---|---|---|
| 0 | 0 | 0 | 0 | 0 | 0 | 0 |
| 0 | 0 | 1 | 0 | 0 | 0 | 1 |
| 0 | 1 | 0 | 0 | 0 | 0 | 0 |
| 0 | 1 | 1 | 0 | 0 | 1 | 0 |
| 1 | 0 | 0 | 0 | 0 | 0 | 0 |
| 1 | 0 | 1 | 0 | 1 | 0 | 0 |
| 1 | 1 | 0 | 0 | 0 | 0 | 0 |
| 1 | 1 | 1 | 1 | 0 | 0 | 0 |

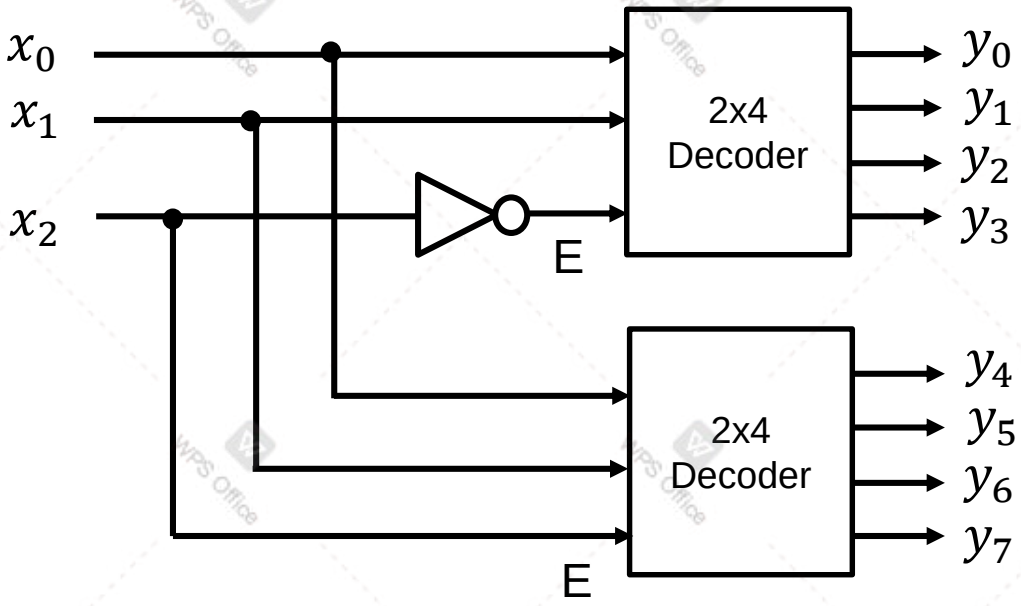
Logic diagram



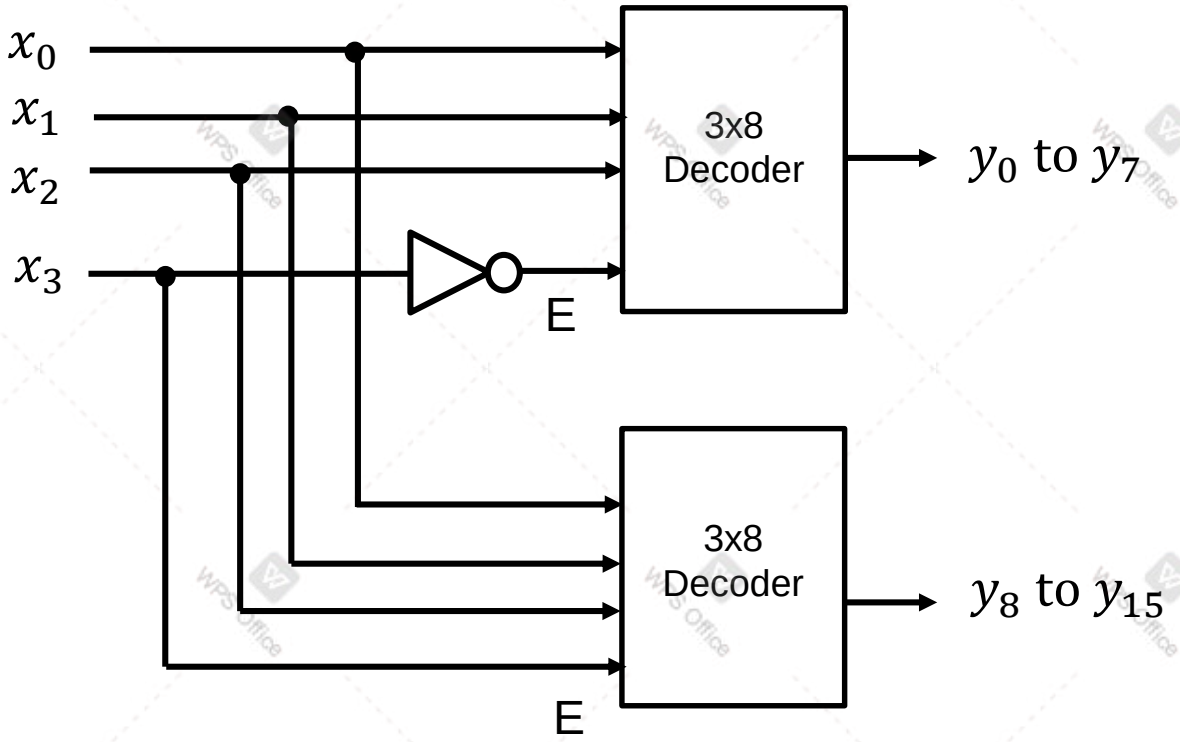

Enable input

# 3-to-8 Decoder using 2-to-4 Decoder

- The 3-to-8 decoder can be implemented using two 2-to-4 decoder with enable and one NOT gate



# 4-to-16 Decoder using 3-to-8 Decoder



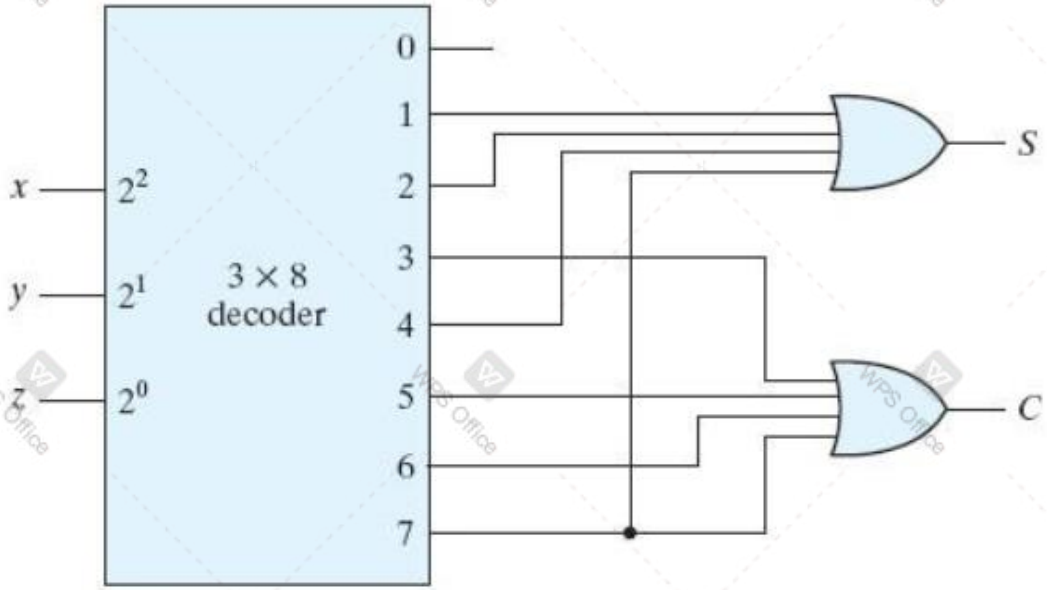
# Combinational Logic Implementation using Decoder

- A full adder implemented using 3x8 decoder

| $A$ | $B$ | $C_i$ | $S$ | $C_o$ |
|---|---|---|---|---|
| 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 |

$$S(x, y, z) = \Sigma m(1,2,4,7)$$

$$C(x, y, z) = \Sigma m(3,5,6,7)$$

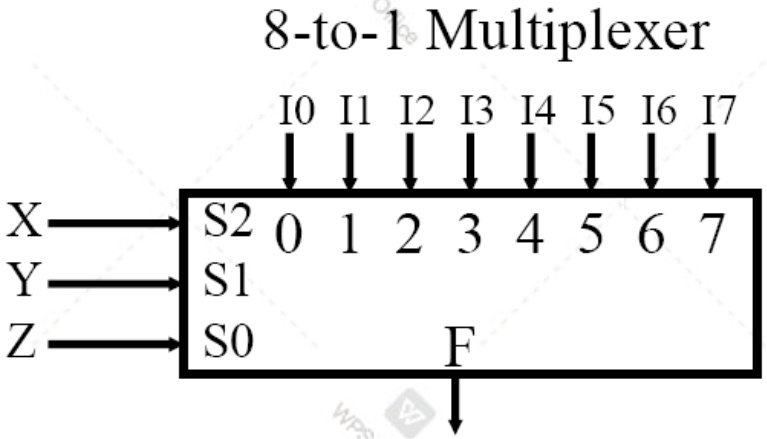


# Multiplexer and Demultiplexer Circuits

# Multiplexer

- Multiplexers (MUX) are circuits which select one of many inputs
- In here, we assume that we have one-bit inputs
  (in general, each input may have more than one bit)
- Suppose we have eight inputs: I0, I1, I2, I3, I4, I5, I6, I7
- We want one of them to be output based on selection signals
- 3 bits of selection signals to decide which input goes to output.

# Multiplexer




| X | Y | Z | F |
|---|---|---|---|
| 0 | 0 | 0 | I0 |
| 0 | 0 | 1 | I1 |
| 0 | 1 | 0 | I2 |
| 0 | 1 | 1 | I3 |
| 1 | 0 | 0 | I4 |
| 1 | 0 | 1 | I5 |
| 1 | 1 | 0 | I6 |
| 1 | 1 | 1 | I7 |

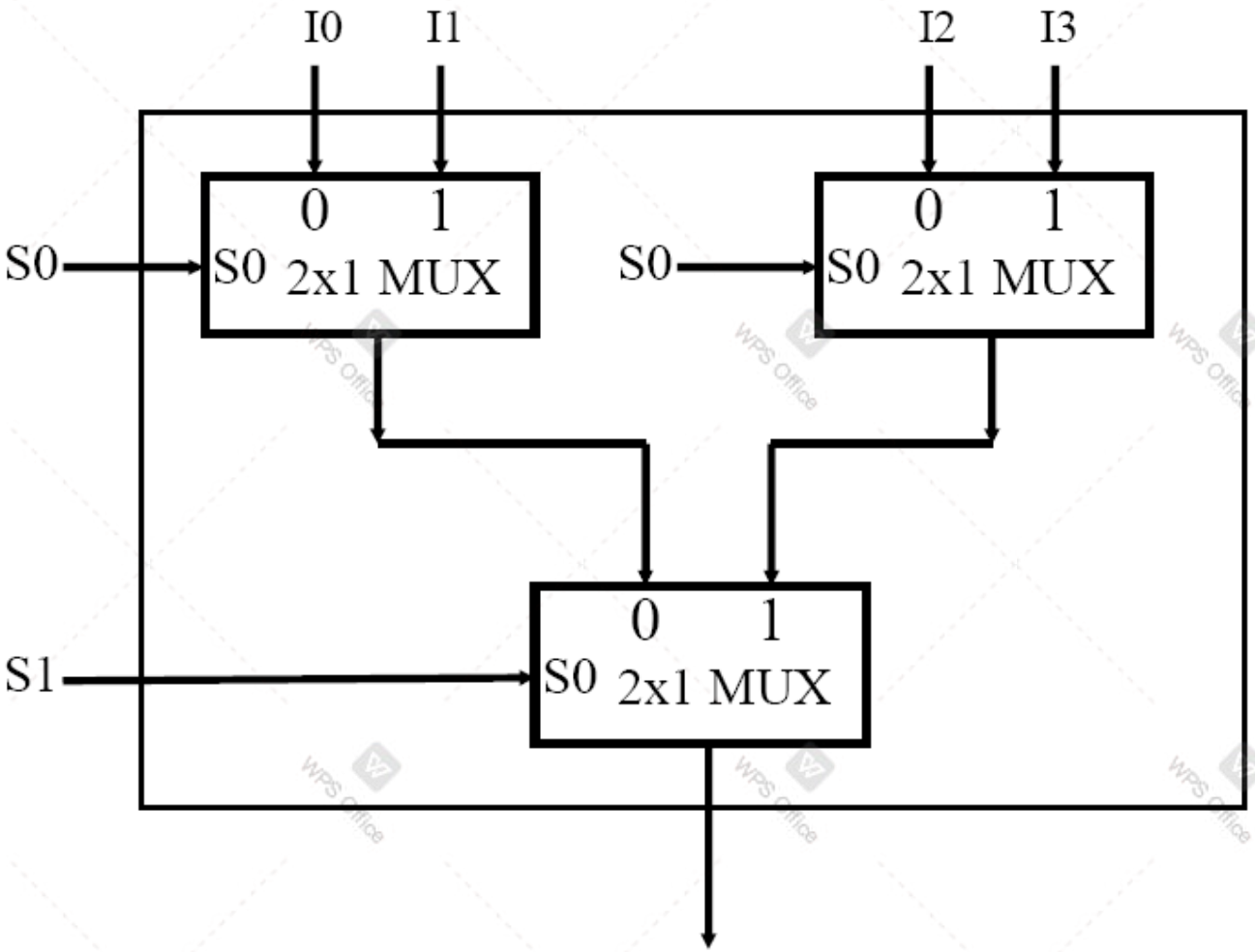
- Note the order of selection signals
  - X is MSB and Z is LSB

- We can write a logic expression for output F as follows

$$F = X' Y' Z' I0 + X' Y' Z I1 + X' Y Z' I2 + X' Y Z I3 + X Y' Z' I4 + X Y' Z I5 + X Y Z' I6 + X Y Z I7$$

- This circuit can be implemented using
  - eight 4-input AND gates and one 8-input OR gates

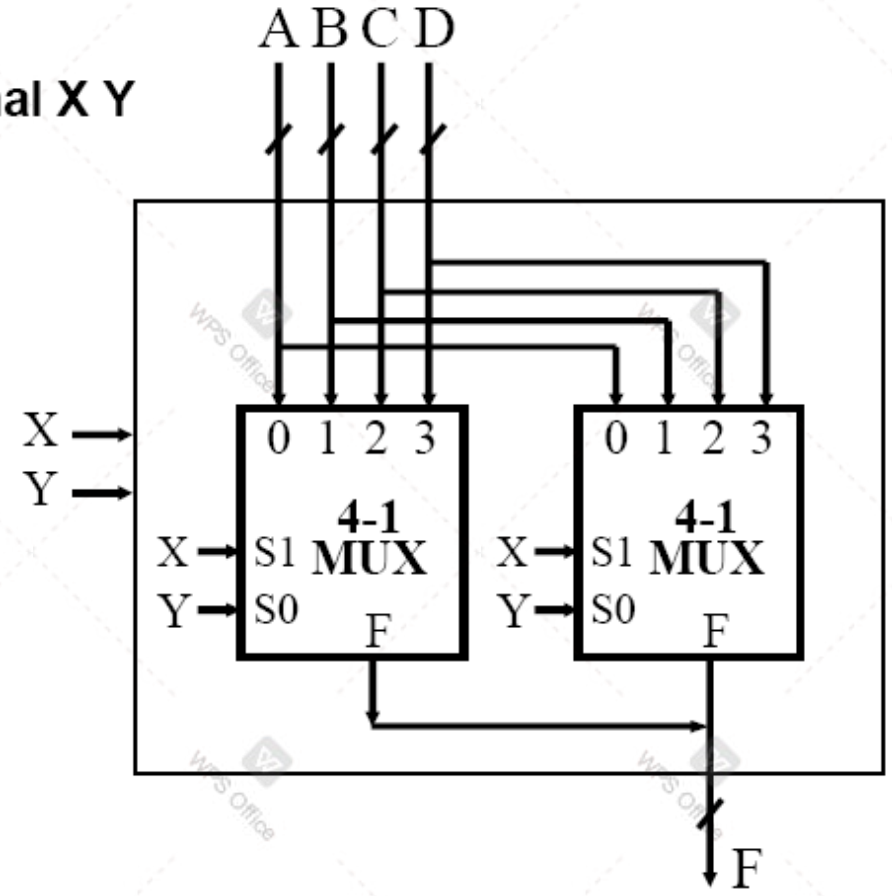
# Implementing 4-to-1 MUX using 2-to-1 MUX



# Making 2-bit 4-to-1 Multiplexer

- Four *2-bit* inputs A, B, C, D
- One *2-bit* output F
- Two bits of selection signal X Y

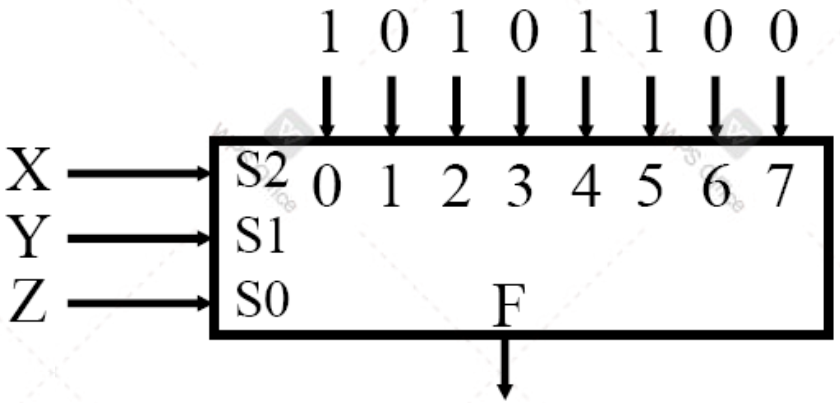
| X | Y | F |
|---|---|---|
| 0 | 0 | A |
| 0 | 1 | B |
| 1 | 0 | C |
| 1 | 1 | D |




Note: Left and right MUXs are processing one bit each of the selected 2-bit input and those are then combined to produce 2-bit form of the selected input

# Synthesis of Logic Functions using MUX

- Easiest way is to use function inputs as selection signals
- Input to multiplexer is a set of 1s and 0s depending on the function to be implemented
- To implement logic function F specified in the truth table
  - Three select signals are X, Y, and Z, and output is F
  - Eight inputs to multiplexer are 1 0 1 0 1 1 0 0
  - Depending on the input signals
  - multiplexer will select proper output

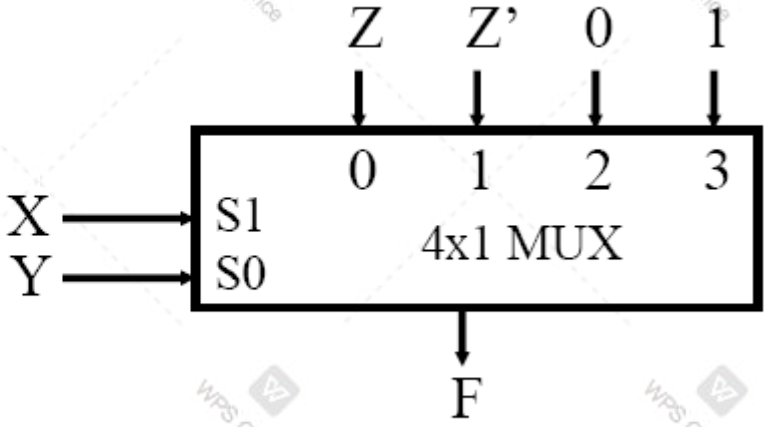



| X | Y | Z | F |
|---|---|---|---|
| 0 | 0 | 0 | 1 |
| 0 | 0 | 1 | 0 |
| 0 | 1 | 0 | 1 |
| 0 | 1 | 1 | 0 |
| 1 | 0 | 0 | 1 |
| 1 | 0 | 1 | 1 |
| 1 | 1 | 0 | 0 |
| 1 | 1 | 1 | 0 |

# Implementing 3-variable functions with 4x1 MUX

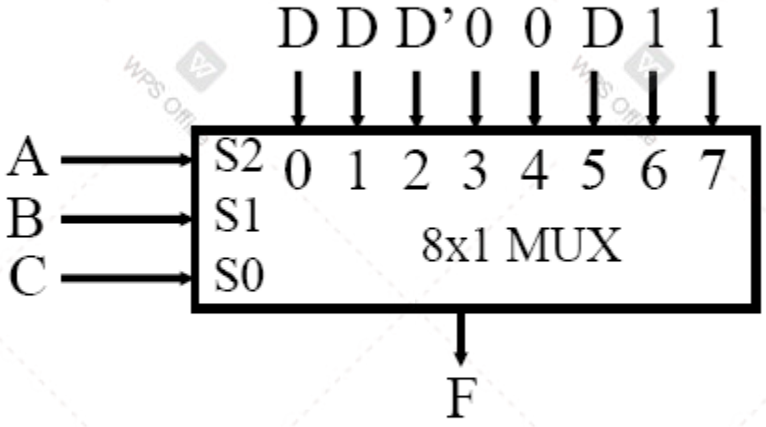
- Divide the outputs into 4 groups based on X and Y.
- Write the outputs as a function of Z
- There are only 4 possibilities: F=Z, F=Z', F=0, F=1

| X | Y | Z | F | |
|---|---|---|---|---|
| 0 | 0 | 0 | 0 | F=Z |
| 0 | 0 | 1 | 1 | |
| 0 | 1 | 0 | 1 | F=Z' |
| 0 | 1 | 1 | 0 | |
| 1 | 0 | 0 | 0 | F=0 |
| 1 | 0 | 1 | 0 | |
| 1 | 1 | 0 | 1 | F=1 |
| 1 | 1 | 1 | 1 | |



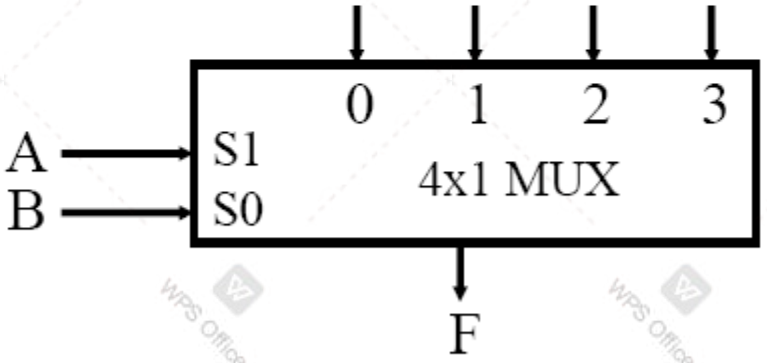
# Implementing 4-variable functions with 8x1 MUX

| A | B | C | D | F | |
|---|---|---|---|---|---|
| 0 | 0 | 0 | 0 | 0 | F=D |
| 0 | 0 | 0 | 1 | 1 | |
| 0 | 0 | 1 | 0 | 0 | F=D |
| 0 | 0 | 1 | 1 | 1 | |
| 0 | 1 | 0 | 0 | 1 | F=D’ |
| 0 | 1 | 0 | 1 | 0 | |
| 0 | 1 | 1 | 0 | 0 | F=0 |
| 0 | 1 | 1 | 1 | 0 | |
| 1 | 0 | 0 | 0 | 0 | F=0 |
| 1 | 0 | 0 | 1 | 0 | |
| 1 | 0 | 1 | 0 | 0 | F=D |
| 1 | 0 | 1 | 1 | 1 | |
| 1 | 1 | 0 | 0 | 1 | F=1 |
| 1 | 1 | 0 | 1 | 1 | |
| 1 | 1 | 1 | 0 | 1 | F=1 |
| 1 | 1 | 1 | 1 | 1 | |



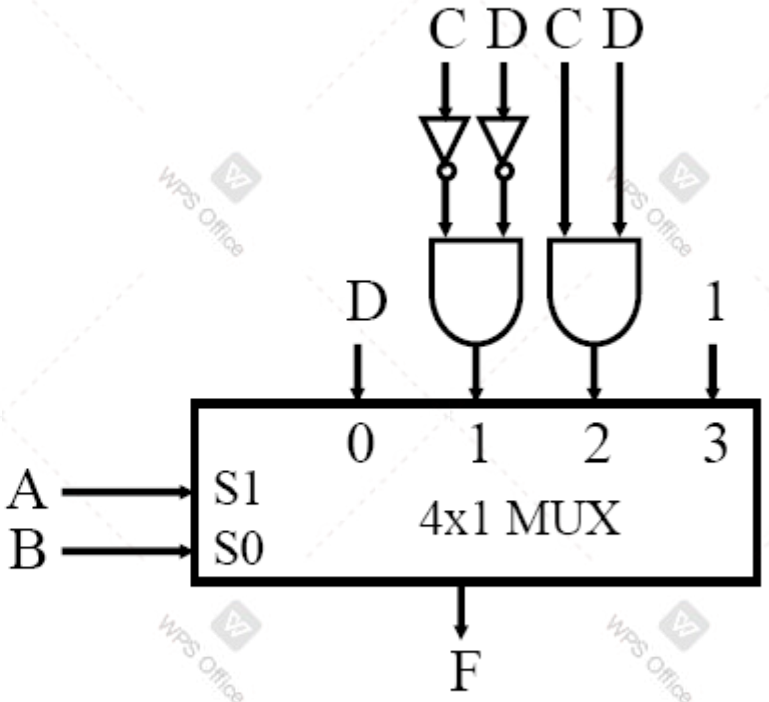
# Implementing 4-variable functions with 4x1 MUX

| A | B | C | D | F | |
|---|---|---|---|---|---|
| 0 | 0 | 0 | 0 | 0 | |
| 0 | 0 | 0 | 1 | 1 | |
| 0 | 0 | 1 | 0 | 0 | F= ? |
| 0 | 0 | 1 | 1 | 1 | |
| 0 | 1 | 0 | 0 | 1 | |
| 0 | 1 | 0 | 1 | 0 | |
| 0 | 1 | 1 | 0 | 0 | F= ? |
| 0 | 1 | 1 | 1 | 0 | |
| 1 | 0 | 0 | 0 | 0 | |
| 1 | 0 | 0 | 1 | 0 | |
| 1 | 0 | 1 | 0 | 0 | F= ? |
| 1 | 0 | 1 | 1 | 1 | |
| 1 | 1 | 0 | 0 | 1 | |
| 1 | 1 | 0 | 1 | 1 | |
| 1 | 1 | 1 | 0 | 1 | F= ? |
| 1 | 1 | 1 | 1 | 1 | |



# Implementing 4-variable functions with 4x1 MUX

| A | B | C | D | F | |
|---|---|---|---|---|---|
| 0 | 0 | 0 | 0 | 0 | |
| 0 | 0 | 0 | 1 | 1 | F=D |
| 0 | 0 | 1 | 0 | 0 | |
| 0 | 0 | 1 | 1 | 1 | |
| 0 | 1 | 0 | 0 | 1 | |
| 0 | 1 | 0 | 1 | 0 | F=C'D' |
| 0 | 1 | 1 | 0 | 0 | |
| 0 | 1 | 1 | 1 | 0 | |
| 1 | 0 | 0 | 0 | 0 | |
| 1 | 0 | 0 | 1 | 0 | F=CD |
| 1 | 0 | 1 | 0 | 0 | |
| 1 | 0 | 1 | 1 | 1 | |
| 1 | 1 | 0 | 0 | 1 | |
| 1 | 1 | 0 | 1 | 1 | F=1 |
| 1 | 1 | 1 | 0 | 1 | |
| 1 | 1 | 1 | 1 | 1 | |

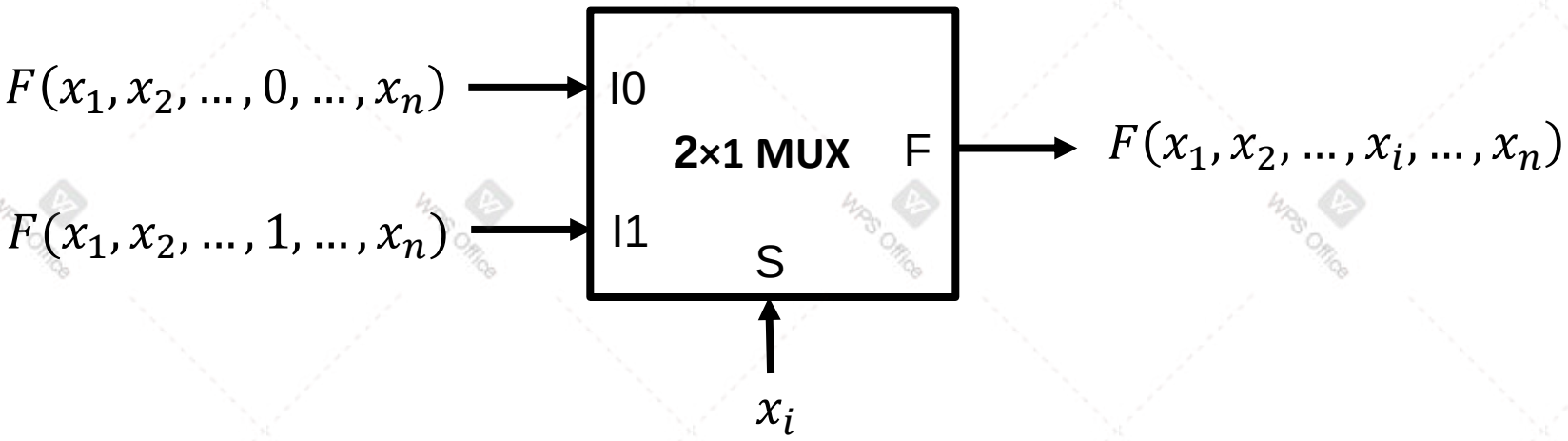


# Shannon's Expansion Theorem based MUX Implementation

- Recall Shannon's expansion theorem

$$F(x_1, x_2, \ldots, x_i, \ldots, x_n) = \bar{x}_i \cdot F(x_1, x_2, \ldots, 0, \ldots, x_n) + x_i \cdot F(x_1, x_2, \ldots, 1, \ldots, x_n)$$

- In 2×1 MUX, the o/p is expressed as $\mathrm{F} = \bar{\mathrm{S}} \cdot \mathrm{I0} + \mathrm{S} \cdot \mathrm{I1}$



# Example: Use of Shannon's Expansion

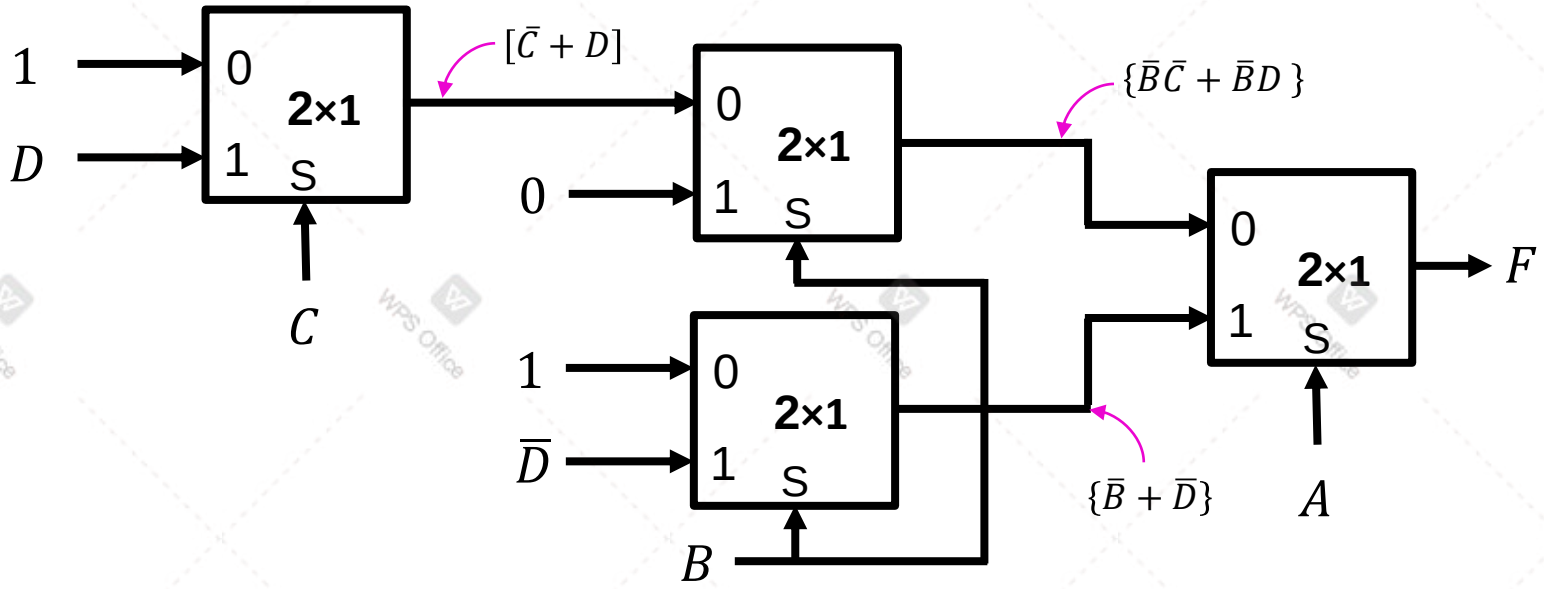
- Realize $F(A,B,C,D) = A\bar{B} + \bar{B}\bar{C} + A\bar{D} + \bar{B}D$ using MUXs only

$$F = \bar{A} \cdot \{\bar{B}\bar{C} + \bar{B}D\} + A \cdot \{\bar{B} + \bar{B}\bar{C} + \bar{D} + \bar{B}D\}$$

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

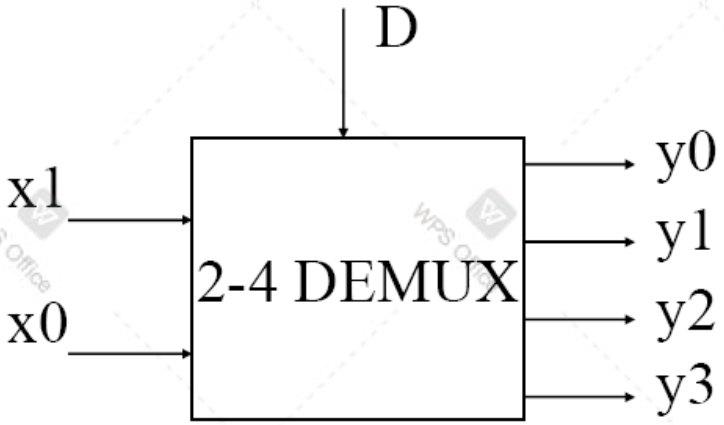
$$= \bar{A} \cdot \{\bar{B} \cdot [\bar{C} + D] + B \cdot [0]\} + A \cdot \{\bar{B} \cdot [1] + B \cdot [\bar{D}]\}$$

$$= \bar{A} \cdot \{\bar{B} \cdot [\bar{C} \cdot 1 + C \cdot D] + B \cdot [0]\} + A \cdot \{\bar{B} \cdot [1] + B \cdot [\bar{D}]\}$$

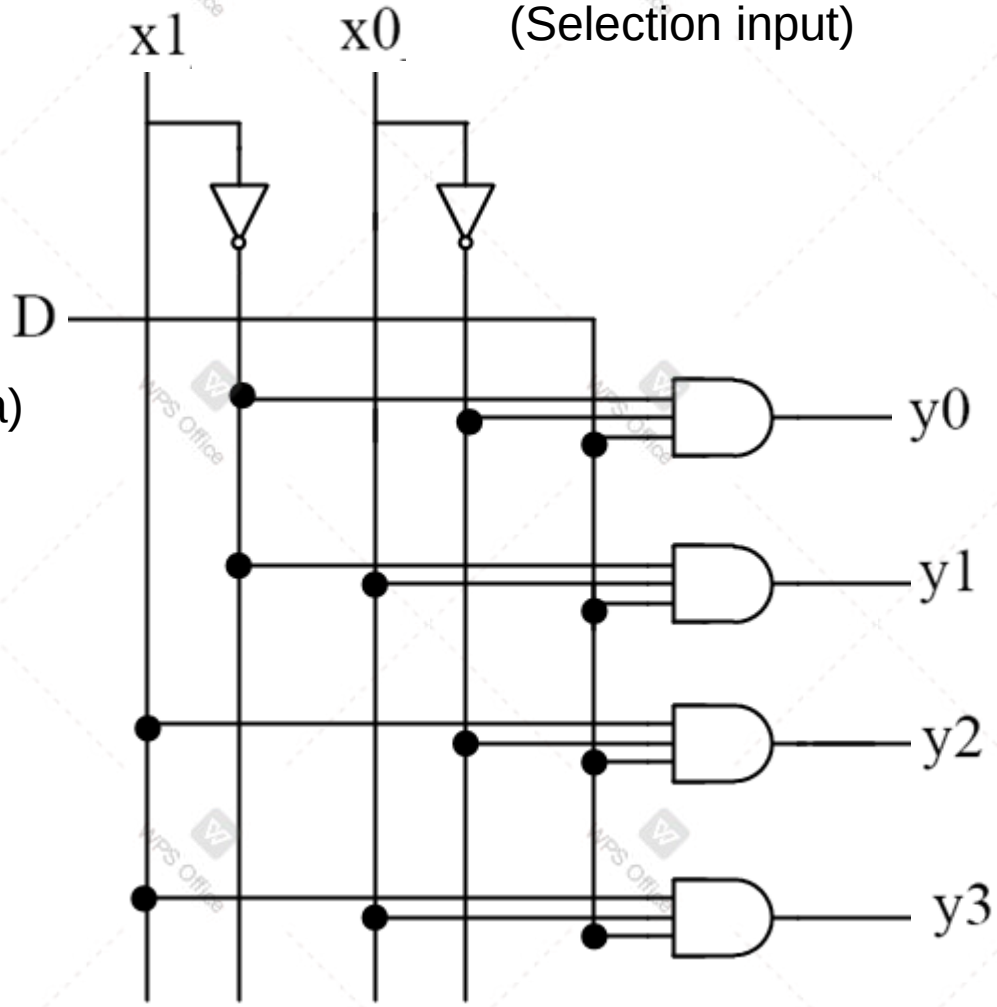


# Demultiplexer

- Perform the opposite function of multiplexers
- Placing the value of a single data input onto one of the multiple data outputs
- Similar implementation as decoder with enable
- Enable input of decoder serves as the data input for the demultiplexer



# 2-to-4 Demultiplexer Realization



# Sequential Logic

Latches and Flip-flops

# Sequential Logic

- The logic circuits discussed previously are known as combinational, in that the output depends only on the condition of the latest inputs

- In contrast, there exist logic circuits where the output depends not only on the latest inputs, but also on the condition of earlier inputs

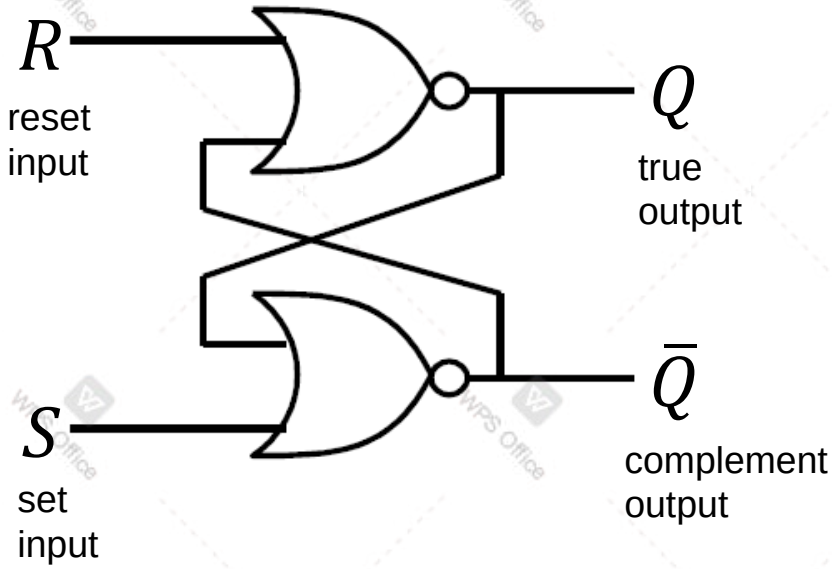
- These circuits are known as *sequential logic*, and implicitly they contain memory elements

# Memory Element

- A memory element stores data – usually one bit per element
- A snapshot of the memory is called the **state**
- A one bit memory is often called a **bistable**, i.e., it has 2 stable internal states (0 and 1)
- **Latches** and **flip-flops** are particular implementations of bistables

# SR Latch

- An SR (“set-reset”) latch is a memory element comprising of two inputs: set ($S$) and reset ($R$), and two outputs: $Q$ and $\bar{Q}$

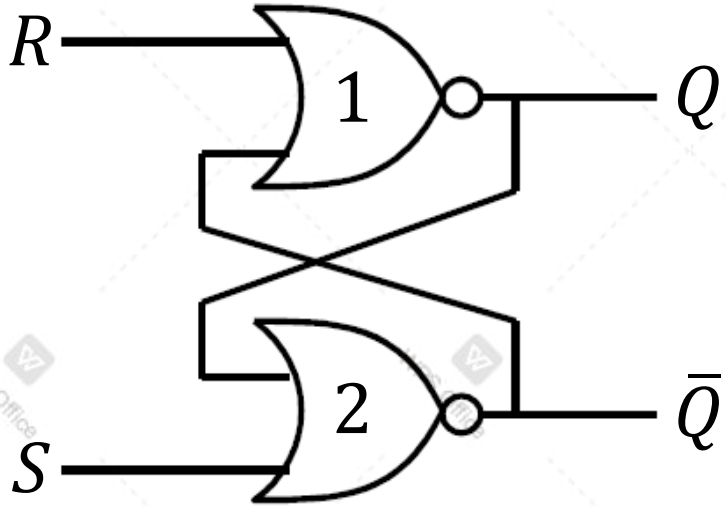



Characteristic Table

| $S$ $R$ | $Q'$ $\bar{Q}'$ | comment |
|---|---|---|
| 0 0 | $Q$ $\bar{Q}$ | hold |
| 0 1 | 0 1 | reset |
| 1 0 | 1 0 | set |
| 1 1 | 0 0 | illegal |

Where $Q'$ is the next state and $Q$ is the current state

# SR Latch - Operation

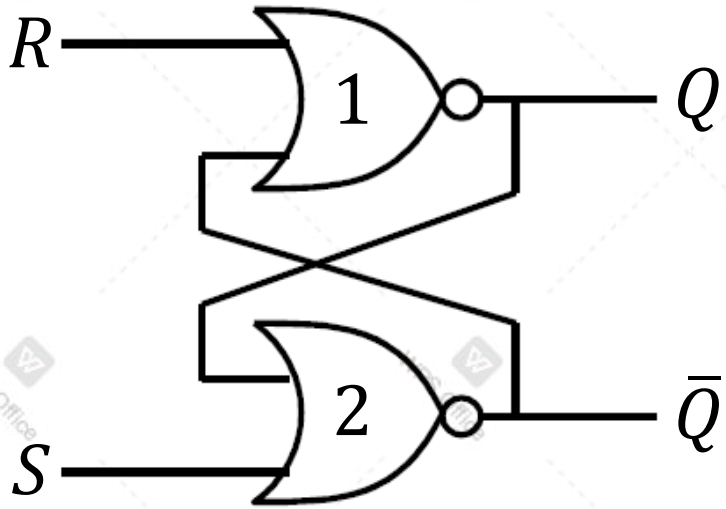



NOR truth table

| a | b | y | |
|---|---|---|---|
| 0 | 0 | 1 | *b* complemented |
| 0 | 1 | 0 | |
| 1 | 0 | 0 | always 0 |
| 1 | 1 | 0 | |

- $R = 1$ and $S = 0$
  - Gate 1 output in ‘always 0’ condition, $Q = 0$
  - Gate 2 in ‘complement’ condition, so $\bar{Q} = 1$
  - This is the (R)eset condition

# SR Latch - Operation

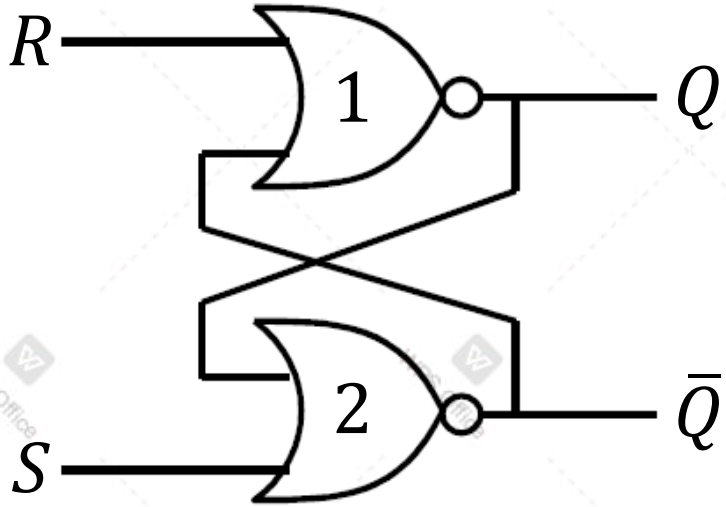



NOR truth table

| a | b | y | |
|---|---|---|---|
| 0 | 0 | 1 | *b* complemented |
| 0 | 1 | 0 | |
| 1 | 0 | 0 | always 0 |
| 1 | 1 | 0 | |

- *S* = 0 and *R* to 0
  - Gate 2 remains in ‘complement’ condition, $\bar{Q} = 1$
  - Gate 1 into ‘complement’ condition, $Q = 0$
  - This is the hold condition

# SR Latch - Operation

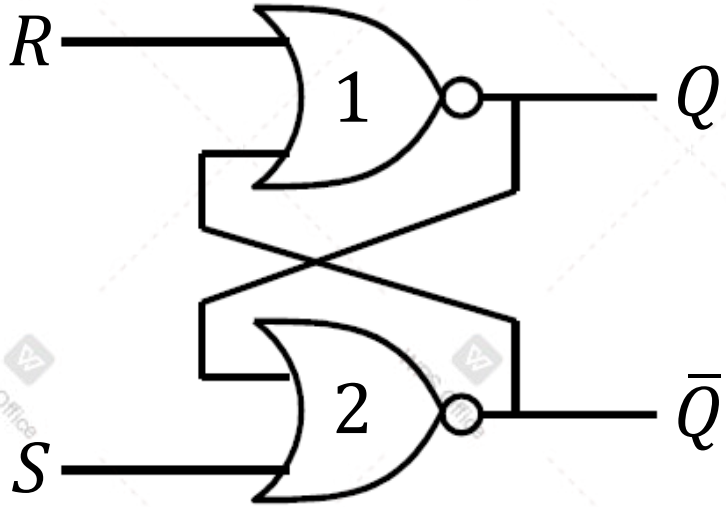



NOR truth table

| a | b | y | |
|---|---|---|---|
| 0 | 0 | 1 | *b* complemented |
| 0 | 1 | 0 | |
| 1 | 0 | 0 | always 0 |
| 1 | 1 | 0 | |

- *S* = 1 and *R* = 0
  - Gate 1 into ‘complement’ condition, $Q = 1$
  - Gate 2 in ‘always 0’ condition, $\bar{Q} = 0$
  - This is the (S)et condition

# SR Latch - Operation




NOR truth table

| a | b | y | |
|---|---|---|---|
| 0 | 0 | 1 | *b* complemented |
| 0 | 1 | 0 | |
| 1 | 0 | 0 | always 0 |
| 1 | 1 | 0 | |

- $S = 1$ and $R = 1$
  - Gate 1 in ‘always 0’ condition, $Q = 0$
  - Gate 2 in ‘always 0’ condition, $\bar{Q} = 0$
  - This is the illegal condition

# SR Latch – State Transition Table

- A *state transition table* is an alternative way of viewing its operation

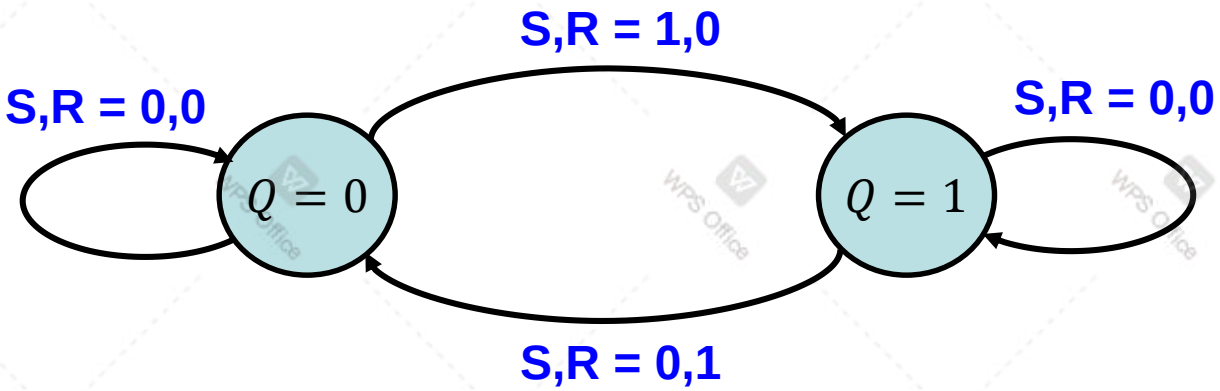
| $Q\ S\ R$ | $Q'$ | comment |
|---|---|---|
| 0 0 0 | 0 | hold |
| 0 0 1 | 0 | reset |
| 0 1 0 | 1 | set |
| 0 1 1 | 0 | illegal |
| 1 0 0 | 1 | hold |
| 1 0 1 | 0 | reset |
| 1 1 0 | 1 | set |
| 1 1 1 | 0 | illegal |

- A state transition table can also be expressed in the form of a *state diagram*

# SR Latch – State Diagram

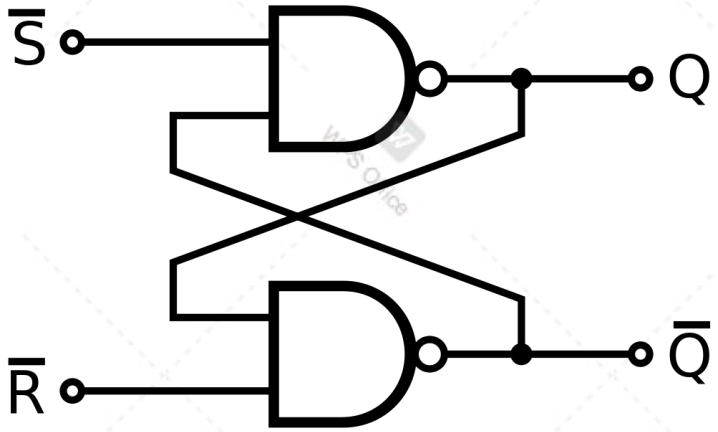
- The *state diagram* shows the input conditions required to transition between states. There are 4 possible transitions

- In this case, the state diagram has 2 states, i.e., $Q = 0$ and $Q = 1$



# SR Latch - NAND version

- We can also use NAND gates to make an SR latch

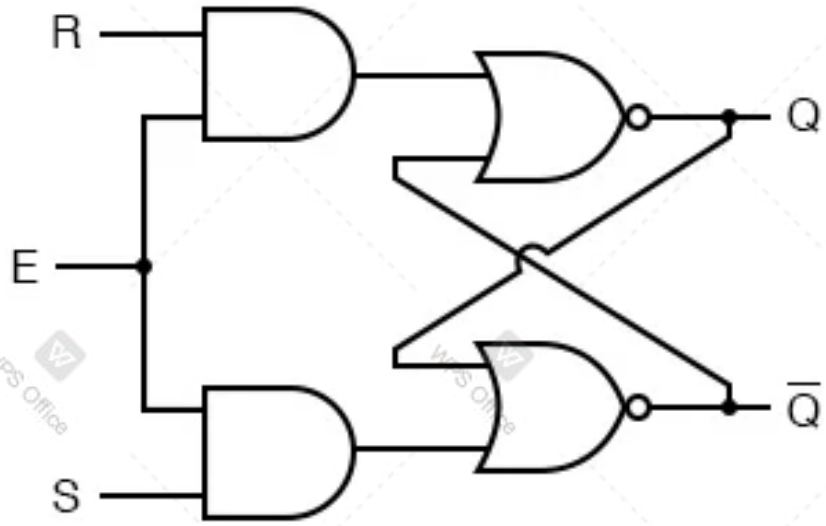



| $S\ R$ | $Q'\ \bar{Q}'$ | comment |
|---|---|---|
| 0 0 | $Q\ \ \bar{Q}$ | illegal |
| 0 1 | 0 1 | reset |
| 1 0 | 1 0 | set |
| 1 1 | 0 0 | hold |

- Note that the S and R inputs need to be inverted before they are applied to this circuit to make its behavior same as that of the NOR gates based SR latch

# Gated SR Latch

- Sometimes, it is required that the SR latch changes state only when certain conditions are met, regardless of its inputs
- The conditional input is called the enable (E)




| $E\ S\ R$ | $Q'\ \bar{Q}'$ | comment |
|---|---|---|
| 0 0 0 | $Q\ \ \bar{Q}$ | hold |
| 0 0 1 | $Q\ \ \bar{Q}$ | hold |
| 0 1 0 | $Q\ \ \bar{Q}$ | hold |
| 0 1 1 | $Q\ \ \bar{Q}$ | hold |
| 1 0 0 | $Q\ \ \bar{Q}$ | hold |
| 1 0 1 | 0 1 | reset |
| 1 1 0 | 1 0 | set |
| 1 1 1 | 0 0 | illegal |

# Clocks and Synchronous Circuits

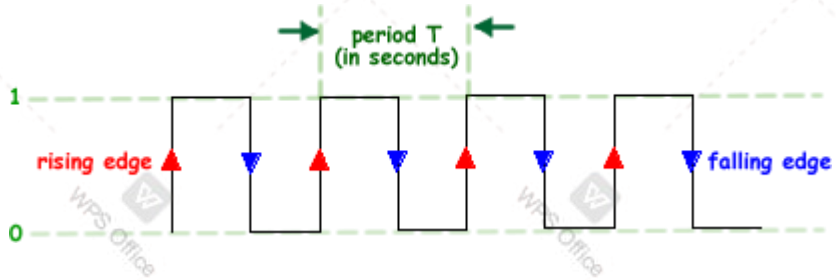
- For the SR latch, we can see that the output state changes occur directly in response to changes in the inputs. This is called *asynchronous* operation
- However, virtually all sequential circuits currently employ the notion of *synchronous* operation, that is, the output of a sequential circuit is constrained to change only at a time specified by a global *enabling* signal. This signal is generally known as the system *clock*

# Clocks and Synchronous Circuits

- What is a clock and what is it for?
  - Typically it is a square wave signal at a particular frequency
  - It imposes order on the state changes
  - Allows lots of states to appear to update simultaneously
- How can we modify an asynchronous circuit to act synchronously, i.e., in synchronism with a clock signal?

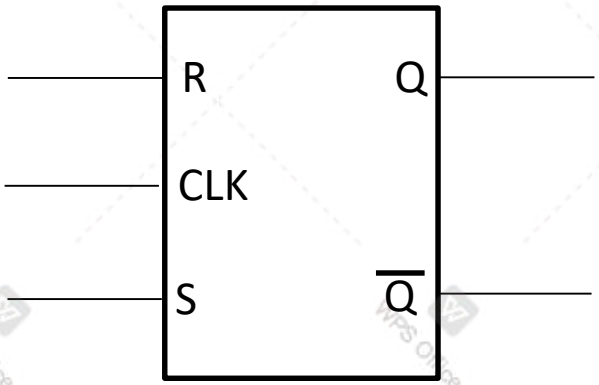
# Latches and Flip-Flops?

- A latch is asynchronous in nature – its state can change whenever the inputs are applied
- It is often more simple to design sequential circuits if the outputs change only on the either rising (positive going) or falling (negative going) '*edges*' of the clock signal



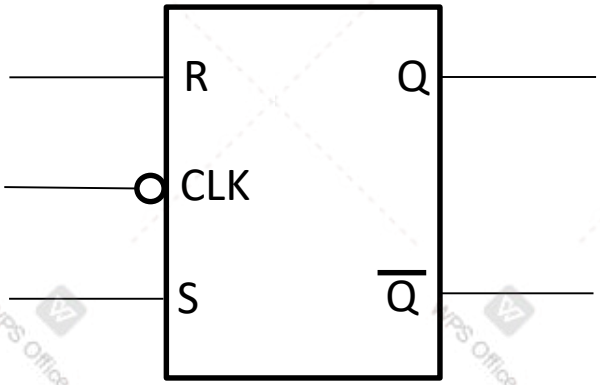

- A latch with clock functionality is called a flip-flop

# SR Flip-Flops




Positive Edge-Triggered

Active clock edge is the **Rising Edge**




Negative Edge-Triggered

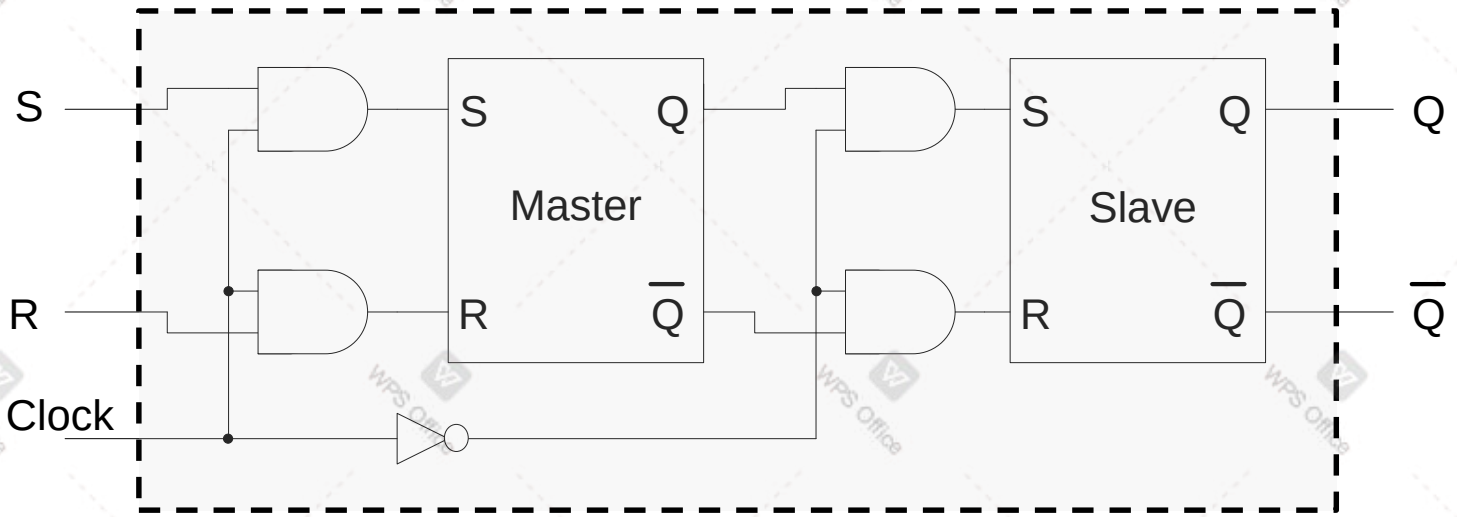
Active clock edge is the **Falling Edge**

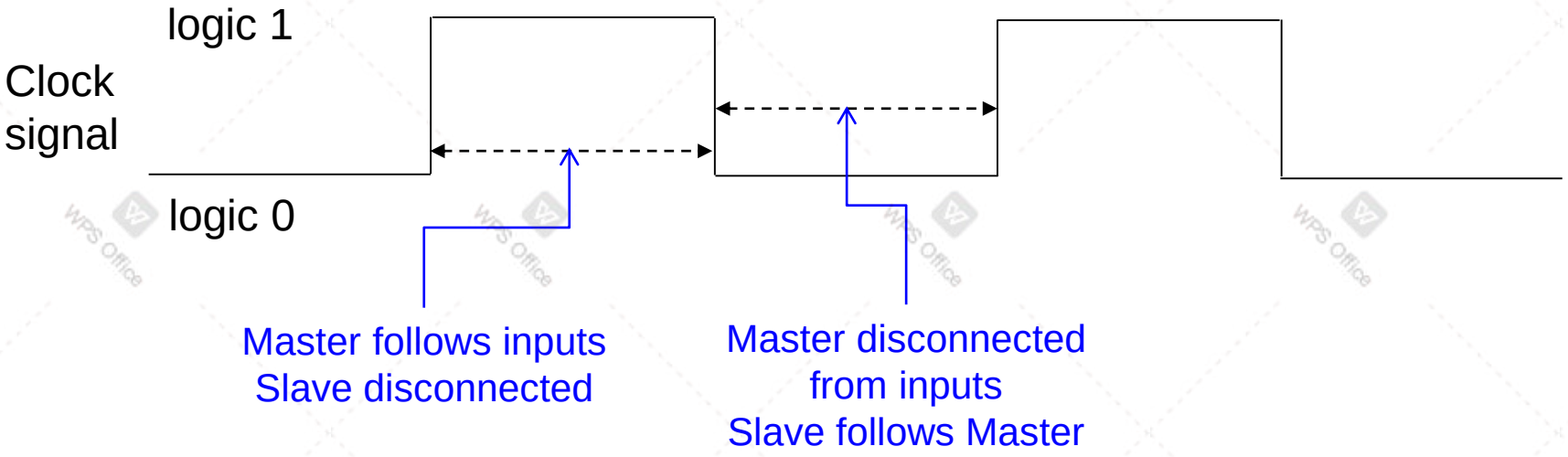
*Note the symbol* o *indicating negative logic applied to clock (which refers to negative edge triggering).*

# Master-Slave Flip-Flops

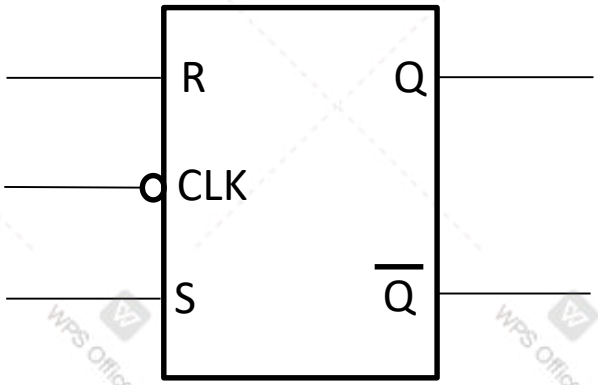
- A third type of triggering is also possible – this is called Master-Slave configuration
- A Master-Slave flip-flop is built with two latches – a Master latch followed by a Slave latch
- The Master Latch responds directly to the inputs while the CLK is HIGH – during this time the Slave is disconnected from the Master and stays in the same state
- When CLK is LOW, the Master Latch is disconnected from the inputs and its output drives the Slave Latch. During this time, the Master will not change state.

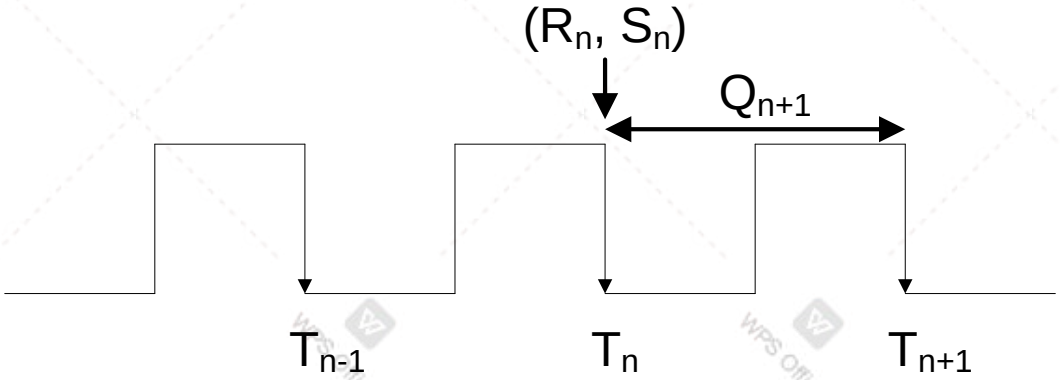
# Master-Slave SR Flip-Flop



# Operation of a SR Flip-Flop

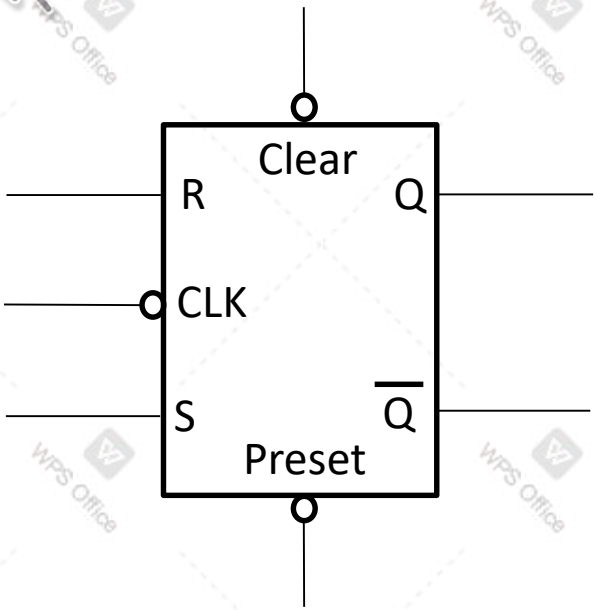



| $R_n$ | $S_n$ | $Q_{n+1}$ |
|---|---|---|
| 0 | 0 | $Q_n$ |
| 0 | 1 | 1 |
| 1 | 0 | 0 |

# SR Flip-flop with PRESET and CLEAR Inputs

CLEAR and PRESET are **Asynchronous Inputs** – They take precedence over S and R




Clear = 0 Preset = 1 Q=0 $\overline{Q}$=1

Clear = 1 Preset = 0 Q=1 $\overline{Q}$=0

Clear = 1, Preset = 1
Normal SR Flip-flop Operation

If Clear=0, Preset=0, then –

Q=0, $\overline{Q}$=1 if Clear Overrides Preset

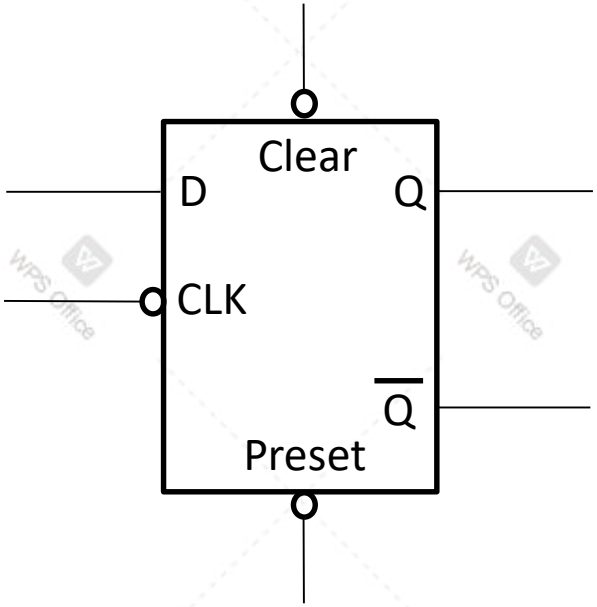
Q=1, $\overline{Q}$=0 if Preset Overrides Clear

# Other Types of Flip-Flops

| | |
|---|---|
| **D Flip-flop** | Shifts one bit of data from input to output; only one input required |
| **T Flip-flop** | Toggles the bit stored (from 1 to 0 or 0 to 1); only one input required |
| **JK Flip-flop** | Like the SR flip-flop except that it allows both inputs being high; two inputs required |

*One can change from one type of flip-flop to another with fairly simple additional circuitry*

# D Flip-flop with PRESET and CLEAR Inputs




Clear = 0 Preset = 1 Q=0 $\overline{Q}$=1

Clear = 1 Preset = 0 Q=1 $\overline{Q}$=0

Clear = 1, Preset = 1
Normal D Flip-flop Operation

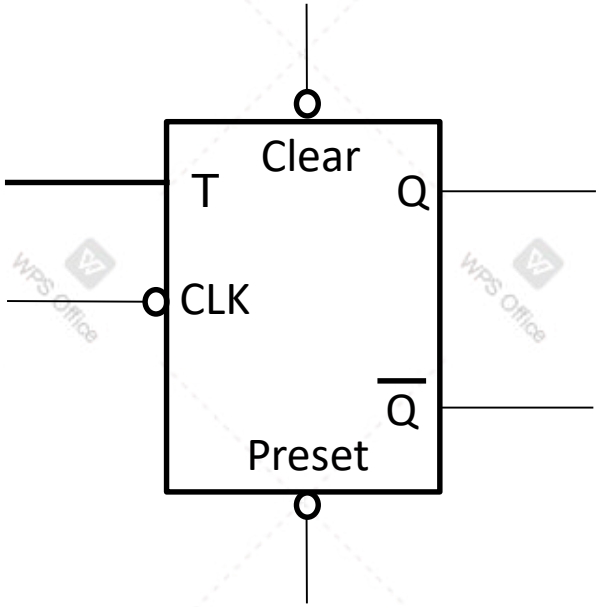
$D_n$=0 $Q_{n+1}$=0
$D_n$=1 $Q_{n+1}$=1

If Clear=0, Preset=0, then –

Q=0, $\overline{Q}$=1 if Clear Overrides Preset

Q=1, $\overline{Q}$=0 if Preset Overrides Clear

# T Flip-flop with PRESET and CLEAR Inputs




Clear = 0 Preset = 1 Q=0 $\overline{Q}$=1

Clear = 1 Preset = 0 Q=1 $\overline{Q}$=0

Clear = 1, Preset = 1
Normal T Flip-flop Operation

T=0 $Q_{n+1} = Q_n$

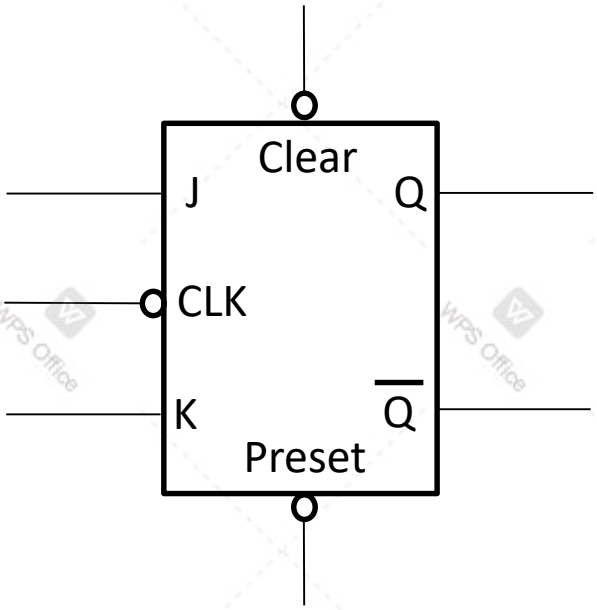
T=1 $Q_{n+1} = \overline{Q_n}$

If Clear=0, Preset=0, then –

Q=0, $\overline{Q}$=1 if Clear Overrides Preset

Q=1, $\overline{Q}$=0 if Preset Overrides Clear

# JK Flip-flop with PRESET and CLEAR Inputs




Clear = 0 Preset = 1 Q=0 $\overline{Q}$=1

Clear = 1 Preset = 0 Q=1 $\overline{Q}$=0

Clear = 1, Preset = 1
Normal J-K Flip-flop Operation

| $J_n$ | $K_n$ | $Q_{n+1}$ |
|---|---|---|
| 0 | 0 | $Q_n$ |
| 0 | 1 | 0 |
| 1 | 0 | 1 |
| 1 | 1 | $\overline{Q}_n$ |

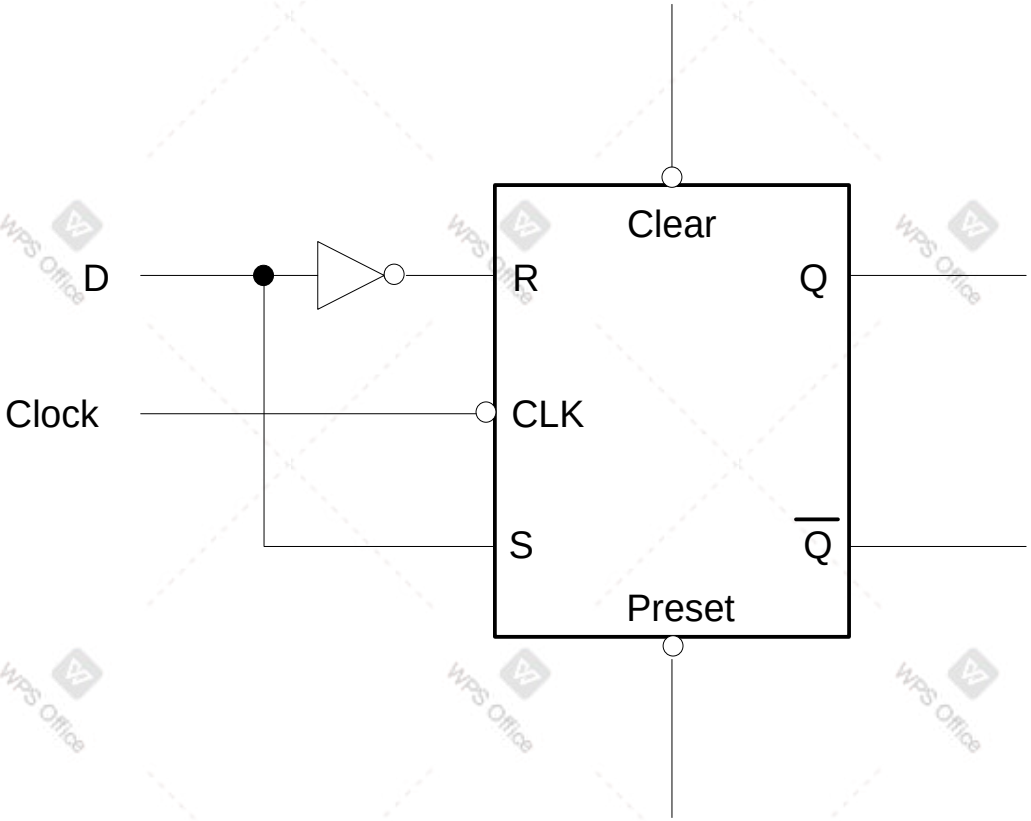
If Clear=0, Preset=0, then –

Q=0, $\overline{Q}$=1 if Clear Overrides Preset

Q=1, $\overline{Q}$=0 if Preset Overrides Clear

# Converting One Type of Flip-flop to Another

## Example: SR to D

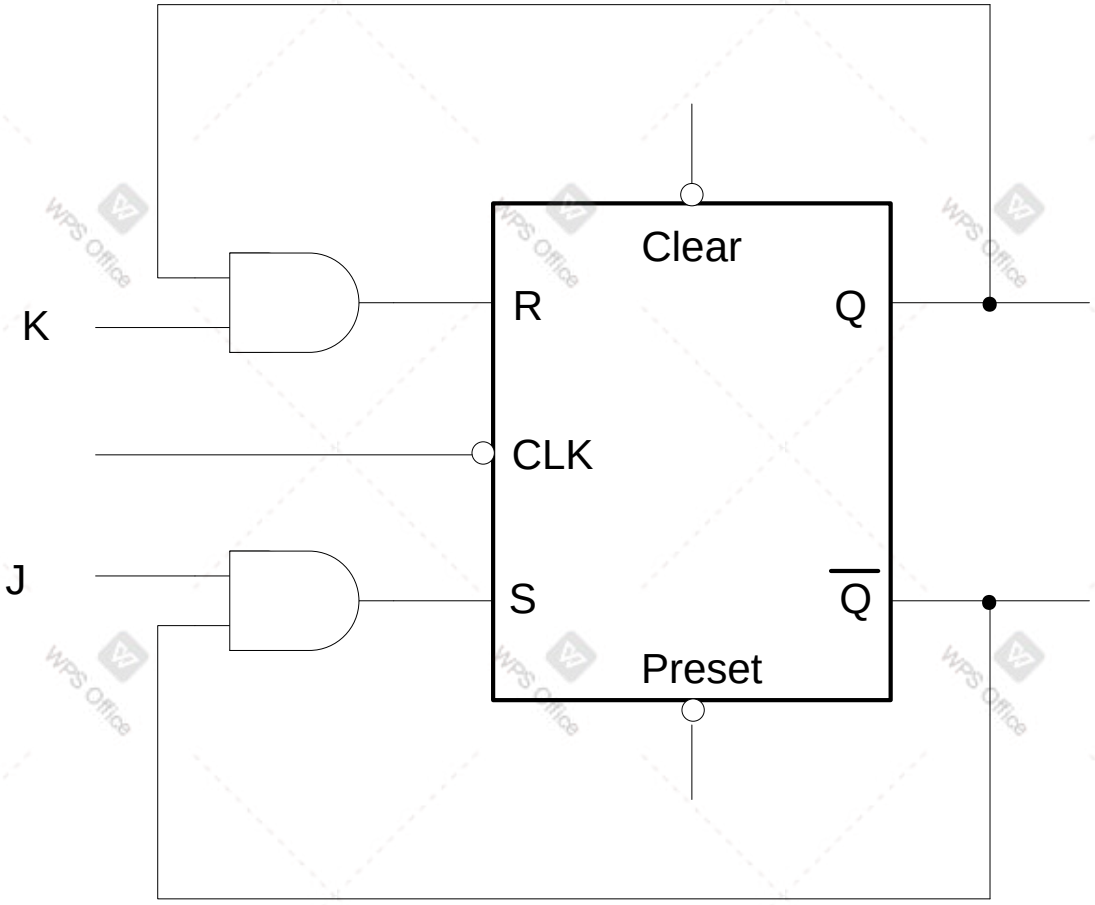



| $D_n$ | $Q_{n+1}$ |
|---|---|
| 0 | 0 |
| 1 | 1 |

| $R_n$ | $S_n$ | $Q_{n+1}$ |
|---|---|---|
| 0 | 0 | $Q_n$ |
| 0 | 1 | 1 |
| 1 | 0 | 0 |

# Converting One Type of Flip-flop to Another

## Example: SR to JK




| $J_n$ | $K_n$ | $Q_{n+1}$ |
|---|---|---|
| 0 | 0 | $Q_n$ |
| 0 | 1 | 0 |
| 1 | 0 | 1 |
| 1 | 1 | $\overline{Q_n}$ |

| $R_n$ | $S_n$ | $Q_{n+1}$ |
|---|---|---|
| 0 | 0 | $Q_n$ |
| 0 | 1 | 1 |
| 1 | 0 | 0 |

# Example: SR to JK

Table of ($R_n$, $S_n$) required given $J_n$, $K_n$ and $Q_n$

| $J_n$ | $K_n$ | $Q_{n+1}$ |
|---|---|---|
| 0 | 0 | $Q_n$ |
| 0 | 1 | 0 |
| 1 | 0 | 1 |
| 1 | 1 | $\overline{Q}_n$ |

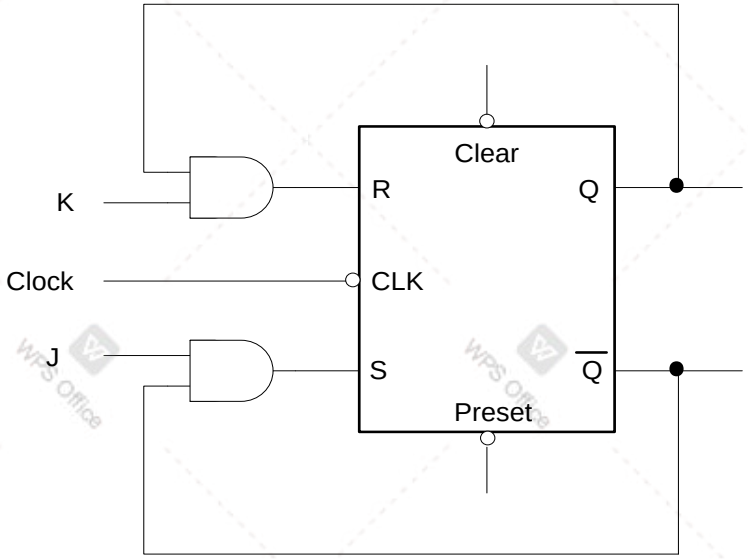
| $R_n$ | $S_n$ | $Q_{n+1}$ |
|---|---|---|
| 0 | 0 | $Q_n$ |
| 1 | 0 | 0 |
| 0 | 1 | 1 |
| 1 | 1 | x |

$R_n$,$S_n$

| $Q_n$ \ $J_n$,$K_n$ | 00 | 01 | 11 | 10 |
|---|---|---|---|---|
| 0 | x0 | x0 | 01 | 01 |
| 1 | 0x | 10 | 10 | 0x |

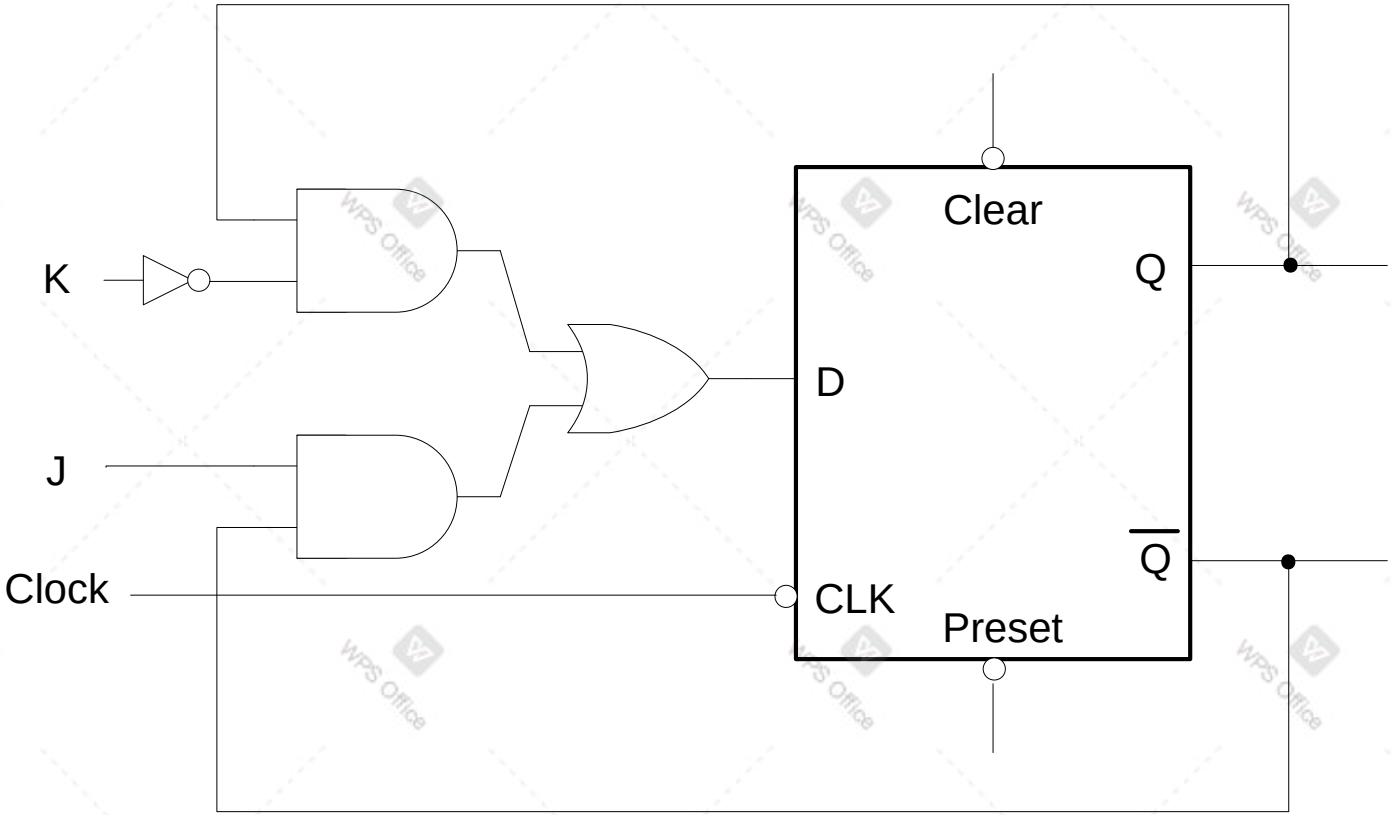
This gives –

$R_n = K_n Q_n$ $S_n = J_n \overline{Q}_n$



# Converting One Type of Flip-flop to Another

## Example: D to JK



# Example: D to JK

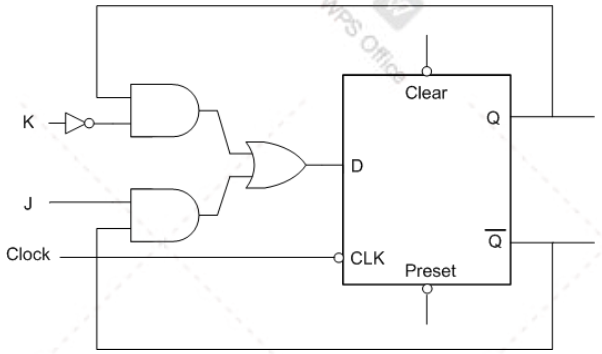
Table of $D_n$ required given $J_n$, $K_n$ and $Q_n$

$D_n$

| $Q_n$ \ $J_n,K_n$ | 00 | 01 | 11 | 10 |
|---|---|---|---|---|
| 0 | 0 | 0 | 1 | 1 |
| 1 | 1 | 0 | 0 | 1 |

This gives –

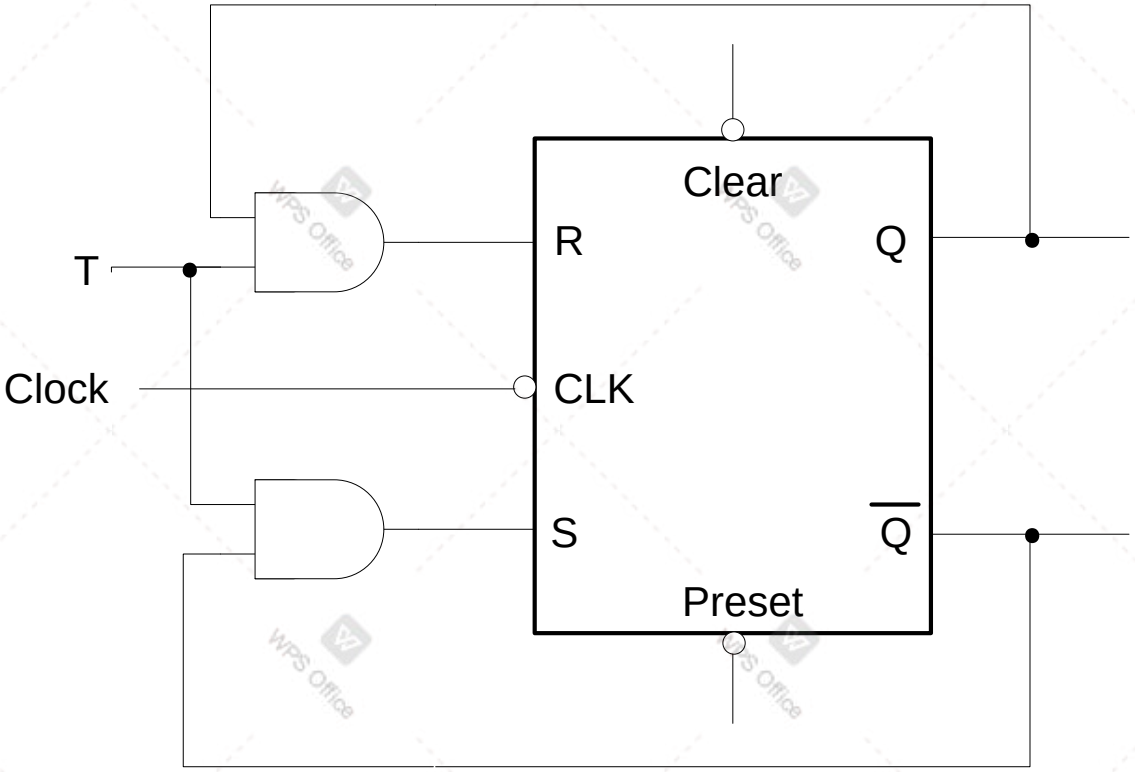
$$D_n = J_n\overline{Q_n} + \overline{K_n}Q_n$$




| $Q_n$ | $J_n$ | $K_n$ | $D_n$ |
|---|---|---|---|
| 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 | 0 |

# Converting One Type of Flip-flop to Another

## Example: SR to T



# Analysis and Design of Sequential Circuits

# Applications of Flip-flops

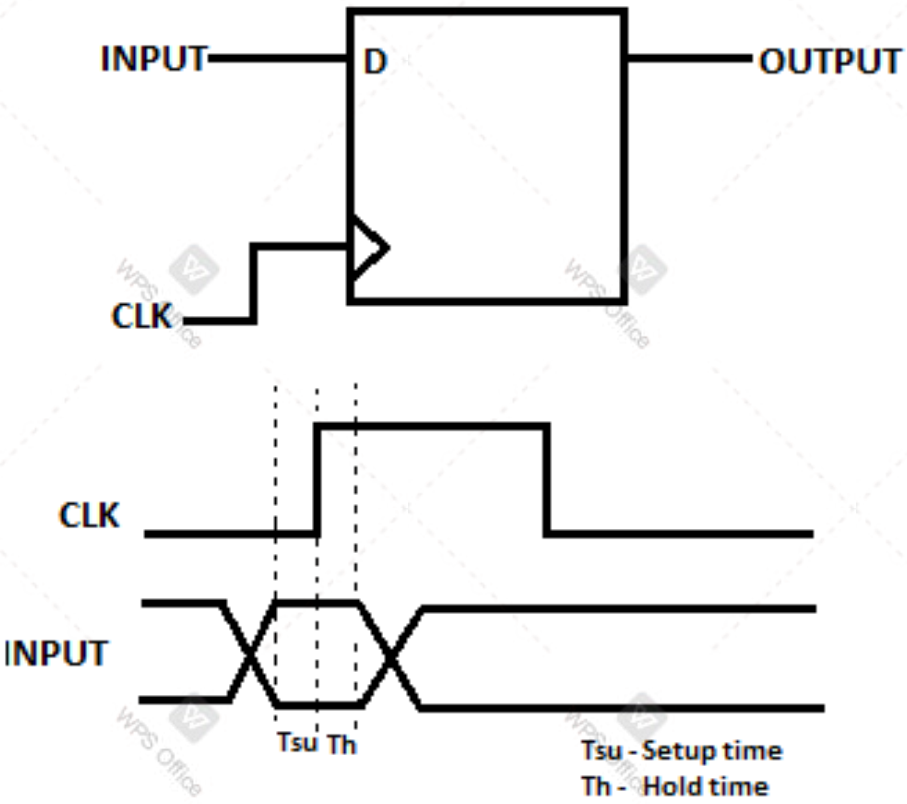
- Counters
  - A clocked sequential circuit that goes through a predetermined sequence of states
  - A commonly used counter is an $n$-bit binary counter. This has $n$ FFs and $2^n$ states which are passed through in the order $0, 1, 2, \ldots, (2^n-1), 0, 1, 2, \ldots$
  - Typical uses include:
    - Counting
    - Producing delays of a particular duration
    - Sequencers for control logic in a processor
    - Divide by $m$ counter (a divider), as used in a digital watch

# Applications of Flip-flops

- Storage and Transfer of Data
  - Memories
    - Primary memory (RAM and ROM) and
    - Secondary memory (Hard Drive, CD, etc.).
  - Shift register
    - Parallel loading shift register: can be used for parallel to serial conversion in serial data communication
    - Serial in, parallel out shift register: can be used for serial to parallel conversion in a serial data communication system

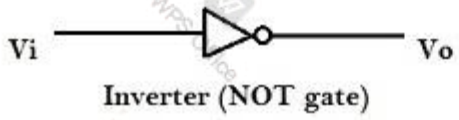
# Flip-flop Timings

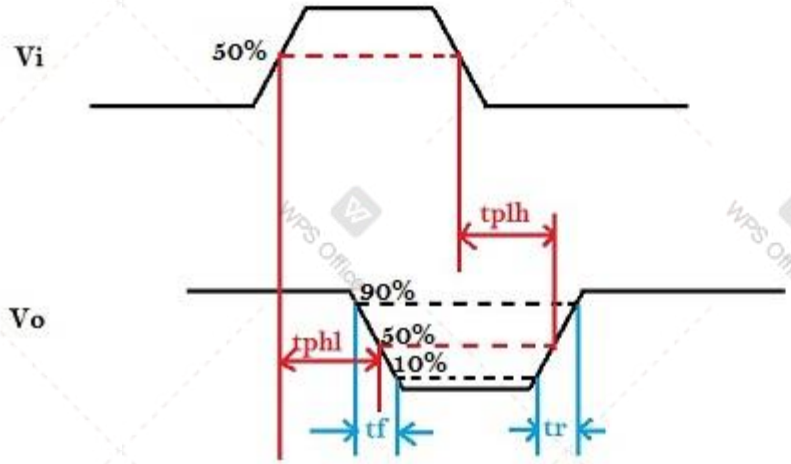
- Various timings must be satisfied if a FF is to operate properly:
  - **Setup time:** is the minimum duration that the data must be stable (unchanging) at the input before the clock edge
  - **Hold time:** is the minimum duration that the data must remain stable on the FF input after the clock edge



# Propagation Delay of Logic Gate

- The propagation delay (tp) of a logic gate defines how quickly it responds to a change at its input(s)
- It is measured between 50% transition points of the input and output waveforms



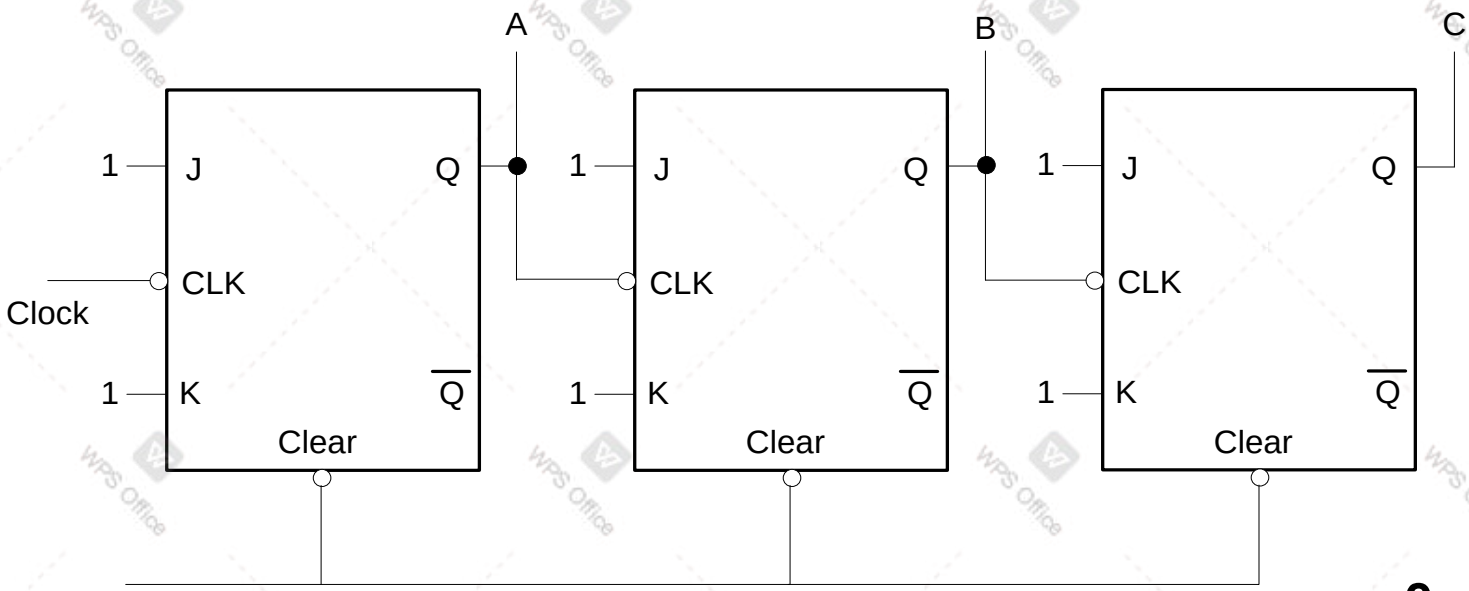




tr = Rise transition time
tf = Fall transition time
tphl = Propagation delay high-low
tplh = Propagation delay low-high

# Counters

- There are two basic types of counters, namely ripple counters and synchronous counters
- Ripple counters
  - Realized by cascading the flip-flops
  - Simpler to design but suffer from some problems
- Synchronous counters
  - all the FF clock inputs are directly connected to the clock signal and so all FF outputs change at the same time, i.e., synchronously
  - more complex combinational logic is now needed to generate the appropriate FF input signals (which will be different depending upon the type of FF chosen)

# 3-Bit, Modulo-8 Ripple Counter

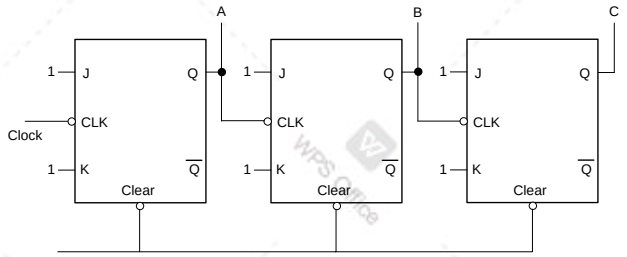


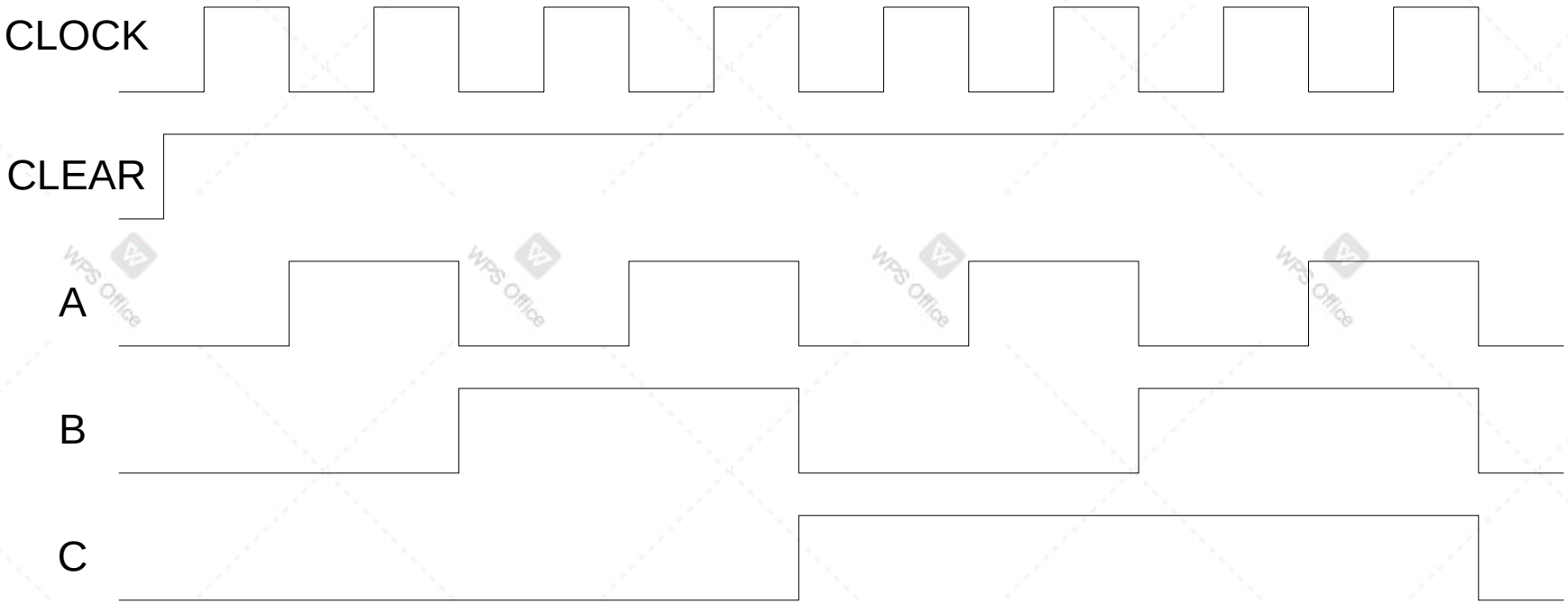

Clear = 0, Counter Reset to 000
Clear = 1, Counter Starts Counting

Counts number of clock pulses (falling edges) and cycles back to 0 after state 7

| | C | B | A |
|---|---|---|---|
| 0 | 0 | 0 | 0 |
| 1 | 0 | 0 | 1 |
| 2 | 0 | 1 | 0 |
| 3 | 0 | 1 | 1 |
| 4 | 1 | 0 | 0 |
| 5 | 1 | 0 | 1 |
| 6 | 1 | 1 | 0 |
| 7 | 1 | 1 | 1 |

# 3-Bit, Modulo-8 Ripple Counter




Also called a **Divide-By-8 Counter** as the frequency of the Most Significant Bit output (i.e. C) has frequency 1/8$^{th}$ of the original CLOCK frequency

# K-Bit, Modulo-$2^K$ Ripple Counter

Use same approach as in the previous slide – i.e. Q-output of the $n^{th}$ stage connected CLOCK input of the $(n+1)^{th}$ stage

Need K number of J-K flip-flops

Use CLEAR to initialize counter to 00….000

Counter counts as 0, 1, 2,..., $2^K$-2, $2^K$-1, 0, 1,…

This will be a Divide-By-$2^K$ Counter

*Ripple Counters are easy to build but have some problems (outlined later) because of which they cannot be used when high speeds (i.e. high frequency clock signals) are needed!*

# Divide-By-N Ripple Counter when N is not a Power of 2

In the previous slide, we showed how to make a Modulo-$2^K$ Ripple Counter. This counter will have $2^K$ states, i.e. a run length of $2^K$ as it will keep cycling through $2^K$ states, i.e. 0 to $2^K$-1.

Suppose we want a counter which runs through N states (0,1,…….N-1) where N is **not a power of 2** – How can we do this?

# Designing a Divide-By-N Ripple Counter when N is not a Power of 2

1. Find n such that $2^{n-1} \leq N \leq 2^n$

2. Connect n flip-flops as a ripple counter (i.e Q-output of a flip-flop feeding the CLOCK input of the next flip-flop

3. Get the binary equivalent of the number N

4. NAND all flip-flop outputs which will be 1 for a count of N and connect the NAND output to the CLEAR inputs of all the flip-flops

# Example: Design a Divide-By-6 Ripple Counter

- Need 3 flip-flops (say denoted by ABC) and connected in cascaded manner

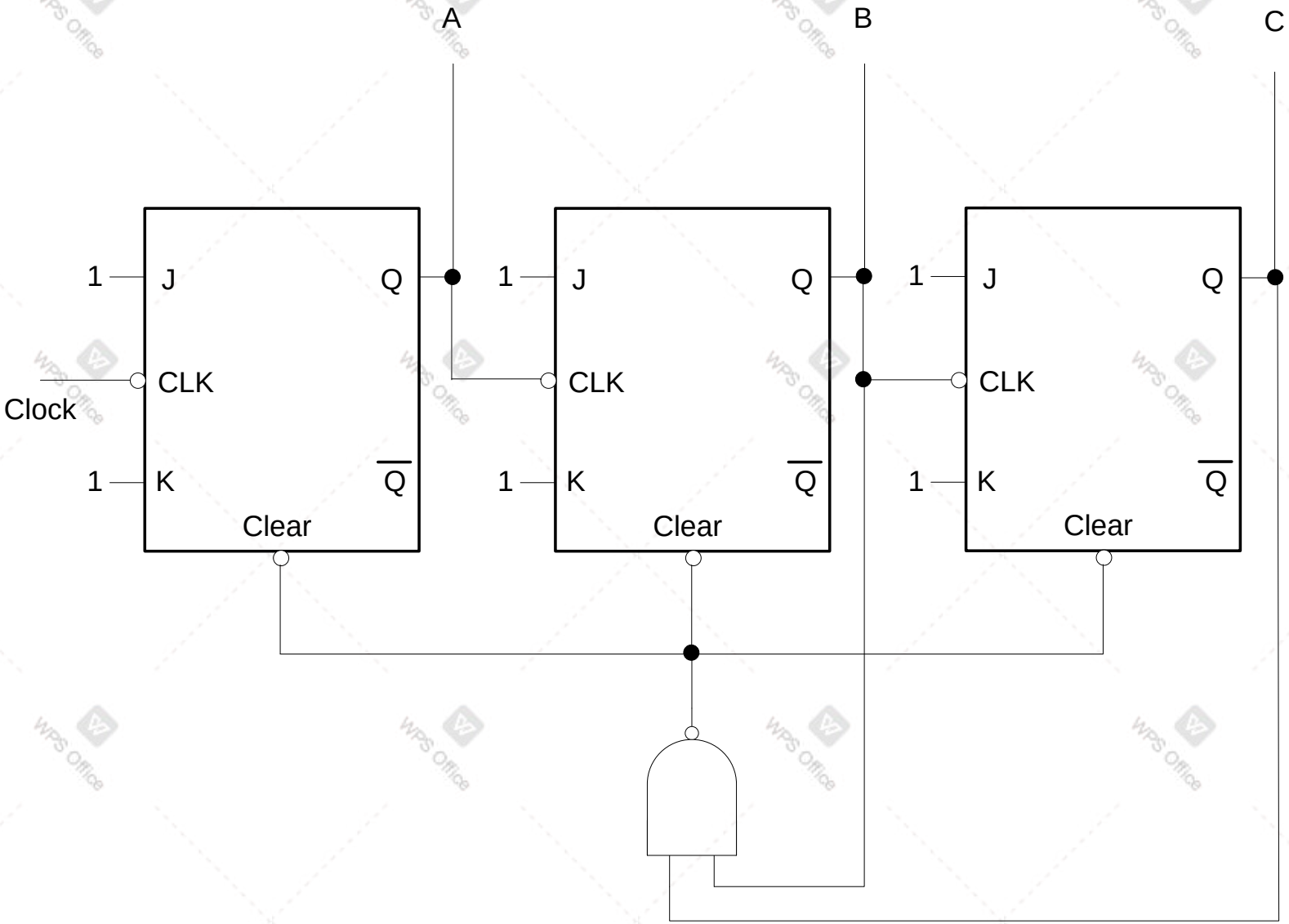
- 6 corresponds to CBA of 110

- CLEAR should be driven by $\overline{CB}$

- Counter counts in sequence 0,1, 2, 3, 4, 5. The moment the counter reaches 6 the flip-flops are all cleared and the counter gets reset to 000.

*The counter does reach state 6 but the moment it does that it gets reset to 000. Therefore, the count sequence observed will be …0,1,2,3,4,5,0,……..giving a Divide-by-6 counter (6 states)*

# Example: Design a Divide-By-6 Ripple Counter



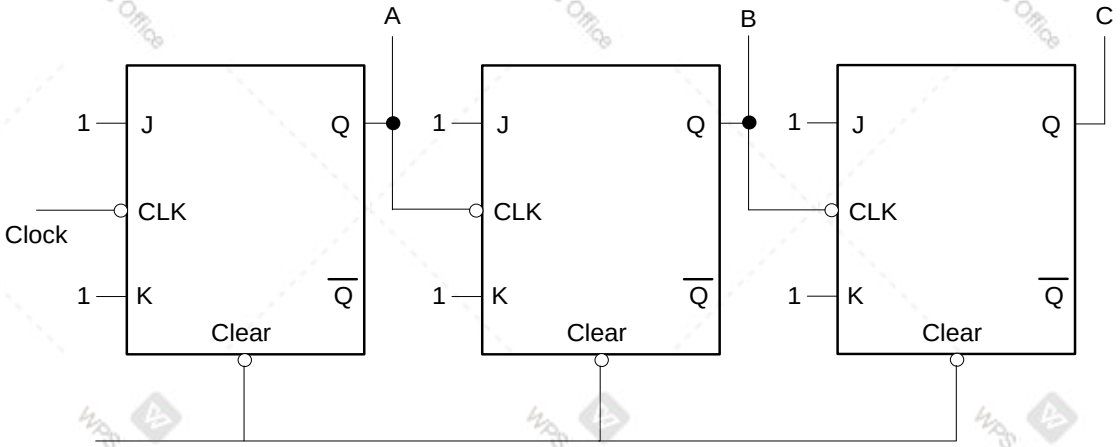
# An Inherent Problem with Ripple Counters

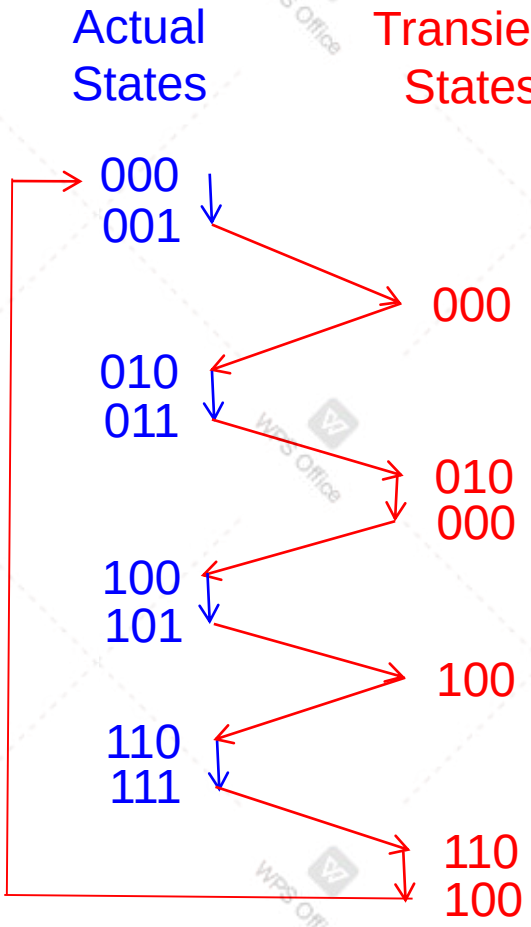
The flip-flops in a Ripple Counter do not change states together!

Examining it closely, we can see that the change propagates from the lower order flip-flops to the higher ones.

The flip-flops eventually reach the desired final state but “false” (i.e. transient states) appear in between. This limits the speed at which these counters can be used.

# Transient States in a Ripple Counter




If we use the counter states to trigger some other circuit then we run the risk of false triggering (because of the Transient States)!

*This problem is avoided by using Synchronous Counters, where all the flip-flops are forced to change state at the same time*

# Synchronous Counter

We consider a simple example of building a **Divide-By-4 Synchronous Counter** using J-K flip-flops to illustrate the technique. (Other types of flip-flops may also be used with appropriate changes in the design.)

Since we want to make a Divide-By-4 Counter, we will need two J-K flip-flops, A and B.

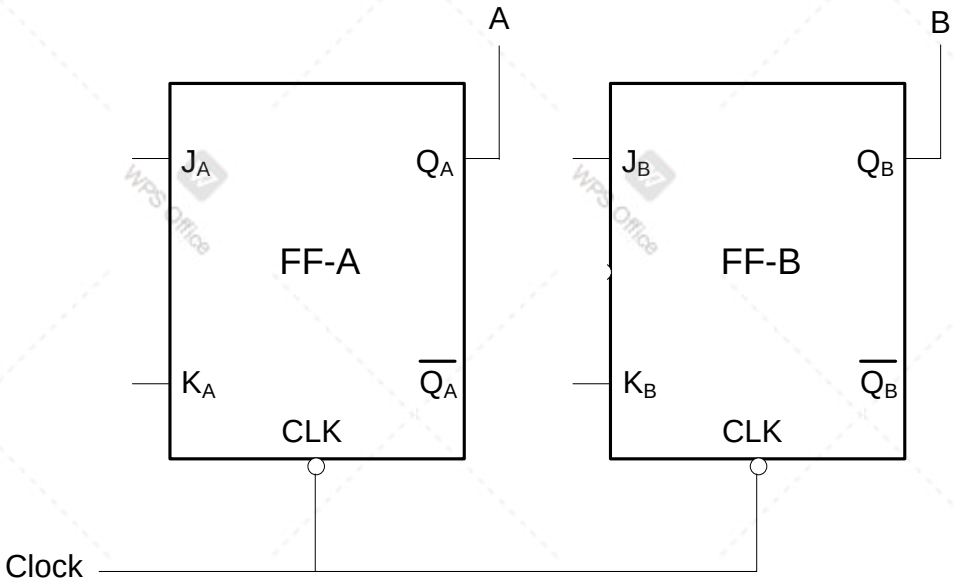
# Divide-By-4 Synchronous Counter

Since, the counter is a synchronous one, the CLK input of both the FFs will be directly driven by the CLOCK signal – that will make the two flip-flops change states together, eliminating the transient states that were appearing in the case of the Ripple Counter.

We now need to ensure that the $J_A$, $K_A$, $J_B$ and $K_B$ inputs of the flip-flops A and B get the right values so that the count sequence 00,01,10,11, 00…. is achieved.

# Divide-By-4 Synchronous Counter

The basic structure of our counter will be as shown below:




Note that we still have to find what to input for $J_A$, $K_A$, $J_B$ and $K_B$

# Divide-By-4 Synchronous Counter:

| Present State | | Next State | | FF Inputs | | | |
|---|---|---|---|---|---|---|---|
| $Q_B$ | $Q_A$ | $Q_B$ | $Q_A$ | $J_B$ | $K_B$ | $J_A$ | $K_A$ |
| 0 | 0 | 0 | 1 | 0 | x | 1 | x |
| 0 | 1 | 1 | 0 | 1 | x | x | 1 |
| 1 | 0 | 1 | 1 | x | 0 | 1 | x |
| 1 | 1 | 0 | 0 | x | 1 | x | 1 |

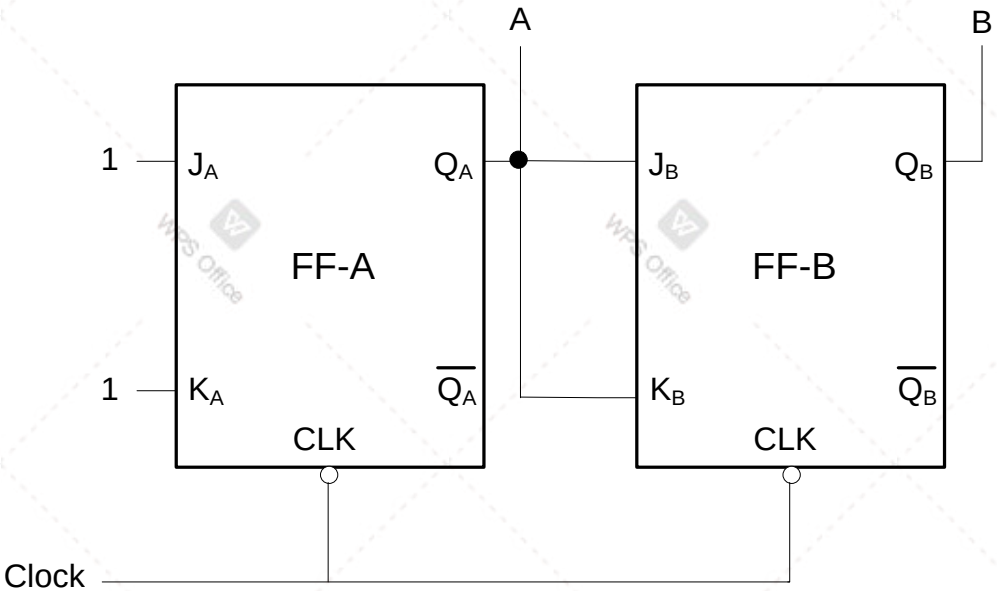
| $J_n$ | $K_n$ | $Q_{n+1}$ |
|---|---|---|
| 0 | 0 | $Q_n$ |
| 0 | 1 | 0 |
| 1 | 0 | 1 |
| 1 | 1 | $\overline{Q_n}$ |

Using Karnaugh maps or otherwise, we can show that

$J_A=K_A=1$ $J_B=K_B=Q_A$

# Divide-By-4 Synchronous Counter:

The final design is shown below




Verify that this will operate synchronously with no false transient states.

# Analysis of Sequential Circuits

- The behavior of a sequential circuit can be determined from the inputs, the outputs and the states of its flip-flops
- The outputs and the next states are the function of its inputs and the present states
- The analysis of a sequential circuit consists of obtaining a table or diagram for the time sequence of inputs, outputs and internal states
- The systematic techniques which aid the analysis and design of a sequential circuit include:
  - State table
  - State diagram
  - State equations

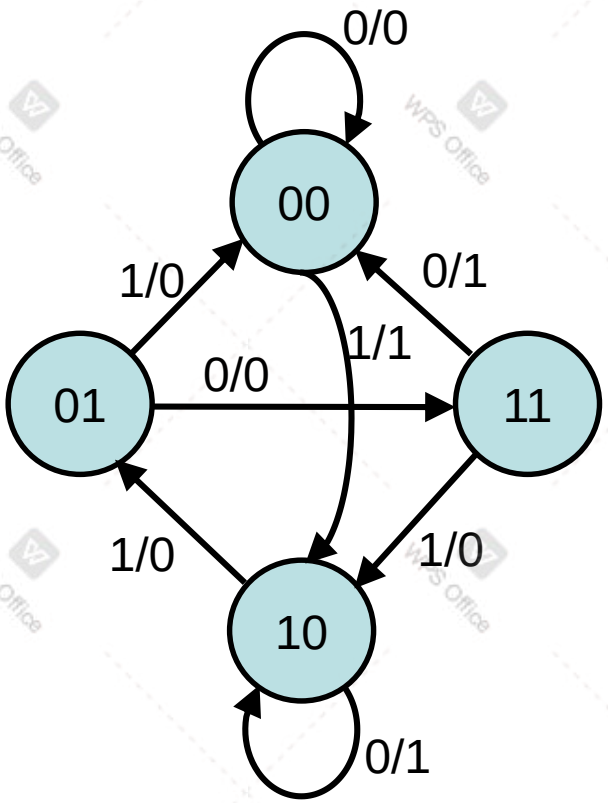
## Procedure of designing sequential circuits

- Write down the state transition table
- Determine the FF excitation
- Determine the combinational logic necessary to generate the required FF excitation from the current states

  Note: remember to take into account any unused counts since these can be used as don’t care states when determining the combinational logic circuits
- The procedure can be used to design counters having an arbitrary count sequence

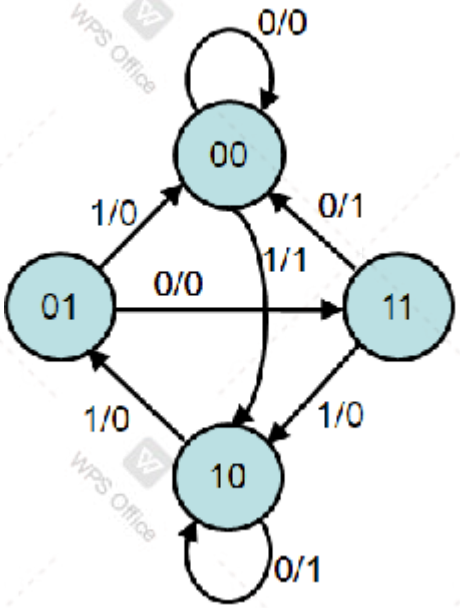
# Example

- A sequential circuit has one input and one output and its state diagram is shown below. Design the sequential circuit using (i) T FFs (ii) D FFs and (iii) JK FFs.



# Example: Solution

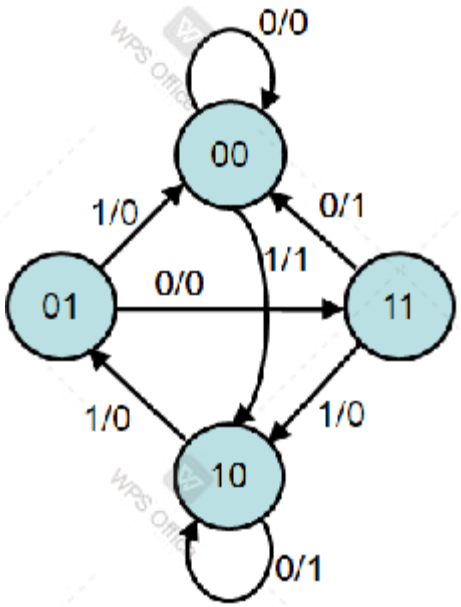
- The circuit consists of four states so it would require two FFs and those are denoted as A and B.
- From the state diagram, write the state table with single input and output being denoted as x and y, respectively.




| Present state | Next state | | Output | |
|---|---|---|---|---|
| | x = 0 | x = 1 | x = 0 | x = 1 |
| A B | A B | A B | y | y |
| 0 0 | 0 0 | 1 0 | 0 | 1 |
| 0 1 | 1 1 | 0 0 | 0 | 0 |
| 1 0 | 1 0 | 0 1 | 1 | 0 |
| 1 1 | 0 0 | 1 0 | 1 | 0 |

# Example: Solution Part (i)

- For designing the circuit using T FFs, we need to draw the excitation table as shown below, from which we can find the FF input equations and circuit output function

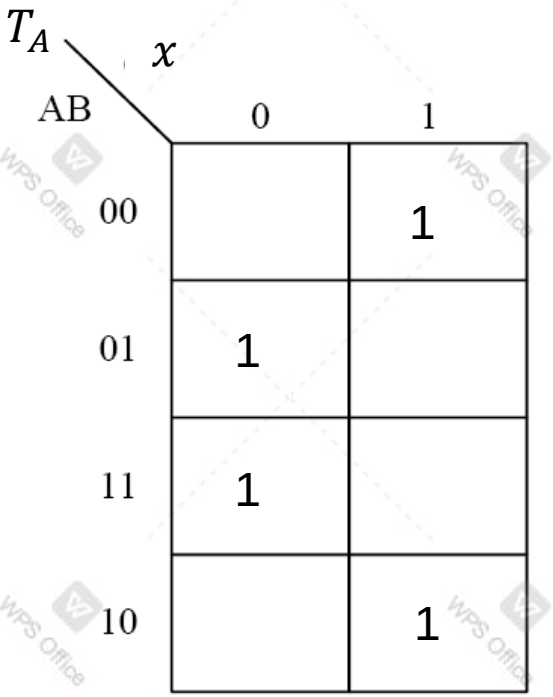



| Present state | | Input | Next state | | FF inputs | | Output |
|---|---|---|---|---|---|---|---|
| A | B | x | A | B | $T_A$ | $T_B$ | y |
| 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 |
| 0 | 0 | 1 | 1 | 0 | 1 | 0 | 1 |
| 0 | 1 | 0 | 1 | 1 | 1 | 0 | 0 |
| 0 | 1 | 1 | 0 | 0 | 0 | 1 | 0 |
| 1 | 0 | 0 | 1 | 0 | 0 | 0 | 1 |
| 1 | 0 | 1 | 0 | 1 | 1 | 1 | 0 |
| 1 | 1 | 0 | 0 | 0 | 1 | 1 | 1 |
| 1 | 1 | 1 | 1 | 0 | 0 | 1 | 0 |

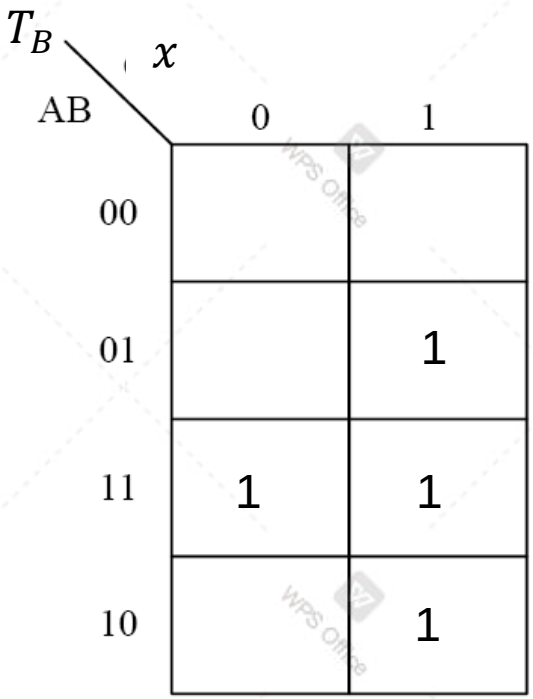
# Example: Solution Part (i)

- From the excitation table, we have

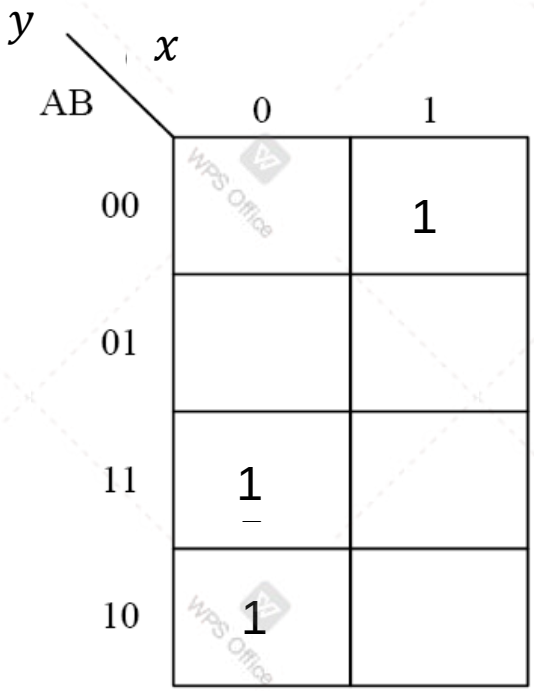
$$T_A = \Sigma m(1,2,5,6)\ ,\ T_B = \Sigma m(3,5,6,7)\ ,\ y = \Sigma m(1,4,6)$$




$$T_A = \bar{B}x + B\bar{x} = B \oplus x$$



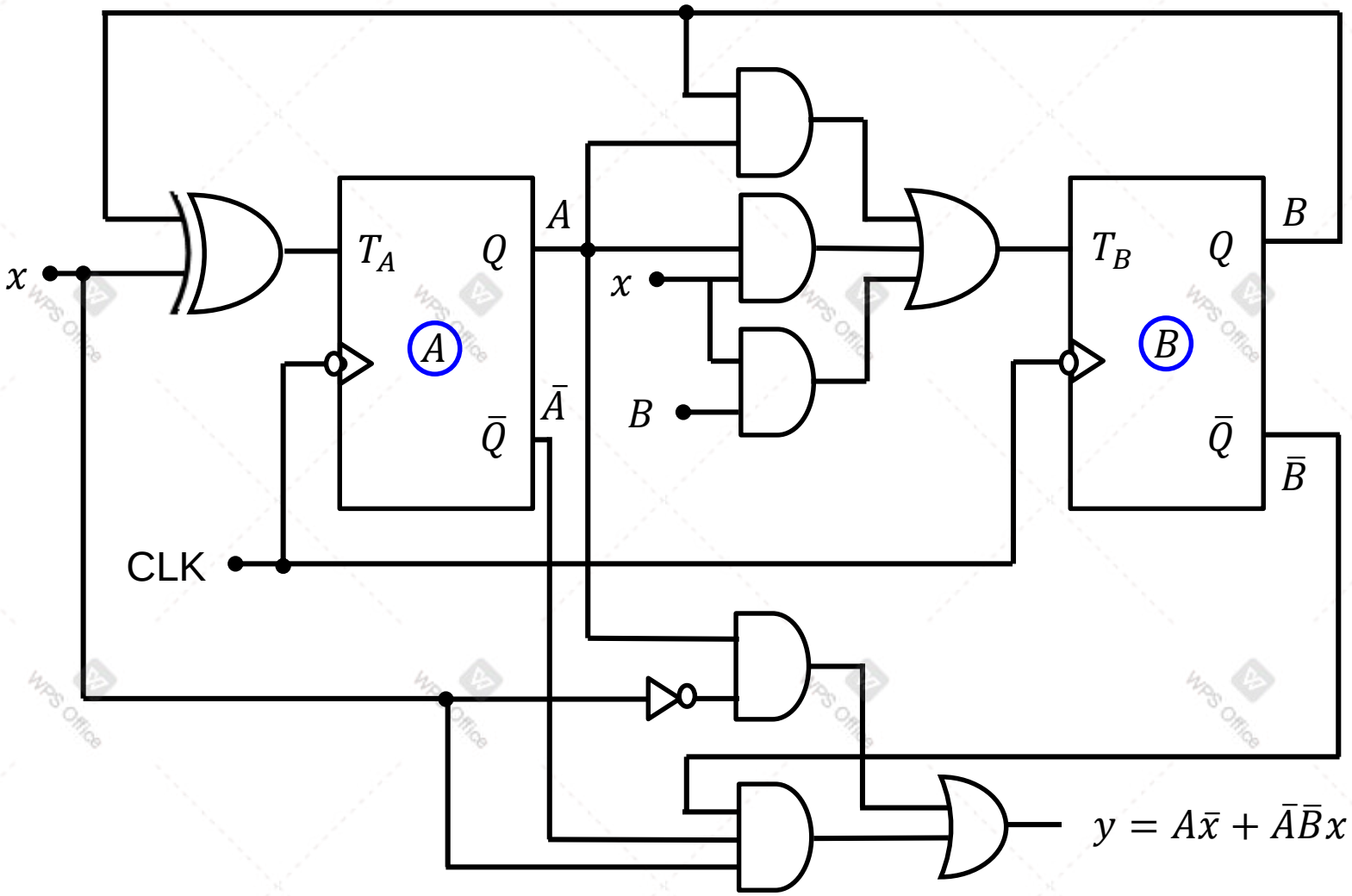

$$T_B = AB + Bx + Ax$$




$$y = A\bar{x} + \bar{A}\bar{B}x$$

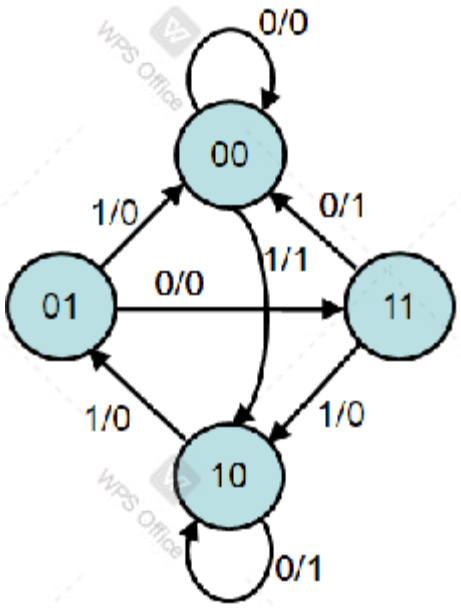
# Example: Solution Part (i)

Logic circuit using T flip-flops



# Example: Solution Part (ii)

- For designing the circuit using D FFs, we need to draw the excitation table as shown below, from which we can find the FF input equations and circuit output function

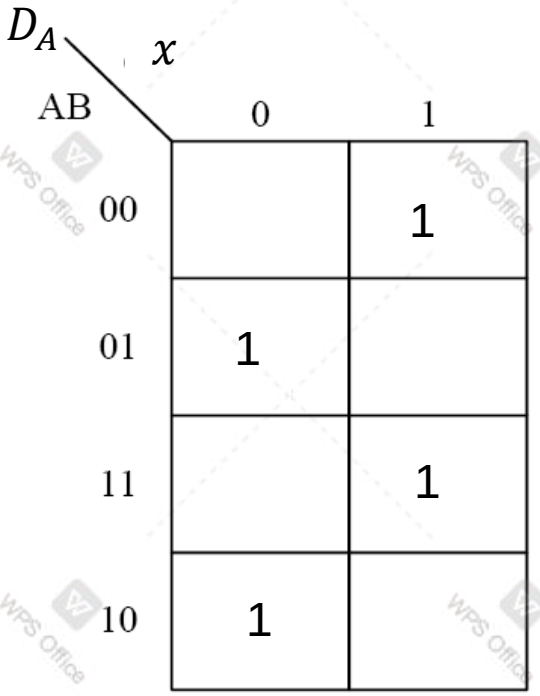



| Present state | | Input | Next state | | FF inputs | | Output |
|---|---|---|---|---|---|---|---|
| A | B | x | A | B | $D_A$ | $D_B$ | y |
| 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 |
| 0 | 0 | 1 | 1 | 0 | 1 | 0 | 1 |
| 0 | 1 | 0 | 1 | 1 | 1 | 1 | 0 |
| 0 | 1 | 1 | 0 | 0 | 0 | 0 | 0 |
| 1 | 0 | 0 | 1 | 0 | 1 | 0 | 1 |
| 1 | 0 | 1 | 0 | 1 | 0 | 1 | 0 |
| 1 | 1 | 0 | 0 | 0 | 0 | 0 | 1 |
| 1 | 1 | 1 | 1 | 0 | 1 | 0 | 0 |

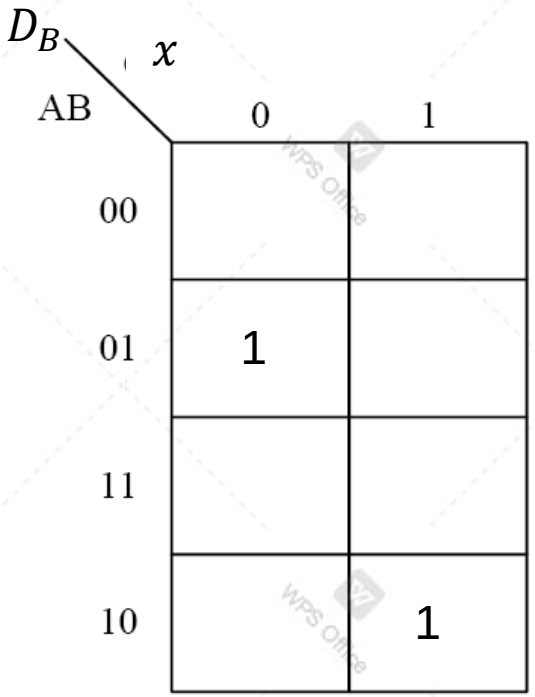
# Example: Solution Part (ii)

- From the excitation table, we have

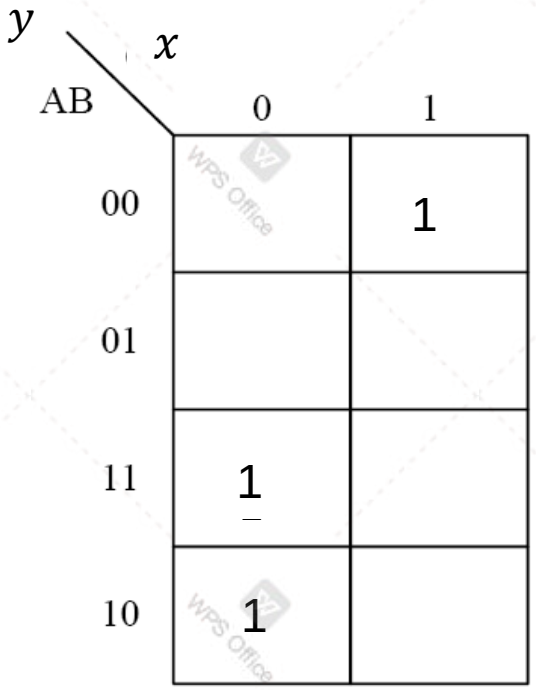
$$D_A = \Sigma m(1,2,4,7)\ ,\ D_B = \Sigma m(2,5)\ ,\ y = \Sigma m(1,4,6)$$




$$D_A = A \oplus B \oplus x$$



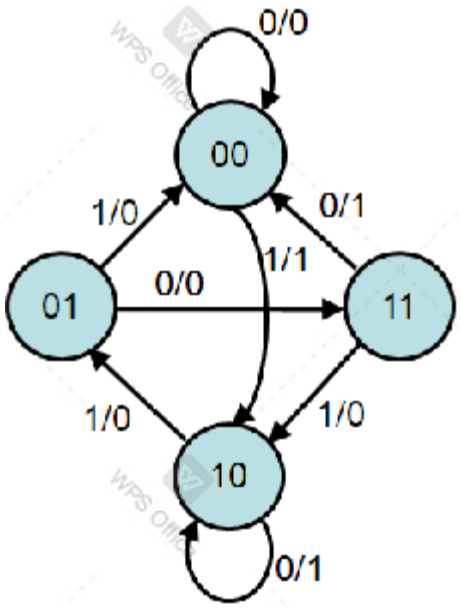

$$D_B = \bar{A}B\bar{x} + A\bar{B}x$$




$$y = A\bar{x} + \bar{A}\bar{B}x$$

# Example: Solution Part (iii)

- For designing the circuit using JK FFs, we need to draw the excitation table as shown below, from which we can find the FF input equations and circuit output function




| Present state | | Input | Next state | | FF inputs | | Output |
|---|---|---|---|---|---|---|---|
| A | B | x | A | B | $J_A K_A$ | $J_B K_B$ | y |
| 0 | 0 | 0 | 0 | 0 | 0 x | 0 x | 0 |
| 0 | 0 | 1 | 1 | 0 | 1 x | 0 x | 1 |
| 0 | 1 | 0 | 1 | 1 | 1 x | x 0 | 0 |
| 0 | 1 | 1 | 0 | 0 | 0 x | x 1 | 0 |
| 1 | 0 | 0 | 1 | 0 | x 0 | 0 x | 1 |
| 1 | 0 | 1 | 0 | 1 | x 1 | 1 x | 0 |
| 1 | 1 | 0 | 0 | 0 | x 1 | x 1 | 1 |
| 1 | 1 | 1 | 1 | 0 | x 0 | x 1 | 0 |

# Example: Solution Part (ii)

- From the excitation table, we have

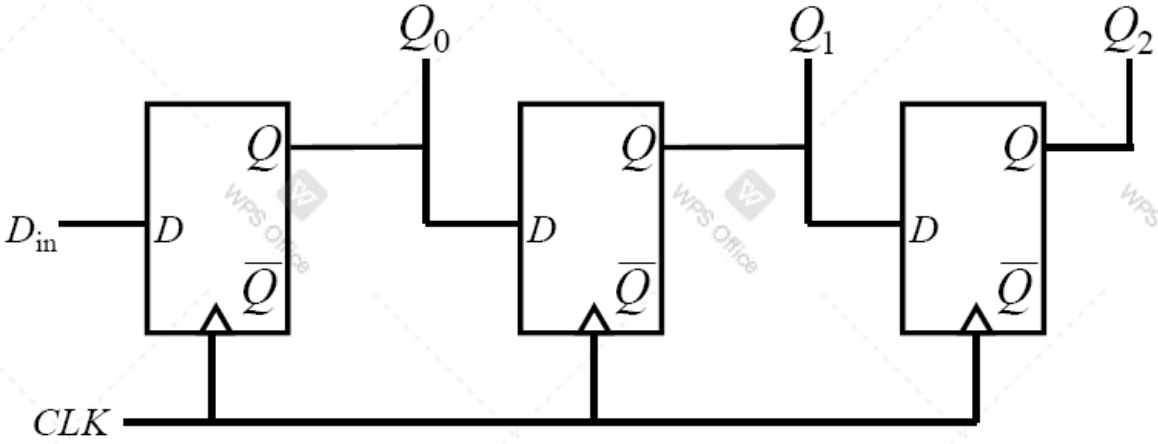
$$J_A = \Sigma m(1,2) + \Sigma d(4,5,6,7)$$
$$K_A = \Sigma m(5,6) + \Sigma d(0,1,2,3)$$
$$J_B = \Sigma m(5) + \Sigma d(2,3,6,7)$$
$$K_B = \Sigma m(3,6,7) + \Sigma d(0,1,4,5)$$
$$y = \Sigma m(1,4,6)$$

- On simplification, we have

$$J_A = B \oplus x \qquad K_A = B \oplus x$$
$$J_B = Ax \qquad K_B = A + x$$
$$y = A\bar{x} + \bar{A}\bar{B}x$$

# Shift Register

- A shift register can be implemented using a chain of D-type FFs




- It has a serial input $D_{\text{in}}$ and parallel outputs $Q_0$, $Q_1$ and $Q_2$
- Note, the data moves one position to the right on application of positive edge of the clock signal

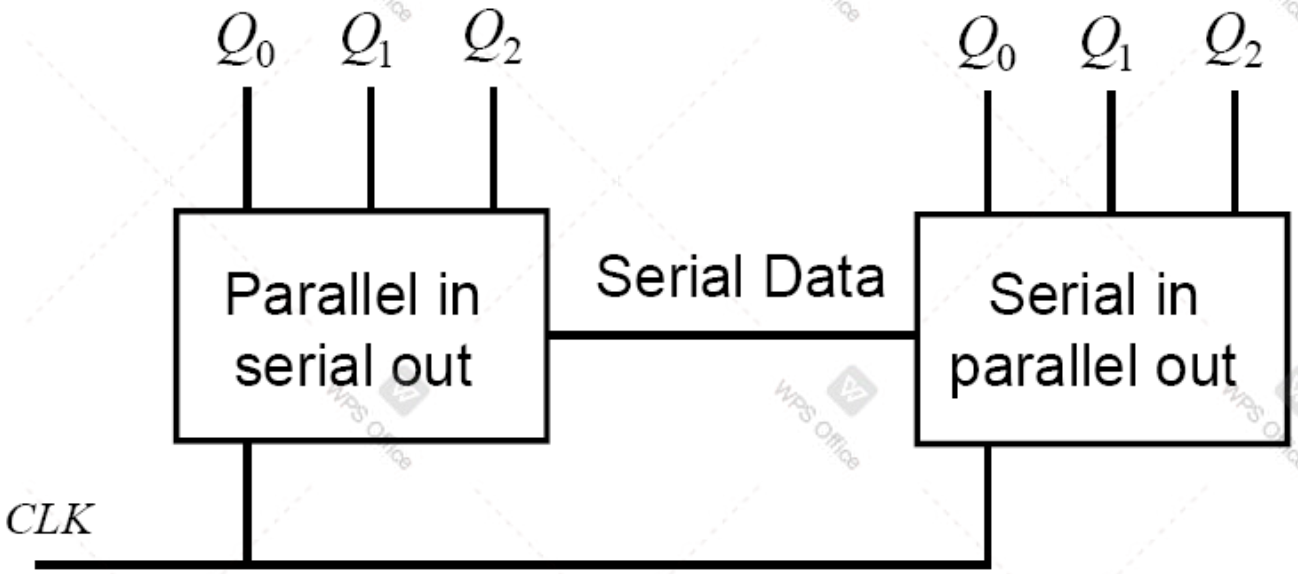
# Shift Register

- Preset and Clear inputs on the FFs can be utilized to provide a parallel data input feature

- Data can then be clocked out through $Q_2$ in a serial fashion, i.e., we now have a parallel in, serial out arrangement

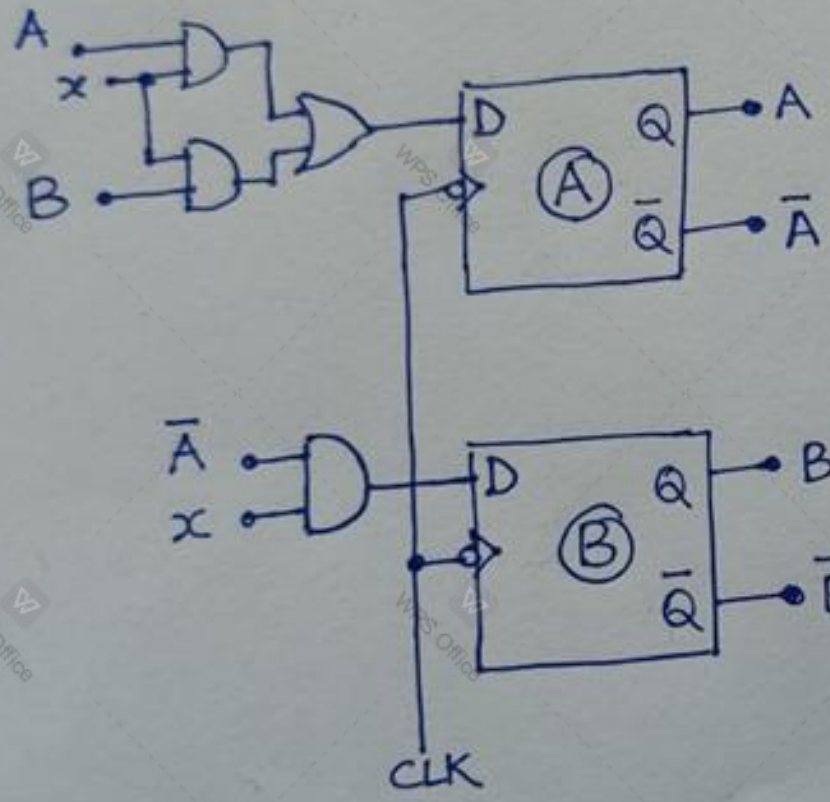
- This along with the previous serial in, parallel out shift register arrangement can be used as the basis for a serial data link

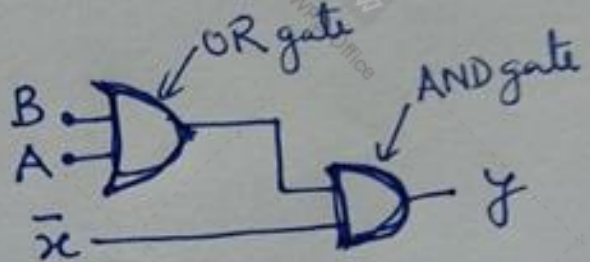
# Serial Data Link




- One data bit at a time is sent across the serial
- data link
- Note, less wires are required than for a parallel data link

Example-1 Derive the state table and state diagram for the sequential circuit shown below.



**Solution:** From the given circuit and the characteristics of D-FF, we can express following relations

$$A_{n+1} = Ax + B\bar{x}$$

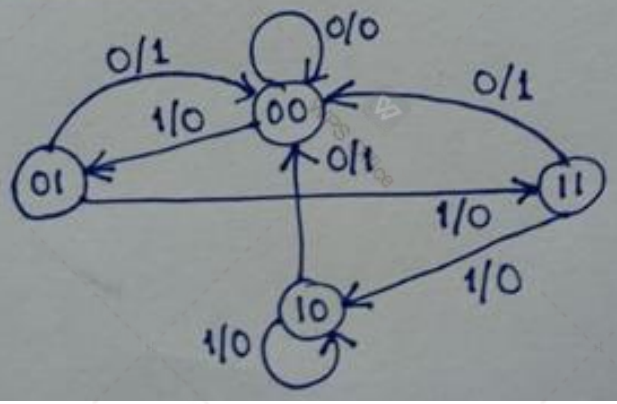
$$B_{n+1} = \bar{A}x$$

$$y = (A+B)\bar{x}$$

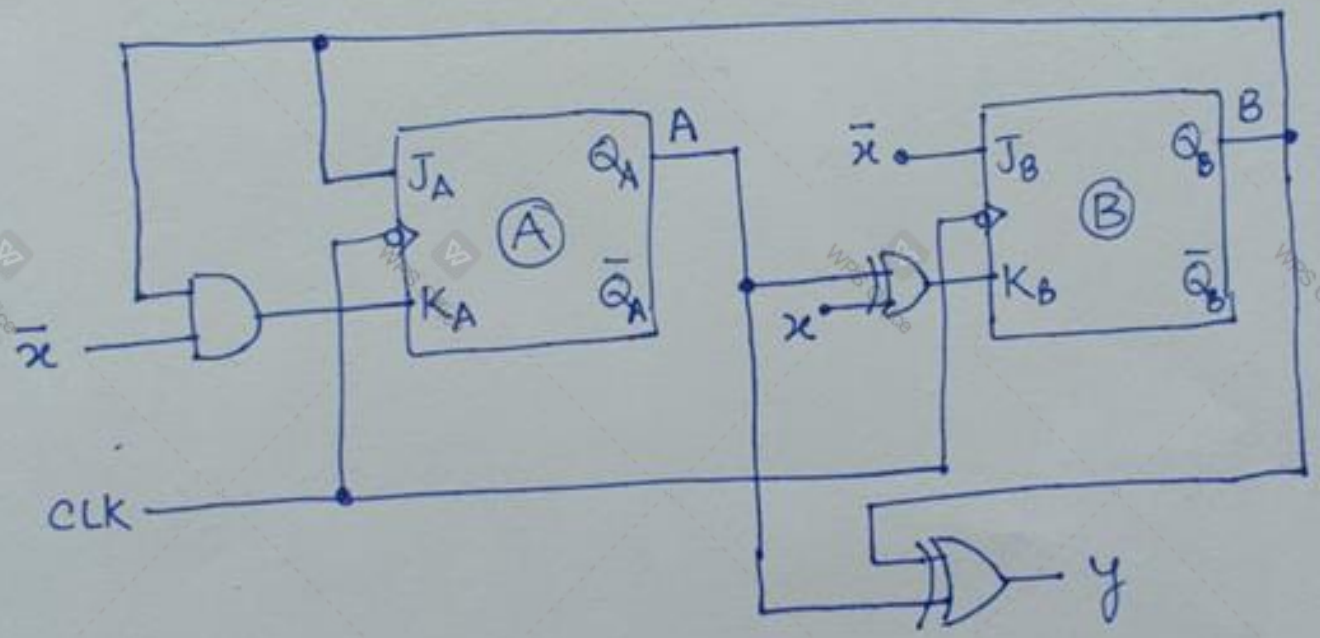
**State table:**

| Present State AB | Next State AB | | Output y | |
|---|---|---|---|---|
| | x=0 | x=1 | x=0 | x=1 |
| 00 | 00 | 01 | 0 | 0 |
| 01 | 00 | 11 | 1 | 0 |
| 10 | 00 | 10 | 1 | 0 |
| 11 | 00 | 10 | 1 | 0 |

**State diagram:**



Example-2 Derive the state table and state diagram for sequential circuit shown below.



Solution: The flip-flop input functions for the given sequential circuit are:

$J_A = B$ $\quad J_B = \bar{x}$ $\quad y = A \oplus B$

$K_A = B\bar{x}$ $\quad K_B = A \oplus x$

State table:

| Present state AB | Next State x=0 AB | Next State x=1 AB | Output y |
|---|---|---|---|
| 0 0 | 0 1 | 0 0 | 0 |
| 0 1 | 1 1 | 1 0 | 1 |
| 1 0 | 1 1 | 1 0 | 1 |
| 1 1 | 0 0 | 1 1 | 0 |

State diagram: Here each circle is coded with state binary number/output