

DIGITAL ELECTRONICS

III Semester CSE & CST



Mr. Srikanta Patnaik

Mr. Ajit Kumar Patro

Mr. Ami Kumar Parida

DEPARTMENT OF ELECTRONICS AND COMMUNICATION ENGINEERING

SCHOOL OF ENGINEERING & TECHNOLOGY

GIET UNIVERSITY, GUNUPUR, ODISHA

UNIT-1

Number Systems and Codes: Binary, Octal, Hexadecimal, and Decimal Number System and their Conversion; Representation of Signed Binary and Floating-Point Number; Binary Arithmetic using 1's and 2's Complements, Binary Codes - BCD Code, Gray Code, ASCII Character Code.

Boolean Algebra and Logic Gates: Axioms and Laws of Boolean Algebra; Reducing Boolean Expressions; Logic Levels and Pulse Waveforms; Logic Gates; Boolean Expressions and Logic Diagrams.

Gate-level Minimization: Canonical and Standard Forms; K-maps - Two, Three, and Four Variable K-maps, Don't-Care Conditions; NAND and NOR Implementation; Other Two-Level Implementations, Exclusive-OR Function.

UNIT-2

Combinational Logic: Combinational Circuits; Analysis Procedure; Design Procedure; Adders; SubTRACTors; Parallel Binary Adders; Binary Adder-SubTRACTor; Binary Multiplier; Magnitude Comparator; Decoders; Encoders; Multiplexers; De-multiplexers.

Synchronous Sequential Logic: Sequential Circuits; Latches, Flip-Flops; Master-Slave Flip-Flop; Conversion of Flip-Flops; Analysis of Clocked Sequential Circuits; Mealy and Moore Models of Finite State Machines.

UNIT-3

Registers and Counters: Shift Registers; Data Transmission in Shift Registers; SISO, SIPO, PISO, and PIPO Shift Registers; Counters; Asynchronous Counters; Design of Asynchronous Counters; Synchronous Counters; Design of Synchronous Counters; Ring Counter.

Memory and Programmable Logic: Introduction; Random-Access Memory; Memory Decoding; Error Detection and Correction; Read-Only Memory; Programmable Logic Array; Programmable Array Logic; Sequential Programmable Devices.

UNIT-4

Analog-to-Digital and Digital-to-Analog Converters: Digital-to-Analog Converters - R-2R Ladder D/A Converter, Weighted Resistor D/A Converter; Analog-to-Digital Converters - Counter-type A/D Converter, Parallel Comparator A/D Converter, Dual-Slope A/D Converter, Successive-Approximation A/D Converter, A/D Converter using Voltage-to-Frequency.

IC Logic Families: Special Characteristics; RTL, DTL, TTL, ECL, IIL, MOS, and CMOS Logic Circuits.

BOOKS

Text Books:

- | | |
|-----|---|
| [1] | M. Morris Mano, and Michael D. Ciletti, <i>Digital Design: With an Introduction to the Verilog HDL, VHDL, and SystemVerilog</i> . Pearson Education, Sixth Edition, 2017. |
| [2] | A. Anand Kumar, <i>Fundamentals of Digital Circuits</i> . PHI Learning Pvt. Ltd., New Delhi, Fourth Edition, 2016. |
| [3] | R. P. Jain, <i>Modern Digital Electronics</i> . Tata McGraw-Hill Education Pvt. Ltd., Fourth Edition, 2009. |

Reference Books:

- | | |
|-----|--|
| [1] | John P. Uyemura, <i>A First Course in Digital Systems Design: An Integrated Approach</i> . Thomson Press (India) Ltd., India Edition, 2002. |
| [2] | William H. Gothmann, <i>Digital Electronics: An Introduction to Theory and Practice</i> , PHI Learning Pvt. Ltd., New Delhi, Second Edition, 2006. |
| [3] | D.V. Hall, <i>Digital Circuits and Systems</i> . Tata McGraw-Hill Education Pvt. Ltd., 1989. |
| [4] | Charles H. Roth, <i>Digital System Design using VHDL</i> . Tata McGraw-Hill Education Pvt. Ltd., Second Edition, 2012. |

UNIT-1

Chapter-1

Number Systems and Codes

Binary, Octal,

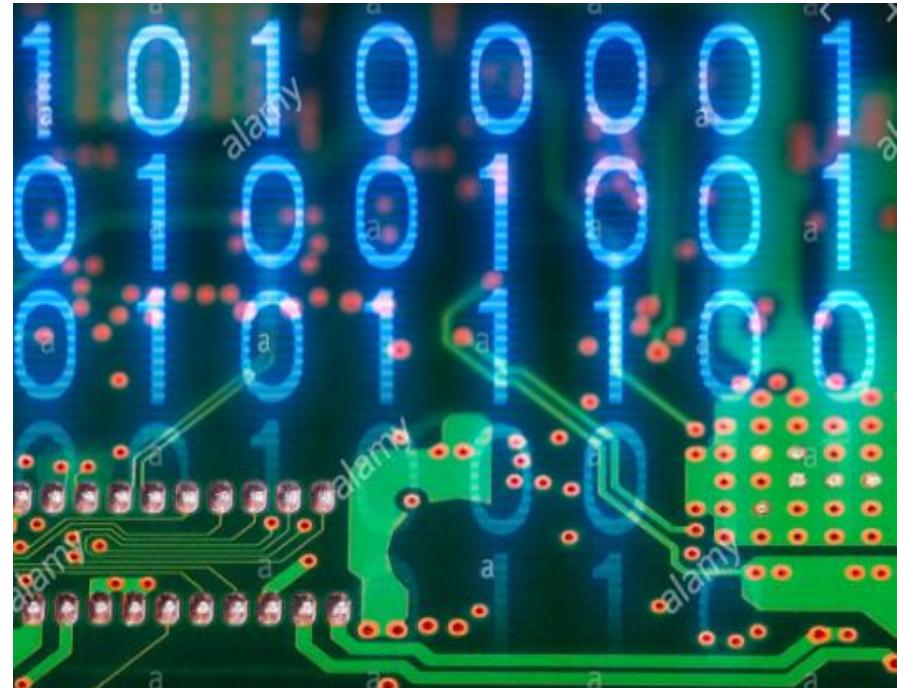
Hexadecimal, and

Decimal Number System

and their Conversion

What is Digital Electronics?

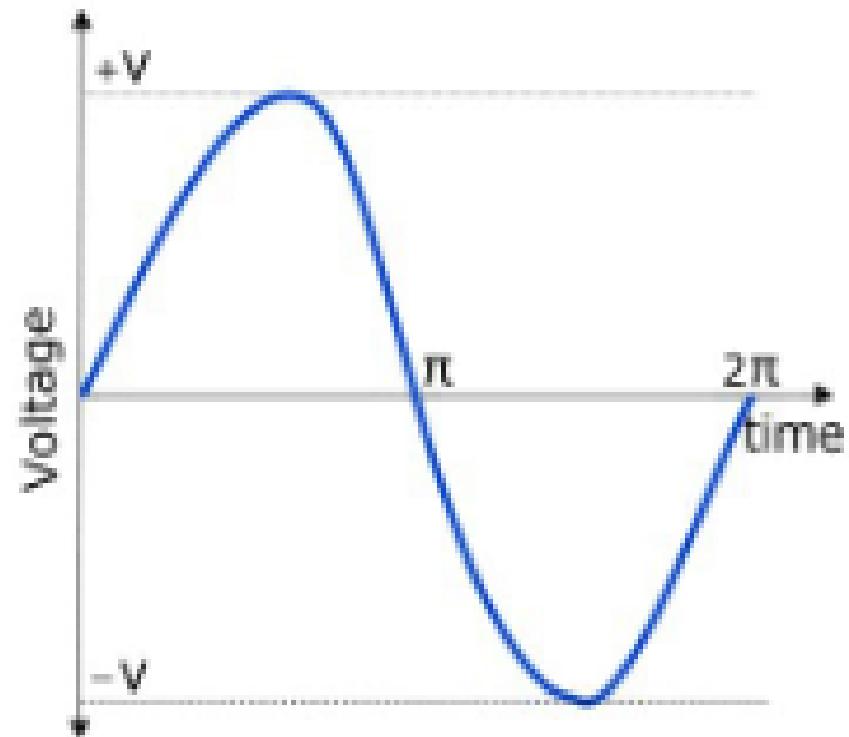
Digital electronics is the branch of electronics which is based on the combination and switching of voltages called logic levels.



Electronics circuits and systems are of two kinds - *analog and digital*.

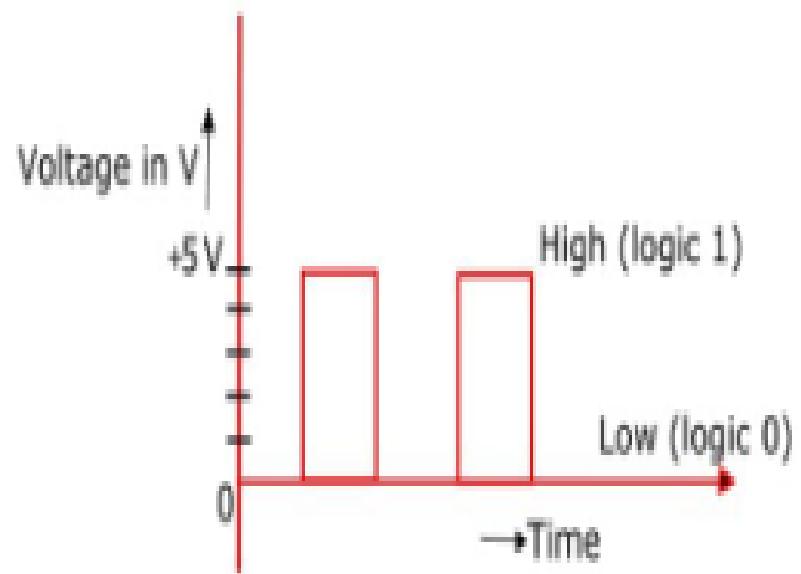
What is Analog circuits ?

- Analog circuits are those in which voltages and currents vary continuously through the given range. They can take infinite values within the specified range.
- For example, the output voltage from an audio amplifier might be any one of the infinite values between -10 V and $+10\text{ V}$ at any particular instant of time.



What is Digital Circuit?

- A digital circuit is one in which the voltage levels assume a finite number of distinct values. There are just discrete voltage levels. However, each voltage level in a particle digital system can be a narrow *band* or *range* of voltages.
- Digital circuits are often called **switching circuits** because the voltage levels in a digital circuit are assumed to be switched from one value to another instantaneously, i.e. the transition time is assumed to be zero.
- Digital circuits are also called **logic circuits** because each type of digital circuit obeys a certain set of logic rules. The manner in which a logic circuit responds to input is referred to as the circuit's logic.



Advantages of digital systems

- Digital systems are easier to design.
- Information storage is easy.
- Accuracy and precision are greater.
- Digital systems are more versatile.
- Digital circuits are less affected by noise.
- More digital circuitry can be fabricated on IC chips.

What is number System?

- A number system is defined as the technique of writing to express numbers.
- The number system has different bases and the most common of them are the decimal, binary, octal, and hexadecimal.
- The **base or radix** of the number system is the total number of the digit used in the number system.

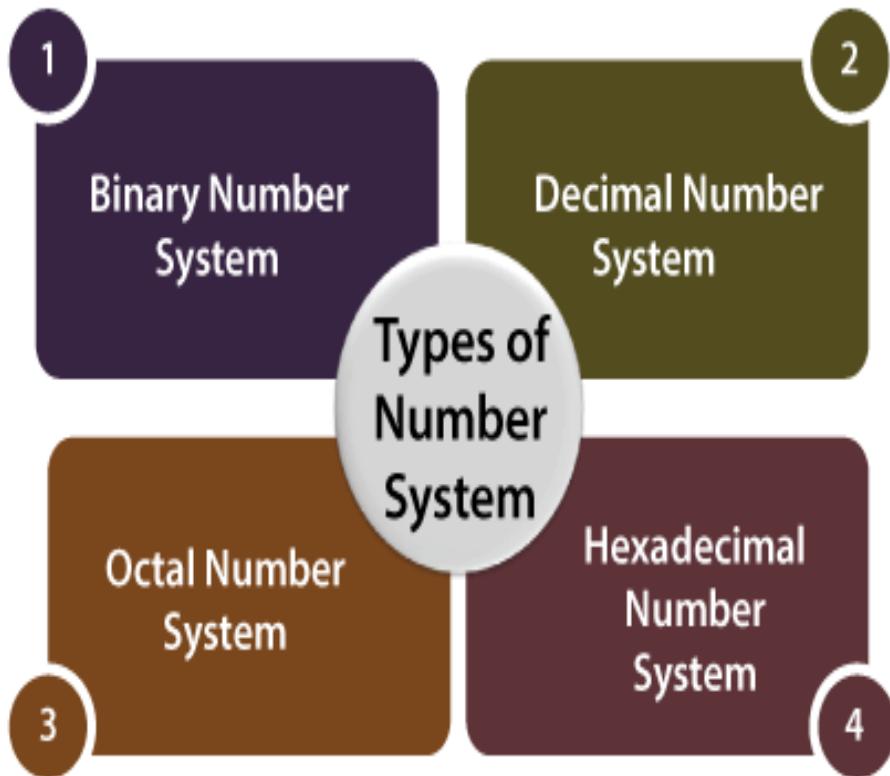
What is the need of Number System?

- **Computers** understand **machine language**. Every letter, symbol etc. that we write in the instructions given to computer, it gets converted into machine language.
- This machine language comprises of numbers. In order to understand the language used by computers and other digital system it is crucial to have a better understanding of number system.

Types of Number System

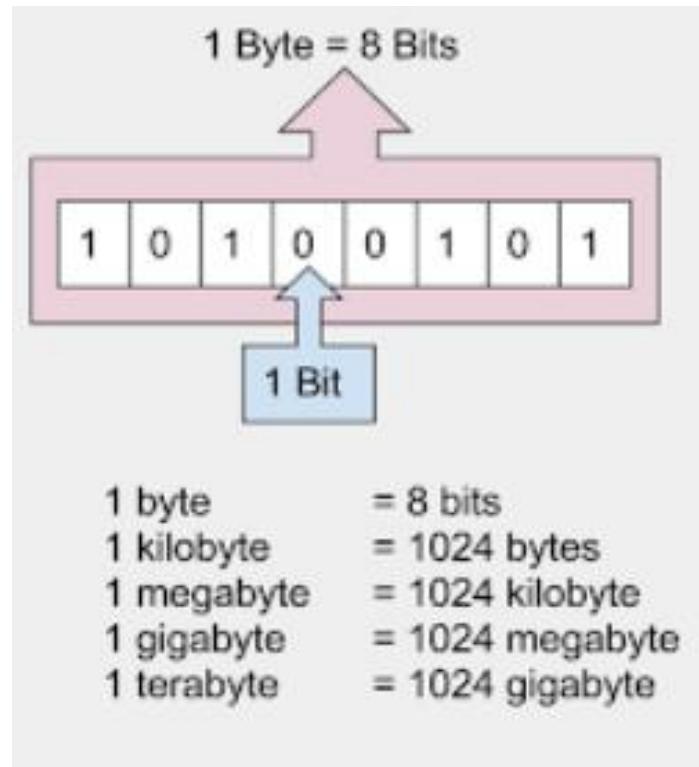
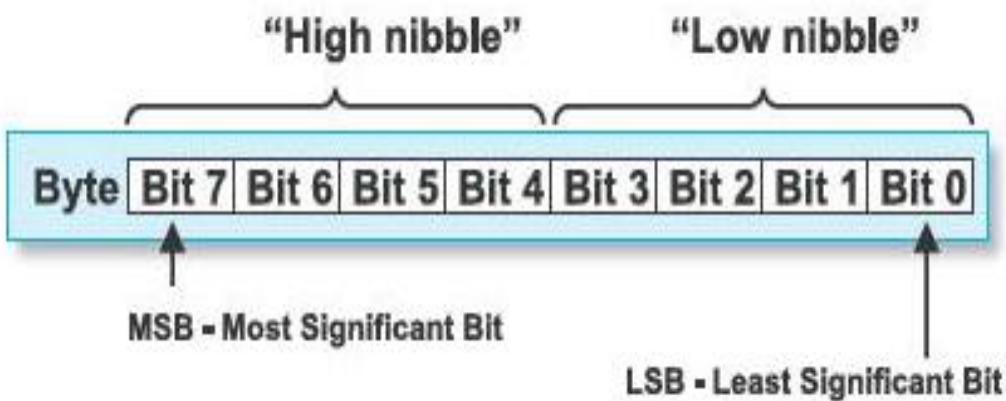
- There are various types of number systems used for representing information.
- The number systems are of following types.

1. **Binary Number System**
2. **Decimal Number System**
3. **Octal Number System**
4. **Hexadecimal Number System**



1. Binary Number System

- Generally, a binary number system is used in the **digital computers.**
- In this number system, it carries only two digits, either 0 or 1. So the **base or radix of the number system is 2.**
- There are two types of electronic pulses present in a binary number system. The first one is the absence of an electronic pulse representing '0' and second one is the presence of electronic pulse representing '1'.
- Each digit is known as a **bit**.
- A four-bit collection (1101) is known as a **nibble**.
- A collection of eight bits (11001010) is known as a **byte**.



Examples

- a) $(10101)_2$
- b) $(11011.1101)_2$
- c) $(11001.11)_2$
- d) $(0.101)_2$
- e) $(110.1)_2$



2. Decimal Number System

- The decimal numbers are used in our **day to day life.**
- The decimal number system contains ten digits and these digits are 0, 1, 2, 3, 4, 5, 6, 7, 8 & 9.
- The **base or radix of the number system is 10** because total 10 digits are available in the number system.

Examples:

- a) $(2546)_{10}$ b) $(36.46)_{10}$ c) $(4.567)_{10}$ d) $(984.76)_{10}$

3. Octal Number System

- The octal number system is a number system which uses eight digits to express any number.
- The digits used are 0, 1, 2, 3, 4, 5, 6 & 7.
- The base of the octal number system or radix is 8. This is because the total number of digits in the number system is 8.

Examples:

- a) $(765)_8$ b) $(23.46)_8$ c) $(67.4)_8$ d) $(2.546)_8$

4. Hexadecimal Number System

- It is another technique to represent the number in the digital system called the **hexadecimal number system**.
- The number system has a base of 16 means there are total 16 symbols(0, 1, 2, 3, 4, 5, 6, 7, 8, 9, A, B, C, D, E, F) used for representing a number.
- The single-bit representation of decimal values 10, 11, 12, 13, 14, and 15 are represented by A, B, C, D, E, and F.
- Only 4 bits are required for representing a number in a hexadecimal number.

Examples:

a) $(2C.4F)_{16}$

b) $(7AD.9E)_{16}$

c) $(81B.6F)_{16}$

Decimal to Binary Conversion

Example: Convert the decimal number 43 into its binary equivalent.

| | | Remainder | |
|---|----|-----------|--|
| 2 | 43 | | |
| 2 | 21 | 1 | |
| 2 | 10 | 1 | |
| 2 | 5 | 0 | |
| 2 | 2 | 1 | |
| 2 | 1 | 0 | |
| | 0 | 1 | |

LSB
↑
MSB

The remainders are to be read from bottom to top to obtain the binary equivalent.

$$43_{10} = \underline{101011}_2$$

Example: Convert the decimal number 10.25 in to its binary equivalent.

$(10.25)_{10}$

Integer part :

| | | |
|---|----|---|
| 2 | 10 | 0 |
| 2 | 5 | 1 |
| 2 | 2 | 0 |
| 2 | 1 | |
| | | |

$$(10)_{10} = (1010)_2$$

Fractional part

$$\begin{array}{l} 0.25 \times 2 = 0.50 \\ 0.50 \times 2 = 1.00 \end{array}$$

$$(0.25)_{10} = (0.01)_2$$

Note: Keep multiplying the fractional part with 2 until decimal part 0.00 is obtained.

$$(0.25)_{10} = (0.01)_2$$

Answer: $(10.25)_{10} = (1010.01)_2$

Decimal to Octal Conversion

Example: Convert the decimal number 473 into its octal equivalent.

| | | Remainder |
|---|-----|-----------|
| 8 | 473 | |
| 8 | 59 | 1 |
| 8 | 7 | 3 |
| | 0 | 7 |

LSD
↑
MSD

Reading the remainders from bottom to top,

$$473_{10} = 731_8$$

Example: Convert $(0.513)_{10}$ to octal.

$$0.513 \times 8 = 4.104$$

$$0.104 \times 8 = 0.832$$

$$0.832 \times 8 = 6.656$$

$$0.656 \times 8 = 5.248$$

$$0.248 \times 8 = 1.984$$

$$0.984 \times 8 = 7.872$$

The answer, to seven significant figures, is obtained from the integer part of the products:

$$(0.513)_{10} = (0.406517 \dots)_8$$

Decimal to Hexadecimal Conversion

Example: Convert the decimal number 423 into its hexadecimal equivalent .

| | | Remainder |
|----|-----|-----------|
| 16 | 423 | |
| 16 | 26 | 7 |
| 16 | 1 | A |
| | 0 | 1 |

Reading the remainders from bottom to top we get,

$$423_{10} = 1A7_{16}$$

Example: Convert the $(2598.675)_{10}$ to hexadecimal

The given decimal number is a mixed number. Convert the integer and the fraction parts separately to hex.

Conversion of 2598_{10}

| Successive division | Remainder | |
|---------------------|-----------|-----|
| | Decimal | Hex |
| 16 2598 | | |
| 16 162 | 6 | ↑ 6 |
| 16 10 | 2 | 2 |
| 0 | 10 | A |

Reading the remainders upwards, $2598_{10} = A26_{16}$.

Conversion of 0.675_{10}

Given fraction is 0.675

| | |
|-------------------|--------|
| 0.675×16 | 10.8 |
| 0.800×16 | 12.8 |
| 0.800×16 | 12.8 |
| 0.800×16 | ↓ 12.8 |

Reading the integers to the left of hexadecimal point downwards, $0.675_{10} = 0.ACCC_{16}$.

Therefore, $2598.675_{10} = A26.ACCC_{16}$.

Binary to Decimal Conversion

Example: Convert the binary number 11010 in to its decimal equivalent.

$$\begin{aligned}11010_2 &= 1 \times 2^4 + 1 \times 2^3 + 0 \times 2^2 + 1 \times 2^1 + 0 \times 2^0 \\&= 16 + 8 + 0 + 2 + 0 \\&= 26_{10}\end{aligned}$$

Example: Convert the binary number 1010.01 in to its decimal equivalent.

$$(1010.01)_2$$

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

$$= 8 + 0 + 2 + 0 + 0 + 0.25$$

$$= 10.25$$

$$(1010.01)_2 = (10.25)_{10}$$

Octal to Decimal Conversion

Example: Convert the octal number 726 in to its decimal equivalent.

$$\begin{aligned}726_8 &= 7 \times 8^2 + 2 \times 8^1 + 6 \times 8^0 \\&= 448 + 16 + 6 \\&= 470_{10}\end{aligned}$$

Example: Convert the octal number 12.2 in to its decimal equivalent.

$$(12.2)_8$$

$$= 1 \times 8^1 + 2 \times 8^0 + 2 \times 8^{-1}$$

$$= 8 + 2 + 0.25 = 10.25$$

$$(12.2)_8 = (10.25)_{10}$$

Hexadecimal to Decimal Conversion

Example: Convert the hexadecimal number 27FA in to its decimal equivalent.

$$\begin{aligned}27FA_{16} &= 2 \times 16^3 + 7 \times 16^2 + 15 \times 16^1 + 10 \times 16^0 \\&= 8192 + 1792 + 240 + 10 \\&= 10234_{10}\end{aligned}$$

Example: Convert $(A0F9.0EB)_{16}$ to decimal.

$$\begin{aligned} A0F9.0EB_{16} &= (10 \times 16^3) + (0 \times 16^2) + (15 \times 16^1) + (9 \times 16^0) \\ &\quad + (0 \times 16^{-1}) + (14 \times 16^{-2}) + (11 \times 16^{-3}) \end{aligned}$$

$$= 40960 + 0 + 240 + 9 + 0 + 0.0546 + 0.0026$$

$$= 41209.0572_{10}$$

Binary to octal and vice versa

| Octal Number | Binary Equivalent |
|--------------|-------------------|
| 0 | 000 |
| 1 | 001 |
| 2 | 010 |
| 3 | 011 |
| 4 | 100 |
| 5 | 101 |
| 6 | 110 |
| 7 | 111 |

Example: Convert the binary number 1101010 in to its octal equivalent.

$$1101010_2 = ?_8$$

Step 1: Divide the binary digits into groups of 3 starting from right

001 101 010

Step 2: Convert each group into one octal digit

$$001_2 = 0 \times 2^2 + 0 \times 2^1 + 1 \times 2^0 = 1$$

$$101_2 = 1 \times 2^2 + 0 \times 2^1 + 1 \times 2^0 = 5$$

$$010_2 = 0 \times 2^2 + 1 \times 2^1 + 0 \times 2^0 = 2$$

Hence, $1101010_2 = 152_8$

Example: Convert the octal number 562 in to its binary equivalent.

$$562_8 = ?_2$$

Step 1: Convert each octal digit to 3 binary digits

$$5_8 = 101_2, \quad 6_8 = 110_2, \quad 2_8 = 010_2$$

Step 2: Combine the binary groups

$$562_8 = \underline{101} \quad \underline{110} \quad \underline{010}$$

5 6 2

Hence, $562_8 = 101110010_2$

Binary to Hexadecimal and vice versa

| Binary equivalent | Hexadecimal |
|--------------------------|--------------------|
| 0000 | 0 |
| 0001 | 1 |
| 0010 | 2 |
| 0011 | 3 |
| 0100 | 4 |
| 0101 | 5 |
| 0110 | 6 |
| 0111 | 7 |
| 1000 | 8 |
| 1001 | 9 |
| 1010 | A |
| 1011 | B |
| 1100 | C |
| 1101 | D |
| 1110 | E |
| 1111 | F |

Example: Convert the binary number 111101₂ in to its hexadecimal equivalent.

$$111101_2 = ?_{16}$$

Step 1: Divide the binary digits into groups of four starting from the right

$$\underline{0011} \quad \underline{1101}$$

Step 2: Convert each group into a hexadecimal digit

$$0011_2 = 0 \times 2^3 + 0 \times 2^2 + 1 \times 2^1 + 1 \times 2^0 = 3_{10} = 3_{16}$$

$$1101_2 = 1 \times 2^3 + 1 \times 2^2 + 0 \times 2^1 + 1 \times 2^0 = 13_{10} = D_{16}$$

Hence, $111101_2 = 3D_{16}$

Example: Convert the hexadecimal number 2AB in to its binary equivalent.

$$2AB_{16} = ?_2$$

Step 1: Convert each hexadecimal digit to a 4 digit binary number

$$2_{16} = 2_{10} = 0010_2$$

$$A_{16} = 10_{10} = 1010_2$$

$$B_{16} = 11_{10} = 1011_2$$

Step 2: Combine the binary groups

$$2AB_{16} = \underline{0010} \quad \underline{1010} \quad \underline{1011}$$

2 A B

$$\text{Hence, } 2AB_{16} = 001010101011_2$$

Octal to Hexadecimal Conversion

- To convert an octal number to hexadecimal, the simplest way is to first convert to binary and then binary to hexadecimal.

Example: Convert 756.603_8 to hexadecimal.

Given octal number is

7 5 6 . 6 0 3

Convert each octal digit to binary

111 101 110 . 110 000 011

Groups of four bits are

0001 1110 1110 . 1100 0001 1000

Convert each four-bit group to hex

1 E E . C 1 8

The result is

IEE.C18₁₆

Hexadecimal to Octal Conversion

- To convert a hexadecimal number to octal , the simplest way is to first convert to binary and then binary to octal.

Example: Convert $B9F.AE_{16}$ to octal.

Given hex number is

B 9 F . A E

Convert each hex digit to binary

1011 1001 1111 . 1010 1110

Groups of three bits are

101 110 011 111 . 101 011 100

Convert each three-bit group to octal

5 6 3 7 . 5 3 4

The result is

5637.534

MISCELLANEOUS EXAMPLES

Example:

Given that $16_{10} = 100_b$, find the value of b.

Solution

Given $16_{10} = 100_b$

Convert 100_b to decimal.

$$\text{Therefore, } 16 = 1 \times b^2 + 0 \times b^1 + 0 \times b^0$$

$$\text{or } 16 = b^2$$

$$\text{or } b = 4$$

Example: What should be the radix of the number used in the following addition?

$$34 + 15 = 50$$

Solution

Given $34 + 15 = 50$

or $(34)_r + (15)_r = (50)_r$

or $3 \times r^1 + 4 \times r^0 + 1 \times r^1 + 5 \times r^0 = 5 \times r^1 + 0 \times r^0$

or $3r + 4 + 1r + 5 = 5r + 0$

or $4r + 9 = 5r$

or $5r - 4r = 9$

or $r = 9$

Example: Determine the possible base in the following arithmetic operation.

$$\frac{41}{3} = 13$$

Solution

$$\frac{41}{3} = 13$$

$$\frac{4 \times b^1 + 1 \times b^0}{3 \times b^0} = 1 \times b^1 + 3 \times b^0$$

$$\frac{4b + 1}{3} = b + 3$$

$$4b + 1 = 3b + 9$$

$$4b - 3b = 9 - 1$$

$$b = 8$$

DIGITAL ELECTRONICS

III Semester CSE & CST



Mr. Srikanta Patnaik

Mr. Ajit Kumar Patro

Mr. Ami Kumar Parida

DEPARTMENT OF ELECTRONICS AND COMMUNICATION ENGINEERING

SCHOOL OF ENGINEERING & TECHNOLOGY

GIET UNIVERSITY, GUNUPUR, ODISHA

Representation of Signed Binary Number

1's Complement

- The 1's complement of any binary number is performed by simply changing all 1's into 0's and 0's into 1's, i.e. each bit is replaced by its complement.

Binary number

1011

010101

1100.01

10010.110

1's complement

0100

101010

0011.10

01101.001

2's Complement

- The 2's complement of any binary number is determined by adding 1 to 1's complement of that number.
- 2's complement of number =
1's complement of a number + 1

Example:

Find the 2's complement of the $(1010)_2$

Solution

Given binary number = 1010

1's complement of 1010 = 0101

$$\begin{array}{r} +1 \\ \hline 0110 \end{array}$$

2's complement of 1010 = 0110

Example:

Find the 2's complement of the $(11.01)_2$

Solution

Given binary number = 11.01

1's complement of 11.01 = 00.10

$$\begin{array}{r} +1 \\ \hline 00.11 \end{array}$$

2's complement of 11.01 = 00.11

Binary number: 1010 1111 11.01

2's complement: 0110 0001 00.11

Note: The 2's complement of a number can be obtained by leaving all least significant 0's and the first one (1) unchanged, and replacing all 1's with 0's and 0's with 1's in all higher significant bits.

REPRESENTATION OF SIGNED NUMBER

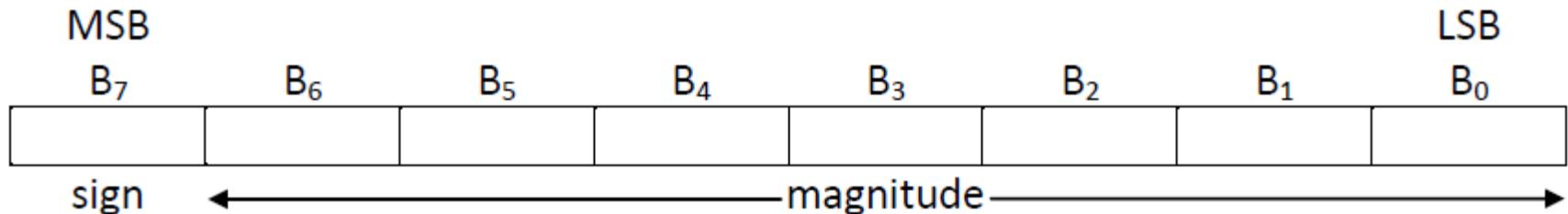
- There are two ways of representing signed numbers
 - Sign-magnitude form
 - Complement form

There are two complement forms:

- 1's complement form
- 2's complement form

Sign-magnitude Representation

- The sign-magnitude format for 8-bit signed number



- Here, the most significant bit (MSB) represents sign of the number.
- If MSB is 1, number is negative and if MSB is 0, number is positive.
- The remaining (seven) bits represent magnitude of the number.

Example: Represent the following signed number in sign-magnitude form.

(a) 25

(b) - 25

(c) 68

(d) - 68

B₇ B₆ B₅ B₄ B₃ B₂ B₁ B₀

25

| | | | | | | | |
|---|---|---|---|---|---|---|---|
| 0 | 0 | 0 | 1 | 1 | 0 | 0 | 1 |
|---|---|---|---|---|---|---|---|

- 25

| | | | | | | | |
|---|---|---|---|---|---|---|---|
| 1 | 0 | 0 | 1 | 1 | 0 | 0 | 1 |
|---|---|---|---|---|---|---|---|

68

| | | | | | | | |
|---|---|---|---|---|---|---|---|
| 0 | 1 | 0 | 0 | 0 | 1 | 0 | 0 |
|---|---|---|---|---|---|---|---|

- 68

| | | | | | | | |
|---|---|---|---|---|---|---|---|
| 1 | 1 | 0 | 0 | 0 | 1 | 0 | 0 |
|---|---|---|---|---|---|---|---|

NOTE: The actual range of numbers we can represent in sign-magnitude format for n-bit would be

$$- (2^{n-1} - 1) \text{ to } + (2^{n-1} - 1)$$

Example:

If we have **8 bits** to represent a signed binary number, (1-bit for the **sign bit** and 7-bits for the **magnitude bits**), then the actual range of numbers we can represent in sign-magnitude format would be:

$$- (2^{8-1} - 1) \text{ to } + (2^{8-1} - 1)$$

$$- (2^7 - 1) \text{ to } + (2^7 - 1)$$

$$- 127 \text{ to } + 127$$

1's complement Representation

- If the number is positive, the magnitude is represented in its true binary form and a sign bit 0 is placed in front of the MSB.
- If the number is negative, the magnitude is represented in its 1's complement form and a sign bit 1 is placed in front of the MSB.

Example: Represent the following signed number in 1's complement form.

(a) 25

(b) - 25

(c) 68

(d) - 68

B₇ B₆ B₅ B₄ B₃ B₂ B₁ B₀

| | | | | | | | | |
|----|---|---|---|---|---|---|---|---|
| 25 | 0 | 0 | 0 | 1 | 1 | 0 | 0 | 1 |
|----|---|---|---|---|---|---|---|---|

| | | | | | | | | |
|------|---|---|---|---|---|---|---|---|
| - 25 | 1 | 1 | 1 | 0 | 0 | 1 | 1 | 0 |
|------|---|---|---|---|---|---|---|---|

| | | | | | | | | |
|----|---|---|---|---|---|---|---|---|
| 68 | 0 | 1 | 0 | 0 | 0 | 1 | 0 | 0 |
|----|---|---|---|---|---|---|---|---|

| | | | | | | | | |
|------|---|---|---|---|---|---|---|---|
| - 68 | 1 | 0 | 1 | 1 | 1 | 0 | 1 | 1 |
|------|---|---|---|---|---|---|---|---|

2's complement Representation

- If the number is positive, the magnitude is represented in its true binary form and a sign bit 0 is placed in front of the MSB.
- If the number is negative, the magnitude is represented in its 2's complement form and a sign bit 1 is placed in front of the MSB.

Example: Represent the following signed number in 2's complement form.

- (a) 25 (b) - 25 (c) 68 (d) - 68

B₇ B₆ B₅ B₄ B₃ B₂ B₁ B₀

| | | | | | | | | |
|----|---|---|---|---|---|---|---|---|
| 25 | 0 | 0 | 0 | 1 | 1 | 0 | 0 | 1 |
|----|---|---|---|---|---|---|---|---|

| | | | | | | | | |
|------|---|---|---|---|---|---|---|---|
| - 25 | 1 | 1 | 1 | 0 | 0 | 1 | 1 | 1 |
|------|---|---|---|---|---|---|---|---|

| | | | | | | | | |
|----|---|---|---|---|---|---|---|---|
| 68 | 0 | 1 | 0 | 0 | 0 | 1 | 0 | 0 |
|----|---|---|---|---|---|---|---|---|

| | | | | | | | | |
|------|---|---|---|---|---|---|---|---|
| - 68 | 1 | 0 | 1 | 1 | 1 | 1 | 0 | 0 |
|------|---|---|---|---|---|---|---|---|

Example: Each of the following numbers is a signed binary number. Determine the decimal value in each case, if they are in (i) sign-magnitude form, (ii) 2's complement form, and (iii) 1's complement form.

- (a) 01101 (b) 010111 (c) 10111 (d) 1101010

Solution:

| Given number | Sign-magnitude form | 2's complement form | 1's complement form |
|--------------|---------------------|---------------------|---------------------|
| 01101 | + 13 | + 13 | + 13 |
| 010111 | + 23 | + 23 | + 23 |
| 10111 | - 7 | - 9 | - 8 |
| 1101010 | - 42 | - 22 | - 21 |

Binary Arithmetic's

1. Binary Addition

The rules of binary addition are the following:

$$0 + 0 = 0$$

$$0 + 1 = 1$$

$$1 + 0 = 1$$

$$1 + 1 = 10, \text{i.e. } 0 \text{ with a carry of } 1$$

Example: Add the binary numbers
1101.101 and 111.011

Solution

$$\begin{array}{r} \text{1 1 1 1 1} & \text{1 1} \\ \text{1 1 0 1 . 1} & \text{0 1} \\ + \text{1 1 1 . 0 1 1} & \\ \hline \text{1 0 1 0 1 . 0 0 0} & \end{array}$$

carry

2. Binary Subtraction

The rules of binary subtraction are the following:

$$0 - 0 = 0$$

$$0 - 1 = 1, \text{ with a borrow of 1}$$

$$1 - 0 = 1$$

$$1 - 1 = 0$$

Example: Subtract the binary number
101 from 1011

Solution

$$\begin{array}{r} & \text{(borrow)} \\ & 0\ 1 \\ & 1\ \cancel{0}\ 1\ 1 \\ - & 1\ 0\ 1 \\ \hline & 0\ 1\ 1\ 0 \end{array}$$

3. Binary Multiplication

The rules of binary multiplication are the following:

$$0 \times 0 = 0$$

$$0 \times 1 = 0$$

$$1 \times 0 = 0$$

$$1 \times 1 = 1$$

Example: Multiply the binary number 1101 by 110

Solution

$$\begin{array}{r} 1101 \\ \times 110 \\ \hline 0000 \\ 1101 \\ 1101 \\ \hline 1001110 \end{array}$$

4. Binary Division

Example: Divide the binary number 11010 by 101

Solution

$$\begin{array}{r} 101 \xrightarrow{\text{Quotient}} \\ 101) 11010 \\ - 101 \\ \hline 11 \\ - 00 \\ \hline 110 \\ - 101 \\ \hline 1 \end{array}$$

Remainder

$\begin{array}{r} 5 \xleftarrow{\text{Quotient}} \\ 5) 26 \\ - 25 \\ \hline 1 \end{array}$

Rules of Binary Arithmetic

| | Addition | Subtraction | Multiplication | Division |
|------|--------------|--|------------------|--|
| i) | $0 + 0 = 0$ | $0 - 0 = 0$ | $0 \times 0 = 0$ | $0 / 1 = 0$ |
| ii) | $0 + 1 = 1$ | $1 - 0 = 1$ | $0 \times 1 = 0$ | $1 / 1 = 1$ |
| iii) | $1 + 0 = 1$ | $1 - 1 = 0$ | $1 \times 0 = 0$ | $0 / 0 = \text{not allowed}$ (not valid) |
| iv) | $1 + 1 = 10$ | $1 - 0 = 10 - 1$ $(\text{with borrow } 1) =$ 1 | $1 \times 1 = 1$ | $1 / 0 = \text{not allowed}$ (not valid) |

Binary Arithmetic using 1's and 2's Complements

Logic for Binary Arithmetic

The binary arithmetic operation can be written as:

$$A + B = A + B$$

$$A - B = A + (-B)$$

$$-A + B = (-A) + B$$

$$-A - B = (-A) + (-B)$$

The negative binary number can be represented either 1's or 2's complement form.

Binary Arithmetic using 1's Complements

Example-1: Subtract 14 from 25 using 1's complement method.

Solution

$$\begin{array}{r} 25 \\ - 14 \Rightarrow +11110001 \quad (\text{In 1's complement form}) \\ \hline +11 \end{array}$$

 + 1 (End around carry)

$$\hline 00001011$$

The MSB is a 0. So, the result is positive and is in pure binary. Therefore, the result is,
 $00001011 = +11_{10}$

Example-2: Add -25 to 14 using 1's complement method.

Solution

$$\begin{array}{r} +14 & 00001110 \\ -25 \Rightarrow +11100110 & \text{(In 1's complement form)} \\ \hline -11 & 11110100 \quad \text{(No carry)} \end{array}$$

There is no carry. The MSB is a 1. So, the result is negative and is in its 1's complement form. The 1's complement of 11110100 is 00001011. The result is, therefore, -11_{10}

Example-3: Add -14 to -25 using 1's complement method.

Solution

$$\begin{array}{r} \begin{array}{r} -25 \\ -14 \\ \hline \end{array} & \begin{array}{r} 11100110 \\ +11110001 \\ \hline \end{array} & \text{(In 1's complement form)} \\ \Rightarrow & \hline & \text{(In 1's complement form)} \\ \begin{array}{r} -39 \\ 011010111 \\ \hline \end{array} & \begin{array}{r} \\ \downarrow \\ +1 \end{array} & \text{(End around carry)} \\ \hline & 11011000 & \end{array}$$

The MSB is a 1. So, the result is negative and is in its 1's complement form. The 1's complement of 11011000 is 00100111. Therefore, the result is -39.

Example-4: Add 14 to 25 using 1's complement method.

Solution

$$\begin{array}{r} +25 & 00011001 \text{ (In 1's complement form)} \\ +14 \Rightarrow +00001110 \text{ (In 1's complement form)} \\ \hline +39 & 00100111 \end{array}$$

There is no carry. The MSB is a 0. So, the result is positive and is in pure binary. Therefore, the result is, $00100111 = +39$.

Binary Arithmetic using 2's Complements

Example-1: Subtract 14 from 46 using 2's complement method.

Solution

$$\begin{array}{rcl} +14 & = & 00001110 \\ -14 & = & 11110010 \quad (\text{In 2's complement form}) \end{array}$$

$$\begin{array}{rcl} +46 & & 00101110 \\ -14 & \Rightarrow & +11110010 \quad (\text{2's complement form of } -14) \\ \hline +32 & & 00100000 \quad (\text{Ignore the carry}) \end{array}$$

There is a carry, ignore it. The MSB is 0. So, the result is positive and is in normal binary form. Therefore, the result is $+00100000 = +32$.

Example-2: Add -75 to 26 using 2's complement method.

Solution

$$\begin{array}{rcl} +75 & = & 01001011 \\ -75 & = & 10110101 \quad (\text{In 2's complement form}) \\ \\ +26 & & 00011010 \\ -75 & \Rightarrow & \underline{+10110101} \quad (2\text{'s complement form of } -75) \\ \hline -49 & & 11001111 \quad (\text{No carry}) \end{array}$$

There is no carry, the MSB is a 1. So, the result is negative and is in 2's complement form. The magnitude is 2's complement of 11001111, that is, $00110001 = 49$. Therefore, the result is -49.

Example-3: Add -14 to -25 using 2's complement method.

Solution

$$+14 = 00001110$$

$$-14 = 11110010 \text{ (In 2's complement form)}$$

$$+25 = 00011001$$

$$-25 = 11100111 \text{ (In 2's complement form)}$$

$$\begin{array}{r} -14 \\ +25 \\ \hline \end{array} \quad 11110010 \text{ (2's complement form of -14)}$$

$$\begin{array}{r} -25 \\ +25 \\ \hline \end{array} \Rightarrow \begin{array}{r} 11100111 \\ 111011001 \\ \hline \end{array} \quad \begin{array}{l} \text{(2's complement form of -25)} \\ \text{(Ignore the carry)} \end{array}$$

There is a carry, ignore it. The MSB is a 1; so the result is negative and is in 2's complement form. The 2's complement of 11011001 is 00100111. Therefore, the result is -39.

Floating-Point Number Representation

- Floating-Point is useful for representing a number in a **wide range**: **very small** to **very large**.
- It is widely used in the scientific world.

- There are many ways to write a number in scientific notation, but there is always a unique **normalized** representation, with exactly one non-zero digit to the left of the point.

Example:

$$257 = 25.7 \times 10^1 = \textcolor{red}{2.57 \times 10^2} = \dots$$

$$0.232 \times 10^3 = \textcolor{red}{2.32 \times 10^2} = 23.2 \times 10^1 = \dots$$

$$01001 = \textcolor{red}{1.001 \times 2^3} = \dots$$

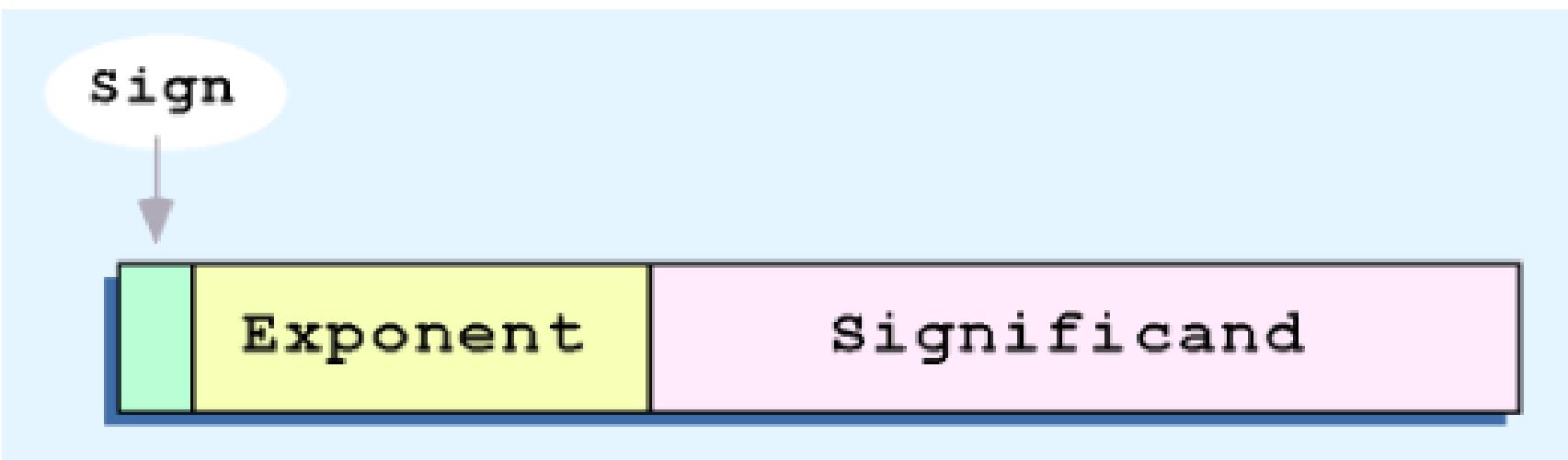
- What's the normalized representation of 00101101.101 ?

$$00101101.101 = 1.01101101 \times 2^5$$

- What's the normalized representation of 0.0001101001110 ?

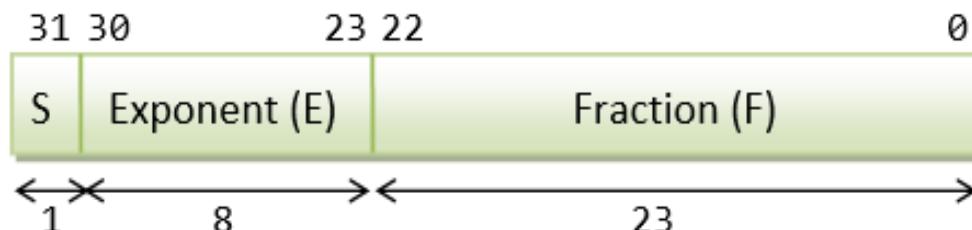
$$0.0001101001110 = 1.101001110 \times 2^{-4}$$

- We can represent floating-point numbers with three binary fields:
 - a sign bit S,
 - an exponent field E, and
 - A significand (mantissa) / fraction field F.



- The IEEE 754 standard defines several different precisions.
 - Single precision (32 bits) [1 + 8 + 23]
 - Double precision (64 bits) [1 + 11 + 52]
 - Extended precision (80 bits) [1 + 15 + 64]

- Single precision numbers include a 1-bit sign, an 8-bit exponent and a 23-bit fraction, for a total of 32 bits.



- Double precision numbers have a 1-bit sign, an 11-bit exponent and a 52-bit fraction, for a total of 64 bits.



Sign (S)

- The sign bit is
0 for positive numbers and
1 for negative numbers.

Exponent (E)

- The E field represents the exponent as a biased number.
- It contains the actual exponent *plus 127* for single precision, or the actual exponent *plus 1023* in double precision.
- This converts all single-precision exponents from -126 to +127 into unsigned numbers from 1 to 254, and all double-precision exponents from -1022 to +1023 into unsigned numbers from 1 to 2046.

Significand / Fraction (F)

- The field F contains a binary fraction.
- The actual mantissa of the floating-point value is (1 + F).
 - In other words, there is an implicit 1 to the left of the binary point.
 - For example, if F is 01101..., the mantissa would be 1.01101...

Example: What is the single-precision representation of 347.625?

- First convert the number to binary:

$$347.625 = 101011011.101_2$$

- Normalize the number by shifting the binary point until there is a single 1 to the left:

$$101011011.101 \times 2^0 = 1.\textcolor{violet}{01011011101} \times 2^8$$

- The bits to the right of the binary point comprise the fraction field F.
- The number of times you shifted gives the exponent.
- The field E should contain:

$$\text{exponent} + 127 = \textcolor{red}{8} + 127 = 135 = \textcolor{blue}{10000111}_2$$

- Sign bit: 0 if positive, 1 if negative.

0|**10000111**|**0101101101**000000000000

Example: Express -3.75 as a floating point number using IEEE single precision.

- First, let's normalize according to IEEE rules:

$$-3.75 = -11.11_2 = -1.\textcolor{purple}{111} \times 2^{\textcolor{red}{1}}$$

- Since it is a negative number , so S = **1**.
- The bias is 127, so we add $127 + \textcolor{red}{1} = 128$ (this is our exponent)

$$E = 128 = \textcolor{blue}{10000000}_2$$

1| **1000000|** **111**000000000000000000000000000000

Example: Find out the floating point number from the following representation.

0 | 10000011 | 00111000000000000000000000

0 | 10000011 | 00111000000000000000000000

$$E = 10000011_2 = 131$$

$$N = (-1)^S \times (1 + F) \times 2^{E-\text{bias}}$$

$$= (-1)^0 \times (1.00111000000000000000000000) \times 2^{131-127}$$

$$= 1 \times (1.00111000000000000000000000) \times 2^4$$

$$= 1.00111000000000000000000000 \times 2^4$$

$$= 10011.10$$

$$= 19.5$$

Example: Find out the floating point number from the following representation.

1 1000 0001 011 0000 0000 0000 0000

1 1000 0001 011 0000 0000 0000 0000

$$E = 1000000_2 = 129$$

$$N = (-1)^S \times (1 + F) \times 2^{E-\text{bias}}$$

$$= (-1)^1 \times (1.01100000000000000000000000) \times 2^{129-127}$$

$$= -1 \times (1.01100000000000000000000000) \times 2^2$$

$$= -1.01100000000000000000000000 \times 2^2$$

$$= -101.10$$

$$= -5.5$$

DIGITAL ELECTRONICS

III Semester CSE & CST



Mr. Srikanta Patnaik

Mr. Ajit Kumar Patro

Mr. Ami Kumar Parida

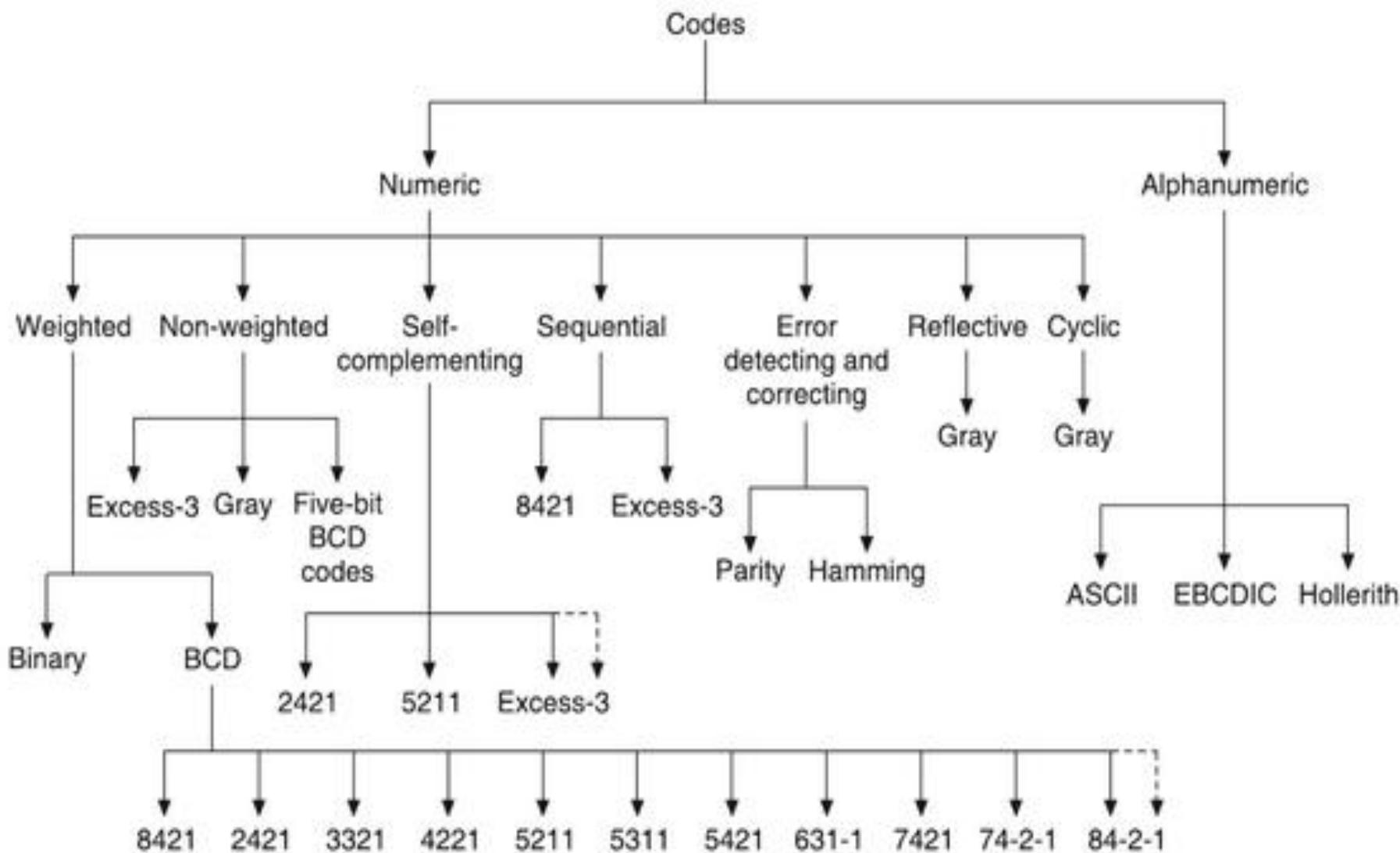
DEPARTMENT OF ELECTRONICS AND COMMUNICATION ENGINEERING

SCHOOL OF ENGINEERING & TECHNOLOGY

GIET UNIVERSITY, GUNUPUR, ODISHA

BINARY CODES

Classification of Binary Codes



1. Numeric and Alphanumeric Codes

- Binary codes can be classified as *numeric codes* and *alphanumeric codes*.
- Numeric codes are codes which represent numeric information, i.e. only numbers as a series of 0s and 1s.
8421, Excess-3, Gray code are numeric codes.
- Numeric codes used to represent the decimal digits are called Binary Coded Decimal (BCD) codes.
8421, 2421, 5211 are BCD codes.
- Alphanumeric codes are codes which represent alphanumeric information, i.e. letters of the alphabet and decimal numbers as a sequence of 0s and 1s.
EBCDIC code and ASCII code are alphanumeric codes.

2. Weighted and Non-weighted Codes

- The BCD codes may be *weighted codes* or *non-weighted codes*.
- The **weighted codes** are those which obey the position-weighting principal. Each position of the number represents a specific weight.
- For each group of four bits, the sum of the weights of those positions where the binary digits is 1 is equal to the decimal digit which the group represents.
8421, 2421, 84-2-1 are weighted codes.
- **Non-weighted codes** are codes which are not assigned with any weight to each digit position, i.e. each digit position within the number is not assigned fixed value.
Excess-3 code and Gray code are non-weighted codes.

3. Positively-weighted and Negatively-weighted Codes

- The weighted codes may be either *positively-weighted codes* or *negatively-weighted codes*.
- **Positively-weighted codes** are those in which all the weights assigned to the binary digits are positive.
- In every positively-weighted code, the first weight must be 1, the second weight must be either 1 or 2, and sum of all the weights must be equal to or greater than 9.

Example: 8421, 2421, 5211, 3321, 4321 codes.

- **Negatively-weighted codes** are those in which some of the weights assigned to the binary digits are negative.

Example: 642-3, 631-1, 84-2-1, 74-2-1 codes.

4. Error Detecting and Error Correcting Codes

- Binary codes may also be *error detecting codes* or *error correcting codes*.
- Codes which allow only error detection are called *error detecting codes*.

Shift counter code, 2-out-of-5, 63210 codes are error detecting codes.

- Codes which allow error detection as well as error correction are called *error correcting codes*.

The Hamming code is a error correcting code.

5. Sequential Codes

- Sequential code is one, in which each succeeding code word is one binary number greater than its preceding code word.

The 8421 and Excess-3 codes are sequential.

The codes 5211, 2421 and 642-3 are not sequential.

6. Self-complementing Codes

- A code is said to be **self-complementing**, if the code word of the 9's complement of N , i.e. of $9-N$ can be obtained from the code word of N by interchanging all the 0s and 1s.
- Therefore, in a self-complementing code, the code for 9 is the complement of the code for 0, the code for 8 is the complement of the code for 1, and so on.

The 2421, 5211, 642-3, 84-2-1 and Excess-3 codes are self-complementing codes.

The 8421 and 5421 codes are not self-complementing codes.

7. Cyclic Codes

- Cyclic codes are those in which each successive code word differs from the preceding one in only one bit position.
- They are also called unit distance codes.
The Gray code is a cyclic code.

8. Reflective Codes

- A **reflective code** is a binary code in which the n least significant bits for code words 2^n through $2^{n+1} - 1$ are the mirror images of those for 0 through $2^n - 1$.

The Gray code is a reflective code.

| Decimal digit | 8 4 2 1 | 2 4 2 1 | 5 2 1 1 | 5 4 2 1 | 6 4 2 -3 | 8 4 -2 -1 | XS-3 |
|--|----------------|----------------|----------------|----------------|-----------------|------------------|-------------|
| 0 | 0000 | 0000 | 0000 | 0000 | 0000 | 0000 | 0011 |
| 1 | 0001 | 0001 | 0001 | 0001 | 0101 | 0111 | 0100 |
| 2 | 0010 | 0010 | 0011 | 0010 | 0010 | 0110 | 0101 |
| 3 | 0011 | 0011 | 0101 | 0011 | 1001 | 0101 | 0110 |
| 4 | 0100 | 0100 | 0111 | 0100 | 0100 | 0100 | 0111 |
| 5 | 0101 | 1011 | 1000 | 1000 | 1011 | 1011 | 1000 |
| 6 | 0110 | 1100 | 1010 | 1001 | 0110 | 1010 | 1001 |
| 7 | 0111 | 1101 | 1100 | 1010 | 1101 | 1001 | 1010 |
| 8 | 1000 | 1110 | 1110 | 1011 | 1010 | 1000 | 1011 |
| 9 | 1001 | 1111 | 1111 | 1100 | 1111 | 1111 | 1100 |
| Unused bit combinations | 1010 | 0101 | 0010 | 0101 | 0001 | 0001 | 0000 |
| | 1011 | 0110 | 0100 | 0110 | 0011 | 0010 | 0001 |
| | 1100 | 0111 | 0110 | 0111 | 0111 | 0011 | 0010 |
| | 1101 | 1000 | 1001 | 1101 | 1000 | 1100 | 1101 |
| | 1110 | 1001 | 1011 | 1110 | 1100 | 1101 | 1110 |
| | 1111 | 1010 | 1101 | 1111 | 1110 | 1110 | 1111 |

The 8421 BCD Code

- The 8421 BCD code is so widely used that it is a common practice to refer to it simply as BCD code.
- In this code, each decimal digit, 0 through 9, is coded by a 4-bit binary number.
- It is also called the *natural binary code* because of the 8, 4, 2 and 1 weights attached to it.
- It is a weighted code and is also sequential. Therefore, it is useful for mathematical operations.

- The main advantage of this code is its ease of conversion to and from decimal.
- It is less efficient than the pure binary, in the sense that it requires more bits.

For example, the decimal number 14 can be represented as 1110 in pure binary but as 0001 0100 in 8421 code.
- Another disadvantage of the BCD code is that, arithmetic operations are more complex than they in pure binary.

The Gray Code

- The Gray code is a non-weighted code, and is not suitable for arithmetic operations. It is not a BCD code.
- It is a cyclic code because successive code words in this code differ in one bit position only, i.e. it is a unit distance code.
- It is also reflective code, i.e. it is both reflective and unit distance.

The n least significant bits for 2^n through $2^{n+1} - 1$ are the mirror images of those for 0 through $2^n - 1$.

- Gray codes are used in instrumentation and data acquisition systems where linear or angular displacement is measured.
- They are also used in shaft encoders, I/O devices, A/D converters and other peripheral equipment.

| Gray Code | | | | Decimal | 4-bit binary |
|-----------|-------|-------|-------|---------|--------------|
| 1-bit | 2-bit | 3-bit | 4-bit | | |
| 0 | 00 | 000 | 0000 | 0 | 0000 |
| 1 | 01 | 001 | 0001 | 1 | 0001 |
| | 11 | 011 | 0011 | 2 | 0010 |
| | 10 | 010 | 0010 | 3 | 0011 |
| | | 110 | 0110 | 4 | 0100 |
| | | 111 | 0111 | 5 | 0101 |
| | | 101 | 0101 | 6 | 0110 |
| | | 100 | 0100 | 7 | 0111 |
| | | | 1100 | 8 | 1000 |
| | | | 1101 | 9 | 1001 |
| | | | 1111 | 10 | 1010 |
| | | | 1110 | 11 | 1011 |
| | | | 1010 | 12 | 1100 |
| | | | 1011 | 13 | 1101 |
| | | | 1001 | 14 | 1110 |
| | | | 1000 | 15 | 1111 |

The ASCII Code

- The American Standard Code for Information Interchange (ASCII) is a widely used alphanumeric code.
- This is basically a 7-bit code. Since the number of different bit patterns that can be created with 7 bits is $2^7 = 128$, the ASCII can be used to encode both lowercase and uppercase characters of the alphabet (52 symbols) and some special symbols as well, in addition to the 10 decimal digits.
- It is used extensively for printers and terminals that interface with small computer systems.

MSBs

| | 000 | 001 | 010 | 011 | 100 | 101 | 110 | 111 |
|------|-----|-----|-------|-----|-----|-----|-----|-----|
| 0000 | NUL | DEL | Space | 0 | @ | P | p | |
| 0001 | SOH | DC1 | ! | 1 | A | Q | a | q |
| 0010 | STX | DC2 | " | 2 | B | R | b | r |
| 0011 | ETX | DC3 | # | 3 | C | S | c | s |
| 0100 | EOT | DC4 | \$ | 4 | D | T | d | t |
| 0101 | ENQ | NAK | % | 5 | E | U | e | u |
| 0110 | ACK | SYN | & | 6 | F | V | f | v |
| 0111 | BEL | ETB | ' | 7 | G | W | g | w |
| 1000 | BS | CAN | (| 8 | H | X | h | x |
| 1001 | HT | EM |) | 9 | I | Y | i | y |
| 1010 | LF | SUB | * | : | J | Z | j | z |
| 1011 | VT | ESC | + | ; | K | [| k | { |
| 1100 | FF | FS | , | < | L | \ | l | |
| 1101 | CR | GS | - | = | M |] | m | } |
| 1110 | SO | RS | . | > | N | ^ | n | ~ |
| 1111 | SI | US | / | ? | O | - | o | DLE |

Abbreviations

| | | | | | |
|-----|------------------|-----|---------------------------|-----|----------------------|
| ACK | Acknowledge | EM | End of medium | NAK | Negative acknowledge |
| BEL | Bell | ENQ | Enquiry | NUL | Null |
| BS | Backspace | EOT | End of transmission | RS | Record separator |
| CAN | Cancel | ESC | Escape | SI | Shift in |
| CR | Carriage return | ETB | End of transmission block | SO | Shift out |
| DC1 | Direct control 1 | ETX | End of text | SOH | Start of heading |
| DC2 | Direct control 2 | FF | Form feed | STX | Start text |
| DC3 | Direct control 3 | FS | Form separator | SUB | Substitute |
| DC4 | Direct control 4 | GS | Group separator | SYN | Synchronous idle |
| DEL | Delete idle | HT | Horizontal tab | US | Unit separator |
| DLE | Data link escape | LF | Line feed | VT | Vertical tab |

DIGITAL ELECTRONICS

III Semester CSE & CST



Mr. Srikanta Patnaik

Mr. Ajit Kumar Patro

Mr. Ami Kumar Parida

DEPARTMENT OF ELECTRONICS AND COMMUNICATION ENGINEERING

SCHOOL OF ENGINEERING & TECHNOLOGY

GIET UNIVERSITY, GUNUPUR, ODISHA

LOGIC LEVELS AND PULSE WAVEFORMS

LOGIC GATES

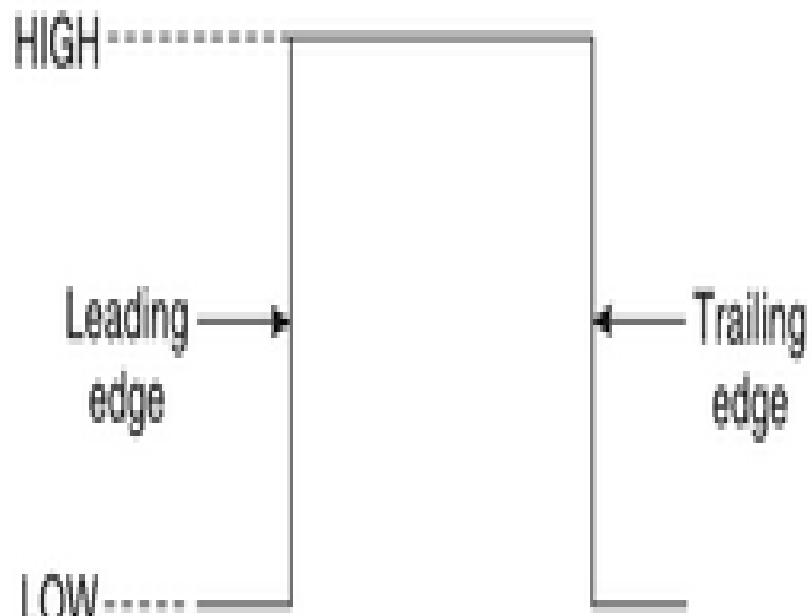
Logic Levels and Pulse Waveforms

- Digital systems use the binary number system. Therefore, two-state devices are used to represent the two binary digits 1 and 0 by two different voltage levels, called **HIGH** and **LOW**.
- If the HIGH voltage level is used to represent 1 and the LOW voltage level to represent 0, the system is called the *positive logic system*.
- If the HIGH voltage level is used to represent 0 and the LOW voltage level to represent 1, the system is called the *negative logic system*.

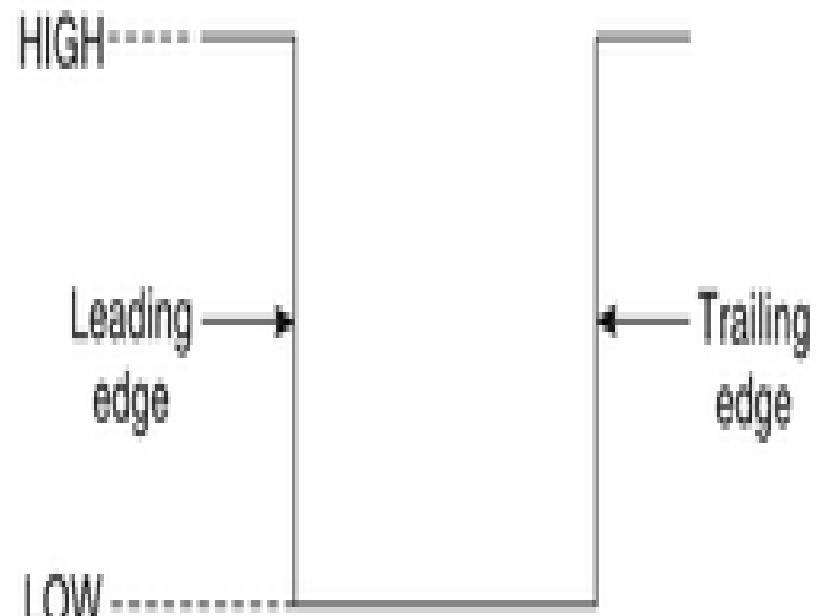
- Normally, the binary 0 and 1 are represented by the logic voltage levels 0 V and + 5 V.
- In a **positive logic system**, 1 is represented by + 5 V (HIGH) and 0 is represented by 0 V (LOW).
- In a **negative logic system**, 0 is represented by + 5 V (HIGH) and 1 is represented by 0 V (LOW).

- In reality, because of circuit variations, the 0 and 1 would be represented by voltage ranges instead of particular voltage levels.
- Usually, any voltage between 0 V and 0.8 V represents the **logic 0** and any voltage between 2 V and 5 V represents the **logic 1**.
- The range between 0.8 V and 2 V is called the *indeterminate range*. If the signal falls between 0.8 V and 2 V, the response is not predictable.

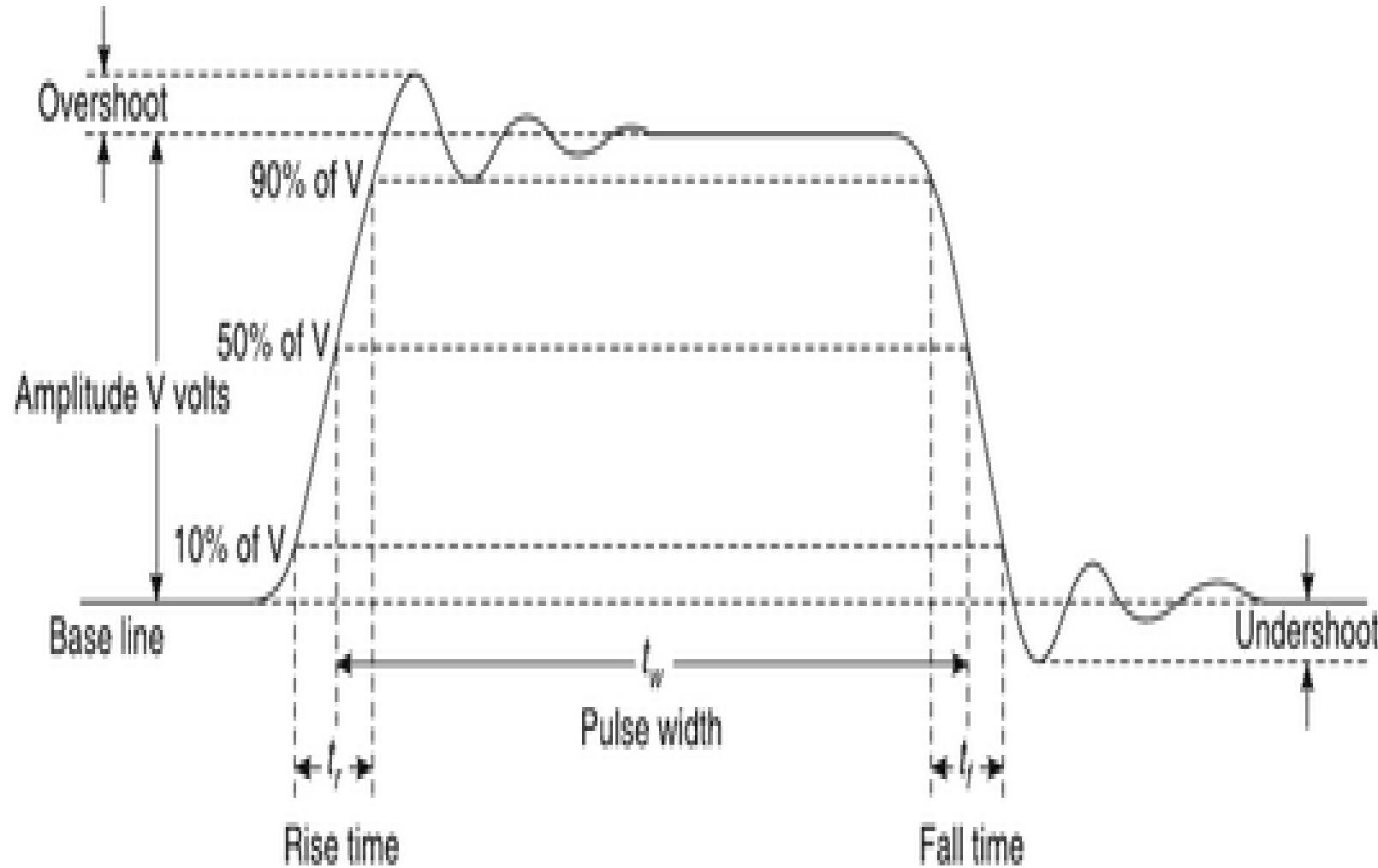
- A pulse may be a positive pulse or a negative pulse.
- A single **positive pulse** is generated when a normally LOW voltage goes to its HIGH level and then returns to its normal LOW level.
- A single **negative pulse** is generated when a normally HIGH voltage goes to its LOW level and then returns to its normal HIGH level.



(a) Positive pulse



(b) Negative pulse



- The **rise time** is defined as the time taken by the pulse to rise from 10% to 90% of the pulse amplitude.
- The **fall time** is defined as the time taken by the pulse to fall from 90% to 10% of the pulse amplitude.
- The duration of the pulse is usually indicated by **pulse width** t_w which is defined as the time between the 50% points on the rising and falling edges.

- In digital systems the waveforms are composed of a series of pulses and can be classified as periodic waveforms and non-periodic waveforms.
- A *periodic waveform* is one which repeats itself at regular intervals of time called the period, T.
- A *non-periodic waveform*, of course, does not repeat itself at regular intervals and may be composed of pulses of different widths and/or differing time intervals between the pulses.

- The reciprocal of the time period is called the *frequency* of the periodic waveform.

$$f = \frac{1}{T}$$

- Duty cycle** of a periodic pulse waveform is the ratio of the ON time (pulse width t_w) to the period of the pulse waveform.

$$\text{duty cycle} = \frac{t_w}{T} \times 100 \%$$

Example: A periodic digital waveform has a pulse width of 25 μs and a period of 150 μs . Determine the frequency and the duty cycle.

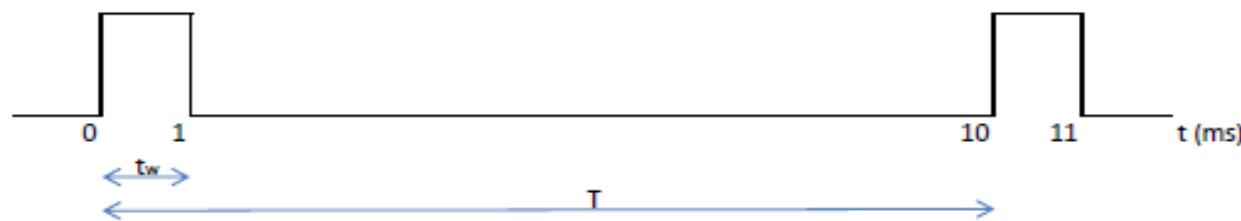
Solution:

Pulse width $t_w = 25 \mu\text{s}$ and period $T = 150 \mu\text{s}$

Frequency $f = \frac{1}{T} = \frac{1}{150 \mu\text{s}} = 6.67 \text{ kHz}$

Duty cycle = $\frac{t_w}{T} \times 100\% = \frac{25 \mu\text{s}}{150 \mu\text{s}} \times 100\% = 16.7\%$

Example: A portion of a periodic digital waveform is shown in Figure. The measurements are in milliseconds. Determine (a) period (b) frequency and (c) duty cycle.



Solution:

$$\text{Period } T = 10 \text{ ms}$$

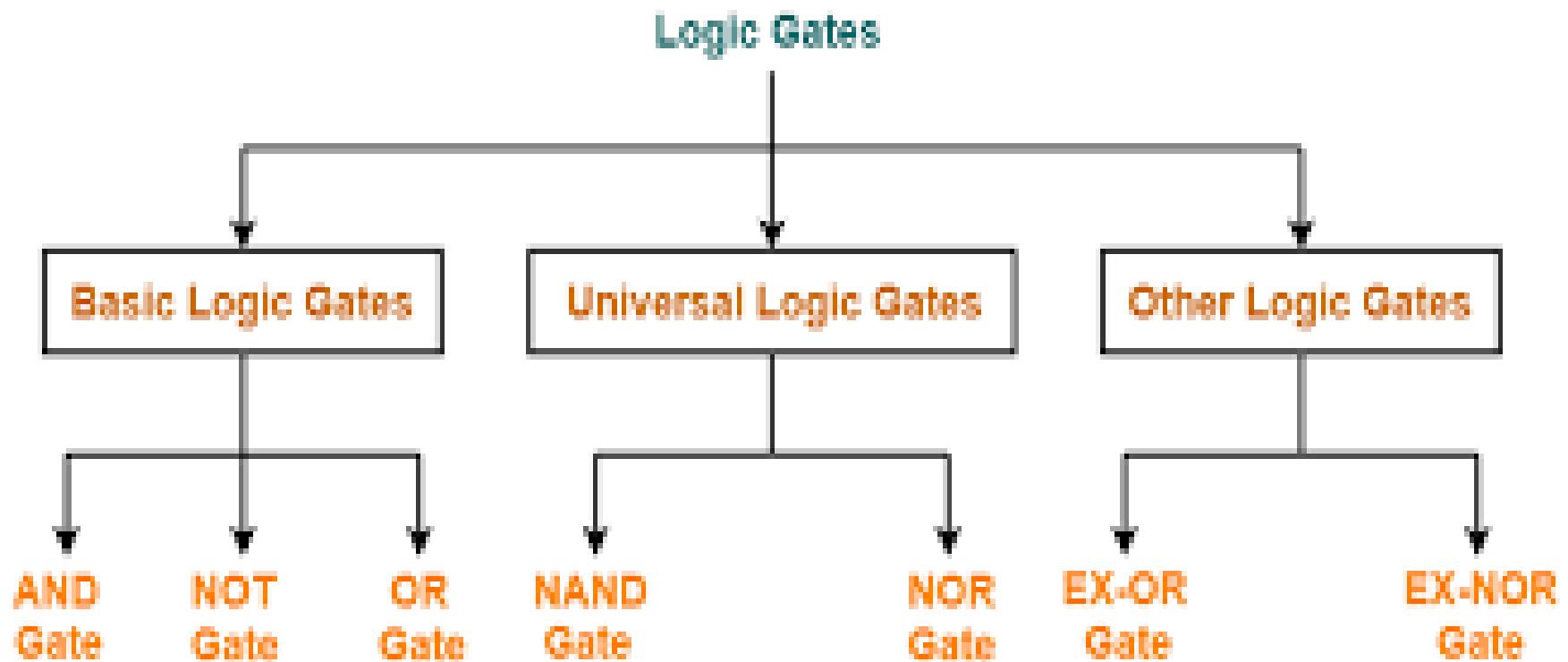
$$\text{Frequency } f = \frac{1}{T} = \frac{1}{10 \text{ ms}} = 100 \text{ Hz}$$

$$\text{Duty cycle} = \frac{t_w}{T} \times 100\% = \frac{1 \text{ ms}}{10 \text{ ms}} \times 100\% = 10\%$$

Logic Gates

- Logic gates are the basic building blocks of any digital system.
- It is an electronic circuit having one or more than one input and only one output.
- The relationship between the input and the output is based on a **certain logic**.
- Based on this, logic gates are named as NOT, AND, OR, NAND, NOR, EX-OR and EX-NOR gate.

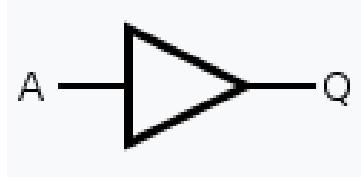
Types of Logic Gates



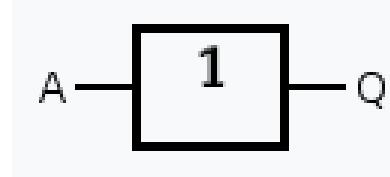
Buffer

- A **buffer**, is a basic logic **gate** that passes its input, unchanged, to its output.
- The main purpose of a **buffer** is to regenerate the input, usually using a strong high and a strong low.
- A **buffer** has one input and one output; its output always equals its input.
- Its symbol is simply a triangle.
- Boolean Expression: $Q = A$

Graphic Symbol



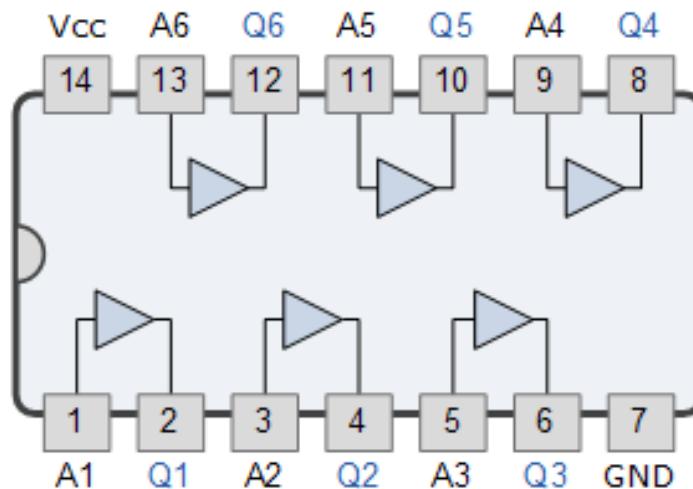
IEEE Symbol



Truth Table

| INPUT | OUTPUT |
|-------|--------|
| A | Q |
| 0 | 0 |
| 1 | 1 |

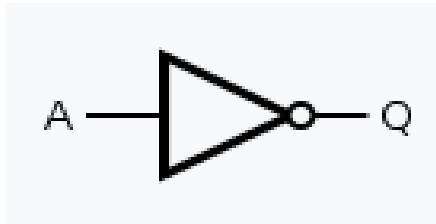
7407 Hex Buffer



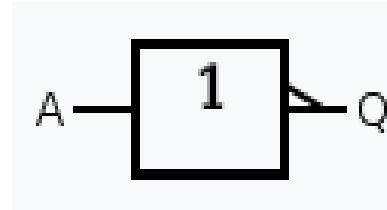
NOT Gate

- The NOT gate is an electronic circuit that produces an inverted version of the input at its output.
- It is also known as an *inverter*.
- If the input variable is A, the inverted output is known as NOT A.
- This is also shown as A' or \overline{A}
- Boolean Expression: $Q = \overline{A}$

Graphic Symbol



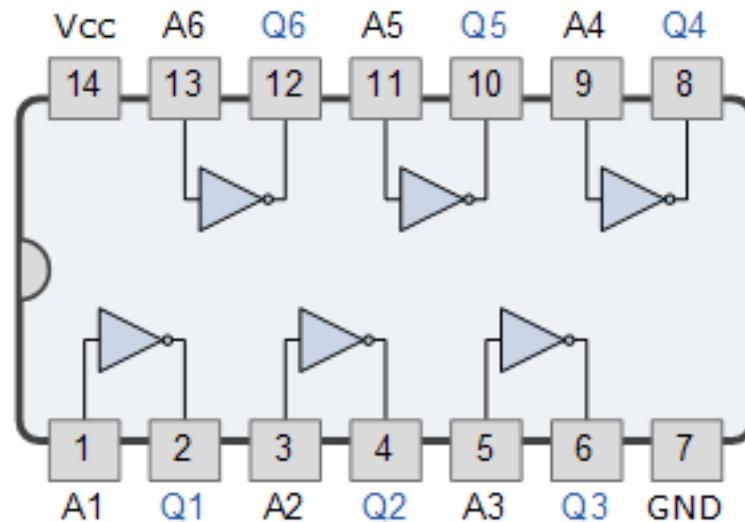
IEEE Symbol



Truth Table

| INPUT | OUTPUT |
|-------|--------|
| A | Q |
| 0 | 1 |
| 1 | 0 |

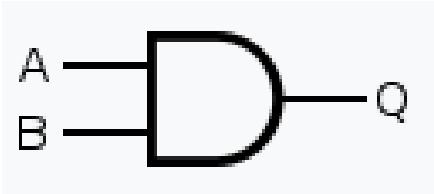
7404 Hex Inverter



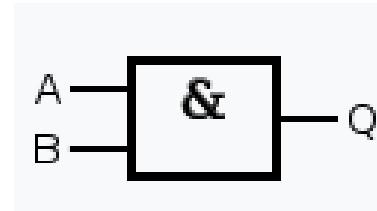
AND Gate

- The AND gate is an electronic circuit that gives a **high** output (1) only if **all** its inputs are high.
- A dot (.) is used to show the AND operation.
- Boolean Expression: $Q = A \cdot B$

Graphic Symbol



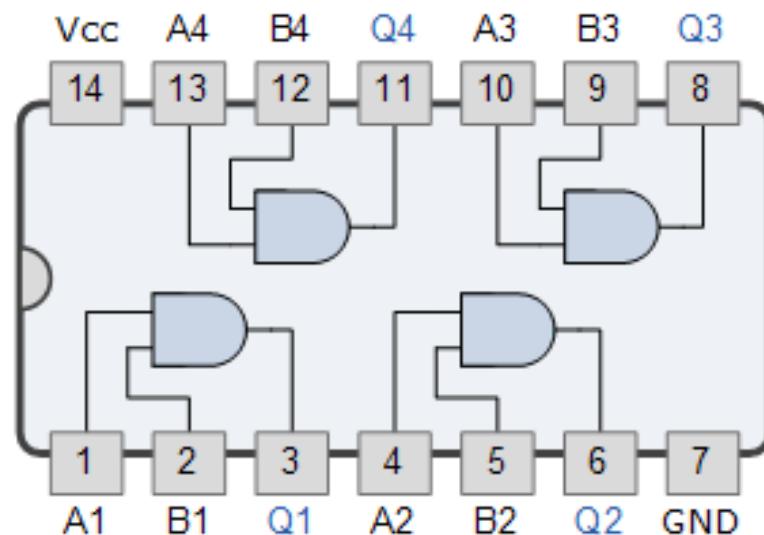
IEEE Symbol



Truth Table

| INPUT | | OUTPUT |
|-------|---|--------|
| A | B | Q |
| 0 | 0 | 0 |
| 0 | 1 | 0 |
| 1 | 0 | 0 |
| 1 | 1 | 1 |

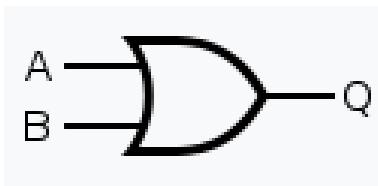
7408 Quad 2-input AND Gate



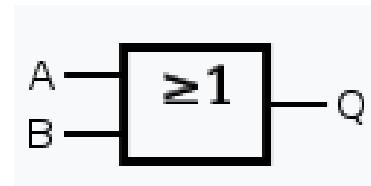
OR Gate

- The OR gate is an electronic circuit that gives a high output (1) if **one or more** of its inputs are high.
- A plus (+) is used to show the OR operation.
- Boolean Expression: $Q = A + B$

Graphic Symbol



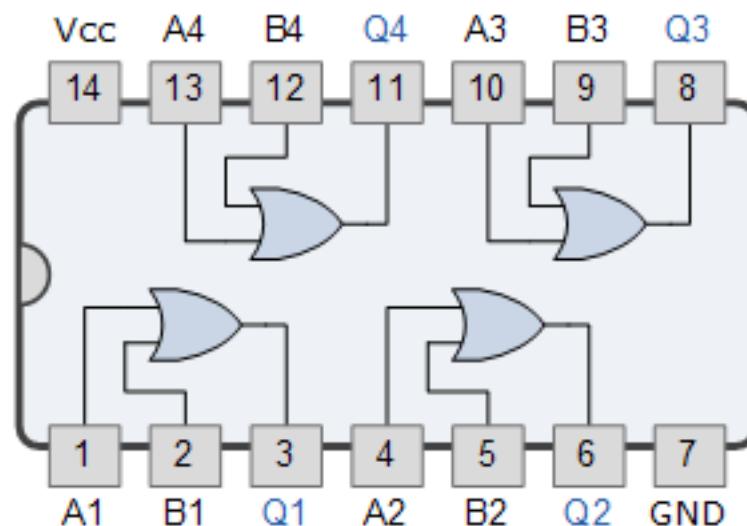
IEEE Symbol



Truth Table

| INPUT | | OUTPUT |
|-------|---|--------|
| A | B | Q |
| 0 | 0 | 0 |
| 0 | 1 | 1 |
| 1 | 0 | 1 |
| 1 | 1 | 1 |

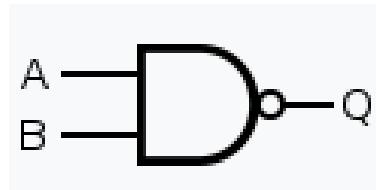
7432 Quad 2-input OR Gate



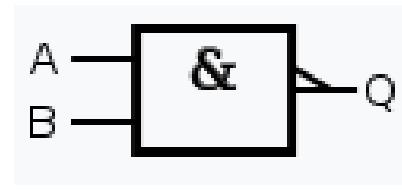
NAND Gate

- This is a NOT-AND gate which is equal to an AND gate followed by a NOT gate.
- The outputs of all NAND gates are high if **any** of the inputs are low.
- The symbol is an AND gate with a small circle on the output. The small circle represents inversion.
- Boolean Expression: $Q = \overline{A \cdot B}$

Graphic Symbol



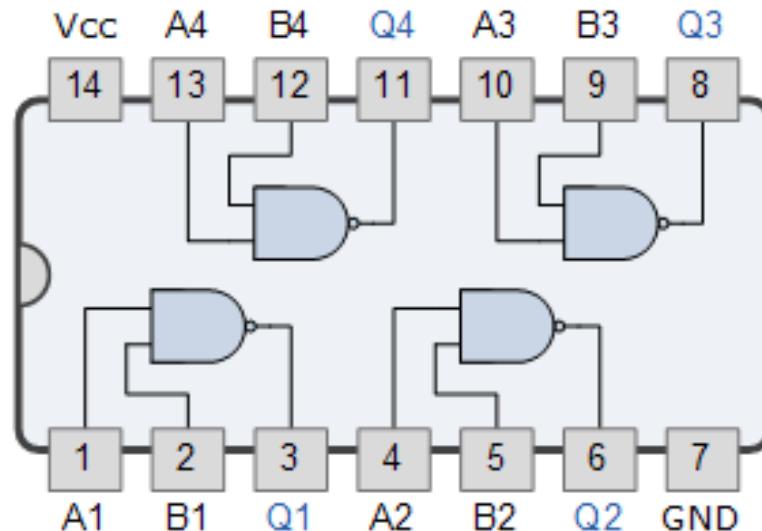
IEEE Symbol



Truth Table

| INPUT | | OUTPUT |
|-------|---|--------|
| A | B | Q |
| 0 | 0 | 1 |
| 0 | 1 | 1 |
| 1 | 0 | 1 |
| 1 | 1 | 0 |

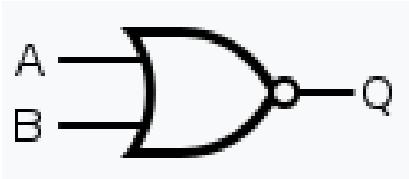
7400 Quad 2-input NAND Gate



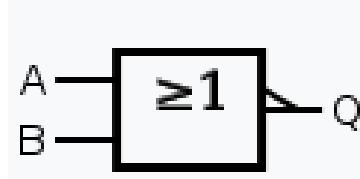
NOR Gate

- This is a NOT-OR gate which is equal to an OR gate followed by a NOT gate.
- The outputs of all NOR gates are low if **any** of the inputs are high.
- The symbol is an OR gate with a small circle on the output. The small circle represents inversion.
- Boolean Expression: $Q = \overline{A + B}$

Graphic Symbol



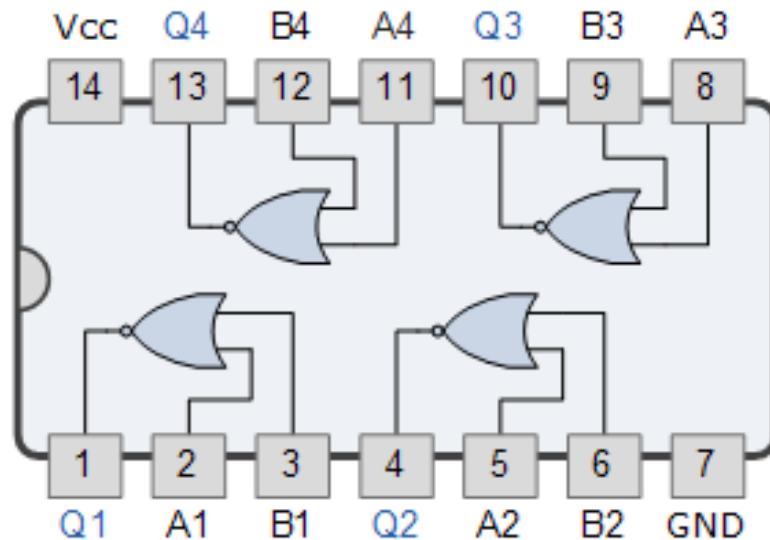
IEEE Symbol



Truth Table

| INPUT | | OUTPUT |
|-------|---|--------|
| A | B | Q |
| 0 | 0 | 1 |
| 0 | 1 | 0 |
| 1 | 0 | 0 |
| 1 | 1 | 0 |

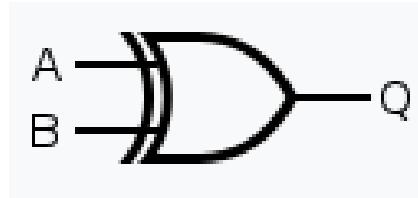
7402 Quad 2-input NOR Gate



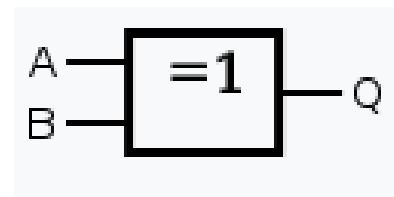
EX-OR Gate

- The '**Exclusive-OR**' gate is a circuit which will give a high output if **either, but not both**, of its two inputs are high.
- An encircled plus sign (\oplus) is used to show the EX-OR operation.
- Boolean Expression: $Q = (A \oplus B) = \overline{A}B + A\overline{B}$

Graphic Symbol



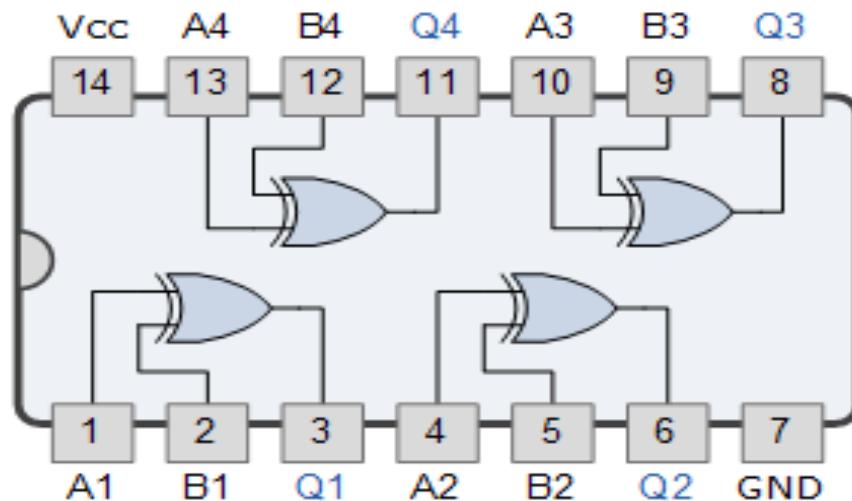
IEEE Symbol



Truth Table

| INPUT | | OUTPUT |
|-------|---|--------|
| A | B | Q |
| 0 | 0 | 0 |
| 0 | 1 | 1 |
| 1 | 0 | 1 |
| 1 | 1 | 0 |

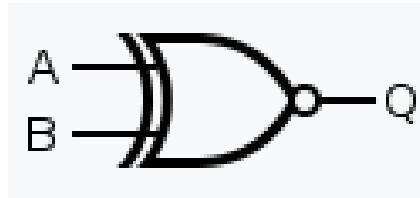
7486 Quad 2-input EX-OR Gate



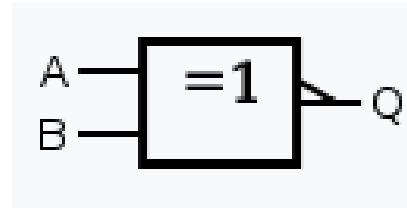
EX-NOR Gate

- The '**Exclusive-NOR**' gate circuit does the opposite to the EX-OR gate.
- It will give a low output if **either, but not both**, of its two inputs are high.
- The symbol is an EX-OR gate with a small circle on the output. The small circle represents inversion.
- Boolean Expression: $Q = (\overline{A} \oplus \overline{B}) = \overline{\overline{A} \cdot \overline{B}} + \overline{A} \cdot \overline{B}$

Graphic Symbol



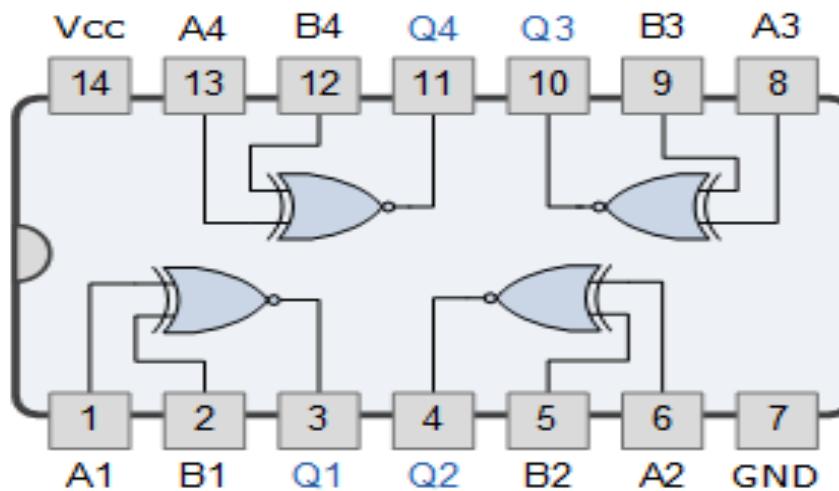
IEEE Symbol



Truth Table

| INPUT | | OUTPUT |
|-------|---|--------|
| A | B | Q |
| 0 | 0 | 1 |
| 0 | 1 | 0 |
| 1 | 0 | 0 |
| 1 | 1 | 1 |

74266 Quad 2-input Ex-NOR Gate



Digital Logic Gate Truth Table Summary

Truth Table Output for Single-input Gates

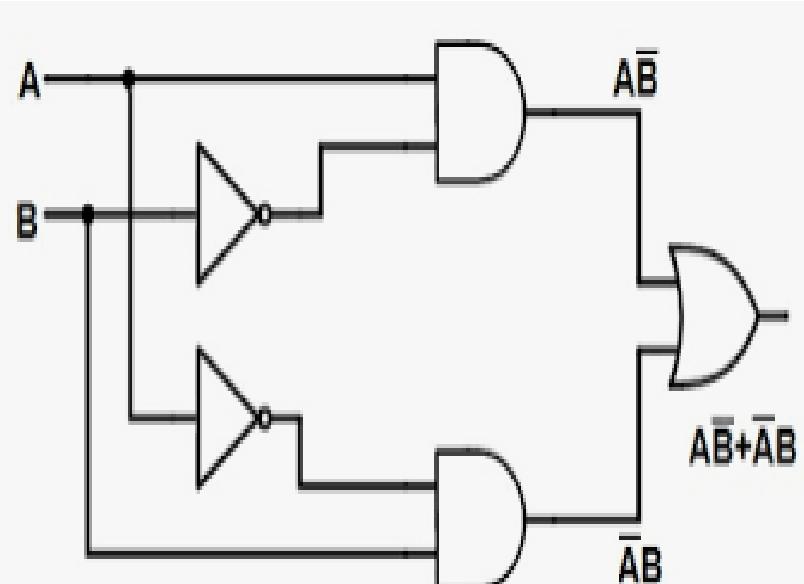
| A | NOT | Buffer |
|---|-----|--------|
| 0 | 1 | 0 |
| 1 | 0 | 1 |

Inputs

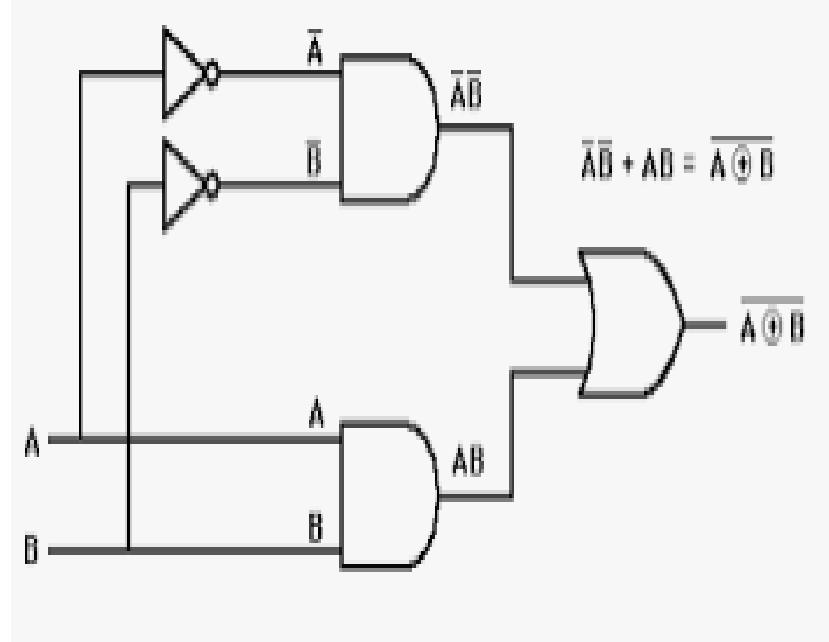
Truth Table Outputs For Each Gate

| B | A | AND | NAND | OR | NOR | EX-OR | EX-NOR |
|---|---|-----|------|----|-----|-------|--------|
| 0 | 0 | 0 | 1 | 0 | 1 | 0 | 1 |
| 0 | 1 | 0 | 1 | 1 | 0 | 1 | 0 |
| 1 | 0 | 0 | 1 | 1 | 0 | 1 | 0 |
| 1 | 1 | 1 | 0 | 1 | 0 | 0 | 1 |

EX-OR Gate



EX-NOR Gate



3-input Ex-OR Gate

The logic function implemented by a 2-input Ex-OR is given as either: “A OR B but NOT both” will give an output at Q. In general, an Ex-OR gate will give an output value of logic “1” ONLY when there are an **ODD** number of 1’s on the inputs to the gate. Then an Ex-OR function with more than two inputs is called an “**odd function**”.

| Truth Table | | | |
|-------------|---|---|---|
| C | B | A | Q |
| 0 | 0 | 0 | 0 |
| 0 | 0 | 1 | 1 |
| 0 | 1 | 0 | 1 |
| 0 | 1 | 1 | 0 |
| 1 | 0 | 0 | 1 |
| 1 | 0 | 1 | 0 |
| 1 | 1 | 0 | 0 |
| 1 | 1 | 1 | 1 |

DIGITAL ELECTRONICS

III Semester CSE & CST



Mr. Srikanta Patnaik Mr. Ajit Kumar Patro Mr. Ami Kumar Parida

DEPARTMENT OF ELECTRONICS AND COMMUNICATION ENGINEERING

SCHOOL OF ENGINEERING & TECHNOLOGY

GIET UNIVERSITY, GUNUPUR, ODISHA

AXIOMS AND LAWS OF BOOLEAN ALGEBRA

Boolean Algebra

- Boolean Algebra is used to analyze and simplify the digital (logic) circuits.
- It uses only the binary numbers i.e. 0 and 1.
- It is also called as **Binary Algebra** or **logical Algebra**.
- Algebra consists of symbolic representation of a statement (generally mathematical statements).
- Similarly, there are expressions, equations and functions in Boolean algebra as well.

Axioms or Postulates of Boolean Algebra

- Axioms or postulates of Boolean algebra are a set of logical expressions that we accept without proof and upon which we can build a set of useful theorems.
- Actually axioms are nothing more than the definitions of three basic logic operations that we have already discussed: AND, OR, INVERT.

OR Operation

$$0 + 0 = 0$$

$$0 + 1 = 1$$

$$1 + 0 = 1$$

$$1 + 1 = 1$$

AND Operation

$$0 \cdot 0 = 0$$

$$0 \cdot 1 = 0$$

$$1 \cdot 0 = 0$$

$$1 \cdot 1 = 1$$

NOT Operation

$$\bar{1} = 0$$

$$\bar{0} = 1$$

1. Postulate 2 : Identity

(a) $A + 0 = A$ (b) $A \cdot 1 = A$

2. Postulate 5 : Complement

(a) $A + \bar{A} = 1$ (b) $A \cdot \bar{A} = 0$

3. Theorem 1 : Idempotent

(a) $A + A = A$ (b) $A \cdot A = A$

4. Theorem 2 : Annulment

(a) $A + 1 = 1$ (b) $A \cdot 0 = 0$

5. Theorem 3 : Involution

$$\bar{\bar{A}} = A$$

- Theorem 1

(a) $A + A = A$

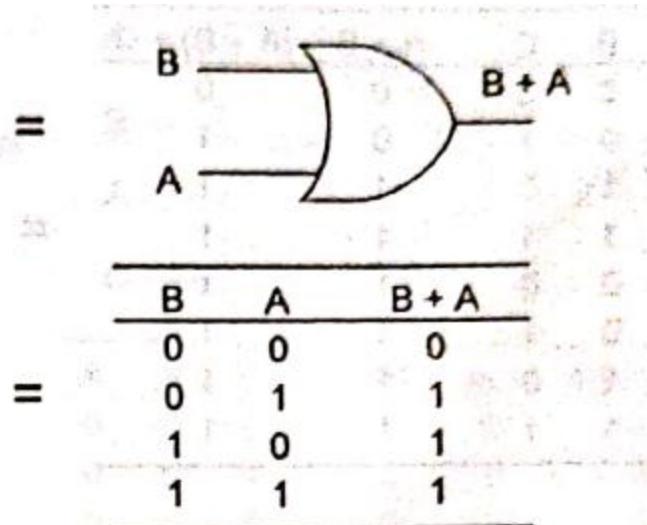
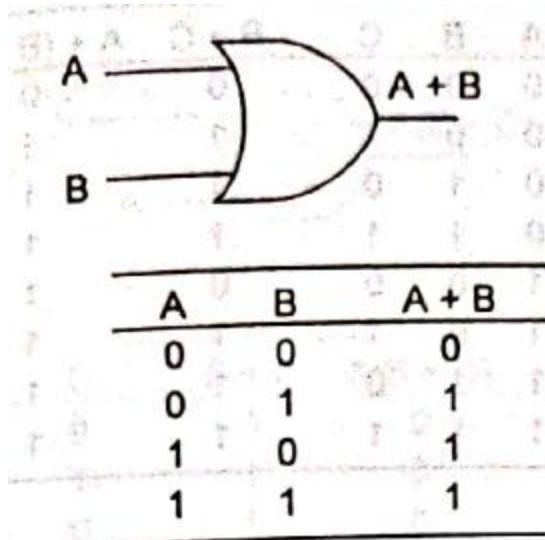
$$\begin{aligned}A + A &= (A + A).1 \\&= (A + A).(A + A') \\&= A.A + A.A' + AA + AA' \\&= A.A + A.A' \\&= A + 0 \\&= A\end{aligned}$$

(b) $A.A = A$

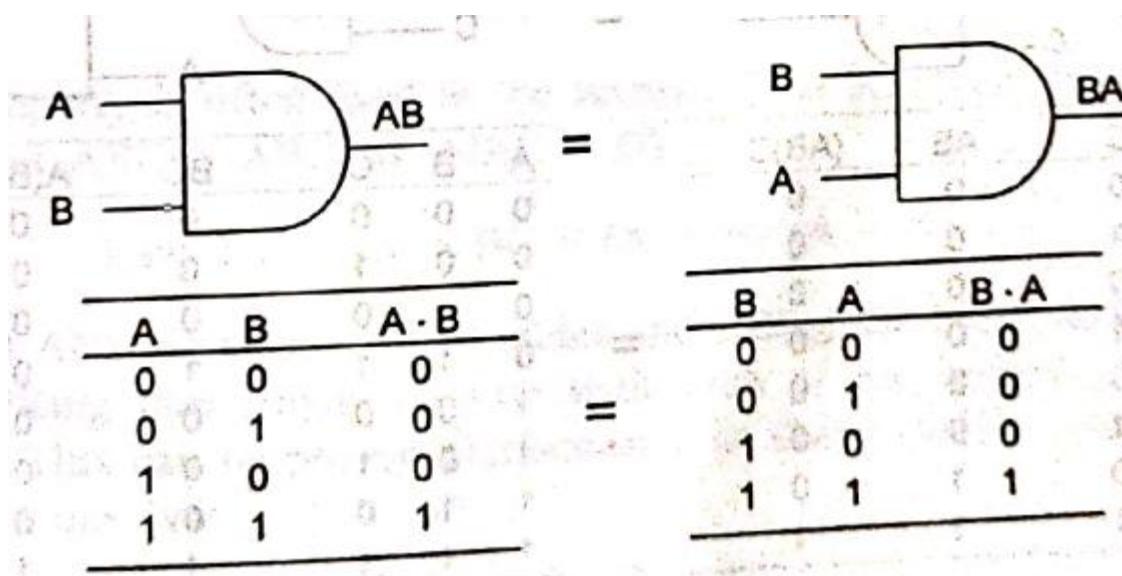
$$\begin{aligned}A.A &= A.A + 0 \\&= A.A + A.A' \\&= A.(A + A') \\&= A.1 \\&= A\end{aligned}$$

6. Postulate 3 : Commutative Law

(a) $A + B = B + A$

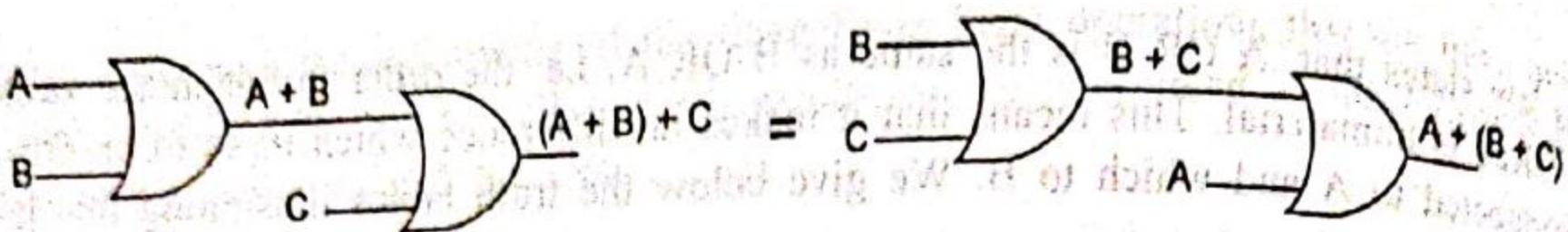


(b) $A \cdot B = B \cdot A$



7. Theorem 4 : Associative Law

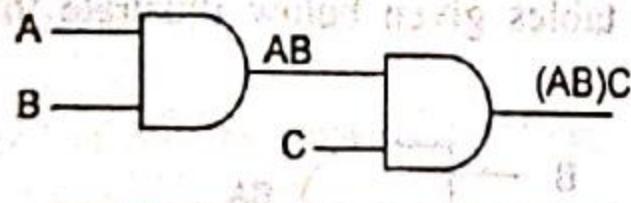
(a) $(A + B) + C = A + (B + C)$



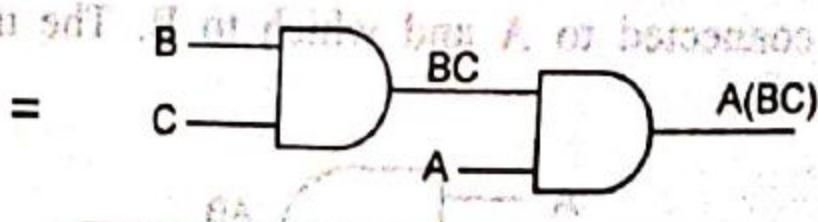
| A | B | C | $A + B$ | $(A + B) + C$ |
|---|---|---|---------|---------------|
| 0 | 0 | 0 | 0 | 0 |
| 0 | 0 | 1 | 0 | 1 |
| 0 | 1 | 0 | 1 | 1 |
| 0 | 1 | 1 | 1 | 1 |
| 1 | 0 | 0 | 1 | 1 |
| 1 | 0 | 1 | 1 | 1 |
| 1 | 1 | 0 | 1 | 1 |
| 1 | 1 | 1 | 1 | 1 |

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

$$(b) (A \cdot B) \cdot C = A \cdot (B \cdot C)$$



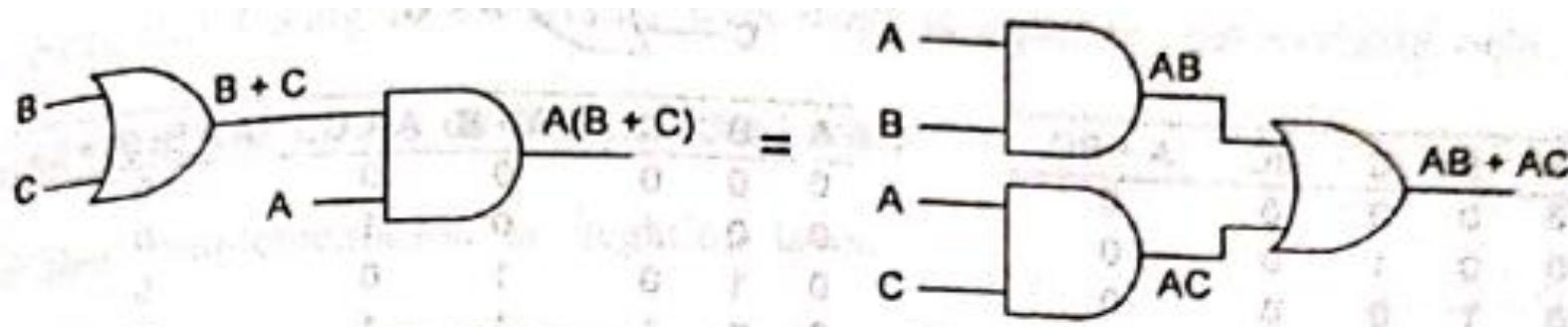
| A | B | C | AB | (AB)C |
|---|---|---|----|-------|
| 0 | 0 | 0 | 0 | 0 |
| 0 | 0 | 1 | 0 | 0 |
| 0 | 1 | 0 | 0 | 0 |
| 0 | 1 | 1 | 0 | 0 |
| 1 | 0 | 0 | 0 | 0 |
| 1 | 0 | 1 | 0 | 0 |
| 1 | 1 | 0 | 1 | 0 |
| 1 | 1 | 1 | 1 | 1 |



| A | B | C | BC | A(BC) |
|---|---|---|----|-------|
| 0 | 0 | 0 | 0 | 0 |
| 0 | 0 | 1 | 0 | 0 |
| 0 | 1 | 0 | 0 | 0 |
| 0 | 1 | 1 | 1 | 0 |
| 1 | 0 | 0 | 0 | 0 |
| 1 | 0 | 1 | 0 | 0 |
| 1 | 1 | 0 | 0 | 0 |
| 1 | 1 | 1 | 1 | 1 |

8. Postulate 4 : Distributive Law

(a) $A.(B + C) = A.B + A.C$

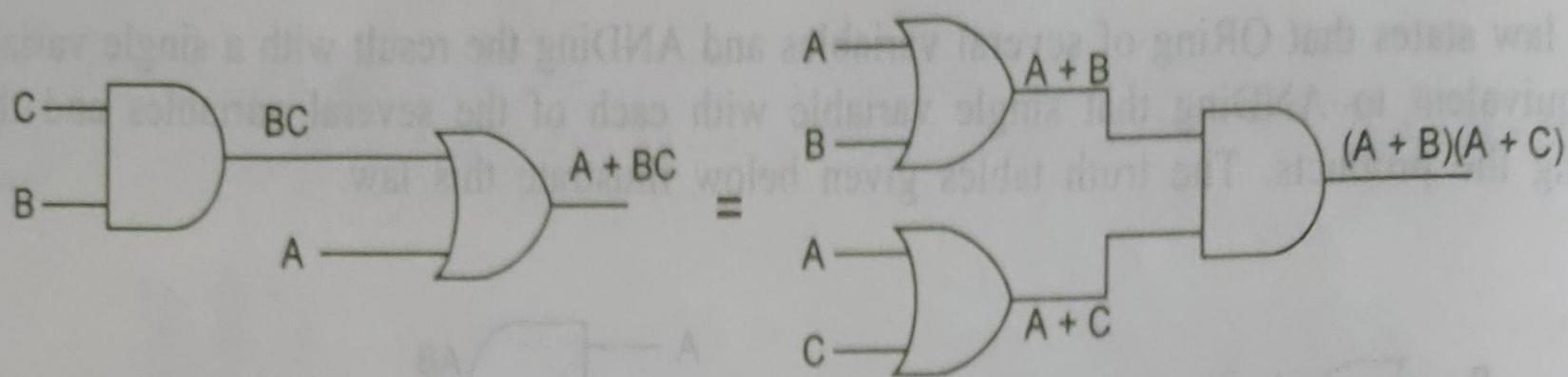


| A | B | C | $B + C$ | $A(B + C)$ |
|---|---|---|---------|------------|
| 0 | 0 | 0 | 0 | 0 |
| 0 | 0 | 1 | 1 | 0 |
| 0 | 1 | 0 | 1 | 0 |
| 0 | 1 | 1 | 1 | 0 |
| 1 | 0 | 0 | 0 | 0 |
| 1 | 0 | 1 | 1 | 1 |
| 1 | 1 | 0 | 1 | 1 |
| 1 | 1 | 1 | 1 | 1 |

| A | B | C | AB | AC | $AB + AC$ |
|---|---|---|----|----|-----------|
| 0 | 0 | 0 | 0 | 0 | 0 |
| 0 | 0 | 1 | 0 | 0 | 0 |
| 0 | 1 | 0 | 0 | 0 | 0 |
| 0 | 1 | 1 | 0 | 0 | 0 |
| 1 | 0 | 0 | 0 | 0 | 0 |
| 1 | 0 | 1 | 0 | 0 | 0 |
| 1 | 1 | 0 | 1 | 0 | 1 |
| 1 | 1 | 1 | 1 | 1 | 1 |

$$(b) A + B.C = (A + B).(A + C)$$

$$\begin{aligned} \text{RHS} &= (A + B).(A + C) \\ &= A.A + A.C + B.A + B.C \\ &= A + A.C + A.B + B.C \\ &= A.(1 + C + B) + B.C \\ &= A.1 + B.C \\ &= A + B.C = \text{LHS} \end{aligned}$$



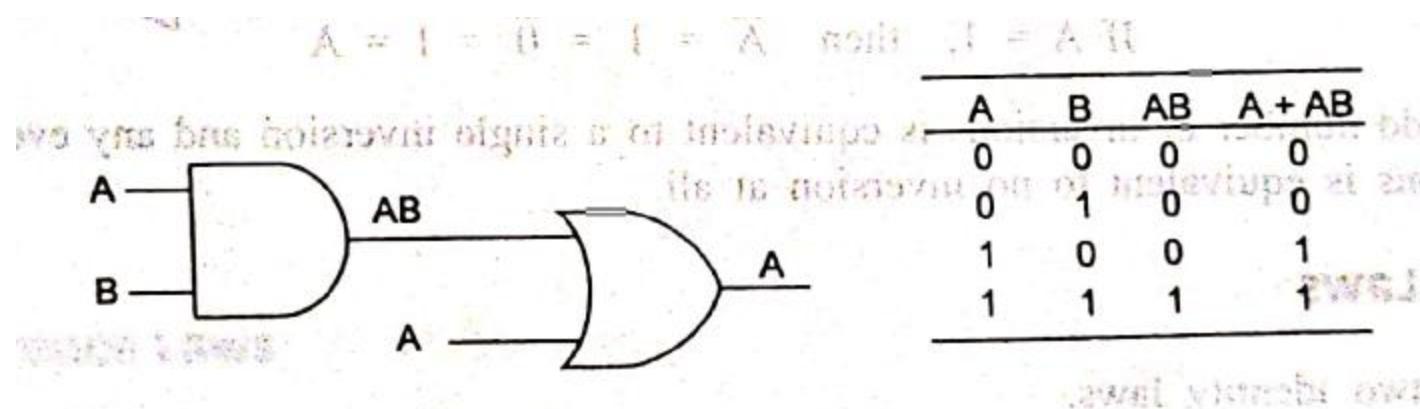
| A | B | C | BC | $A + BC$ |
|---|---|---|----|----------|
| 0 | 0 | 0 | 0 | 0 |
| 0 | 0 | 1 | 0 | 0 |
| 0 | 1 | 0 | 0 | 0 |
| 0 | 1 | 1 | 1 | 1 |
| 1 | 0 | 0 | 0 | 1 |
| 1 | 0 | 1 | 0 | 1 |
| 1 | 1 | 0 | 0 | 1 |
| 1 | 1 | 1 | 1 | 0 |

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

9. Theorem 6 : Absorption

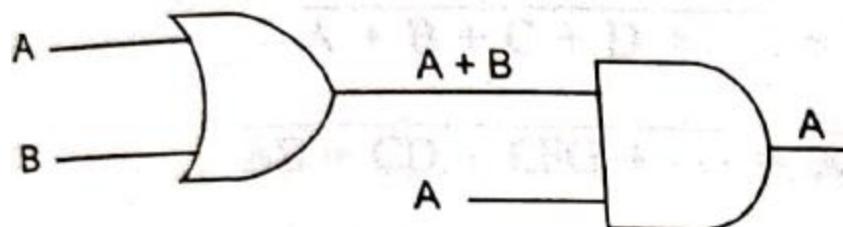
(a) $A + A \cdot B = A$

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



$$(b) A.(A + B) = A$$

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



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

10. Consensus Theorem

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

Proof:

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

= RHS

11. Transposition Theorem

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

Proof:

$$\begin{aligned}\text{RHS} &= (A + C)(\bar{A} + B) \\&= A\bar{A} + C\bar{A} + AB + CB \\&= 0 + \bar{A}C + AB + BC \\&= \bar{A}C + AB + BC(A + \bar{A}) \\&= AB + ABC + \bar{A}C + \bar{A}BC \\&= AB + \bar{A}C \\&= \text{LHS}\end{aligned}$$

12. De Morgan's Theorem

- De Morgan's Theorem states that
 - i. The complement of a sum of variables is equal to the product of their individual complements.

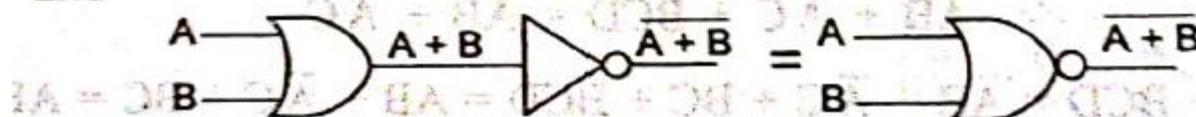
$$\overline{A + B} = \bar{A} \cdot \bar{B}$$

- ii. The complement of the product of variables is equal to the sum of their individual complements.

$$\overline{A \cdot B} = \bar{A} + \bar{B}$$

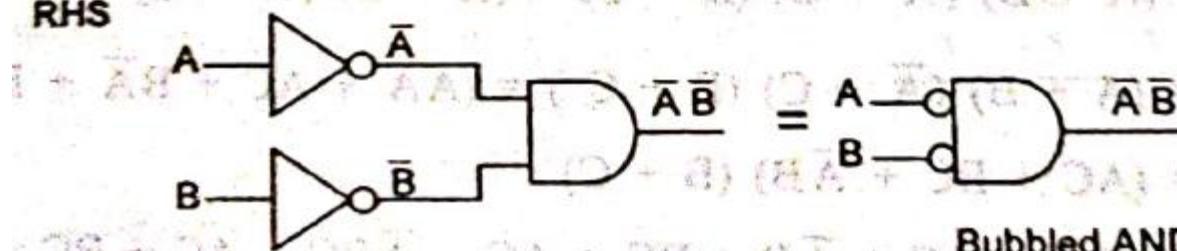
$$\overline{A + B} = \bar{A} \cdot \bar{B}$$

LHS



NOR gate

RHS



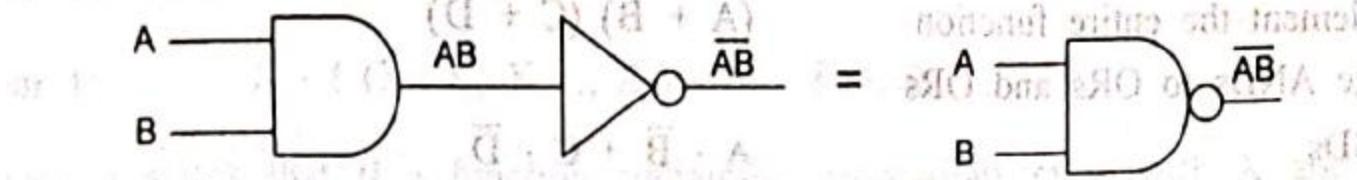
Bubbled AND gate

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

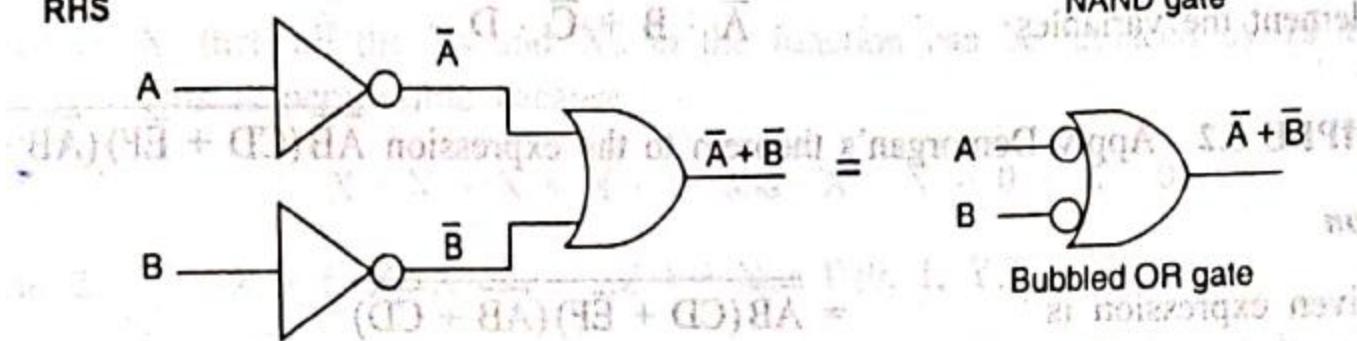
| A | B | \bar{A} | \bar{B} | $\bar{A} \cdot \bar{B}$ |
|---|---|-----------|-----------|-------------------------|
| 0 | 0 | 1 | 1 | 1 |
| 0 | 1 | 1 | 0 | 0 |
| 1 | 0 | 0 | 1 | 0 |
| 1 | 1 | 0 | 0 | 0 |

$$\overline{A \cdot B} = \bar{A} + \bar{B}$$

LHS



RHS



| | A | B | $\bar{A} \cdot \bar{B}$ | $\bar{A} + \bar{B}$ | A | B | \bar{A} | \bar{B} | $\bar{A} + \bar{B}$ |
|--|---|---|-------------------------|---------------------|---|---|-----------|-----------|---------------------|
| | 0 | 0 | 1 | 0 | 0 | 0 | 1 | 1 | 1 |
| | 0 | 1 | 0 | 1 | 0 | 1 | 1 | 0 | 1 |
| | 1 | 0 | 0 | 1 | 1 | 0 | 0 | 1 | 1 |
| | 1 | 1 | 0 | 1 | 1 | 1 | 0 | 0 | 0 |

Duality

- It states that every algebraic expression deducible from the postulates of Boolean algebra remains valid if the operators and identity elements are interchanged.
- If the *dual* of a algebraic expression is desired, we simply interchanged OR and AND operators and replace 1's by 0's and 0's by 1's.

Prove that the dual of EX-OR gate is equal to its complement.

- EX-OR gate : $A \cdot B' + A' \cdot B$
- Dual of EX-OR gate :
$$\begin{aligned} & (A + B') \cdot (A' + B) \\ &= A \cdot A' + AB + B' \cdot A' + B' \cdot B \\ &= 0 + AB + B' \cdot A' + 0 \\ &= AB + A' \cdot B' \end{aligned}$$
- Complement of EX-OR gate
$$\begin{aligned} (AB' + A'B)' &= (AB')' \cdot (A'B)' = (A' + B) \cdot (A + B') \\ &= A'A + A'B' + BA + BB' \\ &= 0 + A'B' + BA + 0 = AB + A'B' \end{aligned}$$

DIGITAL ELECTRONICS

III Semester CSE & CST



Mr. Srikanta Patnaik

Mr. Ajit Kumar Patro

Mr. Ami Kumar Parida

DEPARTMENT OF ELECTRONICS AND COMMUNICATION ENGINEERING

SCHOOL OF ENGINEERING & TECHNOLOGY

GIET UNIVERSITY, GUNUPUR, ODISHA

Reducing Boolean Expressions

Boolean Expressions
and Logic Diagrams

Example: Simplify the following Boolean functions to a minimum number of literals.

1. $x(x' + y)$
2. $x + x'y$
3. $(x + y)(x + y')$
4. $xy + x'z + yz$

Solution

$$\begin{aligned}1. \quad & x(x' + y) \\&= xx' + xy \\&= 0 + xy \\&= xy\end{aligned}$$

$$\begin{aligned}2. \quad & x + x'y \\&= (x + x')(x + y) \\&= 1(x + y) \\&= x + y\end{aligned}$$

Solution

$$3. \quad (x + y)(x + y')$$

$$= xx + xy' + xy + yy'$$

$$= x + xy + xy' + 0$$

$$= x(1 + y + y')$$

$$= x$$

$$4. \quad xy + x'z + yz$$

$$= xy + x'z + yz(x + x')$$

$$= xy + x'z + xyz + x'yz$$

$$= xy(1 + z) + x'z(1 + y)$$

$$= xy + x'z$$

Example: Prove that

$$(x + y)(x' + z)(y + z) = (x + y)(x' + z)$$

Solution

- $(x + y)(x' + z)(y + z) = (x + y)(x' + z)$

$$\begin{aligned} L.H.S. &= (x + y)(x' + z)(y + z) \\ &= (xx' + xz + x'y + yz)(y + z) \\ &= (xz + x'y + yz)(y + z) \\ &= xyz + xzz + x'yy + x'yz + yyz + yzz \\ &= xyz + xz + x'y + x'yz + yz + yz \\ &= (x + x')yz + xz + x'y + yz \\ &= yz + xz + x'y + yz = xz + x'y + yz \end{aligned}$$

$$\begin{aligned} R.H.S. &= (x + y)(x' + z) \\ &= xx' + xz + x'y + yz = xz + x'y + yz \end{aligned}$$

L.H.S. = R.H.S. (Proved)

Example: Reduce the expression

$$\overline{(a + \bar{b} + c) + (b + \bar{c})}$$

- *Solution*

$$\begin{aligned}& \overline{(a + \bar{b} + c) + (b + \bar{c})} \\&= \overline{(a + \bar{b} + c)} \cdot \overline{(b + \bar{c})} \\&= (\bar{a}b\bar{c}) \cdot (\bar{b}c) = 0\end{aligned}$$

Example: Reduce the expression

$$(B + BC)(B + \bar{B}C)(B + D)$$

- *Solution*

$$\begin{aligned}& (B + BC)(B + \bar{B}C)(B + D) \\&= (BB + B\bar{B}C + BCB + BCB\bar{B}C)(B + D) \\&= (B + 0 + BC + 0)(B + D) \\&= (B + BC)(B + D) \\&= B(1 + C)(B + D) \\&= B(B + D) \\&= BB + BD \\&= B + BD \\&= B(1 + D) \\&= B\end{aligned}$$

Example: Reduce the expression

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

- *Solution*

$$\begin{aligned}& A [B + \overline{C} (\overline{AB} + A\overline{C})] \\&= A [B + \overline{C} (\overline{AB} \overline{AC})] \\&= A [B + \overline{C} (\overline{A} + \overline{B})(\overline{A} + C)] \\&= A [B + \overline{C} (\overline{A}\overline{A} + \overline{A}C + \overline{A}\overline{B} + \overline{B}C)] \\&= A [B + \overline{A}\overline{C} + \overline{A}C\overline{C} + \overline{A}\overline{B}\overline{C} + \overline{B}C\overline{C}] \\&= A [B + \overline{A}\overline{C} + 0 + \overline{A}\overline{B}\overline{C} + 0] \\&= AB + A\overline{A}\overline{C} + A\overline{A}\overline{B}\overline{C} \\&= AB + 0 + 0 \\&= AB\end{aligned}$$

Example: Reduce the expression

$$A + B[AC + (B + \bar{C})D]$$

- *Solution*

$$\begin{aligned} A + B[AC + (B + \bar{C})D] \\ &= A + B(AC + BD + \bar{C}D) \\ &= A + BAC + BBD + B\bar{C}D \\ &= A + ABC + BD + BDC \\ &= A(1 + BC) + BD(1 + \bar{C}) \\ &= A \cdot 1 + BD \cdot 1 \\ &= A + BD \end{aligned}$$

Example: Reduce the expression

$$\left(\overline{A + \overline{BC}} \right) (A\overline{B} + ABC)$$

- *Solution*

$$\begin{aligned}& \left(\overline{A + \overline{BC}} \right) (A\overline{B} + ABC) \\&= (\overline{\overline{A + \overline{BC}}}) (A\overline{B} + ABC) \\&= (\overline{A}\overline{\overline{BC}}) (A\overline{B} + ABC) \\&= (\overline{A}BC) (A\overline{B} + ABC) \\&= \overline{A}BC A\overline{B} + \overline{A}BC ABC \\&= A\overline{A}B\overline{B}C + A\overline{A}BBCC \\&= 0 + 0 = 0\end{aligned}$$

Example: Show that

$$AB + A\bar{B}C + B\bar{C} = AC + B\bar{C}$$

- *Solution*

$$\begin{aligned} AB + A\bar{B}C + B\bar{C} &= A(B + \bar{B}C) + B\bar{C} \\ &= A(B + \bar{B})(B + C) + B\bar{C} \\ &= AB + AC + B\bar{C} \\ &= AB(C + \bar{C}) + AC + B\bar{C} \\ &= ABC + AB\bar{C} + AC + B\bar{C} \\ &= AC(1 + B) + B\bar{C}(1 + A) \\ &= AC + B\bar{C} \end{aligned}$$

Example: Show that

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

- *Solution*

$$\begin{aligned} & A\bar{B}C + B + B\bar{D} + ABD + \bar{A}C \\ &= A\bar{B}C + \bar{A}C + B(1 + \bar{D} + AD) \\ &= C(\bar{A} + A\bar{B}) + B \\ &= C(\bar{A} + A)(\bar{A} + \bar{B}) + B \\ &= C\bar{A} + C\bar{B} + B \\ &= (B + C)(B + \bar{B}) + C\bar{A} \\ &= B + C + C\bar{A} \\ &= B + C(1 + \bar{A}) \\ &= B + C \end{aligned}$$

Example: Find the complement of the function $F_1 = x'yz' + x'y'z$.

Solution

- $F_1 = x'yz' + x'y'z$

$$F_1' = (x'yz' + x'y'z)'$$

$$= (x'yz')' (x'y'z)'$$

$$= (x + y' + z)(x + y + z')$$

Example: Find the complement of the function $F_2 = x(y'z' + yz)$.

Solution

- $F_2 = x(y'z' + yz)$

$$F_2' = [x(y'z' + yz)]'$$

$$= x' + (y'z' + yz)'$$

$$= x' + (y'z')'(yz)'$$

$$= x' + (y + z)(y' + z')$$

Example: Find the complement of the function $F_1 = x'yz' + x'y'z$ by taking their duals and complementing each literal.

Solution

- $F_1 = x'yz' + x'y'z$
- The dual of F_1 is $(x' + y + z')(x' + y' + z)$
- Complement each literal:

$$(x + y' + z)(x + y + z') = F_1'$$

Example: Find the complement of the function $F_2 = x(y'z' + yz)$ by taking their duals and complementing each literal.

Solution

- $F_2 = x(y'z' + yz)$
- The dual of F_2 is $x + (y' + z')(y + z)$
- Complement each literal:

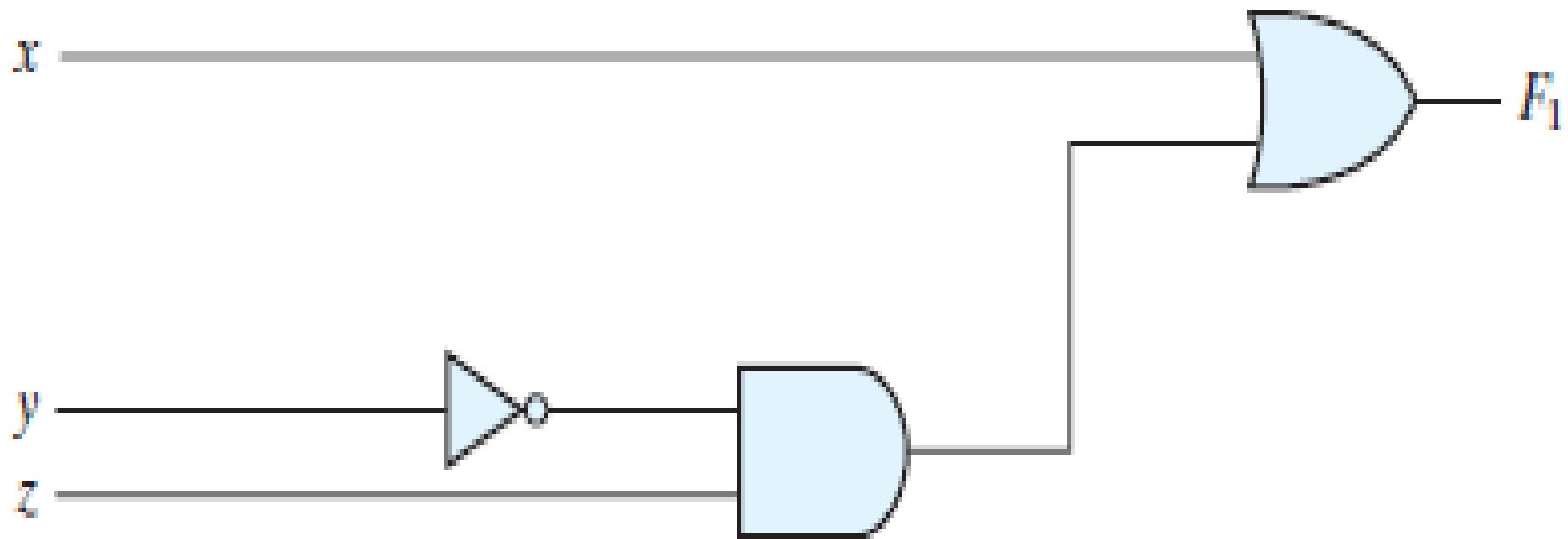
$$x' + (y + z)(y' + z') = F_2'$$

Example: Find the complement of $F = x + yz$; then show that $FF' = 0$ and $F + F' = 1$.

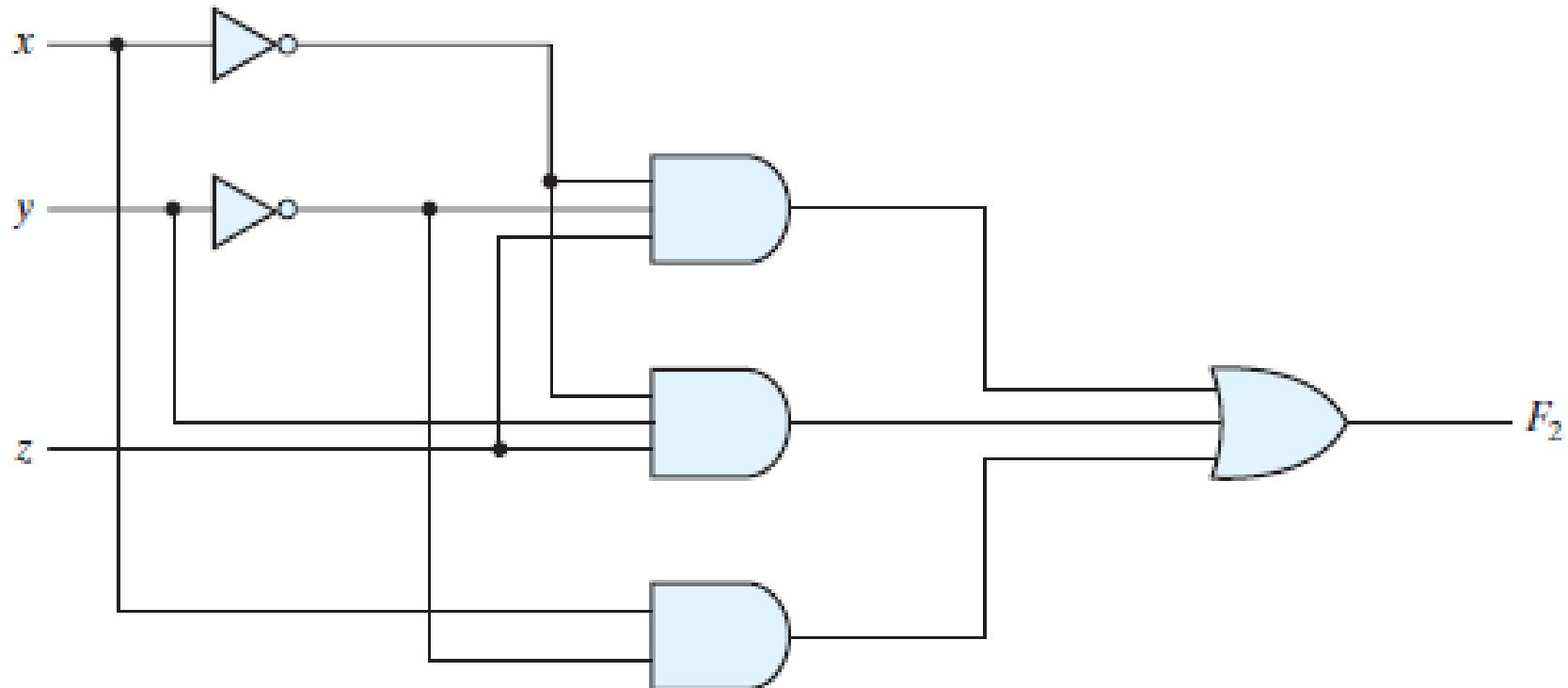
Solution

- $F = x + yz$
- $F' = (x + yz)' = x' (yz)' = x'(y' + z') = x'y' + x'z'$
- $FF' = (x + yz) (x'y' + x'z')$
 $= xx'y' + xx'z' + yzx'y' + yzx'z' = 0$
- $F + F' = (x + yz) + (x + yz)' = 1$

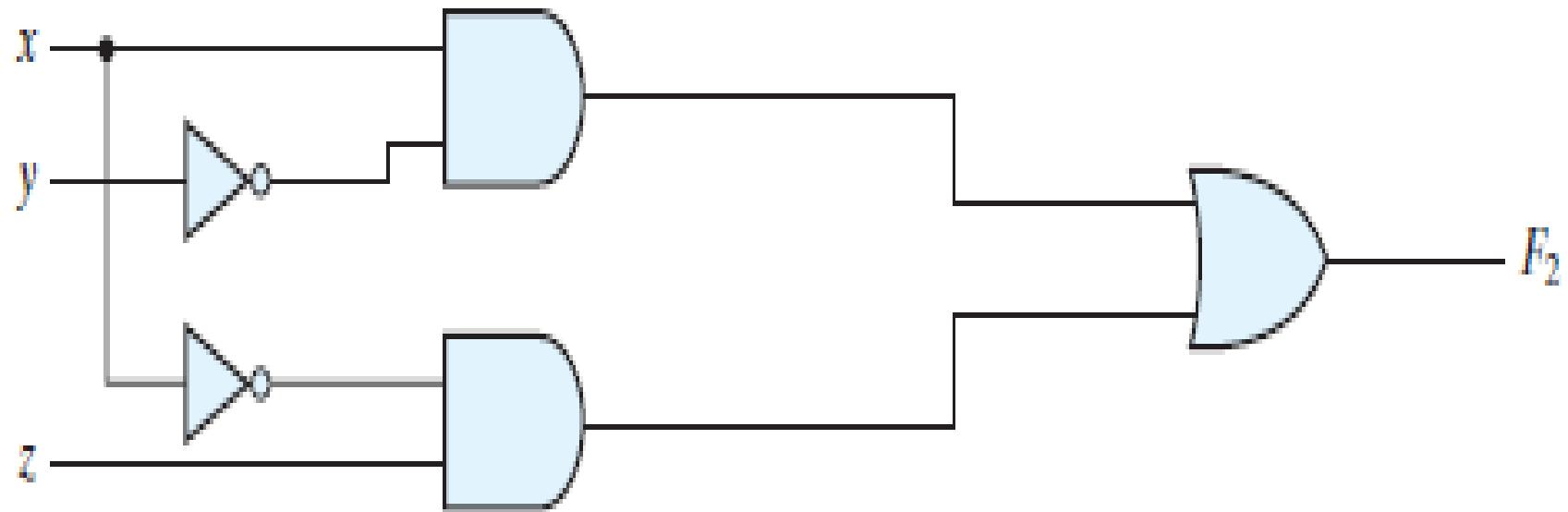
Example: Draw logic diagrams to implement the following Boolean expression: $F_1 = x + y'z$



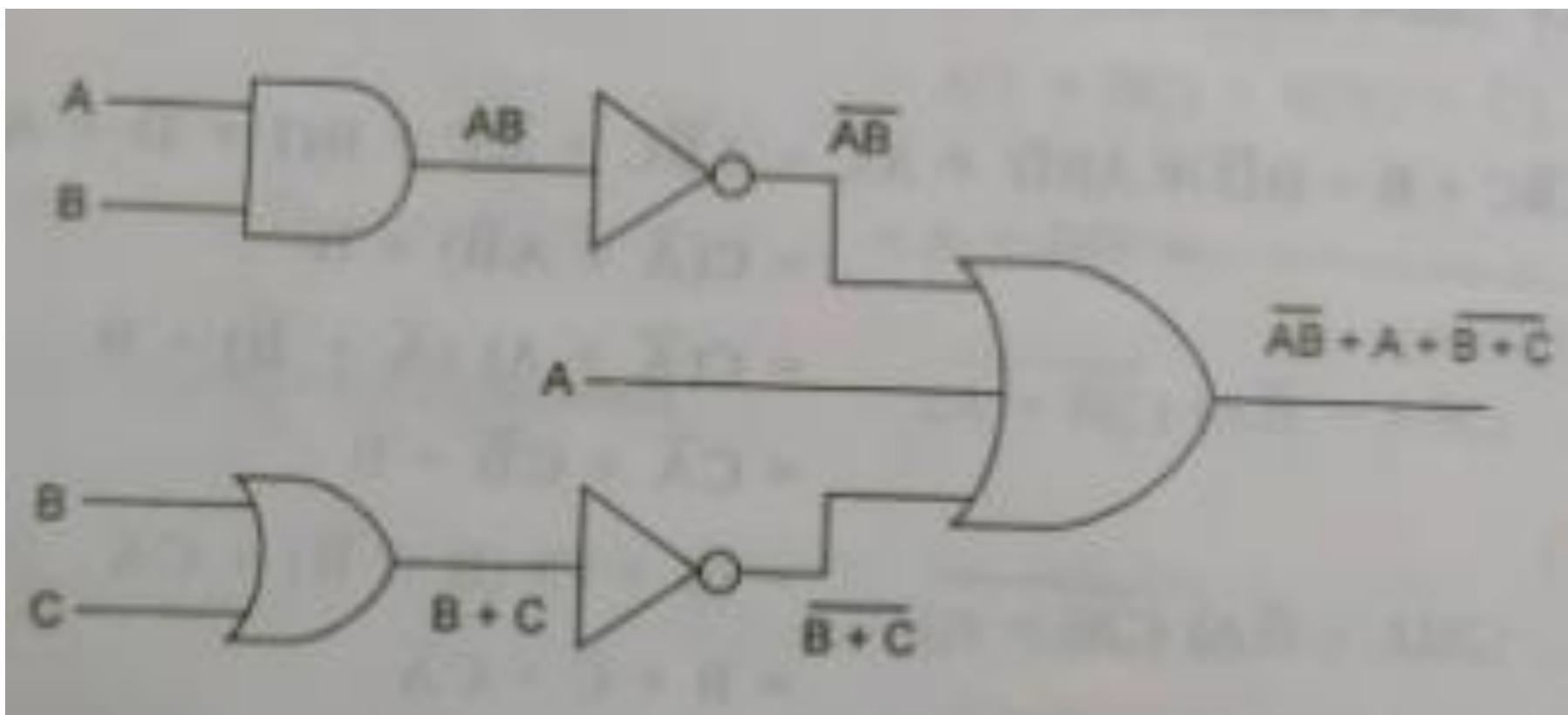
Example: Draw logic diagrams to implement the following Boolean expression: $F_2 = x'y'z + x'yz + xy'$



- $$\begin{aligned}
 F_2 &= x'y'z + x'yz + xy' \\
 &= x'z(y' + y) + xy' = x'z + xy'
 \end{aligned}$$



Example: Draw logic diagrams to implement the following Boolean expression: $\overline{AB} + A + \overline{B + C}$



DIGITAL ELECTRONICS

III Semester CSE & CST



Mr. Srikanta Patnaik

Mr. Ajit Kumar Patro

Mr. Ami Kumar Parida

DEPARTMENT OF ELECTRONICS AND COMMUNICATION ENGINEERING

SCHOOL OF ENGINEERING & TECHNOLOGY

GIET UNIVERSITY, GUNUPUR, ODISHA

GATE-LEVEL MINIMIZATION

CANONICAL AND STANDARD FORMS

Boolean Functions

- Boolean algebra is an algebra that deals with binary variables and logic operations.
- A Boolean function described by an algebraic expression consists of binary variables, the constants 0 and 1, and the logic operation symbols.
- For a given value of the binary variables, the function can be equal to either 1 or 0.

- As an example, consider the Boolean function

$$F_1 = x + y'z$$

- The function F_1 is equal to 1 if x is equal to 1 or if both y' and z are equal to 1. F_1 is equal to 0 otherwise. Therefore, $F_1 = 1$ if $x = 1$ or if $y = 0$ and $z = 1$.

- A Boolean function can be represented in a **truth table**.
- The number of rows in the truth table is 2^n , where n is the number of variables in the function.
- The binary combinations for the truth table are obtained from the binary numbers by counting from 0 through $2^n - 1$.

Table shows the truth tables for the function $F_1 = x + y'z$ and $F_2 = x'y'z + x'yz + xy'$

| x | y | z | F_1 | F_2 |
|-----|-----|-----|-------|-------|
| 0 | 0 | 0 | 0 | 0 |
| 0 | 0 | 1 | 1 | 1 |
| 0 | 1 | 0 | 0 | 0 |
| 0 | 1 | 1 | 0 | 1 |
| 1 | 0 | 0 | 1 | 1 |
| 1 | 0 | 1 | 1 | 1 |
| 1 | 1 | 0 | 1 | 0 |
| 1 | 1 | 1 | 1 | 0 |

Example: Draw the truth table for the Boolean function: $F = A + BC$.

| A | B | C | F |
|-----|-----|-----|-----|
| 0 | 0 | 0 | 0 |
| 0 | 0 | 1 | 0 |
| 0 | 1 | 0 | 0 |
| 0 | 1 | 1 | 1 |
| 1 | 0 | 0 | 1 |
| 1 | 0 | 1 | 1 |
| 1 | 1 | 0 | 1 |
| 1 | 1 | 1 | 1 |

Example: Draw the truth tables for 3-input Universal Gates.

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

Canonical and Standard Forms

Minterms and Maxterms

- A binary variable may appear either in its normal form (x) or in its complement form (x').
- Now consider two binary variables x and y combined with an AND operation.
- Since each variable may appear in either form, there are four possible combinations: $x'y'$, $x'y$, xy' , and xy .
- Each of these four AND terms is called a *minterm*, or a *standard product*.

- In a similar manner, n variables can be combined to form 2^n minterms.
- The binary numbers from 0 to $2^n - 1$ are listed under the n variables.
- Each minterm is obtained from an AND term of the n variables, with each variable being primed if the corresponding bit of the binary number is a 0 and unprimed if a 1.

- A symbol for each minterm is of the form m_j , where the subscript j denotes the decimal equivalent of the binary number of the minterm designated.
- In a similar fashion, n variables forming an OR term, with each variable being primed or unprimed, provide 2^n possible combinations, called *maxterms*, or *standard sums*.

- Any 2^n maxterms for n variables may be determined similarly.
- It is important to note that (1) each maxterm is obtained from an OR term of the n variables, with each variable being unprimed if the corresponding bit is a 0 and primed if a 1, and (2) each maxterm is the complement of its corresponding minterm and vice versa.

$$\overline{m_j} = M_j \quad \text{or} \quad \overline{M_j} = m_j$$

Minterms and Maxterms for Three Binary Variables

| | | | Minterms | | Maxterms | |
|---|---|---|----------|-------------|----------------|-------------|
| x | y | z | Term | Designation | Term | Designation |
| 0 | 0 | 0 | $x'y'z'$ | m_0 | $x + y + z$ | M_0 |
| 0 | 0 | 1 | $x'y'z$ | m_1 | $x + y + z'$ | M_1 |
| 0 | 1 | 0 | $x'yz'$ | m_2 | $x + y' + z$ | M_2 |
| 0 | 1 | 1 | $x'yz$ | m_3 | $x + y' + z'$ | M_3 |
| 1 | 0 | 0 | $xy'z'$ | m_4 | $x' + y + z$ | M_4 |
| 1 | 0 | 1 | $xy'z$ | m_5 | $x' + y + z'$ | M_5 |
| 1 | 1 | 0 | xyz' | m_6 | $x' + y' + z$ | M_6 |
| 1 | 1 | 1 | xyz | m_7 | $x' + y' + z'$ | M_7 |

- A Boolean function can be expressed algebraically from a given truth table
- by forming a minterm for each combination of the variables that produces a 1 in the function and then taking the OR of all those minterms.
- by form a maxterm for each combination of the variables that produces a 0 in the function, and then form the AND of all those maxterms.

Example:

| x | y | z | Function f_1 | Function f_2 |
|---|---|---|----------------|----------------|
| 0 | 0 | 0 | 0 | 0 |
| 0 | 0 | 1 | 1 | 0 |
| 0 | 1 | 0 | 0 | 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 |

$$\begin{aligned}f_1 &= x'y'z + xy'z' + xyz \\&= m_1 + m_4 + m_7\end{aligned}$$

$$\begin{aligned}f_1 &= (x + y + z)(x + y' + z) \\&\quad (x + y' + z')(x' + y + z')(x' + y' + z) \\&= M_0 \cdot M_2 \cdot M_3 \cdot M_5 \cdot M_6\end{aligned}$$

$$\begin{aligned}f_2 &= x'yz + xy'z + xyz' + xyz \\&= m_3 + m_5 + m_6 + m_7\end{aligned}$$

$$\begin{aligned}f_2 &= (x + y + z)(x + y + z') \\&\quad (x + y' + z)(x' + y + z) \\&= M_0 \cdot M_1 \cdot M_2 \cdot M_4\end{aligned}$$

Canonical Forms

- Boolean functions expressed as a sum of minterms or product of maxterms are said to be in *canonical form* .
- For n binary variables, one can obtain 2^n distinct minterms or maxterms and that any Boolean function can be expressed as a sum of minterms or product of maxterms.

Sum of Minterms

- The minterms whose sum defines the Boolean function are those which give the 1's of the function in a truth table.
- If the function is not in this form, it can be made so by first expanding the expression into a sum of AND terms. Each term is then inspected to see if it contains all the variables. If it misses one or more variables, it is ANDed with an expression such as $x + x'$, where x is one of the missing variables.

Product of Maxterms

- The maxterms whose product defines the Boolean function are those which give the 0's of the function in a truth table.
- To express a Boolean function as a product of maxterms, it must first be brought into a form of OR terms. This may be done by using the distributive law, $x + yz = (x + y)(x + z)$. Then any missing variable x in each OR term is ORed with xx' .

Example: Express the Boolean function $F = A + B'C$ as a sum of minterms.

The function has three variables: A , B , and C .

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

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

Combining all terms, we have

$$F = A + B'C = ABC + ABC' + AB'C + AB'C' + AB'C + A'B'C$$

$$F = A'B'C + AB'C' + AB'C + ABC' + ABC$$

$$= m_1 + m_4 + m_5 + m_6 + m_7$$

$$F(A, B, C) = \sum(1, 4, 5, 6, 7)$$

Example: Express the Boolean function

$F = xy + x'z$ as a product of maxterms.

First, convert the function into OR terms

$$\begin{aligned}F &= xy + x'z = (xy + x')(xy + z) = (x + x')(y + x')(x + z)(y + z) \\&= (x' + y)(x + z)(y + z)\end{aligned}$$

The function has three variables: x, y, and z.

$$x' + y = x' + y + zz' = (x' + y + z)(x' + y + z')$$

$$x + z = x + yy' + z = (x + y + z)(x + y' + z)$$

$$y + z = xx' + y + z = (x + y + z)(x' + y + z)$$

Combining all the terms, we have

$$\begin{aligned}F &= (x + y + z)(x + y' + z)(x' + y + z)(x' + y + z') \\&= M_0 \cdot M_2 \cdot M_4 \cdot M_5\end{aligned}$$

$$F(x, y, z) = \prod(0, 2, 4, 5)$$

Conversion between Canonical Forms

- The complement of a function expressed as the sum of minterms equals the sum of minterms missing from the original function.
- This is because the original function is expressed by those minterms which make the function equal to 1, whereas its complement is a 1 for those minterms for which the function is a 0.

Example:

- Consider the function

$$F(A, B, C) = \sum(1, 4, 5, 6, 7)$$

- This function has a complement that can be expressed as

$$F'(A, B, C) = \sum(0, 2, 3) = m_0 + m_2 + m_3$$

- Now, if we take the complement of F' by DeMorgan's theorem, we obtain F in a different form:

$$\begin{aligned} F &= (m_0 + m_2 + m_3)' = m'_0 \cdot m'_2 \cdot m'_3 \\ &= M_0 M_2 M_3 = \prod(0, 2, 3) \end{aligned}$$

- From the definition of minterms and maxterms as shown in Table

$$m'_j = M_j$$

- That is, the maxterm with subscript j is a complement of the minterm with the same subscript j and vice versa.

Standard Forms

- The two canonical forms of Boolean algebra are basic forms that one obtains from reading a given function from the truth table.
- These forms are very seldom the ones with the least number of literals, because each minterm or maxterm must contain, by definition, *all* the variables, either complemented or uncomplemented.

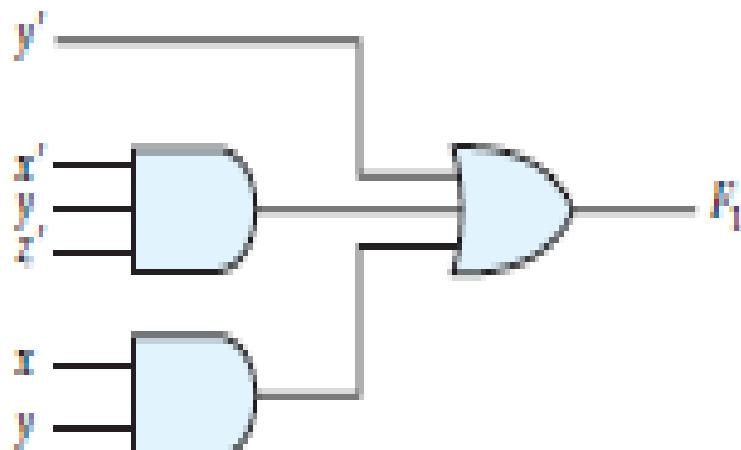
- Another way to express Boolean functions is in *standard* form.
- In this configuration, the terms that form the function may contain one, two, or any number of literals.
- There are two types of standard forms: the **sum of products** and **products of sums**.

- The *sum of products* is a Boolean expression containing AND terms, called *product terms*, with one or more literals each. The *sum* denotes the ORing of these terms.
- An example of a function expressed as a sum of products is $F_1 = y' + xy + x'yz'$
- The expression has three product terms, with one, two, and three literals. Their sum is, in effect, an OR operation.

- A *product of sums* is a Boolean expression containing OR terms, called *sum* terms. Each term may have any number of literals. The *product* denotes the ANDing of these terms.
- An example of a function expressed as a product of sums is $F_2 = x(y' + z)(x' + y + z')$
- This expression has three sum terms, with one, two, and three literals. The product is an AND operation.

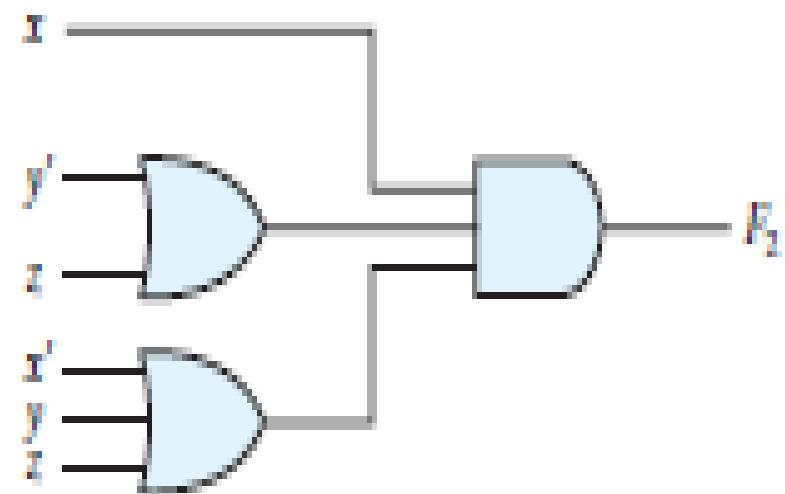
Sum of Products

$$F_1 = y' + xy + x'yz'$$



Product of Sums

$$F_2 = x(y' + z)(x' + y + z')$$



Example: Convert each of the following expressions into sum of products and product of sums:

1. $(AB + C)(B + C'D)$
2. $x' + x(x + y')(y + z')$

Solution:

1. Sum of Products

$$(AB + C)(B + C'D)$$

$$= ABB + ABC'D + BC + CC'D$$

$$= AB + ABC'D + BC$$

$$= AB(1 + C'D) + BC = AB + BC$$

Product of Sums

$$(AB + C)(B + C'D)$$

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

Solution:

2. Sum of Products

$$\begin{aligned} & x' + x(x + y')(y + z') \\ &= x' + x(xy + xz' + y'y + y'z') \\ &= x' + xxy + xxz' + xy'z' \\ &= x' + xy + xz' + xy'z' \end{aligned}$$

Product of Sums

$$\begin{aligned} & x' + x(x + y')(y + z') \\ &= (x' + x)(x' + x + y')(x' + y + z') \\ &= 1 \cdot 1 \cdot (x' + y + z') \\ &= (x' + y + z') \end{aligned}$$

Example: Express each function in sum-of-minterms and product-of-maxterms form.

1. $(xy + z)(y + xz)$
2. $(A' + B)(B' + C)$

Solution:

$$\begin{aligned}1. \quad & (xy + z)(y + xz) \\&= xy^2 + xyxz + yz + xzz \\&= xy + xyz + yz + xz \\&= xy(z + z') + (x + x')yz + x(y + y')z + xyz \\&= xyz + xyz' + xyz + x'y'z + xyz + xy'z + xyz \\&= x'y'z + xy'z + xyz' + xyz \\&= m_3 + m_5 + m_6 + m_7 \\F(x, y, z) &= \sum(3, 5, 6, 7) = \prod(0, 1, 2, 4)\end{aligned}$$

Solution:

$$2. \quad (A' + B)(B' + C)$$

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

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

$$= M_4 \cdot M_5 \cdot M_2 \cdot M_6$$

$$F(A, B, C) = \prod(2, 4, 5, 6) = \sum(0, 1, 3, 7)$$

or

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

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

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

$$= m_1 + m_0 + m_3 + m_7$$

$$F(A, B, C) = \sum(0, 1, 3, 7) = \prod(2, 4, 5, 6)$$

Example: Convert each of the following to the other canonical form:

1. $F(x, y, z) = \Sigma(1, 3, 5)$
2. $F(x, y, z) = \Pi(0, 3, 6, 7)$
3. $F(A, B, C, D) = \Sigma(0, 2, 6, 11, 13, 14)$
4. $F(A, B, C, D) = \Pi(0, 1, 3, 4, 6, 8, 11, 12)$

Solution:

1. $F(x, y, z) = \sum(1, 3, 5)$
 $= \prod(0, 2, 4, 6, 7)$
2. $F(x, y, z) = \prod(0, 3, 6, 7)$
 $= \sum(1, 2, 4, 5)$
3. $F(A, B, C, D) = \sum(0, 2, 6, 11, 13, 14)$
 $= \prod(1, 3, 4, 5, 7, 8, 9, 10, 12, 15)$
4. $F(A, B, C, D) = \prod(0, 1, 3, 4, 6, 8, 11, 12)$
 $= \sum(2, 5, 7, 9, 10, 13, 14, 15)$

Example: Express the complement of the following functions in sum-of-minterms form:

1. $F(x, y, z) = \Sigma(0, 3, 6, 7)$
2. $F(A, B, C, D) = \Sigma(2, 4, 7, 10, 12, 14)$
3. $F(A, B, C) = \Pi(3, 5, 7)$
4. $F(w, x, y, z) = \Pi(0, 1, 3, 4, 6, 8, 11, 12)$

Solution:

$$1. \quad F(x, y, z) = \sum(0, 3, 6, 7)$$

$$F'(x, y, z) = \sum(1, 2, 4, 5)$$

$$2. \quad F(A, B, C, D) = \sum(2, 4, 7, 10, 12, 14)$$

$$F'(A, B, C, D) = \sum(0, 1, 3, 5, 6, 8, 9, 11, 13, 15)$$

$$3. \quad F(A, B, C) = \prod(3, 5, 7) = \sum(0, 1, 2, 4, 6)$$

$$F'(A, B, C) = \sum(3, 5, 7)$$

$$4. \quad F(w, x, y, z) = \prod(0, 1, 3, 4, 6, 8, 11, 12)$$

$$F'(w, x, y, z) = \sum(0, 1, 3, 4, 6, 8, 11, 12)$$

DIGITAL ELECTRONICS

III Semester CSE & CST



Mr. Srikanta Patnaik

Mr. Ajit Kumar Patro

Mr. Ami Kumar Parida

DEPARTMENT OF ELECTRONICS AND COMMUNICATION ENGINEERING

SCHOOL OF ENGINEERING & TECHNOLOGY

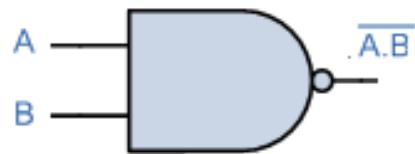
GIET UNIVERSITY, GUNUPUR, ODISHA

NAND AND NOR IMPLEMENTATION

Universal Gates

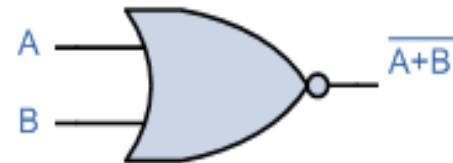
- A **universal gate** is a gate that can implement any Boolean function without the need to use any other gate type.
- The **NAND** and **NOR** gates are universal gates.
- The NAND and NOR gates are said to be *universal* gates because any logic circuit can be implemented with it.

NAND Gate



| INPUT | | OUTPUT |
|-------|---|--------|
| A | B | Q |
| 0 | 0 | 1 |
| 0 | 1 | 1 |
| 1 | 0 | 1 |
| 1 | 1 | 0 |

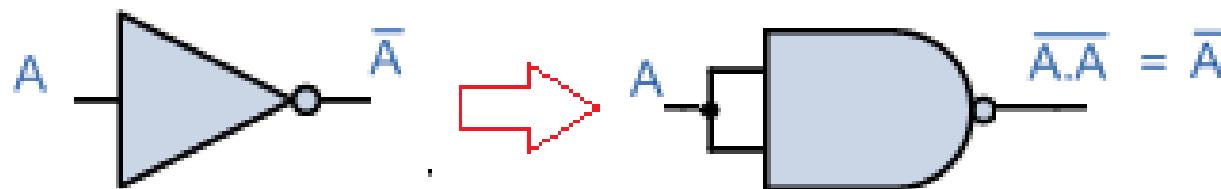
NOR Gate



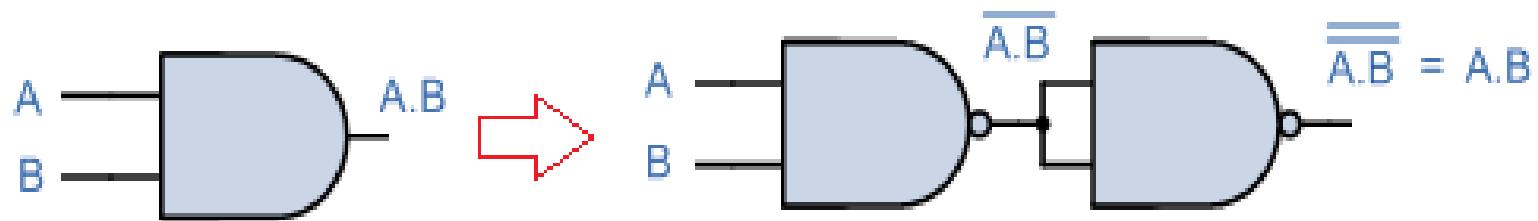
| INPUT | | OUTPUT |
|-------|---|--------|
| A | B | Q |
| 0 | 0 | 1 |
| 0 | 1 | 0 |
| 1 | 0 | 0 |
| 1 | 1 | 0 |

Logic Gates using only NAND Gates

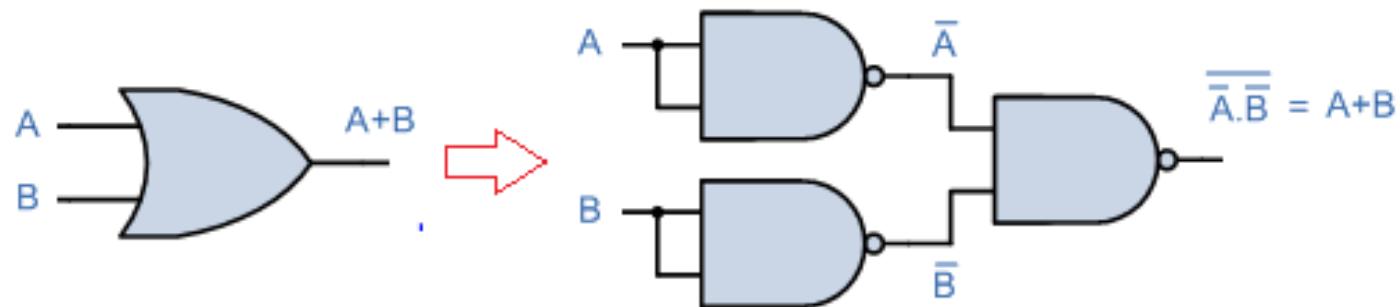
- NOT Gate



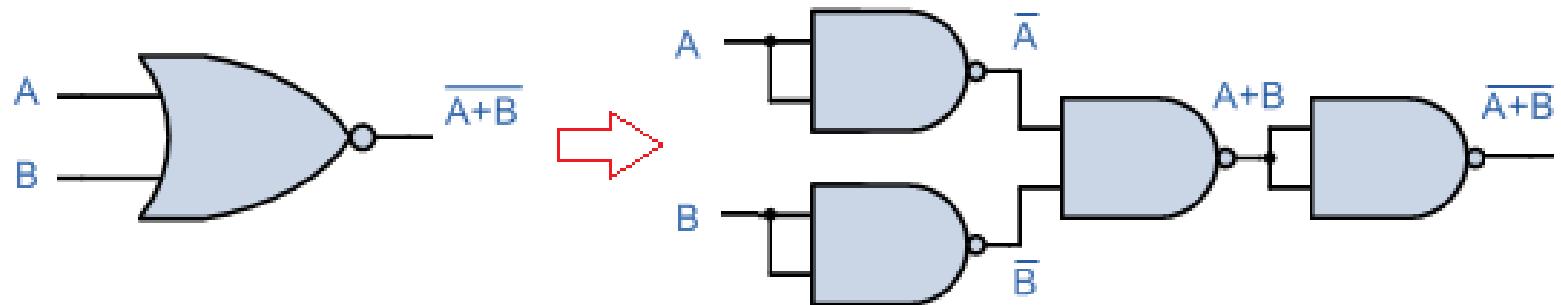
- AND Gate



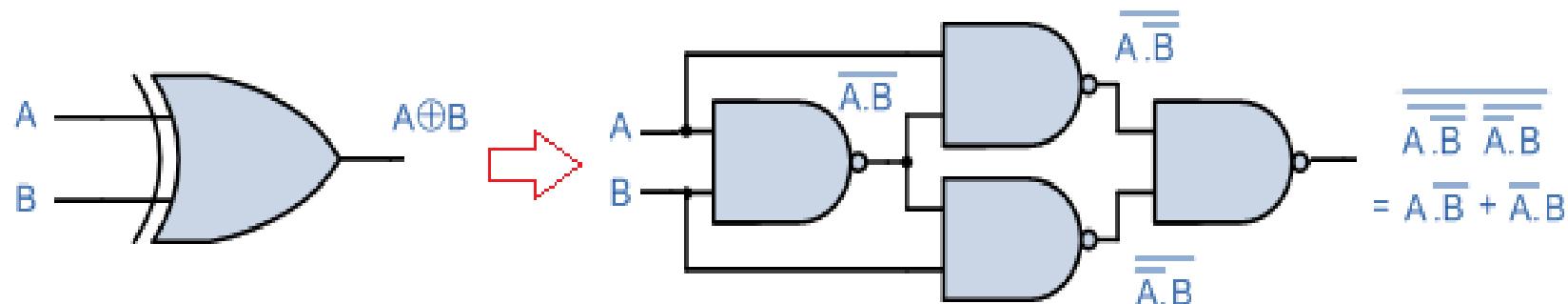
- OR Gate



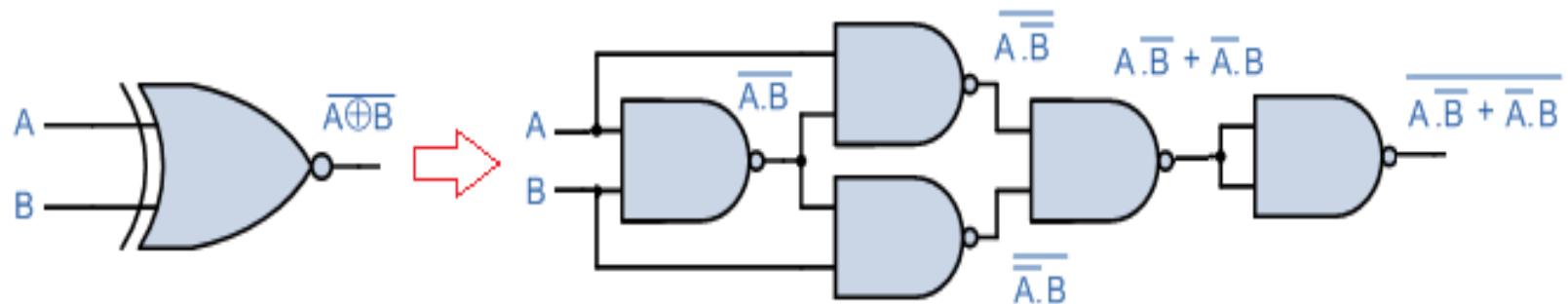
- NOR Gate



- EX-OR Gate

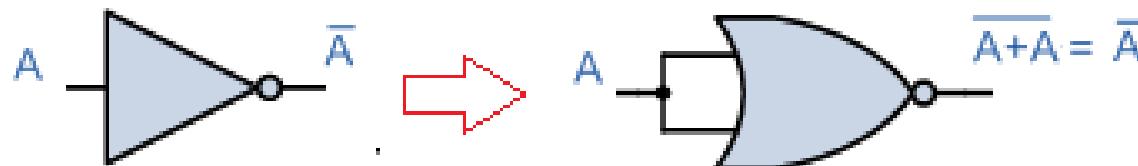


- EX-NOR Gate

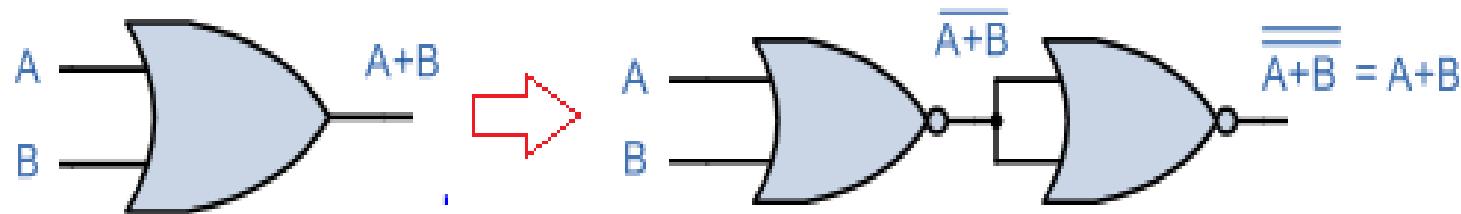


Logic Gates using only NOR Gates

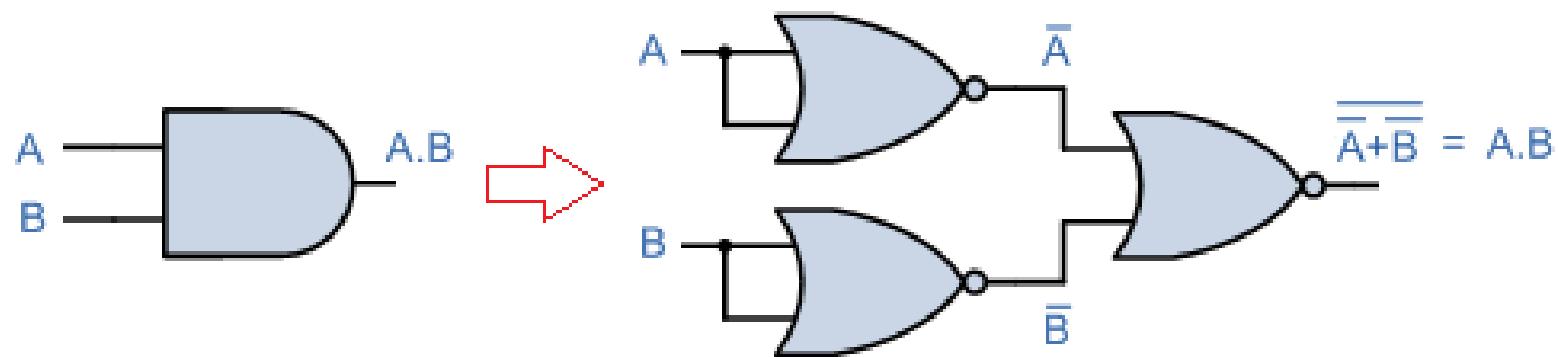
- NOT Gate



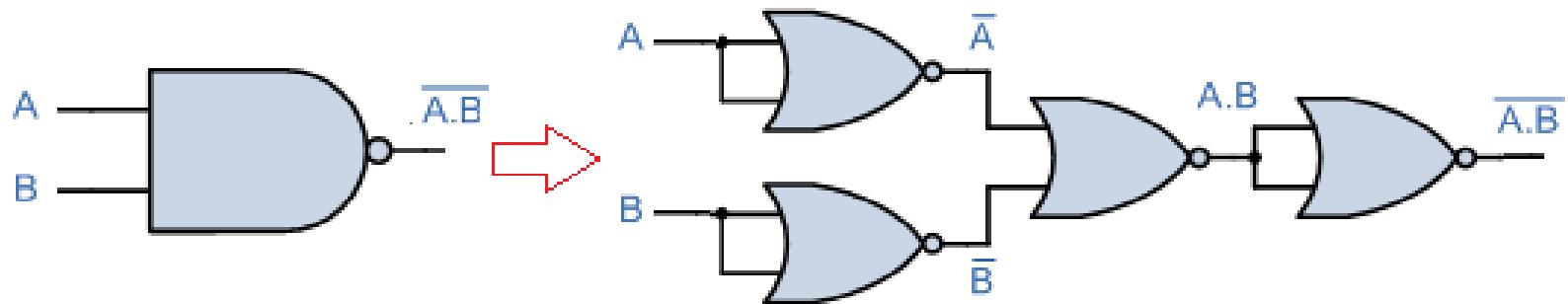
- OR Gate



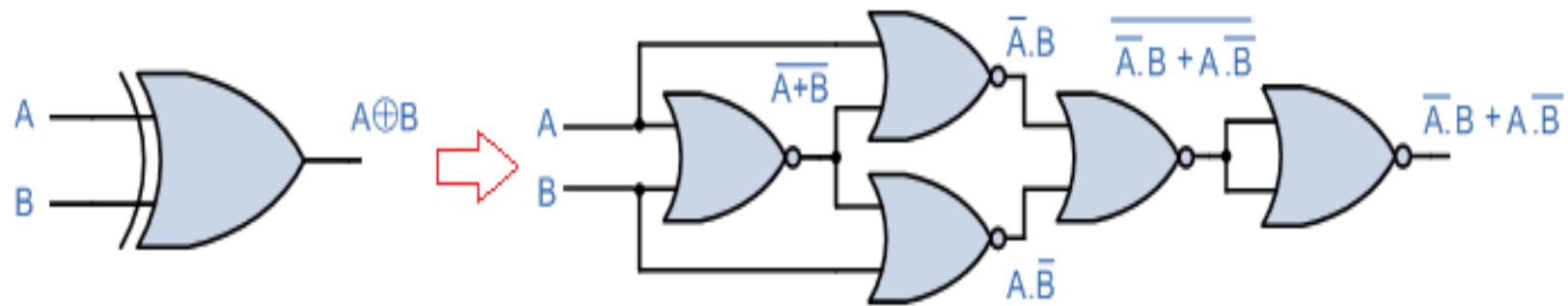
- AND Gate



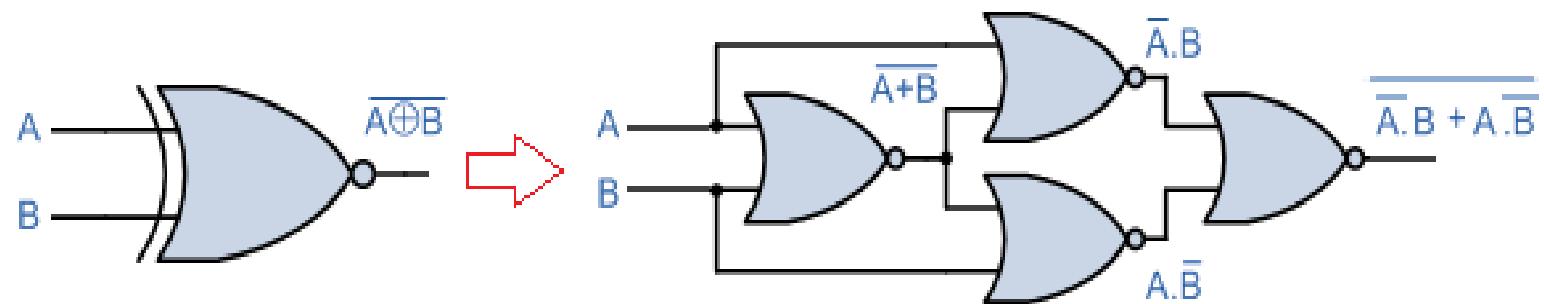
- NAND Gate



- EX-OR Gate



- EX-NOR Gate



Equivalent Gates

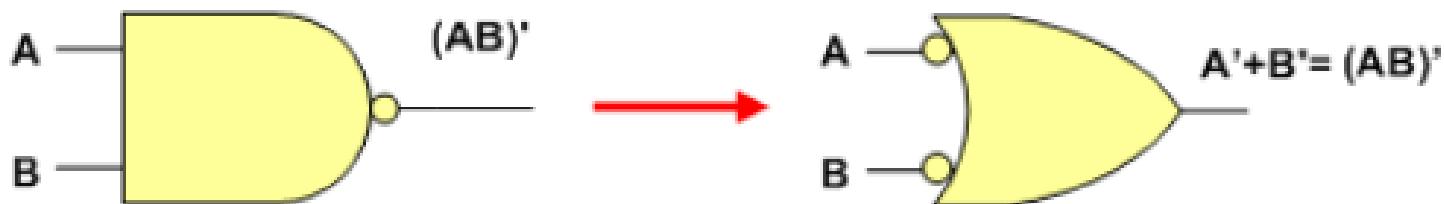
- Note that a bubble denotes complementation (inverter) and two bubbles along the same line represent double complementation, so both can be removed.
- Two NOT gates in series are same as a buffer because they cancel each other as $\bar{\bar{A}} = A$



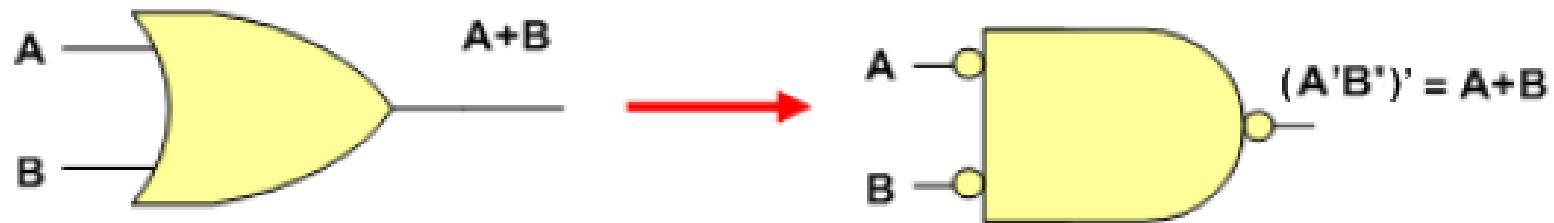
- An **AND** gate is equivalent to an **inverted-input NOR** gate.



- A **NAND** gate is equivalent to an **inverted-input OR** gate.



- An **OR** gate is equivalent to an **inverted-input NAND** gate.



- A **NOR** gate is equivalent to an **inverted-input AND** gate.



NAND Implementation

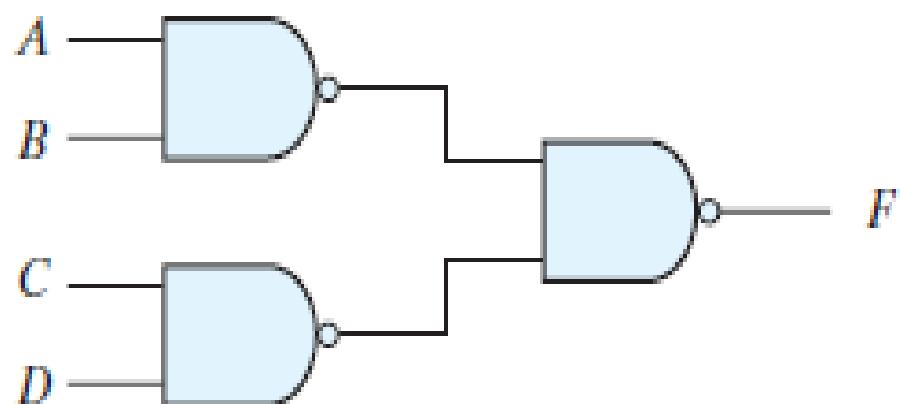
- The implementation of Boolean functions with NAND gates requires that the functions be in sum-of-products form.

$$F = AB + CD$$

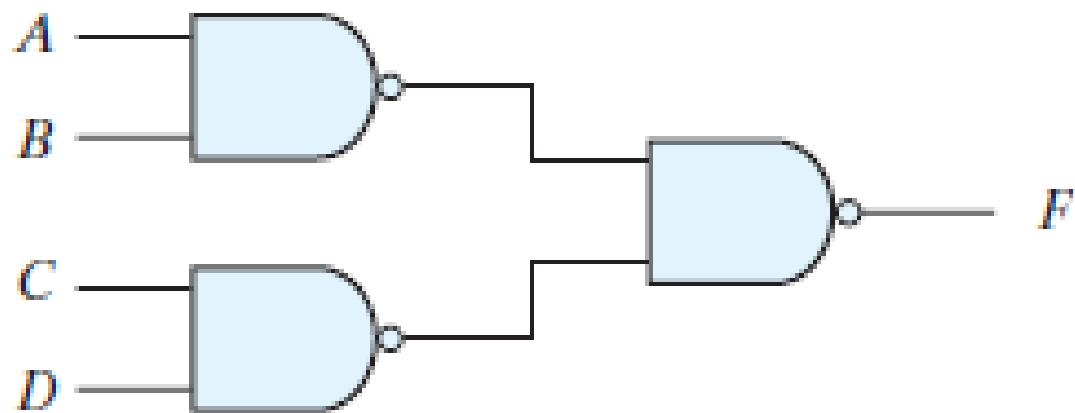
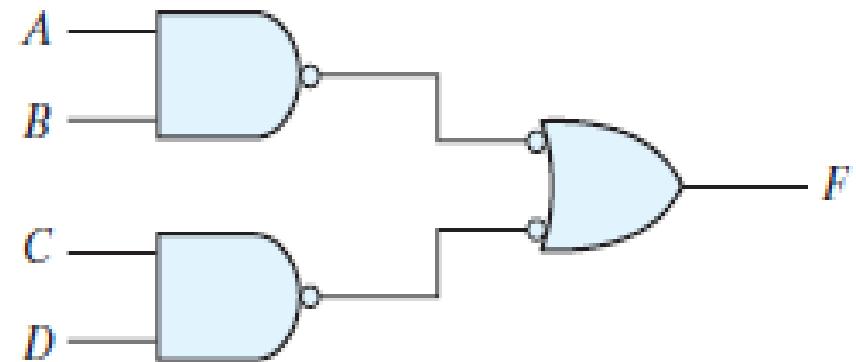
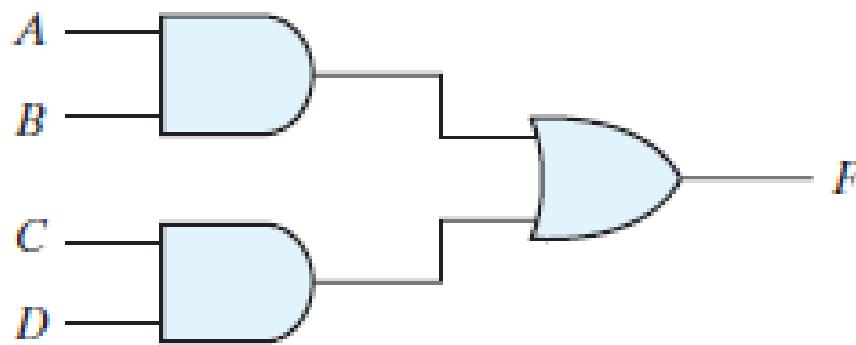
$$F = AB + CD$$

$$= \overline{\overline{AB} + \overline{CD}}$$

$$= \overline{\overline{AB}} \cdot \overline{\overline{CD}}$$



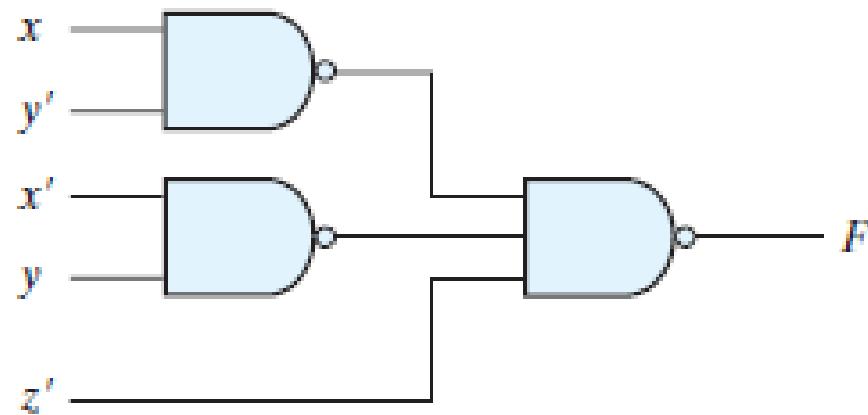
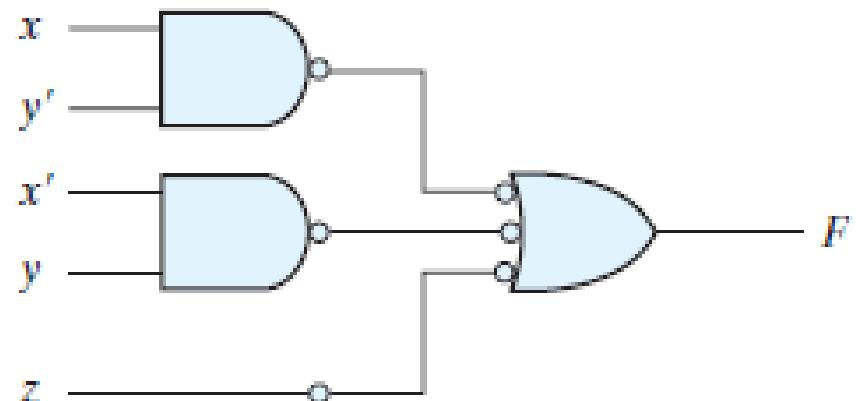
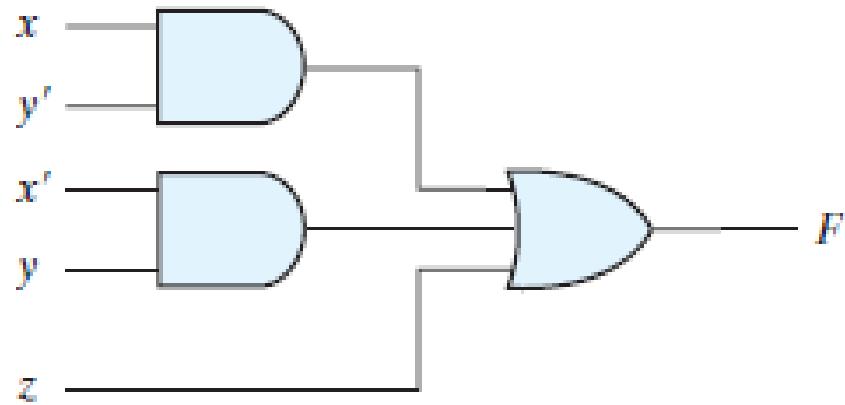
Three ways to implement $F = AB + CD$



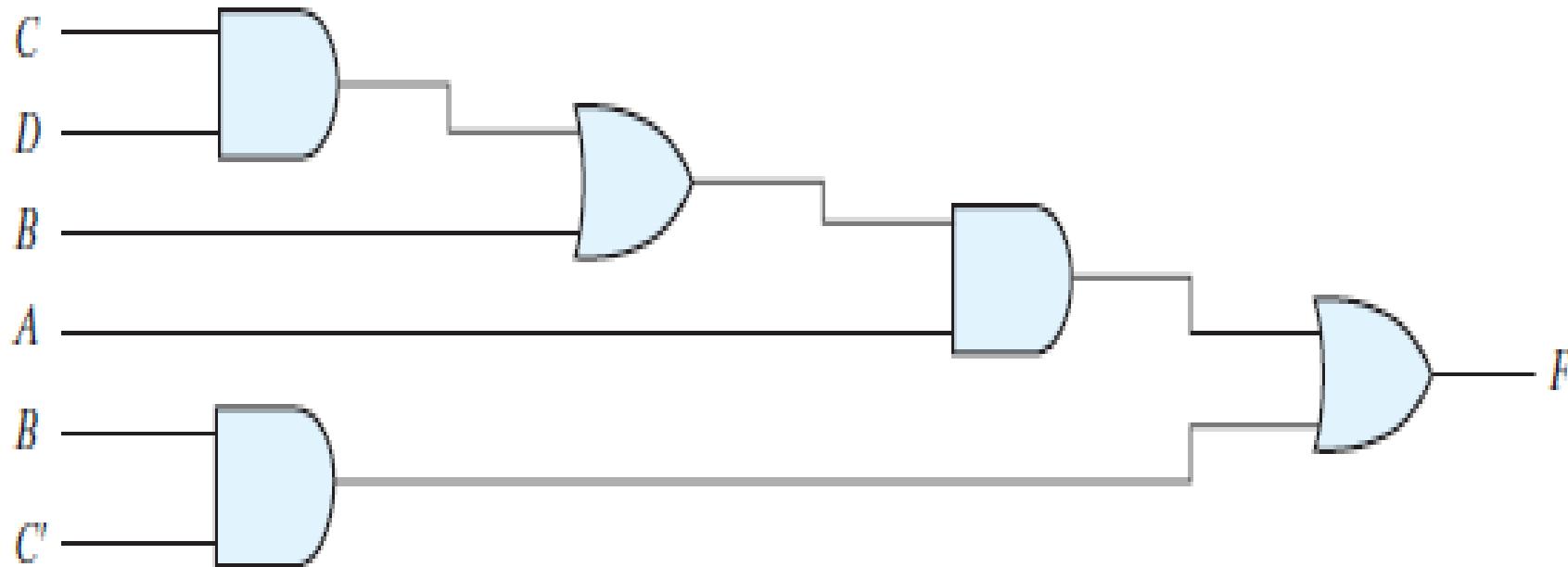
- The general procedure for implementation of a Boolean function with NAND gates using mixed notation is as follows:
 1. Draw the Boolean function by basic gates such as AND, OR and NOT gates.
 2. Convert all AND gates to NAND gates with AND-invert graphic symbols.
 3. Convert all OR gates to NAND gates with invert-OR graphic symbols.
 4. Check all the bubbles in the diagram. For every bubble that is not compensated by another small circle along the same line, insert an inverter (a one-input NAND gate) or complement the input literal.

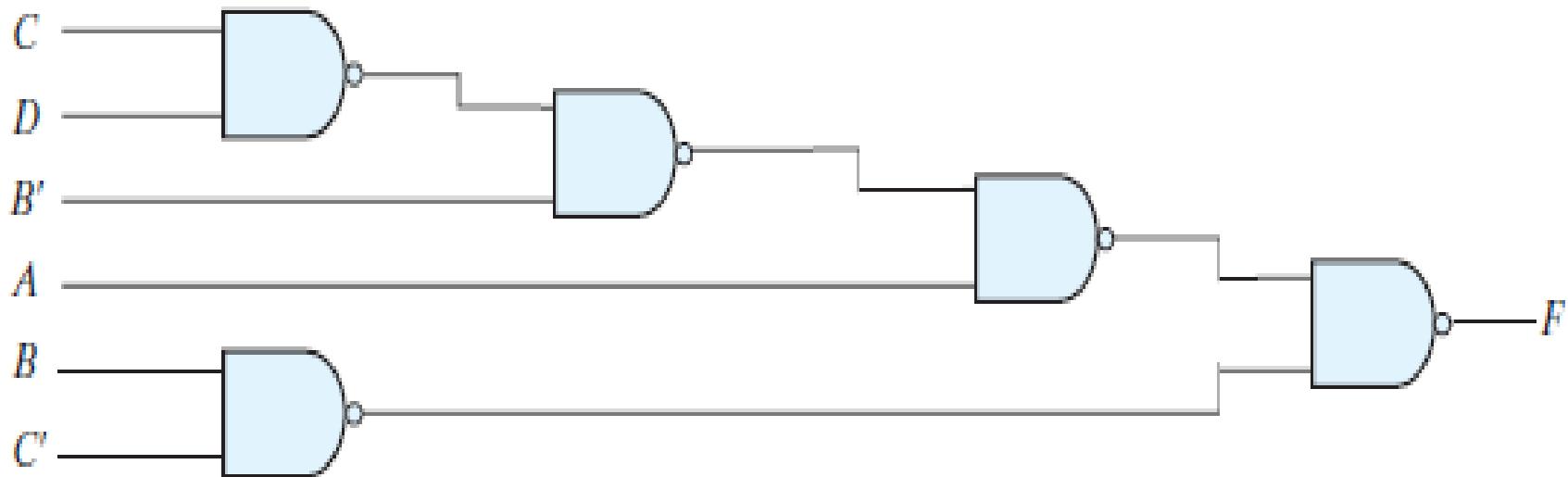
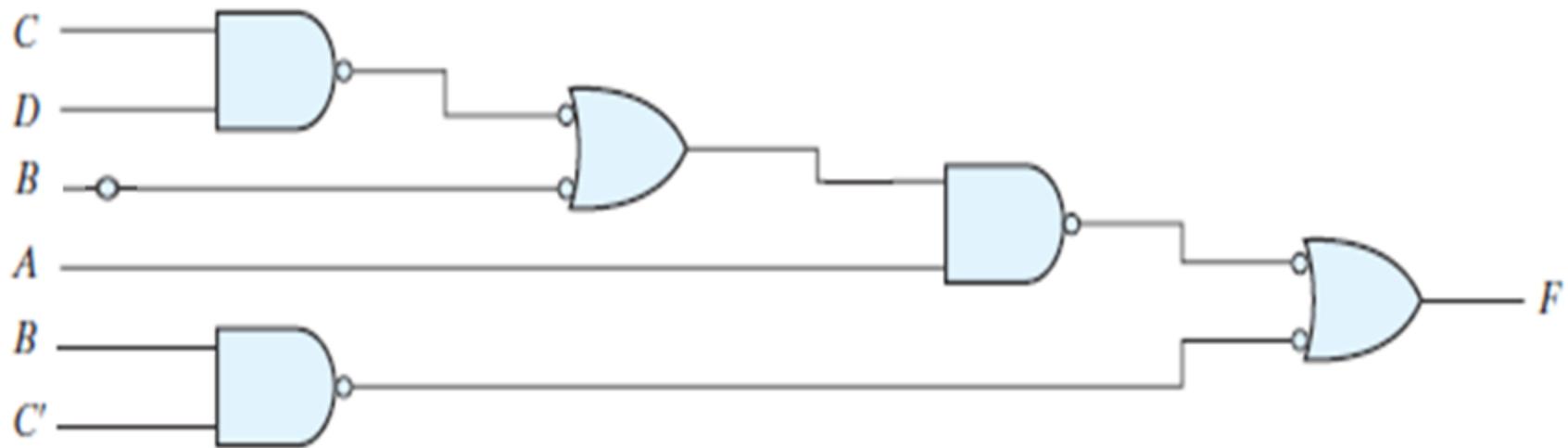
Example: Implement the following Boolean function with NAND gates:

$$F = xy' + x'y + z$$

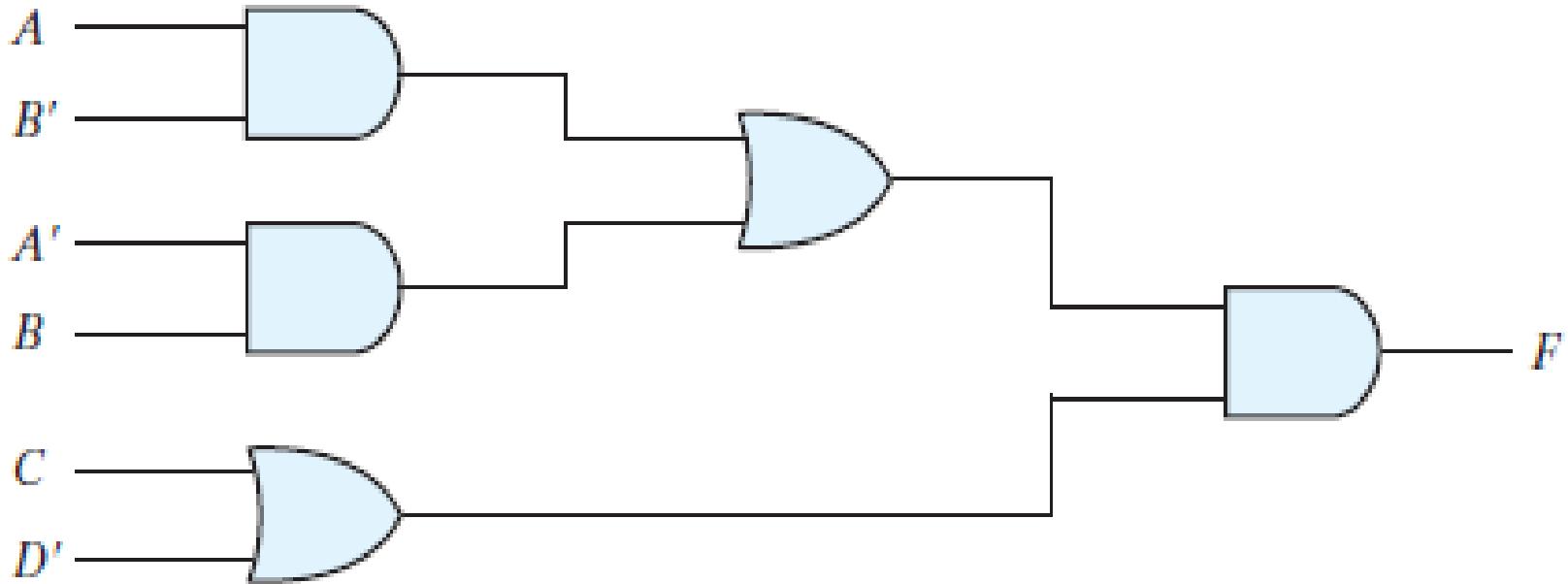


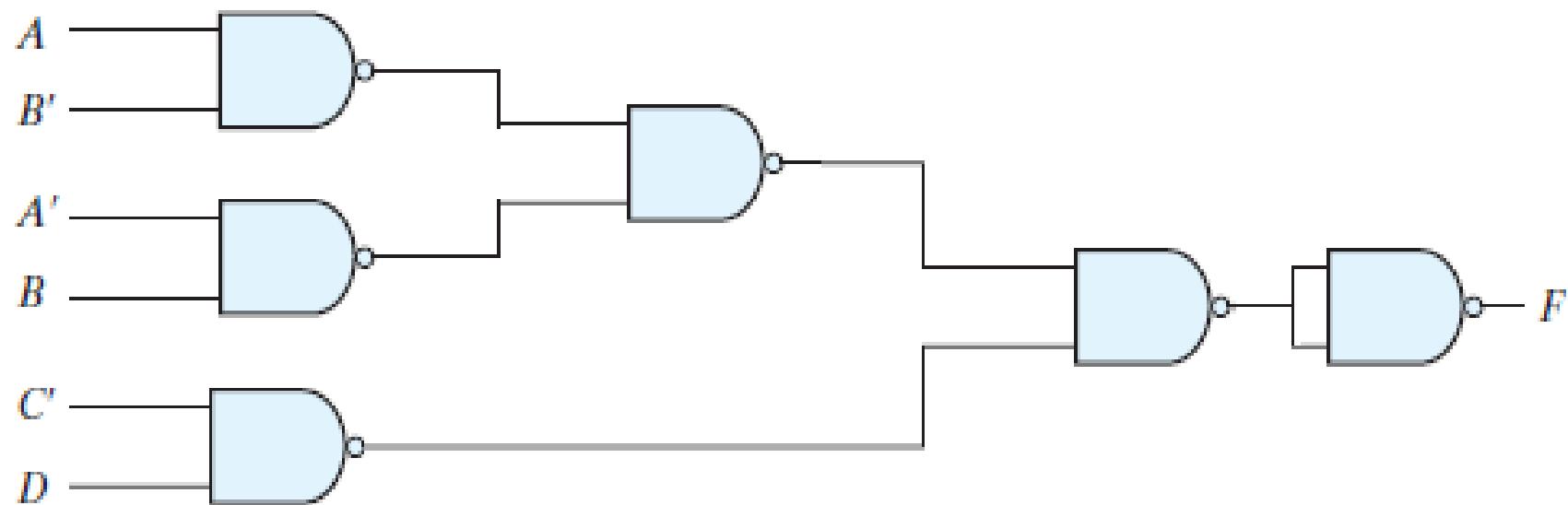
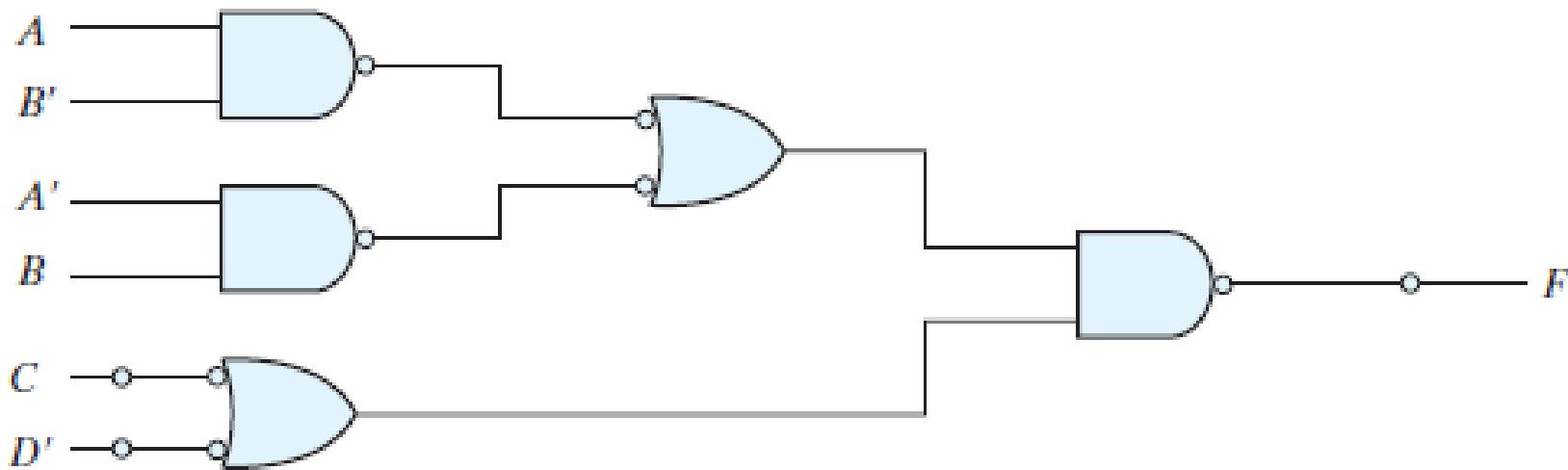
Example: Implement the following Boolean function with NAND gates:

$$F = A(CD + B) + BC'$$




Example: Implement the following Boolean function with NAND gates:

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




NOR Implementation

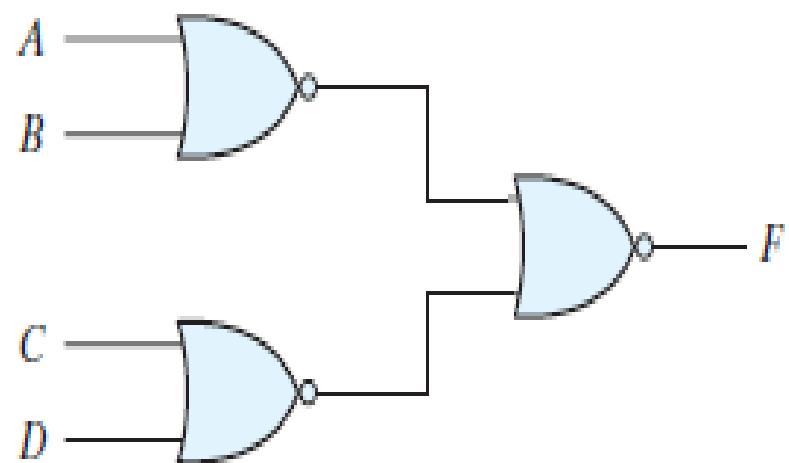
- The implementation of Boolean functions with NOR gates requires that the functions be in product-of-sums form.

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

$$F = (A + B) \cdot (C + D)$$

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

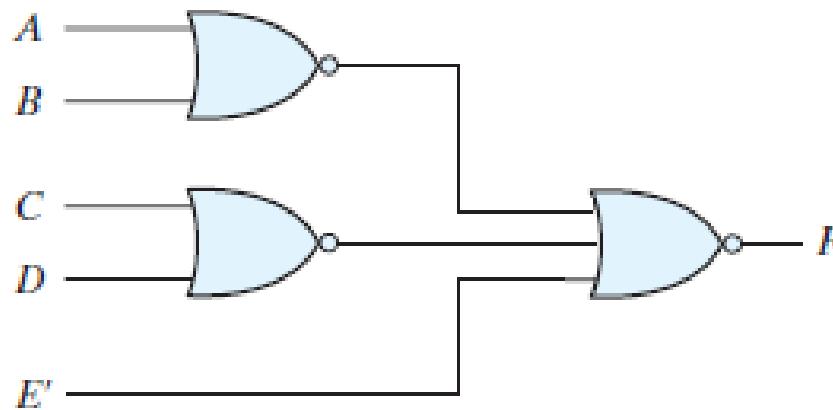
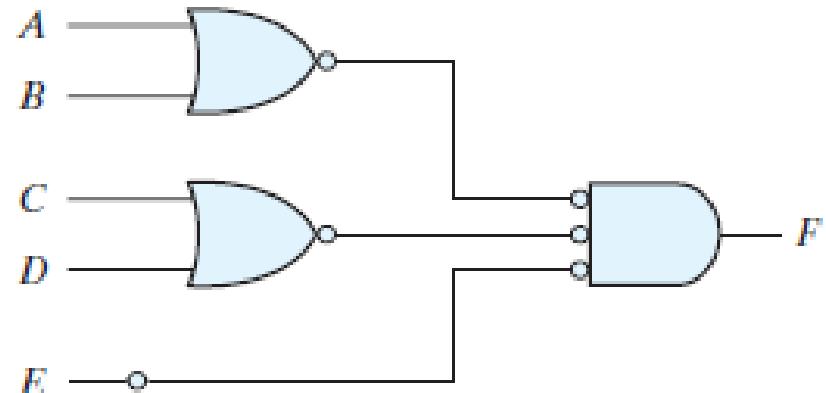
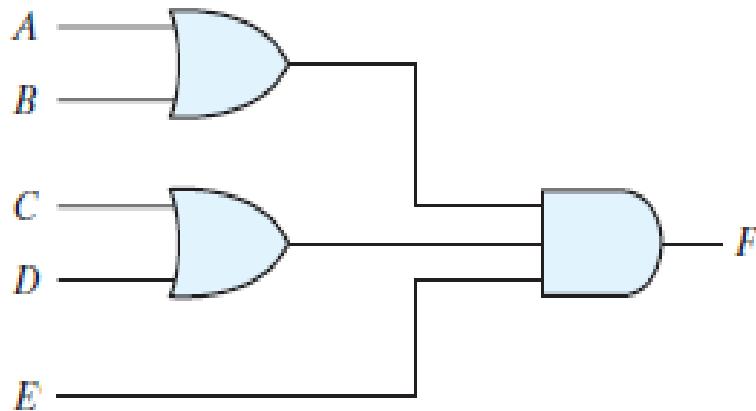
$$= \overline{(A + B)} + \overline{(C + D)}$$



- The general procedure for implementation of a Boolean function with NOR gates using mixed notation is as follows:
 1. Draw the Boolean function by basic gates such as AND, OR and NOT gates.
 2. Convert all OR gates to NOR gates with OR-invert graphic symbols.
 3. Convert all AND gates to NOR gates with invert-AND graphic symbols.
 4. Check all the bubbles in the diagram. For every bubble that is not compensated by another small circle along the same line, insert an inverter (a one-input NOR gate) or complement the input literal.

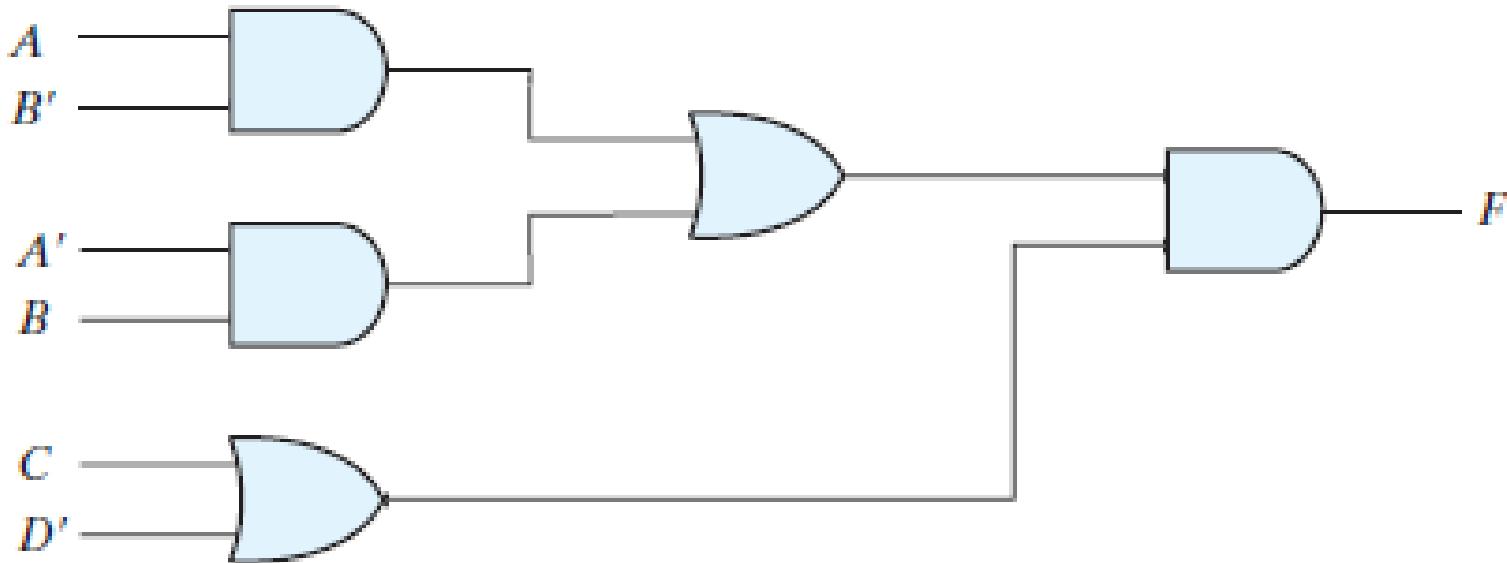
Example: Implement the following Boolean function with NOR gates:

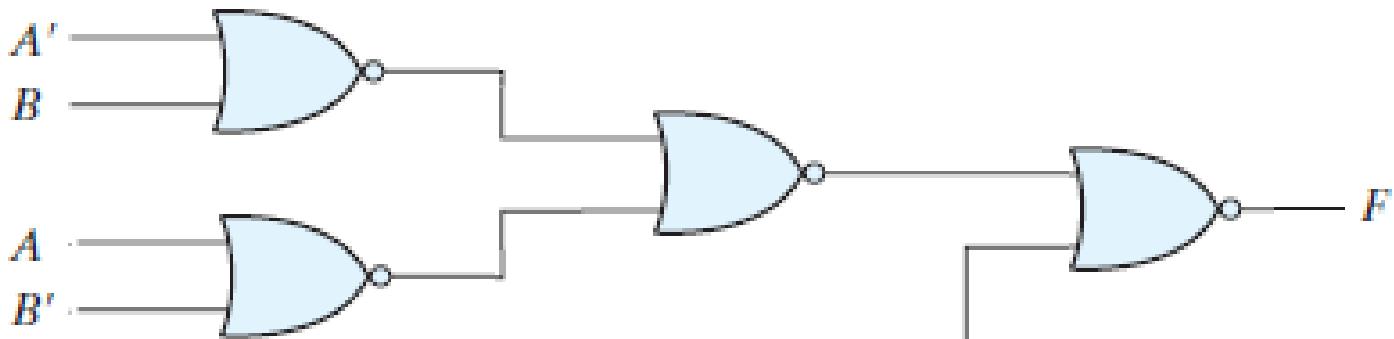
$$F = (A + B)(C + D)E$$



Example: Implement the following Boolean function with NOR gates:

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





DIGITAL ELECTRONICS

III Semester CSE & CST



Mr. Srikanta Patnaik

Mr. Ajit Kumar Patro

Mr. Ami Kumar Parida

DEPARTMENT OF ELECTRONICS AND COMMUNICATION ENGINEERING

SCHOOL OF ENGINEERING & TECHNOLOGY

GIET UNIVERSITY, GUNUPUR, ODISHA

OTHER TWO-LEVEL IMPLEMENTATIONS

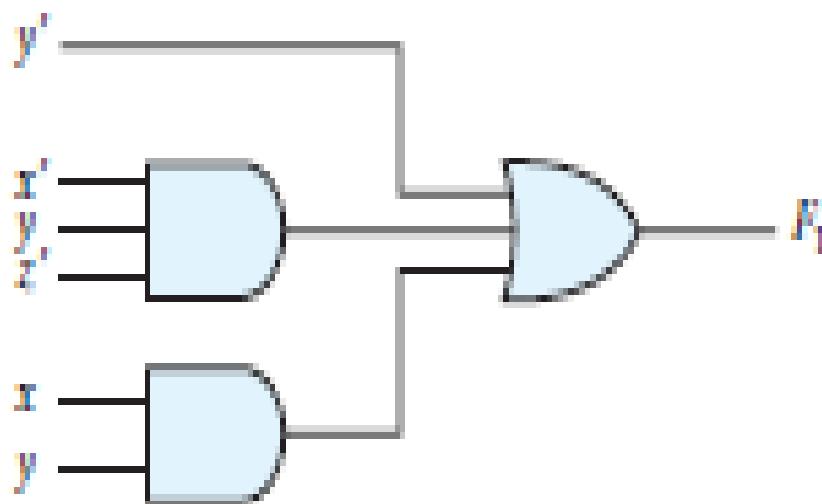
Two-level Logic

- Two level logic means that the logic design uses maximum two logic gates between input and output.
- This does not mean that the whole design will contain only two logic gates but the single path from input to output may contain no more than two logic gates.

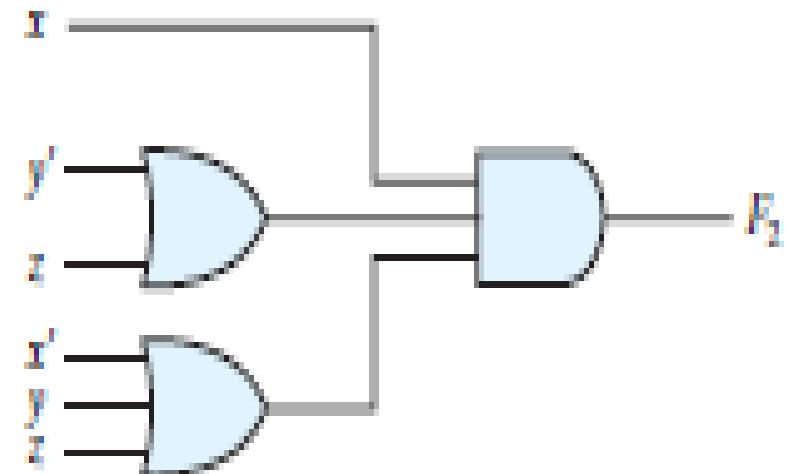
Two-level Implementation

$$F_1 = y' + xy + x'yz'$$

$$F_2 = x(y' + z)(x' + y + z)$$



Sum of Products

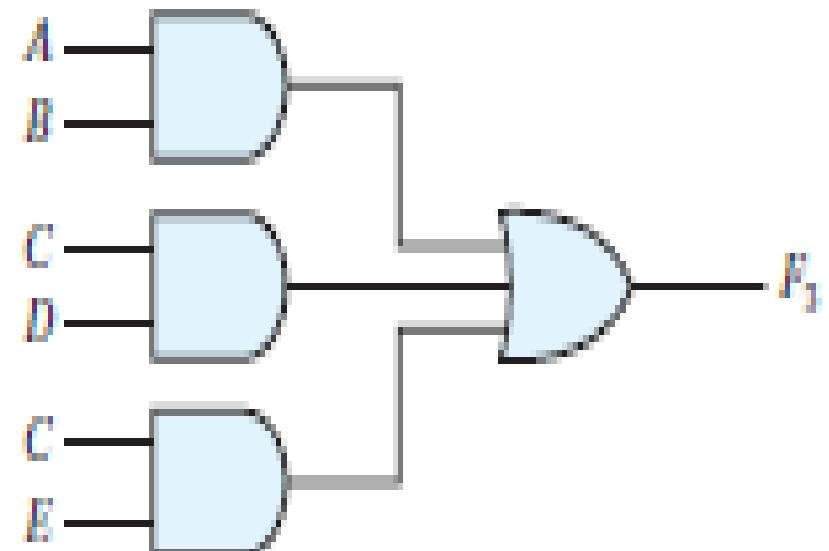
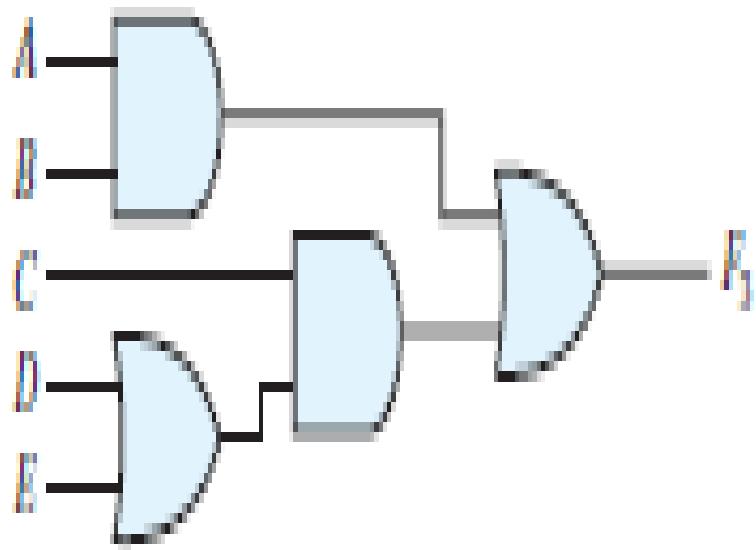


Product of Sums

Multi-level Implementation

$$F_3 = AB + C(D + E)$$

$$\begin{aligned} F_3 &= AB + C(D + E) \\ &= AB + CD + CE \end{aligned}$$



- The standard type (SOP / POS) circuit configuration is referred to as a *two-level implementation*.
- In general, a *two-level implementation* is preferred because it produces the least amount of delay through the gates when the signal propagates from the inputs to the output.
- However, the number of inputs to a given gate might not be practical.

Other Two-Level Implementations

- For two-level logic implementation, we consider four types of gates: **AND**, **OR**, **NAND**, and **NOR**.
- If we assign one type of gate for the first level and one type for the second level, we find that there are 16 possible combinations of two-level forms.

- Each two-level combination implements different logic functions. There are two main types in these 16 combinations.
 - Degenerate Form
 - Non-Degenerate Form

Degenerate Form

- The two-level combination that degenerates into a single logic function as known as degenerate form.
- There are 8 degenerate forms in those 16 combinations. Each of these degenerate forms is given below.

AND-AND

OR-OR

AND-NAND

OR-NOR

NAND-NOR

NOR-NAND

NAND-OR

NOR-AND

Non-Degenerate Form

- Those combinations of Two-level logic, which implements Sum of Product form or Product of sum form are Non-degenerate forms.
- The remaining 8 of the total 16 are all non-degenerate forms which are given below.

AND-OR

OR-AND

NAND-NAND

NOR-NOR

AND-NOR

OR-NAND

NAND-AND

OR-NAND

AOI Logic and OAI Logic

- The logic function

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

is called an AND-OR-INVERT function.

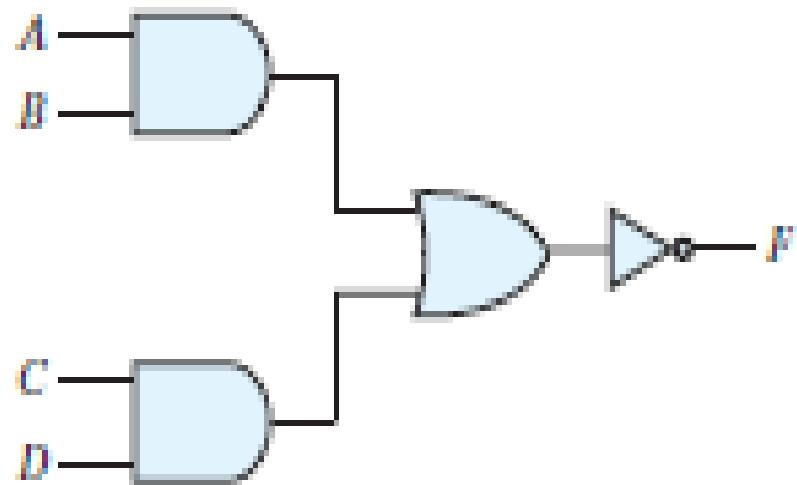
- The logic function

$$F = [(A + B)(C + D)]' = (A + B)' + (C + D)'$$

is called an OR-AND-INVERT function.

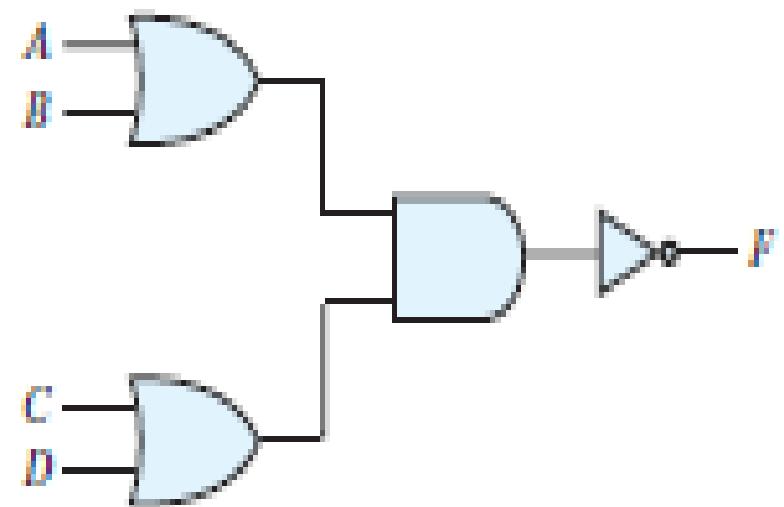
AND-OR-INVERT function

$$F = (AB + CD)'$$



OR-AND-INVERT function

$$F = [(A + B)(C + D)]'$$

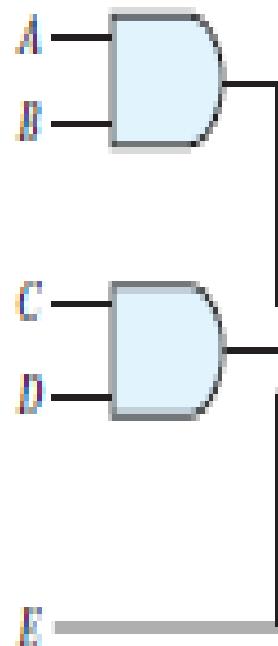


AND–OR–INVERT Implementation

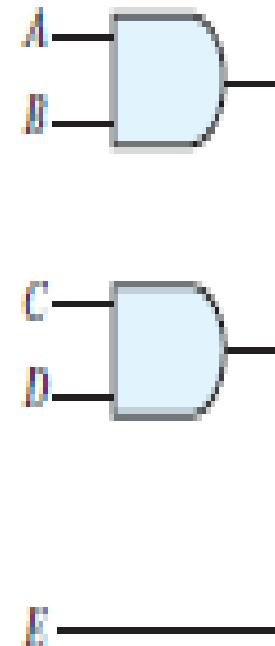
- The two forms, NAND–AND and AND–NOR, are equivalent and can be treated together. Both perform the AND–OR–INVERT function.
- The AND–NOR form resembles the AND–OR form, but with an inversion done by the bubble in the output of the NOR gate.
- It implements the function $F = (AB + CD + E)'$
- An AND–OR implementation requires an expression in sum-of-products form. The AND–OR–INVERT implementation is similar, except for the inversion.

AND-OR-INVERT circuits,

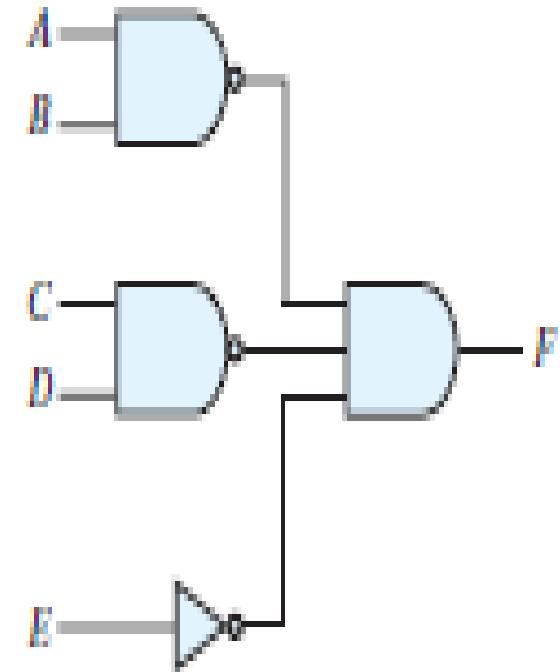
$$F = (AB + CD + E)'$$



(a) AND-NOR



(b) AND-NOR



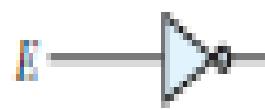
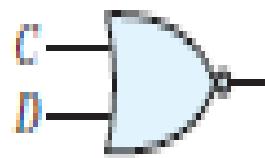
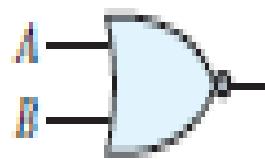
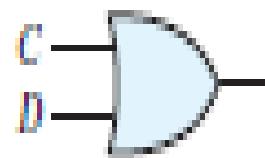
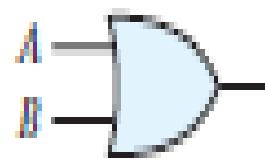
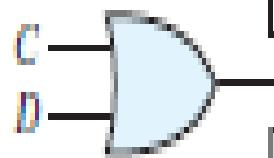
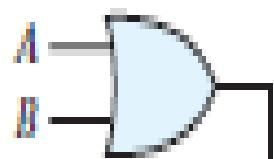
(c) NAND-AND

OR-AND-INVERT Implementation

- The OR-NAND and NOR-OR forms perform the OR-AND-INVERT function.
- The OR-NAND form resembles the OR-AND form, except for the inversion done by the bubble in the NAND gate.
- It implements the function $F = [(A + B)(C + D)E]'$
- The OR-AND-INVERT implementation requires an expression in product-of-sums form.

OR-AND-INVERT circuits,

$$F = [(A + B)(C + D)E]'$$



(a) OR-NAND

(b) OR-NAND

(c) NOR-OR

Implementation with Other Two-Level Forms

- Table summarizes the procedures for implementing a Boolean function in any one of the four 2-level forms.

| Equivalent Nondegenerate Form | | Implements the Function | Simplify F' into | To Get an Output of |
|----------------------------------|----------|-------------------------------|---|---------------------------|
| (a) | (b)* | | | |
| AND-NOR | NAND-AND | AND-OR-INVERT | Sum-of-products form by combining 0's in the map. | F |
| OR-NAND | NOR-OR | OR-AND-INVERT | Product-of-sums form by combining 1's in the map and then complementing. | F |

*Form (b) requires an inverter for a single literal term.

Example: Implement the function
 $F = x'y'z' + xyz'$ with the four 2-level forms
AND–NOR, NAND–AND, OR–NAND and
NOR–OR.

Solution:

- AND-OR-INVERT form

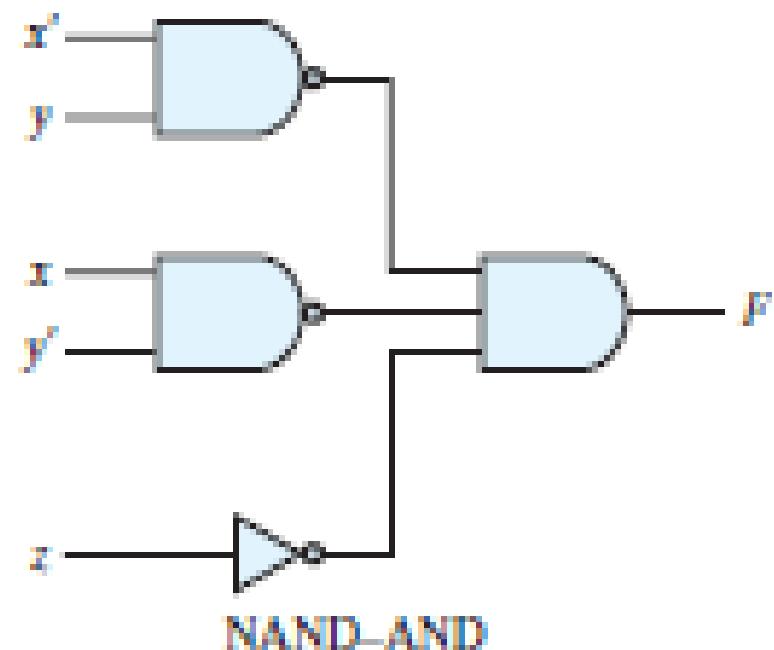
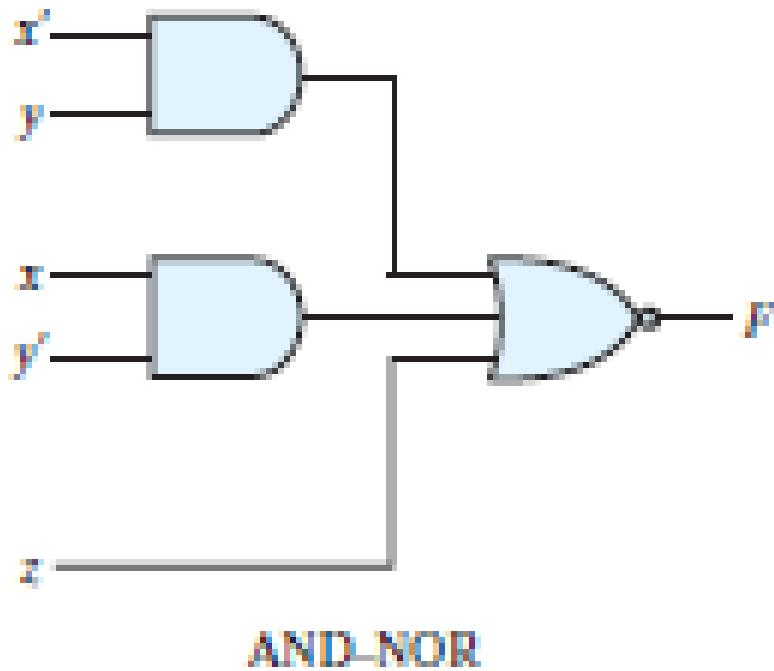
$$F = x'y'z' + xyz'$$

$$\begin{aligned} F' &= (x'y'z' + xyz')' = (x'y'z')' (xyz')' \\ &= (x + y + z) (x' + y' + z) \\ &= xx' + xy' + xz + x'y + yy' + yz + x'z + y'z + zz \\ &= x'y + xy' + xz + x'z + yz + y'z + z \\ &= x'y + xy' + (x + x')z + (y + y')z + z \\ &= x'y + xy' + z + z + z = x'y + xy' + z \\ F' &= x'y + xy' + z \end{aligned}$$

$$F = (x'y + xy' + z)'$$

AND–NOR and NAND–AND Implementations

$$F = x'y'z' + xyz' = (x'y + xy' + z)'$$



- OR-AND-INVERT form

$$F = x'y'z' + xyz'$$

$$F' = (x'y'z' + xyz')'$$

$$= (x'y'z')' (xyz')'$$

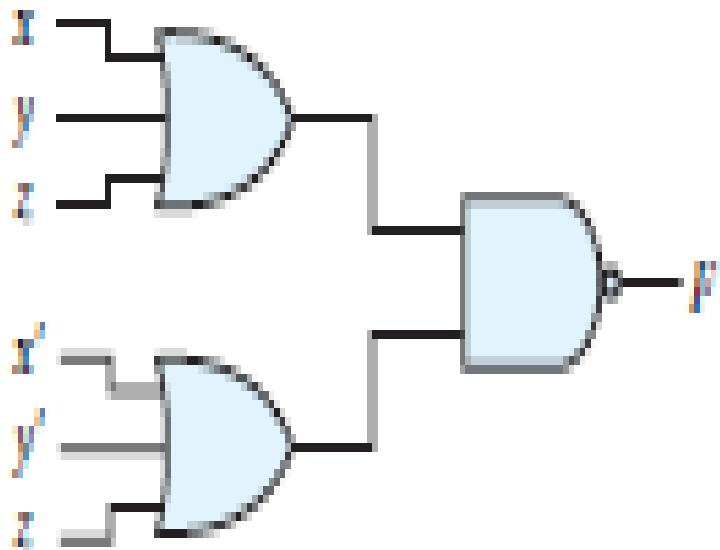
$$= (x + y + z) (x' + y' + z)$$

$$F' = (x + y + z) (x' + y' + z)$$

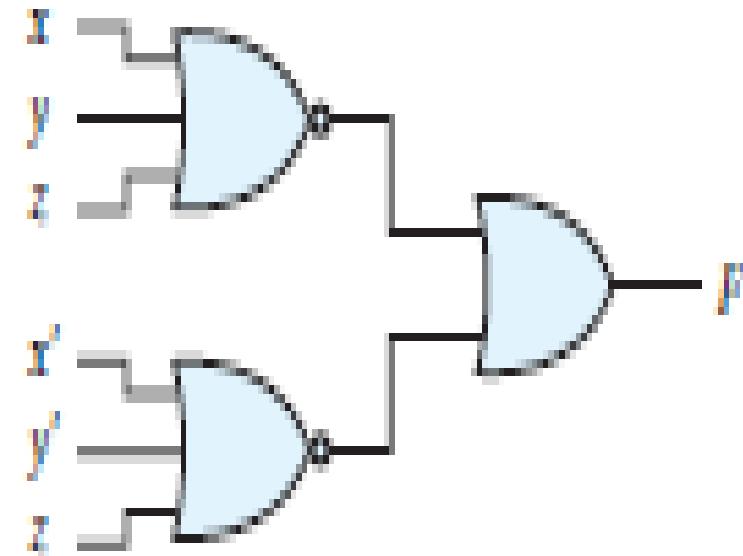
$$F = [(x + y + z) (x' + y' + z)]'$$

OR-NAND and NOR-OR Implementations

$$F = x'y'z' + xyz' = [(x + y + z)(x' + y' + z)]'$$



OR-NAND



NOR-OR

DIGITAL ELECTRONICS

III Semester CSE & CST



Mr. Srikanta Patnaik Mr. Ajit Kumar Patro Mr. Ami Kumar Parida

DEPARTMENT OF ELECTRONICS AND COMMUNICATION ENGINEERING

SCHOOL OF ENGINEERING & TECHNOLOGY

GIET UNIVERSITY, GUNUPUR, ODISHA

K-maps - Two, Three, and Four Variable K-maps, Don't-Care Conditions

THE MAP METHOD

- The map method provides a simple, straightforward procedure for minimizing Boolean functions.
- This method may be regarded as a pictorial form of a truth table.
- The map method is also known as the *Karnaugh map* or *K-map*.
- The simplified expressions produced by the map are always in one of the two standard forms: **sum of products** or **product of sums**.

Two-Variable K-Map

- The two-variable K-map is shown in Fig.
- There are four minterms for two variables; hence, the map consists of four squares, one for each minterm.

| | |
|-------|-------|
| m_0 | m_1 |
| m_2 | m_3 |

(a)

| | | |
|------------------|-----------------|----------------|
| $x \backslash y$ | 0 | 1 |
| 0 | m_0 $x'y'$ | m_1 $x'y$ |
| 1 | m_2 xy' | m_3 xy |

(b)

- The map is redrawn in (b) to show the relationship between the squares and the two variables x and y .
- The 0 and 1 marked in each row and column designate the values of variables.
- Variable x appears **primed** in row 0 and **unprimed** in row 1.
- Similarly, y appears **primed** in column 0 and **unprimed** in column 1.

The rules of K-map simplification are:

- Groupings can contain only 1s; no 0s.
- Groups can be formed only at right angles (horizontal or vertical); diagonal groups are not allowed.
- The number of 1s in a group must be a power of 2 – even if it contains a single 1.
- The groups must be made as large as possible.

- Each cell containing a one must be in at least one group.
- Groups can overlap.
- Groups can wrap around the sides of the K -map.
 - The leftmost cell in a row may be grouped with the rightmost cell and the top cell in a column may be grouped with the bottom cell.

Example: Simplify the Boolean functions

$$F(x, y) = \sum(0, 1) \quad F(x, y) = \sum(2, 3) \quad F(x, y) = \sum(0, 2) \quad F(x, y) = \sum(1, 3)$$

A Karnaugh map for two variables x and y . The horizontal axis is labeled x and the vertical axis is labeled y . The cells are arranged as follows:

| | | |
|---|---|---|
| | 0 | 1 |
| 0 | 1 | 1 |
| 1 | 0 | 0 |

The minterms 0 and 1 are highlighted with red ovals.

A Karnaugh map for two variables x and y . The horizontal axis is labeled x and the vertical axis is labeled y . The cells are arranged as follows:

| | | |
|---|---|---|
| | 0 | 1 |
| 0 | 0 | 0 |
| 1 | 1 | 1 |

The minterms 2 and 3 are highlighted with red ovals.

A Karnaugh map for two variables x and y . The horizontal axis is labeled x and the vertical axis is labeled y . The cells are arranged as follows:

| | | |
|---|---|---|
| | 0 | 1 |
| 0 | 1 | 0 |
| 1 | 1 | 0 |

The minterms 0 and 2 are highlighted with red ovals.

A Karnaugh map for two variables x and y . The horizontal axis is labeled x and the vertical axis is labeled y . The cells are arranged as follows:

| | | |
|---|---|---|
| | 0 | 1 |
| 0 | 0 | 1 |
| 1 | 0 | 1 |

The minterms 1 and 3 are highlighted with red ovals.

$$F(x, y) = x'$$

$$F(x, y) = x$$

$$F(x, y) = y'$$

$$F(x, y) = y$$

Example: Simplify the Boolean functions

$$F(x, y) = \sum(1, 2, 3)$$

$$F(x, y) = \sum(0, 2, 3)$$

$$F(x, y) = \sum(0, 1, 3)$$

$$F(x, y) = \sum(0, 1, 2)$$

| x | y | 0 | 1 |
|---|---|---|---|
| 0 | 0 | 0 | 1 |
| 1 | 1 | 1 | 1 |

| x | y | 0 | 1 |
|---|---|---|---|
| 0 | 0 | 1 | 0 |
| 1 | 1 | 1 | 1 |

| x | y | 0 | 1 |
|---|---|---|---|
| 0 | 0 | 1 | 1 |
| 1 | 0 | 0 | 1 |

| x | y | 0 | 1 |
|---|---|---|---|
| 0 | 0 | 1 | 1 |
| 1 | 1 | 1 | 0 |

$$F(x, y) = x + y$$

$$F(x, y) = x + y'$$

$$F(x, y) = x' + y$$

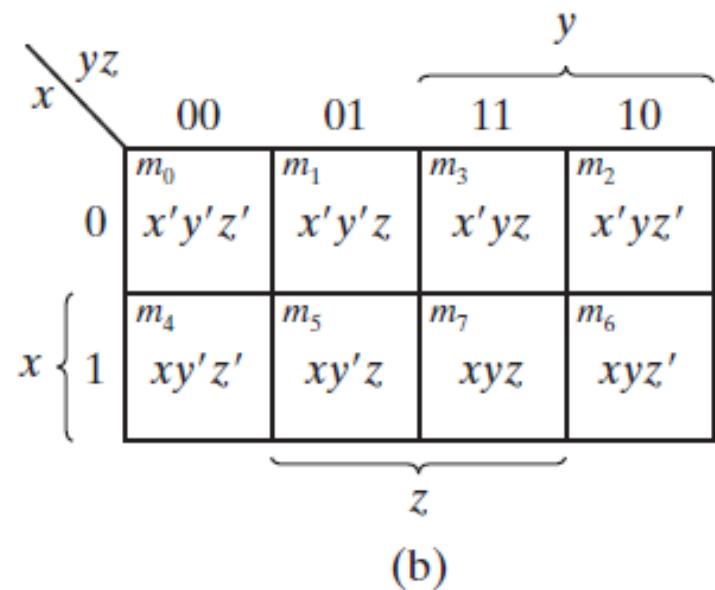
$$F(x, y) = x' + y'$$

Three-Variable K-Map

- A three-variable K-map is shown in Fig.
- There are eight minterms for three binary variables; therefore, the map consists of eight squares.
- Note that the minterms are arranged, **not in a binary sequence, but in a sequence similar to the Gray code**.
- The characteristic of this sequence is that only **one bit changes in value from one adjacent column to the next**.
- The map drawn in part (b) is marked with numbers in each row and each column to show the relationship between the squares and the three variables.

| | | | |
|-------|-------|-------|-------|
| m_0 | m_1 | m_3 | m_2 |
| m_4 | m_5 | m_7 | m_6 |

(a)



(b)

Example: Simplify the Boolean functions

$$F(A, B, C) = \Sigma(0, 4)$$

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

$$F(A, B, C) = \Sigma(1, 3)$$

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

$$F(A, B, C) = \Sigma(4, 5)$$

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

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

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

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

Example: Simplify the Boolean functions

$$F(A, B, C) = \Sigma(0, 2)$$

| A | BC | | |
|---|----|----|----|
| | 00 | 01 | 11 |
| 0 | 1 | 0 | 0 |
| 1 | 0 | 0 | 0 |

$$F(A, B, C) = \Sigma(4, 6)$$

| A | BC | | |
|---|----|----|----|
| | 00 | 01 | 11 |
| 0 | 0 | 0 | 0 |
| 1 | 1 | 0 | 0 |

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

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

Example: Simplify the Boolean functions

$$F(A, B, C) = \sum(0, 1, 2, 3)$$

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

$$F(A, B, C) = \sum(0, 1, 4, 5)$$

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

$$F(A, B, C) = \sum(0, 2, 4, 6)$$

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

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

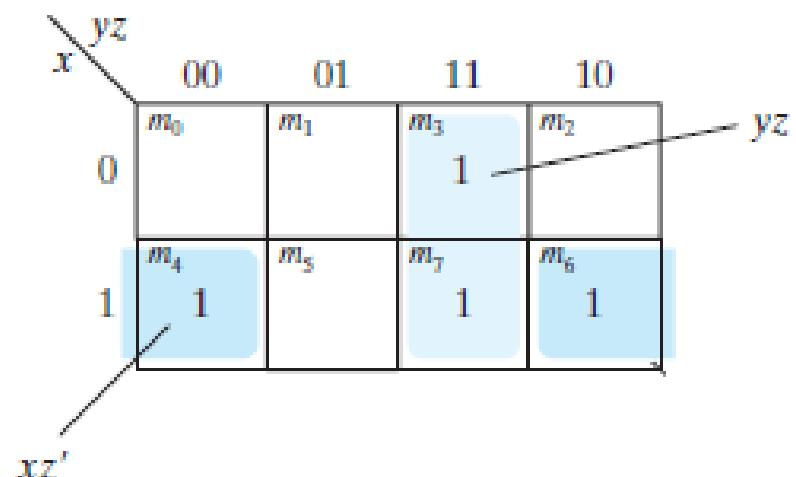
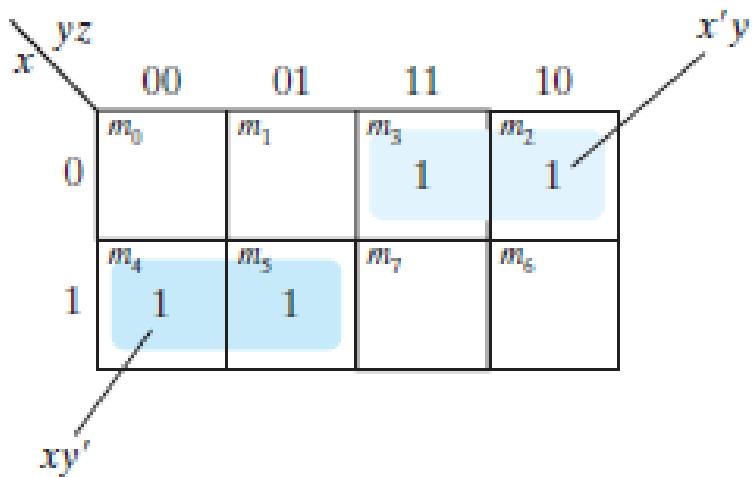
$$F(A, B, C) = \overline{B}$$

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

Example: Simplify the Boolean functions

$$F(x, y, z) = \sum(2, 3, 4, 5)$$

$$F(x, y, z) = \sum(3, 4, 6, 7)$$

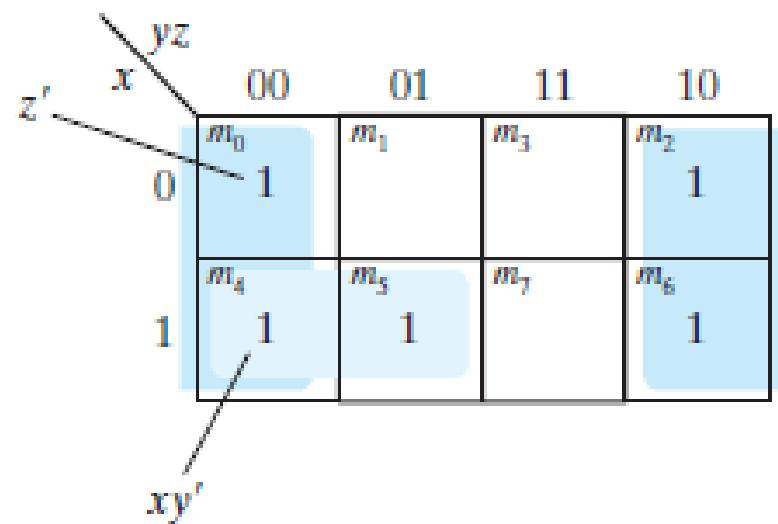
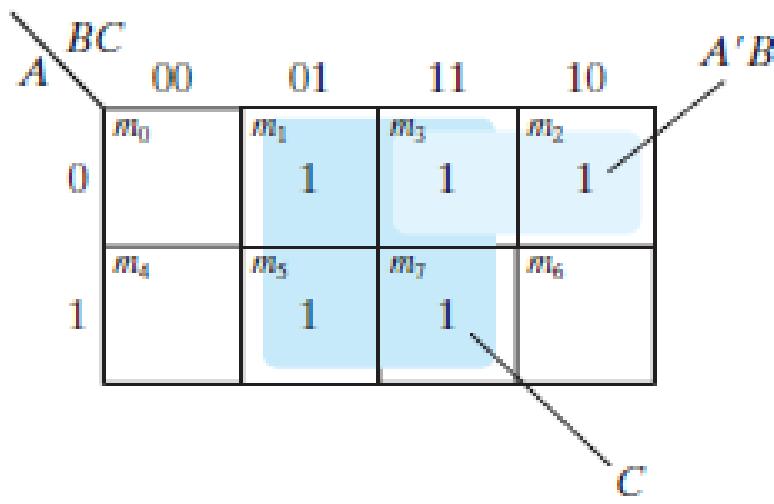


$$F(x, y, z) = x'y + xy'$$

$$F(x, y, z) = xz' + yz$$

Example: Simplify the Boolean functions

$$F(A, B, C) = \sum(1, 2, 3, 5, 7) \quad F(x, y, z) = \sum(0, 2, 4, 5, 6)$$



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

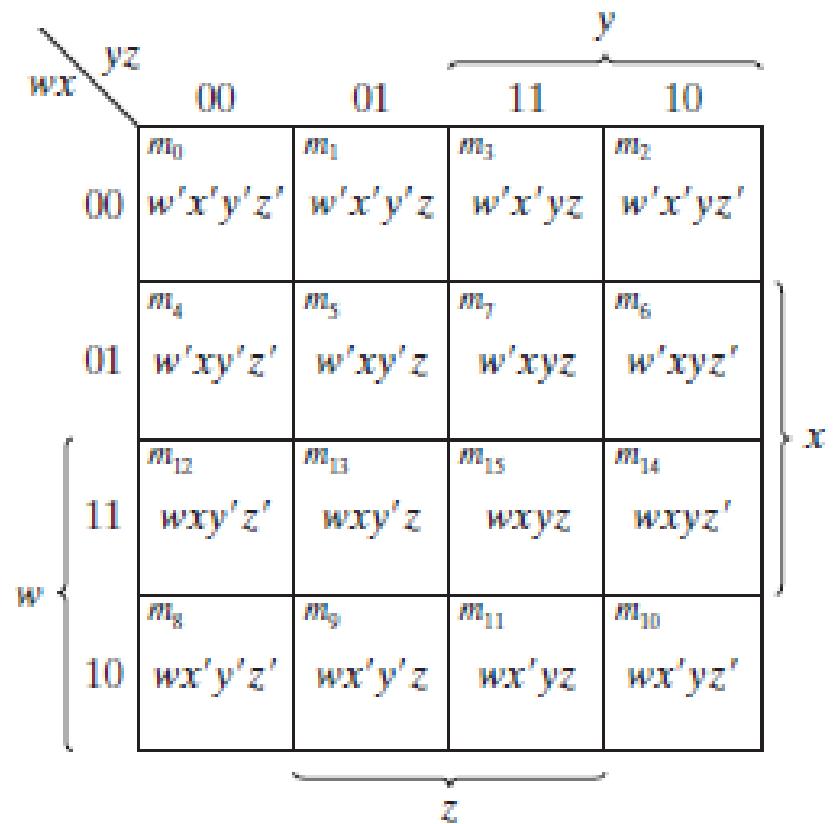
$$F(x, y, z) = z' + xy'$$

Four-Variable K-Map

- The map for Boolean functions of four binary variables (w, x, y, z) is shown in Fig.
- In Fig. (a) are listed the 16 minterms and the squares assigned to each.
- In Fig. (b), the map is redrawn to show the relationship between the squares and the four variables.
- The rows and columns are numbered in a **Gray code sequence**, with only one digit changing value between two adjacent rows or columns.
- The minterm corresponding to each square can be obtained from the concatenation of the row number with the column number.

| | | | |
|----------|----------|----------|----------|
| m_0 | m_1 | m_3 | m_2 |
| m_4 | m_5 | m_7 | m_6 |
| m_{12} | m_{13} | m_{15} | m_{14} |
| m_8 | m_9 | m_{11} | m_{10} |

(a)



(b)

The combination of adjacent squares that is useful during the simplification process is easily determined from inspection of the four-variable map:

- One square represents one minterm, giving a term with four literals.
- Two adjacent squares represent a term with three literals.
- Four adjacent squares represent a term with two literals.
- Eight adjacent squares represent a term with one literal.
- Sixteen adjacent squares produce a function that is always equal to 1.

Example: Simplify the Boolean functions

$$F(A, B, C, D) = \Sigma(13, 15)$$

| | | CD | 00 | 01 | 11 | 10 |
|----|----|----|----|----|----|----|
| | | AB | 00 | 01 | 11 | 10 |
| AB | 00 | 0 | 0 | 0 | 0 | 0 |
| | 01 | 0 | 0 | 0 | 0 | 0 |
| | 11 | 0 | 1 | 1 | 0 | 0 |
| | 10 | 0 | 0 | 0 | 0 | 0 |

$$F(A, B, C, D) = ABD$$

$$F(A, B, C, D) = \Sigma(5, 13)$$

| | | CD | 00 | 01 | 11 | 10 |
|----|----|----|----|----|----|----|
| | | AB | 00 | 01 | 11 | 10 |
| AB | 00 | 0 | 0 | 0 | 0 | 0 |
| | 01 | 0 | 0 | 1 | 0 | 0 |
| | 11 | 0 | 1 | 0 | 0 | 0 |
| | 10 | 0 | 0 | 0 | 0 | 0 |

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

Example: Simplify the Boolean functions

$$F(A, B, C, D) = \Sigma(4, 6)$$

| | | CD | 00 | 01 | 11 | 10 |
|----|----|----|----|----|----|----|
| | | AB | 00 | 01 | 11 | 10 |
| AB | CD | 00 | 0 | 0 | 0 | 0 |
| | | 01 | 1 | 0 | 0 | 1 |
| AB | CD | 11 | 0 | 0 | 0 | 0 |
| | | 10 | 0 | 0 | 0 | 0 |

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

$$F(A, B, C, D) = \Sigma(0, 8)$$

| | | CD | 00 | 01 | 11 | 10 |
|----|----|----|----|----|----|----|
| | | AB | 00 | 01 | 11 | 10 |
| AB | CD | 00 | 1 | 0 | 0 | 0 |
| | | 01 | 0 | 0 | 0 | 0 |
| AB | CD | 11 | 0 | 0 | 0 | 0 |
| | | 10 | 1 | 0 | 0 | 0 |

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

Example: Simplify the Boolean functions

$$F(A, B, C, D) = \sum(4, 5, 6, 7)$$

| | | CD | 00 | 01 | 11 | 10 |
|----|----|----|----|----|----|----|
| | | AB | 00 | 01 | 11 | 10 |
| 00 | 00 | 0 | 0 | 0 | 0 | 0 |
| 01 | 01 | 1 | 1 | 1 | 1 | 1 |
| 11 | 11 | 0 | 0 | 0 | 0 | 0 |
| 10 | 10 | 0 | 0 | 0 | 0 | 0 |

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

$$F(A, B, C, D) = \sum(3, 7, 11, 15)$$

| | | CD | 00 | 01 | 11 | 10 |
|----|----|----|----|----|----|----|
| | | AB | 00 | 01 | 11 | 10 |
| 00 | 00 | 0 | 0 | 1 | 0 | 0 |
| 01 | 01 | 0 | 0 | 1 | 0 | 0 |
| 11 | 11 | 0 | 0 | 1 | 0 | 0 |
| 10 | 10 | 0 | 0 | 1 | 0 | 0 |

$$F(A, B, C, D) = CD$$

Example: Simplify the Boolean functions

$$F(A, B, C, D) = \sum(2, 3, 6, 7)$$

| | | CD | |
|----|----|----|----|
| | | 00 | 01 |
| AB | 00 | 0 | 0 |
| | 01 | 0 | 1 |
| 11 | 0 | 0 | 0 |
| 10 | 0 | 0 | 0 |

$$F(A, B, C, D) = \sum(4, 6, 12, 14)$$

| | | CD | |
|----|----|----|----|
| | | 00 | 01 |
| AB | 00 | 0 | 0 |
| | 01 | 1 | 0 |
| 11 | 1 | 0 | 1 |
| 10 | 0 | 0 | 0 |

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

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

Example: Simplify the Boolean functions

$$F(A, B, C, D) = \sum(2, 3, 10, 11) \quad F(A, B, C, D) = \sum(0, 2, 8, 10)$$

| | | CD | |
|----|----|----|----|
| | | 00 | 01 |
| AB | 00 | 0 | 0 |
| | 01 | 0 | 0 |
| 11 | 0 | 0 | 0 |
| 10 | 0 | 0 | 1 |

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

| | | CD | |
|----|----|----|----|
| | | 00 | 01 |
| AB | 00 | 1 | 0 |
| | 01 | 0 | 0 |
| 11 | 0 | 0 | 0 |
| 10 | 1 | 0 | 1 |

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

Example: Simplify the Boolean functions

$$F(A, B, C, D) = \sum(4, 5, 6, 7, 12, 13, 14, 15) \quad F(A, B, C, D) = \sum(0, 1, 2, 3, 8, 9, 10, 11)$$

| | | CD | 00 | 01 | 11 | 10 |
|----|----|----|----|----|----|----|
| | | AB | 00 | 01 | 11 | 10 |
| AB | 00 | 0 | 0 | 0 | 0 | 0 |
| | 01 | 1 | 1 | 1 | 1 | 1 |
| | 11 | 1 | 1 | 1 | 1 | 1 |
| | 10 | 0 | 0 | 0 | 0 | 0 |

$$F(A, B, C, D) = B$$

| | | CD | 00 | 01 | 11 | 10 |
|----|----|----|----|----|----|----|
| | | AB | 00 | 01 | 11 | 10 |
| AB | 00 | 1 | 1 | 1 | 1 | 1 |
| | 01 | 0 | 0 | 0 | 0 | 0 |
| | 11 | 0 | 0 | 0 | 0 | 0 |
| | 10 | 1 | 1 | 1 | 1 | 1 |

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

Example: Simplify the Boolean functions

$$F(A, B, C, D) = \Sigma(1, 3, 5, 7, 9, 11, 13, 15)$$

$$F(A, B, C, D) = \Sigma(0, 2, 4, 6, 8, 10, 12, 14)$$

| | | CD | 00 | 01 | 11 | 10 |
|----|----|----|----|----|----|----|
| | | AB | 00 | 01 | 11 | 10 |
| AB | 00 | 0 | 1 | 1 | 0 | |
| | 01 | 0 | 1 | 1 | 0 | |
| | 11 | 0 | 1 | 1 | 0 | |
| | 10 | 0 | 1 | 1 | 0 | |

$$F(A, B, C, D) = D$$

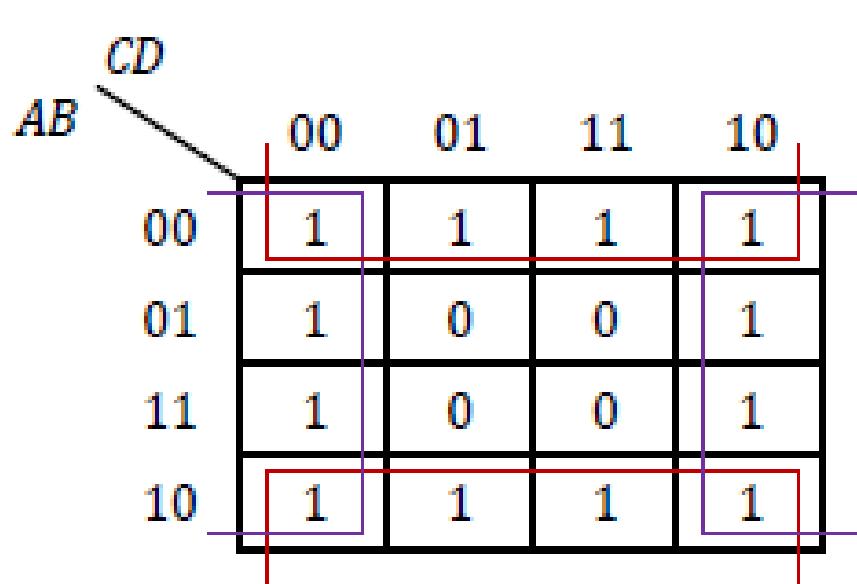
| | | CD | 00 | 01 | 11 | 10 |
|----|----|----|----|----|----|----|
| | | AB | 00 | 01 | 11 | 10 |
| AB | 00 | 1 | 0 | 0 | 1 | |
| | 01 | 1 | 0 | 0 | 1 | |
| | 11 | 1 | 0 | 0 | 1 | |
| | 10 | 1 | 0 | 0 | 1 | |

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

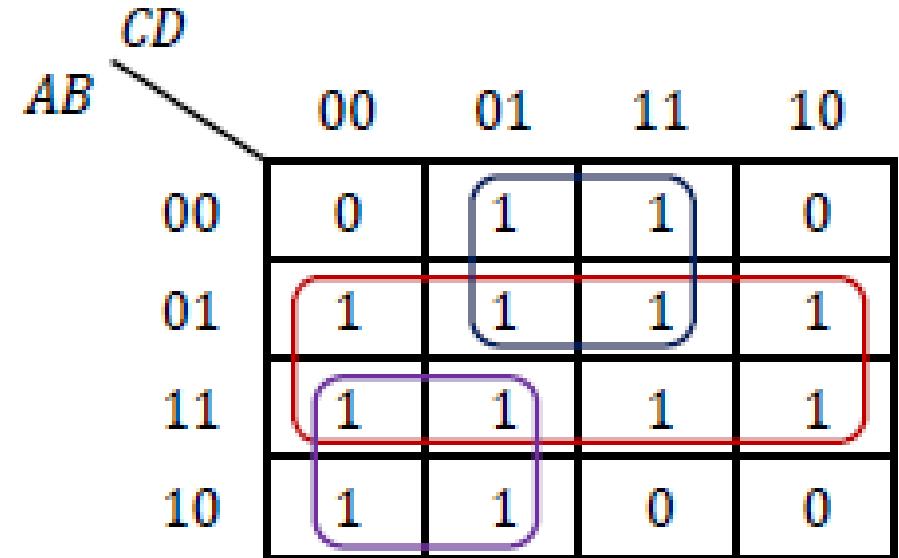
Example: Simplify the following Boolean functions, using four-variable maps:

$$F(A, B, C, D) = \sum (0, 1, 2, 3, 4, 6, 8, 9, 10, 11, 12, 14)$$

$$F(A, B, C, D) = \sum (1, 3, 4, 5, 6, 7, 8, 9, 12, 13, 14, 15)$$



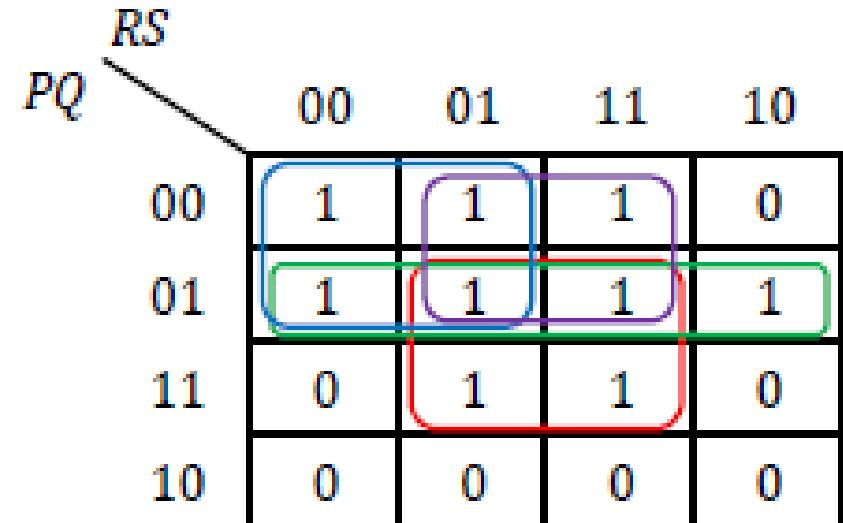
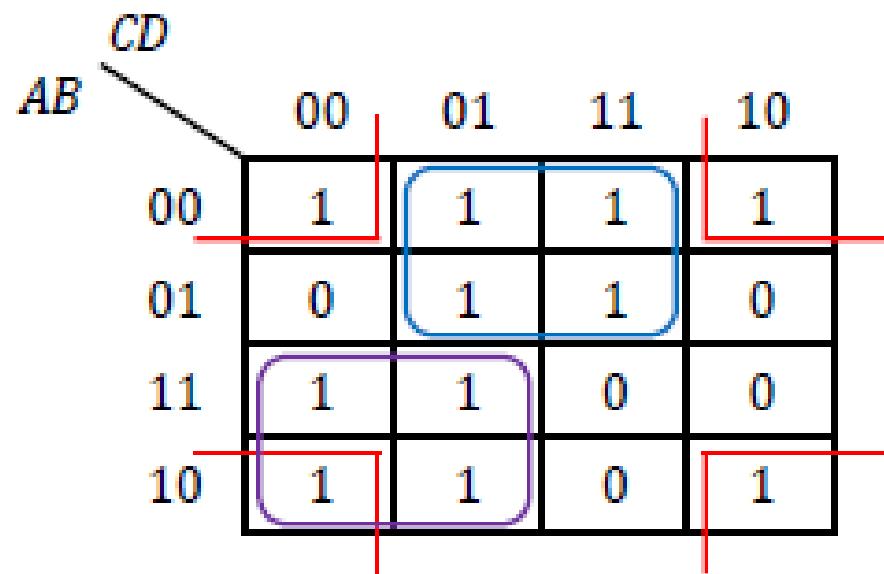
$$F(A, B, C, D) = B' + D'$$



$$F(A, B, C, D) = B + A'D + AC'$$

Example: Simplify the following Boolean functions, using four-variable maps:

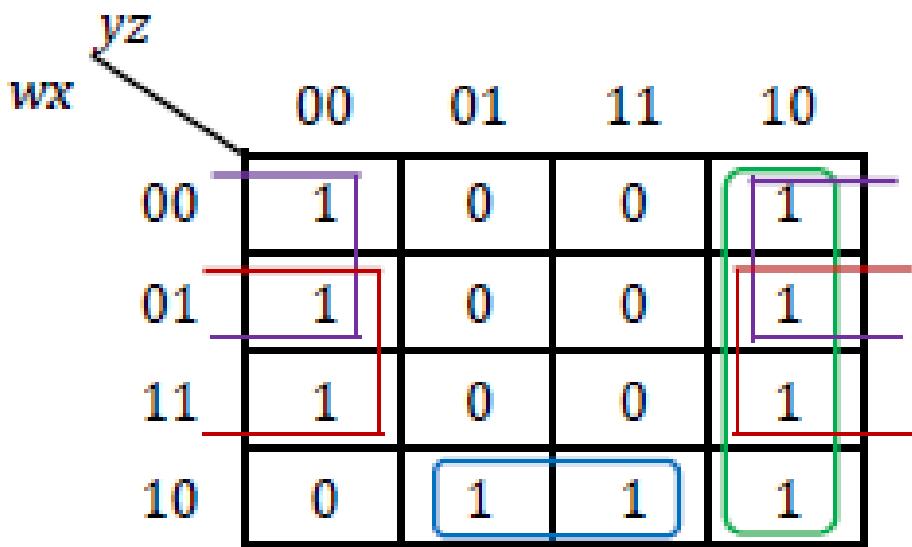
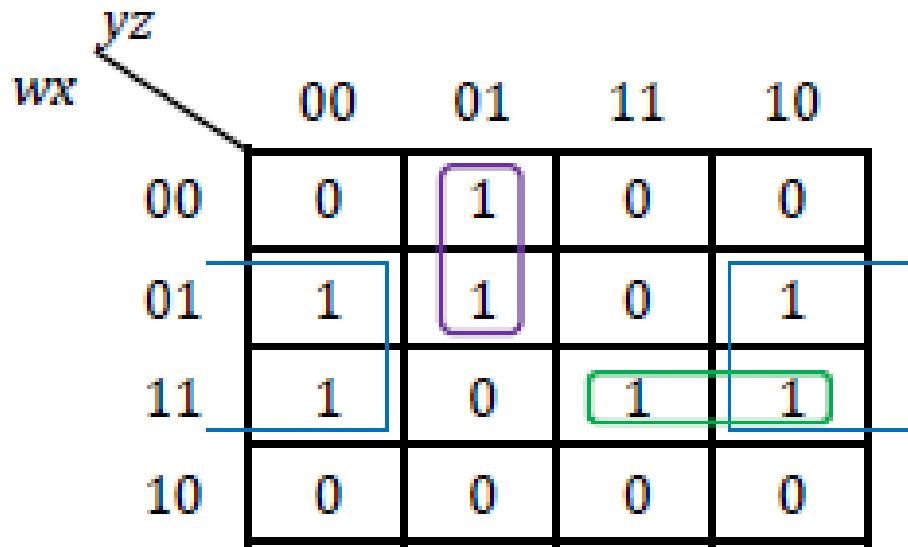
$$F(A, B, C, D) = \sum (0, 1, 2, 3, 5, 7, 8, 9, 10, 12, 13) \quad F(P, Q, R, S) = \sum (0, 1, 3, 4, 5, 6, 7, 13, 15)$$



$$F(A, B, C, D) = B'D' + A'D + AC' \quad F(P, Q, R, S) = P'R' + P'S + P'Q + QS$$

Example: Simplify the following Boolean functions, using four-variable maps:

$$F(w, x, y, z) = \sum (1, 4, 5, 6, 12, 14, 15) \quad F(w, x, y, z) = \sum (0, 2, 4, 6, 9, 10, 11, 12, 14)$$

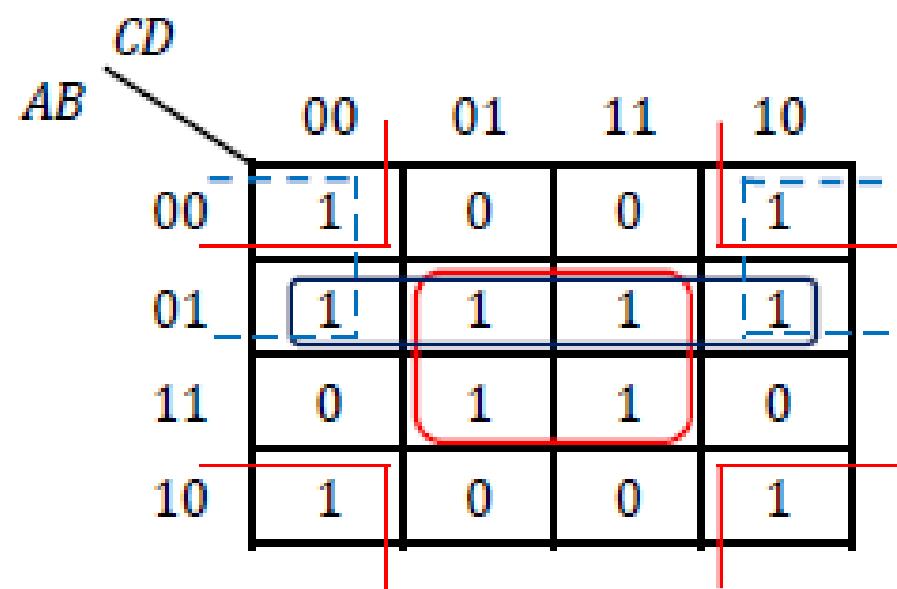


$$F(w, x, y, z) = xz' + w'y'z + wxy$$

$$F(w, x, y, z) = w'z' + xz' + yz' + wx'z$$

Example: Simplify the following Boolean functions, using four-variable maps:

$$F(A, B, C, D) = \sum (3, 4, 5, 7, 9, 13, 14, 15) \quad F(A, B, C, D) = \sum (0, 2, 4, 5, 6, 7, 8, 10, 13, 15)$$



$$F(A, B, C, D) = BD + A'CD + AC'D + A'BC' + ABC \quad F(A, B, C, D) = B'D' + BD + A'B$$

Here ***BD*** is a Redundant group.

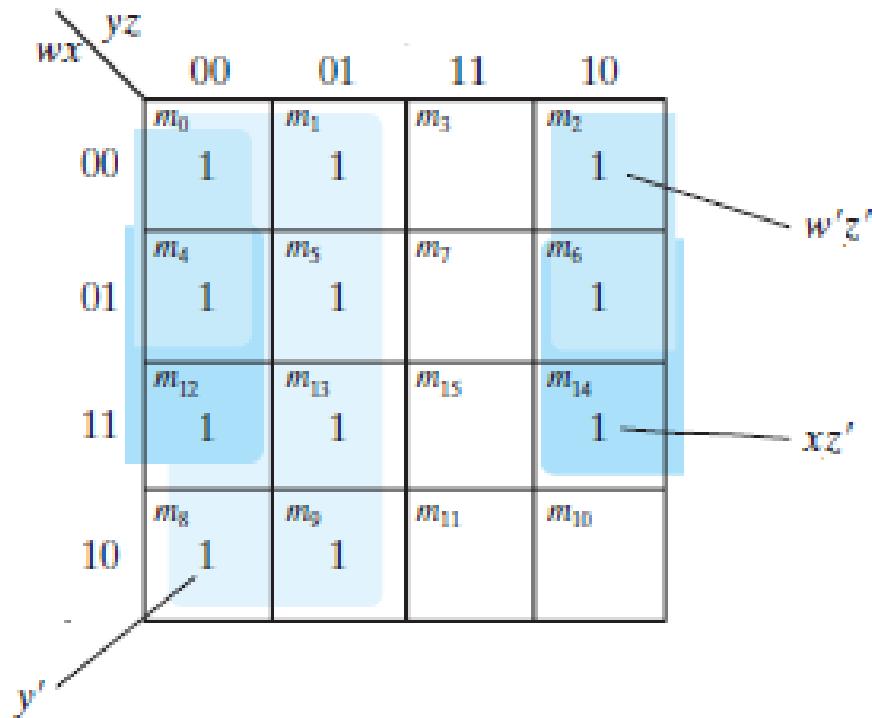
or

$$F(A, B, C, D) = A'CD + AC'D + A'BC' + ABC$$

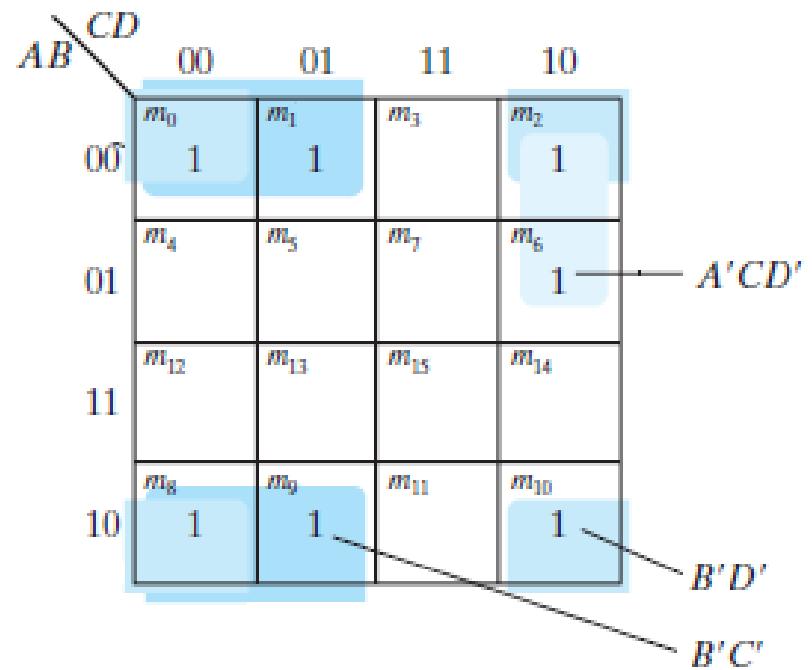
$$F(A, B, C, D) = B'D' + BD + A'D'$$

Example: Simplify the Boolean functions

$$F(w, x, y, z) = \sum (0, 1, 2, 4, 5, 6, 8, 9, 12, 13, 14)$$



$$F(A, B, C, D) = \sum (0, 1, 2, 6, 8, 9, 10)$$



$$F(w, x, y, z) = y' + w'z' + xz'$$

$$F(A, B, C, D) = B'D' + B'C' + A'CD'$$

Prime Implicants

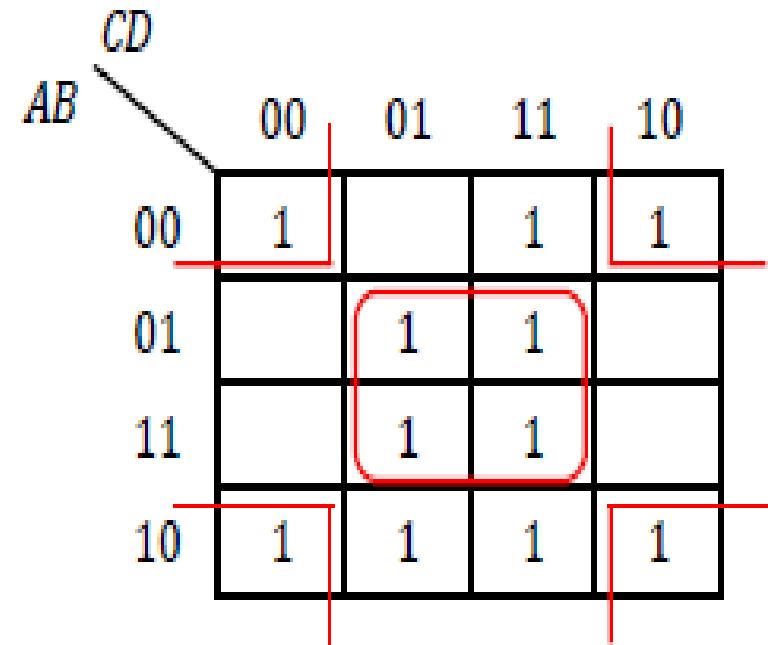
- In choosing adjacent squares in a map, we must ensure that
 1. all the minterms of the function are covered when we combine the squares,
 2. the number of terms in the expression is minimized, and
 3. there are no redundant terms (i.e., minterms already covered by other terms).
- Sometimes there may be two or more expressions that satisfy the simplification criteria.

- The procedure for combining squares in the map may be made more systematic if we understand the meaning of two special types of terms.
- A *prime implicant* is a product term obtained by combining the maximum possible number of adjacent squares in the map.
- If a minterm in a square is covered by only one prime implicant, that prime implicant is said to be *essential*.

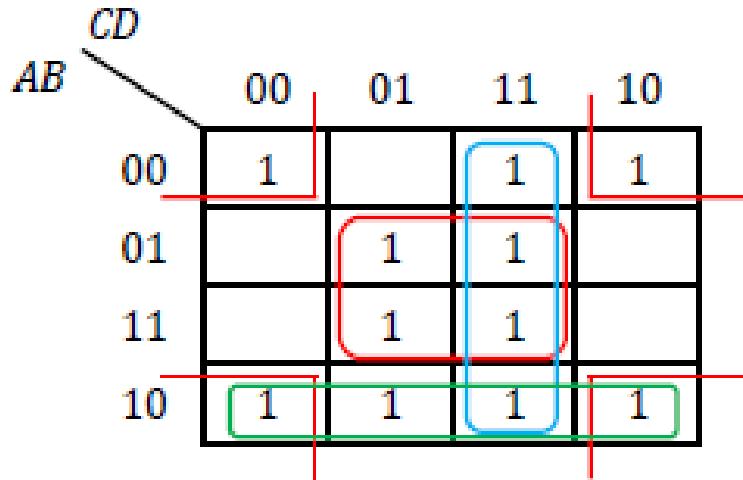
Consider the following four-variable Boolean function:

$$F(A, B, C, D) = \sum (0, 2, 3, 5, 7, 8, 9, 10, 11, 13, 15)$$

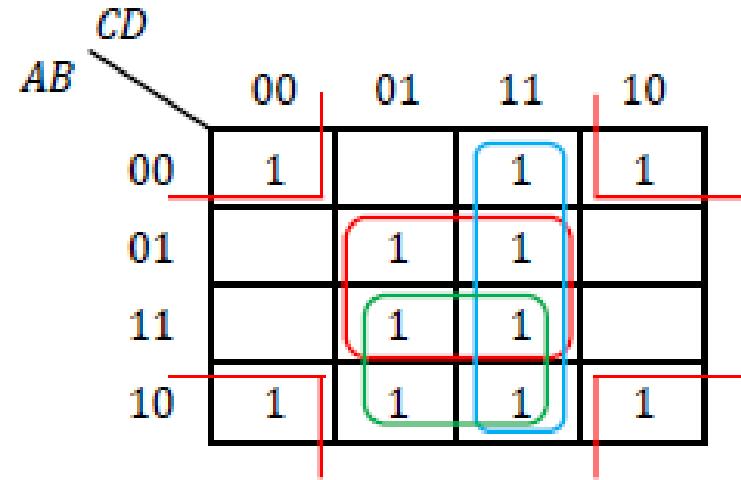
- There is only one way to include minterm m_0 within four adjacent squares. These four squares define the term $B'D'$.
- Similarly, there is only one way that minterm m_5 can be combined with four adjacent squares, and this gives the second term BD .
- The two essential prime implicants cover eight minterms.



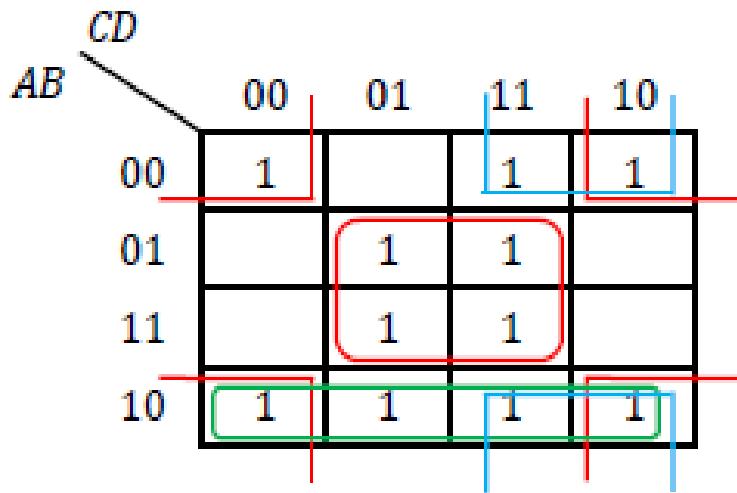
Essential prime implicants
 $B'D'$ and BD



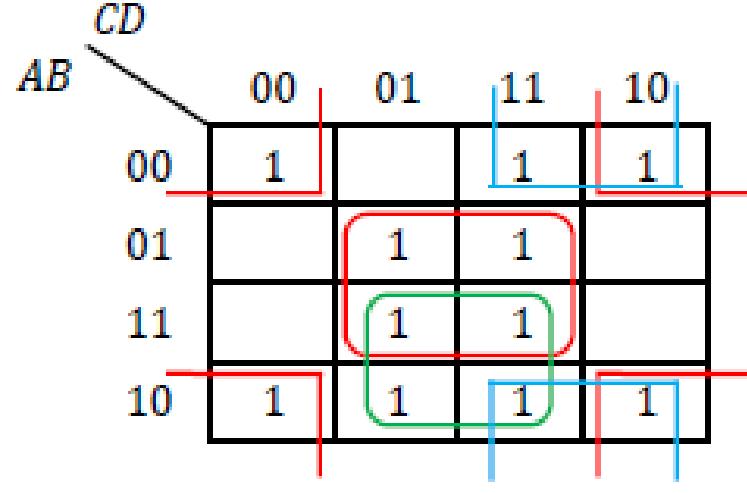
$$F(A, B, C, D) = \textcolor{red}{B'D'} + \textcolor{red}{BD} + \textcolor{blue}{CD} + \textcolor{green}{AB'}$$



$$F(A, B, C, D) = \textcolor{red}{B'D'} + \textcolor{red}{BD} + \textcolor{blue}{CD} + \textcolor{blue}{AD}$$



$$F(A, B, C, D) = \textcolor{red}{B'D'} + \textcolor{red}{BD} + \textcolor{blue}{B'C} + \textcolor{green}{AB'}$$



$$F(A, B, C, D) = \textcolor{red}{B'D'} + \textcolor{red}{BD} + \textcolor{blue}{B'C} + \textcolor{green}{AD}$$

- Figure shows all possible ways that the three minterms (m_3 , m_9 , and m_{11}) can be covered with prime implicants.
- Minterm m_3 can be covered with either prime implicant CD or prime implicant $B'C$.
- Minterm m_9 can be covered with either AB' or AD .
- Minterm m_{11} is covered with any one of the four prime implicants.
- The simplified expression is obtained from the logical sum of the two essential prime implicants and any two prime implicants that cover minterms m_3 , m_9 , and m_{11} .
- There are four possible ways that the function can be expressed with four product terms of two literals each:

$$\begin{aligned}
 F &= \mathbf{\color{red}{B'D'}} + \mathbf{\color{blue}{BD}} + \mathbf{\color{blue}{CD}} + \mathbf{\color{blue}{AB'}} \\
 &= \mathbf{\color{red}{B'D'}} + \mathbf{\color{red}{BD}} + \mathbf{\color{blue}{CD}} + \mathbf{\color{blue}{AD}} \\
 &= \mathbf{\color{red}{B'D'}} + \mathbf{\color{red}{BD}} + \mathbf{\color{red}{B'C}} + \mathbf{\color{blue}{AB'}} \\
 &= \mathbf{\color{red}{B'D'}} + \mathbf{\color{red}{BD}} + \mathbf{\color{red}{B'C}} + \mathbf{\color{blue}{AD}}
 \end{aligned}$$

Find all the prime implicants for the following Boolean functions, and determine which are essential:

$$F(A, B, C, D) = \sum (0, 4, 5, 10, 11, 13, 15)$$

| | <i>CD</i> | 00 | 01 | 11 | 10 |
|-----------|-----------|----|----|----|----|
| <i>AB</i> | 00 | 1 | 0 | 0 | 0 |
| | 01 | 1 | 1 | 0 | 0 |
| | 11 | 0 | 1 | 1 | 0 |
| | 10 | 0 | 0 | 1 | 1 |

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

| | <i>CD</i> | 00 | 01 | 11 | 10 |
|-----------|-----------|----|----|----|----|
| <i>AB</i> | 00 | 1 | 0 | 0 | 0 |
| | 01 | 1 | 1 | 0 | 0 |
| | 11 | 0 | 1 | 1 | 0 |
| | 10 | 0 | 0 | 1 | 1 |

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

| | <i>CD</i> | 00 | 01 | 11 | 10 |
|-----------|-----------|----|----|----|----|
| <i>AB</i> | 00 | 1 | 0 | 0 | 0 |
| | 01 | 1 | 1 | 0 | 0 |
| | 11 | 0 | 1 | 1 | 0 |
| | 10 | 0 | 0 | 1 | 1 |

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

- Essential prime implicants $A'C'D'$ and $AB'C$
- Prime implicants $BC'D$, ABD , ACD and $A'BC'$

DON'T-CARE CONDITIONS

- In some applications the function is not specified for certain combinations of the variables.
- As an example, the four-bit binary code for the decimal digits has six combinations that are not used and consequently are considered to be unspecified.

- Functions that have unspecified outputs for some input combinations are called *incompletely specified functions*.
- For this reason, it is customary to call the unspecified minterms of a function *don't-care conditions*.
- These don't-care conditions can be used on a map to provide further simplification of the Boolean expression.

- A don't-care minterm is a combination of variables whose logical value is not specified.
- To distinguish the don't-care condition from 1's and 0's, an X is used.
- Thus, an X inside a square in the map indicates that we don't care whether the value of 0 or 1 is assigned to F for the particular minterm.

- In choosing adjacent squares to simplify the function in a map, the don't-care minterms may be assumed to be either 0 or 1.
- When simplifying the function, we can choose to include each don't-care minterm with either the 1's or the 0's, depending on which combination gives the simplest expression.

Example: Simplify the Boolean function

$$F(w, x, y, z) = \sum(1, 3, 7, 11, 15)$$

which has the don't-care conditions

$$d(w, x, y, z) = \sum(0, 2, 5)$$

| | wx | yz | | |
|--------|----------|----------|----------|----------|
| | 00 | 01 | 11 | 10 |
| $w'x'$ | X | 1 | 1 | X |
| 00 | m_4 | m_5 | m_7 | m_6 |
| 01 | 0 | X | 1 | 0 |
| 11 | m_{12} | m_{13} | m_{15} | m_{14} |
| 10 | 0 | 0 | 1 | 0 |
| | m_8 | m_9 | m_{11} | m_{10} |
| | 0 | 0 | 1 | 0 |



$$F = yz + w'x'$$

| | wx | yz | | |
|-------|----------|----------|----------|----------|
| | 00 | 01 | 11 | 10 |
| $w'z$ | X | 1 | 1 | X |
| 00 | m_4 | m_5 | m_7 | m_6 |
| 01 | 0 | X | 1 | 0 |
| 11 | m_{12} | m_{13} | m_{15} | m_{14} |
| 10 | 0 | 0 | 1 | 0 |
| | m_8 | m_9 | m_{11} | m_{10} |
| | 0 | 0 | 1 | 0 |



$$F = yz + w'z$$

Example: Simplify the following Boolean function F , together with the don't-care conditions d .

$$F(A, B, C, D) = \sum (1, 5, 6, 12, 13, 14)$$

$$d(A, B, C, D) = \sum (2, 4)$$

$$F(A, B, C, D) = \sum (4, 5, 7, 12, 14, 15)$$

$$d(A, B, C, D) = \sum (3, 8, 10)$$

| | <i>CD</i> | | | | |
|----|-----------|----|----|----|---|
| | 00 | 01 | 11 | 10 | |
| AB | 00 | 0 | 1 | 0 | X |
| 01 | X | 1 | 0 | 1 | |
| 11 | 1 | 1 | 0 | 1 | |
| 10 | 0 | 0 | 0 | 0 | |

| | <i>CD</i> | | | |
|----|-----------|----|----|----|
| | 00 | 01 | 11 | 10 |
| AB | 00 | 0 | X | 0 |
| 01 | 1 | 1 | 1 | 0 |
| 11 | 1 | 0 | 1 | 1 |
| 10 | X | 0 | 0 | X |

$$F(A, B, C, D) = BC' + BD' + A'C'D \quad F(A, B, C, D) = AD' + A'BC' + BCD$$

Example: Simplify the following Boolean function F , together with the don't-care conditions d .

$$F(A, B, C, D) = \sum (1, 3, 5, 7, 9, 15)$$

$$d(A, B, C, D) = \sum (4, 6, 12, 13)$$

$$F(A, B, C, D) = \sum (1, 3, 4, 5, 8, 10, 11, 15)$$

$$d(A, B, C, D) = \sum (0, 2, 7, 14)$$

| | | CD | | | |
|----|----|----|----|----|----|
| | AB | 00 | 01 | 11 | 10 |
| AB | 00 | 0 | 1 | 1 | 0 |
| 00 | X | 1 | 1 | X | |
| 01 | X | X | 1 | 0 | |
| 10 | 0 | 1 | 0 | 0 | |

| | | CD | | | |
|----|----|----|----|----|----|
| | AB | 00 | 01 | 11 | 10 |
| AB | 00 | X | 1 | 1 | X |
| 00 | 1 | 1 | X | 0 | |
| 01 | 0 | 0 | 1 | X | |
| 11 | 1 | 0 | 1 | 1 | |
| 10 | | | | | 1 |

$$F(A, B, C, D) = A'D + BD + C'D$$

$$F(A, B, C, D) = B'D' + A'C' + CD$$

PRODUCT-OF-SUMS SIMPLIFICATION

Example: Simplify the following Boolean functions in POS form.

$$F(x, y, z) = \prod (0, 1, 3, 7)$$

| | | yz | |
|---|---|----|----|
| | | 00 | 01 |
| x | 0 | 0 | 0 |
| 1 | | | 0 |

$$F'(x, y, z) = x'y' + yz$$

$$F(x, y, z) = (x + y)(y' + z')$$

$$F(A, B, C) = \prod (0, 1, 2, 3, 4, 7)$$

| | | BC | |
|---|---|----|----|
| | | 00 | 01 |
| A | 0 | 0 | 0 |
| 1 | 0 | | 0 |

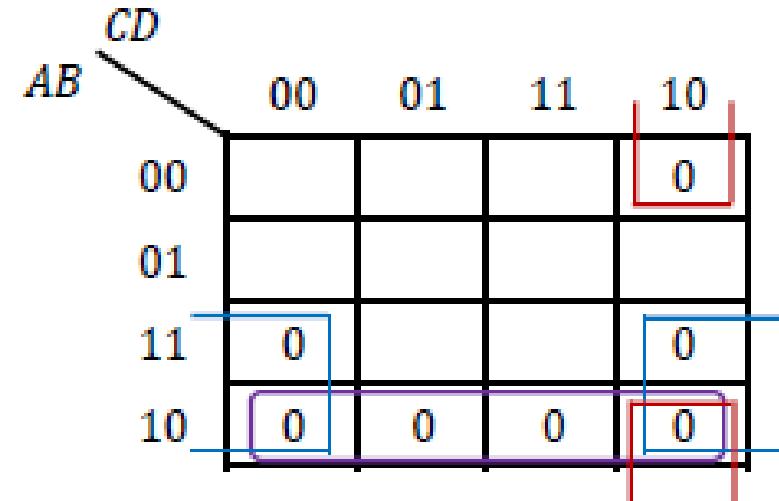
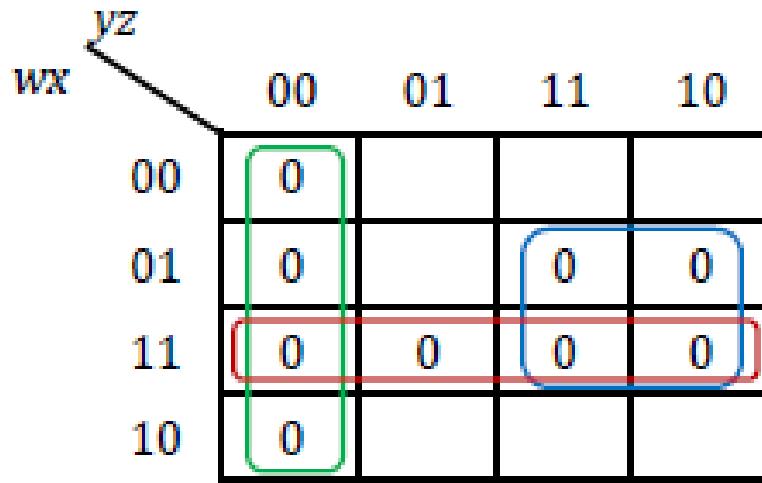
$$F'(A, B, C) = A' + B'C' + BC$$

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

Example: Simplify the following Boolean functions in POS form.

$$F(w, x, y, z) = \prod (0, 4, 6, 7, 8, 12, 13, 14, 15)$$

$$F(A, B, C, D) = \prod (2, 8, 9, 10, 11, 12, 14)$$



$$F'(w, x, y, z) = y'z' + xy + wx$$

$$F(w, x, y, z) = (y + z)(x' + y')(w' + x')$$

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

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

DIGITAL ELECTRONICS

III Semester CSE & CST



Mr. Srikanta Patnaik Mr. Ajit Kumar Patro Mr. Ami Kumar Parida

DEPARTMENT OF ELECTRONICS AND COMMUNICATION ENGINEERING

SCHOOL OF ENGINEERING & TECHNOLOGY

GIET UNIVERSITY, GUNUPUR, ODISHA

COMBINATIONAL LOGIC

Combinational Circuits;

Analysis Procedure;

Design Procedure;

Adders; Subtractors

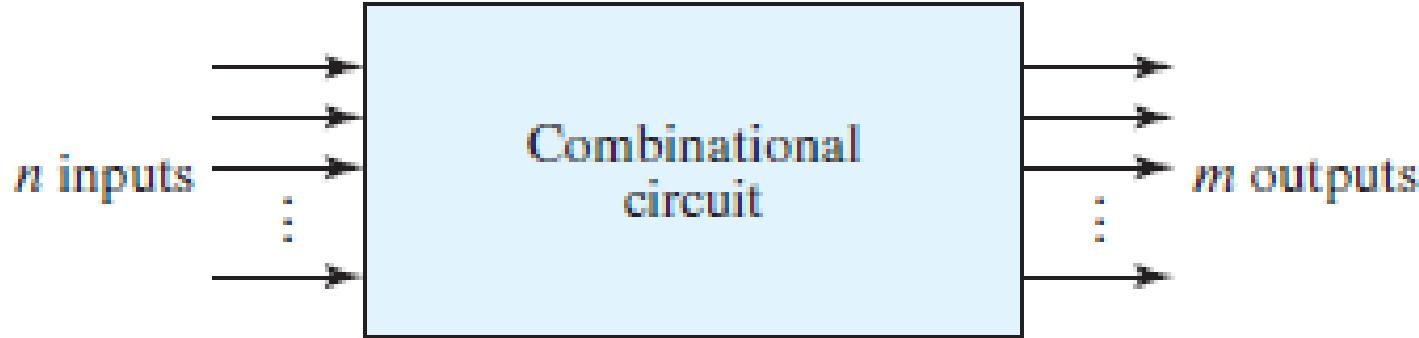
INTRODUCTION

- Logic circuits for digital systems may be **combinational or sequential**.
- A combinational circuit consists of logic gates whose outputs at any time are determined from only the present combination of inputs.
- A combinational circuit performs an operation that can be specified logically by a set of Boolean functions.

- In contrast, sequential circuits employ storage elements in addition to logic gates.
- Their outputs are a function of the inputs and the state of the storage elements.
- Because the state of the storage elements is a function of previous inputs, the outputs of a sequential circuit depend not only on present values of inputs, but also on past inputs, and the circuit behavior must be specified by a time sequence of inputs and internal states.

COMBINATIONAL CIRCUITS

- A combinational circuit consists of an interconnection of logic gates.
- Combinational logic gates react to the values of the signals at their inputs and produce the value of the output signal, transforming binary information from the given input data to a required output data.
- A block diagram of a combinational circuit is shown in Fig.



- The n input binary variables come from an external source; the m output variables are produced by the internal combinational logic circuit and go to an external destination.
- Each input and output variable exists physically as an analog signal whose values are interpreted to be a binary signal that represents logic 1 and logic 0.

- For n input variables, there are 2^n possible combinations of the binary inputs.
- For each possible input combination, there is one possible value for each output variable.
- Thus, a combinational circuit can be specified with a truth table that lists the output values for each combination of input variables.
- A combinational circuit also can be described by m Boolean functions, one for each output variable. Each output function is expressed in terms of the n input variables.

ANALYSIS PROCEDURE

- The analysis of a combinational circuit requires that we determine the function that the circuit implements.
- This task starts with a given logic diagram and culminates with a set of Boolean functions, a truth table, or possibly, an explanation of the circuit operation.

DESIGN PROCEDURE

- The procedure involves the following steps:
1. From the specifications of the circuit, determine the required number of inputs and outputs and assign a symbol to each.
 2. Derive the truth table that defines the required relationship between inputs and outputs.
 3. Obtain the simplified Boolean functions for each output as a function of the input variables.
 4. Draw the logic diagram and verify the correctness of the design (manually or by simulation).

ADDERS

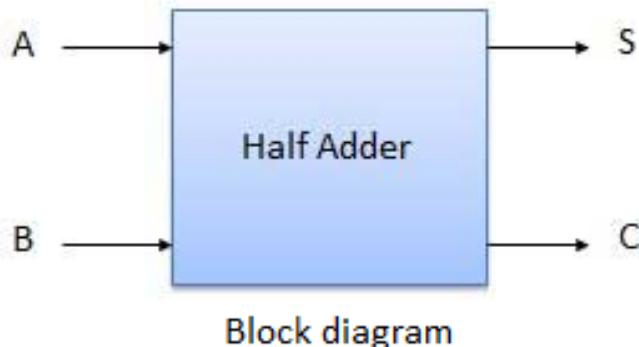
- The most basic arithmetic operation is the addition of two binary digits.
- This simple addition consists of 4 possible operations, namely,

$$\begin{array}{cccc} 0 & 0 & 1 & 1 \\ +0 & +1 & +0 & +1 \\ \hline 0 & 1 & 1 & 1 \leftarrow 0 \end{array}$$

(carry)

Half Adder

- A combinational circuit that performs the addition of two bits is called a *half adder*.
- A half adder is a combinational circuit with two binary inputs (augend and addend bits) and two binary outputs (sum and carry bits).
- It adds the two inputs (**A** and **B**) and produces two outputs, sum (**S**) and carry (**C**).



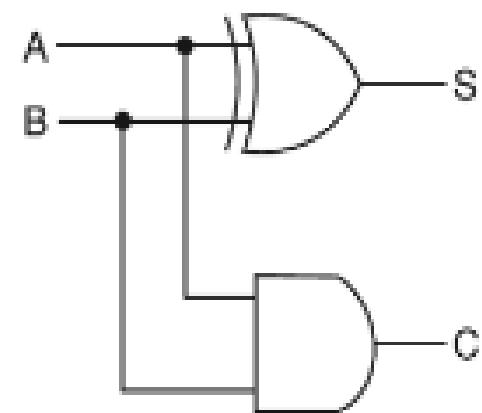
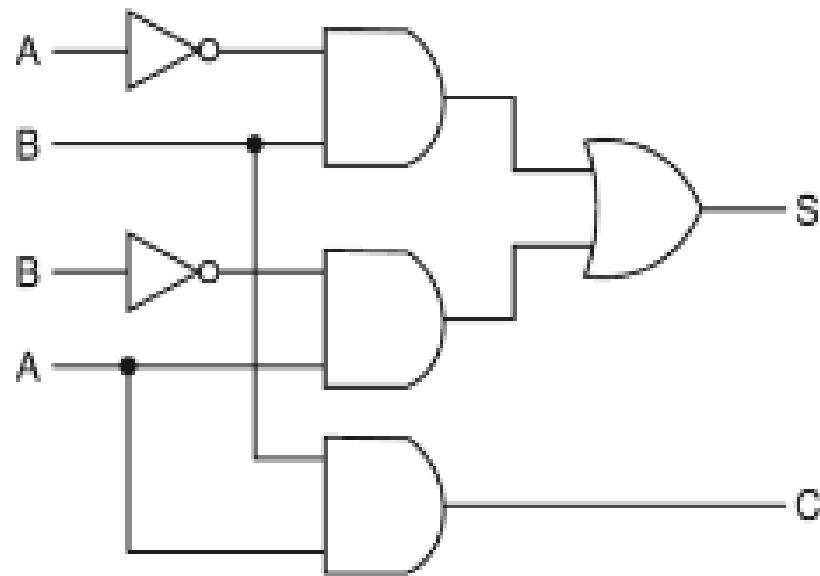
| Inputs | | Outputs | |
|--------|---|---------|---|
| A | B | C | S |
| 0 | 0 | 0 | 0 |
| 0 | 1 | 0 | 1 |
| 1 | 0 | 0 | 1 |
| 1 | 1 | 1 | 0 |

Truth Table

- The simplified Boolean functions for the two outputs can be obtained directly from the truth table. The simplified sum-of-products expressions are

$$S = \bar{A}B + A\bar{B} = A \oplus B$$

$$C = AB$$

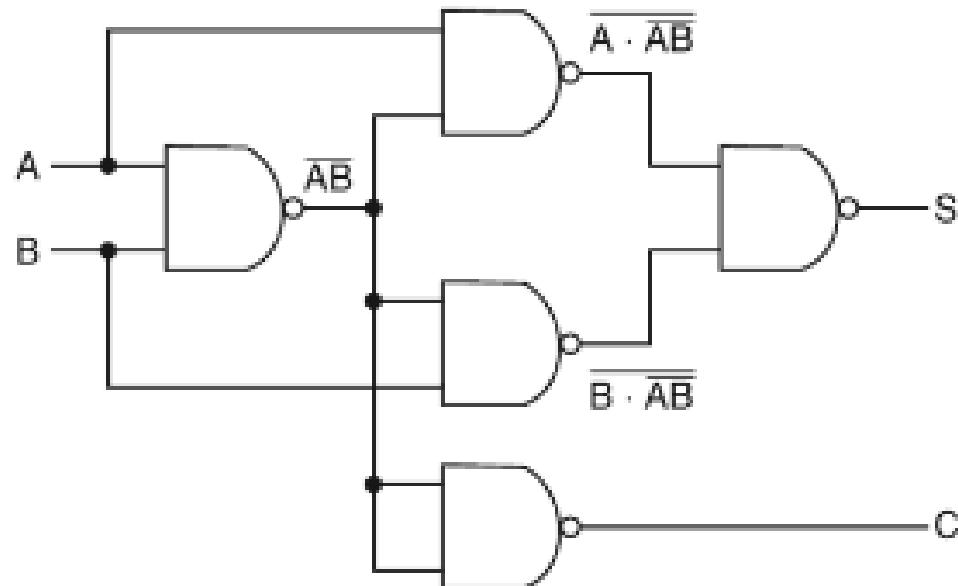


Logic diagrams of half-adder.

Implementation of half adder by using only NAND gates.

$$\begin{aligned}S &= A\bar{B} + \bar{A}B = A\bar{A} + A\bar{B} + \bar{A}B + B\bar{B} \\&= A(\bar{A} + \bar{B}) + B(\bar{A} + \bar{B}) \\&= A \cdot \overline{AB} + B \cdot \overline{AB} \\&= \overline{\overline{A} \cdot \overline{AB} \cdot B \cdot \overline{AB}}\end{aligned}$$

$$C = AB = \overline{\overline{AB}}$$

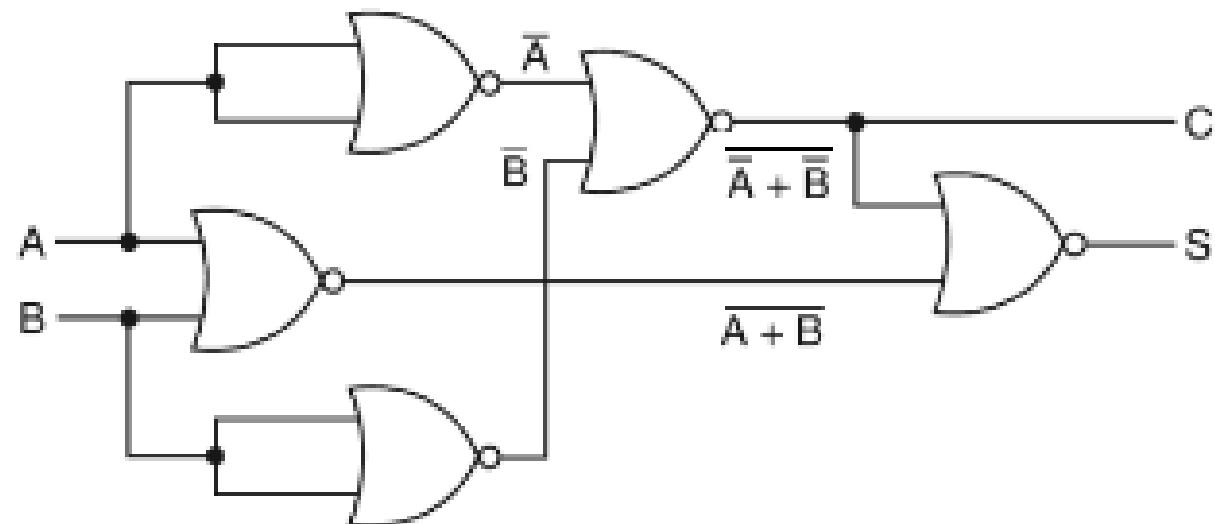


Logic diagram of a half-adder using only 2-input NAND gates.

Implementation of half adder by using only NOR gates.

$$\begin{aligned} S &= A\bar{B} + \bar{A}B = A\bar{A} + A\bar{B} + \bar{A}B + B\bar{B} \\ &= A(\bar{A} + \bar{B}) + B(\bar{A} + \bar{B}) \\ &= (A + B)(\bar{A} + \bar{B}) \\ &= \overline{\overline{A} + \overline{B} + \overline{\bar{A}} + \overline{\bar{B}}} \end{aligned}$$

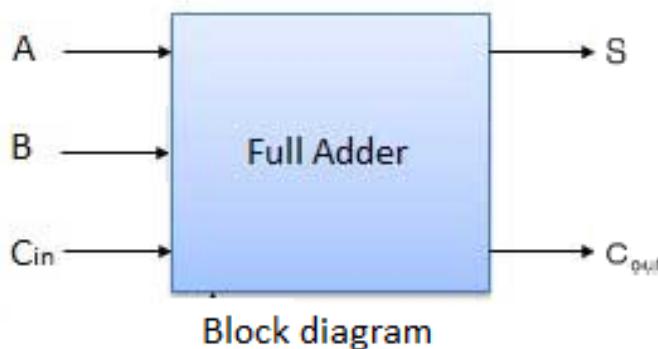
$$C = AB = \overline{\overline{AB}} = \overline{\overline{\bar{A}} + \overline{\bar{B}}}$$



Logic diagram of a half-adder using only 2-input NOR gates.

Full Adder

- A *full adder* is a combinational circuit that forms the arithmetic sum of three bits (two significant bits and a previous carry).
- It consists of three inputs and two outputs.
- The full adder adds the bits A and B and the carry from the previous column called the carry-in C_{in} and outputs the sum bit S and the carry bit called the carry-out C_{out} .



| Inputs | | | Outputs | |
|--------|---|-----------------|------------------|---|
| A | B | C _{in} | C _{out} | S |
| 0 | 0 | 0 | 0 | 0 |
| 0 | 0 | 1 | 0 | 1 |
| 0 | 1 | 0 | 0 | 1 |
| 0 | 1 | 1 | 1 | 0 |
| 1 | 0 | 0 | 0 | 1 |
| 1 | 0 | 1 | 1 | 0 |
| 1 | 1 | 0 | 1 | 0 |
| 1 | 1 | 1 | 1 | 1 |

Truth Table

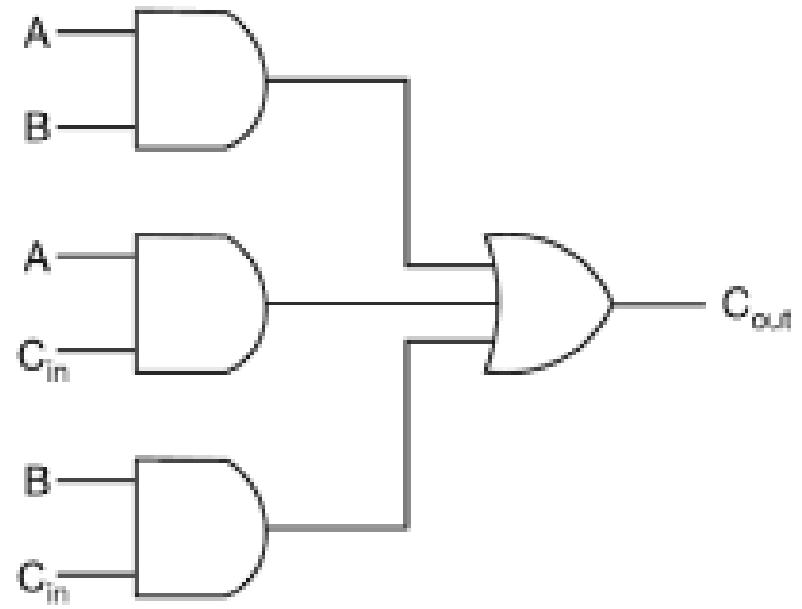
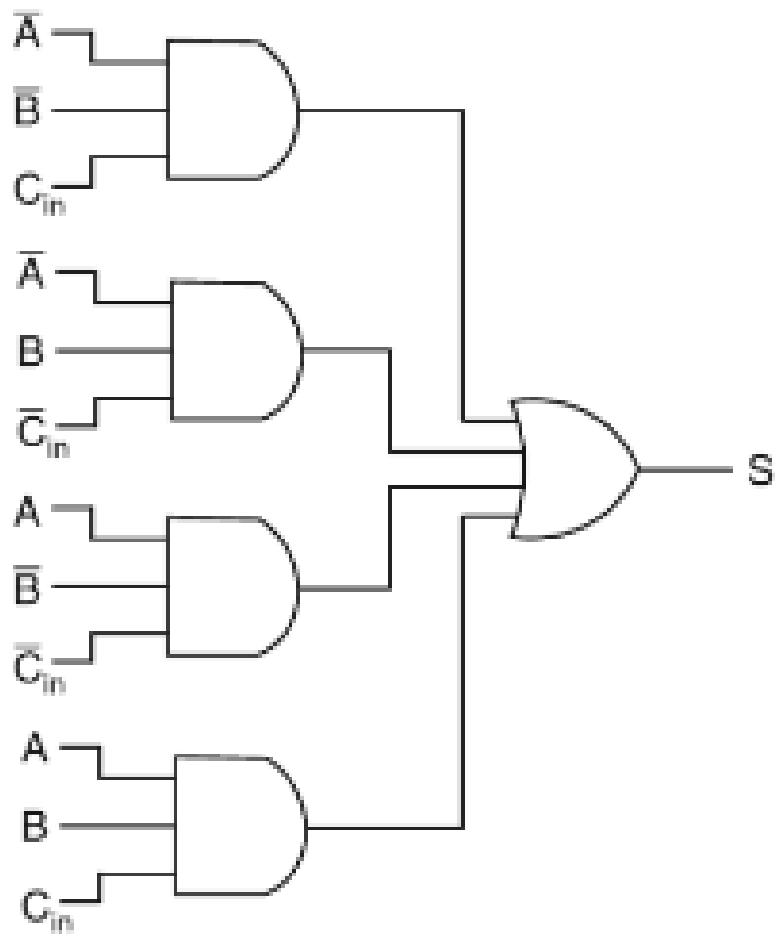
The simplified expressions are

$$S = \sum(1, 2, 4, 7)$$

$$S = \bar{A}\bar{B}C_{in} + \bar{A}B\bar{C}_{in} + A\bar{B}\bar{C}_{in} + ABC_{in}$$

$$C_{out} = \sum(3, 5, 6, 7)$$

$$C_{out} = AB + AC_{in} + BC_{in}$$



Logic diagrams of full-adder.

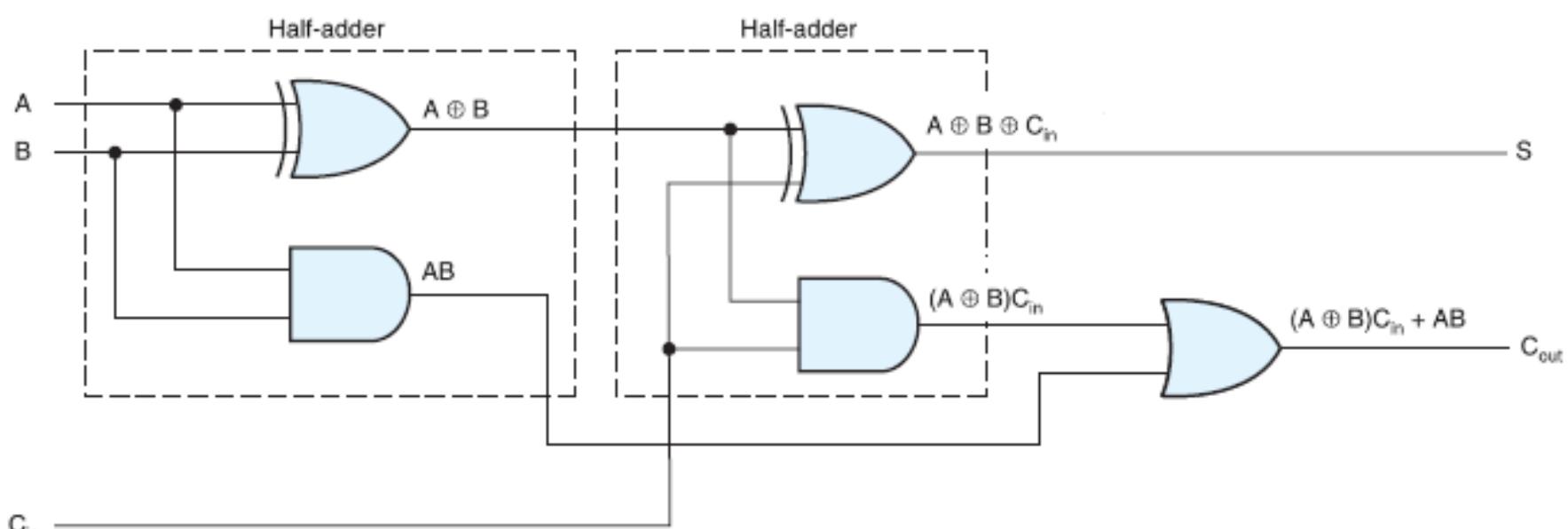
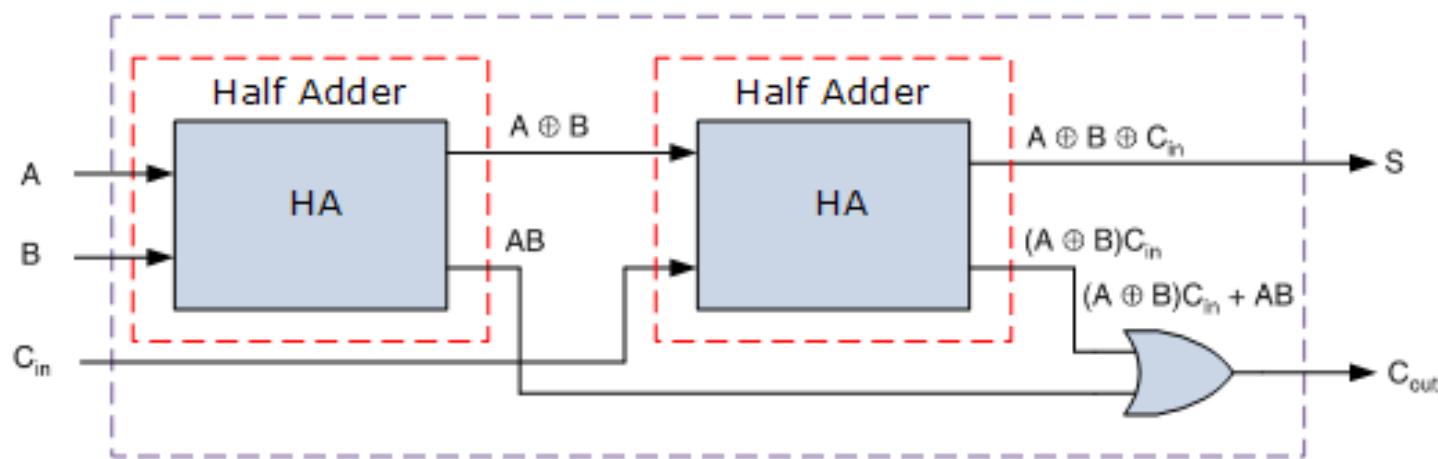
Implementation of full adder with two half adders and an OR gate.

- From the truth table of full adder, the outputs sum and carry are described by

$$\begin{aligned} S &= \bar{A}\bar{B}C_{in} + \bar{A}B\bar{C}_{in} + A\bar{B}\bar{C}_{in} + ABC_{in} \\ &= (\bar{A}\bar{B} + \bar{A}B)\bar{C}_{in} + (AB + \bar{A}\bar{B})C_{in} \\ &= (A \oplus B)\bar{C}_{in} + (\overline{A \oplus B})C_{in} = A \oplus B \oplus C_{in} \end{aligned}$$

and

$$\begin{aligned} C_{out} &= \bar{A}BC_{in} + A\bar{B}C_{in} + AB\bar{C}_{in} + ABC_{in} \\ &= (\bar{A}B + A\bar{B})C_{in} + AB(\bar{C}_{in} + C_{in}) = AB + (A \oplus B)C_{in} \end{aligned}$$

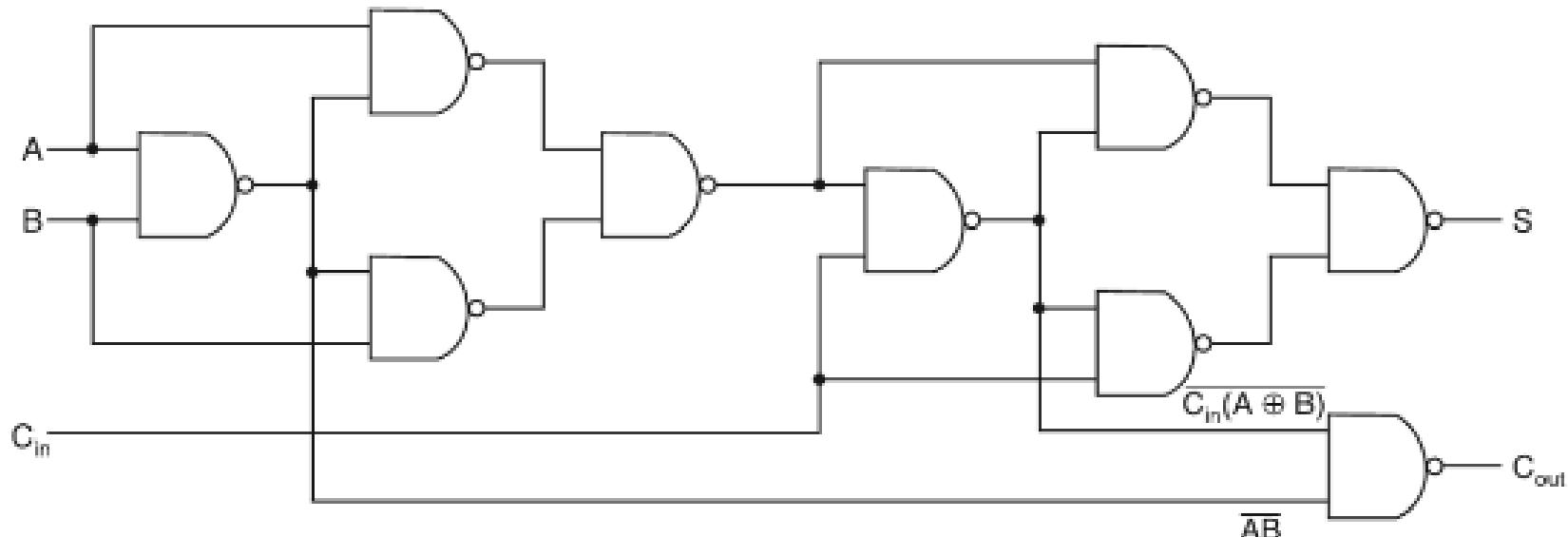


Implementation of full adder by using only NAND gates.

We know that $A \oplus B = \overline{\overline{A} \cdot \overline{AB}} \cdot \overline{\overline{B} \cdot \overline{AB}}$

Then $S = A \oplus B \oplus C_{in} = \overline{(A \oplus B) \cdot \overline{(A \oplus B)C_{in}}} \cdot \overline{C_{in}} \cdot \overline{(A \oplus B)C_{in}}$

$$C_{out} = C_{in}(A \oplus B) + AB = \overline{C_{in}(A \oplus B)} \cdot \overline{AB}$$



Logic diagram of a full-adder using only 2-input NAND gates.

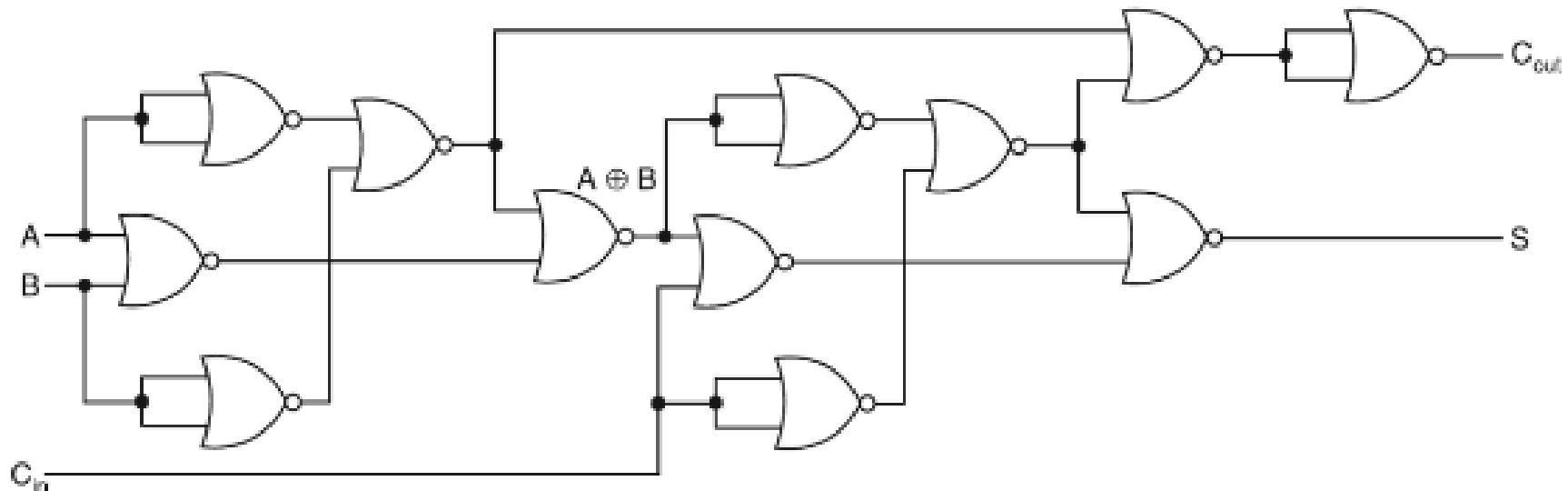
Implementation of full adder by using only NOR gates.

We know that $A \oplus B = \overline{\overline{(A + B)} + \overline{\overline{A} + \overline{B}}}$

Then

$$S = A \oplus B \oplus C_{in} = \overline{\overline{(A \oplus B) + C_{in}}} + \overline{\overline{(A \oplus B)} + \overline{C_{in}}}$$

$$C_{out} = AB + C_{in}(A \oplus B) = \overline{\overline{\overline{A} + \overline{B}}} + \overline{\overline{C_{in}}} + \overline{\overline{A \oplus B}} = \overline{\overline{\overline{A} + \overline{B}}} + \overline{\overline{C_{in}}} + \overline{\overline{A \oplus B}}$$



Logic diagram of a full-adder using only 2-input NOR gates.

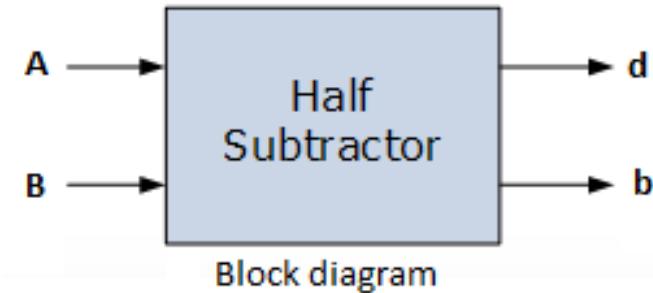
Subtractors

- In subtraction, each subtrahend bit of the number is subtracted from its corresponding significant minuend bit to form a difference bit.
- If the minuend bit is smaller than the subtrahend bit, a 1 is borrowed from the next significant position.

$$\begin{array}{ccccccc} & & & & & \text{(borrow)} \\ 0 & & 1 & & 1 & & 1 \rightarrow 0 \\ -0 & & -0 & & -1 & & -1 \\ \hline 0 & & 1 & & 0 & & 1 \end{array}$$

Half Subtractor

- A *half subtractor* is a combinational circuit that subtracts one bit from the other and produces the difference. It also has an output to specify if a 1 has been borrowed.
- It is used to subtract the LSB of the subtrahend from the LSB of the minuend when one binary number is subtracted from the other.
- A half subtractor is a combinational circuit with two inputs **A** and **B** and two outputs **d** and **b**, **d** indicates the difference and **b** is the output signal generated that informs the next stage that a 1 has been borrowed.



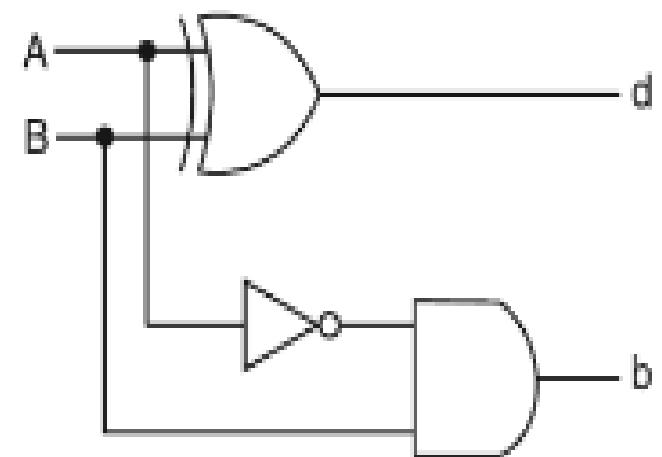
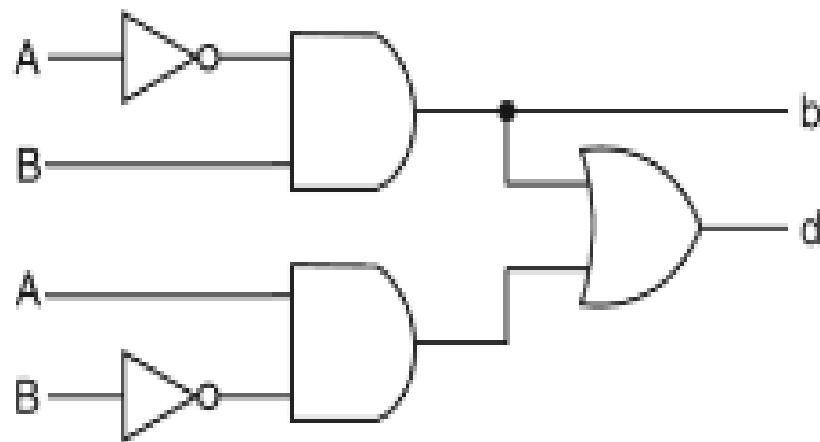
| Inputs | | Outputs | |
|--------|---|---------|---|
| A | B | b | d |
| 0 | 0 | 0 | 0 |
| 0 | 1 | 1 | 1 |
| 1 | 0 | 0 | 1 |
| 1 | 1 | 0 | 0 |

Truth Table

- The simplified Boolean functions for the two outputs can be obtained directly from the truth table. The simplified sum-of-products expressions are

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

$$b = \overline{AB}$$

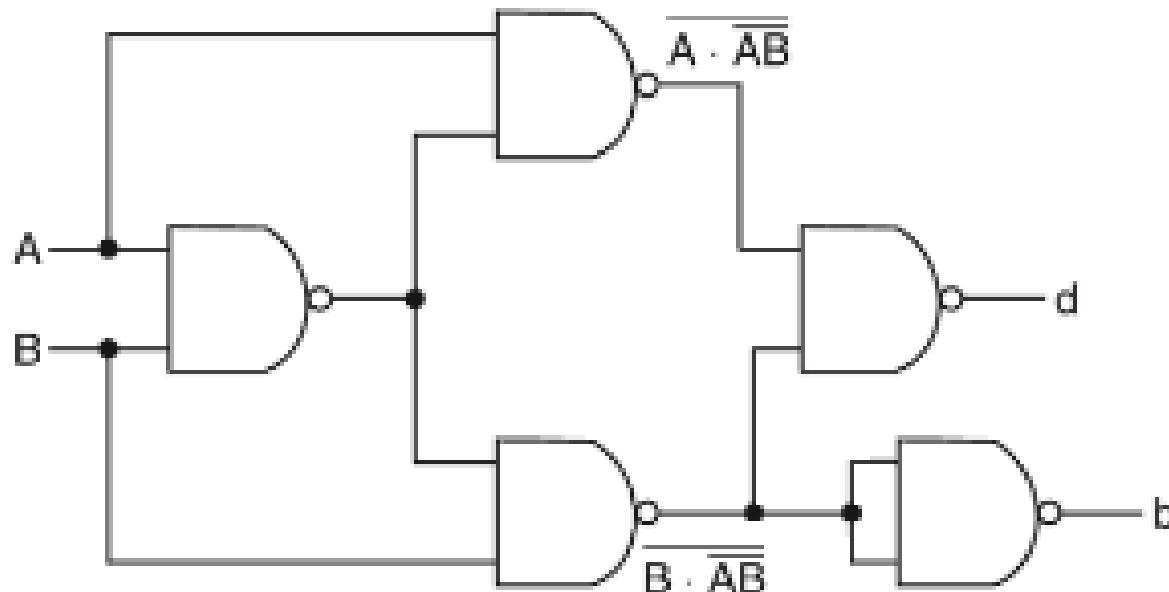


Logic diagrams of a half-subtractor.

Implementation of half subtractor by using only NAND gates.

$$d = A \oplus B = \overline{\overline{A} \cdot \overline{AB}} \cdot \overline{\overline{B} \cdot \overline{AB}}$$

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



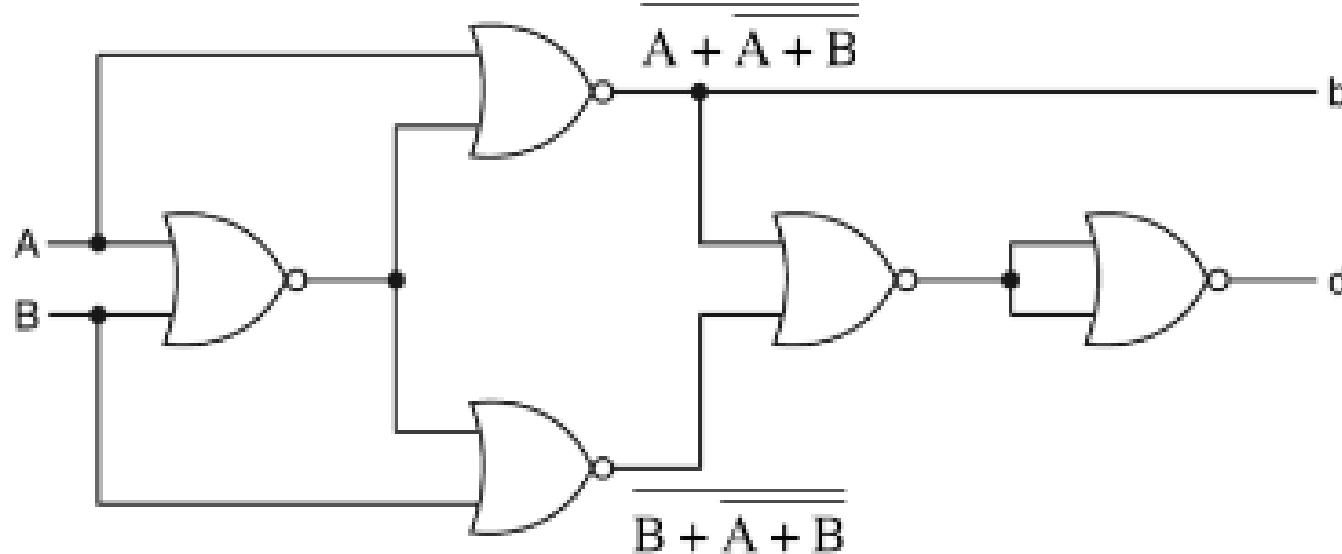
Logic diagram of a half-subtractor using only 2-input NAND gates.

Implementation of half subtractor by using only NOR gates.

$$d = A \oplus B = A\bar{B} + \bar{A}B = A\bar{B} + B\bar{B} + A\bar{A} + \bar{A}B$$

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

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

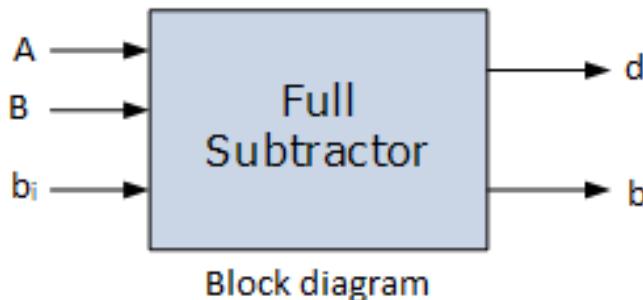


Logic diagram of a half-subtractor using only 2-input NOR gates.

Full Subtractor

- The *full subtractor* is a combinational circuit which is used to perform subtraction of three input bits.
- It subtracts one bit from another bit, when already there is a borrow from this column for the subtraction in the preceding column, and outputs the difference bit and the borrow bit required from the next column.
- So a full subtractor is a combinational circuit with three inputs (A , B , b_i) and two outputs d and b . The two outputs present the difference and output borrow.

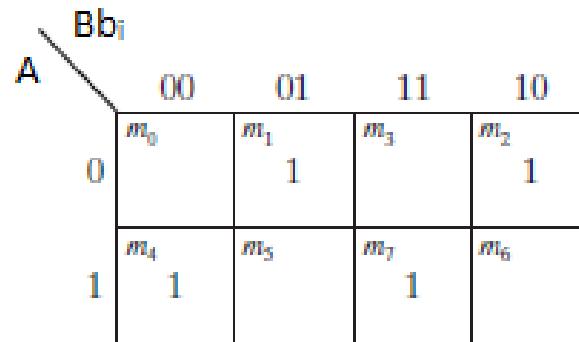
The simplified expressions are



| Inputs | | | Outputs | |
|--------|---|----------------|---------|---|
| A | B | b _i | b | d |
| 0 | 0 | 0 | 0 | 0 |
| 0 | 0 | 1 | 1 | 1 |
| 0 | 1 | 0 | 1 | 1 |
| 0 | 1 | 1 | 1 | 0 |
| 1 | 0 | 0 | 0 | 1 |
| 1 | 0 | 1 | 0 | 0 |
| 1 | 1 | 0 | 0 | 0 |
| 1 | 1 | 1 | 1 | 1 |

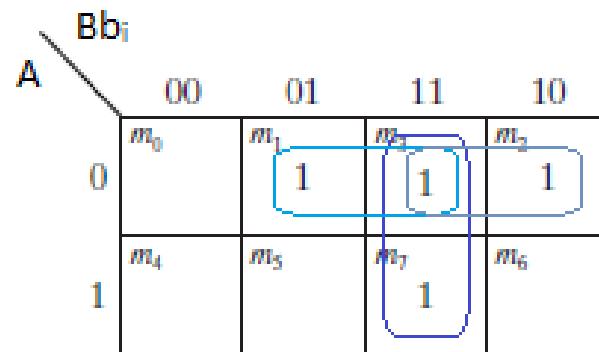
Truth Table

$$d = \sum(1, 2, 4, 7)$$

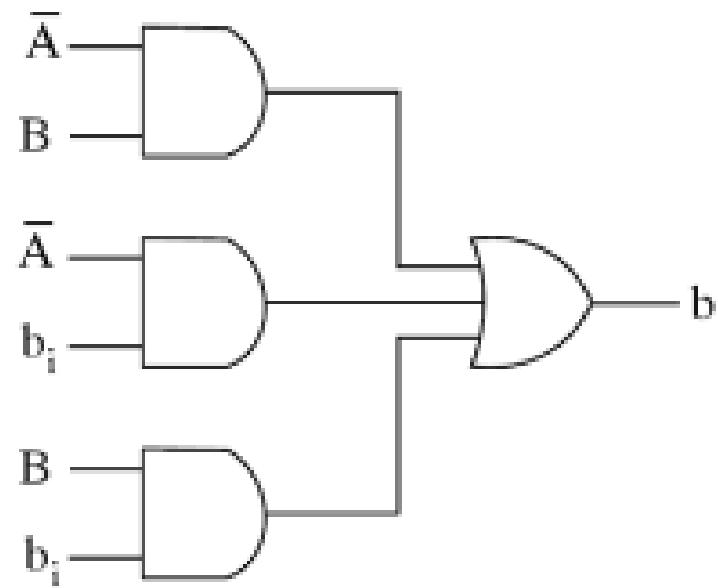
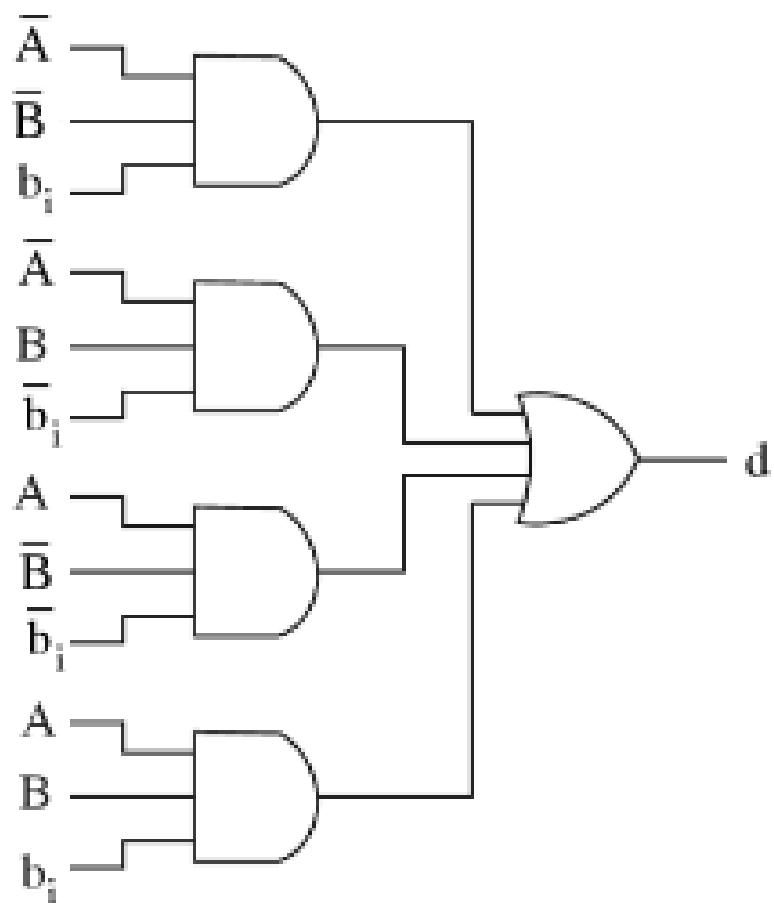


$$d = \bar{A}\bar{B}b_i + \bar{A}B\bar{b}_i + A\bar{B}\bar{b}_i + ABb_i$$

$$b = \sum(1, 2, 3, 7)$$



$$b = \bar{A}B + \bar{A}b_i + Bb_i$$



Logic diagram of a full-subtractor.

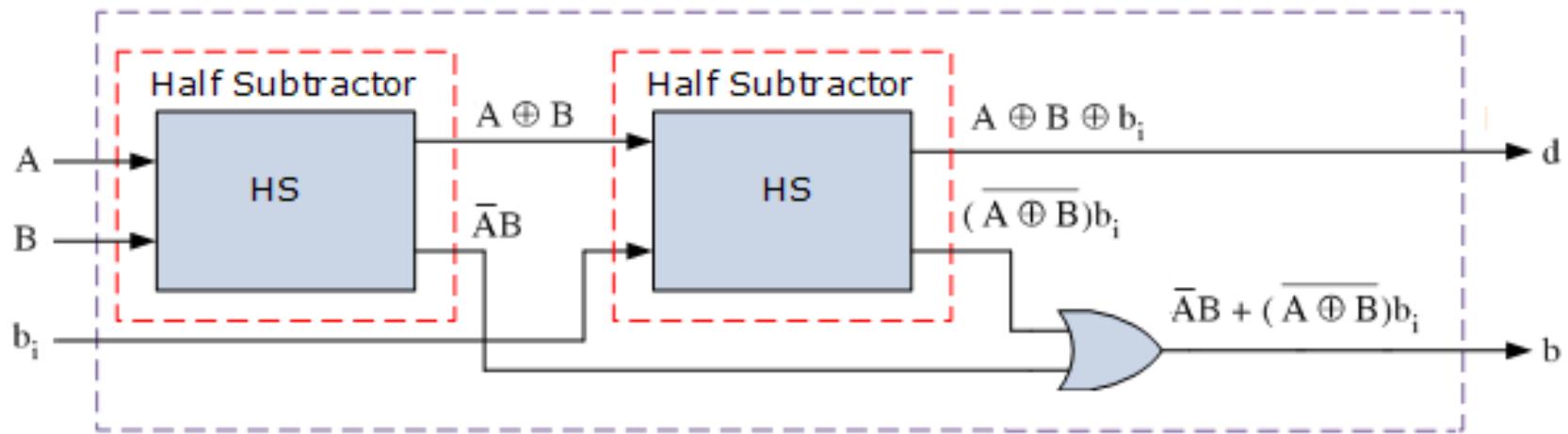
Implementation of full subtractor with two half subtractors and an OR gate.

- From the truth table of full adder, the outputs difference and borrow are described by

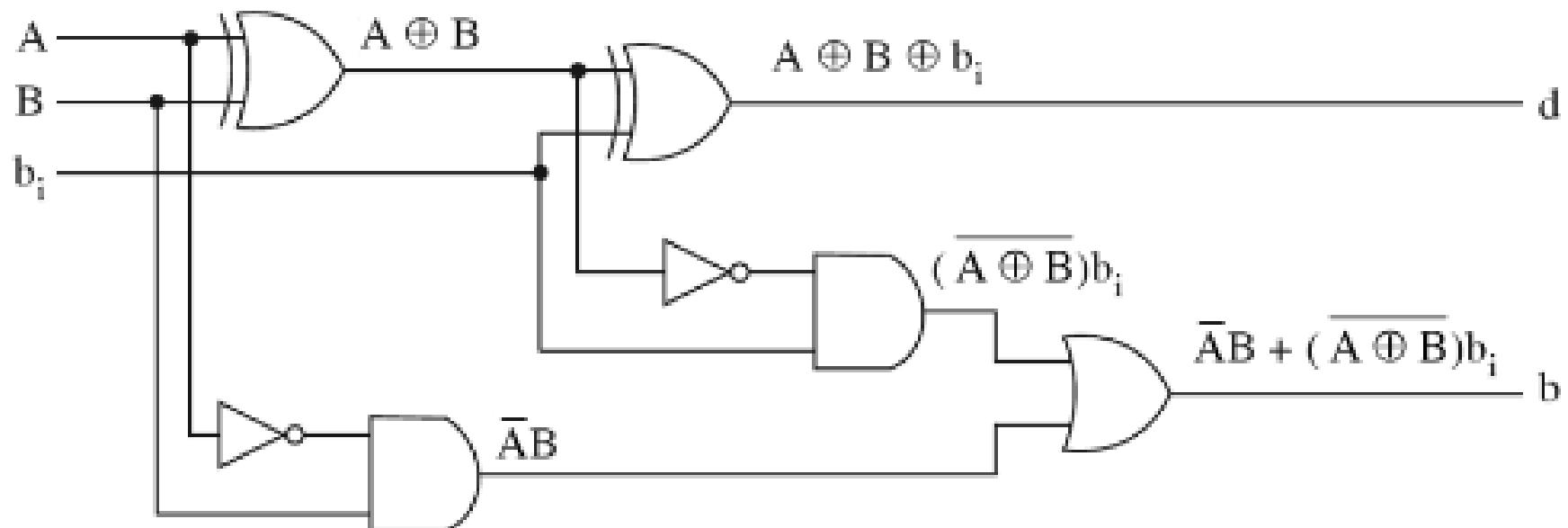
$$\begin{aligned}d &= \bar{A}\bar{B}b_i + \bar{A}B\bar{b}_i + A\bar{B}\bar{b}_i + ABb_i \\&= b_i(AB + \bar{A}\bar{B}) + \bar{b}_i(A\bar{B} + \bar{A}B) \\&= b_i(\overline{A \oplus B}) + \bar{b}_i(A \oplus B) = A \oplus B \oplus b_i\end{aligned}$$

and

$$\begin{aligned}b &= \bar{A}\bar{B}b_i + \bar{A}B\bar{b}_i + \bar{A}Bb_i + ABb_i \\&= \bar{A}B(b_i + \bar{b}_i) + (AB + \bar{A}\bar{B})b_i \\&= \bar{A}B + (\overline{A \oplus B})b_i\end{aligned}$$



Block diagram of a full-subtractor using two half-subtractor.



Logic diagram of a full-subtractor using two half-subtractor.

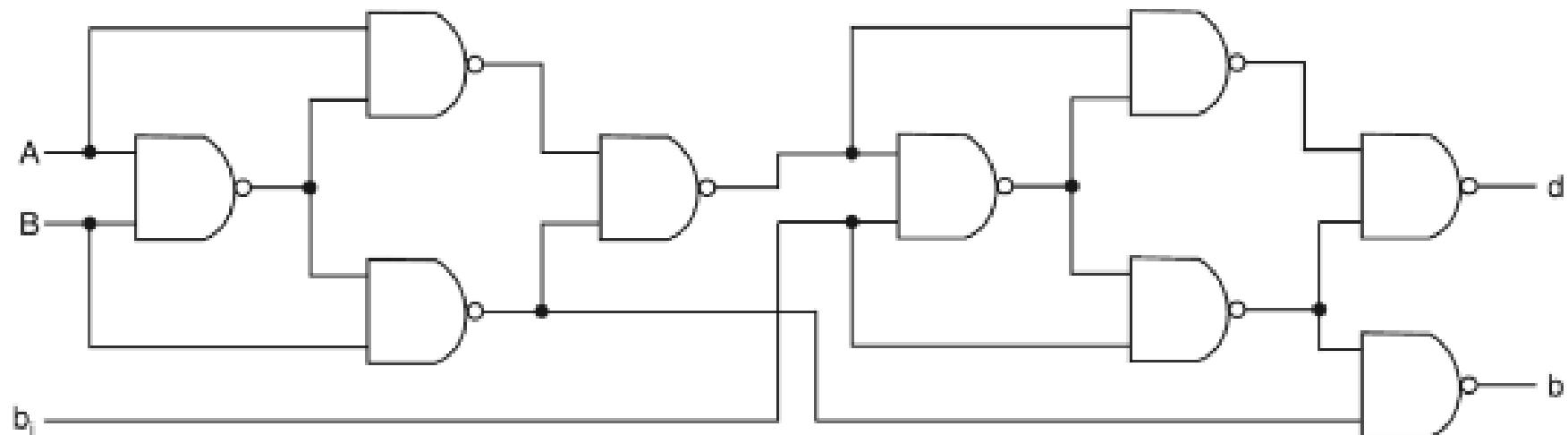
Implementation of full subtractor by using only NAND gates.

$$d = A \oplus B \oplus b_i = \overline{(A \oplus B)} \oplus b_i = \overline{(A \oplus B)} \cdot \overline{(A \oplus B)b_i} + b_i \cdot \overline{(A \oplus B)b_i}$$

$$b = \overline{A}B + b_i(\overline{A \oplus B}) = \overline{A}B + b_i(\overline{A \oplus B})$$

$$= \overline{\overline{A}B} \cdot b_i(A \oplus B) = \overline{B(\overline{A} + \overline{B})} \cdot b_i(\overline{b_i} + (A \oplus B))$$

$$= B \cdot \overline{AB} \cdot b_i[\overline{b_i} \cdot (A \oplus B)]$$

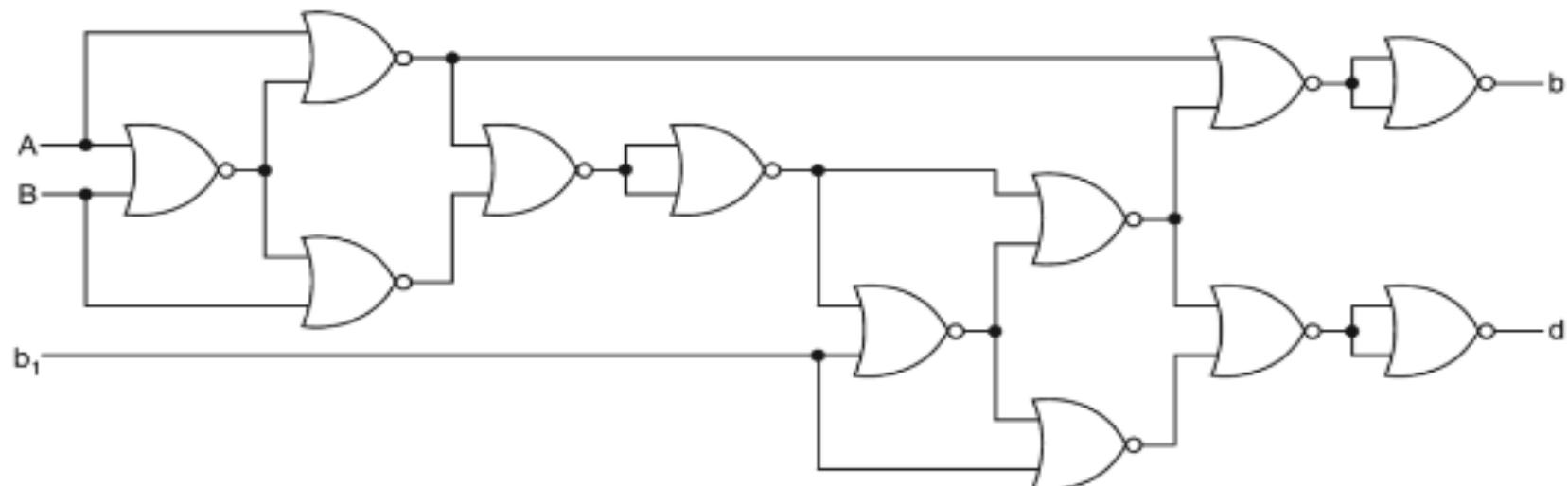


Logic diagram of a full-subtractor using only 2-input NAND gates.

Implementation of full subtractor by using only NOR gates.

$$\begin{aligned}d &= A \oplus B \oplus b_i = \overline{\overline{(A \oplus B)} \oplus b_i} \\&= \overline{(A \oplus B)b_i + (\overline{A \oplus B})\overline{b_i}} \\&= \overline{[(A \oplus B) + (\overline{A \oplus B})b_i][b_i + (\overline{A \oplus B})\overline{b_i}]} \\&= \overline{(A \oplus B) + (\overline{A \oplus B}) + b_i} + \overline{b_i + (\overline{A \oplus B}) + b_i} \\&= \overline{(A \oplus B) + (\overline{A \oplus B}) + b_i} + b_i + \overline{(A \oplus B) + b_i}\end{aligned}$$

$$\begin{aligned}b &= \overline{\overline{A}B + b_i(\overline{A \oplus B})} \\&= \overline{\overline{A}(A + B) + (\overline{A \oplus B})[(A \oplus B) + b_i]} \\&= \overline{\overline{A} + (\overline{A} + B) + (A \oplus B) + (\overline{A \oplus B}) + b_i} \\&= \overline{\overline{A} + (\overline{A} + B) + (A \oplus B) + (\overline{A \oplus B}) + b_i}\end{aligned}$$



Logic diagram of a full subtractor using only 2-input NOR gates.

DIGITAL ELECTRONICS

III Semester CSE & CST



Mr. Srikanta Patnaik Mr. Ajit Kumar Patro Mr. Ami Kumar Parida

DEPARTMENT OF ELECTRONICS AND COMMUNICATION ENGINEERING

SCHOOL OF ENGINEERING & TECHNOLOGY

GIET UNIVERSITY, GUNUPUR, ODISHA

**Binary Parallel Adder;
Binary Adder-Subtractor;
Binary Multiplier;
Magnitude Comparator;
Code Converters**

Binary Parallel Adder

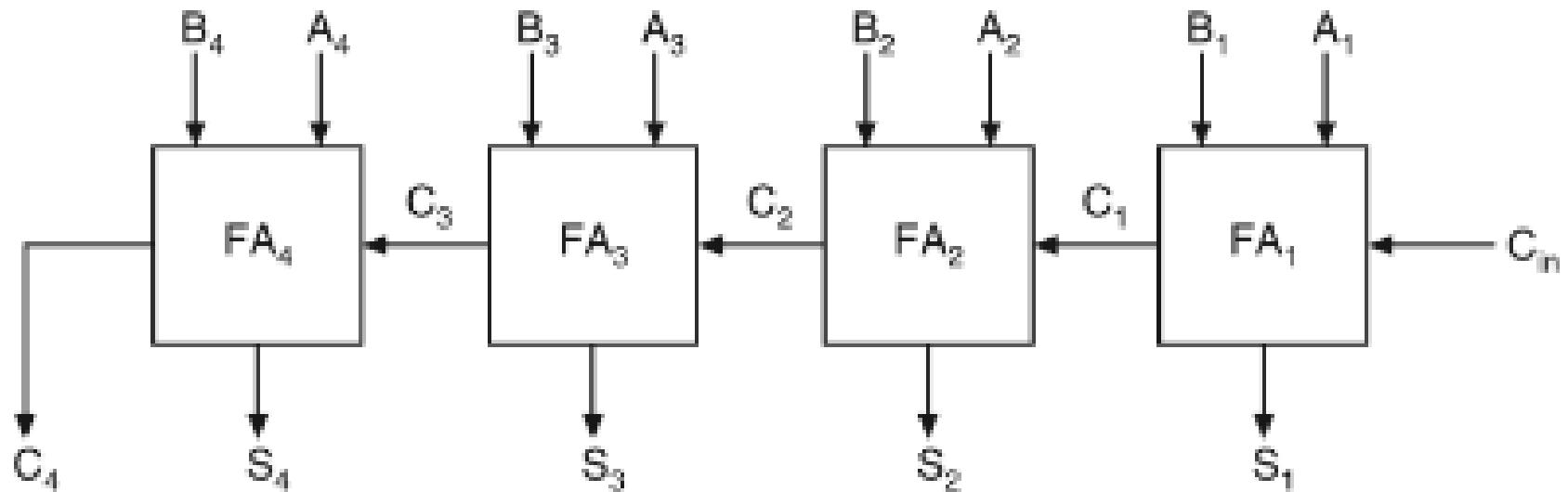
- A **binary adder** is a digital circuit that produces the arithmetic sum of two binary numbers.
- It can be constructed with full adders connected in cascade, with the output carry from each full adder connected to the input carry of the next full adder in the chain.

Example:

- Consider the two binary numbers $A = 1011$ and $B = 0011$. Their sum $S = 1110$ is formed with the four-bit adder as follows:

| Subscript i : | 3 | 2 | 1 | 0 | |
|-----------------|---|---|---|---|-----------|
| Input carry | 0 | 1 | 1 | 0 | C_i |
| Augend | 1 | 0 | 1 | 1 | A_i |
| Addend | 0 | 0 | 1 | 1 | B_i |
| Sum | 1 | 1 | 1 | 0 | S_i |
| Output carry | 0 | 0 | 1 | 1 | C_{i+1} |

- Figure shows the interconnection of four full-adder (FA) circuits to provide a 4-bit binary parallel adder.



Logic diagram of a 4-bit binary parallel adder.

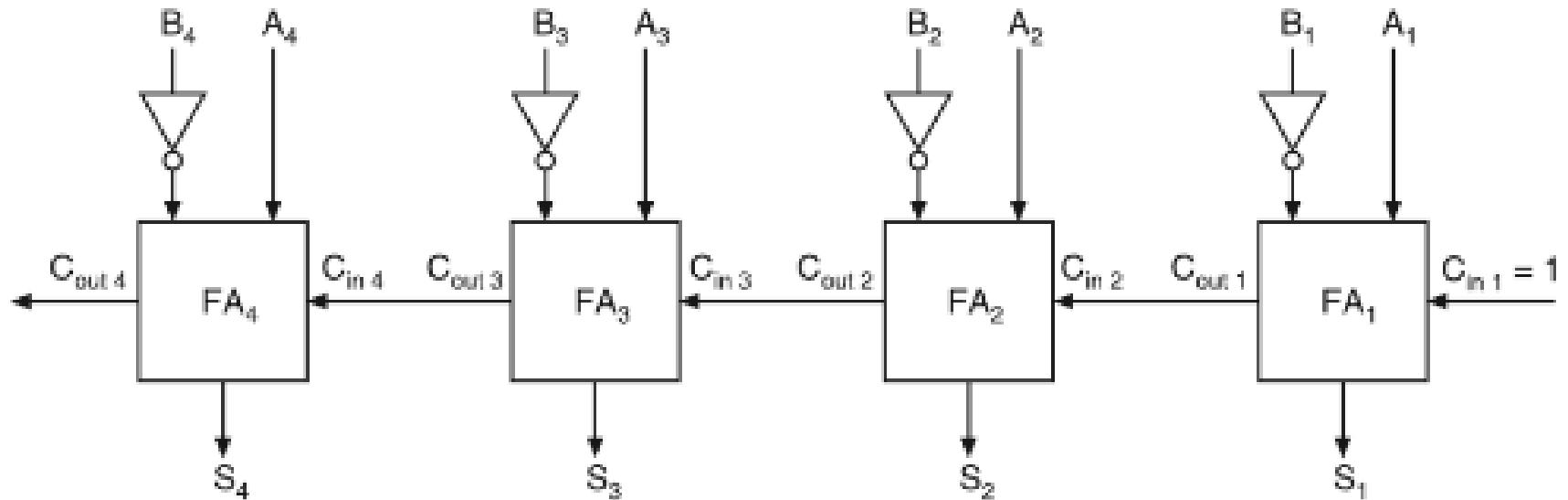
- The augend bits of A and the addend bits of B are designated by subscript numbers from right to left, with subscript 1 denoting the least significant bit.
- The carries are connected in a chain through the full adders. The input carry to the adder is C_{in} , and it ripples through the full adders to the output carry C_4 .
- The S outputs generate the required sum bits.

- An n -bit adder requires n full adders, with each output carry connected to the input carry of the next higher order full adder. The input carry to the least significant position is fixed at 0.
- The parallel adder, in which the carry-out of each full-adder is the carry-in to the next most significant adder is called a *ripple carry adder*.

Binary Parallel Subtractor

- The subtraction of unsigned binary numbers can be done most conveniently by means of complements.
- Remember that the subtraction $A - B$ can be done by taking the 2's complement of B and adding it to A .
- The 2's complement can be obtained by taking the 1's complement and adding 1 to the least significant pair of bits.

- The 1's complement can be implemented with inverters, and a 1 can be added to the sum through the input carry.

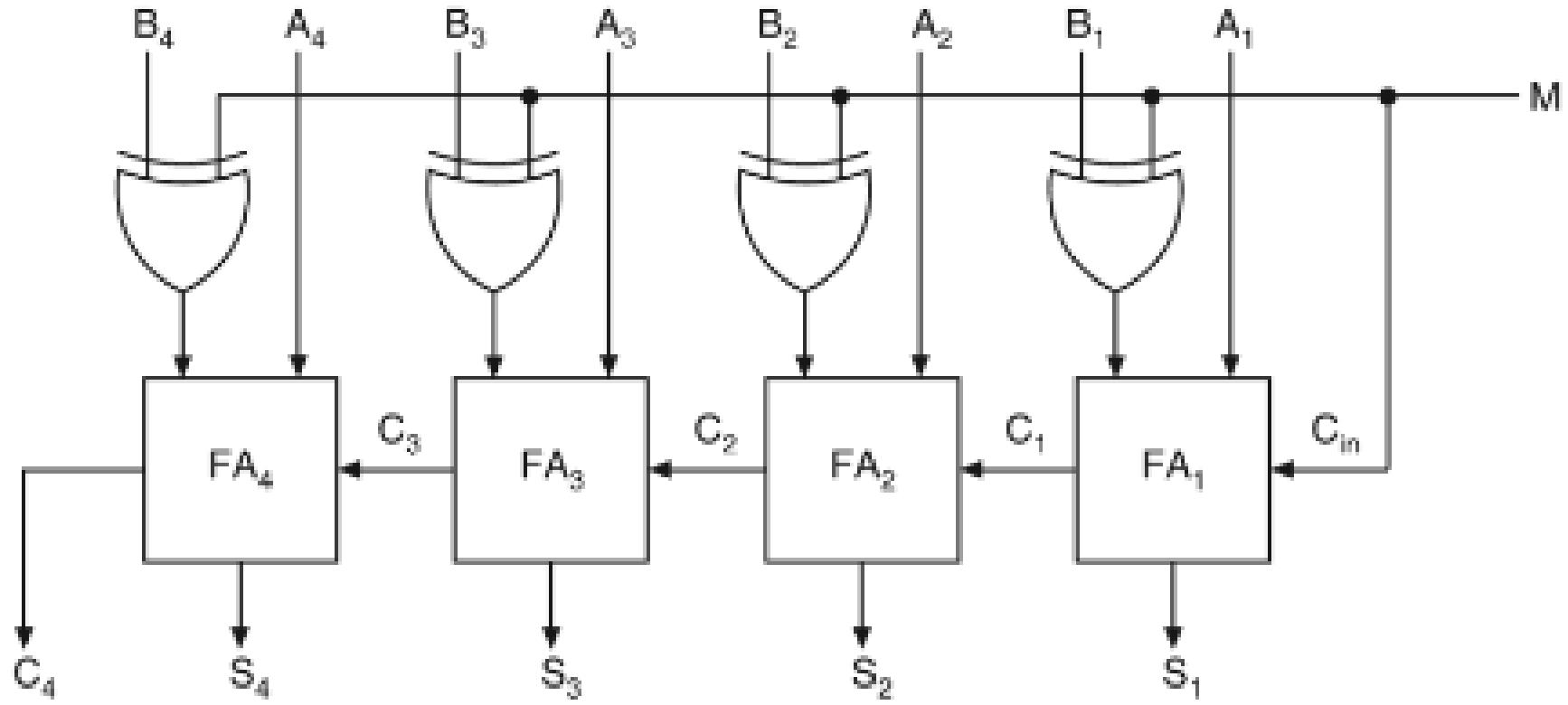


Logic diagram of a 4-bit parallel subtractor.

- The circuit for subtracting $A - B$ consists of an adder with inverters placed between each data input B and the corresponding input of the full adder.
- The input carry C_{in} must be equal to 1 when subtraction is performed.
- The operation thus performed becomes A , plus the 1's complement of B , plus 1. This is equal to A plus the 2's complement of B .
- For unsigned numbers, that gives $A - B$ if $A \geq B$ or the 2's complement of $(B - A)$ if $A < B$.

Binary Adder-Subtractor

- The addition and subtraction operations can be combined into one circuit with one common binary adder by including an exclusive-OR gate with each full adder.
- A 4-bit adder-subtractor circuit is shown in Fig.
- The mode input M controls the operation.
- When $M = 0$, the circuit is an **adder**, and when $M = 1$, the circuit becomes a **subtractor**.



Logic diagram of a 4-bit binary adder-subtractor.

- Each exclusive-OR gate receives input M and one of the inputs of B .
- When $M = 0$, we have $B \oplus 0 = B$. The full adders receive the value of B , the input carry is 0, and the circuit performs A plus B .
- When $M = 1$, we have $B \oplus 1 = B'$ and $C_{in} = 1$. The B inputs are all complemented and a 1 is added through the input carry. The circuit performs the operation A plus the 2's complement of B .

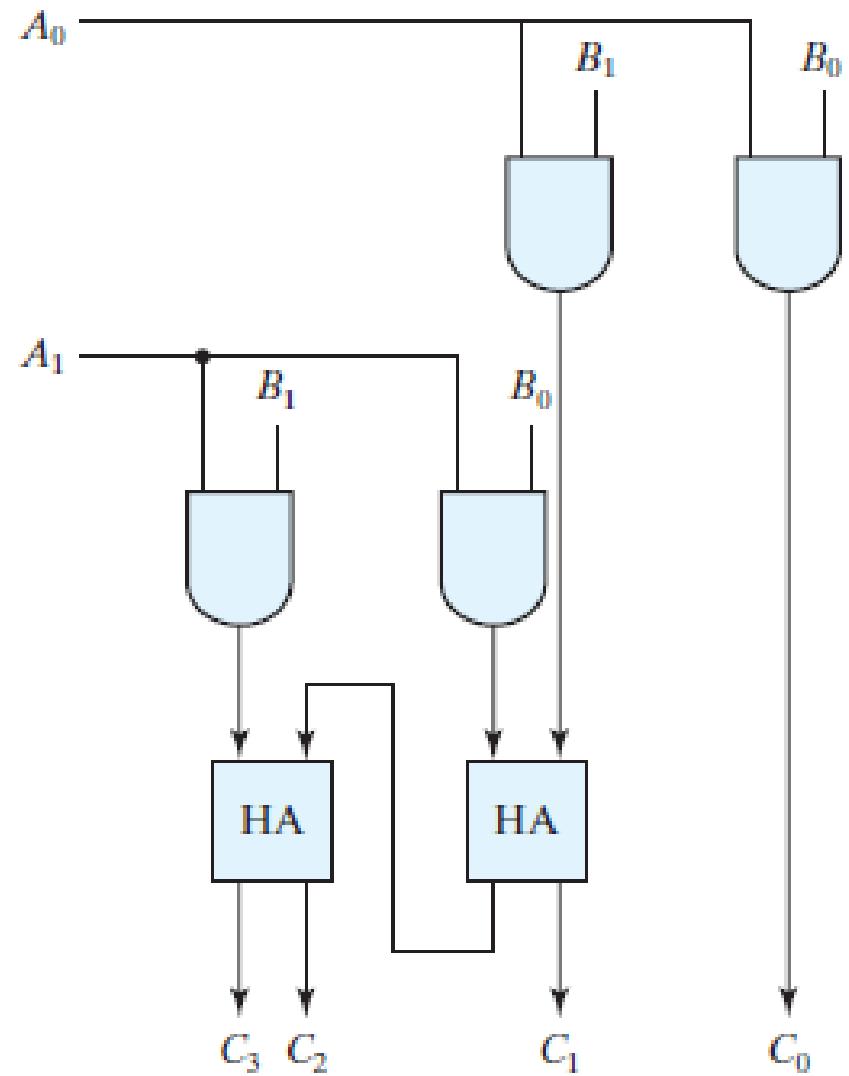
Binary Multiplier

- Multiplication of binary numbers is performed in the same way as multiplication of decimal numbers.
- The multiplicand is multiplied by each bit of the multiplier, starting from the least significant bit.
- Each such multiplication forms a partial product. Successive partial products are shifted one position to the left.
- The final product is obtained from the sum of the partial products.

- Consider the multiplication of two 2-bit numbers as shown in Fig.
- The multiplicand bits are B_1 and B_0 , the multiplier bits are A_1 and A_0 , and the product is $C_3C_2C_1C_0$.
- The first partial product is formed by multiplying B_1B_0 by A_0 .
- The multiplication of two bits such as A_0 and B_0 produces a 1 if both bits are 1; otherwise, it produces a 0. This is identical to an AND operation. Therefore, the partial product can be implemented with AND gates as shown in the diagram.

- The second partial product is formed by multiplying B_1B_0 by A_1 and shifting one position to the left.
- The two partial products are added with two half-adder (HA) circuits.
- Usually, there are more bits in the partial products and it is necessary to use full adders to produce the sum of the partial products.
- Note that the least significant bit of the product does not have to go through an adder, since it is formed by the output of the first AND gate.

$$\begin{array}{r}
 & B_1 & B_0 \\
 & \underline{A_1} & \underline{A_0} \\
 A_0B_1 & & A_0B_0 \\
 \hline
 A_1B_1 & A_1B_0 \\
 \hline
 C_3 & C_2 & C_1 & C_0
 \end{array}$$



Two-bit by two-bit binary multiplier

- A combinational circuit binary multiplier with more bits can be constructed in a similar fashion.
- A bit of the multiplier is ANDed with each bit of the multiplicand in as many levels as there are bits in the multiplier.
- The binary output in each level of AND gates is added with the partial product of the previous level to form a new partial product.
- The last level produces the product.
- For J multiplier bits and K multiplicand bits, we need $(J \times K)$ AND gates and $(J - 1)K$ -bit adders to produce a product of $(J + K)$ bits.

Comparators

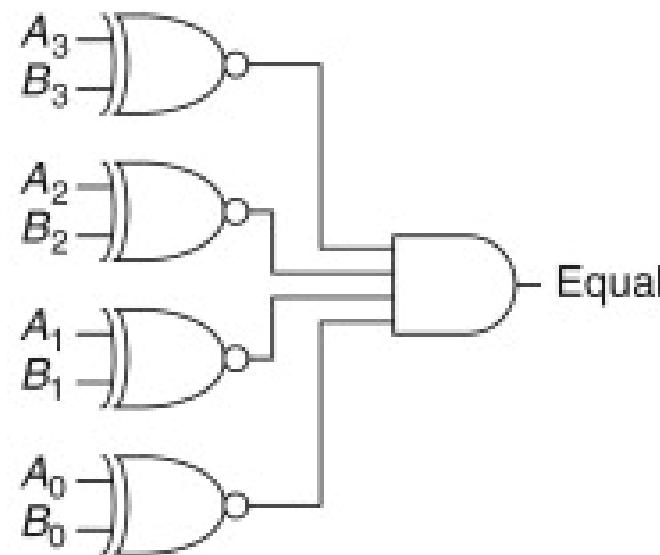
- A **comparator** is a logic circuit, used to compare the magnitudes of two binary numbers.
- The comparison of two numbers is an operation that determines whether one number is greater than, less than, or equal to the other number.
- There are two common types of comparators.
 - Equality Comparator
 - Magnitude Comparator

Equality Comparator

- An *equality comparator* produces a single output indicating whether A is equal to B.
- Two binary numbers are equal, if and only if all their corresponding bits coincide.
- For example, two 4-bit binary numbers, $A_3A_2A_1A_0$ and $B_3B_2B_1B_0$ are equal, if and only if, $A_3 = B_3$, $A_2 = B_2$, $A_1 = B_1$, and $A_0 = B_0$.

- The equality can be expressed logically with an exclusive-NOR function as

$$\text{EQUALITY} = (A_3 \oplus B_3) (A_2 \oplus B_2) (A_1 \oplus B_1) (A_0 \oplus B_0)$$



Logic diagram of equality comparator.

Magnitude Comparator

- A *magnitude comparator* is a combinational circuit that compares two numbers A and B and determines their relative magnitudes.
- The outcome of the comparison is specified by three binary variables that indicate whether $A > B$, $A = B$, or $A < B$.

1-bit Magnitude Comparator

The logic for a 1-bit magnitude comparator: Let the 1-bit numbers be $A = A_0$ and $B = B_0$.

If $A_0 = 1$ and $B_0 = 0$, then $A > B$.

Therefore,

$$A > B : G = A_0 \bar{B}_0$$

If $A_0 = 0$ and $B_0 = 1$, then $A < B$.

Therefore,

$$A < B : L = \bar{A}_0 B_0$$

If A_0 and B_0 coincide, i.e. $A_0 = B_0 = 0$ or if $A_0 = B_0 = 1$, then $A = B$.

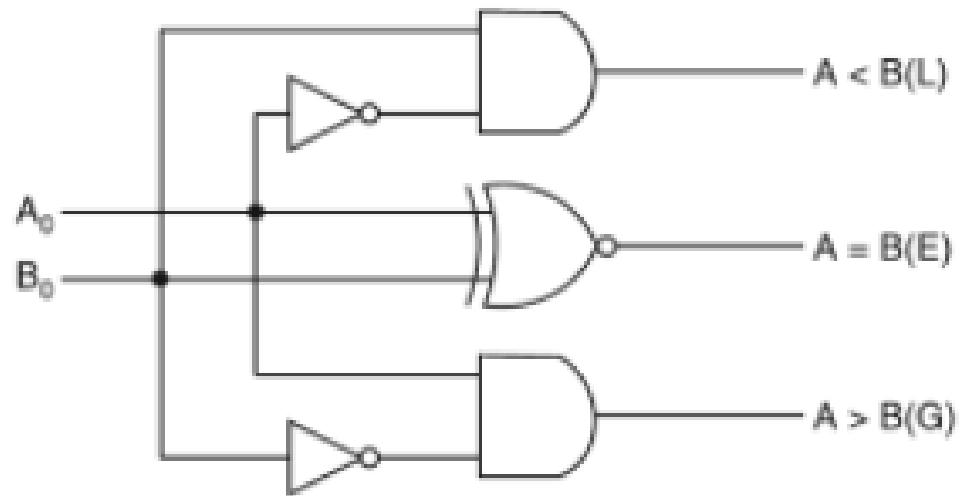
Therefore,

$$A = B : E = A_0 \oplus B_0$$

The truth table and the logic diagram for the 1-bit comparator are shown in Figure . The logic expressions for G , L , and E can also be obtained from the truth table.

| A_0 | B_0 | L | E | G |
|-------|-------|---|---|---|
| 0 | 0 | 0 | 1 | 0 |
| 0 | 1 | 1 | 0 | 0 |
| 1 | 0 | 0 | 0 | 1 |
| 1 | 1 | 0 | 1 | 0 |

(a) Truth table



(b) Logic diagram

1-bit magnitude comparator: .

2-bit Magnitude Comparator

The logic for a 2-bit magnitude comparator: Let the two 2-bit numbers be $A = A_1A_0$ and $B = B_1B_0$.

1. If $A_1 = 1$ and $B_1 = 0$, then $A > B$ or
2. If A_1 and B_1 coincide and $A_0 = 1$ and $B_0 = 0$, then $A > B$. So the logic expression for $A > B$ is

$$A > B : G = A_1\bar{B}_1 + (A_1 \odot B_1)A_0\bar{B}_0$$

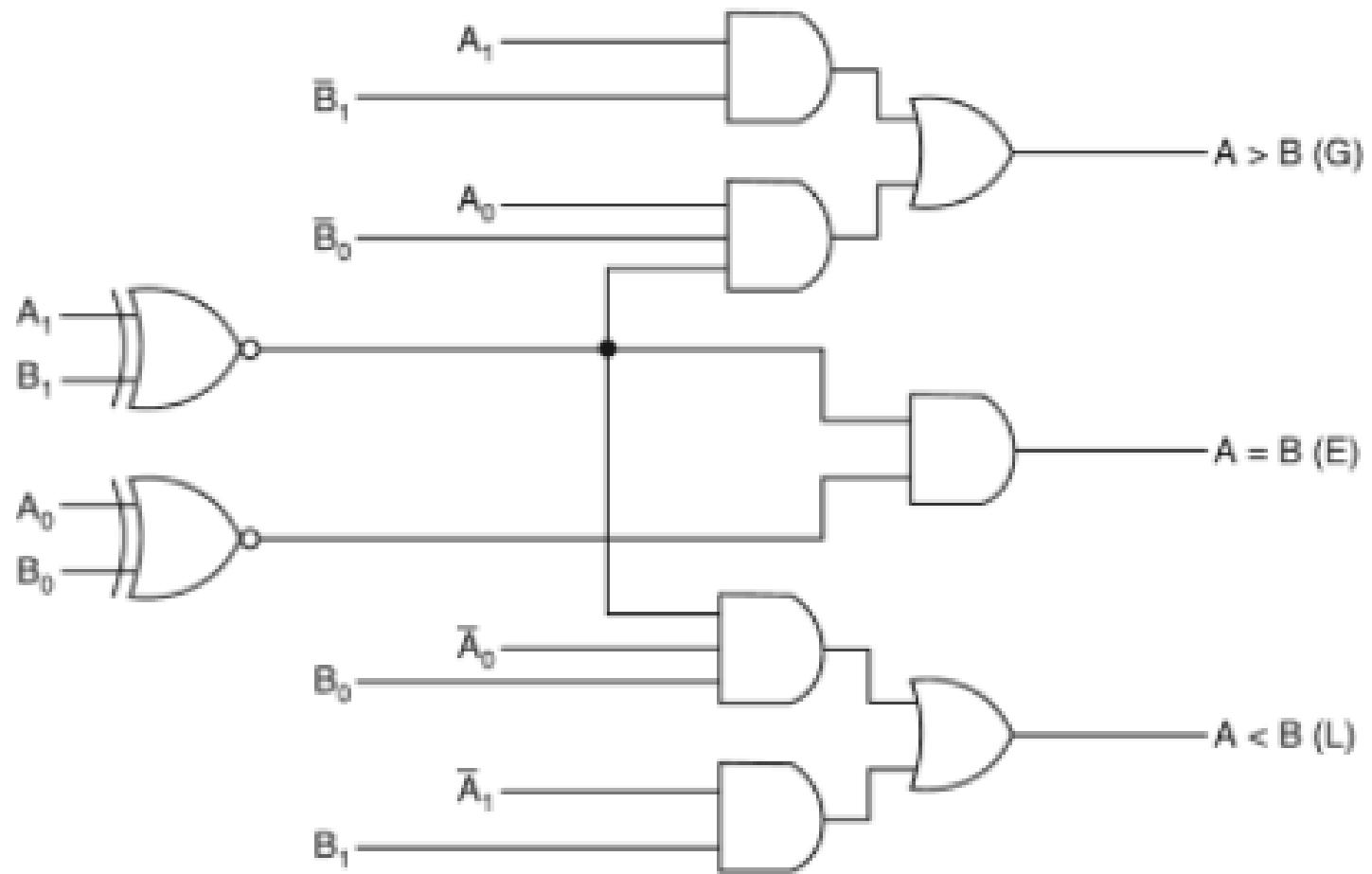
1. If $A_1 = 0$ and $B_1 = 1$, then $A < B$ or
2. If A_1 and B_1 coincide and $A_0 = 0$ and $B_0 = 1$, then $A < B$. So the expression for $A < B$ is

$$A < B : L = \bar{A}_1B_1 + (A_1 \odot B_1)\bar{A}_0B_0$$

If A_1 and B_1 coincide and if A_0 and B_0 coincide then $A = B$. So the expression for $A = B$ is

$$A = B : E = (A_1 \odot B_1)(A_0 \odot B_0)$$

The logic diagram for a 2-bit comparator is as shown in Figure.



Logic diagram of a 2-bit magnitude comparator.

4-bit Magnitude Comparator

The logic for a 4-bit magnitude comparator: Let the two 4-bit numbers be $A = A_3A_2A_1A_0$ and $B = B_3B_2B_1B_0$.

1. If $A_3 = 1$ and $B_3 = 0$, then $A > B$. Or
2. If A_3 and B_3 coincide, and if $A_2 = 1$ and $B_2 = 0$, then $A > B$. Or
3. If A_3 and B_3 coincide, and if A_2 and B_2 coincide, and if $A_1 = 1$ and $B_1 = 0$, then $A > B$. Or
4. If A_3 and B_3 coincide, and if A_2 and B_2 coincide, and if A_1 and B_1 coincide, and if $A_0 = 1$ and $B_0 = 0$, then $A > B$.

From these statements, we see that the logic expression for $A > B$ can be written as

$$(A > B) : G = A_3\bar{B}_3 + (A_3 \odot B_3)A_2\bar{B}_2 + (A_3 \odot B_3)(A_2 \odot B_2)A_1\bar{B}_1 \\ + (A_3 \odot B_3)(A_2 \odot B_2)(A_1 \odot B_1)A_0\bar{B}_0$$

Similarly, the logic expression for $A < B$ can be written as

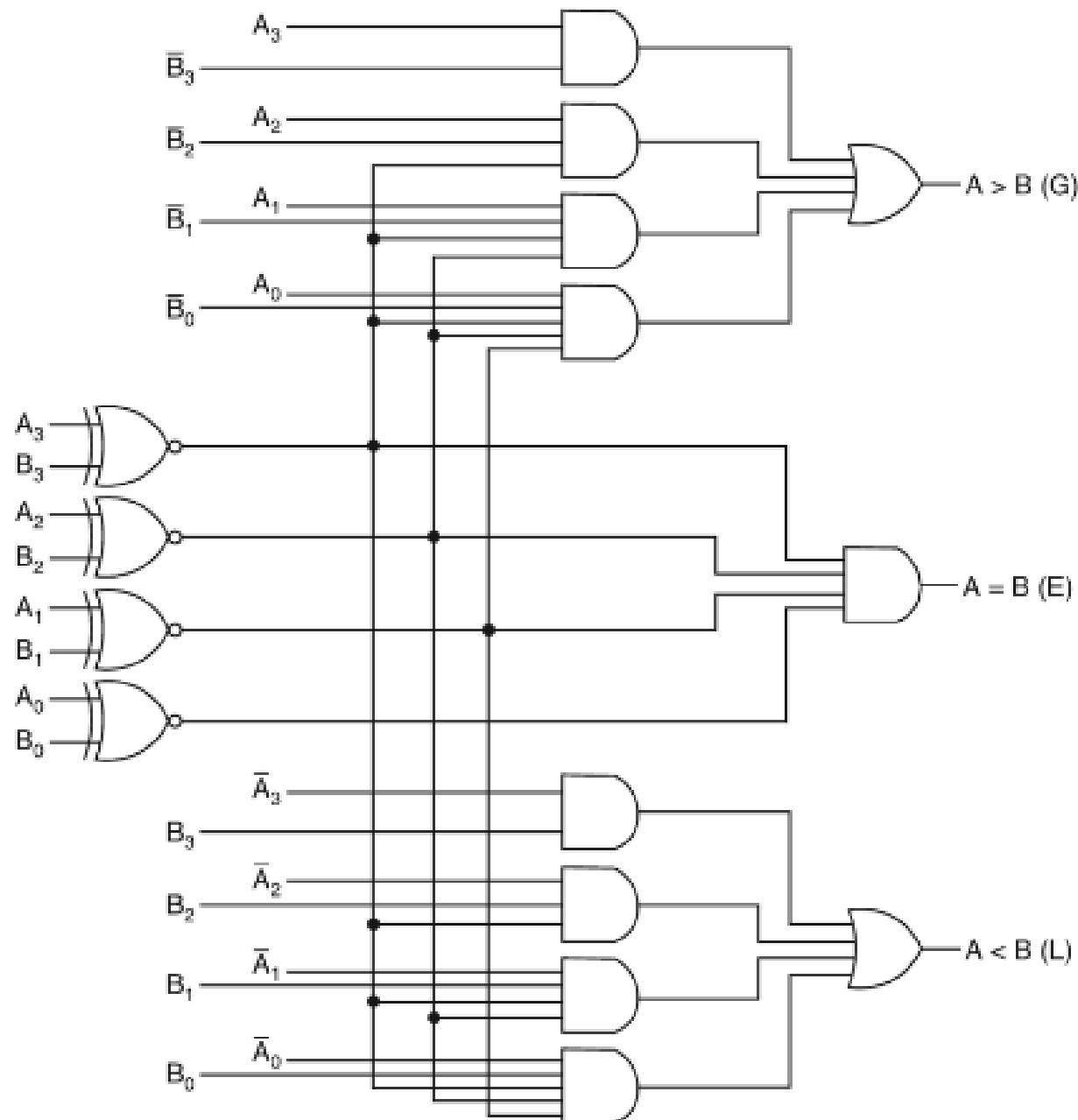
$$(A < B) : L = \bar{A}_3B_3 + (A_3 \odot B_3)\bar{A}_2B_2 + (A_3 \odot B_3)(A_2 \odot B_2)\bar{A}_1B_1 \\ + (A_3 \odot B_3)(A_2 \odot B_2)(A_1 \odot B_1)\bar{A}_0B_0$$

If A_3 and B_3 coincide and if A_2 and B_2 coincide and if A_1 and B_1 coincide and if A_0 and B_0 coincide, then $A = B$.

So the expression for $A = B$ can be written as

$$(A = B) : E = (A_3 \odot B_3)(A_2 \odot B_2)(A_1 \odot B_1)(A_0 \odot B_0)$$

Figure shows the logic diagram of a comparator that implements the logic we have described. Note that, it provides three active-HIGH outputs: $A > B$, $A < B$, and $A = B$.



Logic diagram of a 4-bit magnitude comparator.

Code Converters

- A **code converter** is a logic circuit whose inputs are bit patterns representing numbers (or characters) in one code and whose outputs are the corresponding representations in a different code.
- Code converters are usually multiple output circuits.
- To convert from binary code A to binary code B, the input lines must supply the bit combination of elements as specified by code A and the output lines must generate the corresponding bit combination of code B. A combinational circuit performs this transformation by means of logic gates.

Design of a 4-bit Binary-to-Gray Code Converter.

- The input to the 4-bit binary-to-Gray code converter circuit is a 4-bit binary and the output is a 4-bit Gray code. There are 16 possible combinations of 4-bit binary input and all of them are valid. Hence no don't cares.
- The 4-bit binary code and the corresponding output Gray code are shown in the conversion table.

| 4-bit binary | | | | 4-bit Gray | | | |
|----------------|----------------|----------------|----------------|----------------|----------------|----------------|----------------|
| B ₄ | B ₃ | B ₂ | B ₁ | G ₄ | G ₃ | G ₂ | G ₁ |
| 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 |
| 0 | 0 | 0 | 1 | 0 | 0 | 0 | 1 |
| 0 | 0 | 1 | 0 | 0 | 0 | 1 | 1 |
| 0 | 0 | 1 | 1 | 0 | 0 | 1 | 0 |
| 0 | 1 | 0 | 0 | 0 | 1 | 1 | 0 |
| 0 | 1 | 0 | 1 | 0 | 1 | 1 | 1 |
| 0 | 1 | 1 | 0 | 0 | 1 | 0 | 1 |
| 0 | 1 | 1 | 1 | 0 | 1 | 0 | 0 |
| 1 | 0 | 0 | 0 | 1 | 1 | 0 | 0 |
| 1 | 0 | 0 | 1 | 1 | 1 | 0 | 1 |
| 1 | 0 | 1 | 0 | 1 | 1 | 1 | 1 |
| 1 | 0 | 1 | 1 | 1 | 1 | 1 | 0 |
| 1 | 1 | 0 | 0 | 1 | 0 | 1 | 0 |
| 1 | 1 | 0 | 1 | 1 | 0 | 1 | 1 |
| 1 | 1 | 1 | 0 | 1 | 0 | 0 | 1 |
| 1 | 1 | 1 | 1 | 1 | 0 | 0 | 0 |

Conversion table

4-bit binary-to-Gray code converter

- From the conversion table, we observe that expressions for the outputs G_4 , G_3 , G_2 , and G_1 are as follows:

$$G_4 = \Sigma m(8, 9, 10, 11, 12, 13, 14, 15)$$

$$G_3 = \Sigma m(4, 5, 6, 7, 8, 9, 10, 11)$$

$$G_2 = \Sigma m(2, 3, 4, 5, 10, 11, 12, 13)$$

$$G_1 = \Sigma m(1, 2, 5, 6, 9, 10, 13, 14)$$

K-map for G_4

| | | B_2B_1 | 00 | 01 | 11 | 10 | |
|--|--|----------|----|----|----|----|----|
| | | B_4B_3 | 00 | 0 | 1 | 3 | 2 |
| | | B_4B_3 | 01 | 4 | 5 | 7 | 6 |
| | | B_2B_1 | 11 | 12 | 13 | 15 | 14 |
| | | B_4B_3 | 1 | 1 | 1 | 1 | 1 |
| | | B_2B_1 | 10 | 8 | 9 | 11 | 10 |
| | | B_4B_3 | 1 | 1 | 1 | 1 | 1 |

$$G_4 = B_4$$

K-map for G_3

| | | B_2B_1 | 00 | 01 | 11 | 10 | |
|--|--|----------|----|----|----|----|----|
| | | B_4B_3 | 00 | 0 | 1 | 3 | 2 |
| | | B_4B_3 | 01 | 4 | 5 | 7 | 6 |
| | | B_2B_1 | 11 | 12 | 13 | 15 | 14 |
| | | B_4B_3 | 1 | 1 | 1 | 1 | 1 |
| | | B_2B_1 | 10 | 8 | 9 | 11 | 10 |
| | | B_4B_3 | 1 | 1 | 1 | 1 | 1 |

$$G_3 = \bar{B}_4B_3 + B_4\bar{B}_3$$

K-map for G_2

| | | B_2B_1 | 00 | 01 | 11 | 10 | |
|--|--|----------|----|----|----|----|----|
| | | B_4B_3 | 00 | 0 | 1 | 3 | 2 |
| | | B_4B_3 | 01 | 4 | 5 | 7 | 6 |
| | | B_2B_1 | 11 | 12 | 13 | 15 | 14 |
| | | B_4B_3 | 1 | 1 | 1 | 1 | 1 |
| | | B_2B_1 | 10 | 8 | 9 | 11 | 10 |
| | | B_4B_3 | 1 | 1 | 1 | 1 | 1 |

$$G_2 = \bar{B}_3B_2 + B_3\bar{B}_2$$

K-map for G_1

| | | B_2B_1 | 00 | 01 | 11 | 10 | |
|--|--|----------|----|----|----|----|----|
| | | B_4B_3 | 00 | 0 | 1 | 3 | 2 |
| | | B_4B_3 | 01 | 4 | 5 | 7 | 6 |
| | | B_2B_1 | 11 | 12 | 13 | 15 | 14 |
| | | B_4B_3 | 1 | 1 | 1 | 1 | 1 |
| | | B_2B_1 | 10 | 8 | 9 | 11 | 10 |
| | | B_4B_3 | 1 | 1 | 1 | 1 | 1 |

$$G_1 = \bar{B}_2B_1 + B_2\bar{B}_1$$

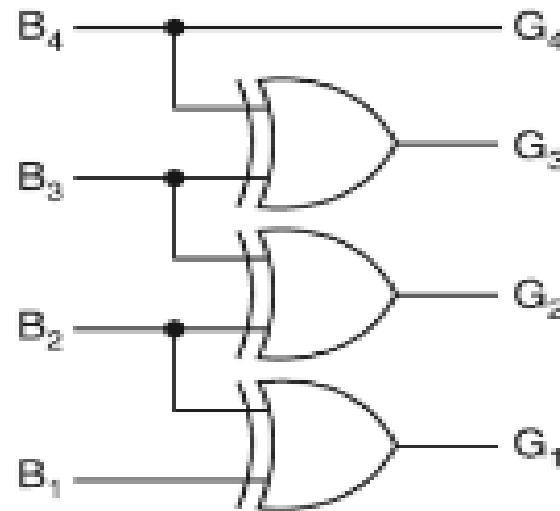
- The minimal expressions for the outputs obtained from the K-map are:

$$G_4 = B_4$$

$$G_3 = \bar{B}_4 B_3 + B_4 \bar{B}_3 = B_4 \oplus B_3$$

$$G_2 = \bar{B}_3 B_2 + B_3 \bar{B}_2 = B_3 \oplus B_2$$

$$G_1 = \bar{B}_2 B_1 + B_2 \bar{B}_1 = B_2 \oplus B_1$$



Logic diagram

4-bit binary-to-Gray code converter.

Design of a 4-bit Gray-to-Binary Code Converter.

- The input to the 4-bit Gray-to-binary code converter circuit is a 4-bit Gray and the output is a 4-bit binary code. There are 16 possible combinations of 4-bit Gray input and all of them are valid. Hence no don't cares.
- The 4-bit Gray code and the corresponding output binary code are shown in the conversion table.

| 4-bit Gray | | | | 4-bit binary | | | |
|------------|-------|-------|-------|--------------|-------|-------|-------|
| G_4 | G_3 | G_2 | G_1 | B_4 | B_3 | B_2 | B_1 |
| 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 |
| 0 | 0 | 0 | 1 | 0 | 0 | 0 | 1 |
| 0 | 0 | 1 | 1 | 0 | 0 | 1 | 0 |
| 0 | 0 | 1 | 0 | 0 | 0 | 1 | 1 |
| 0 | 1 | 1 | 0 | 0 | 1 | 0 | 0 |
| 0 | 1 | 1 | 1 | 0 | 1 | 0 | 1 |
| 0 | 1 | 0 | 1 | 0 | 1 | 1 | 0 |
| 0 | 1 | 0 | 0 | 0 | 1 | 1 | 1 |
| 1 | 1 | 0 | 0 | 1 | 0 | 0 | 0 |
| 1 | 1 | 0 | 1 | 1 | 0 | 0 | 1 |
| 1 | 1 | 1 | 1 | 1 | 0 | 1 | 0 |
| 1 | 1 | 1 | 0 | 1 | 0 | 1 | 1 |
| 1 | 0 | 1 | 0 | 1 | 1 | 0 | 0 |
| 1 | 0 | 1 | 1 | 1 | 1 | 0 | 1 |
| 1 | 0 | 0 | 1 | 1 | 1 | 1 | 0 |
| 1 | 0 | 0 | 0 | 1 | 1 | 1 | 1 |

Conversion table

4-bit Gray-to-binary code converter.

- From the conversion table, we observe that expressions for the outputs B_4 , B_3 , B_2 , and B_1 are as follows:

$$B_4 = \sum m(12, 13, 15, 14, 10, 11, 9, 8) = \sum m(8, 9, 10, 11, 12, 13, 14, 15)$$

$$B_3 = \sum m(6, 7, 5, 4, 10, 11, 9, 8) = \sum m(4, 5, 6, 7, 8, 9, 10, 11)$$

$$B_2 = \sum m(3, 2, 5, 4, 15, 14, 9, 8) = \sum m(2, 3, 4, 5, 8, 9, 14, 15)$$

$$B_1 = \sum m(1, 2, 7, 4, 13, 14, 11, 8) = \sum m(1, 2, 4, 7, 8, 11, 13, 14)$$

K-map for B_4

| | | G_2G_1 | 00 | 01 | 11 | 10 | |
|--|--|----------|----|----|----|----|---|
| | | G_4G_3 | 00 | 0 | 1 | 3 | 2 |
| | | 00 | 4 | 5 | 7 | 6 | |
| | | 01 | 12 | 13 | 15 | 14 | |
| | | 11 | 1 | 1 | 1 | 1 | |
| | | 10 | 8 | 9 | 11 | 10 | |
| | | 10 | 1 | 1 | 1 | 1 | |

$$B_4 = G_4$$

K-map for B_3

| | | G_2G_1 | 00 | 01 | 11 | 10 | |
|--|--|----------|----|----|----|----|---|
| | | G_4G_3 | 00 | 0 | 1 | 3 | 2 |
| | | 00 | 4 | 5 | 7 | 6 | |
| | | 01 | 1 | 1 | 1 | 1 | |
| | | 11 | 12 | 13 | 15 | 14 | |
| | | 10 | 8 | 9 | 11 | 10 | |
| | | 10 | 1 | 1 | 1 | 1 | |

$$B_3 = G_4 \oplus G_3$$

K-map for B_2

| | | G_2G_1 | 00 | 01 | 11 | 10 | |
|--|--|----------|----|----|----|----|---|
| | | G_4G_3 | 00 | 0 | 1 | 3 | 2 |
| | | 00 | 4 | 5 | 7 | 6 | |
| | | 01 | 1 | 1 | | | |
| | | 11 | 12 | 13 | 15 | 14 | |
| | | 10 | 8 | 9 | 11 | 10 | |
| | | 10 | 1 | 1 | | | |

$$B_2 = G_4 \oplus G_3 \oplus G_2$$

K-map for B_1

| | | G_2G_1 | 00 | 01 | 11 | 10 | |
|--|--|----------|----|----|----|----|---|
| | | G_4G_3 | 00 | 0 | 1 | 3 | 2 |
| | | 00 | 4 | 5 | 7 | 6 | |
| | | 01 | 1 | | 1 | | |
| | | 11 | 12 | 13 | 15 | 14 | |
| | | 10 | 8 | 9 | 11 | 10 | |
| | | 10 | 1 | | 1 | | |

$$B_1 = G_4 \oplus G_3 \oplus G_2 \oplus G_1$$

- The minimal expressions for the outputs obtained from the K-map are:

$$B_4 = G_4$$

$$B_3 = \bar{G}_4 G_3 + G_4 \bar{G}_3 = G_4 \oplus G_3$$

$$B_2 = \bar{G}_4 \bar{G}_3 G_2 + \bar{G}_4 G_3 \bar{G}_2 + G_4 \bar{G}_3 \bar{G}_2 + G_4 G_3 G_2$$

$$= \bar{G}_4 (\bar{G}_3 G_2 + G_3 \bar{G}_2) + G_4 (\bar{G}_3 \bar{G}_2 + G_3 G_2)$$

$$= \bar{G}_4 (G_3 \oplus G_2) + G_4 (\overline{G_3 \oplus G_2}) = G_4 \oplus G_3 \oplus G_2 = B_3 \oplus G_2$$

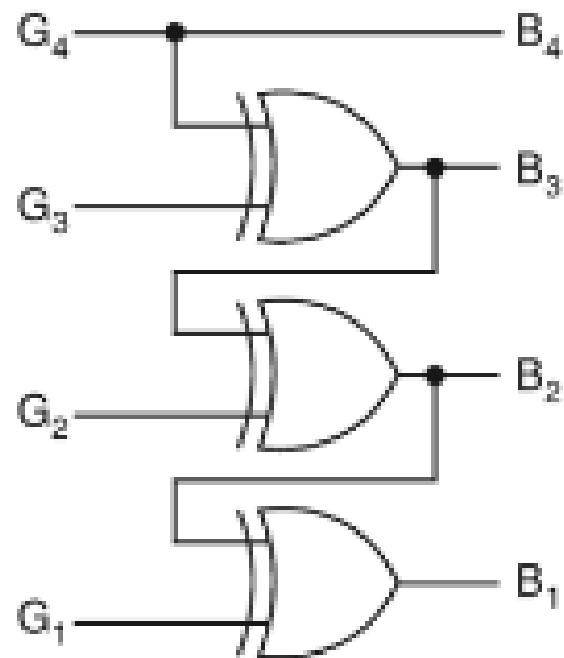
$$\begin{aligned}
B_1 &= \bar{G}_4 \bar{G}_3 \bar{G}_2 G_1 + \bar{G}_4 \bar{G}_3 G_2 \bar{G}_1 + \bar{G}_4 G_3 \bar{G}_2 \bar{G}_1 + \bar{G}_4 G_3 G_2 G_1 \\
&\quad + G_4 \bar{G}_3 \bar{G}_2 \bar{G}_1 + G_4 \bar{G}_3 G_2 G_1 + G_4 G_3 G_2 \bar{G}_1 + G_4 G_3 \bar{G}_2 G_1 \\
&= \bar{G}_4 \bar{G}_3 (\bar{G}_2 G_1 + G_2 \bar{G}_1) + \bar{G}_4 G_3 (\bar{G}_2 \bar{G}_1 + G_2 G_1) \\
&\quad + G_4 \bar{G}_3 (\bar{G}_2 \bar{G}_1 + G_2 G_1) + G_4 G_3 (\bar{G}_2 G_1 + G_2 \bar{G}_1) \\
&= \bar{G}_4 \bar{G}_3 (G_2 \oplus G_1) + \bar{G}_4 G_3 (\overline{G_2 \oplus G_1}) + G_4 \bar{G}_3 (\overline{G_2 \oplus G_1}) + G_4 G_3 (G_2 \oplus G_1) \\
&= (\bar{G}_4 \bar{G}_3 + G_4 G_3) (G_2 \oplus G_1) + (\bar{G}_4 G_3 + G_4 \bar{G}_3) (\overline{G_2 \oplus G_1}) \\
&= (\overline{G_4 \oplus G_3}) (G_2 \oplus G_1) + (G_4 \oplus G_3) (\overline{G_2 \oplus G_1}) \\
&= G_4 \oplus G_3 \oplus G_2 \oplus G_1 \\
&= B_2 \oplus G_1
\end{aligned}$$

$$B_4 = G_4$$

$$B_3 = G_4 \oplus G_3$$

$$B_2 = B_3 \oplus G_2$$

$$B_1 = B_2 \oplus G_1$$



Logic diagram

4-bit Gray-to-binary code converter.

DIGITAL ELECTRONICS

III Semester CSE & CST



Mr. Srikanta Patnaik Mr. Ajit Kumar Patro Mr. Ami Kumar Parida

DEPARTMENT OF ELECTRONICS AND COMMUNICATION ENGINEERING

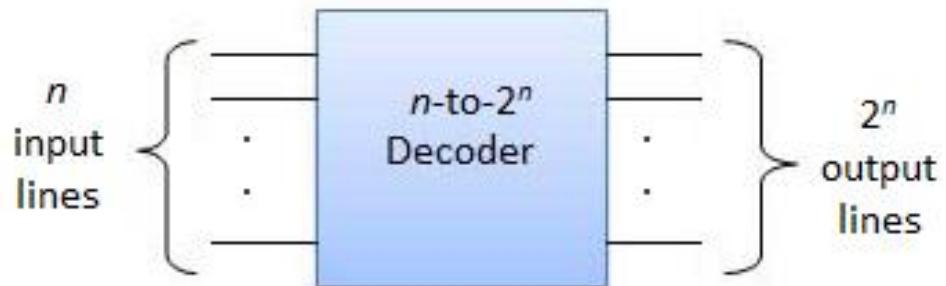
SCHOOL OF ENGINEERING & TECHNOLOGY

GIET UNIVERSITY, GUNUPUR, ODISHA

**Decoders; Encoders;
Multiplexers; De-multiplexers**

Decoders

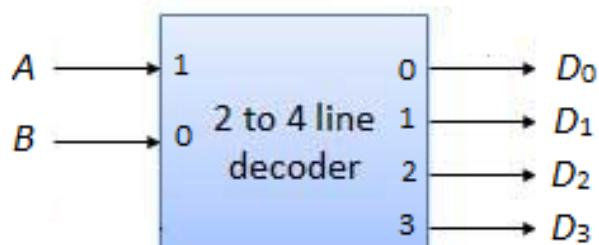
- A *decoder* is a combinational circuit that converts binary information from n input lines to a maximum of 2^n unique output lines.
- If the n -bit coded information has unused combinations, the decoder may have fewer than 2^n outputs.



- The decoders presented here are called n -to- m -line decoders, where $m \leq 2^n$.
- Their purpose is to generate the 2^n (or fewer) minterms of n input variables.
- Each combination of inputs will assert a unique output.
- The name *decoder* is also used in conjunction with other code converters, such as a BCD-to-seven-segment decoder.

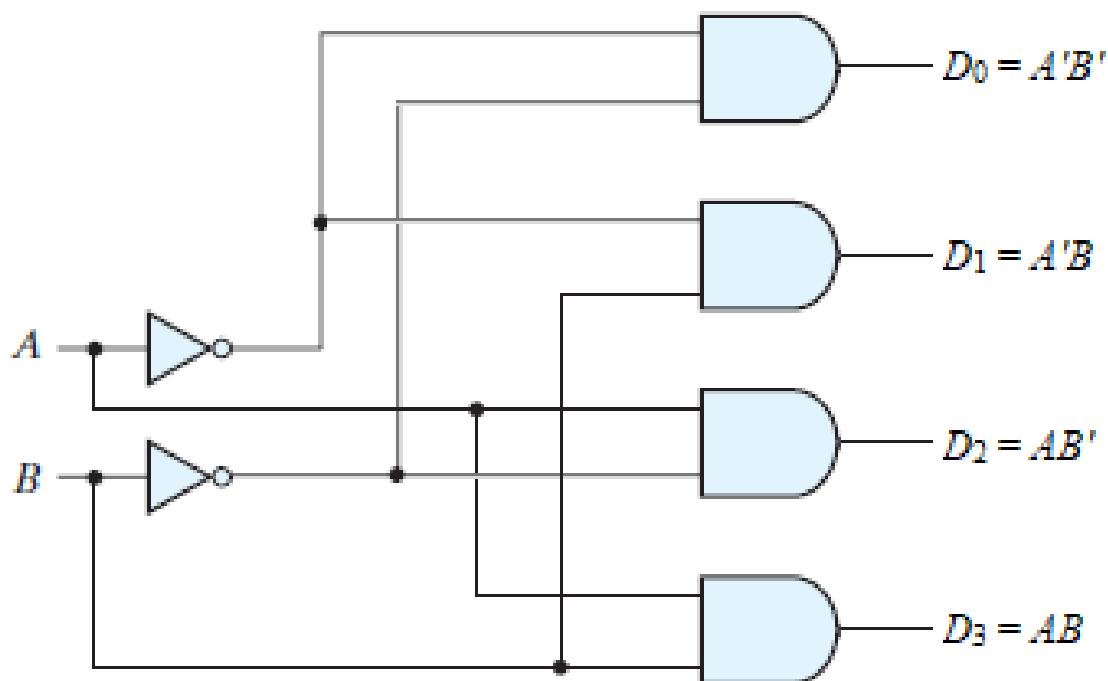
2-to-4-Line Decoder

- In 2-to-4-line decoder circuit, the two inputs are decoded into four outputs, each representing one of the minterms of the two input variables.
- The two inputs are represented by A and B and outputs are represented by D_0 , D_1 , D_2 and D_3 .



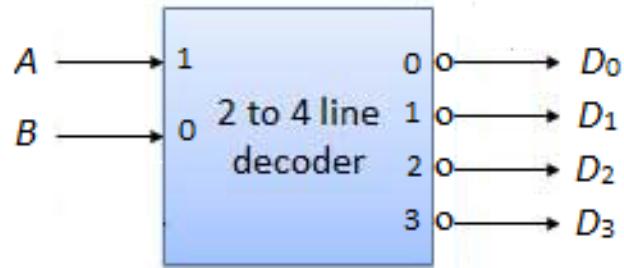
| Inputs | | Outputs | | | |
|--------|---|---------|-------|-------|-------|
| A | B | D_0 | D_1 | D_2 | D_3 |
| 0 | 0 | 1 | 0 | 0 | 0 |
| 0 | 1 | 0 | 1 | 0 | 0 |
| 1 | 0 | 0 | 0 | 1 | 0 |
| 1 | 1 | 0 | 0 | 0 | 1 |

Truth table

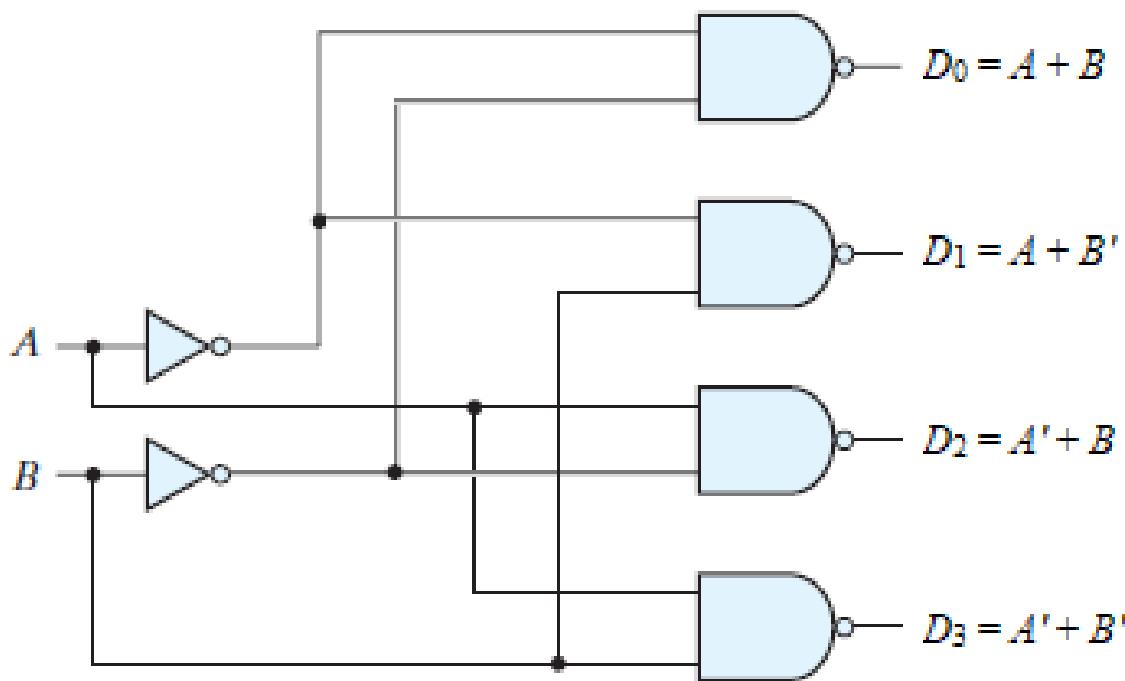


2-to-4-Line Active Low Decoder

- Some decoders are constructed with NAND gates. Since a NAND gate produces the AND operation with an inverted output, it becomes more economical to generate the decoder minterms in their complemented form.
- The use of NAND gates as the decoding element, results in an active - “LOW” output while the rest will be “HIGH”.



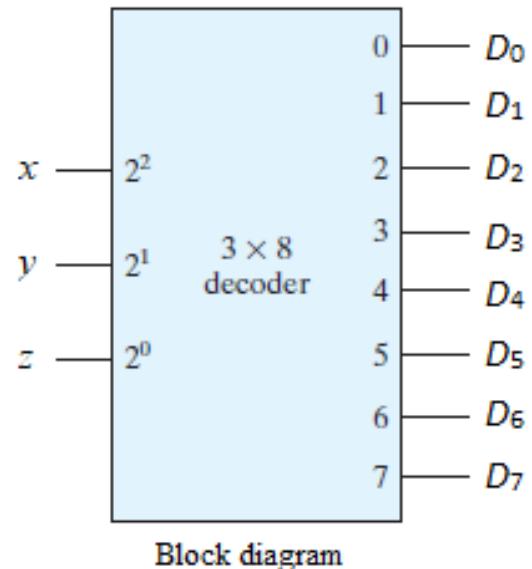
| Inputs | | Outputs | | | |
|--------|---|---------|-------|-------|-------|
| A | B | D_0 | D_1 | D_2 | D_3 |
| 0 | 0 | 0 | 1 | 1 | 1 |
| 0 | 1 | 1 | 0 | 1 | 1 |
| 1 | 0 | 1 | 1 | 0 | 1 |
| 1 | 1 | 1 | 1 | 1 | 0 |



Truth table

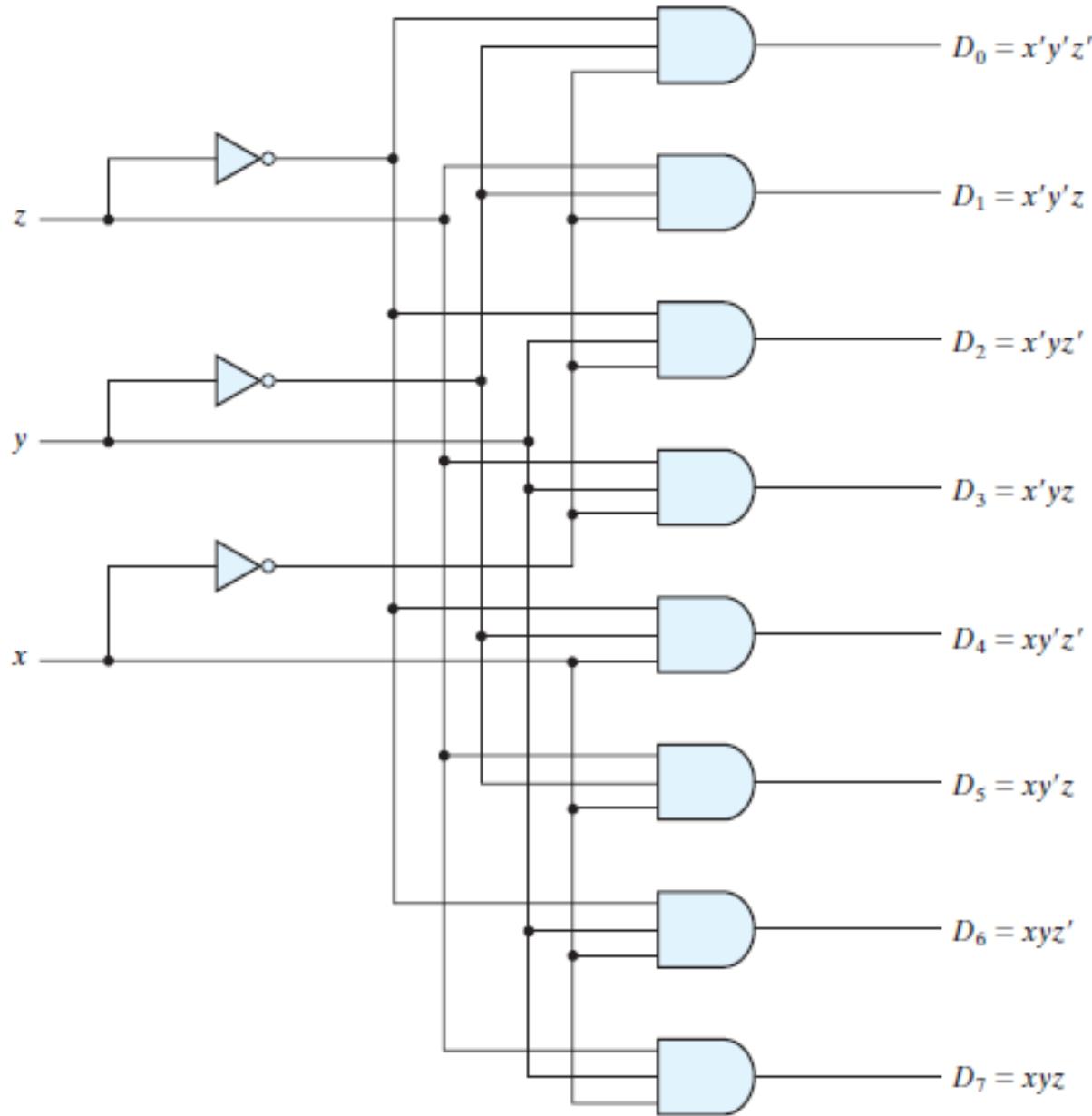
3-to-8-Line Decoder

- In 3-to-8-line decoder circuit, the three inputs are decoded into eight outputs, each representing one of the minterms of the three input variables.
- A particular application of this decoder is binary-to-octal conversion. The input variables represent a binary number, and the outputs represent the eight digits of a number in the octal number system.
- However, a 3-to-8-line decoder can be used for decoding any three-bit code to provide eight outputs, one for each element of the code.



Block diagram

Truth Table of a Three-to-Eight-Line Decoder

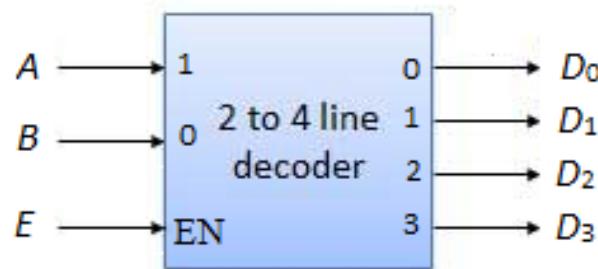


Three-to-eight-line decoder

Decoders with Enable Inputs

- Generally, decoders have the “enable” input.
- The enable input performs no logical operation, but is only responsible for making the decoder **active** or **inactive**.
- If the enable “ E ”
 - is zero, then all outputs are zero regardless of the input values.
 - is one, then the decoder performs its normal operation.

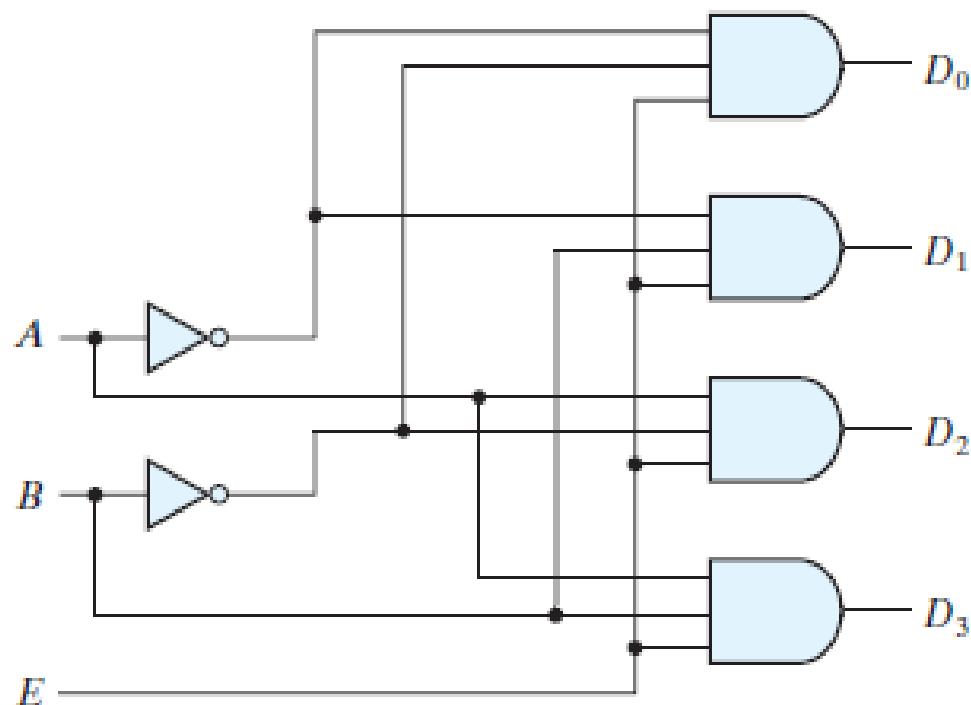
- Decoders with enable inputs can be connected together to form a larger decoder circuit.
- For example, consider the 2-to-4 line decoder with enable input.
- If enable E is zero, then all outputs of the decoder will be zeros, regardless of the values of A and B .
- However, if E is one, then the decoder will perform its normal operation, as shown in the truth table.



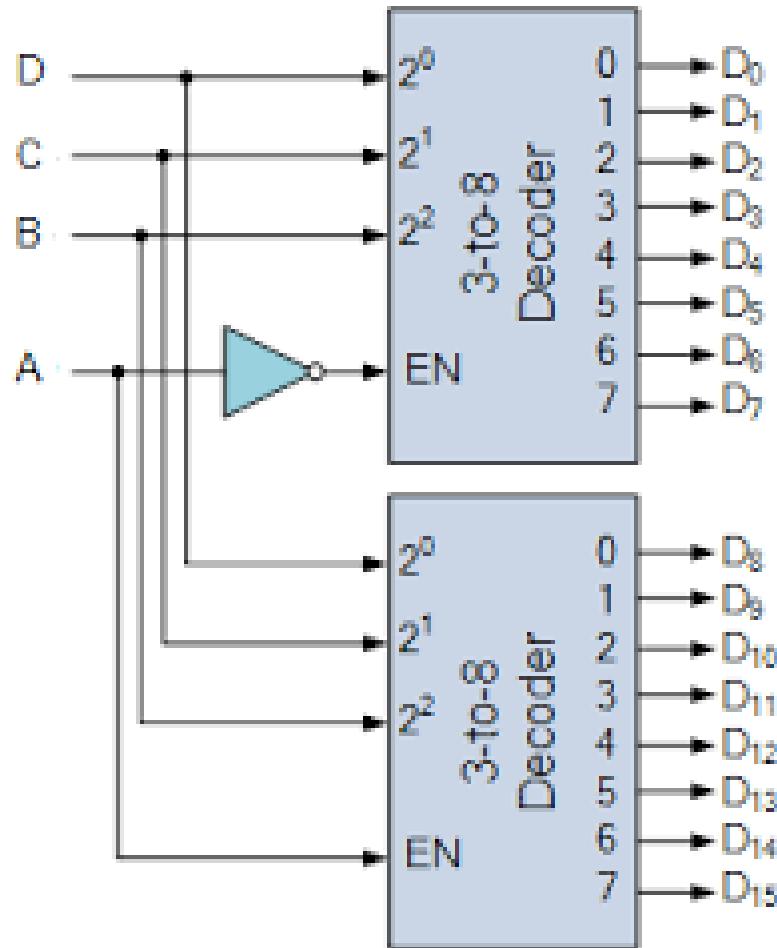
Inputs **Outputs**

| E | A | B | D ₀ | D ₁ | D ₂ | D ₃ |
|---|---|---|----------------|----------------|----------------|----------------|
| 0 | X | X | 0 | 0 | 0 | 0 |
| 1 | 0 | 0 | 1 | 0 | 0 | 0 |
| 1 | 0 | 1 | 0 | 1 | 0 | 0 |
| 1 | 1 | 0 | 0 | 0 | 1 | 0 |
| 1 | 1 | 1 | 0 | 0 | 0 | 1 |

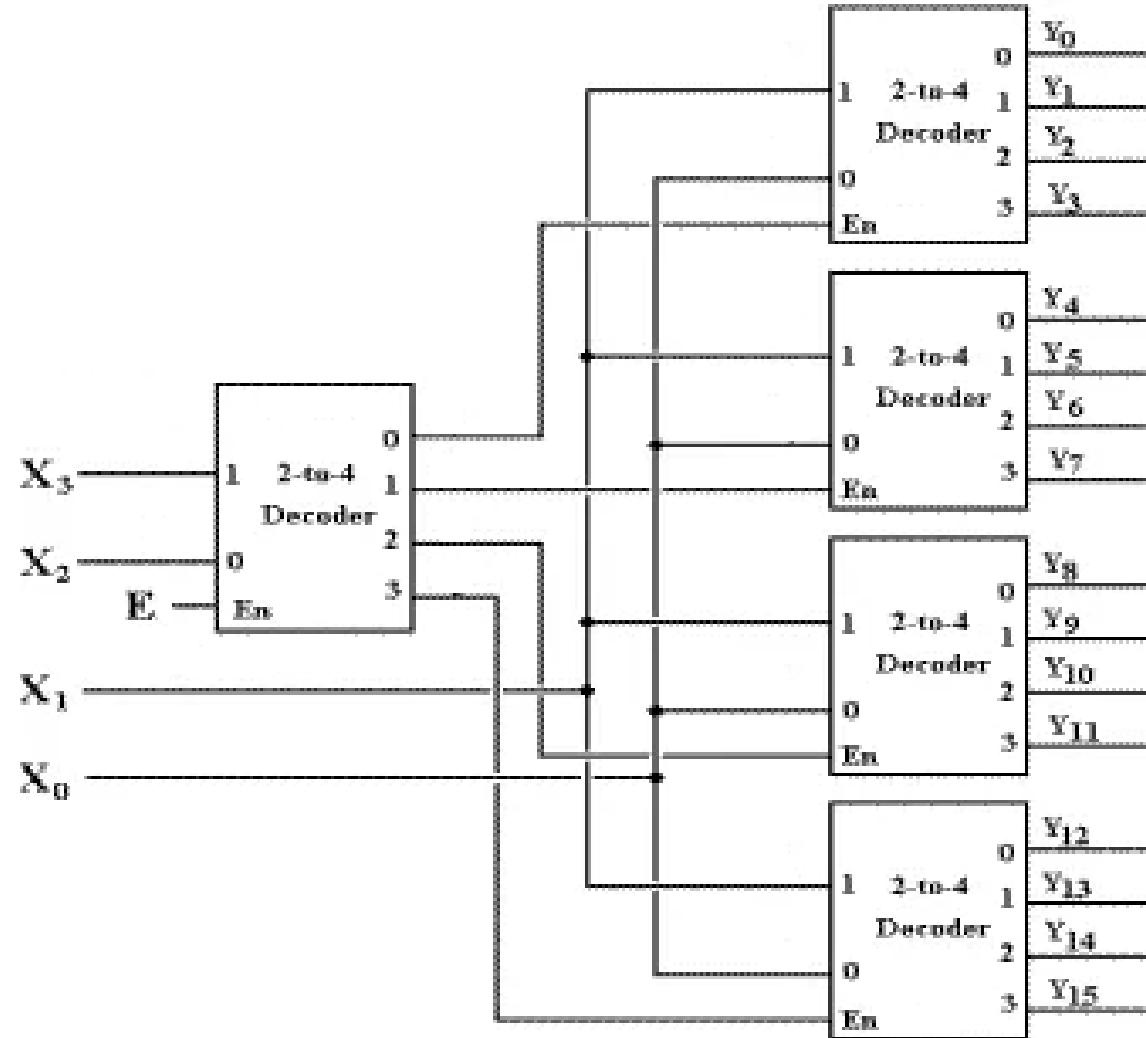
Truth table



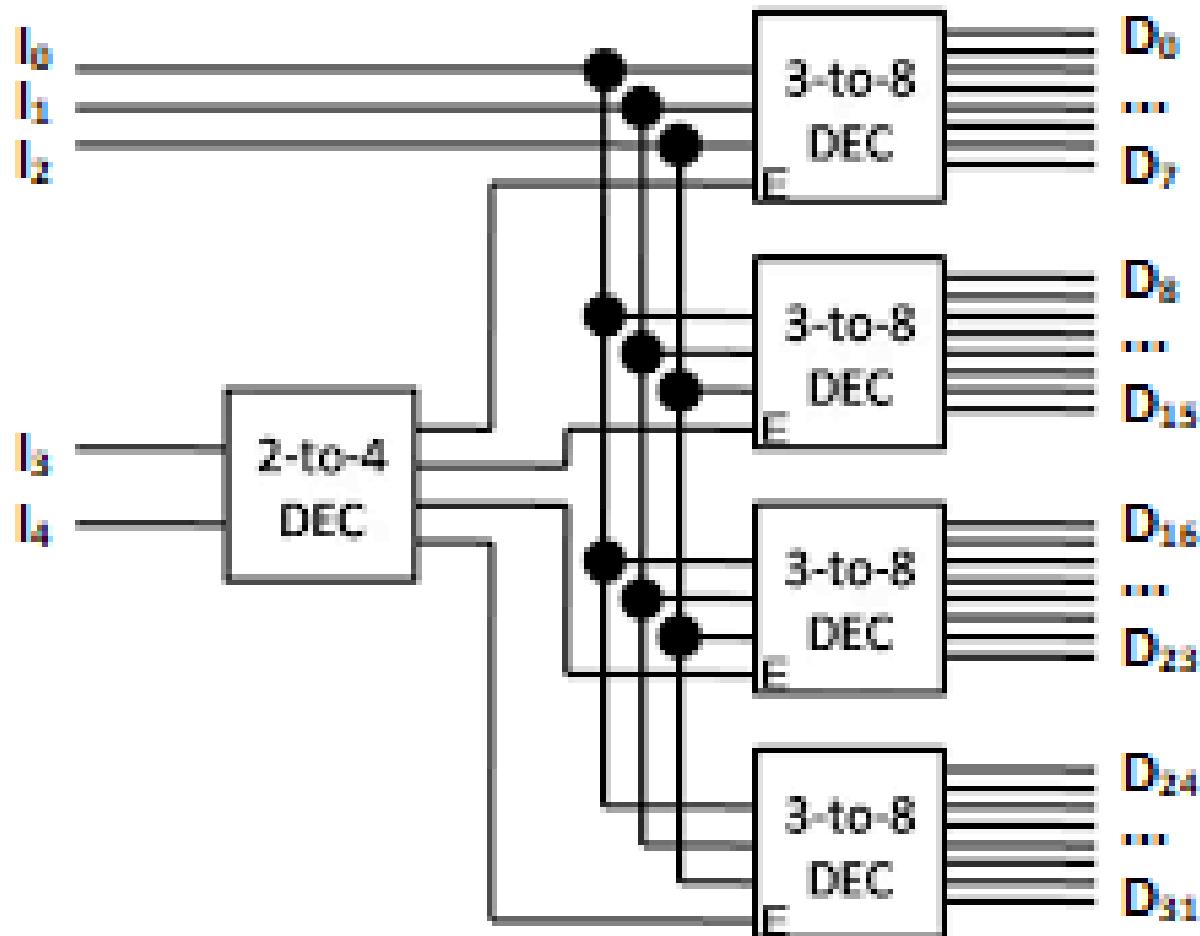
Example: Construct a 4-to-16-line decoder with two 3-to-8-line decoders with enable.



Example: Construct a 4-to-16-line decoder with five 2-to-4-line decoders with enable.



Example: Construct a 5-to-32-line decoder with four 3-to-8-line decoders with enable and a 2-to-4-line decoder.



Combinational Logic Implementation

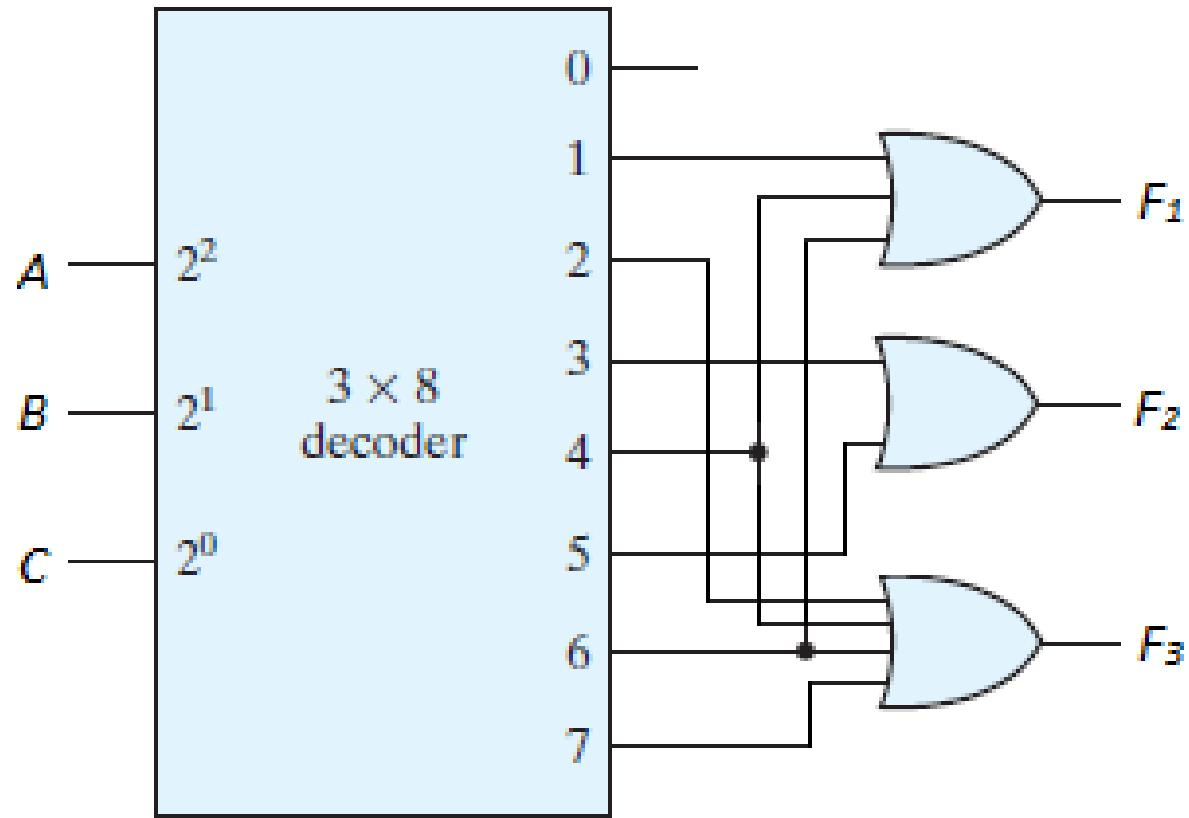
- Example: A combinational circuit is specified by the following three Boolean functions:

$$F_1(A, B, C) = \sum(1, 4, 6)$$

$$F_2(A, B, C) = \sum(3, 5)$$

$$F_3(A, B, C) = \sum(2, 4, 6, 7)$$

Implement the circuit with a decoder.



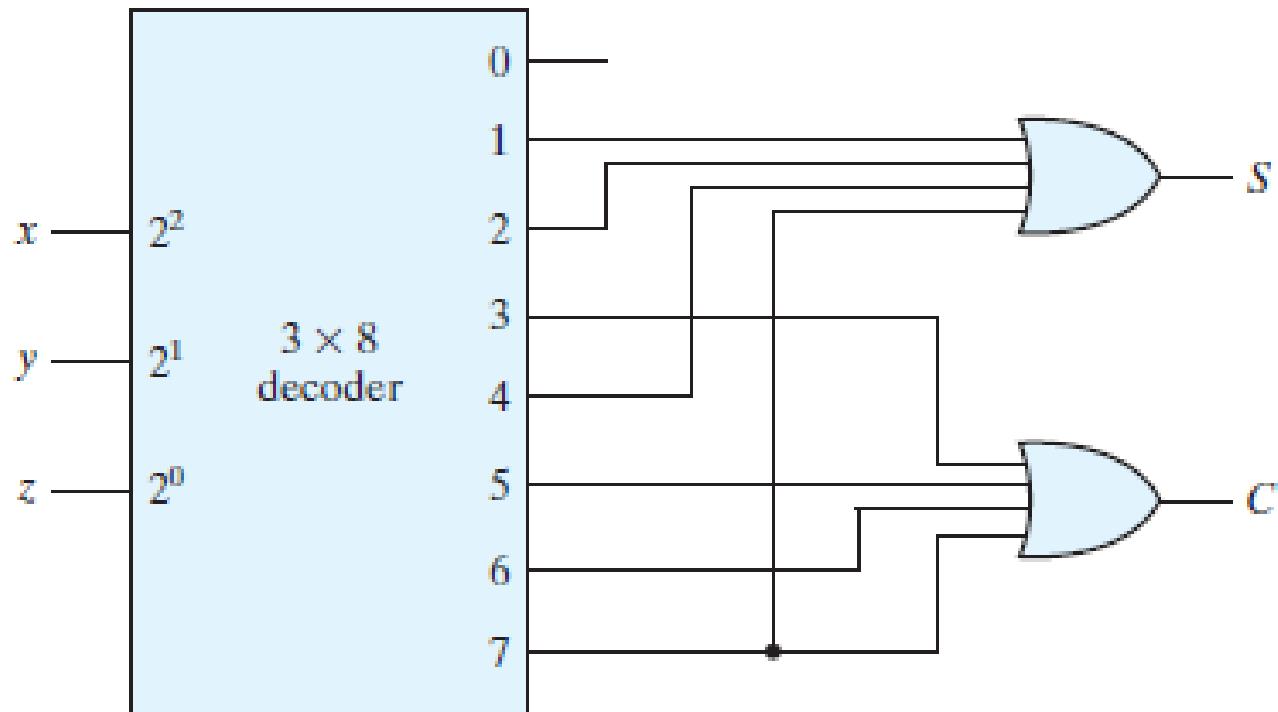
Example: Implement a full adder with a decoder and two OR gates.

- From the truth table of the full adder, we obtain the functions for the combinational circuit in sum-of-minterms form:

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

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

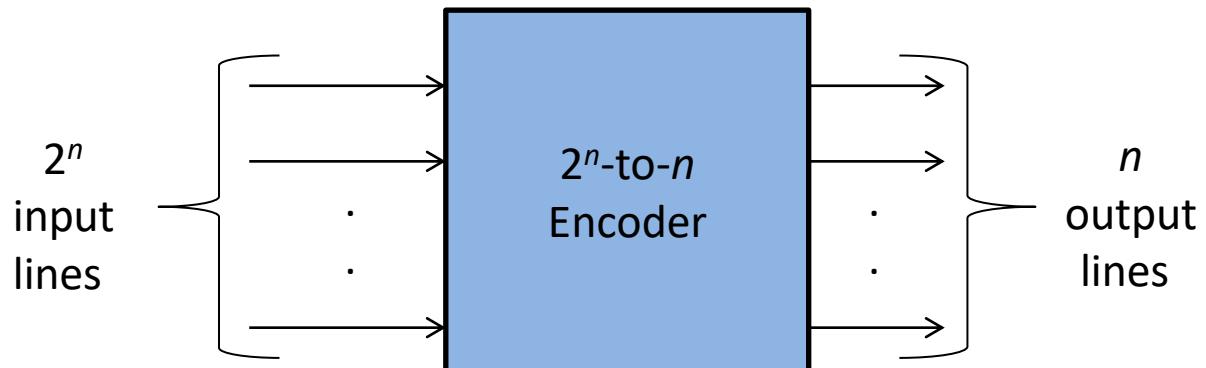
Since there are three inputs and a total of eight minterms, we need a 3-to-8-line decoder.



Implementation of a full adder with a decoder

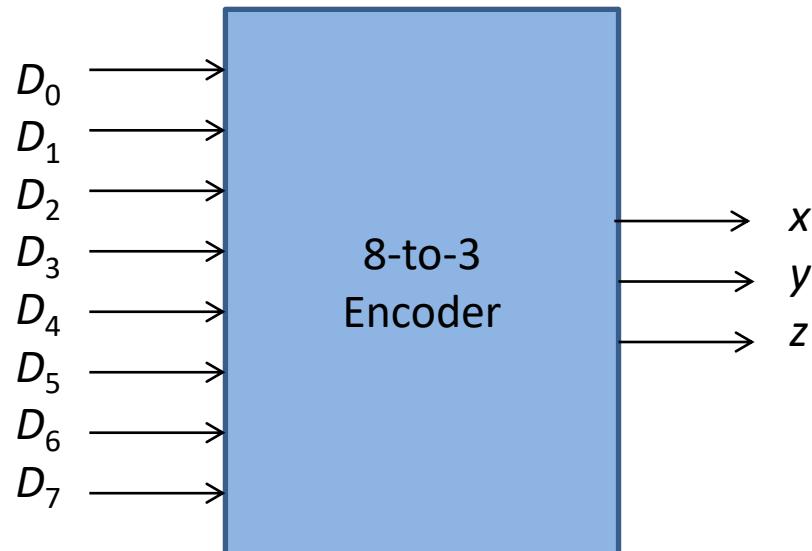
Encoders

- An encoder is a digital circuit that performs the inverse operation of a decoder.
- An encoder has 2^n (or fewer) input lines and n output lines.
- The output lines, as an aggregate, generate the binary code corresponding to the input value.



Octal-to-Binary Encoder

- It has eight inputs (one for each of the octal digits) and three outputs that generate the corresponding binary number.
- It is assumed that only one input has a value of 1 at any given time.



Truth Table of an Octal-to-Binary Encoder

| Inputs | | | | | | | | Outputs | | |
|--------|-------|-------|-------|-------|-------|-------|-------|---------|-----|-----|
| D_0 | D_1 | D_2 | D_3 | D_4 | D_5 | D_6 | D_7 | x | y | z |
| 1 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 |
| 0 | 1 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 1 |
| 0 | 0 | 1 | 0 | 0 | 0 | 0 | 0 | 0 | 1 | 0 |
| 0 | 0 | 0 | 1 | 0 | 0 | 0 | 0 | 0 | 1 | 1 |
| 0 | 0 | 0 | 0 | 1 | 0 | 0 | 0 | 1 | 0 | 0 |
| 0 | 0 | 0 | 0 | 0 | 1 | 0 | 0 | 1 | 0 | 1 |
| 0 | 0 | 0 | 0 | 0 | 0 | 1 | 0 | 1 | 1 | 0 |
| 0 | 0 | 0 | 0 | 0 | 0 | 0 | 1 | 1 | 1 | 1 |

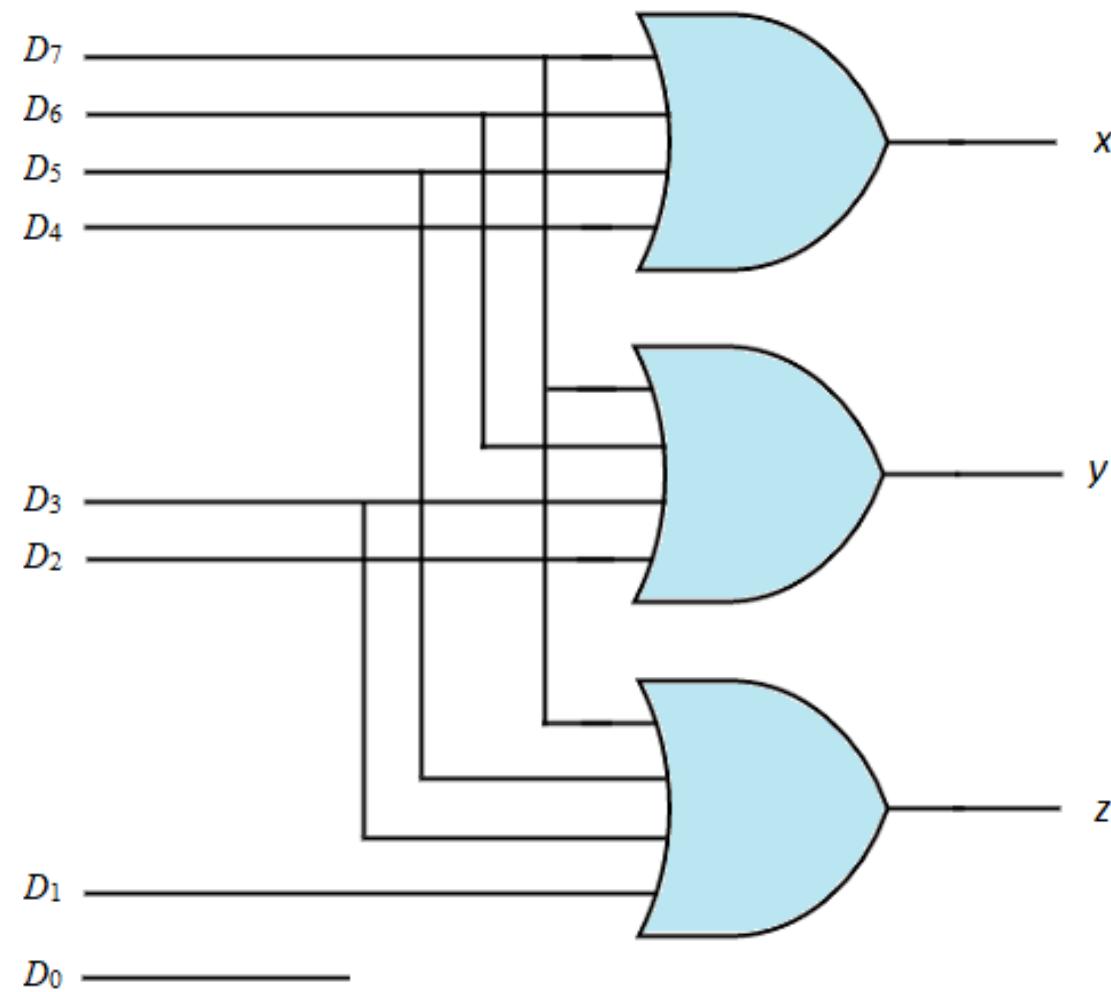
- From the truth table, output z is equal to 1 when the input octal digit is 1, 3, 5, or 7. Output y is 1 for octal digits 2, 3, 6, or 7, and output x is 1 for digits 4, 5, 6, or 7.
- These conditions can be expressed by the following Boolean output functions:

$$z = D_1 + D_3 + D_5 + D_7$$

$$y = D_2 + D_3 + D_6 + D_7$$

$$x = D_4 + D_5 + D_6 + D_7$$

- The encoder can be implemented with three OR gates.



- There are two ambiguities associated with the design of a octal-to-binary encoder:
 1. Only one input can be active at any given time. If two inputs are active simultaneously, the output produces an undefined combination. For example, if D_3 and D_6 are 1 simultaneously, the output of the encoder will be 111 because all three outputs are equal to 1. The output 111 does not represent either binary 3 or binary 6.
 - To resolve this ambiguity, encoder circuits must establish an input priority to ensure that only one input is encoded.

2. An output with all 0's is generated when all the inputs are 0; but this output is the same as when D_0 is equal to 1.
 - The discrepancy can be resolved by providing one more output to indicate whether at least one input is equal to 1.

Priority Encoder

- A **priority encoder** is an encoder circuit that includes the priority function.
- The operation of the priority encoder is such that if two or more inputs are equal to 1 at the same time, the input having the highest priority will take precedence.

Four-input Priority Encoder

- The truth table of a four-input priority encoder is given in Table.
- In addition to the two outputs x and y , the circuit has a third output designated by V ; this is a *valid* bit indicator that is set to 1 when one or more inputs are equal to 1.
- If all inputs are 0, there is no valid input and V is equal to 0.
- The other two outputs are not inspected when V equals 0 and are specified as don't-care conditions.

Truth Table of a Priority Encoder

| Inputs | | | | Outputs | | |
|--------|-------|-------|-------|---------|-----|-----|
| D_0 | D_1 | D_2 | D_3 | x | y | V |
| 0 | 0 | 0 | 0 | X | X | 0 |
| 1 | 0 | 0 | 0 | 0 | 0 | 1 |
| X | 1 | 0 | 0 | 0 | 1 | 1 |
| X | X | 1 | 0 | 1 | 0 | 1 |
| X | X | X | 1 | 1 | 1 | 1 |

| | | D_2D_3 | D_0D_1 | | |
|----|--|----------|----------|----------|----------|
| | | 00 | 01 | 11 | 10 |
| | | m_0 | m_1 | m_3 | m_2 |
| 00 | | X | 1 | 1 | 1 |
| 01 | | m_4 | m_5 | m_7 | m_6 |
| 11 | | m_{12} | m_{13} | m_{15} | m_{14} |
| 10 | | m_8 | m_9 | m_{11} | m_{10} |

$x = D_2 + D_3$

| | | D_2D_3 | D_0D_1 | | |
|----|--|----------|----------|----------|----------|
| | | 00 | 01 | 11 | 10 |
| | | m_0 | m_1 | m_3 | m_2 |
| 00 | | X | 1 | 1 | |
| 01 | | m_4 | m_5 | m_7 | m_6 |
| 11 | | m_{12} | m_{13} | m_{15} | m_{14} |
| 10 | | m_8 | m_9 | m_{11} | m_{10} |

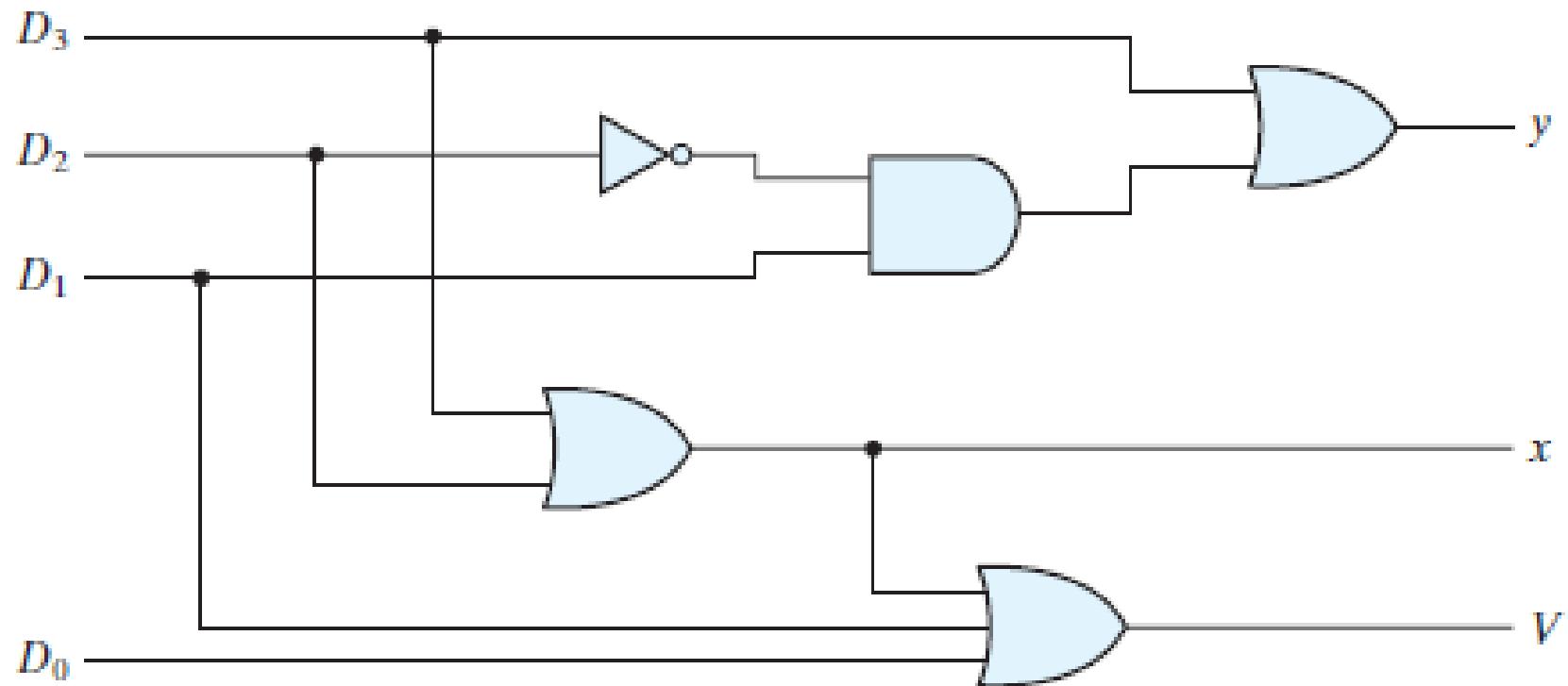
$y = D_3 + D_1D'_2$

- The simplified Boolean expressions for the priority encoder are obtained from the maps.
- The condition for output V is an OR function of all the input variables.
- The priority encoder is implemented in Fig. according to the following Boolean functions:

$$x = D_2 + D_3$$

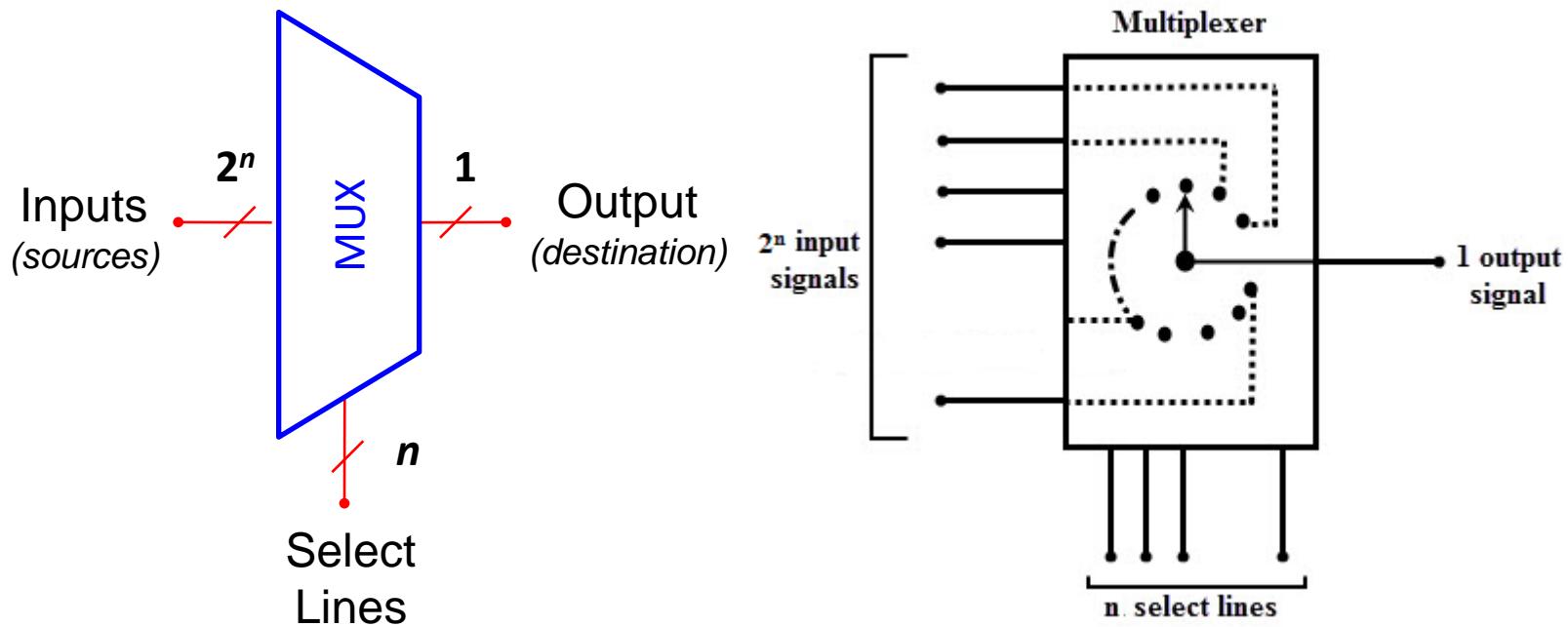
$$y = D_3 + D_1D_2'$$

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



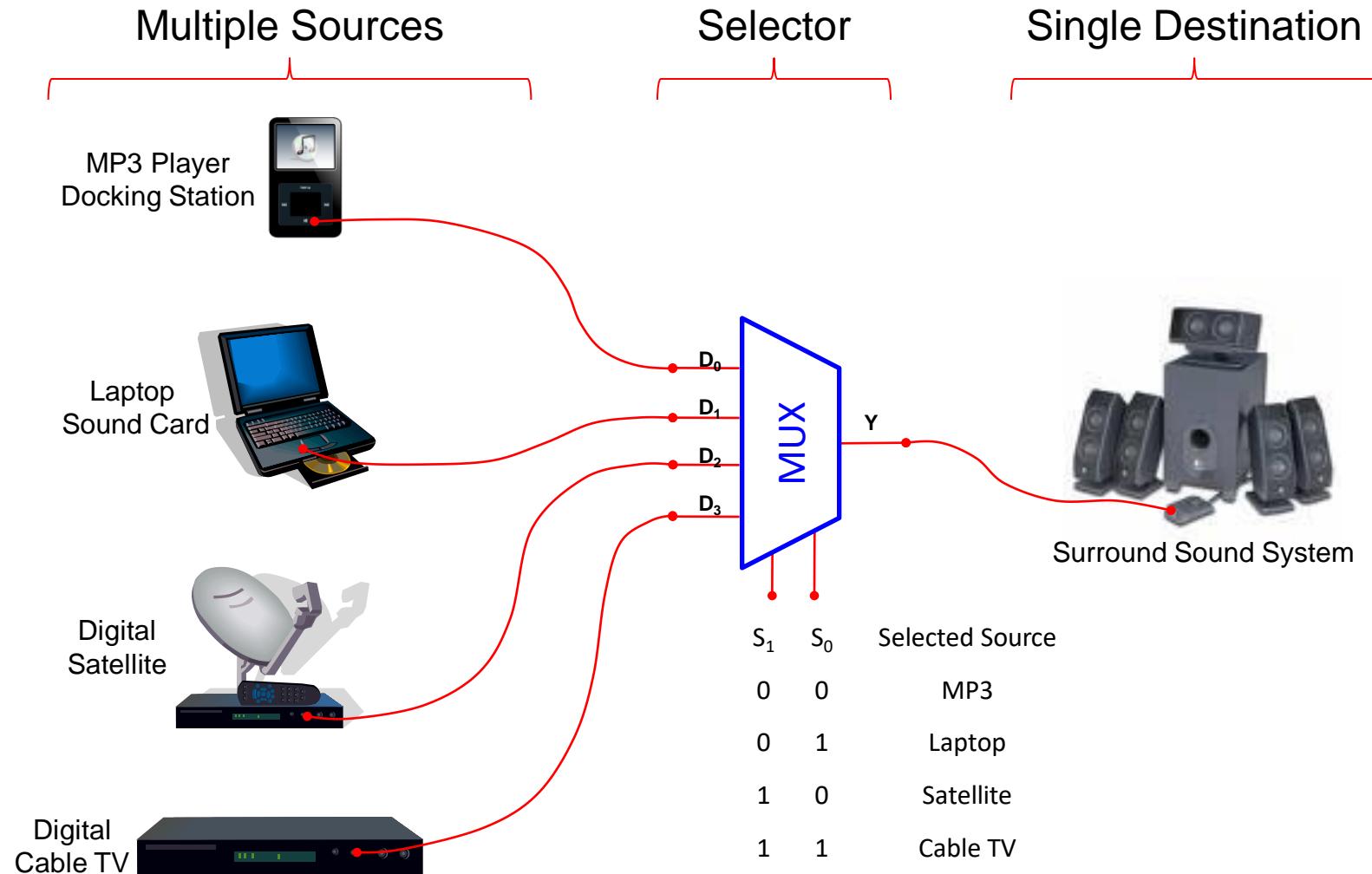
Multiplexers

- A **multiplexer** is a combinational circuit that selects binary information from one of many input lines and directs it to a single output line.
- The selection of a particular input line is controlled by a set of selection lines.
- Normally, there are 2^n input lines and n selection lines whose bit combinations determine which input is selected.



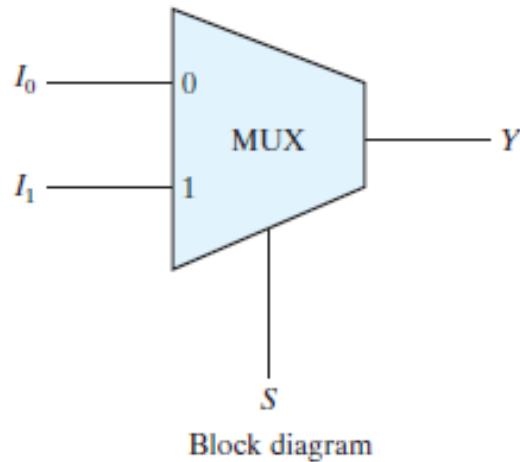
- A multiplexer is also called a *data selector*, since it selects one of many inputs and steers the binary information to the output line.

Typical Application of a MUX



Two-to-One-line Multiplexer

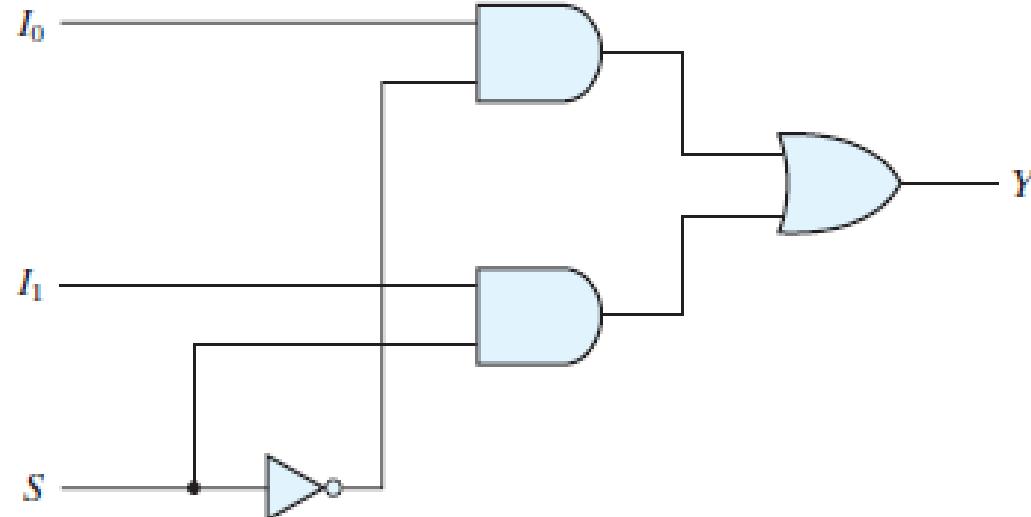
- A two-to-one-line multiplexer connects one of two 1-bit sources to a common destination, as shown in Fig.
- The circuit has two data input lines, one output line, and one selection line S .



| S | Y |
|-----|-------|
| 0 | I_0 |
| 1 | I_1 |

$$Y = I_0 \bar{S} + I_1 S$$

Function table

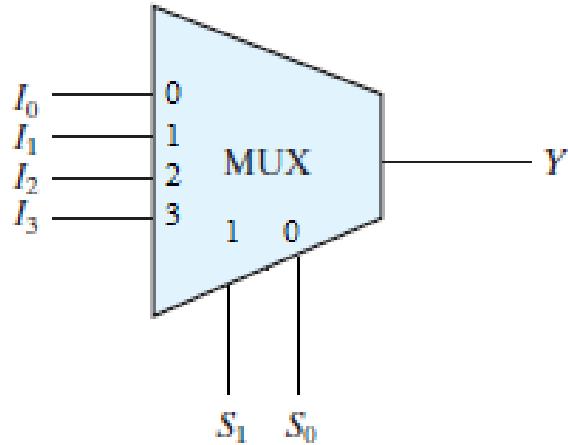


Logic diagram

- When $S = 0$, the upper AND gate is enabled and I_0 has a path to the output.
- When $S = 1$, the lower AND gate is enabled and I_1 has a path to the output.
- The multiplexer acts like an electronic switch that selects one of two sources.

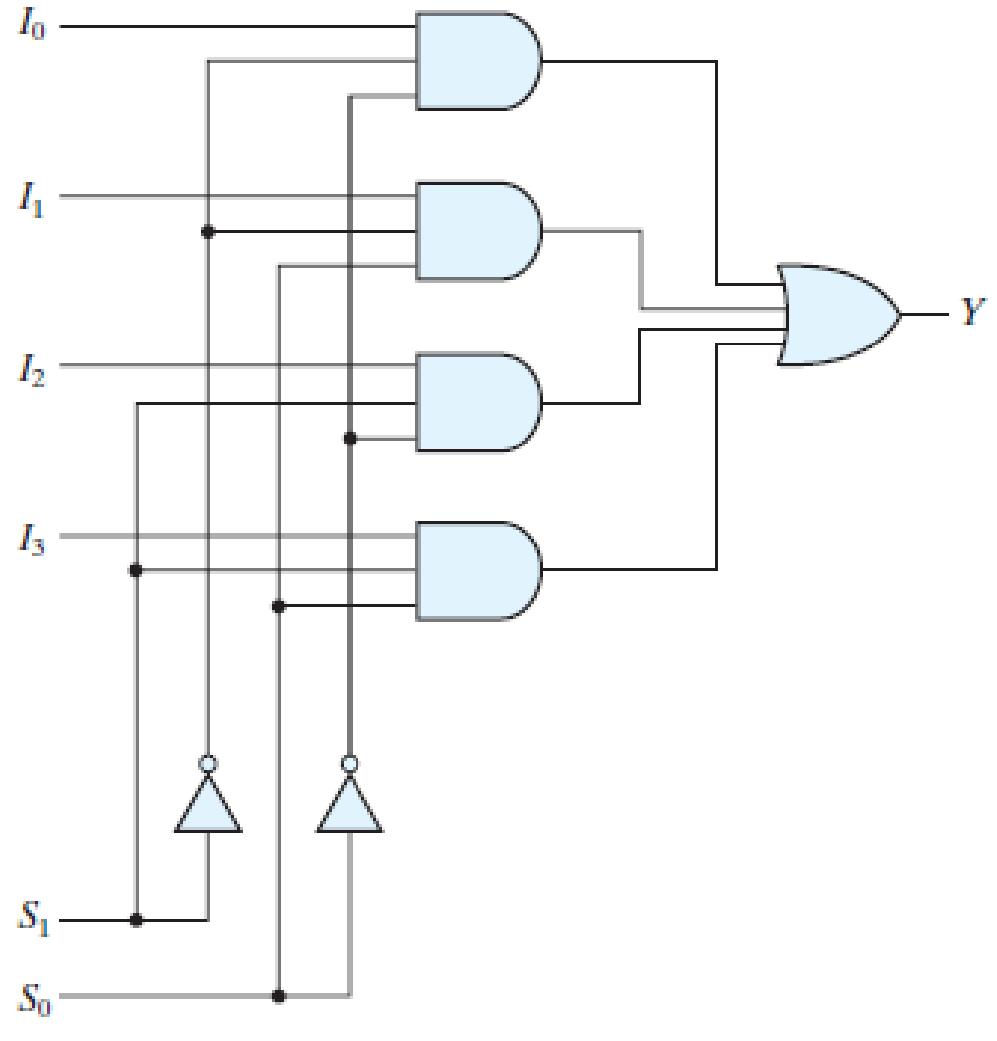
Four-to-One-line Multiplexer

- A four-to-one-line multiplexer is shown in Fig.
- Each of the four inputs, I_0 through I_3 , is applied to one input of an AND gate.
- Selection lines S_1 and S_0 are decoded to select a particular AND gate.
- The outputs of the AND gates are applied to a single OR gate that provides the one-line output.
- The function table lists the input that is passed to the output for each combination of the binary selection values.



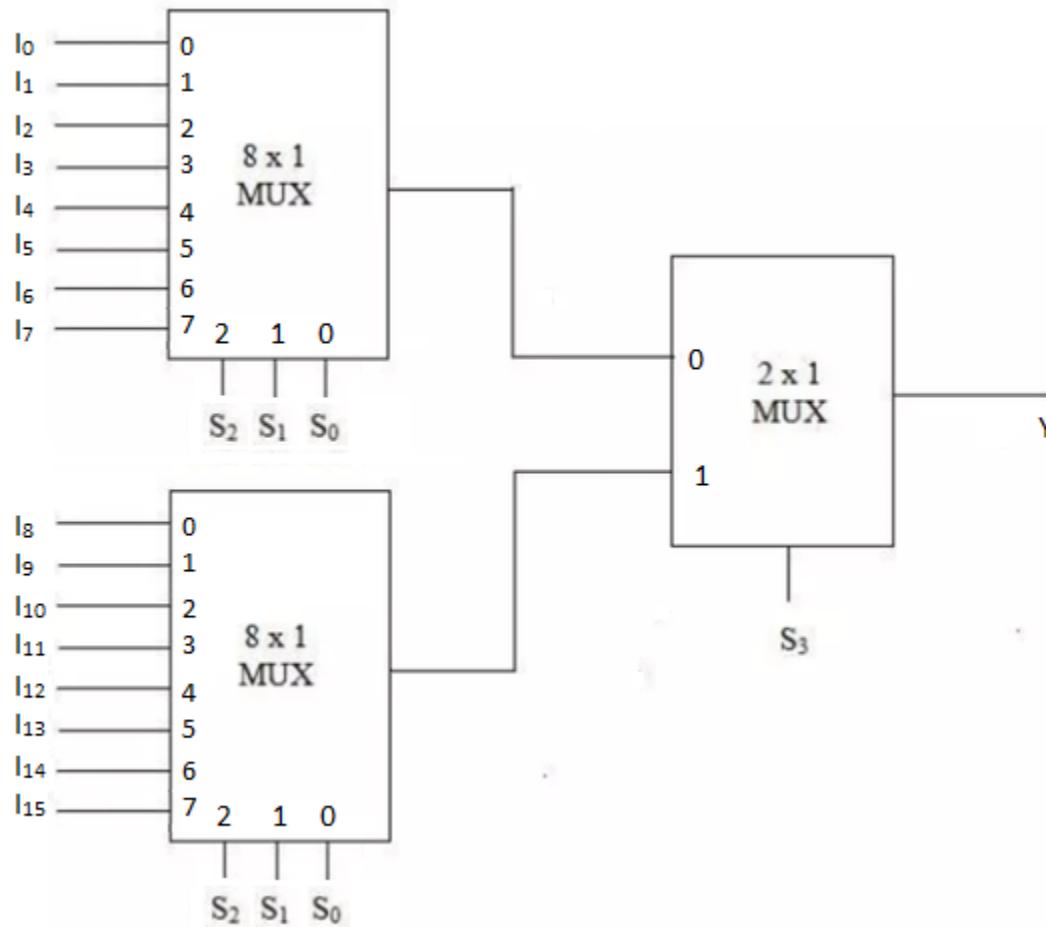
| S_1 | S_0 | Y |
|-------|-------|-------|
| 0 | 0 | I_0 |
| 0 | 1 | I_1 |
| 1 | 0 | I_2 |
| 1 | 1 | I_3 |

Function table

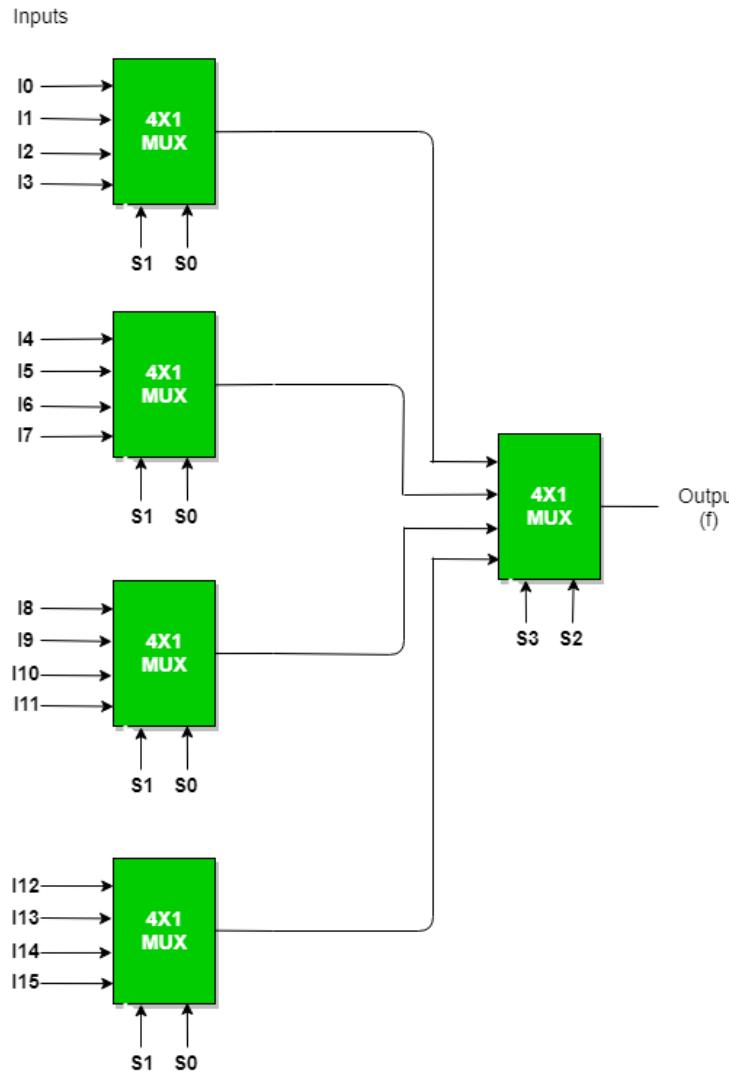


$$Y = I_0 \bar{S}_1 \bar{S}_0 + I_1 \bar{S}_1 S_0 + I_2 S_1 \bar{S}_0 + I_3 S_1 S_0$$

Example: Construct a 16×1 multiplexer with two 8×1 and one 2×1 multiplexers.



Example: Construct a 16×1 multiplexer with five 4×1 multiplexers.

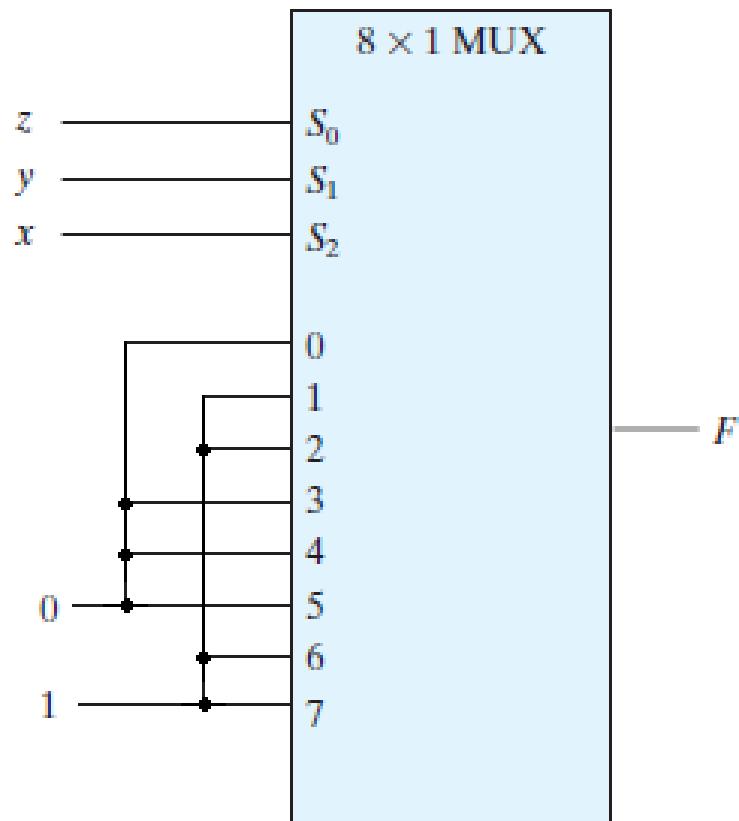


Boolean Function Implementation

- The minterms of a function are generated in a multiplexer by the circuit associated with the selection inputs.
- The individual minterms can be selected by the data inputs, thereby providing a method of implementing a Boolean function.

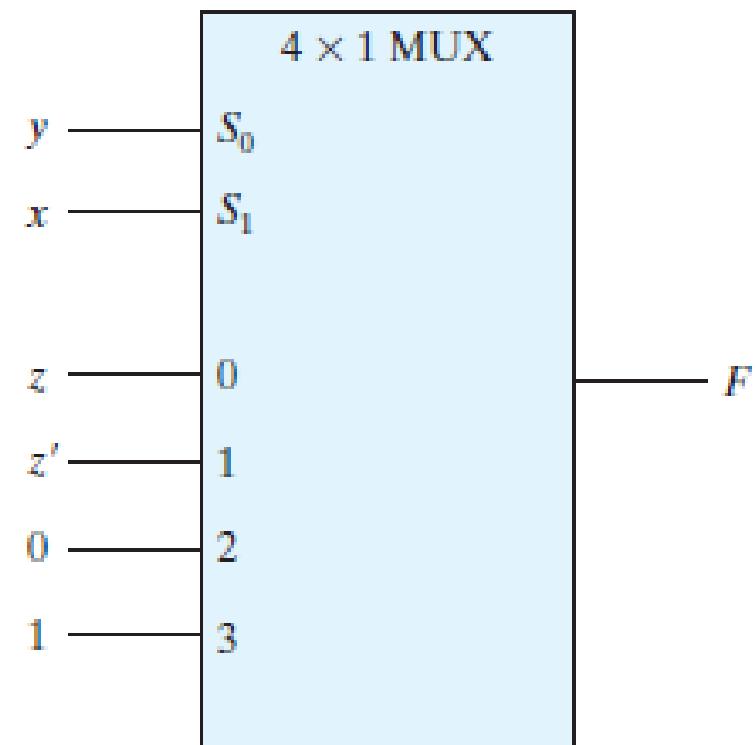
Example: Implement the following Boolean function with a multiplexer.

$$F(x, y, z) = \sum(1, 2, 6, 7)$$



$$F(x, y, z) = \sum(1, 2, 6, 7)$$

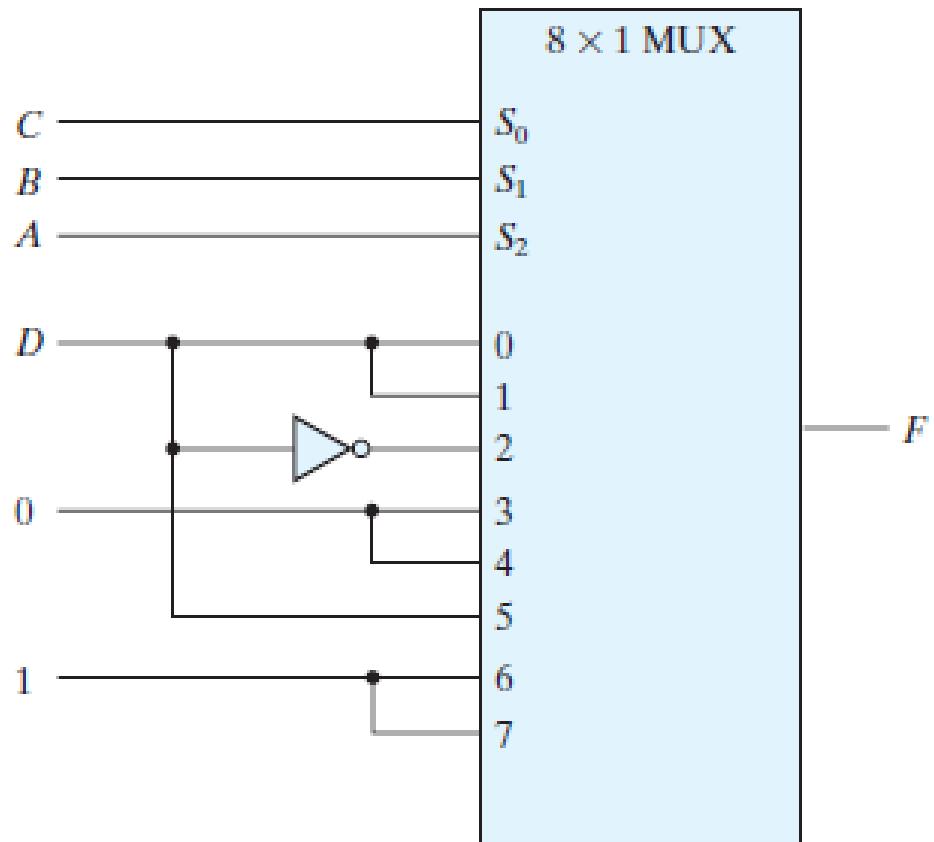
| x | y | z | F | |
|-----|-----|-----|-----|----------|
| 0 | 0 | 0 | 0 | $F = z$ |
| 0 | 0 | 1 | 1 | |
| 0 | 1 | 0 | 1 | $F = z'$ |
| 0 | 1 | 1 | 0 | |
| 1 | 0 | 0 | 0 | $F = 0$ |
| 1 | 0 | 1 | 0 | |
| 1 | 1 | 0 | 1 | $F = 1$ |
| 1 | 1 | 1 | 1 | |



Example: Implement the following Boolean function with a multiplexer.

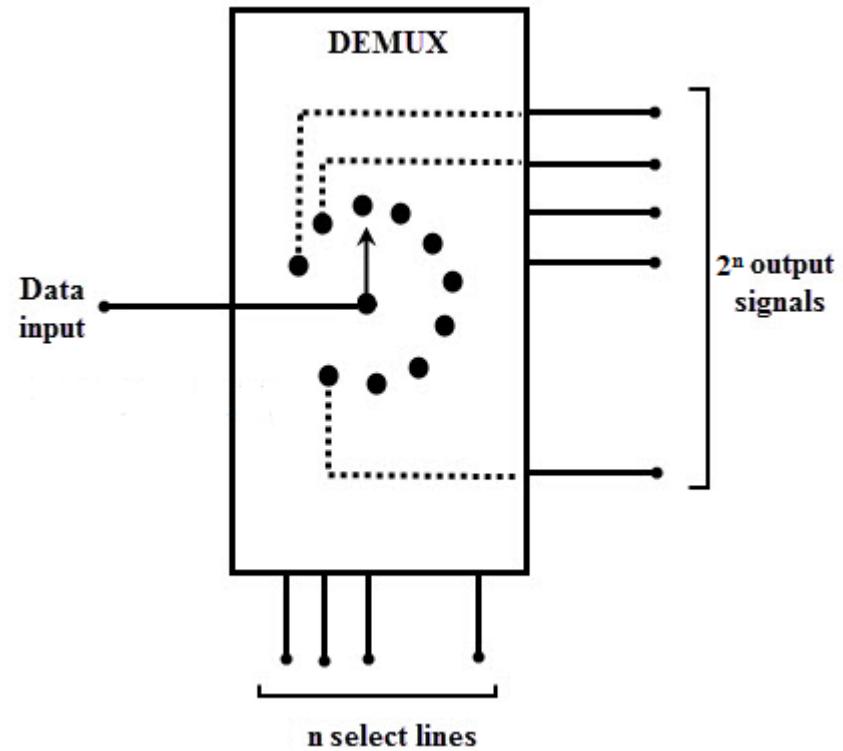
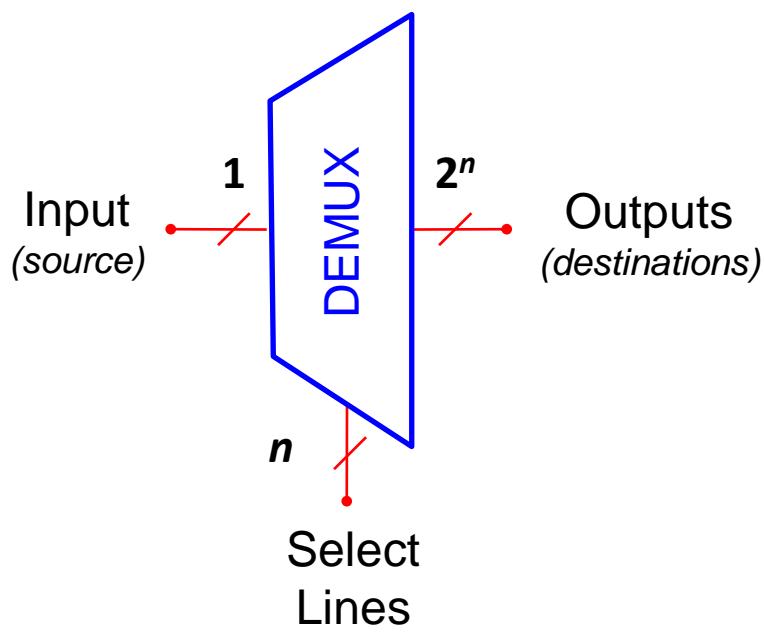
$$F(A, B, C, D) = \sum(1, 3, 4, 11, 12, 13, 14, 15)$$

| 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 |



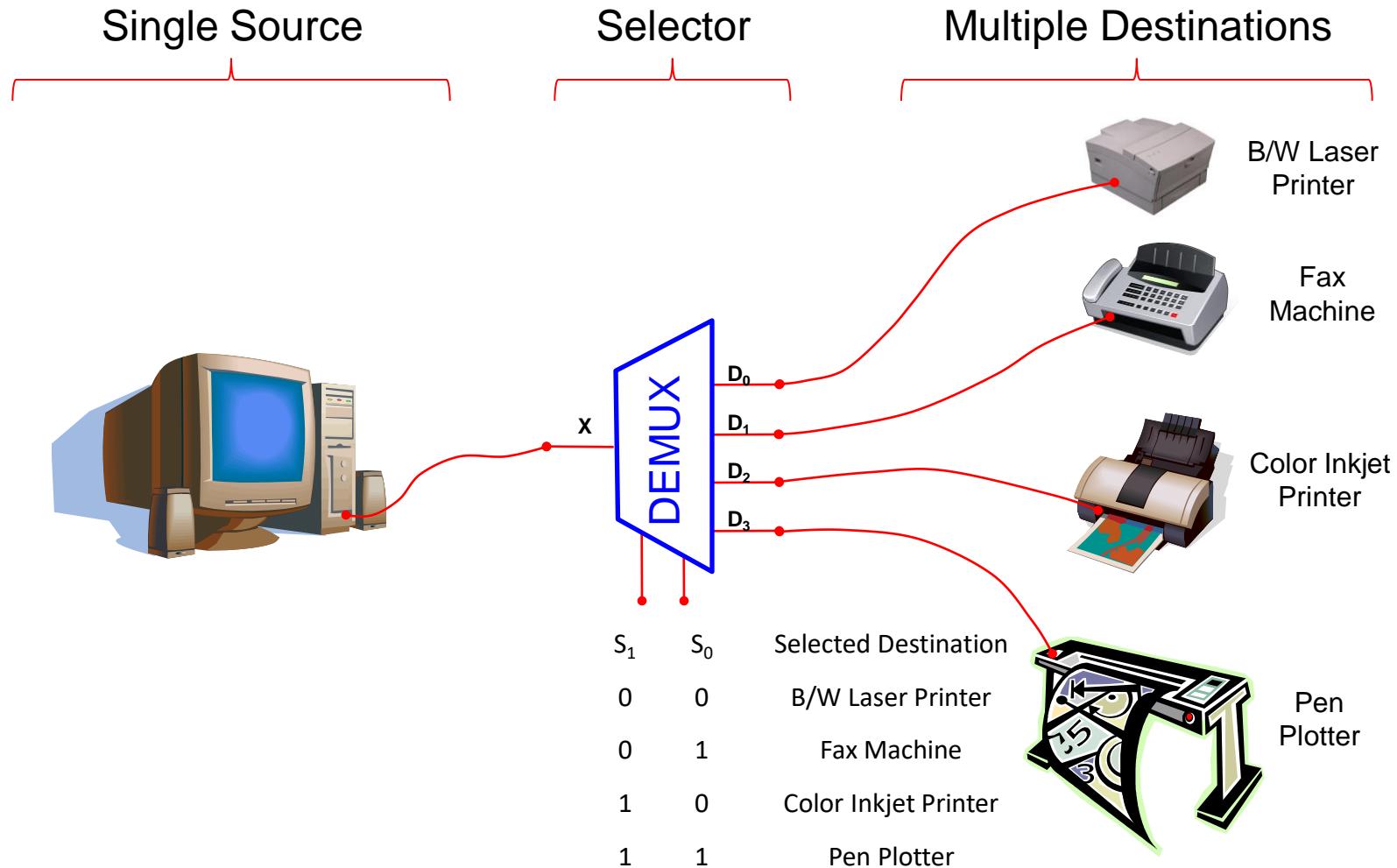
De-multiplexers

- A de-multiplexer performs the reverse operation of multiplexer; it takes single input and distributes it over several outputs.
- A decoder with enable input can function as a *de-multiplexer* - a circuit that receives information from a single line and directs it to one of 2^n possible output lines.
- The selection of a specific output is controlled by the bit combination of n selection lines.



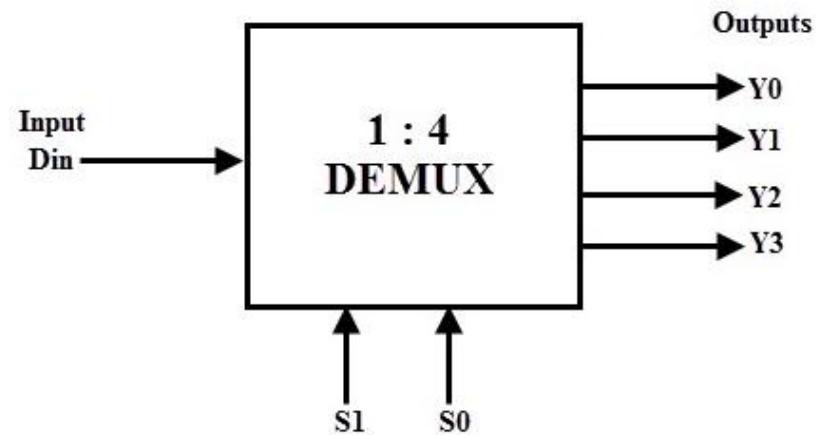
- A de-multiplexer is also called a *data distributor*, since it transmits the same data to different destinations.

Typical Application of a DEMUX



1-to-4 Line De-multiplexer

- A 1-to-4 line de-multiplexer has a single input (D), two selection lines (S_1 and S_0) and four outputs (Y_0 to Y_3).



- The input data goes to any one of the four outputs at a given time for a particular combination of select lines.

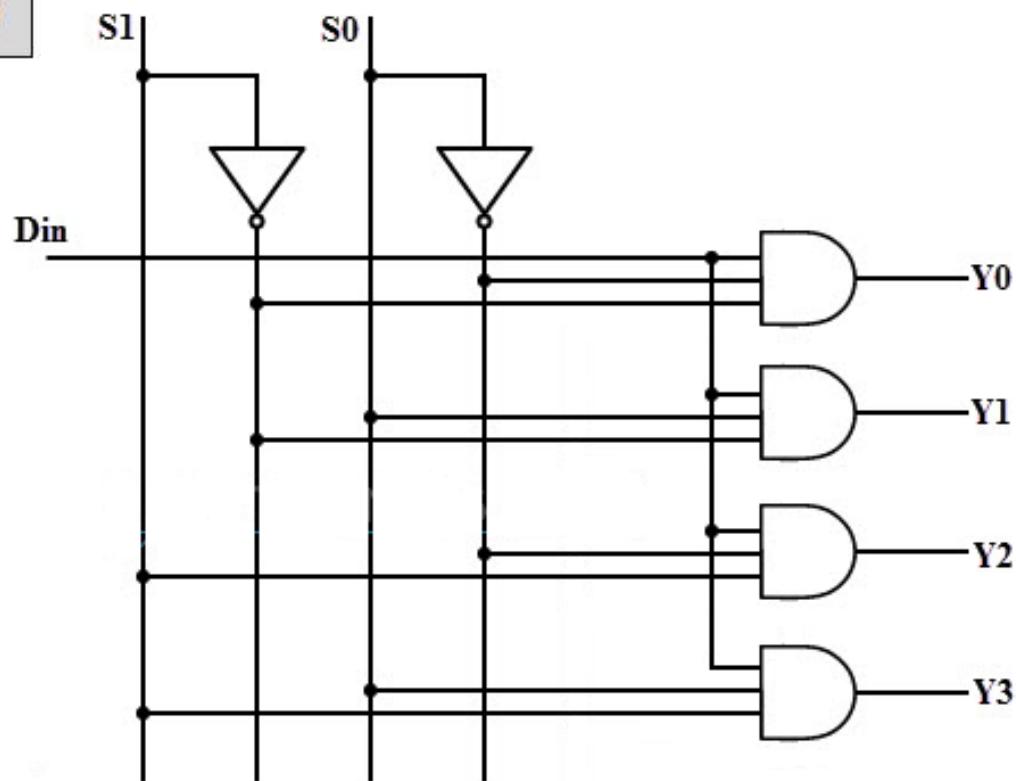
| Data Input | Select Inputs | | Outputs | | | |
|------------|----------------|----------------|----------------|----------------|----------------|----------------|
| D | S ₁ | S ₀ | Y ₃ | Y ₂ | Y ₁ | Y ₀ |
| D | 0 | 0 | 0 | 0 | 0 | D |
| D | 0 | 1 | 0 | 0 | D | 0 |
| D | 1 | 0 | 0 | D | 0 | 0 |
| D | 1 | 1 | D | 0 | 0 | 0 |

$$Y_0 = \overline{S}_1 \overline{S}_0 D$$

$$Y_1 = \overline{S}_1 S_0 D$$

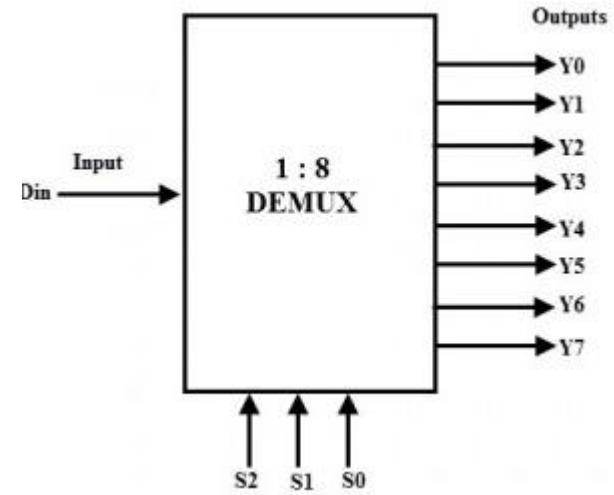
$$Y_2 = S_1 \overline{S}_0 D$$

$$Y_3 = S_1 S_0 D$$



1-to-8 Line De-multiplexer

- A 1-to-8 de-multiplexer consists of single input D_{in} , three select inputs S_2 , S_1 and S_0 and eight outputs from Y_0 to Y_7 .



- It distributes one input line to one of 8 output lines depending on the combination of select inputs.

| Data Input | Select Inputs | | | Outputs | | | | | | | | |
|------------|---------------|----------------|----------------|----------------|----------------|----------------|----------------|----------------|----------------|----------------|----------------|----------------|
| | D | S ₂ | S ₁ | S ₀ | Y ₇ | Y ₆ | Y ₅ | Y ₄ | Y ₃ | Y ₂ | Y ₁ | Y ₀ |
| D | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | D |
| D | 0 | 0 | 1 | 0 | 0 | 0 | 0 | 0 | 0 | D | 0 | 0 |
| D | 0 | 1 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | D | 0 | 0 |
| D | 0 | 1 | 1 | 0 | 0 | 0 | 0 | D | 0 | 0 | 0 | 0 |
| D | 1 | 0 | 0 | 0 | 0 | 0 | D | 0 | 0 | 0 | 0 | 0 |
| D | 1 | 0 | 1 | 0 | 0 | D | 0 | 0 | 0 | 0 | 0 | 0 |
| D | 1 | 1 | 0 | 0 | D | 0 | 0 | 0 | 0 | 0 | 0 | 0 |
| D | 1 | 1 | 1 | D | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 |

$$Y_0 = \overline{S}_2 \overline{S}_1 \overline{S}_0 D$$

$$Y_1 = \overline{S}_2 \overline{S}_1 S_0 D$$

$$Y_2 = \overline{S}_2 S_1 \overline{S}_0 D$$

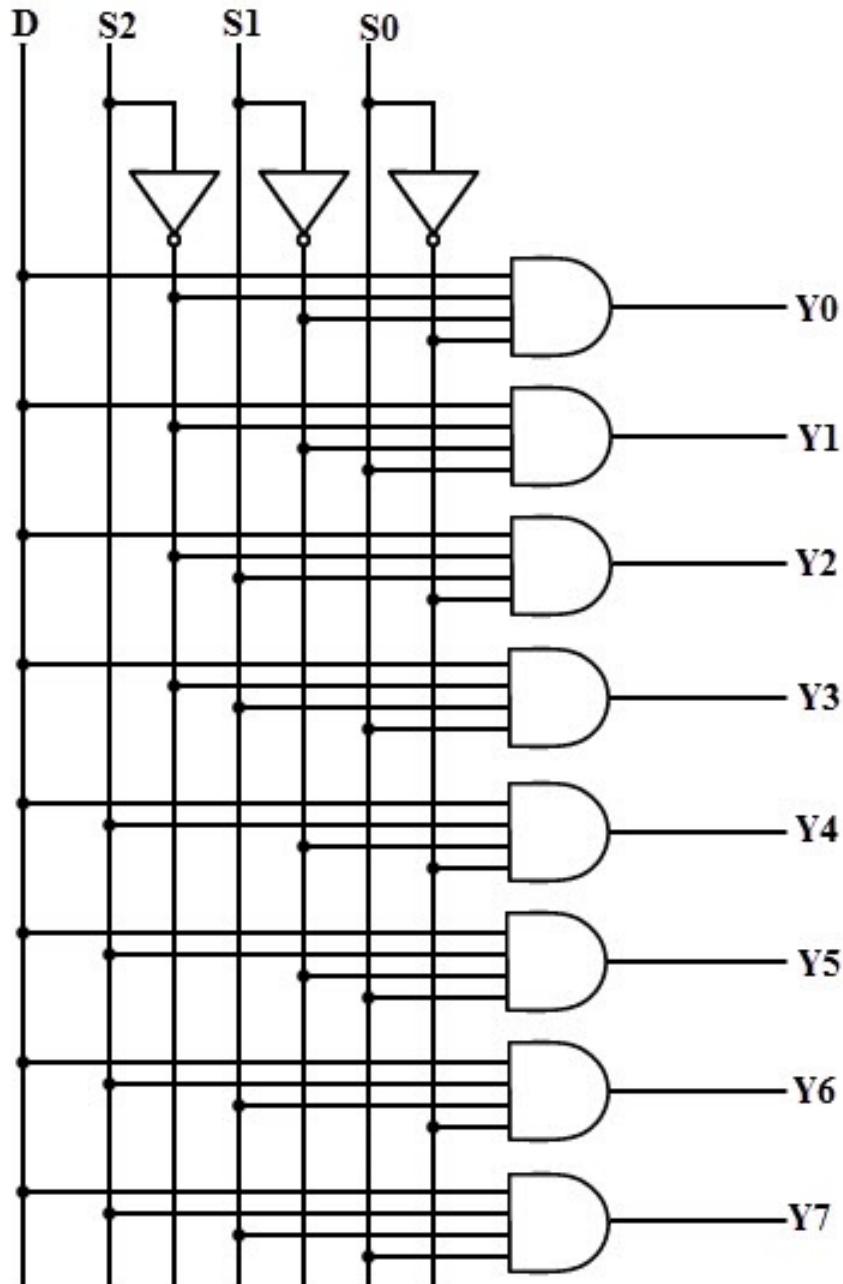
$$Y_3 = \overline{S}_2 S_1 S_0 D$$

$$Y_4 = S_2 \overline{S}_1 \overline{S}_0 D$$

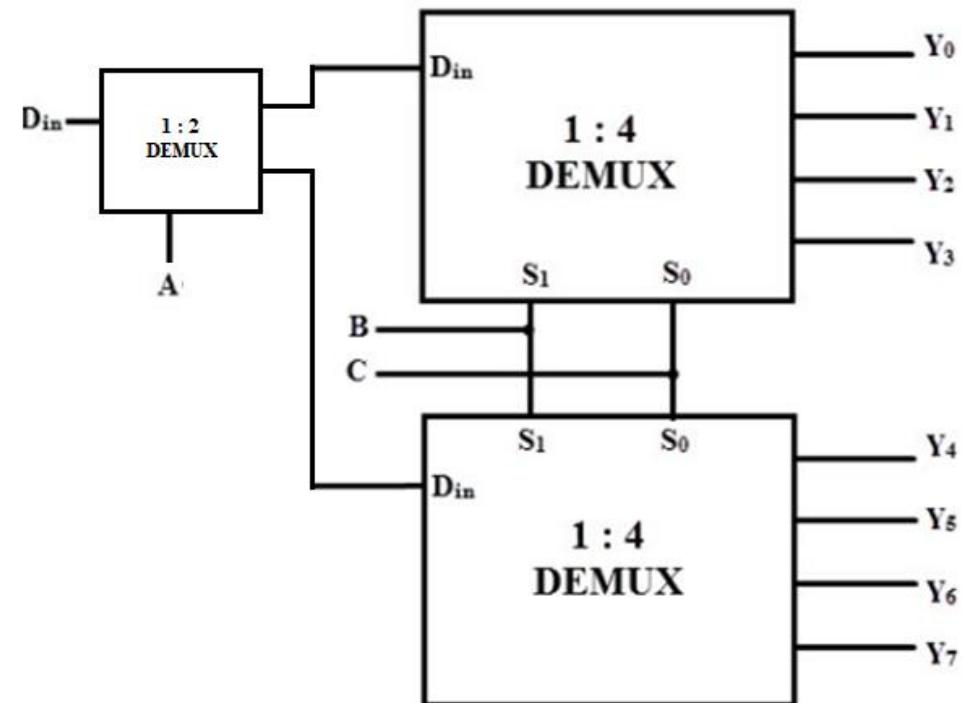
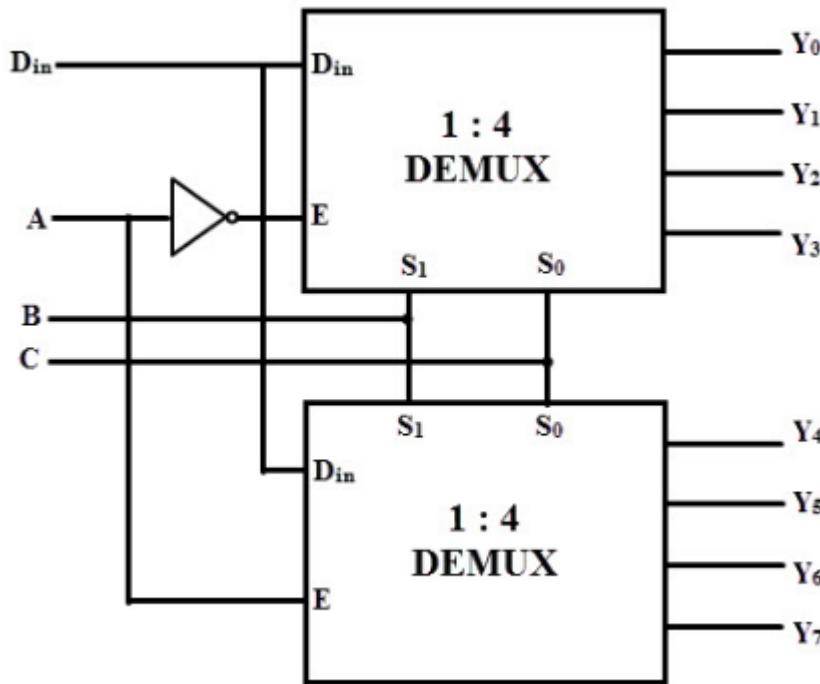
$$Y_5 = S_2 \overline{S}_1 S_0 D$$

$$Y_6 = S_2 S_1 \overline{S}_0 D$$

$$Y_7 = S_2 S_1 S_0 D$$



Example: Construct a 1-to-8 DEMUX using two 1-to-4 De-multiplexers.



DIGITAL ELECTRONICS

III Semester CSE & CST



Mr. Srikanta Patnaik Mr. Ajit Kumar Patro Mr. Ami Kumar Parida

DEPARTMENT OF ELECTRONICS AND COMMUNICATION ENGINEERING

SCHOOL OF ENGINEERING & TECHNOLOGY

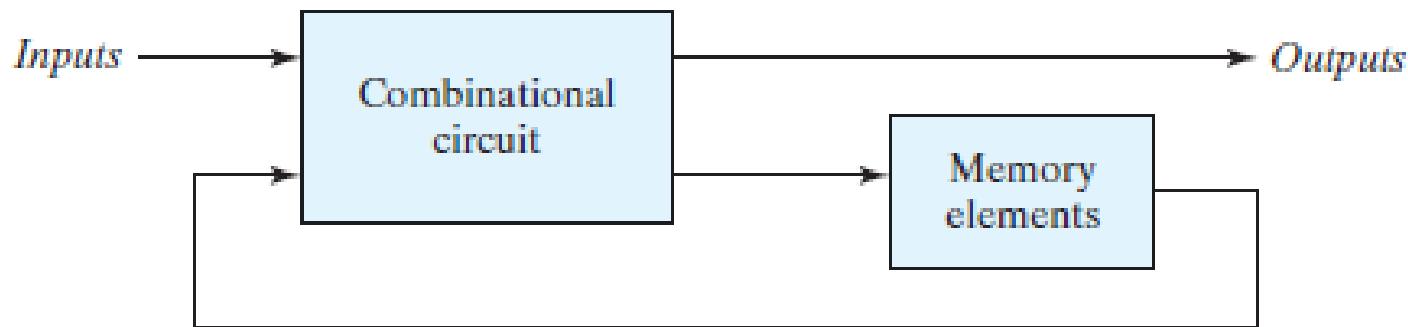
GIET UNIVERSITY, GUNUPUR, ODISHA

Synchronous Sequential Logic

Sequential Circuits;
Latches, Flip-Flops;
Master-Slave Flip-Flop

SEQUENTIAL CIRCUITS

- A block diagram of a sequential circuit is shown in Fig.



- It consists of a combinational circuit to which storage elements are connected to form a feedback path.

- The storage elements are devices capable of storing binary information. The binary information stored in these elements at any given time defines the *state* of the sequential circuit at that time.
- The block diagram demonstrates that the outputs in a sequential circuit are a function not only of the inputs, but also of the present state of the storage elements.
- In contrast, the outputs of combinational logic depend only on the present values of the inputs.

- There are two main types of sequential circuits, and their classification is a function of the timing of their signals.
- A *synchronous* sequential circuit is a system whose behavior can be defined from the knowledge of its signals at discrete instants of time.
- The behavior of an *asynchronous* sequential circuit depends upon the input signals at any instant of time and the order in which the inputs change.

- A synchronous sequential circuit employs signals that affect the storage elements at only discrete instants of time.
- Synchronization is achieved by a timing device called a *clock generator*, which provides a clock signal having the form of a periodic train of *clock pulses*.
- The clock signal is commonly denoted by the identifiers *clock* and *clk*.

- The clock pulses are distributed throughout the system in such a way that storage elements are affected only with the arrival of each pulse.
- Synchronous sequential circuits that use clock pulses to control storage elements are called *clocked sequential circuits*.

Difference between Combinational and Sequential Circuits

Combinational Circuit

- A circuit whose output depends only upon the present inputs.
- These circuits do not have any memory element.
- There is no feedback between input and output.
- This is time independent.
- It does not have clock signal.
- Elementary building blocks: Logic gates.
- Used for arithmetic as well as Boolean operations.
- Speed is fast.
- It is designed easy.
- It is easy to use and handle.
- Examples - Encoder, Décoder, Multiplexer, De-multiplexer.

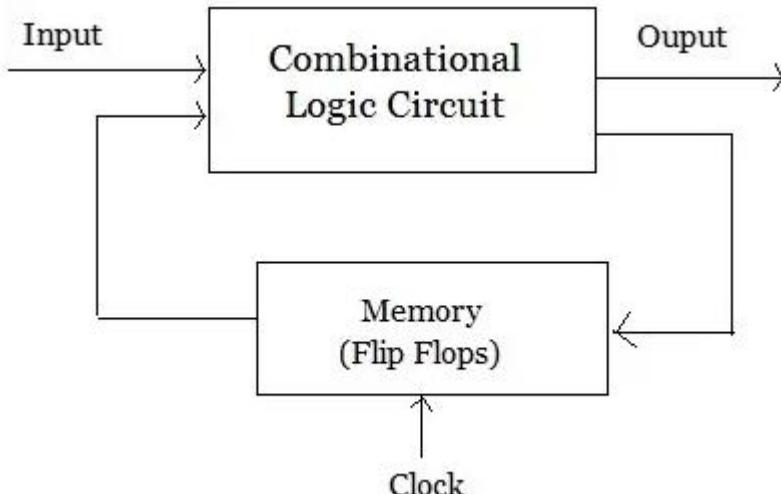
Sequential Circuit

- A circuit output depends upon the present inputs as well as past history of the inputs.
- These circuits have memory element.
- There exists a feedback path between input and output.
- This is time dependent.
- It may or may not have clock signal but most sequential circuit have a clock signal.
- Elementary building blocks: Flip-flops.
- Mainly used for storing data.
- Speed is slow.
- It is designed tough as compared to combinational circuits.
- It is not easy to use and handle.
- Examples - Flip-flops, Counters, Shift registers.

Difference between Synchronous and Asynchronous Sequential Circuits

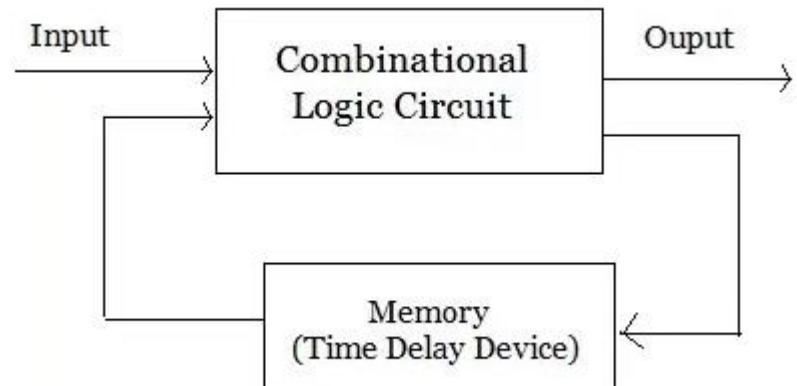
Synchronous Sequential Circuit

- A sequential circuit whose output behavior depends on the input at a discrete-time is called synchronous sequential circuits. Synchronous sequential circuits that use clock pulses in the inputs of memory elements are called *clocked sequential circuits*.



Asynchronous Sequential Circuit

- The sequential circuit whose output depends on the sequence in which the input changes are called asynchronous sequential circuits.



- Synchronous sequential circuits are digital circuits governed by clock signals.
- Output behavior depends on the input at discrete time.
- In synchronous sequential circuits memory elements are used like clocked flip flop.
- Synchronous sequential circuits are easier to describe, analyze and design.
- Due to the presence of clock pulse the operating speed of synchronous sequential circuits is low.
- In these circuits change in state occurs in response to clock pulse.
- Asynchronous sequential circuits are digital circuits that are not driven by clock. They can be called as *self-timed circuits*.
- Output depends on the sequence in which the input changes.
- In asynchronous sequential circuits un-coded memory elements are used like un-coded flip flop or time delay elements.
- Asynchronous sequential circuits are more difficult to describe, analyze and design.
- Because of absence of clock pulse, it can be operate faster than synchronous sequential circuits.
- In these circuits state change occurs whenever input variable change.

STORAGE ELEMENTS

- A storage element in a digital circuit can maintain a binary state indefinitely (as long as power is delivered to the circuit), until directed by an input signal to switch states.
- *Storage elements that operate with signal levels (rather than signal transitions) are referred to as latches ; those controlled by a clock transition are flip-flops.*
- Latches are said to be level sensitive devices; flip-flops are edge-sensitive devices.

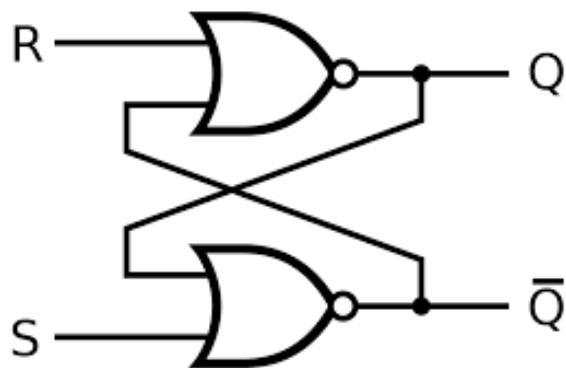
- The two types of storage elements are related because latches are the basic circuits from which all flip-flops are constructed.
- Although latches are useful for storing binary information and for the design of asynchronous sequential circuits, they are not practical for use as storage elements in synchronous sequential circuits.

LATCHES

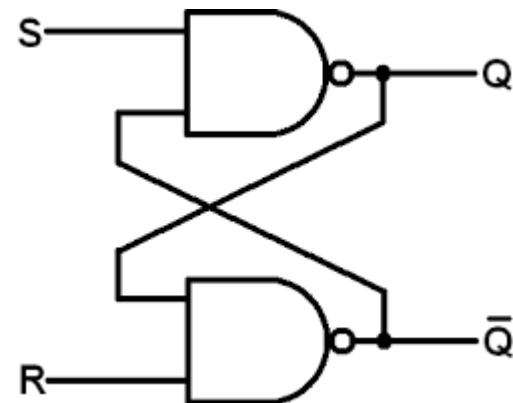
- Latch is an electronic logic circuit with two stable states namely high output as well as low output i.e. it is a bistable multivibrator.
- The latch has two useful states.
 - When output $Q = 1$ and $Q' = 0$, the latch is said to be in the *set state*.
 - When $Q = 0$ and $Q' = 1$, it is in the *reset state*.
- Outputs Q and Q' are normally the complement of each other.

SR Latch

- The *SR* latch is a circuit with two cross-coupled NOR gates or two cross-coupled NAND gates, and two inputs labeled *S* for set and *R* for reset.

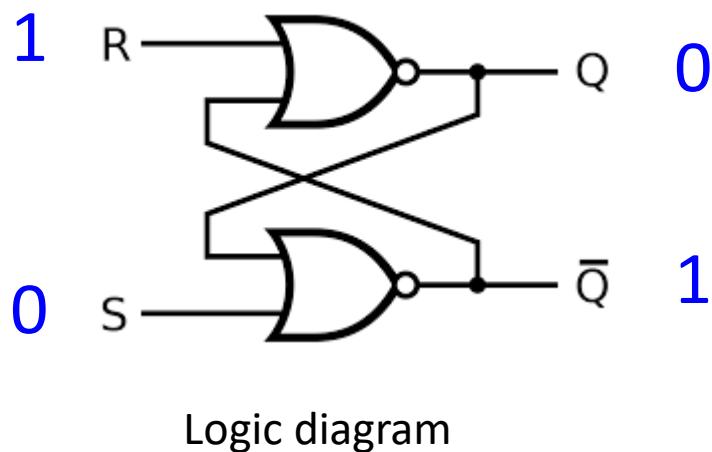


SR latch with NOR gates



SR latch with NAND gates

SR latch with NOR gates



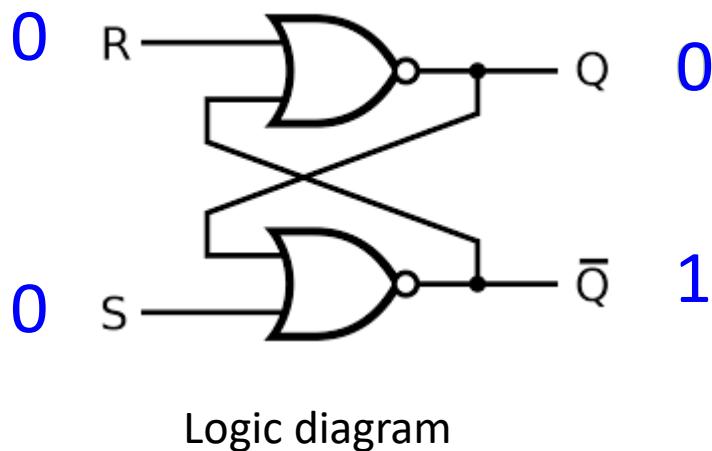
| <i>S</i> | <i>R</i> | <i>Q</i> | \bar{Q} |
|----------|----------|----------|-----------|
| 0 | 0 | 0 | 1 |
| 0 | 1 | 0 | 1 |
| 1 | 0 | 1 | 0 |
| 1 | 1 | 0 | 1 |

Reset

Function table

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

SR latch with NOR gates

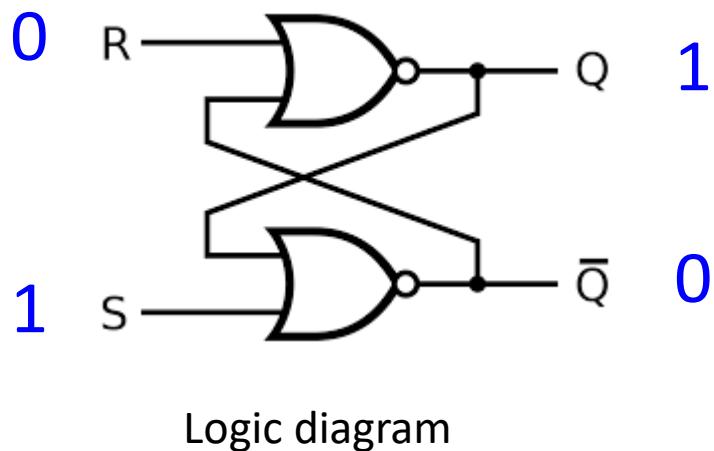


| S | R | Q | \bar{Q} |
|-----|-----|-----------|-----------|
| 0 | 0 | 0 | 1 |
| 0 | 1 | 0 | 1 |
| 1 | 0 | 1 | 0 |
| 1 | 1 | No change | Reset |

Function table

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

SR latch with NOR gates

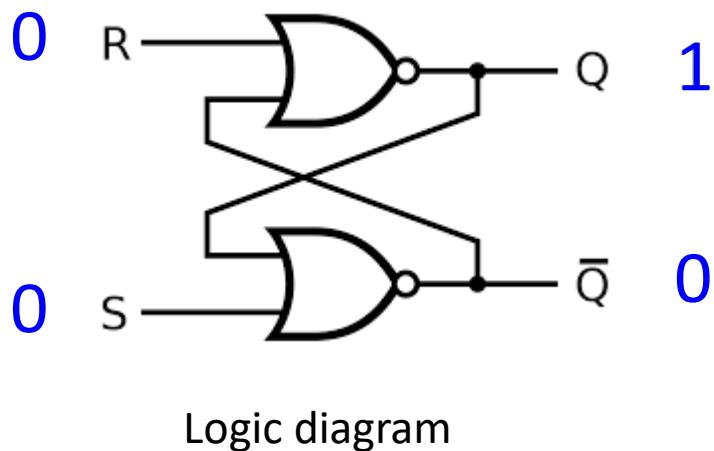


| <i>S</i> | <i>R</i> | <i>Q</i> | \bar{Q} | |
|----------|----------|----------|-----------|------------------|
| 0 | 0 | 0 | 1 | <i>No change</i> |
| 0 | 1 | 0 | 1 | <i>Reset</i> |
| 1 | 0 | 1 | 0 | <i>Set</i> |
| 1 | 1 | | | |

Function table

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

SR latch with NOR gates

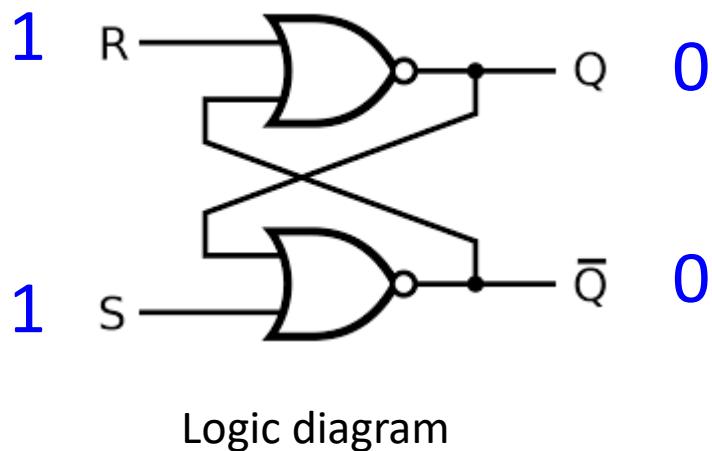


| <i>S</i> | <i>R</i> | <i>Q</i> | \bar{Q} | |
|----------|----------|----------|-----------|-----------|
| 0 | 0 | 0 | 1 | No change |
| 1 | 0 | 1 | 0 | No change |
| 0 | 1 | 0 | 1 | Reset |
| 1 | 0 | 1 | 0 | Set |
| 1 | 1 | | | |

Function table

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

SR latch with NOR gates

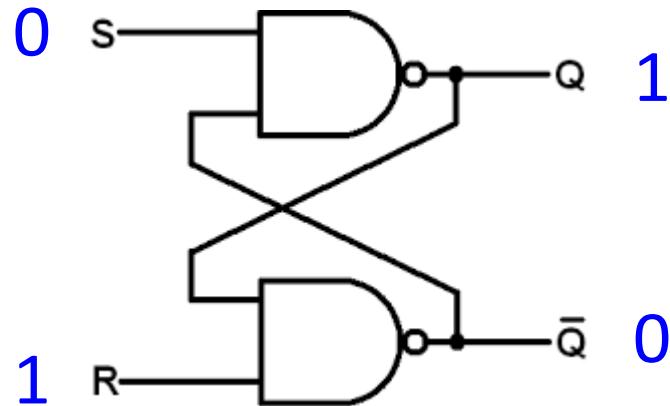


| S | R | Q | \bar{Q} | |
|-----|-----|-----|-----------|---------------|
| 0 | 0 | 0 | 1 | No change |
| 1 | 0 | 0 | 1 | No change |
| 0 | 1 | 0 | 1 | Reset |
| 1 | 0 | 1 | 0 | Set |
| 1 | 1 | 0 | 0 | Indeterminate |

Function table

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

SR latch with NAND gates



Logic diagram

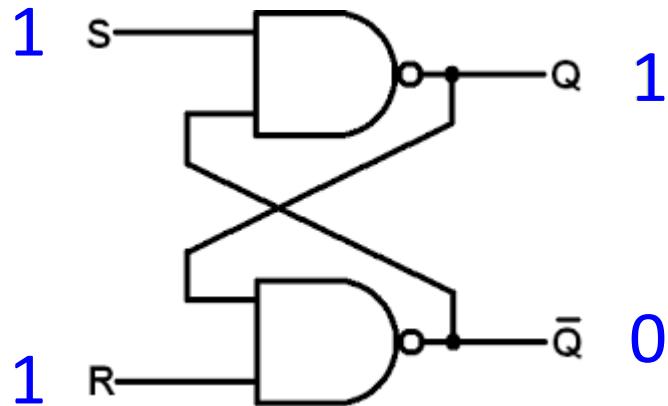
| S | R | Q | \bar{Q} |
|---|---|---|-----------|
| 0 | 0 | | |
| 0 | 1 | 1 | 0 |
| 1 | 0 | 0 | 1 |
| 1 | 1 | | |

1 0 Set

| Input | | Output |
|-------|---|----------------------------|
| A | B | $Y = \overline{A \cdot B}$ |
| 0 | 0 | 1 |
| 0 | 1 | 1 |
| 1 | 0 | 1 |
| 1 | 1 | 0 |

Function table

SR latch with NAND gates



Logic diagram

| <i>S</i> | <i>R</i> | <i>Q</i> | \bar{Q} |
|----------|----------|----------|-----------|
| 0 | 0 | | |
| 0 | 1 | 1 | 0 |
| 1 | 0 | | |
| 1 | 1 | 1 | 0 |

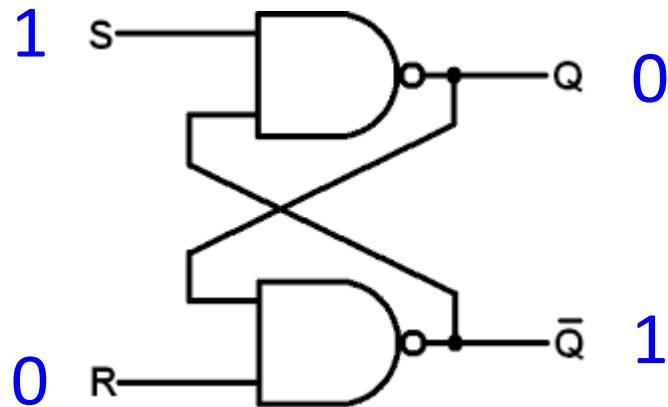
Set

No change

| Input | | Output |
|-------|---|----------------------------|
| A | B | $Y = \overline{A \cdot B}$ |
| 0 | 0 | 1 |
| 0 | 1 | 1 |
| 1 | 0 | 1 |
| 1 | 1 | 0 |

Function table

SR latch with NAND gates



Logic diagram

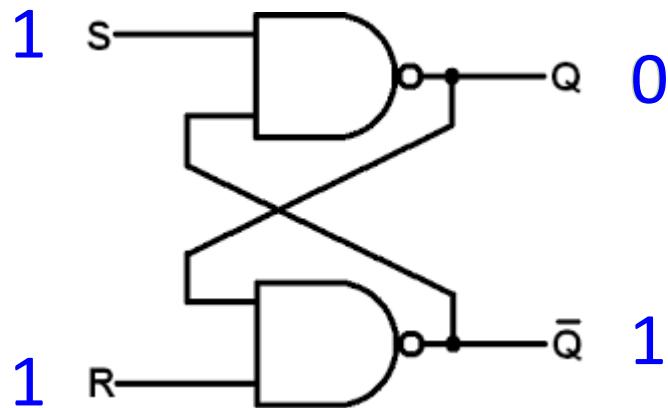
| S | R | Q | \bar{Q} |
|---|---|---|-----------|
| 0 | 0 | 0 | 1 |
| 0 | 1 | 1 | 0 |
| 1 | 0 | 0 | 1 |
| 1 | 1 | 1 | 0 |

Set
Reset
No change

| Input | | Output |
|-------|---|----------------------------|
| A | B | $Y = \overline{A \cdot B}$ |
| 0 | 0 | 1 |
| 0 | 1 | 1 |
| 1 | 0 | 1 |
| 1 | 1 | 0 |

Function table

SR latch with NAND gates



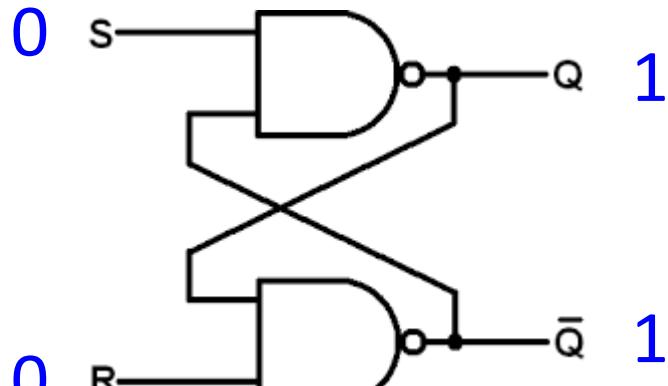
Logic diagram

| S | R | Q | \bar{Q} | |
|---|---|---|-----------|------------------|
| 0 | 0 | | | |
| 0 | 1 | 1 | 0 | <i>Set</i> |
| 1 | 0 | 0 | 1 | <i>Reset</i> |
| 1 | 1 | 1 | 0 | <i>No change</i> |
| 0 | 1 | 0 | 1 | <i>No change</i> |

Function table

| Input | | Output |
|-------|---|----------------------------|
| A | B | $Y = \overline{A \cdot B}$ |
| 0 | 0 | 1 |
| 0 | 1 | 1 |
| 1 | 0 | 1 |
| 1 | 1 | 0 |

SR latch with NAND gates

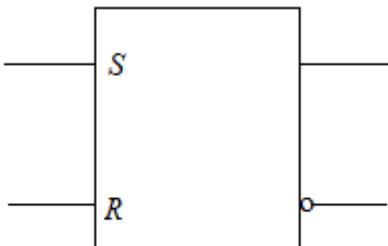


Logic diagram

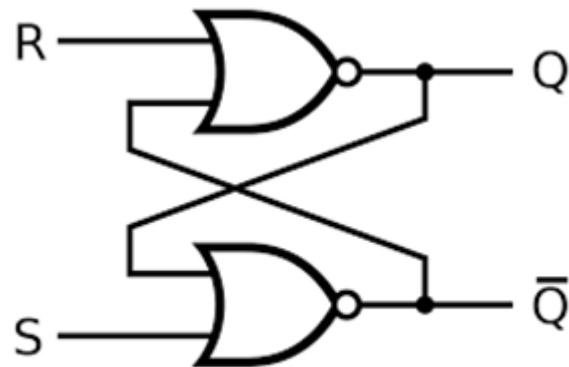
| S | R | Q | \bar{Q} | |
|---|---|---|-----------|---------------|
| 0 | 0 | 1 | 1 | Indeterminate |
| 0 | 1 | 1 | 0 | Set |
| 1 | 0 | 0 | 1 | Reset |
| 1 | 1 | 1 | 0 | No change |
| 0 | 1 | 0 | 1 | No change |

Function table

| Input | | Output |
|-------|---|----------------------------|
| A | B | $Y = \overline{A \cdot B}$ |
| 0 | 0 | 1 |
| 0 | 1 | 1 |
| 1 | 0 | 1 |
| 1 | 1 | 0 |



(a) Graphic symbol

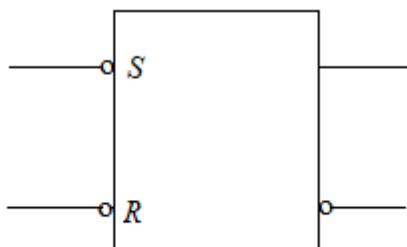


(b) Logic diagram

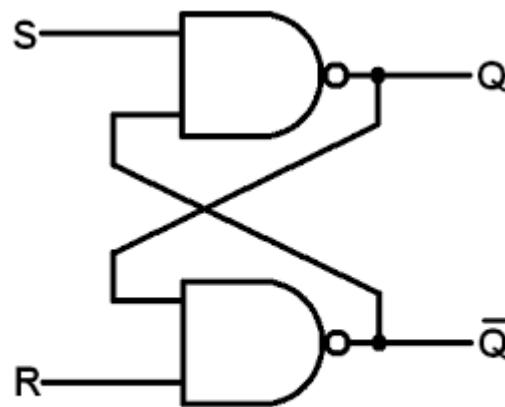
| S | R | Q | \bar{Q} | |
|---|---|---|-----------|---------------|
| 0 | 0 | Q | Q' | No change |
| 0 | 1 | 0 | 1 | Reset |
| 1 | 0 | 1 | 0 | Set |
| 1 | 1 | 0 | 0 | Indeterminate |

(c) Function table

SR latch with NOR gates



(a) Graphic symbol



(b) Logic diagram

| S | R | Q | \bar{Q} | |
|---|---|---|-----------|---------------|
| 0 | 0 | 1 | 1 | Indeterminate |
| 0 | 1 | 1 | 0 | Set |
| 1 | 0 | 0 | 1 | Reset |
| 1 | 1 | Q | Q' | No change |

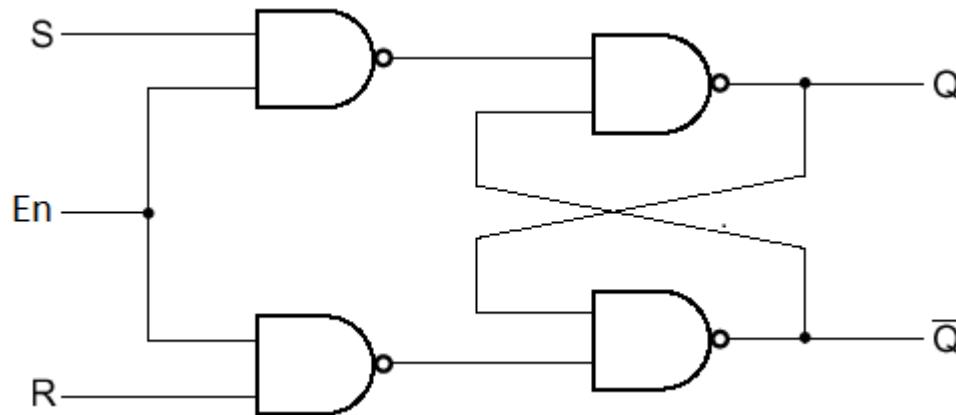
(c) Function table

SR latch with NAND gates

- In comparing the NAND with the NOR latch, note that the input signals for the NAND require the complement of those values used for the NOR latch.
- Because the NAND latch requires a 0 signal to change its state, it is sometimes referred to as an $S'R'$ latch.
- *SR* latch with NOR gates is called *Active-high SR Latch* because which is SET when $S = 1$ (HIGH).
- *SR* latch with NAND gates is called *Active-low SR Latch* because which is SET when $S = 0$ (LOW).

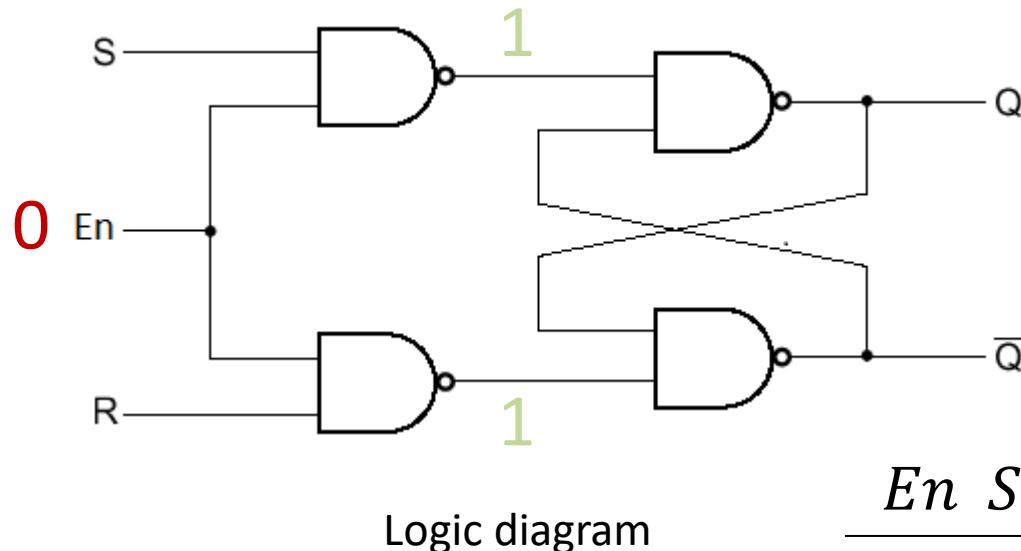
SR latch with control input

- The operation of the basic *SR* latch can be modified by providing an additional input signal that determines (controls) *when* the state of the latch can be changed by determining whether *S* and *R* (or *S'* and *R'*) can affect the circuit.
- An *SR* latch with a control input is shown in Fig.



- It consists of the basic *SR* latch and two additional NAND gates.
- The control input *En* acts as an *enable* signal for the other two inputs.
- The outputs of the NAND gates stay at the logic-1 level as long as the enable signal remains at 0.
- This is the quiescent condition for the *SR* latch.
- When the enable input goes to 1, information from the *S* or *R* input is allowed to affect the latch.

SR latch with control input

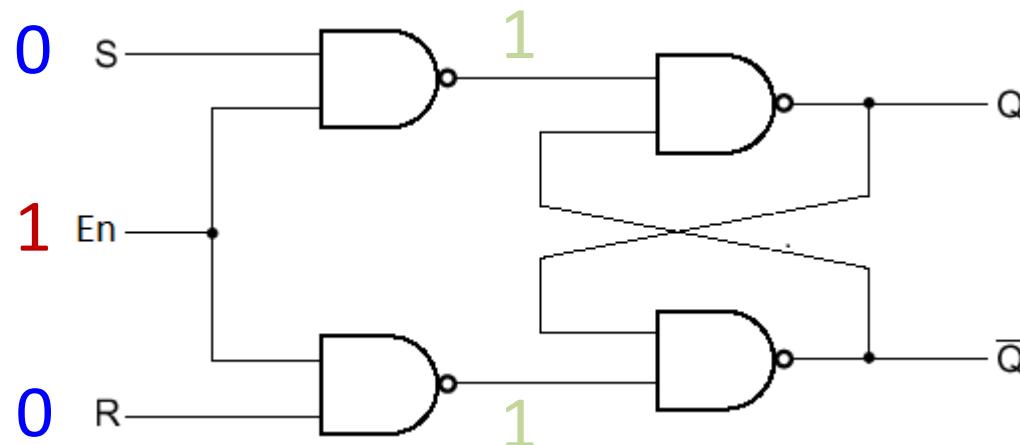


| S | R | Q | \bar{Q} |
|-----|-----|-----|-----------|
| 0 | 0 | 1 | 1 |
| 0 | 1 | 1 | 0 |
| 1 | 0 | 0 | 1 |
| 1 | 1 | Q | Q' |

| | | | Next state of | |
|------|-----|-----|---------------|-----------|
| En | S | R | Q | \bar{Q} |
| 0 | X | X | Q | Q' |
| 1 | 0 | 0 | | |
| 1 | 0 | 1 | | |
| 1 | 1 | 0 | | |
| 1 | 1 | 1 | | |

Function table

SR latch with control input



Logic diagram

| S | R | Q | \bar{Q} |
|---|---|---|-----------|
| 0 | 0 | 1 | 1 |
| 0 | 1 | 1 | 0 |
| 1 | 0 | 0 | 1 |
| 1 | 1 | Q | Q' |

Indeterminate

Set

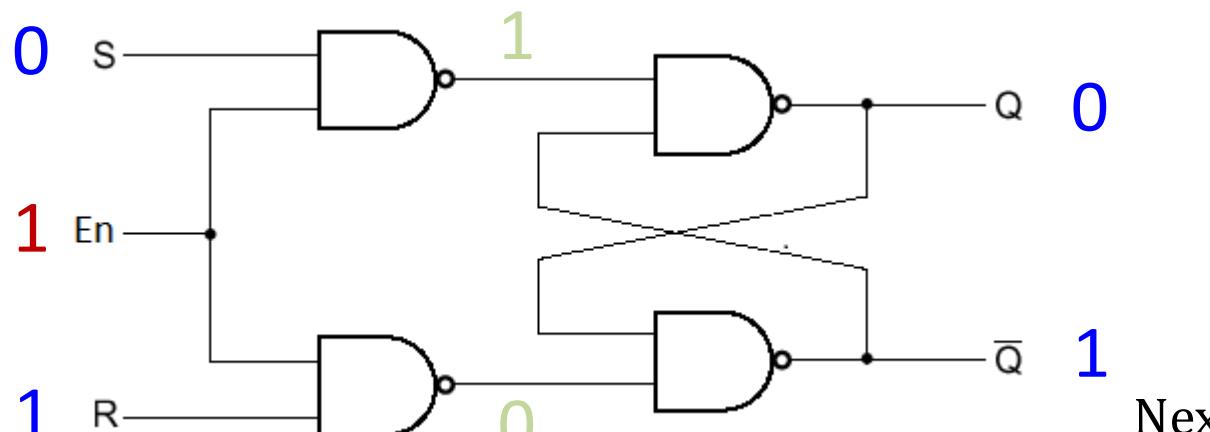
Reset

No change

| | | | Next state of | |
|----|---|---|------------------|-----------|
| En | S | R | Q | \bar{Q} |
| 0 | X | X | Q | Q' |
| 1 | 0 | 0 | Q | Q' |
| | | | <i>No change</i> | |
| | | | <i>No change</i> | |
| | | | | |
| | | | | |
| | | | | |
| | | | | |

Function table

SR latch with control input



Logic diagram

| S | R | Q | \bar{Q} |
|---|---|---|-----------|
| 0 | 0 | 1 | 1 |
| 0 | 1 | 1 | 0 |
| 1 | 0 | 0 | 1 |
| 1 | 1 | Q | \bar{Q} |

Indeterminate

Set

Reset

No change

| | | | Next state of | |
|----|---|---|---------------|------------|
| En | S | R | Q | \bar{Q} |
| 0 | X | X | Q | \bar{Q}' |
| 1 | 0 | 0 | Q | \bar{Q}' |
| 1 | 0 | 1 | 0 | 1 |
| 1 | 1 | 0 | | |
| 1 | 1 | 1 | | |

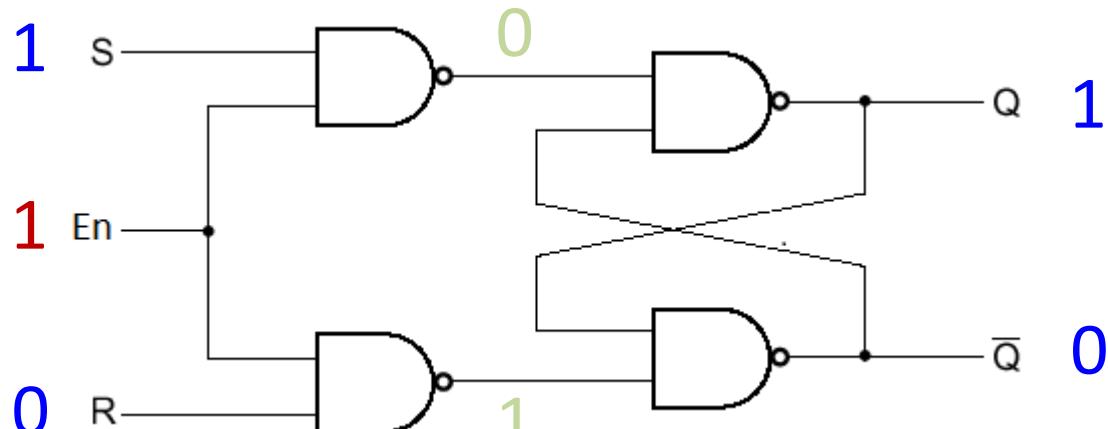
No change

No change

Reset

Function table

SR latch with control input



1

0

Next state of

 $Q \quad \bar{Q}$

0 X X

1 0 0

1 0 1

1 1 0

1 1 1

 $Q \quad Q' \quad No\ change$ $Q \quad Q' \quad No\ change$

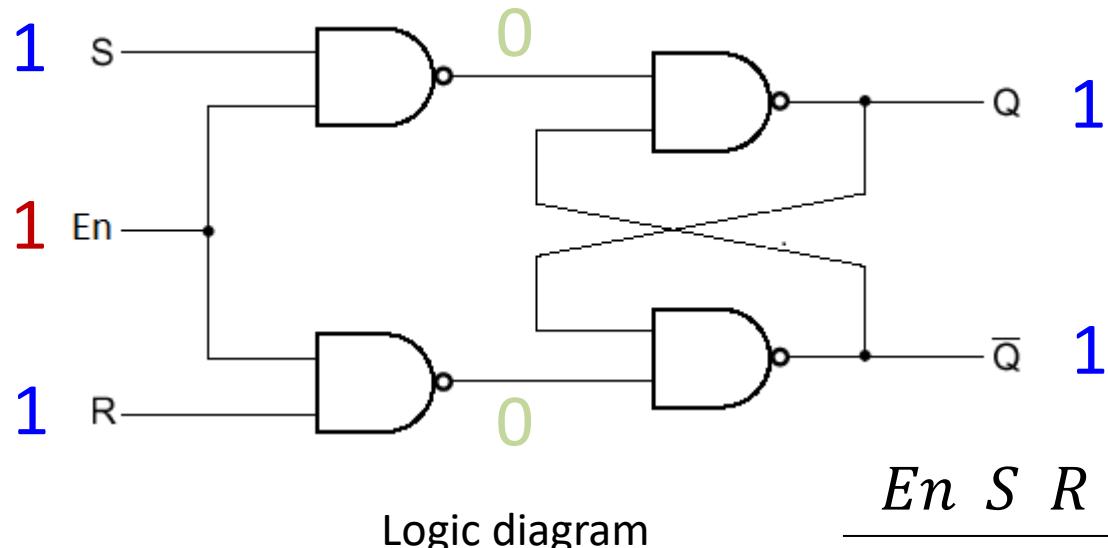
0 1 Reset

1 0 Set

| S | R | Q | \bar{Q} | |
|-----|-----|-----|-----------|---------------|
| 0 | 0 | 1 | 1 | Indeterminate |
| 0 | 1 | 1 | 0 | Set |
| 1 | 0 | 0 | 1 | Reset |
| 1 | 1 | Q | Q' | No change |

Function table

SR latch with control input



| S | R | Q | \bar{Q} |
|---|---|---|-----------|
| 0 | 0 | 1 | 1 |
| 0 | 1 | 1 | 0 |
| 1 | 0 | 0 | 1 |
| 1 | 1 | Q | \bar{Q} |

Indeterminate

Set

Reset

No change

| | | | Next state of | |
|----|---|---|---------------|------------|
| En | S | R | Q | \bar{Q} |
| 0 | X | X | Q | \bar{Q}' |
| 1 | 0 | 0 | Q | \bar{Q}' |
| 1 | 0 | 1 | 0 | 1 |
| 1 | 1 | 0 | 1 | 0 |
| 1 | 1 | 1 | 1 | 1 |

No change

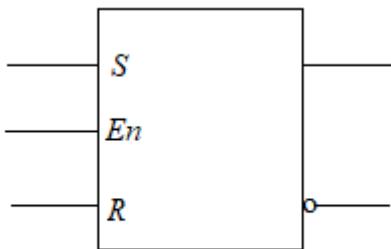
No change

Reset

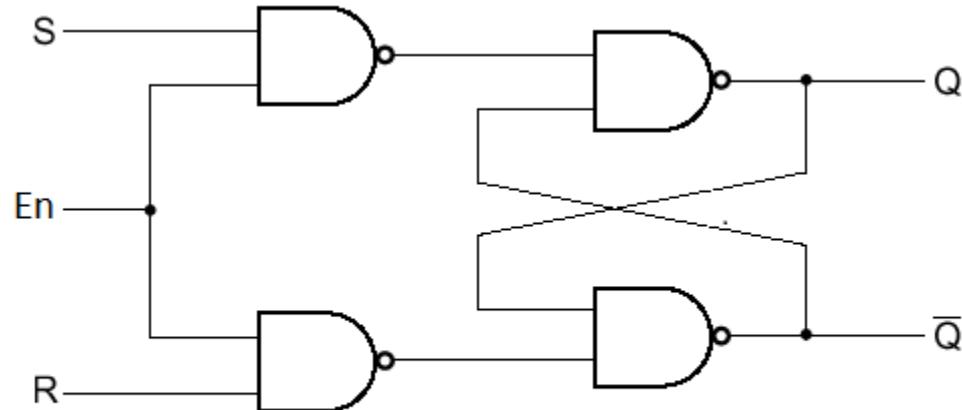
Set

Indeterminate

Function table



(a) Graphic symbol



(b) Logic diagram

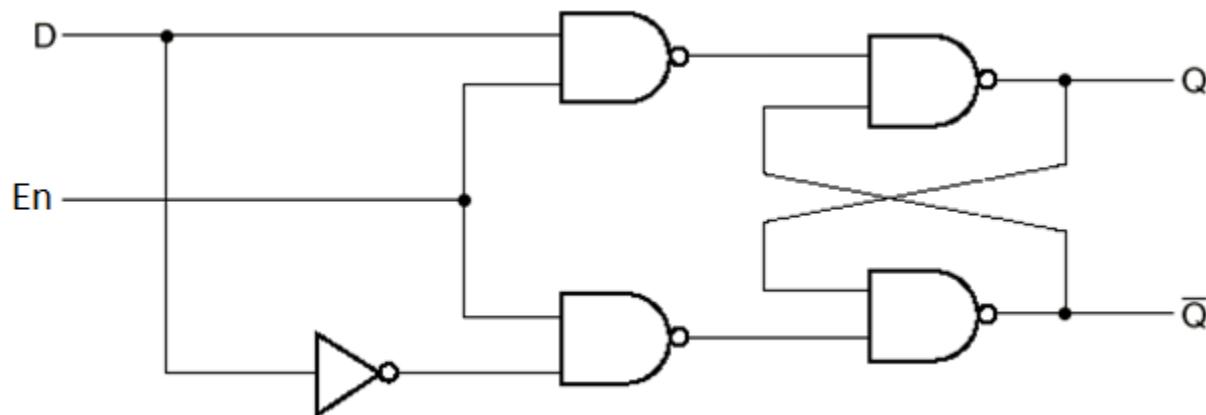
| | | | Next state of | | |
|------|-----|-----|---------------|-----------|---------------|
| En | S | R | Q | \bar{Q} | |
| 0 | X | X | Q | Q' | No change |
| 1 | 0 | 0 | Q | Q' | No change |
| 1 | 0 | 1 | 0 | 1 | Reset |
| 1 | 1 | 0 | 1 | 0 | Set |
| 1 | 1 | 1 | 1 | 1 | Indeterminate |

(c) Function table

SR latch with control input

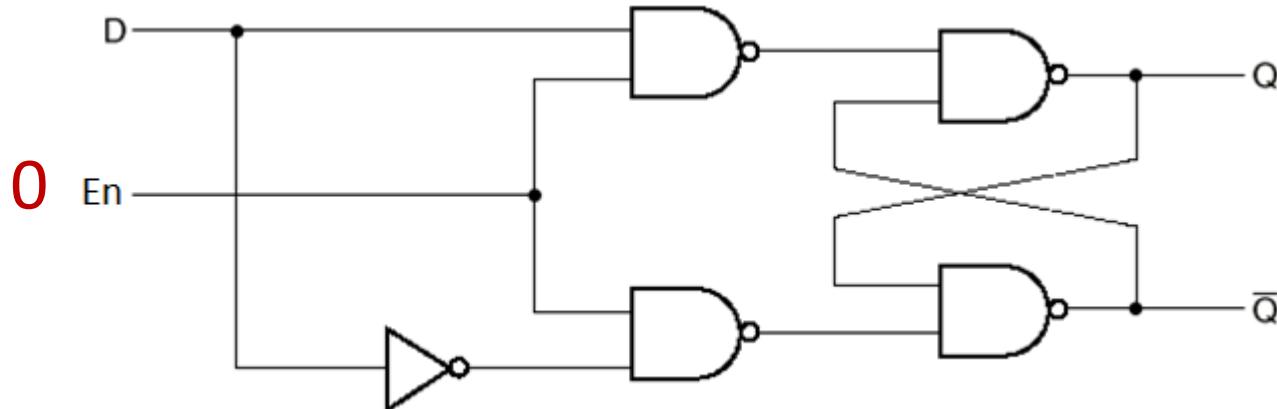
D Latch (Transparent Latch)

- One way to eliminate the undesirable condition of the indeterminate state in the *SR* latch is to ensure that inputs *S* and *R* are never equal to 1 at the same time.
- This is done in the *D* latch, shown in Fig.



- This latch has only two inputs: D (data) and En (enable).
- The D input goes directly to the S input, and its complement is applied to the R input.
- As long as the enable input is at 0, the cross-coupled SR latch has both inputs at the 1 level and the circuit cannot change state regardless of the value of D .
- The D input is sampled when $En = 1$.
 - If $D = 1$, the Q output goes to 1, placing the circuit in the set state.
 - If $D = 0$, output Q goes to 0, placing the circuit in the reset state.

D Latch



Logic diagram

| | | | Next state of | |
|----|---|---|---------------|-----------|
| En | S | R | Q | \bar{Q} |
| 0 | X | X | Q | Q' |
| 1 | 0 | 0 | Q | Q' |
| 1 | 0 | 1 | 0 | 1 |
| 1 | 1 | 0 | 1 | 0 |
| 1 | 1 | 1 | 1 | 1 |

No change

Reset

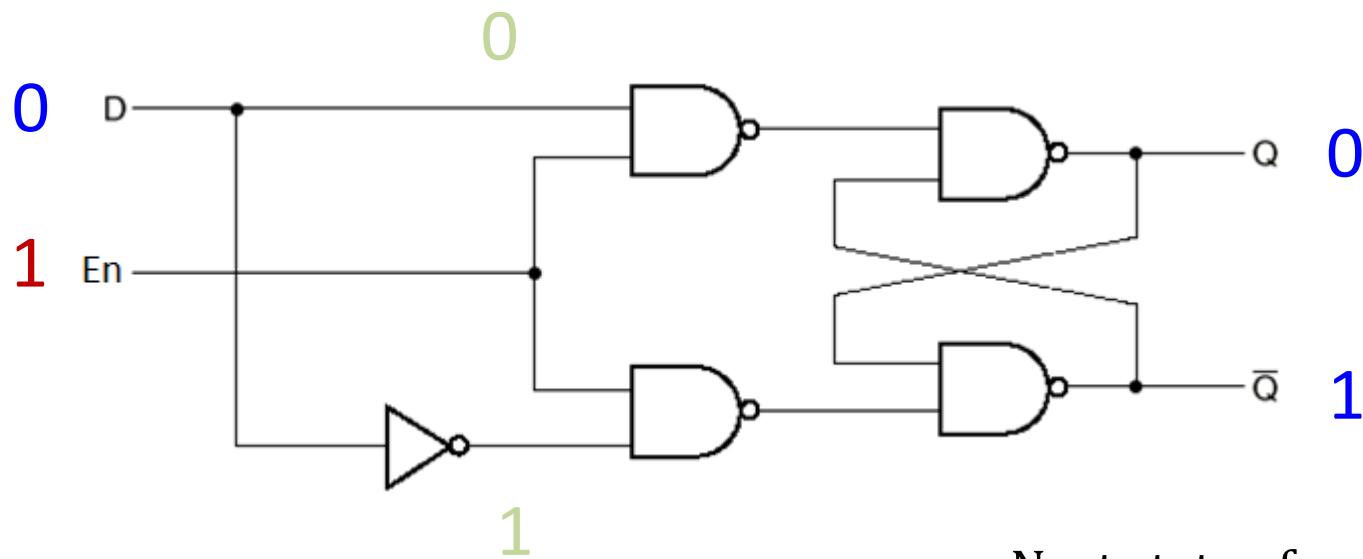
Set

Indeterminate

| | | Next state of | |
|----|---|---------------|-----------|
| En | D | Q | \bar{Q} |
| 0 | X | Q | Q' |
| 1 | 0 | | |
| 1 | 1 | | |

Function table

D Latch



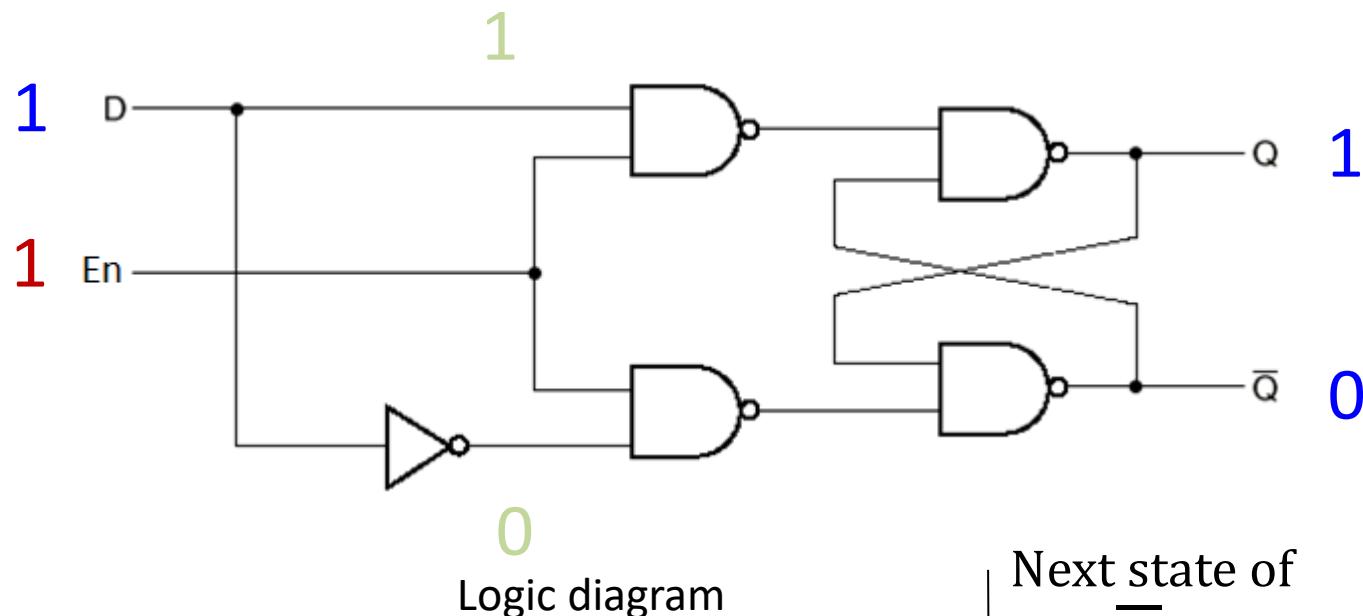
| | | | Next state of | |
|------|-----|-----|---------------|-----------|
| En | S | R | Q | \bar{Q} |
| 0 | X | X | Q | Q' |
| 1 | 0 | 0 | Q | Q' |
| 1 | 0 | 1 | 0 | 1 |
| 1 | 1 | 0 | 1 | 0 |
| 1 | 1 | 1 | 1 | 1 |

No change, *Reset*, *Set*, *Indeterminate*

| En | D | Next state of | |
|------|-----|---------------|-----------|
| | | Q | \bar{Q} |
| 0 | X | Q | Q' |
| 1 | 0 | 0 | 1 |
| 1 | 1 | 1 | 1 |

Function table

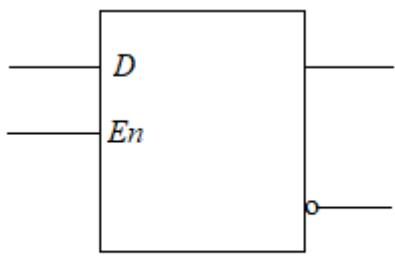
D Latch



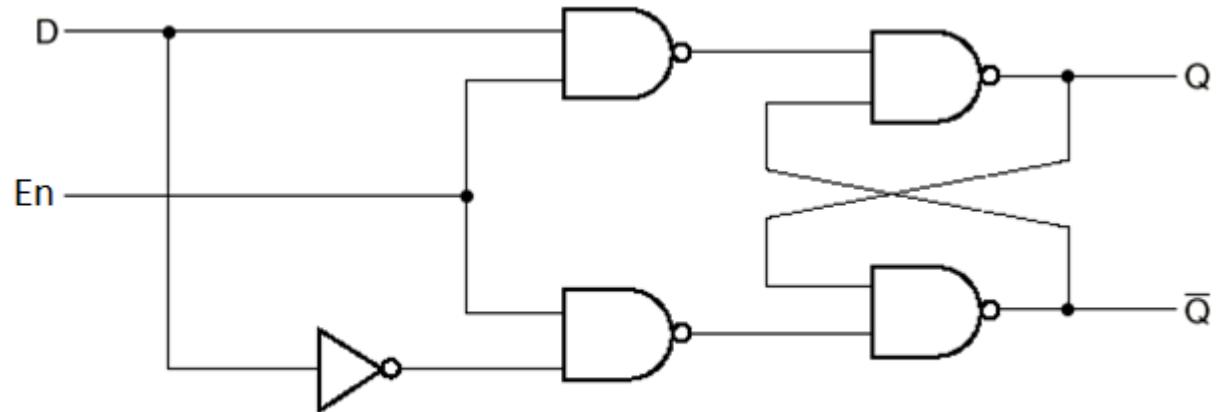
| | | | Next state of | | |
|------|-----|-----|---------------|-----------|---------------|
| En | S | R | Q | \bar{Q} | |
| 0 | X | X | Q | Q' | No change |
| 1 | 0 | 0 | Q | Q' | No change |
| 1 | 0 | 1 | 0 | 1 | Reset |
| 1 | 1 | 0 | 1 | 0 | Set |
| 1 | 1 | 1 | 1 | 1 | Indeterminate |

| En | D | Next state of | | |
|------|-----|---------------|-----------|-----------|
| | | Q | \bar{Q} | |
| 0 | X | Q | Q' | No change |
| 1 | 0 | 0 | 1 | Reset |
| 1 | 1 | 1 | 0 | Set |

Function table



(a) Graphic symbol



(b) Logic diagram

| | | Next state of | |
|----|---|---------------|------------|
| En | D | Q | \bar{Q} |
| 0 | X | Q | \bar{Q}' |
| | | | No change |
| 1 | 0 | 0 | 1 |
| | | | Reset |
| 1 | 1 | 1 | 0 |
| | | | Set |

(c) Function table

D latch

- The D latch receives that designation from its ability to hold *data* in its internal storage.
- It is suited for use as a temporary storage for binary information between a unit and its environment.
- The binary information present at the data input of the D latch is transferred to the Q output when the enable input is asserted. The output follows changes in the data input as long as the enable input is asserted.
- This situation provides a path from input D to the output, and for this reason, the circuit is often called a *transparent* latch.

FLIP-FLOPS

- The storage elements (memory) used in clocked sequential circuits are called *flip-flops*.
- A flip-flop is a binary storage device capable of storing one bit of information.
- In a stable state, the output of a flip-flop is either 0 or 1.
- A sequential circuit may use many flip-flops to store as many bits as necessary.

Difference between Flip-flop and Latch

Flip-flop

- Flip-flop is a bistable device i.e., it has two stable states that are represented as 0 and 1.
- It checks the inputs but changes the output only at times defined by the clock signal.
- It is a edge triggered device.
- Flip-flops are designed with latches by adding an extra clock signal.
- Flip-flop always have a clock signal.
- A flip-flop can be clocked for all time.
- Flip-Flops are very slow and consume more power.

Latch

- Latch is also a bistable device whose states are also represented as 0 and 1.
- It checks the inputs continuously and responds to the changes in inputs immediately.
- It is a level triggered device.
- The structure of latch is built with logic gates.
- Latch doesn't have a clock signal.
- A latch may be clock less or clocked.
- Latches are very fast and consume less power.

Clock response in latch and flip-flop



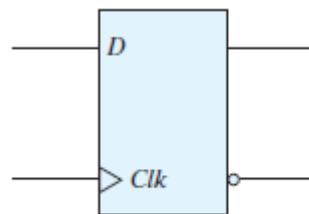
(a) Response to positive level



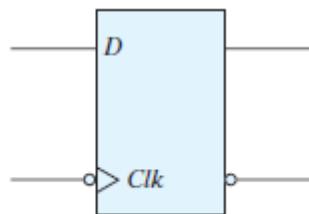
(b) Positive-edge response



(c) Negative-edge response

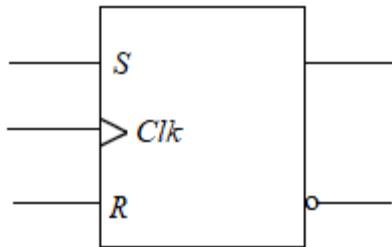


(a) Positive-edge

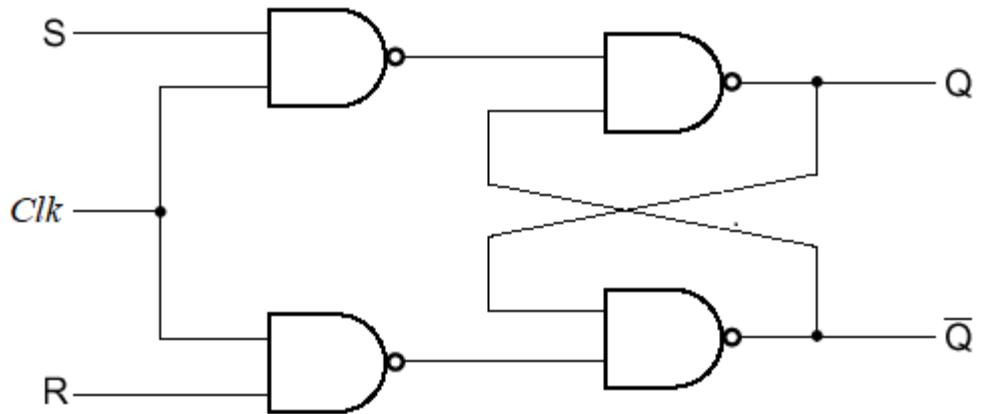


(a) Negative-edge

SR Flip-Flop



(a) Graphic symbol



(b) Logic diagram

| | | | Next state of | | |
|-------|-----|-----|---------------|-----------|---------------|
| Clk | S | R | Q | \bar{Q} | |
| 0 | X | X | Q | \bar{Q} | No change |
| 1 | 0 | 0 | Q | \bar{Q} | No change |
| 1 | 0 | 1 | 0 | 1 | Reset |
| 1 | 1 | 0 | 1 | 0 | Set |
| 1 | 1 | 1 | 1 | 1 | Indeterminate |

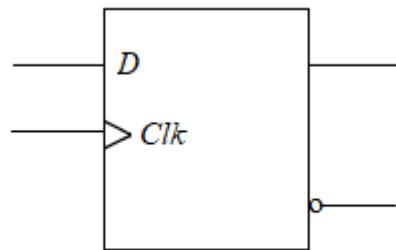
(c) Function table

| S | R | Q_{n+1} |
|-----|-----|-----------|
| 0 | 0 | Q_n |
| 0 | 1 | 0 |
| 1 | 0 | 1 |
| 1 | 1 | X |

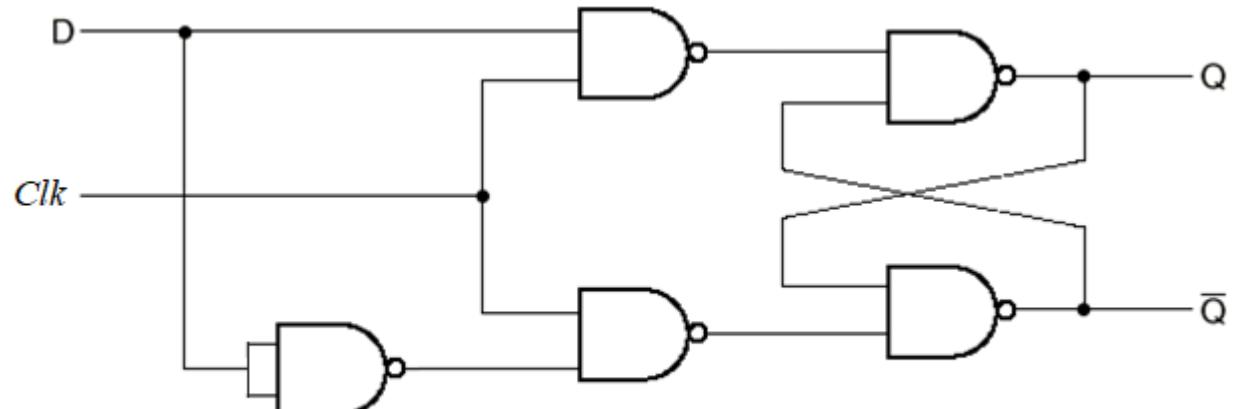
No change
Reset
Set
Indeterminate

(d) Characteristic table

D Flip-Flop



(a) Graphic symbol



(b) Logic diagram

| | | Next state of | | | |
|-------|-----|---------------|-----------|-----------|--|
| Clk | D | Q | \bar{Q} | | |
| 0 | X | Q | \bar{Q} | No change | |
| 1 | 0 | 0 | 1 | Reset | |
| 1 | 1 | 1 | 0 | Set | |

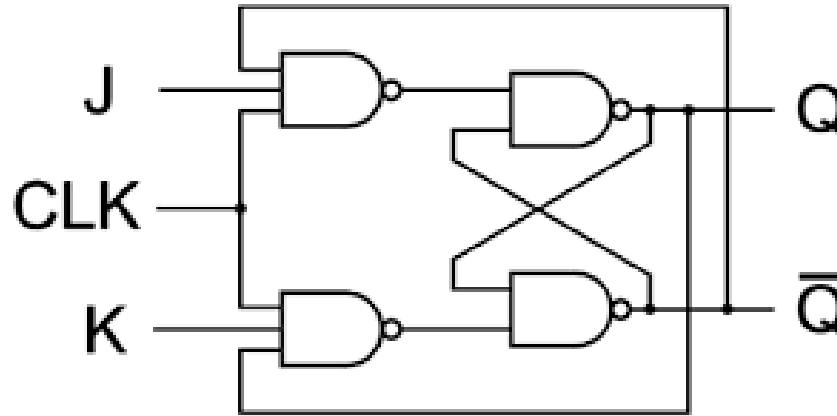
(c) Function table

| D | Q_{n+1} |
|-----|-----------|
| 0 | 0 |
| 1 | 1 |

(d) Characteristic table

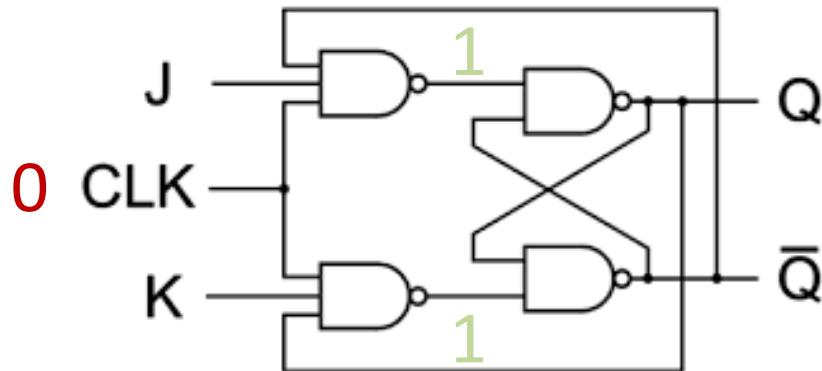
JK Flip-Flop

- There are three operations that can be performed with a flip-flop: set it to 1, reset it to 0, or complement its output.
- With only a single input, the *D* flip-flop can set or reset the output, depending on the value of the *D* input immediately before the clock transition.
- Synchronized by a clock signal, the *JK* flip-flop has two inputs and performs all three operations.
- The circuit diagram of a *JK* flip-flop constructed with a *SR* flip-flop and gates is shown in Fig.



- The two-input NAND gates of the RS flip-flop is replaced by the two 3 inputs NAND gates with the third input of each gate connected to the outputs at Q and \bar{Q} .
- The J input sets the flip-flop to 1, the K input resets it to 0, and when both inputs are enabled, the output is complemented.

JK Flip-Flop



| S | R | Q | \bar{Q} |
|-----|-----|-----|-----------|
| 0 | 0 | 1 | 1 |
| 0 | 1 | 1 | 0 |
| 1 | 0 | 0 | 1 |
| 1 | 1 | Q | Q' |

Indeterminate

Set

Reset

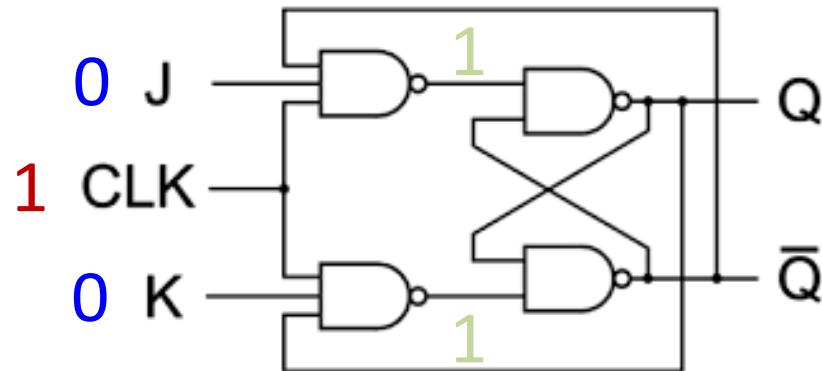
No change

| Clk | J | K | Q | \bar{Q} | Next state of | |
|-------|-----|-----|-----|-----------|---------------|-----------|
| | | | Q | \bar{Q} | Q | \bar{Q} |
| 0 | X | X | | | Q | Q' |
| 1 | 0 | 0 | | | 1 | 0 |
| 1 | 0 | 1 | 0 | 1 | 1 | 0 |
| | | | | | 0 | 1 |
| 1 | 1 | 0 | 0 | 1 | 1 | 0 |
| | | | | | 1 | 0 |
| 1 | 1 | 1 | 0 | 1 | 1 | 0 |
| | | | | | 0 | 1 |

No change

Function table

JK Flip-Flop



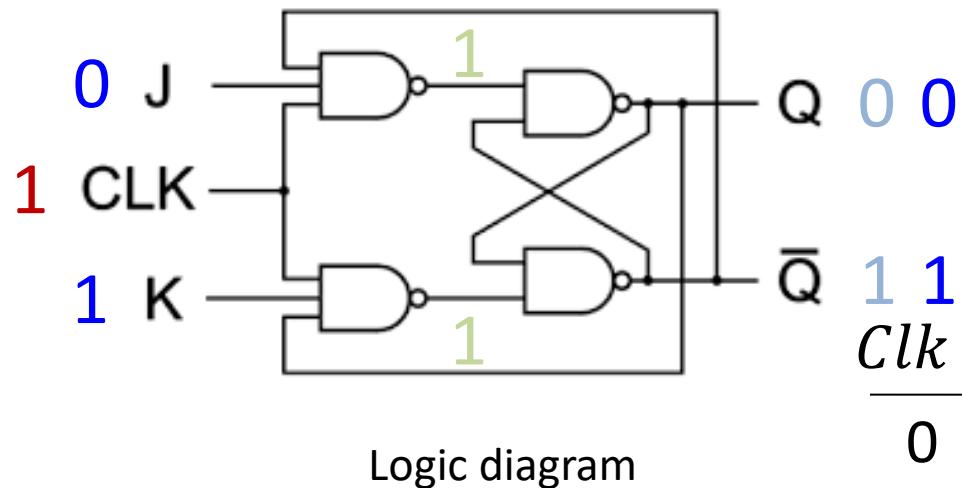
| S | R | Q | \bar{Q} |
|-----|-----|-----|-----------|
| 0 | 0 | 1 | 1 |
| 0 | 1 | 1 | 0 |
| 1 | 0 | 0 | 1 |
| 1 | 1 | Q | Q' |

Indeterminate
Set
Reset
No change

| Clk | J | K | Q | \bar{Q} | Next state of Q | \bar{Q} |
|-------|-----|-----|-----|-----------|----------------------|-----------|
| 0 | X | X | | | Q | Q' |
| 1 | 0 | 0 | | | Q | Q' |
| 1 | 0 | 1 | 0 | 1 | 1 | 0 |
| 1 | 1 | 0 | 0 | 1 | 1 | 0 |
| 1 | 1 | 1 | 0 | 1 | 1 | 0 |

Function table

JK Flip-Flop



| S | R | Q | \bar{Q} |
|-----|-----|-----|-----------|
| 0 | 0 | 1 | 1 |
| 0 | 1 | 1 | 0 |
| 1 | 0 | 0 | 1 |
| 1 | 1 | Q | Q' |

Indeterminate

Set

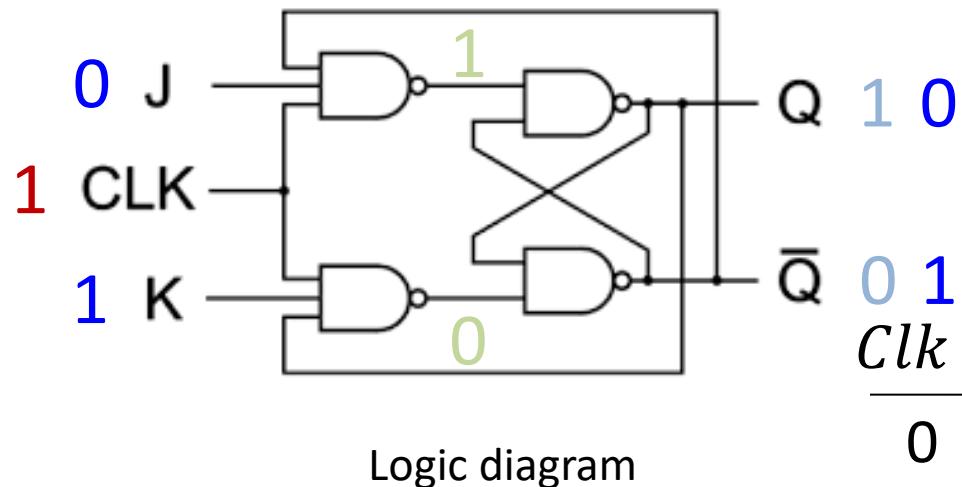
Reset

No change

Function table

| Clk | J | K | Q | \bar{Q} | Next state of Q | \bar{Q} |
|-------|-----|-----|-----|-----------|----------------------|-----------|
| 0 | X | X | | | Q | Q' |
| 1 | 0 | 0 | | | Q | Q' |
| 1 | 0 | 1 | 0 | 1 | 0 | 1 |
| | | | | | 1 | 0 |
| | 1 | 1 | 0 | 1 | 0 | 1 |
| | | | | | 1 | 0 |
| | 1 | 1 | 1 | 0 | 0 | 1 |
| | | | | | 1 | 0 |

JK Flip-Flop

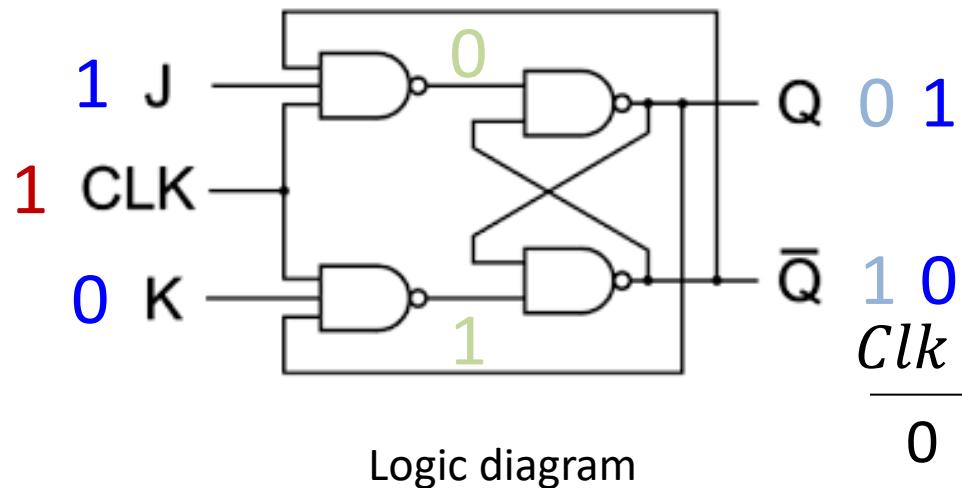


| S | R | Q | \bar{Q} | |
|-----|-----|-----|-----------|----------------------|
| 0 | 0 | 1 | 1 | <i>Indeterminate</i> |
| 0 | 1 | 1 | 0 | <i>Set</i> |
| 1 | 0 | 0 | 1 | <i>Reset</i> |
| 1 | 1 | Q | \bar{Q} | <i>No change</i> |

| Clk | J | K | Q | \bar{Q} | Next state of Q | \bar{Q} |
|-------|-----|-----|-----|-----------|----------------------|-----------|
| 0 | X | X | | | Q | Q' |
| 1 | 0 | 0 | | | Q | Q' |
| 1 | 0 | 1 | 0 | 1 | 0 | 1 |
| | | | 1 | 0 | 0 | 1 |
| | | | 1 | 0 | 1 | 0 |
| | | | 1 | 0 | 0 | 1 |
| | | | 1 | 0 | 1 | 0 |

Function table

JK Flip-Flop



| S | R | Q | \bar{Q} |
|-----|-----|-----|-----------|
| 0 | 0 | 1 | 1 |
| 0 | 1 | 1 | 0 |
| 1 | 0 | 0 | 1 |
| 1 | 1 | Q | Q' |

Indeterminate

Set

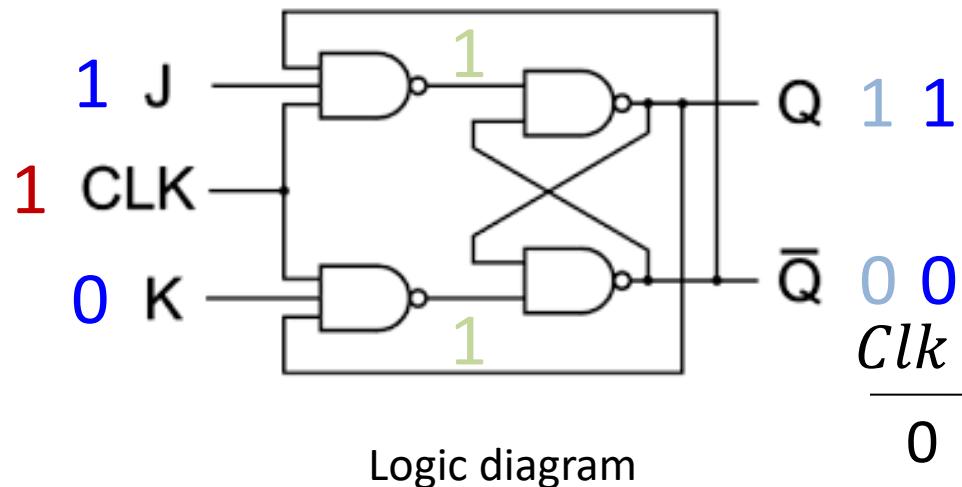
Reset

No change

| Clk | J | K | Q | \bar{Q} | Q | \bar{Q} | Next state of |
|-------|-----|-----|-----|-----------|-----|-----------|------------------|
| 0 | X | X | | | Q | Q' | <i>No change</i> |
| 1 | 0 | 0 | | | Q | Q' | <i>No change</i> |
| 1 | 0 | 1 | 0 | 1 | 0 | 1 | |
| | | | 1 | 0 | 0 | 1 | <i>Reset</i> |
| 1 | 1 | 0 | 0 | 1 | 1 | 0 | |
| | | | 1 | 0 | 1 | 0 | |
| 1 | 1 | 1 | 0 | 1 | | | |
| | | | 1 | 0 | | | |

Function table

JK Flip-Flop

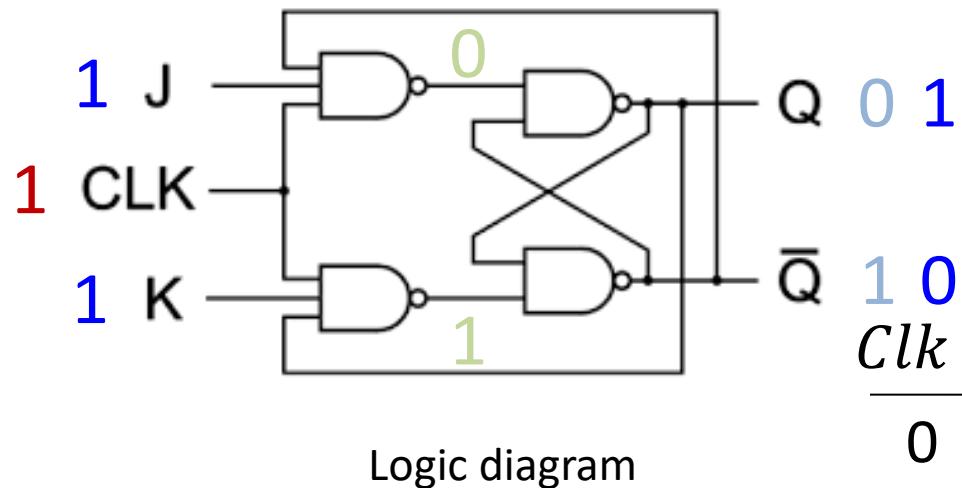


| S | R | Q | \bar{Q} |
|-----|-----|-----|---------------------|
| 0 | 0 | 1 | 1 |
| 0 | 1 | 1 | 0 |
| 1 | 0 | 0 | 1 |
| 1 | 1 | Q | \bar{Q} No change |

| Clk | J | K | Q | \bar{Q} | Next state of | | |
|-------|-----|-----|-----|-----------|---------------|-----------|-----------|
| | | | Q | \bar{Q} | Q | \bar{Q} | |
| 0 | X | X | | | Q | Q' | No change |
| 1 | 0 | 0 | | | Q | Q' | No change |
| 1 | 0 | 1 | 0 | 1 | 0 | 1 | |
| | | | 1 | 0 | 0 | 1 | Reset |
| 1 | 1 | 0 | 0 | 1 | 1 | 0 | |
| | | | 1 | 0 | 1 | 0 | Set |
| 1 | 1 | 1 | 0 | 1 | | | |
| | | | 1 | 0 | | | |

Function table

JK Flip-Flop



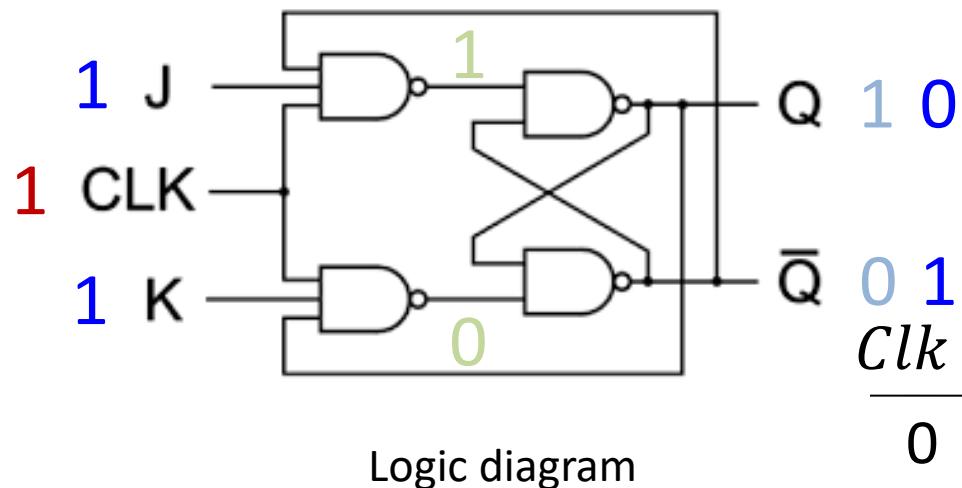
| S | R | Q | \bar{Q} |
|-----|-----|-----|-----------|
| 0 | 0 | 1 | 1 |
| 0 | 1 | 1 | 0 |
| 1 | 0 | 0 | 1 |
| 1 | 1 | Q | \bar{Q} |

No change

| Clk | J | K | Q | \bar{Q} | Next state of Q | \bar{Q} |
|-------|-----|-----|-----|-----------|----------------------|------------|
| 0 | X | X | | | Q | \bar{Q}' |
| 1 | 0 | 0 | | | Q | \bar{Q}' |
| 1 | 0 | 1 | 0 | 1 | 0 | 1 |
| | | | 1 | 0 | 0 | 1 |
| | | | 1 | 0 | 1 | 0 |
| 1 | 1 | 0 | 0 | 1 | 1 | 0 |
| | | | 1 | 0 | 1 | 0 |
| 1 | 1 | 1 | 0 | 1 | 1 | 0 |
| | | | 1 | 0 | | |

Function table

JK Flip-Flop

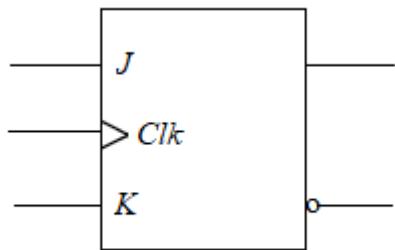


| S | R | Q | \bar{Q} | |
|-----|-----|-----|-----------|----------------------|
| 0 | 0 | 1 | 1 | <i>Indeterminate</i> |
| 0 | 1 | 1 | 0 | <i>Set</i> |
| 1 | 0 | 0 | 1 | <i>Reset</i> |
| 1 | 1 | Q | \bar{Q} | <i>No change</i> |

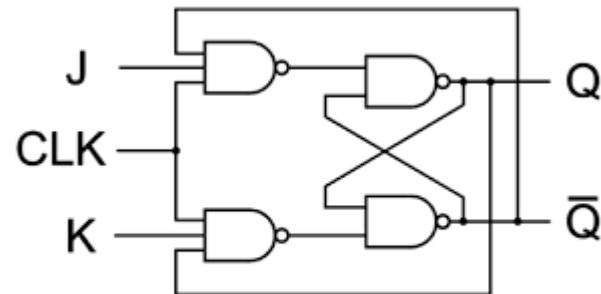
| Clk | J | K | Q | \bar{Q} | Next state of Q | \bar{Q} |
|-------|-----|-----|-----|-----------|----------------------|-----------|
| 0 | X | X | | | Q | Q' |
| 1 | 0 | 0 | | | Q | Q' |
| 1 | 0 | 1 | 0 | 1 | 0 | 1 |
| | | | 1 | 0 | 0 | 1 |
| | | | 1 | 0 | 1 | 0 |
| 1 | 1 | 0 | 0 | 1 | 1 | 0 |
| | | | 1 | 0 | 1 | 0 |
| 1 | 1 | 1 | 0 | 1 | 1 | 0 |
| | | | 1 | 0 | 0 | 1 |

Function table

JK Flip-Flop



(a) Graphic symbol



(b) Logic diagram

| | | | Next state of | |
|------------|----------|----------|---------------|------------|
| <i>Clk</i> | <i>J</i> | <i>K</i> | <i>Q</i> | \bar{Q} |
| 0 | X | X | <i>Q</i> | \bar{Q}' |
| 1 | 0 | 0 | <i>Q</i> | \bar{Q}' |
| 1 | 0 | 1 | 0 | 1 |
| 1 | 1 | 0 | 1 | 0 |
| 1 | 1 | 1 | \bar{Q}' | <i>Q</i> |

(c) Function table

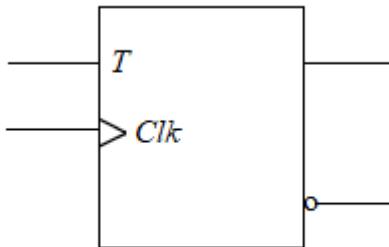
| <i>J</i> | <i>K</i> | Q_{n+1} |
|----------|----------|----------------------|
| 0 | 0 | <i>Q_n</i> |
| 0 | 1 | 0 |
| 1 | 0 | 1 |
| 1 | 1 | \bar{Q}_n |

(d) Characteristic table

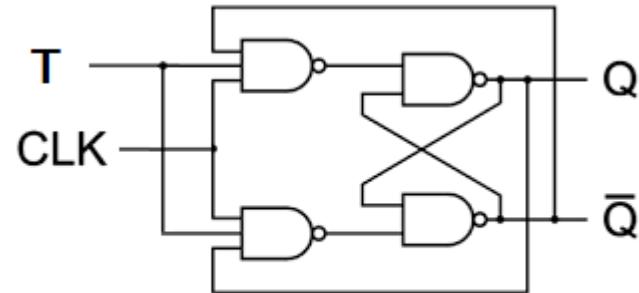
T Flip-Flop

- The T (toggle) flip-flop is a complementing flip-flop and can be obtained from a JK flip-flop when inputs J and K are tied together.
- When $T = 0$ ($J = K = 0$), a clock edge does not change the output.
- When $T = 1$ ($J = K = 1$), a clock edge complements the output.
- The complementing flip-flop is useful for designing binary counters.

T Flip-Flop



(a) Graphic symbol



(b) Logic diagram

| | | Next state of | | | |
|-------|-----|---------------|-----------|-------------------|--|
| Clk | T | Q | \bar{Q} | | |
| 0 | X | Q | Q' | <i>No change</i> | |
| 1 | 0 | Q | Q' | <i>No change</i> | |
| 1 | 1 | Q' | Q | <i>Complement</i> | |

(c) Function table

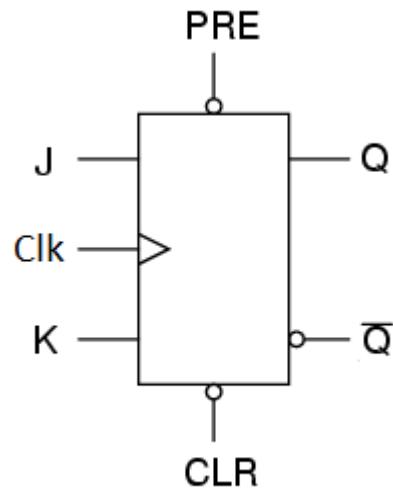
| T | Q_{n+1} |
|-----|-------------|
| 0 | Q_n |
| 1 | \bar{Q}_n |

No change *Complement*

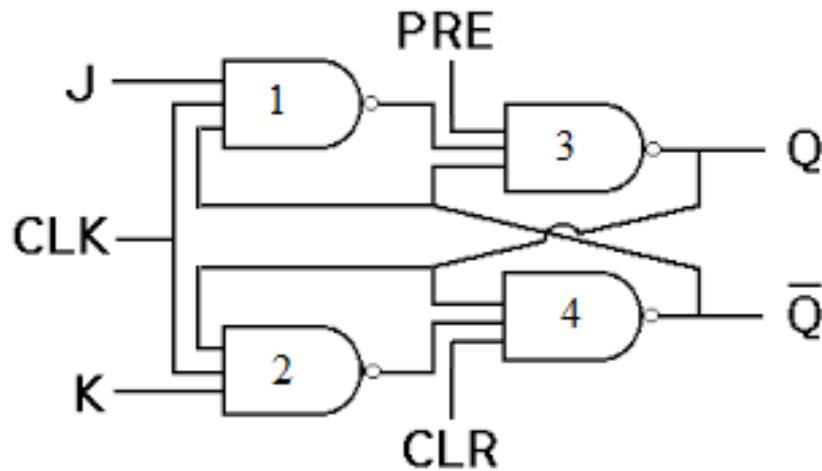
(d) Characteristic table

Direct Inputs

- Some flip-flops have asynchronous inputs that are used to force the flip-flop to a particular state independently of the clock.
- These inputs are called the *preset* (PRE) and *clear* (CLR).
- The input that sets the flip-flop to 1 is called *preset* or *direct set*.
- The input that clears the flip-flop to 0 is called *clear* or *direct reset*.
- When power is turned on in a digital system, the state of the flip-flops is unknown. The direct inputs are useful for bringing all flip-flops in the system to a known starting state prior to the clocked operation.



(a) Graphic symbol



(b) Logic diagram

| <i>PRE</i> | <i>CLR</i> | Q_{n+1} | \bar{Q}_{n+1} | |
|------------|------------|-----------|-----------------|------------------|
| 0 | 0 | <i>X</i> | <i>X</i> | <i>Forbidden</i> |
| 0 | 1 | 1 | 0 | <i>Preset</i> |
| 1 | 0 | 0 | 1 | <i>Clear</i> |
| 1 | 1 | Q_n | \bar{Q}_n | <i>No change</i> |

(c) Function table

- If $\text{PRE} = 0$ and $\text{CLR} = 1$, the output of NAND gate 3 is forced to be 1, i.e., $Q = 1$ and the flip-flop is set, overwriting the previous state of the flip-flop.
- If $\text{PRE} = 1$ and $\text{CLR} = 0$, the output of NAND gate 4 is forced to be 1, i.e., $Q' = 1$ and the flip-flop is reset, overwriting the previous state of the flip-flop.
- Once the state of the flip-flop is established asynchronously, the inputs PRE and CLR must be connected to logic 1 before the next clock is applied.
- The condition $\text{PRE} = \text{CLR} = 0$ must not be applied, since this leads to an uncertain state.

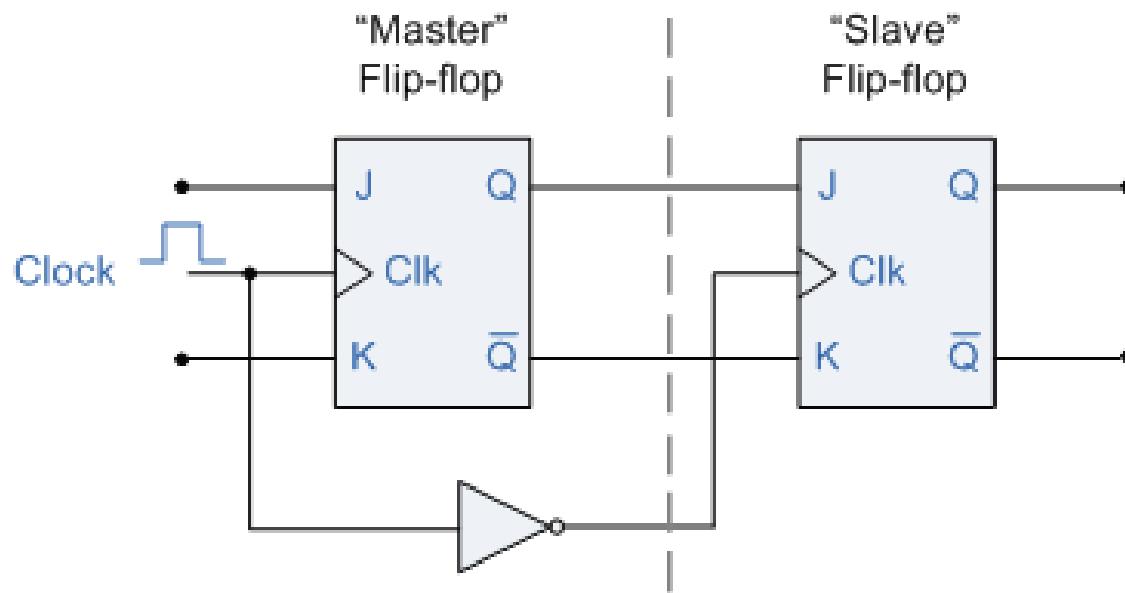
Race Around Condition

- For JK flip-flop, if $J = K = 1$, and if $Clk = 1$ for a long period of time, then Q output will toggle as long as Clk is high, which makes the output of the flip-flop unstable or uncertain. This problem is called *race around condition* in JK flip-flop.
- There are three methods to eliminate race around condition as described below:

- Increasing the delay of flip-flop
 - The propagation delay (Δt) should be made greater than the duration of the clock pulse (T). But it is not a good solution as increasing the delay will decrease the speed of the system.
- Use of edge-triggered flip-flop
 - If the clock is High for a time interval less than the propagation delay of the flip-flop then racing around condition can be eliminated. This is done by using the edge-triggered flip-flop rather than using the level-triggered flip-flop.
- Use of Master-Slave JK flip-flop
 - If the flip-flop is made to toggle over one clock period then racing around condition can be eliminated. This is done by using Master-Slave JK flip-flop.

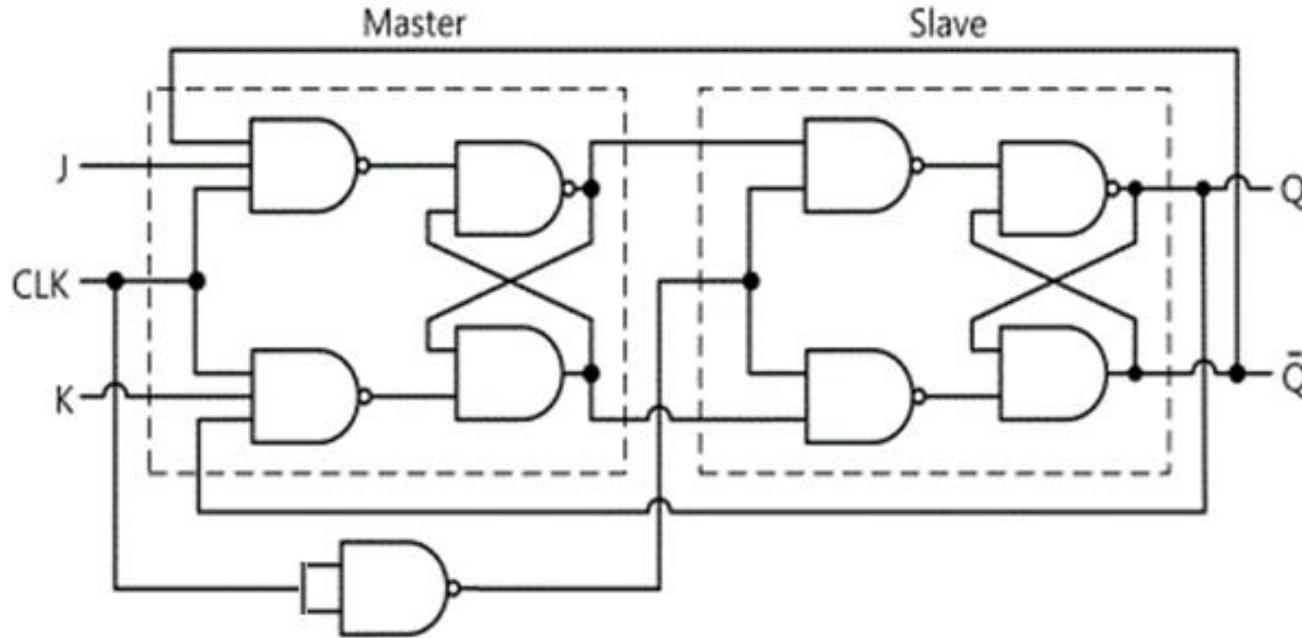
Master-Slave JK Flip-flop

- Master-Slave flip-flop are the cascaded combination of two flip-flops among which the first is designated as master flip-flop while the next is called slave flip-flop.



- Here the master flip-flop is triggered by the external clock pulse train while the slave is activated at its inversion i.e. if the master is positive edge-triggered, then the slave is negative edge-triggered and vice-versa.
- This means that the data enters into the flip-flop at leading/trailing edge of the clock pulse while it is obtained at the output pins during trailing/leading edge of the clock pulse.

- The Master-Slave flip-flop is basically two gated SR flip-flops connected together in a series configuration with the slave having an inverted clock pulse.
- The outputs from Q and Q' from the “Slave” flip-flop are fed back to the inputs of the “Master” with the outputs of the “Master” flip-flop being connected to the two inputs of the “Slave” flip-flop.
- This feedback configuration from the slave’s output to the master’s input gives the characteristic toggle of the JK flip-flop as shown below.



- When C/k is **HIGH**, the first flip-flop (*i.e.*, the master) is enabled and the outputs Q_m and Q'_m respond to the inputs J and K according to the characteristic table. At this time the second flip-flop (*i.e.*, the slave) is disabled because the C/k is **LOW** to the second flip-flop.

- Similarly, when Clk becomes **LOW**, the master becomes disabled and the slave becomes active, since now the Clk to it is HIGH. Therefore, the outputs Q and Q' follow the outputs Q_m and Q'_m respectively.
- Since the second flip-flop just follows the first one, it is referred to as a slave and the first one is called the master.

DIGITAL ELECTRONICS

III Semester CSE & CST



Mr. Srikanta Patnaik

Mr. Ajit Kumar Patro

Mr. Ami Kumar Parida

DEPARTMENT OF ELECTRONICS AND COMMUNICATION ENGINEERING

SCHOOL OF ENGINEERING & TECHNOLOGY

GIET UNIVERSITY, GUNUPUR, ODISHA

Flip-Flop Analysis; Conversion of Flip-Flops

Flip-Flop Analysis

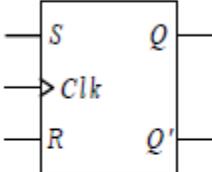
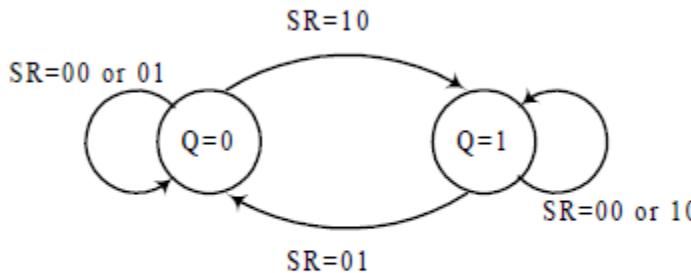
- **Characteristic Table:** A characteristic table defines the logical properties of a flip-flop by describing its operation in tabular form.
- **Characteristic Equation:** The logical properties of a flip-flop, as described in the characteristic table, can be expressed algebraically with a characteristic equation.

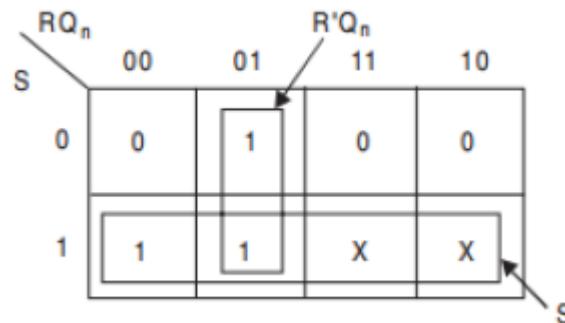
- **State Table:** The time sequence of inputs, outputs, and flip-flop states can be enumerated in a *state table* (sometimes called a *transition table*).
- The table consists of four sections labeled *present state*, *input*, *next state*, and *output*.
- **State Diagram:** The information available in a state table can be represented graphically in the form of a state diagram.

- In this type of diagram, a state is represented by a circle, and the (clock-triggered) transitions between states are indicated by directed lines connecting the circles.
- The binary number inside each circle identifies the state of the flip-flops.
- The directed lines are labeled with two binary numbers separated by a slash.

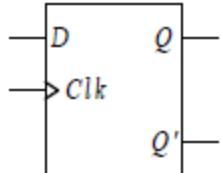
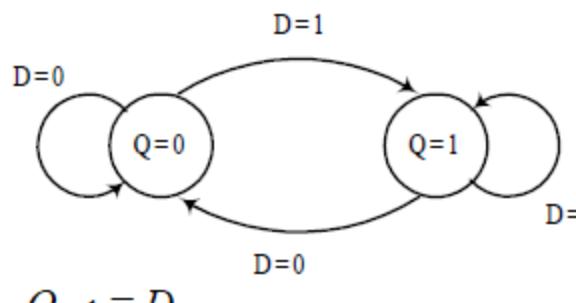
- The input value during the present state is labeled first, and the number after the slash gives the output during the present state with the given input.
- **Excitation Table:** A table that lists the required inputs for a given change of state is called an *excitation table*.

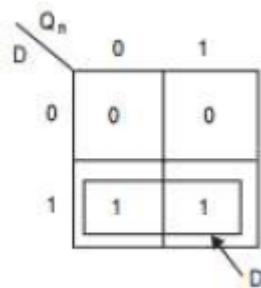
SR Flip-Flop

| Characteristic Table | State Table | State Diagram / Characteristic Equation | Excitation Table |
|---|--|---|---|
|  $\begin{array}{ c c } \hline S & Q \\ \hline \rightarrow Clk & \\ \hline R & Q' \\ \hline \end{array}$ $\begin{array}{ c c } \hline S & R & Q_n & Q_{n+1} \\ \hline \hline 0 & 0 & 0 & 0 \\ 0 & 0 & 1 & 1 \\ 0 & 1 & 0 & 0 \\ 0 & 1 & 1 & 0 \\ \hline \end{array}$ $\begin{array}{ c c } \hline S & R & & Q_{n+1} \\ \hline \hline 0 & 0 & & Q_n \\ 0 & 1 & & 1 \\ 1 & 0 & & 0 \\ 1 & 1 & & X \\ \hline \end{array}$ | $\begin{array}{ c c } \hline S & R & Q_n & Q_{n+1} \\ \hline \hline 0 & 0 & 0 & 0 \\ 0 & 0 & 1 & 1 \\ 0 & 1 & 0 & 0 \\ 0 & 1 & 1 & 0 \\ \hline \end{array}$ $\begin{array}{ c c } \hline S & R & & Q_{n+1} \\ \hline \hline 0 & 0 & & Q_n \\ 0 & 1 & & 1 \\ 1 & 0 & & 0 \\ 1 & 1 & & X \\ \hline \end{array}$ |  <p>The state diagram shows two states: $Q=0$ and $Q=1$. Transitions are as follows:</p> <ul style="list-style-type: none"> From $Q=0$ to $Q=0$: SR=00 or 01 From $Q=0$ to $Q=1$: SR=10 From $Q=1$ to $Q=1$: SR=00 or 10 From $Q=1$ to $Q=0$: SR=01 <p>Characteristic Equation:</p> $Q_{n+1} = S + R'Q_n$ | $\begin{array}{ c c } \hline Q_n & Q_{n+1} & S & R \\ \hline \hline 0 & 0 & 0 & \times \\ 0 & 1 & 1 & 0 \\ 1 & 0 & 0 & 1 \\ 1 & 1 & \times & 0 \\ \hline \end{array}$ |

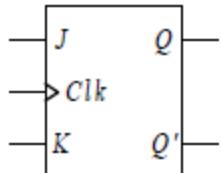
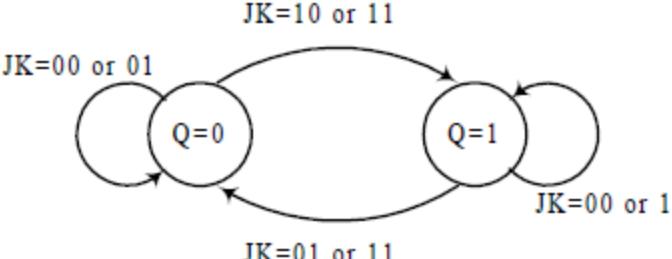


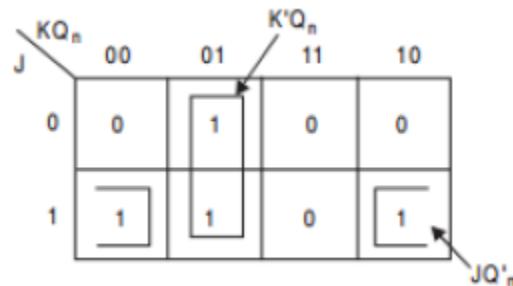
D Flip-Flop

| Characteristic Table | State Table | State Diagram / Characteristic Equation | Excitation Table |
|---|--|--|---|
|  $\begin{array}{ c c } \hline D & Q \\ \hline \text{Clk} & \\ \hline Q' & \\ \hline \end{array}$ $\begin{array}{c c} D & Q_{n+1} \\ \hline 0 & 0 \\ 1 & 1 \\ \hline \end{array}$ | $\begin{array}{c ccc} & D & Q_n & Q_{n+1} \\ \hline 0 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 1 & 0 & 1 & 1 \\ \hline 1 & 1 & 1 & 1 \end{array}$ |  <p>$Q_{n+1} = D$</p> | $\begin{array}{c ccc} & Q_n & Q_{n+1} & D \\ \hline 0 & 0 & 0 & 0 \\ 0 & 1 & 1 & 1 \\ 1 & 0 & 0 & 0 \\ 1 & 1 & 1 & 1 \end{array}$ |

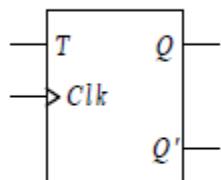
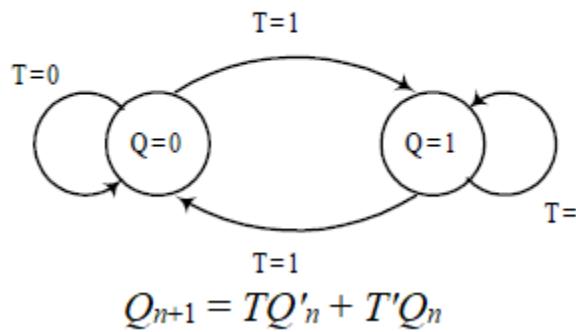


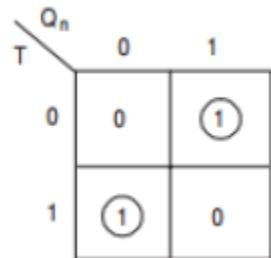
JK Flip-Flop

| Characteristic Table | State Table | State Diagram / Characteristic Equation | Excitation Table |
|---|--|---|---|
|  $\begin{array}{ c c } \hline J & Q \\ \hline \xrightarrow{\text{Clk}} & \\ \hline K & Q' \\ \hline \end{array}$ $\begin{array}{ c c c } \hline J & K & Q_{n+1} \\ \hline \hline 0 & 0 & 0 \\ 0 & 0 & 1 \\ 0 & 1 & 0 \\ 0 & 1 & 1 \\ \hline 1 & 0 & 1 \\ 1 & 1 & 0 \\ \hline \end{array}$ $\begin{array}{ c c } \hline Q_n & \\ \hline 0 & 0 \\ 1 & 1 \\ \hline \end{array}$ $\begin{array}{ c c } \hline Q'_n & \\ \hline 1 & 1 \\ 0 & 0 \\ \hline \end{array}$ | $\begin{array}{cccc} J & K & Q_n & Q_{n+1} \\ \hline \hline 0 & 0 & 0 & 0 \\ 0 & 0 & 1 & 1 \\ 0 & 1 & 0 & 0 \\ 0 & 1 & 1 & 0 \\ \hline 1 & 0 & 0 & 1 \\ 1 & 0 & 1 & 1 \\ 1 & 1 & 0 & 1 \\ 1 & 1 & 1 & 0 \\ \hline \end{array}$ | <p>JK=10 or 11</p>  $Q_{n+1} = JQ'_n + K'Q_n$ | $\begin{array}{cccc} Q_n & Q_{n+1} & J & K \\ \hline \hline 0 & 0 & 0 & \times \\ 0 & 1 & 1 & \times \\ 1 & 0 & \times & 1 \\ 1 & 1 & \times & 0 \\ \hline \end{array}$ |



T Flip-Flop

| Characteristic Table | State Table | State Diagram / Characteristic Equation | Excitation Table |
|---|--|--|---|
|  $\begin{array}{ c c } \hline T & Q \\ \hline \text{Clk} & \\ \hline Q' & \\ \hline \end{array}$ $\begin{array}{c c} T & Q_{n+1} \\ \hline 0 & Q_n \\ 1 & \bar{Q}_n \end{array}$ | $\begin{array}{c ccc} & T & Q_n & Q_{n+1} \\ \hline 0 & 0 & 0 & 0 \\ 0 & 1 & 1 & 1 \\ 1 & 0 & 1 & 0 \end{array}$ |  $Q_{n+1} = TQ'_n + T'Q_n$ | $\begin{array}{c ccc} & Q_n & Q_{n+1} & T \\ \hline 0 & 0 & 0 & 0 \\ 0 & 1 & 1 & 1 \\ 1 & 0 & 1 & 1 \\ 1 & 1 & 0 & 0 \end{array}$ |



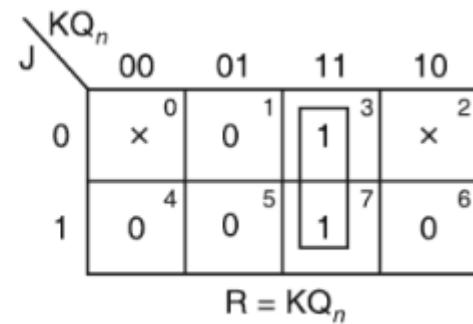
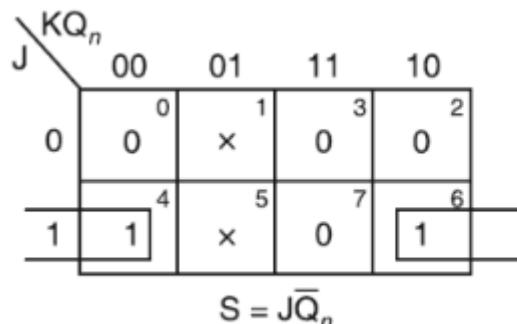
Conversion of Flip-Flops

- To convert one type of flip-flop into another type, a combinational circuit is designed such that if the inputs of the required flip-flop are fed as inputs to the combinational circuit and the output of the combinational circuit is connected to the inputs of the actual flip-flop, then the output of the actual flip-flop is the output of the required flip-flop.

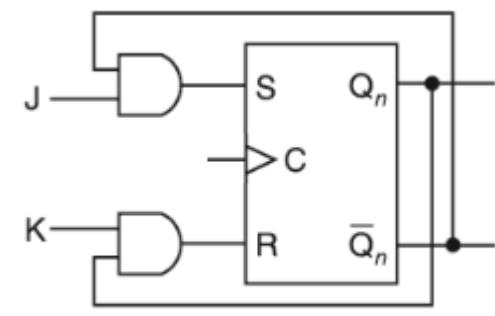
Conversion of an S-R Flip-flop to a J-K Flip-flop.

| External Inputs | | Present State | Next State | Flip-flop Inputs | |
|-----------------|---|---------------|------------|------------------|---|
| J | K | Q_n | Q_{n+1} | S | R |
| 0 | 0 | 0 | 0 | 0 | x |
| 0 | 0 | 1 | 1 | x | 0 |
| 0 | 1 | 0 | 0 | 0 | x |
| 0 | 1 | 1 | 0 | 0 | 1 |
| 1 | 0 | 0 | 1 | 1 | 0 |
| 1 | 0 | 1 | 1 | x | 0 |
| 1 | 1 | 0 | 1 | 1 | 0 |
| 1 | 1 | 1 | 0 | 0 | 1 |

(a) Conversion table



(b) K-maps for S and R

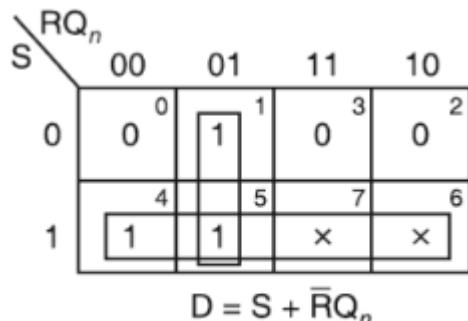


(c) Logic diagram

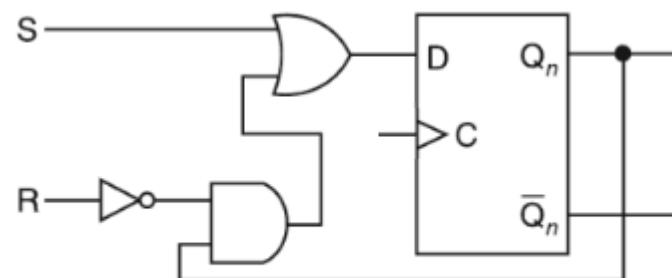
Conversion of an D Flip-flop to a S-R Flip-flop.

| External Inputs | | Present State | Next State | Flip-flop Input |
|-----------------|---|---------------|------------|-----------------|
| S | R | Q_n | Q_{n+1} | D |
| 0 | 0 | 0 | 0 | 0 |
| 0 | 0 | 1 | 1 | 1 |
| 0 | 1 | 0 | 0 | 0 |
| 0 | 1 | 1 | 0 | 0 |
| 1 | 0 | 0 | 1 | 1 |
| 1 | 0 | 1 | 1 | 1 |
| 1 | 1 | 0 | x | x |
| 1 | 1 | 1 | x | x |

(a) Conversion table



(b) K-map for D

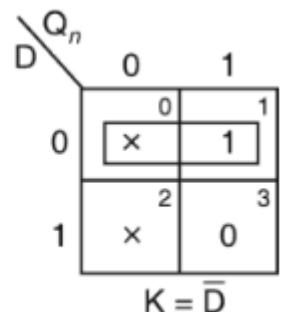
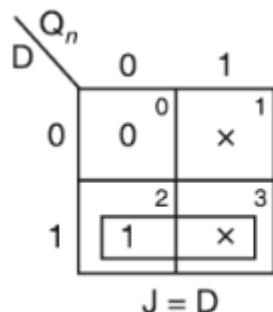


(c) Logic diagram

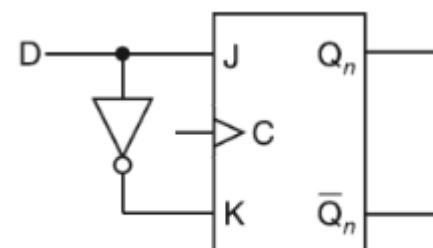
Conversion of an J-K Flip-flop to a D Flip-flop.

| External Input | Present State | Next State | Flip-flop Inputs | |
|----------------|---------------|------------|------------------|---|
| D | Q_n | Q_{n+1} | J | K |
| 0 | 0 | 0 | 0 | x |
| 0 | 1 | 0 | x | 1 |
| 1 | 0 | 1 | 1 | x |
| 1 | 1 | 1 | x | 0 |

(a) Conversion table



(b) K-maps for J and K

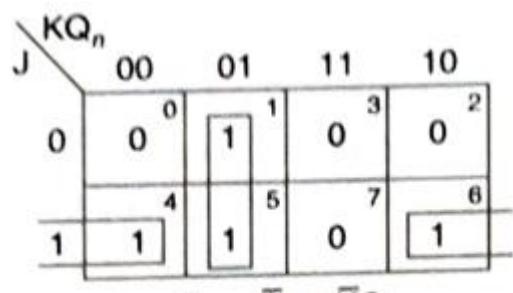


(c) Logic diagram

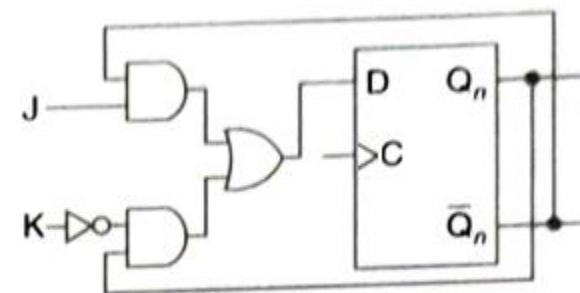
Conversion of an D Flip-flop to a J-K Flip-flop.

| External Inputs | | Present State | Next State | Flip-flop Input |
|-----------------|---|---------------|------------|-----------------|
| J | K | Q_n | Q_{n+1} | D |
| 0 | 0 | 0 | 0 | 0 |
| 0 | 0 | 1 | 1 | 1 |
| 0 | 1 | 0 | 0 | 0 |
| 0 | 1 | 1 | 0 | 0 |
| 1 | 0 | 0 | 1 | 1 |
| 1 | 0 | 1 | 1 | 1 |
| 1 | 1 | 0 | 1 | 1 |
| 1 | 1 | 1 | 0 | 0 |

(a) Conversion table



(b) K-maps for D



(c) Logic diagram

Problem

- A PN flip-flop has four operations: clear to 0, no change, complement, and set to 1, when inputs P and N are 00, 01, 10, and 11, respectively.
 - (a) Tabulate the characteristic table.
 - (b) Derive the characteristic equation.
 - (c) Tabulate the excitation table.
 - (d) Show how the PN flip-flop can be converted to a D flip-flop.

DIGITAL ELECTRONICS

III Semester CSE & CST



Mr. Srikanta Patnaik Mr. Ajit Kumar Patro Mr. Ami Kumar Parida

DEPARTMENT OF ELECTRONICS AND COMMUNICATION ENGINEERING

SCHOOL OF ENGINEERING & TECHNOLOGY

GIET UNIVERSITY, GUNUPUR, ODISHA

Analysis of Clocked Sequential Circuits; Mealy and Moore Models of Finite State Machines; Design Procedure

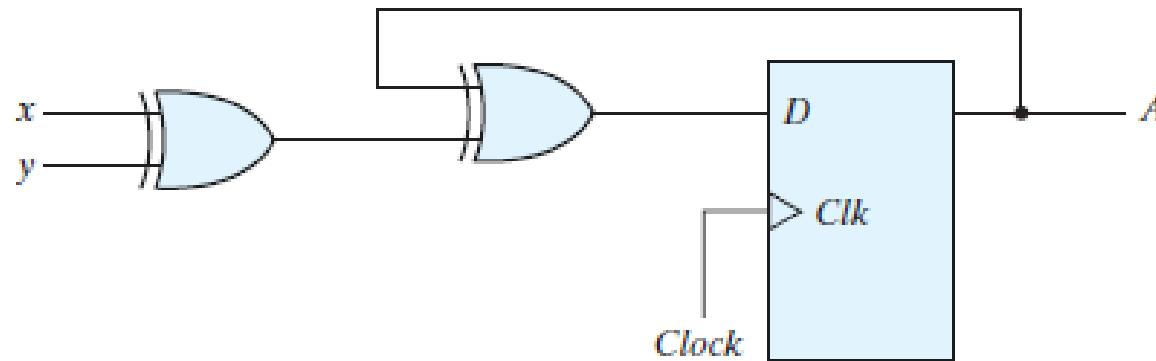
Analysis of Clocked Sequential Circuits

- Analysis describes what a given circuit will do under certain operating conditions.
- The behavior of a clocked sequential circuit is determined from the inputs, the outputs, and the state of its flip-flops.
- The outputs and the next state are both a function of the inputs and the present state.
- The analysis of a sequential circuit consists of obtaining a table or a diagram for the time sequence of inputs, outputs, and internal states.
- It is also possible to write Boolean expressions that describe the behavior of the sequential circuit.

- For analysis of clocked sequential circuits, the steps are described below:
 - Circuit diagram
 - State equations
 - State table
 - State diagram

- The next-state values can also be obtained by evaluating the state equations from the characteristic equation. This is done by using the following procedure:
 1. Determine the flip-flop input equations in terms of the present state and input variables.
 2. Substitute the input equations into the flip-flop characteristic equation to obtain the state equations.
 3. Use the corresponding state equations to determine the next-state values in the state table.

Example: Derive the state table and the state diagram of the sequential circuit shown in Fig.

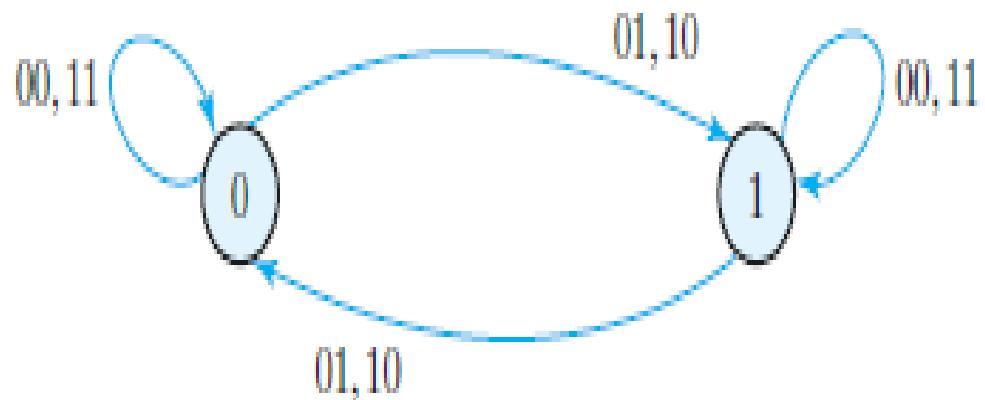


- It has one D flip-flop A , and two input x and y .
- The input equation $D_A = A \oplus x \oplus y$
- The characteristic equation of a D flip-flop
 - $$Q(t + 1) = D$$
- The next-state values are obtained from the state equation
 - $$A(t + 1) = D_A = A \oplus x \oplus y$$

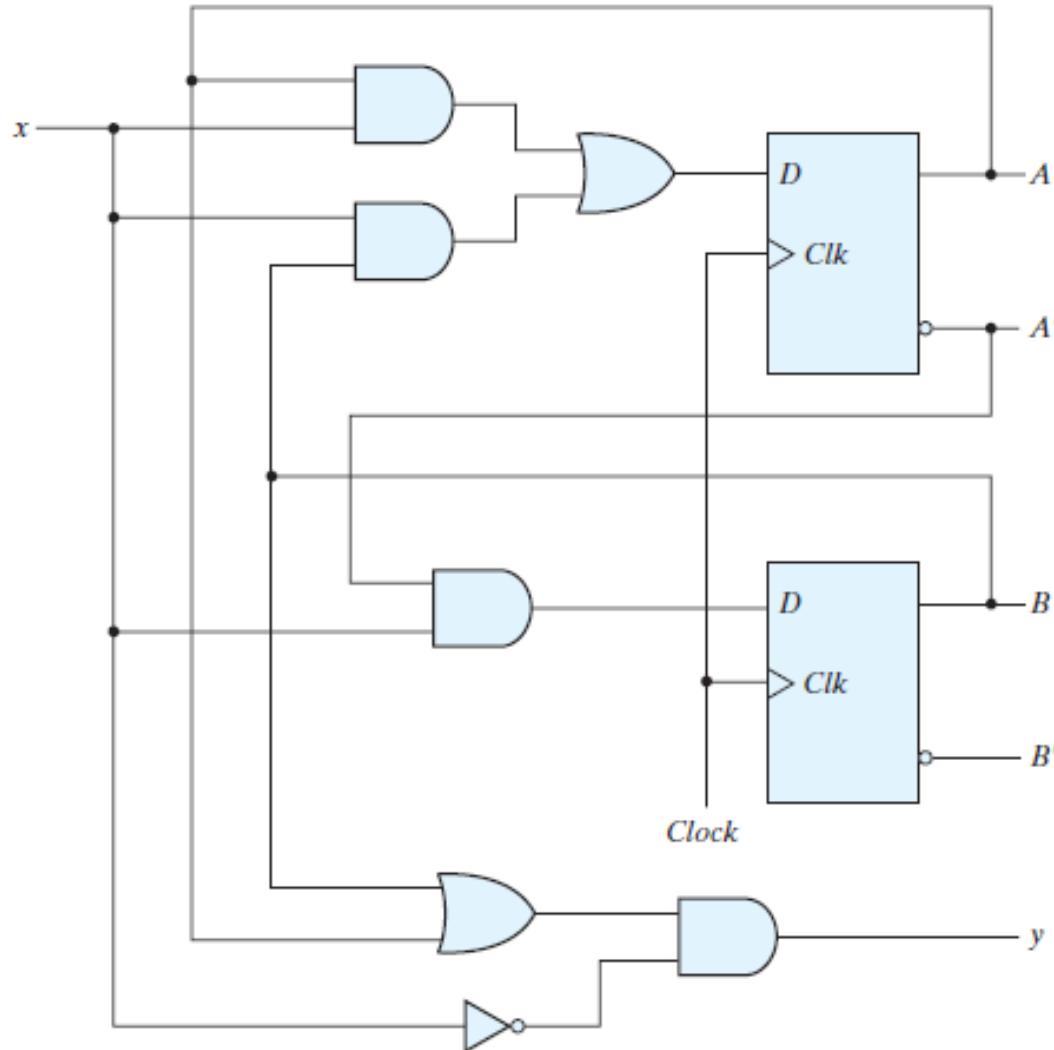
State table

| Present state | Inputs | | Next state |
|---------------|--------|-----|------------|
| A | x | y | A |
| 0 | 0 | 0 | 0 |
| 0 | 0 | 1 | 1 |
| 0 | 1 | 0 | 1 |
| 0 | 1 | 1 | 0 |
| 1 | 0 | 0 | 1 |
| 1 | 0 | 1 | 0 |
| 1 | 1 | 0 | 0 |
| 1 | 1 | 1 | 1 |

State diagram



Example: Derive the state table and the state diagram of the sequential circuit shown in Fig.



- It consists of two D flip-flops A and B , an input x and an output y .
- The circuit can be specified by the flip-flop input equations:

$$D_A = Ax + Bx \quad D_B = A'x$$

- The out equation: $y = (A + B)x'$
- The characteristic equation of a D flip-flop

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

- The next-state values are obtained from the state equation

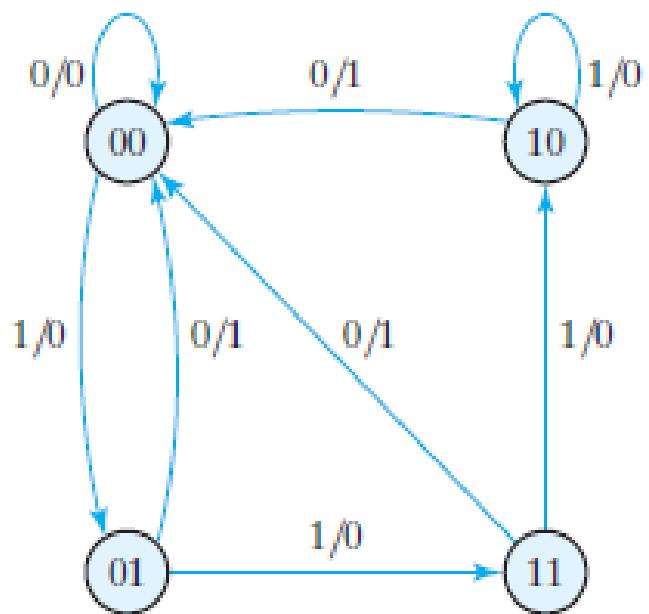
$$A(t + 1) = D_A = Ax + Bx$$

$$B(t + 1) = D_B = A'x$$

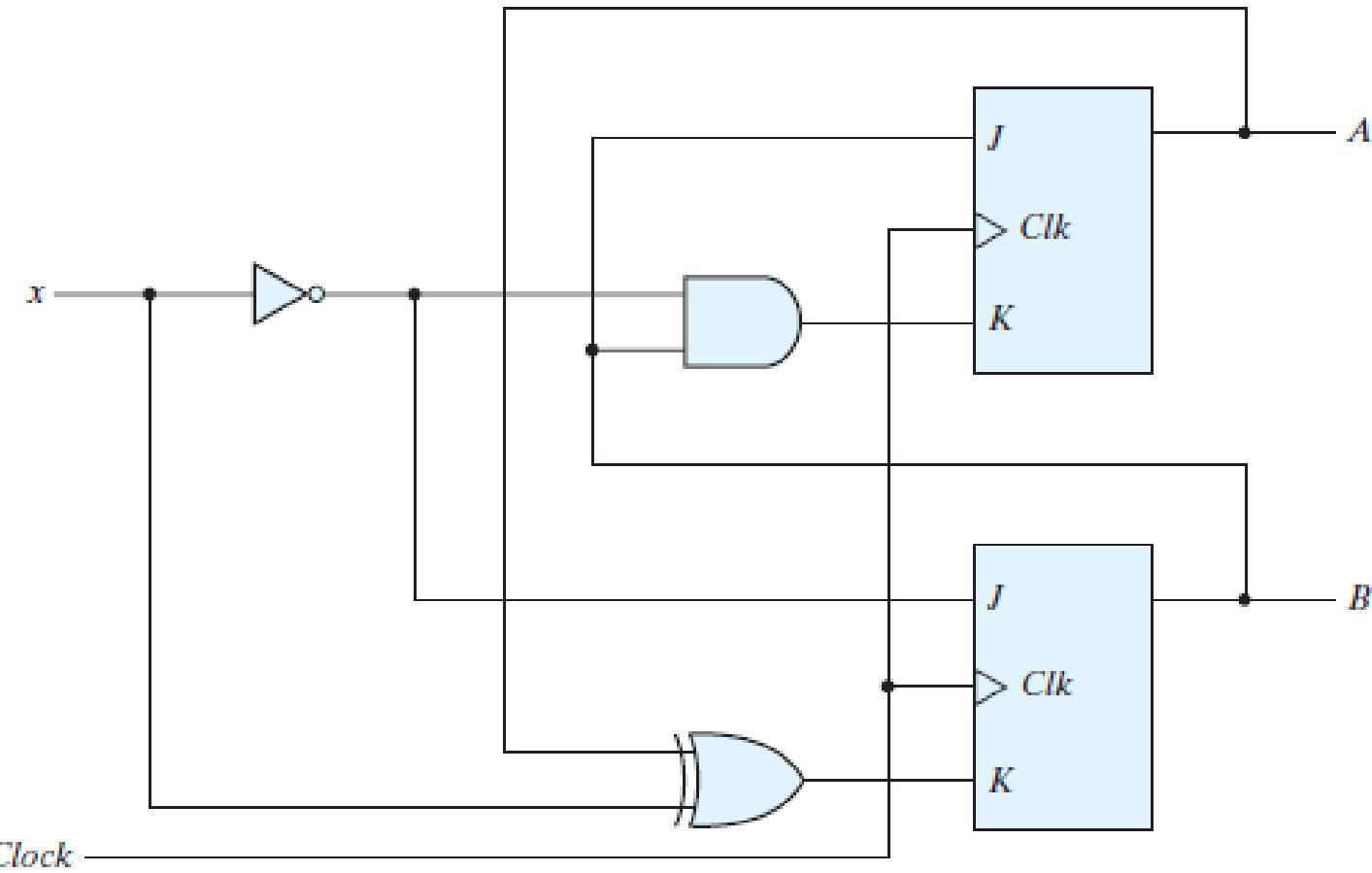
State table

| Present State | | Input <i>x</i> | Next State | | Output <i>y</i> |
|---------------|----------|-------------------|------------|----------|--------------------|
| <i>A</i> | <i>B</i> | | <i>A</i> | <i>B</i> | |
| 0 | 0 | 0 | 0 | 0 | 0 |
| 0 | 0 | 1 | 0 | 1 | 0 |
| 0 | 1 | 0 | 0 | 0 | 1 |
| 0 | 1 | 1 | 1 | 1 | 0 |
| 1 | 0 | 0 | 0 | 0 | 1 |
| 1 | 0 | 1 | 1 | 0 | 0 |
| 1 | 1 | 0 | 0 | 0 | 1 |
| 1 | 1 | 1 | 1 | 0 | 0 |

State diagram



Example: Derive the state table and the state diagram of the sequential circuit shown in Fig.



- It has two JK flip-flops A and B and one input x .
- The circuit can be specified by the flip-flop input equations

$$J_A = B \quad K_A = Bx'$$

$$J_B = x \quad K_B = A \oplus x$$

- The characteristic equation of a JK flip-flop

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

- The characteristic equations for the flip-flops are obtained by substituting A or B for the name of the flip-flop, instead of Q :

$$A(t + 1) = J_A A' + K_A' A$$

$$B(t + 1) = J_B B' + K_B' B$$

- Substituting the values of J_A and K_A from the input equations, we obtain the state equation for A :

$$A(t + 1) = BA' + (Bx')'A = A'B + AB' + Ax$$

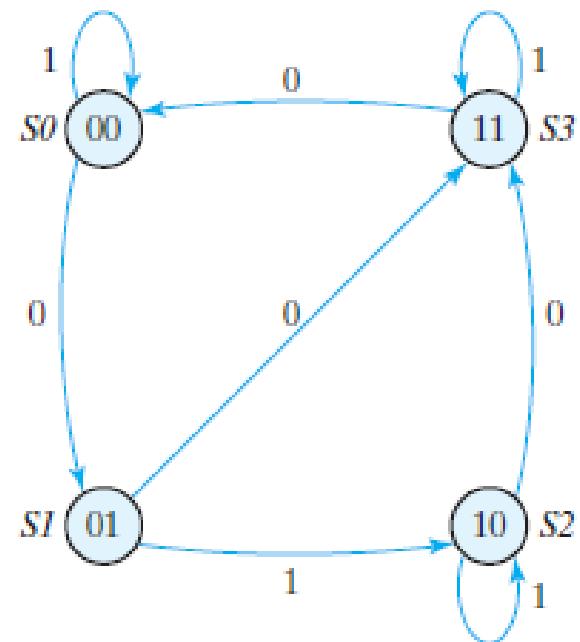
- Similarly, the state equation for flip-flop B can be derived from the characteristic equation by substituting the values of J_B and K_B :

$$B(t + 1) = x'B' + (A \oplus x)'B = B'x' + ABx + A'Bx'$$

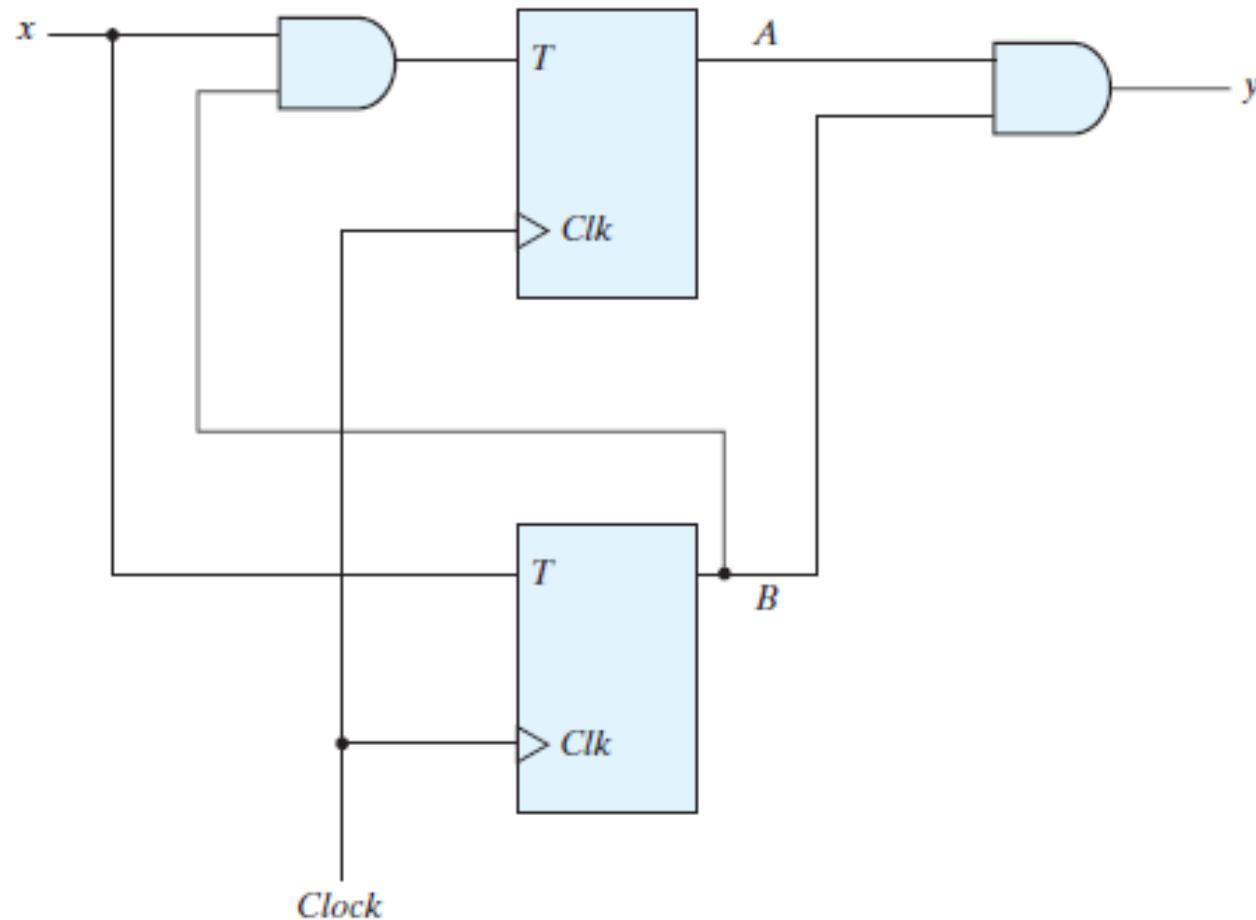
State table

| Present State | | Input | Next State | |
|---------------|---|-------|------------|---|
| A | B | x | A | B |
| 0 | 0 | 0 | 0 | 1 |
| 0 | 0 | 1 | 0 | 0 |
| 0 | 1 | 0 | 1 | 1 |
| 0 | 1 | 1 | 1 | 0 |
| 1 | 0 | 0 | 1 | 1 |
| 1 | 0 | 1 | 1 | 0 |
| 1 | 1 | 0 | 0 | 0 |
| 1 | 1 | 1 | 1 | 1 |

State diagram



Example: Derive the state table and the state diagram of the sequential circuit shown in Fig.



- It has two T flip-flops A and B , one input x , and one output y and can be described algebraically by two input equations and an output equation:

$$T_A = Bx \quad T_B = x$$

$$y = AB$$

- The characteristic equation of T flip-flop

$$Q(t + 1) = T \oplus Q = T'Q + TQ'$$

- The values for the next state can be derived from the state equations by substituting T_A and T_B in the characteristic equations, yielding

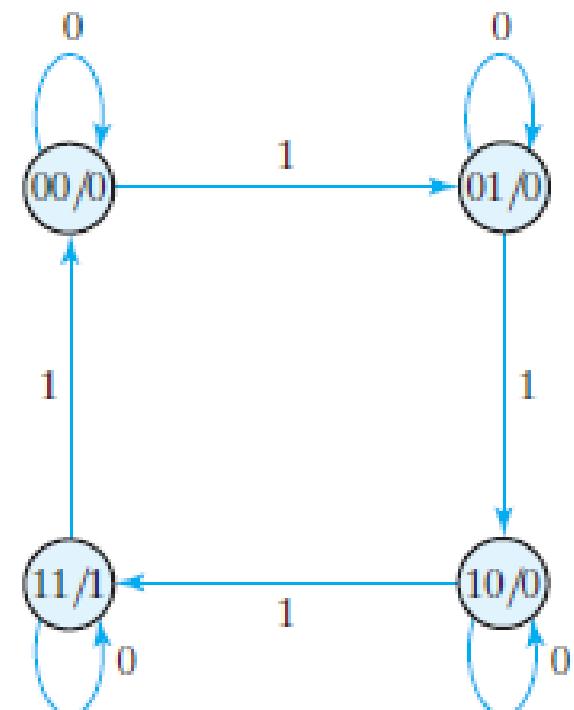
$$A(t + 1) = T_A'A + T_A A' = (Bx)'A + (Bx)A' = AB' + Ax' + A'Bx$$

$$B(t + 1) = T_B'B + T_B B' = x'B + xB' = x \oplus B$$

State table

| Present State | | Input <i>x</i> | Next State | | Output <i>y</i> |
|---------------|----------|-------------------|------------|----------|--------------------|
| <i>A</i> | <i>B</i> | | <i>A</i> | <i>B</i> | |
| 0 | 0 | 0 | 0 | 0 | 0 |
| 0 | 0 | 1 | 0 | 1 | 0 |
| 0 | 1 | 0 | 0 | 1 | 0 |
| 0 | 1 | 1 | 1 | 0 | 0 |
| 1 | 0 | 0 | 1 | 0 | 0 |
| 1 | 0 | 1 | 1 | 1 | 0 |
| 1 | 1 | 0 | 1 | 1 | 1 |
| 1 | 1 | 1 | 0 | 0 | 1 |

State diagram



Problems

- A sequential circuit with two D flip-flops A and B , two inputs, x and y ; and one output z is specified by the following next-state and output equations:

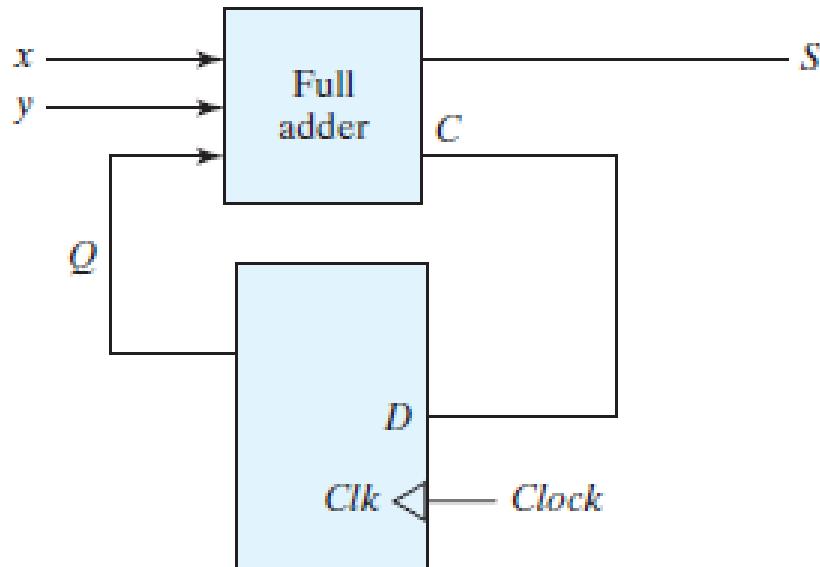
$$A(t + 1) = xy' + xB$$

$$B(t + 1) = xA + xB'$$

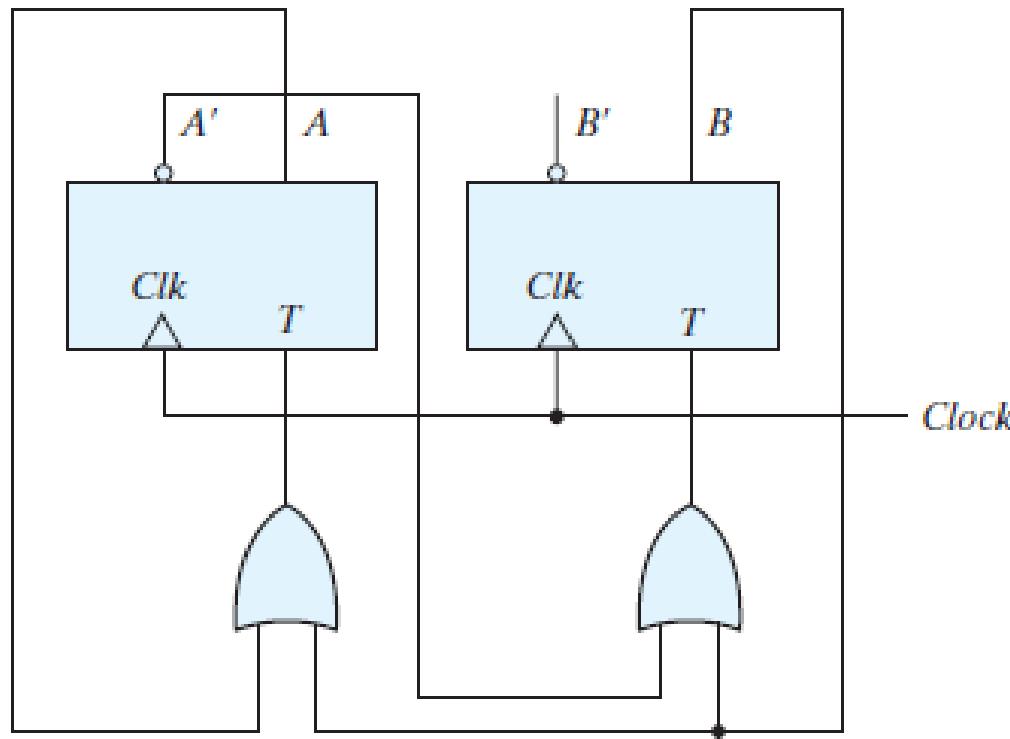
$$z = A$$

- (a) Draw the logic diagram of the circuit.
- (b) List the state table for the sequential circuit.
- (c) Draw the corresponding state diagram.

- A sequential circuit has one flip-flop Q , two inputs x and y , and one output S . It consists of a full-adder circuit connected to a D flip-flop, as shown in Fig. Derive the state table and state diagram of the sequential circuit.



- Derive the state table and the state diagram of the sequential circuit shown in Fig.



- A sequential circuit has two JK flip-flops A and B and one input x . The circuit is described by the following flip-flop input equations:

$$\begin{array}{ll} J_A = x & K_A = B \\ J_B = x & K_B = A' \end{array}$$

- (a) Derive the state equations $A(t + 1)$ and $B(t + 1)$ by substituting the input equations for the J and K variables.
- (b) Draw the state diagram of the circuit.

- A sequential circuit has two JK flip-flops A and B , two inputs x and y , and one output z . The flip-flop input equations and circuit output equation are

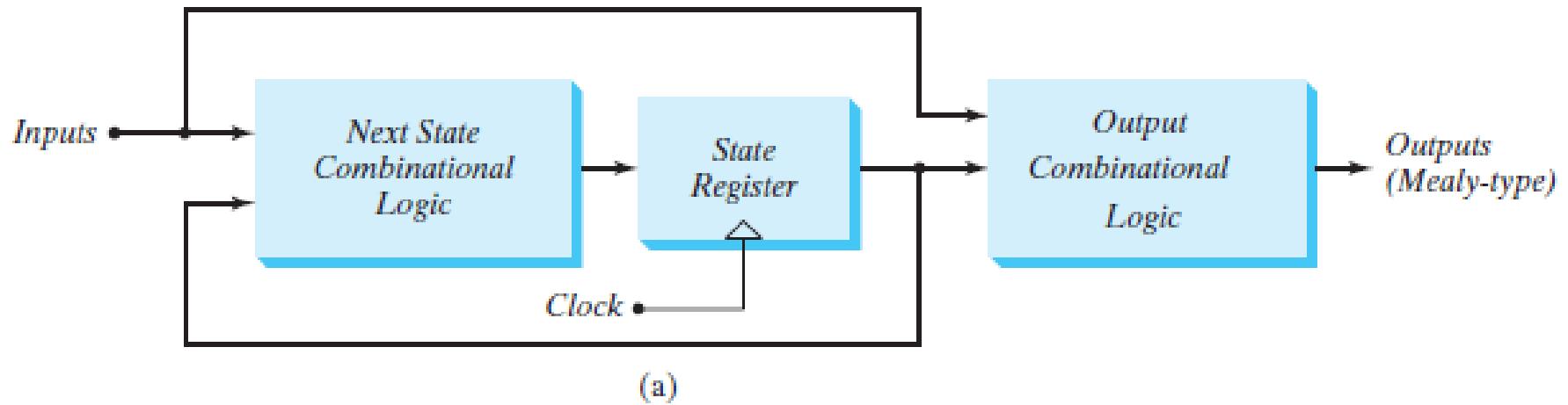
$$\begin{array}{ll} J_A = Bx + B'y' & K_A = B'xy' \\ J_B = A'x & K_B = A + xy' \\ z = Ax'y' + Bx'y' \end{array}$$

- (a) Draw the logic diagram of the circuit.
- (b) Tabulate the state table.
- (c) Derive the state equations for A and B .

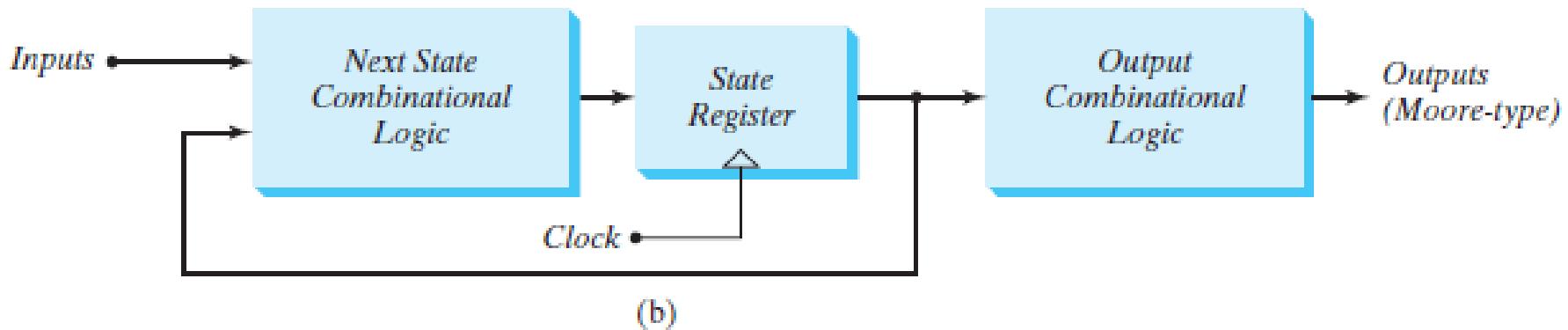
Mealy and Moore Models of Finite State Machines

- The most general model of a sequential circuit has inputs, outputs, and internal states.
- It is customary to distinguish between two models of sequential circuits: the **Mealy model** and the **Moore model**.
- They differ only in the way the output is generated.
- In the **Mealy model**, the output is a function of both the present state and the input.
- In the **Moore model**, the output is a function of only the present state.

Mealy Machine



Moore Machine



Block diagrams of Mealy and Moore state machines

- A circuit may have both types of outputs.
- The two models of a sequential circuit are commonly referred to as a finite state machine, abbreviated FSM.
 - The Mealy model of a sequential circuit is referred to as a Mealy FSM or Mealy machine.
 - The Moore model is referred to as a Moore FSM or Moore machine.

- In **Mealy model** state diagram, both the input and output values separated by a slash along the directed lines between the states.
- In **Moore model** state diagram, the input value in the state diagram is labeled along the directed line, but the output value is indicated inside the circle together with the present state.

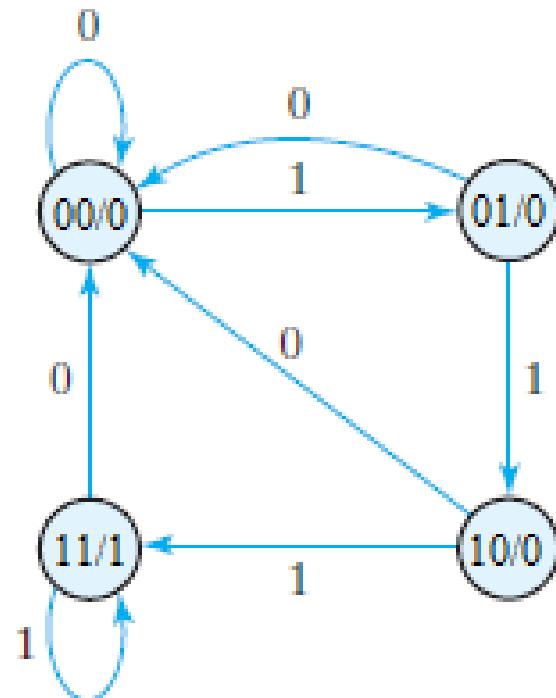
- In a **Moore model**, the outputs of the sequential circuit are synchronized with the clock, because they depend only on flip-flop outputs that are synchronized with the clock.
- In a **Mealy model**, the outputs may change if the inputs change during the clock cycle. Thus, the output of the Mealy machine is the value that is present immediately before the active edge of the clock.

Design Procedure

- Design procedures or methodologies specify hardware that will implement a desired behavior.
- The design of a clocked sequential circuit starts from a set of specifications and culminates in a logic diagram or a list of Boolean functions from which the logic diagram can be obtained.
- A synchronous sequential circuit is made up of flip-flops and combinational gates.

- The procedure for designing synchronous sequential circuits can be summarized by a list of recommended steps:
 1. From the word description and specifications of the desired operation, derive a state diagram for the circuit.
 2. Reduce the number of states if necessary.
 3. Assign binary values to the states.
 4. Obtain the binary-coded state table.
 5. Choose the type of flip-flops to be used.
 6. Derive the simplified flip-flop input equations and output equations.
 7. Draw the logic diagram.

Example: Design the sequential circuit specified by the state diagram of Fig., using D flip-flops.



- We choose two D flip-flops to represent the four states, and we label their outputs A and B . There is one input x and one output y .

State Table

- The flip-flop input equations D_A and D_B can be obtained from the columns flip-flop inputs in the state table and expressed in sum-of-minterms form as

$$D_A(A, B, x) = \sum(3, 5, 7)$$

$$D_B(A, B, x) = \sum(1, 5, 7)$$

$$y(A, B, x) = \sum(6, 7)$$

where A and B are the present-state values of flip-flops A and B , and x is the input. The minterms for output y are obtained from the output column in the state table.

Maps for D input and output y equations

| | | Bx | 00 | 01 | 11 | 10 |
|--|--|------|-------|-------|-------|-------|
| | | A | m_0 | m_1 | m_3 | m_2 |
| | | 0 | 1 | | | |
| | | 1 | | 1 | 1 | |

$$D_A = Ax + Bx$$

| | | Bx | 00 | 01 | 11 | 10 |
|--|--|------|-------|-------|-------|-------|
| | | A | m_0 | m_1 | m_3 | m_2 |
| | | 0 | 1 | | | |
| | | 1 | | 1 | 1 | |

$$D_B = Ax + B'x$$

| | | Bx | 00 | 01 | 11 | 10 |
|--|--|------|-------|-------|-------|-------|
| | | A | m_0 | m_1 | m_3 | m_2 |
| | | 0 | | | | |
| | | 1 | | 1 | 1 | 1 |

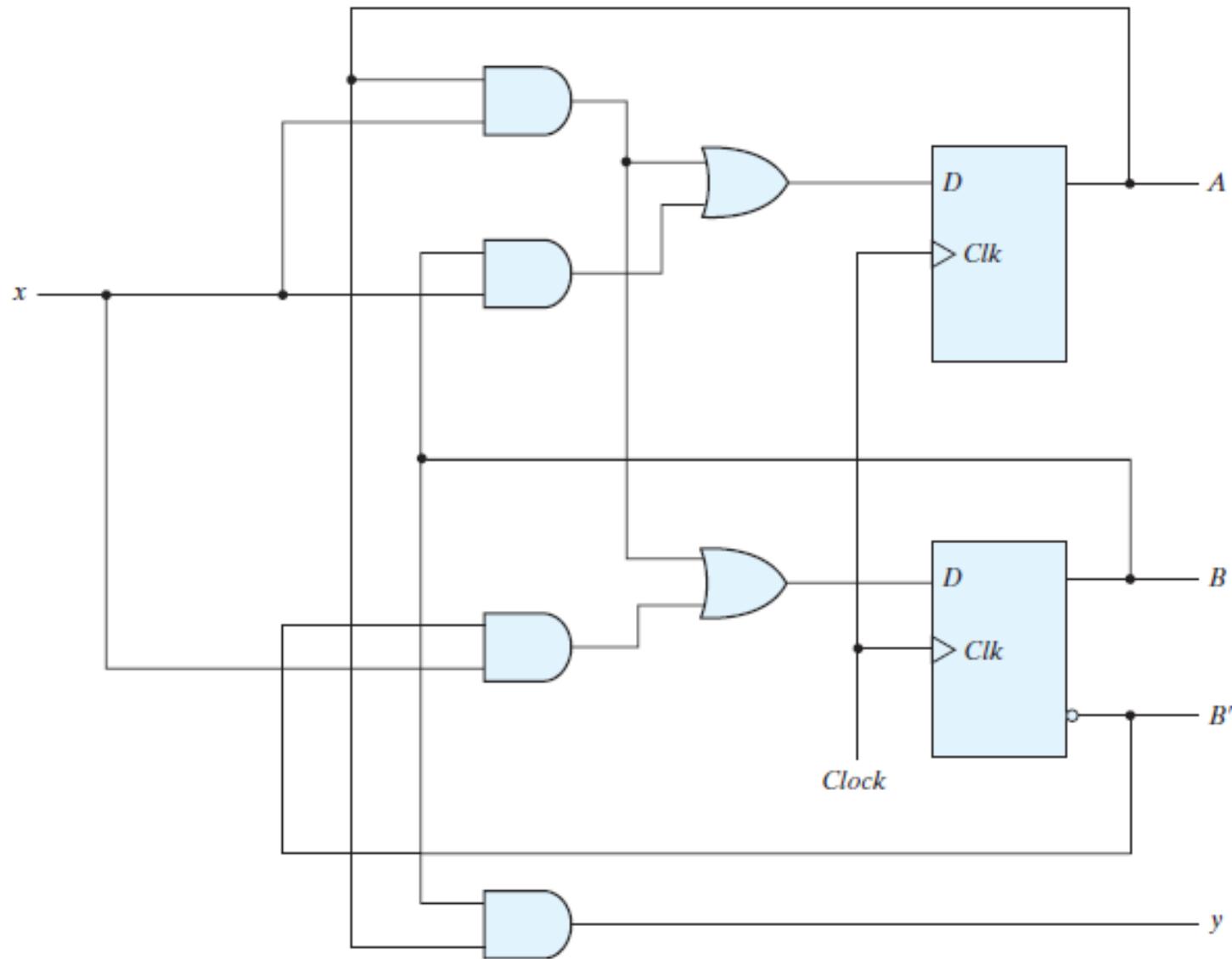
$$y = AB$$

- The simplified equations are

$$D_A = Ax + Bx$$

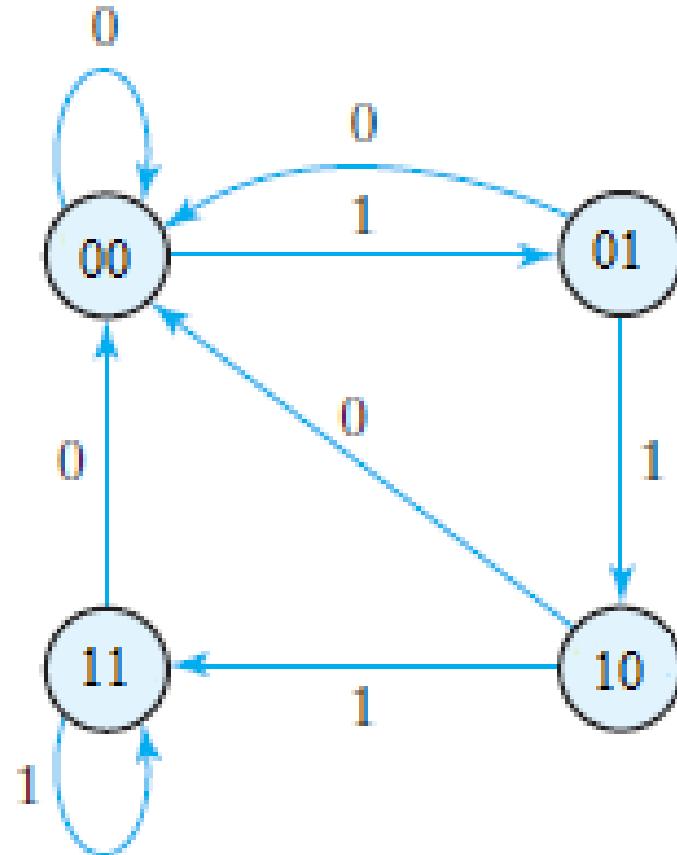
$$D_B = Ax + B'x$$

$$y = AB$$



Logic diagram of the sequential circuit with *D* flip-flops

Example: Design the sequential circuit specified by the state diagram of Fig., using *JK* flip-flops.



State Table and JK Flip-Flop Inputs

| Present State | | Input | Next State | | Flip-Flop Inputs | | | |
|---------------|---|-------|------------|---|------------------|-------|-------|-------|
| A | B | x | A | B | J_A | K_A | J_B | K_B |
| 0 | 0 | 0 | 0 | 0 | 0 | X | 0 | X |
| 0 | 0 | 1 | 0 | 1 | 0 | X | 1 | X |
| 0 | 1 | 0 | 1 | 0 | 1 | X | X | 1 |
| 0 | 1 | 1 | 0 | 1 | 0 | X | X | 0 |
| 1 | 0 | 0 | 1 | 0 | X | 0 | 0 | X |
| 1 | 0 | 1 | 1 | 1 | X | 0 | 1 | X |
| 1 | 1 | 0 | 1 | 1 | X | 0 | X | 0 |
| 1 | 1 | 1 | 0 | 0 | X | 1 | X | 1 |

Maps for J and K input equations

| | | 00 | 01 | 11 | 10 |
|---|-------|-------|-------|-------|-------|
| | | m_0 | m_1 | m_3 | m_2 |
| | | 0 | | | 1 |
| 1 | m_4 | X | X | X | X |
| | | | | | |

$$J_A = Bx'$$

| | | 00 | 01 | 11 | 10 |
|---|-------|-------|-------|-------|-------|
| | | m_0 | m_1 | m_3 | m_2 |
| | | 0 | X | X | X |
| 1 | m_4 | | | 1 | |
| | | | | | |

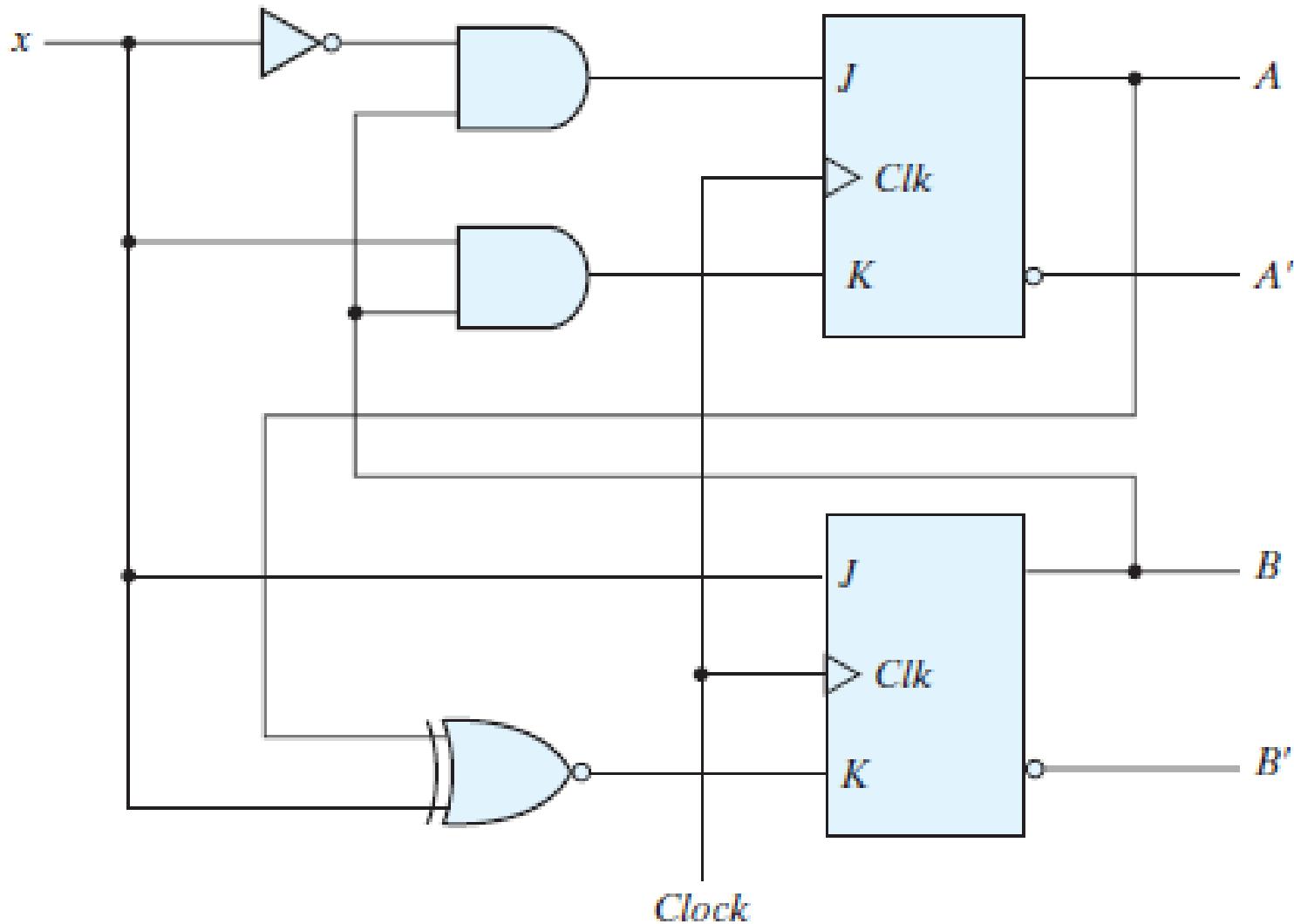
$$K_A = Bx$$

| | | 00 | 01 | 11 | 10 |
|---|-------|-------|-------|-------|-------|
| | | m_0 | m_1 | m_3 | m_2 |
| | | 0 | 1 | X | X |
| 1 | m_4 | | 1 | X | X |
| | | | | | |

$$J_B = x$$

| | | 00 | 01 | 11 | 10 |
|---|-------|-------|-------|-------|-------|
| | | m_0 | m_1 | m_3 | m_2 |
| | | 0 | X | X | |
| 1 | m_4 | | | 1 | |
| | | | | | |

$$K_B = (A \oplus x)'$$



Logic diagram of the sequential circuit with *JK* flip-flops

Problem

- Design a sequential circuit with two D flip-flops A and B , and one input x . When $x = 0$, the state of the circuit remains the same. When $x = 1$, the circuit goes through the state transitions from 00 to 01, to 11, to 10, back to 00, and repeats.

DIGITAL ELECTRONICS

III Semester CSE & CST



Mr. Srikanta Patnaik

Mr. Ajit Kumar Patro

Mr. Ami Kumar Parida

DEPARTMENT OF ELECTRONICS AND COMMUNICATION ENGINEERING

SCHOOL OF ENGINEERING & TECHNOLOGY

GIET UNIVERSITY, GUNUPUR, ODISHA

Registers and Counters

Shift Registers;
Data Transmission in Shift Registers;
SISO, SIPO, PISO, and PIPO Shift Registers

Registers

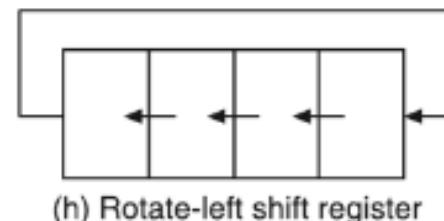
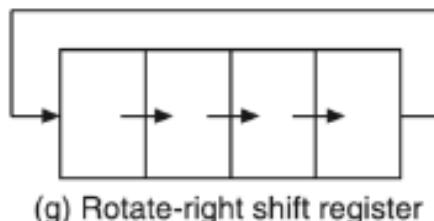
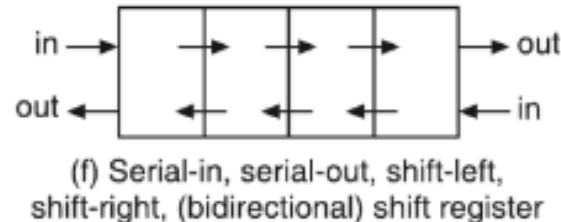
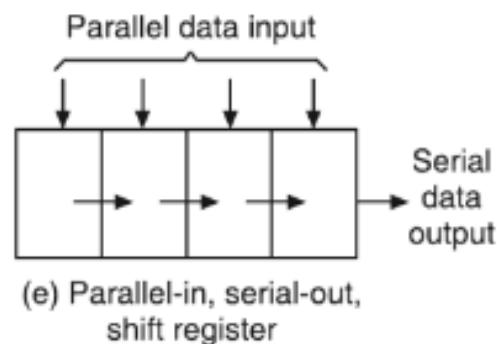
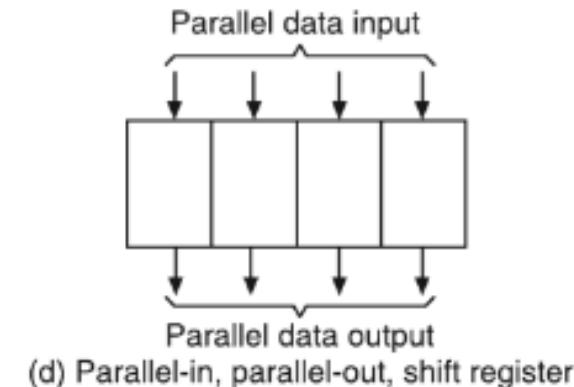
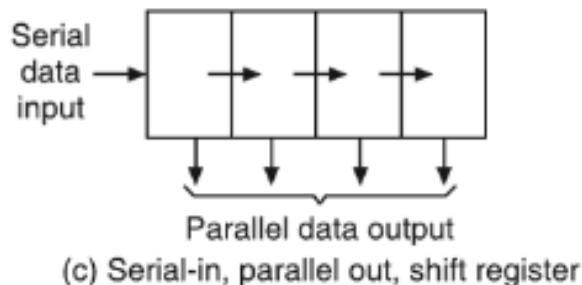
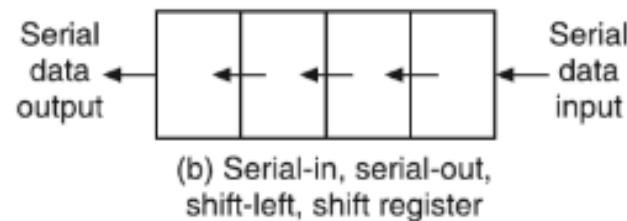
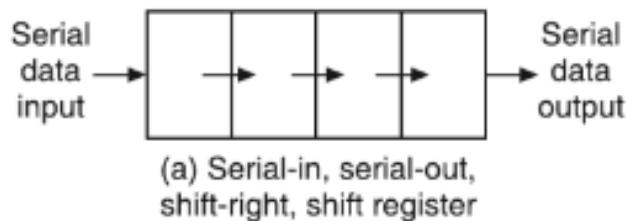
- A *register* is a group of flip-flops, each one of which shares a common clock and is capable of storing one bit of information.
- An n -bit register consists of a group of n flip-flops capable of storing n bits of binary information.

Shift Registers

- A register capable of shifting the binary information held in each cell to its neighboring cell, in a selected direction, is called a *shift register*.
- The logical configuration of a shift register consists of a chain of flip-flops in cascade, with the output of one flip-flop connected to the input of the next flip-flop.
- All flip-flops receive common clock pulses, which activate the shift of data from one stage to the next.

Data Transmission in Shift Registers

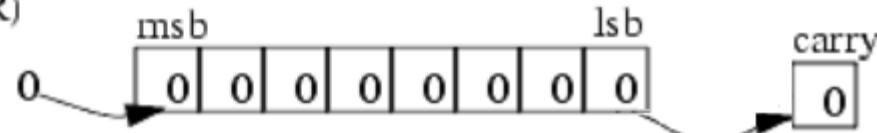
- Data may be shifted into or out of the register either in serial form or in parallel form.
- So there are four basic types of shift registers:
 - Serial-in, Serial-out (SISO)
 - Serial-in, Parallel-out (SIPO)
 - Parallel-in, Serial-out (PISO)
 - Parallel-in, Parallel-out (PIPO)
- The process of data shifting in these registers is illustrated in Figure.



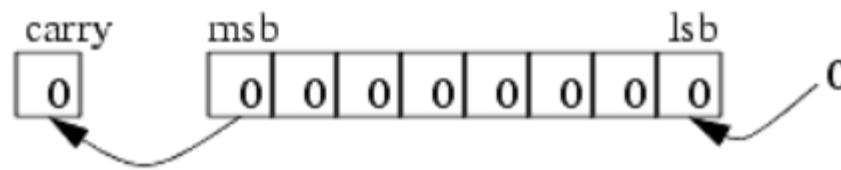
Data transfer in registers.

- Shift register has two shift operations that are controlled by the signals *SHR* (Shift Right) and *SHL* (Shift Left).
- In addition to the shift operations, a register may also have the capability of the rotate operations *ROR* (Rotate Right) and *ROL* (Rotate Left).

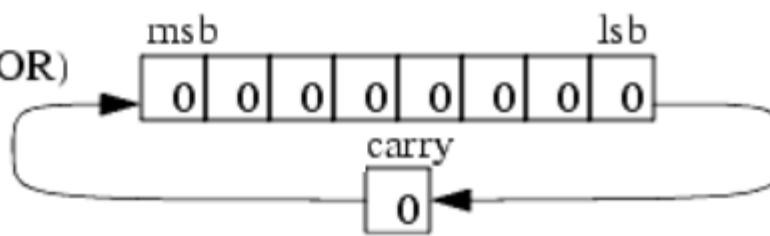
Shift Right (SHR)



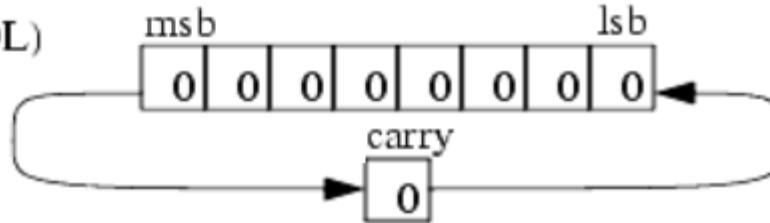
Shift Left (SHL)



Rotate Right (ROR)



Rotate Left (ROL)



MSB

LSB

| | | | | | | | |
|---|---|---|---|---|---|---|---|
| 1 | 0 | 1 | 0 | 1 | 1 | 0 | 0 |
|---|---|---|---|---|---|---|---|

SHR 1

| | | | | | | | |
|---|---|---|---|---|---|---|---|
| 0 | 1 | 0 | 1 | 0 | 1 | 1 | 0 |
|---|---|---|---|---|---|---|---|

SHL 1

| | | | | | | | |
|---|---|---|---|---|---|---|---|
| 0 | 1 | 0 | 1 | 1 | 0 | 0 | 0 |
|---|---|---|---|---|---|---|---|

ROR 1

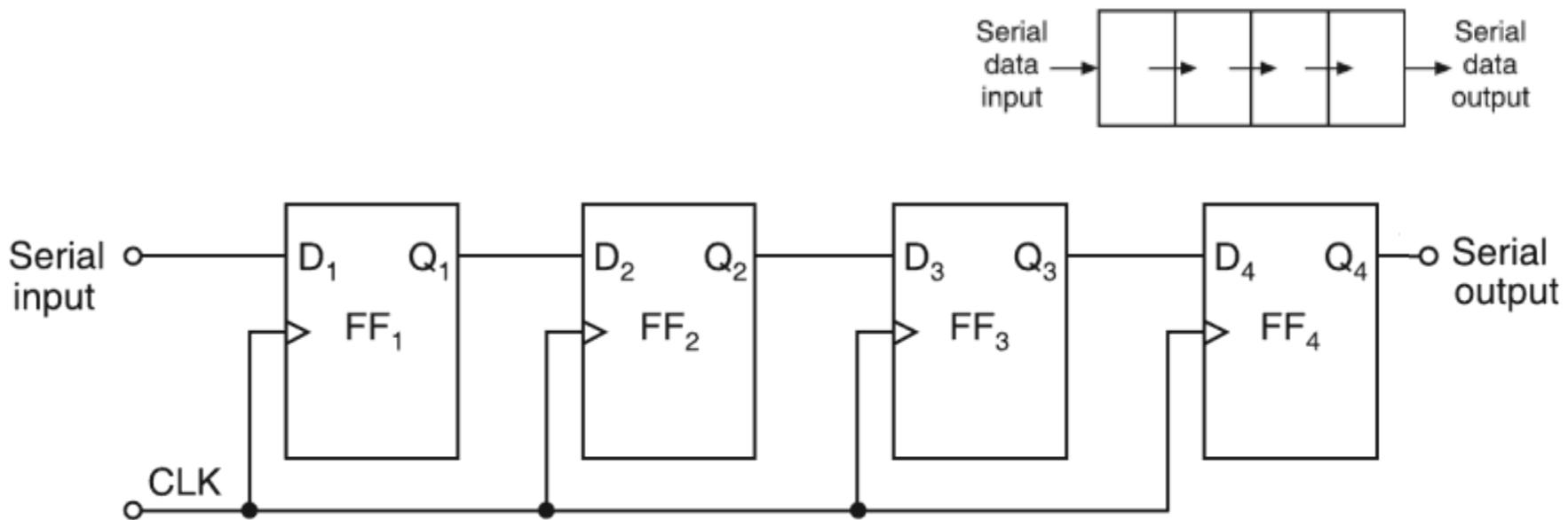
| | | | | | | | |
|---|---|---|---|---|---|---|---|
| 0 | 1 | 0 | 1 | 0 | 1 | 1 | 0 |
|---|---|---|---|---|---|---|---|

ROL 1

| | | | | | | | |
|---|---|---|---|---|---|---|---|
| 0 | 1 | 0 | 1 | 1 | 0 | 0 | 1 |
|---|---|---|---|---|---|---|---|

Serial-In Serial-Out (SISO) Shift Register

- This type of shift register accepts data serially, i.e. one bit at a time, and also outputs data serially.
- The logic diagram of a 4-bit serial-in, serial-out, shift-right, shift register using D FFs is shown in Figure.



Logic diagram of a 4-bit serial-in, serial-out, shift-right, shift register.

- With four stages, i.e. four FFs, the register can store up to four bits of data. Serial data is applied at the D input of the first FF. The Q output of the first FF is connected to the D input of the second FF, the Q output of the second FF is connected to the D input of the third FF, and the Q output of the third FF is connected to the D input of the fourth FF. The data is outputted from the Q terminal of the last FF.
- When serial data is transferred into a register, each new bit is clocked into the first FF at the positive-edge of each clock pulse. The bit that was previously stored by the first FF is transferred to the second FF. The bit that was stored by the second FF is transferred to the third FF, and so on. The bit that was stored by the last FF is shifted out.

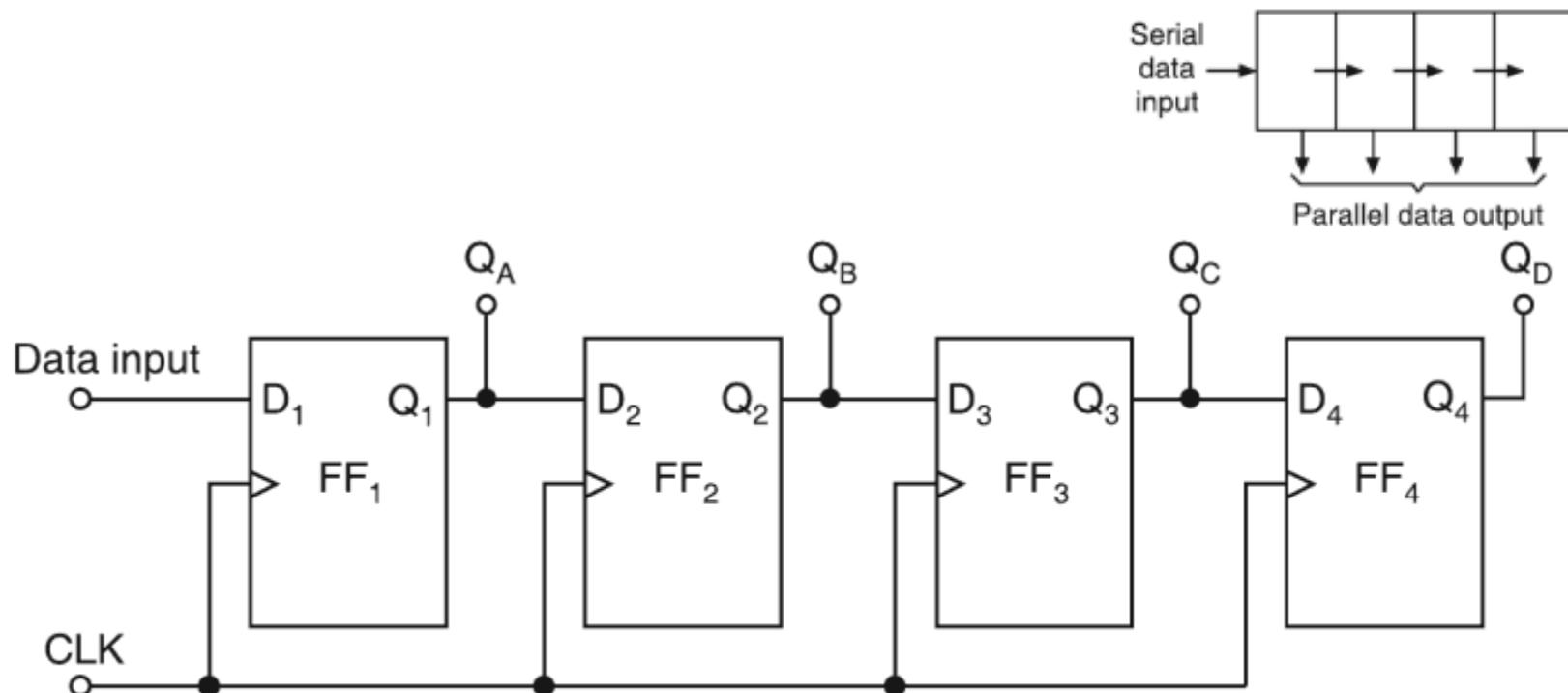
| Clock Pulse | Data | D ₁ | D ₂ | D ₃ | D ₄ | Q ₁ | Q ₂ | Q ₃ | Q ₄ |
|-------------|------|----------------|----------------|----------------|----------------|----------------|----------------|----------------|----------------|
| | 1010 | | | | | | | | |
| 1st | LSB | 0 | | | | 0 | | | |
| 2nd | Data | 1 | 0 | | | 1 | 0 | | |
| 3rd | In | 0 | 1 | 0 | | 0 | 1 | 0 | |
| 4th | MSB | 1 | 0 | 1 | 0 | 1 | 0 | 1 | 0 |
| 5th | | | 1 | 0 | 1 | | 1 | 0 | 1 |
| 6th | | | | 1 | 0 | | | 1 | 0 |
| 7th | | | | | 1 | | | | 1 |

LSB Data Out MSB

- In n -bit serial-in serial-out (SISO) shift register n number of clock pulses are required to store the n -bit data but in $(2n - 1)$ th clock pulse the n -bit data out from the shift register.

Serial-In Parallel-Out (SIPO) Shift Register

- In this type of register, the data bits are entered into the register serially, but the data stored in the register is shifted out in parallel form.



Logic diagram of a 4-bit serial-in, parallel-out, shift register.

- Figure shows the logic diagram of a 4-bit serial-in, parallel-out, shift register using D FFs.
- Once the data bits are stored, each bit appears on its respective output line and all bits are available simultaneously, rather than on a bit-by-bit basis as with the serial output.

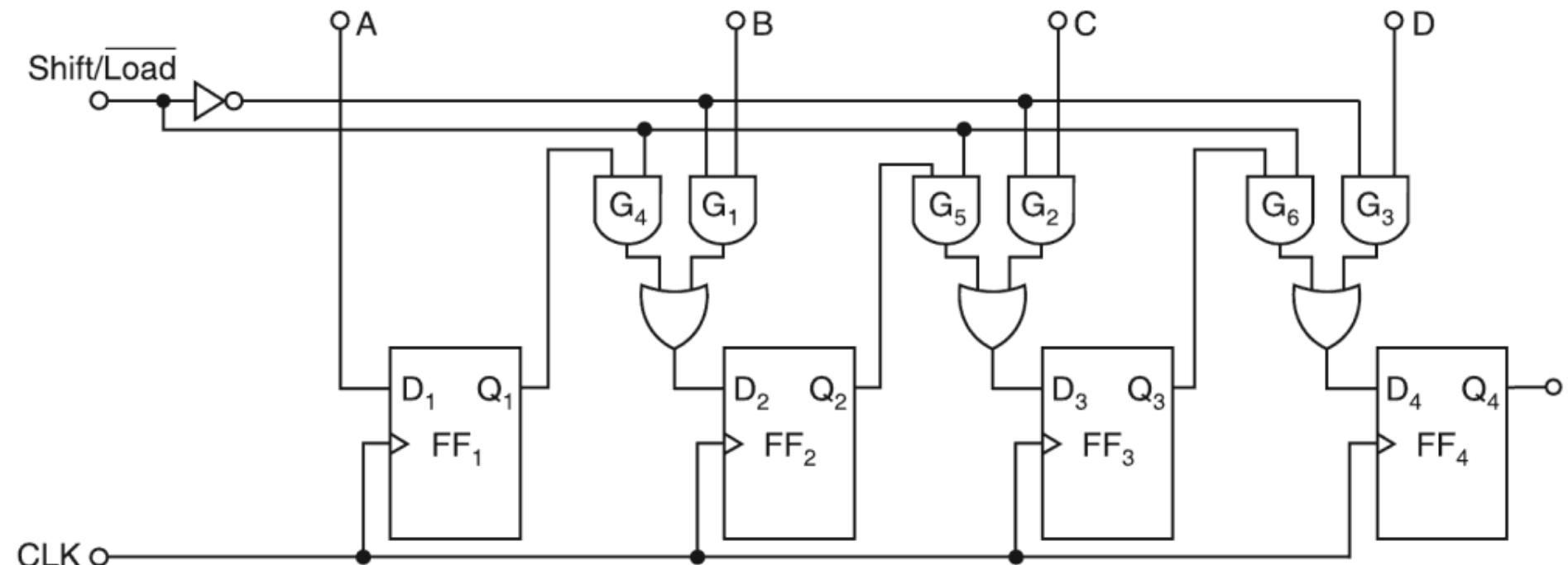
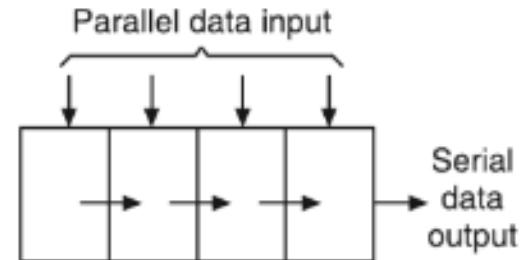
| Clock Pulse | Data | D ₁ | D ₂ | D ₃ | D ₄ | Q ₁ | Q ₂ | Q ₃ | Q ₄ |
|-------------|------|----------------|----------------|----------------|----------------|----------------|----------------|----------------|----------------|
| | 1010 | | | | | | | | |
| 1st | LSB | 0 | | | | 0 | | | |
| 2nd | Data | 1 | 0 | | | 1 | 0 | | |
| 3rd | In | 0 | 1 | 0 | | 0 | 1 | 0 | |
| 4th | MSB | 1 | 0 | 1 | 0 | 1 | 0 | 1 | 0 |

MSB LSB

Data Out

- In n -bit serial-in parallel-out (SIPO) shift register n number of clock pulses are required to store the n -bit data but in n th clock pulse the n -bit data out from the shift register.

Parallel-In Serial-Out (PISO) Shift Register



Logic diagram of a 4-bit parallel-in, serial-out, shift register.

- For a parallel-in, serial-out, shift register, the data bits are entered simultaneously into their respective stages on parallel lines, rather than on a bit-by-bit basis on one line as with serial data inputs, but the data bits are transferred out of the register serially, i.e. on a bit-by-bit basis over a single line.
- Figure illustrates a 4-bit parallel-in, serial-out, shift register using D FFs. There are four data lines A, B, C, and D through which the data is entered into the register in parallel form.

- The signal Shift/ $\overline{\text{Load}}$ allows (a) the data to be entered in parallel form into the register and (b) the data to be shifted out serially from terminal Q₄.
- When Shift/ $\overline{\text{Load}}$ line is HIGH, gates G₁, G₂, and G₃ are disabled, but gates G₄, G₅, and G₆ are enabled allowing the bits to shift-right from one stage to the next.
- When Shift/ $\overline{\text{Load}}$ line is LOW, gates G₄, G₅, and G₆ are disabled, but whereas gates G₁, G₂, and G₃ are enabled allowing the data input to appear at the D inputs of the respective FFs.

- When a clock pulse is applied, these data bits are shifted to the Q output terminals of the FFs and, therefore, data is inputted in one step.
- The OR gate allows either the normal shifting operation or the parallel data entry depending on which AND gates are enabled by the level on the Shift/ $\overline{\text{Load}}$ input.

| Clock Pulse | Shift /Load | Data | D ₁ | D ₂ | D ₃ | D ₄ | Q ₁ | Q ₂ | Q ₃ | Q ₄ |
|-------------|-------------|------|----------------|----------------|----------------|----------------|----------------|----------------|----------------|----------------|
| | | 1010 | | | | | | | | |
| 1st | 0 | | 1 | 0 | 1 | 0 | 1 | 0 | 1 | 0 |
| 2nd | 1 | | | | | | | 1 | 0 | 1 |
| 3rd | 1 | | | | | | | 1 | 0 | 0 |
| 4th | 1 | | | | | | | | | 1 |

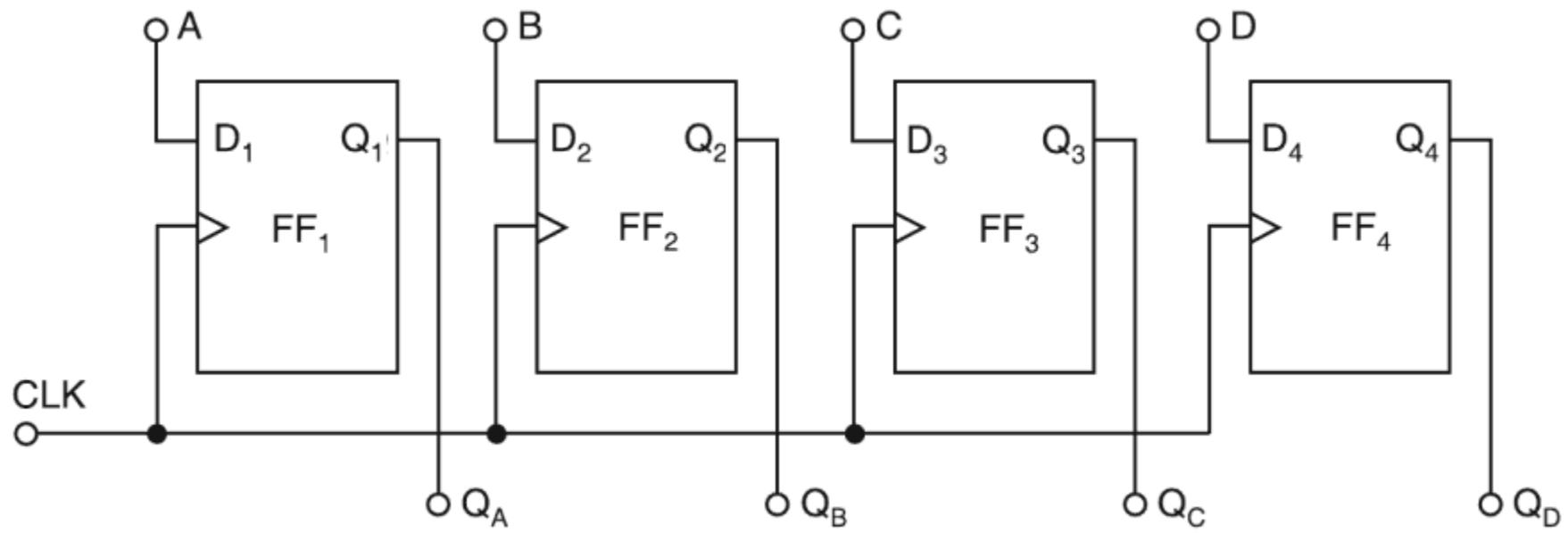
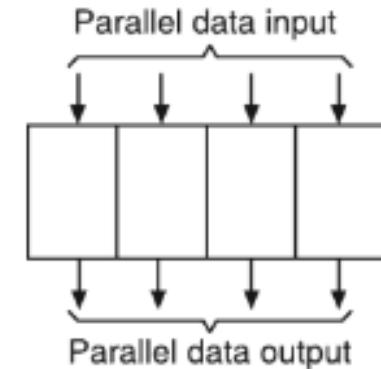
MSB Data In LSB

} Data Out
} MSB

- In n -bit parallel-in serial-out (PISO) shift register a **single** clock pulse is required to store the n -bit data but in nth clock pulse n -bit data out from the shift register.

Parallel-In Parallel-Out (PIPO) Shift Register

- In a parallel-in, parallel-out, shift register, the data is entered into the register in parallel form, and also the data is taken out of the register in parallel form.



Logic diagram of a 4-bit parallel-in, parallel-out, shift register.

- Figure shows a 4-bit parallel-in, parallel-out, shift register using D FFs. Data is applied to the D input terminals of the FFs. When a clock pulse is applied, at the positive-edge of that pulse, the D inputs are shifted into Q outputs of the FFs. The register now stores the data. The stored data is available instantaneously for shifting out in parallel form.

| Clock Pulse | Data | D ₁ | D ₂ | D ₃ | D ₄ | Q ₁ | Q ₂ | Q ₃ | Q ₄ |
|-------------|------|----------------|----------------|----------------|----------------|----------------|----------------|----------------|----------------|
| | 1010 | | | | | | | | |
| 1st | | 1 | 0 | 1 | 0 | 1 | 0 | 1 | 0 |

MSB Data In LSB
MSB Data Out LSB

- In n -bit parallel-in parallel-out (PIPO) shift register a **single** clock pulse is required to store the n -bit data but in **same** clock pulse the n -bit data out from the shift register.

DIGITAL ELECTRONICS

III Semester CSE & CST



Mr. Srikanta Patnaik Mr. Ajit Kumar Patro Mr. Ami Kumar Parida

DEPARTMENT OF ELECTRONICS AND COMMUNICATION ENGINEERING

SCHOOL OF ENGINEERING & TECHNOLOGY

GIET UNIVERSITY, GUNUPUR, ODISHA

Counters

Asynchronous Counters;

Design of Asynchronous Counters;

Synchronous Counters;

Design of Synchronous Counters;

Ring Counter

Counters

- A *counter* is essentially a register that goes through a predetermined sequence of binary states. The gates in the counter are connected in such a way as to produce the prescribed sequence of states.
- A register that goes through a prescribed sequence of states upon the application of input pulses is called a *counter*.
- The input pulses may be clock pulses, or they may originate from some external source and may occur at a fixed interval of time or at random.

- The sequence of states may follow the binary number sequence or any other sequence of states.
- A counter that follows the binary number sequence is called a *binary counter*.
- An n -bit binary counter consists of n flip-flops and can count in binary from 0 through $2^n - 1$.

- Counters are available in two categories: **ripple / asynchronous counters** and **synchronous counters**.
- In a **ripple counter**, a flip-flop output transition serves as a source for triggering other flip-flops.
- In other words, the C input of some or all flip-flops are triggered, not by the common clock pulses, but rather by the transition that occurs in other flip-flop outputs.
- In a **synchronous counter**, the C inputs of all flip-flops receive the common clock.

Comparison of Synchronous and Asynchronous Counters

Synchronous Counter

- In synchronous counter, the common clock input is connected to all the flip-flops and thus they are clocked simultaneously.
- Design and implementation becomes tedious and complex as the number of states increases.

Asynchronous Counter

- In asynchronous counter, each flip-flop is triggered by the previous flip-flop and the counter has a commutative setting time. Hence, the flip-flops are not clocked simultaneously.
- Design and implementation is very simple even for more number of states.

- A counter may be an *up-counter* or a *down-counter*.
- An *up-counter* is a counter which counts in the upward direction, i.e. 0, 1, 2, 3, ..., N .
- A *down-counter* is a counter which counts in the downward direction, i.e. N , $N - 1$, $N - 2$, $N - 3$, ..., 1, 0.
- Each of the counts of the counter is called the *state* of the counter.
- The number of states through which the counter passes before returning to the starting state is called the *modulus* of the counter.

- Since a 2-bit counter has 4 states, it is called a mod-4 counter. It divides the input clock signal frequency by 4, therefore, it is also called a divide-by-4 counter. It requires 2 FFs.
- Similarly, a 3-bit counter uses 3 FFs and has $2^3 = 8$ states. It divides the input clock frequency by 2^3 , i.e. 8.
- In general, an n -bit counter will have n FFs and 2^n states, and divides the input frequency by 2^n . Hence, it is a **divide-by- 2^n** counter.

- A counter which goes through all the possible states before restarting is called the *full modulus counter*.
- A counter in which maximum number of states can be changed is called the *variable modulus counter*.
- The final state of the counter sequence is called the *terminal count*.

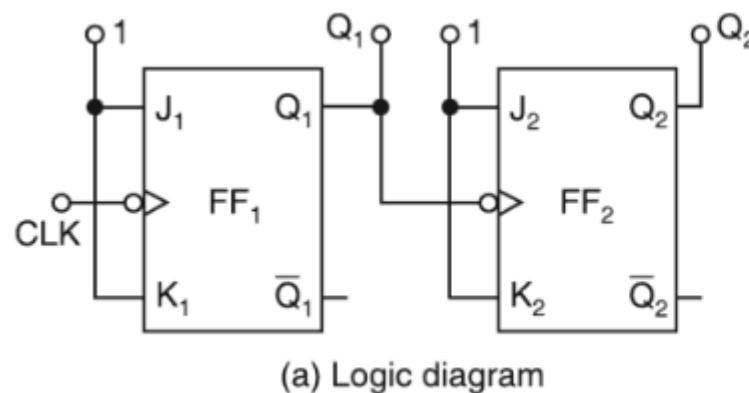
Asynchronous Counters

- Asynchronous Counters are also called *ripple counters*.
- The ripple counter is the simplest type of counter, the easiest to design and requires the least amount of hardware.
- A binary ripple counter consists of a series connection of complementing flip-flops, with the output of each flip-flop connected to the C input of the next higher order flip-flop.

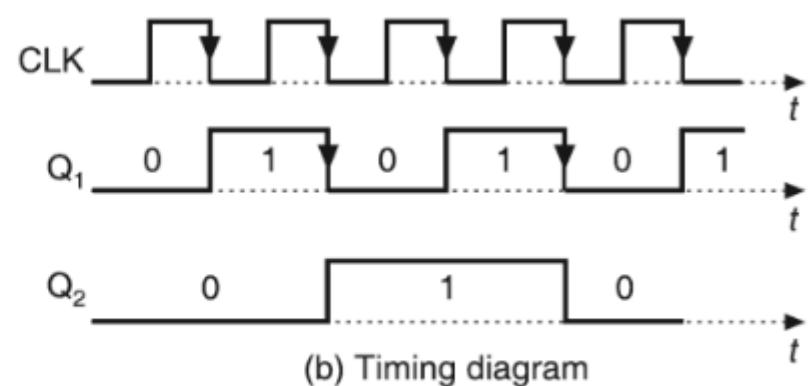
- The flip-flop holding the least significant bit receives the incoming count pulses.
- A complementing flip-flop can be obtained from a JK flip-flop with the J and K inputs tied together or from a T flip-flop. The inputs of JK flip-flop and T flip-flop are connected to a permanent logic 1.
- A third possibility is to use a D flip-flop with the complement output connected to the D input. In this way, the D input is always the complement of the present state, and the next clock pulse will cause the flip-flop to complement.

2-bit Ripple Up-counter using Negative Edge-triggered Flip-Flops

- The 2-bit up-counter counts in the order 0, 1, 2, 3, 0, 1, ..., i.e. 00, 01, 10, 11, 00, 01, ..., etc.
- Figure shows a 2-bit ripple up-counter, using negative edge-triggered J-K FFs, and its timing diagram.



(a) Logic diagram

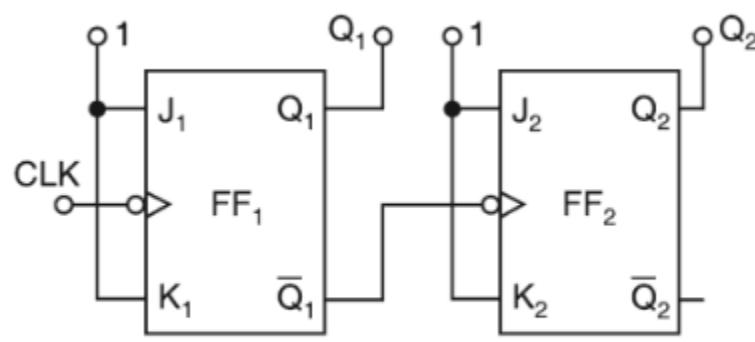


(b) Timing diagram

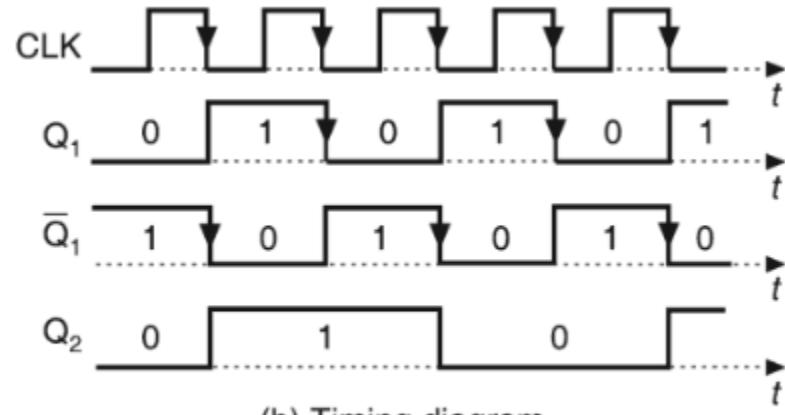
Asynchronous 2-bit up-counter using negative edge-triggered flip-flops.

2-bit Ripple Down-counter using Negative Edge-triggered Flip-Flops

- The 2-bit down-counter counts in the order 0, 3, 2, 1, 0, 3, ..., i.e. 00, 11, 10, 01, 00, 11, ..., etc.
- Figure shows a 2-bit ripple down-counter, using negative edge-triggered J-K FFs, and its timing diagram.



(a) Logic diagram



(b) Timing diagram

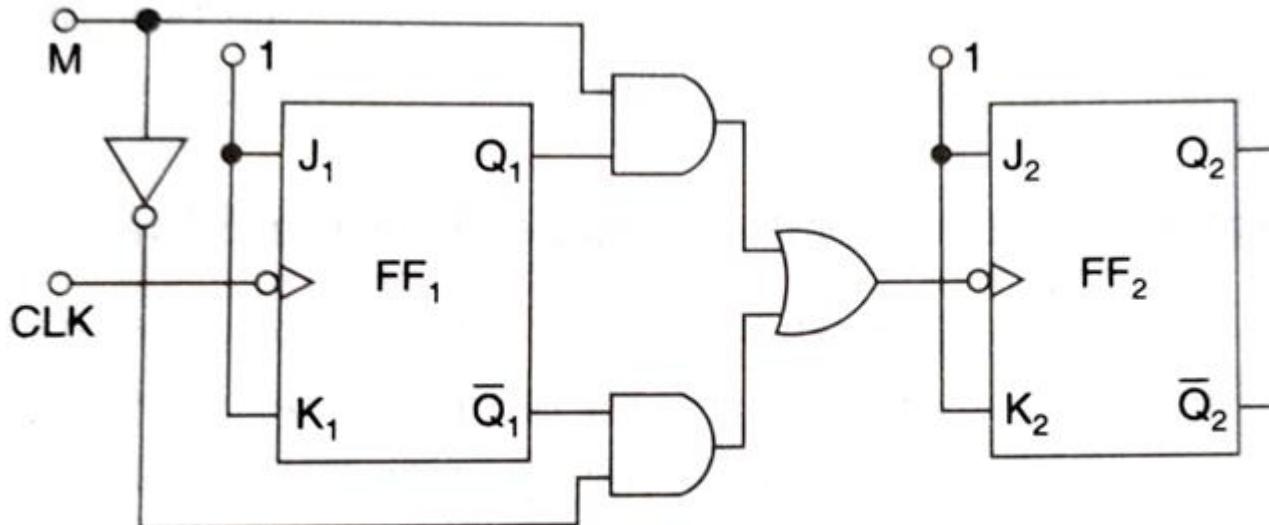
Asynchronous 2-bit down-counter using negative edge-triggered flip-flops.

2-bit Ripple Up-down Counter using Negative Edge-triggered Flip-Flops

- An up-down counter is a counter which can count both in upward and down directions.
- An up/down counter is also a forward/backward counter or a bidirectional counter.
- So, a control signal or a mode signal M is required to chose the direction of count.
- When $M = 1$ for up counting, Q_1 is transmitted to clock of FF_2 and when $M = 0$ for down counting, \bar{Q}_1 is transmitted to clock of FF_2 .

- This is achieved by using two AND gates and one OR gate as shown in Figure.
- The external clock signal is applied to FF_1 .

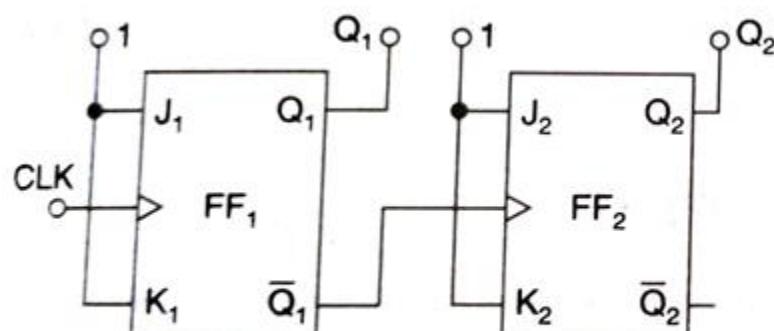
$$\begin{aligned}\text{Clock signal to } \text{FF}_2 &= (Q_1 \cdot \text{Up}) + (\bar{Q}_1 \cdot \text{Down}) \\ &= Q_1 M + \bar{Q}_1 \bar{M}\end{aligned}$$



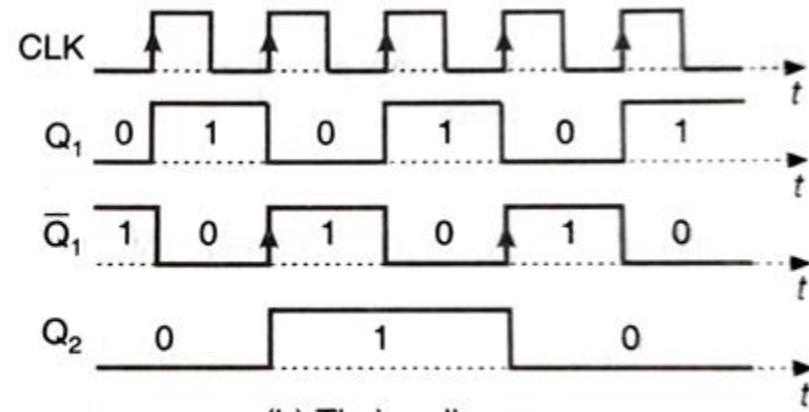
Asynchronous 2-bit up-down counter using negative edge-triggered flip-flops.

2-bit Ripple Up-counter using Positive Edge-triggered Flip-Flops

- A 2-bit ripple up-counter, using positive edge-triggered J-K FFs, and its timing diagram are shown in Figure.
- The counting sequence is 00, 01, 10, 11, 00, 01, ..., etc.



(a) Logic diagram

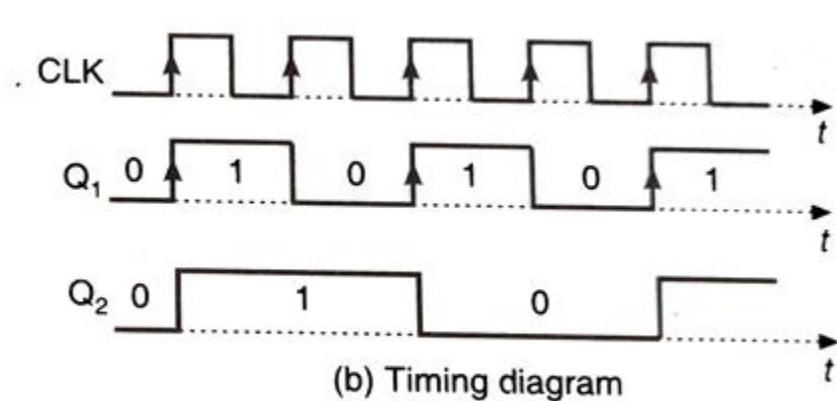
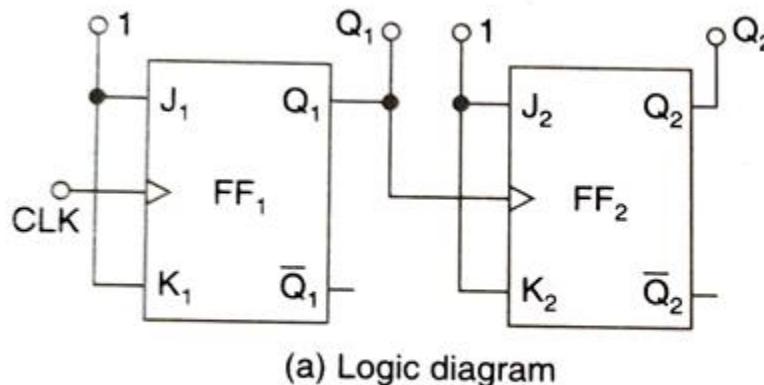


(b) Timing diagram

Asynchronous 2-bit up-counter using positive-edge triggered J-K flip-flops.

2-bit Ripple Down-counter using Positive Edge-triggered Flip-Flops

- A 2-bit ripple down-counter, using positive edge-triggered J-K FFs, and its timing diagram are shown in Figure.
- The counting sequence is 00, 11, 10, 01, 00, 11, ..., etc.

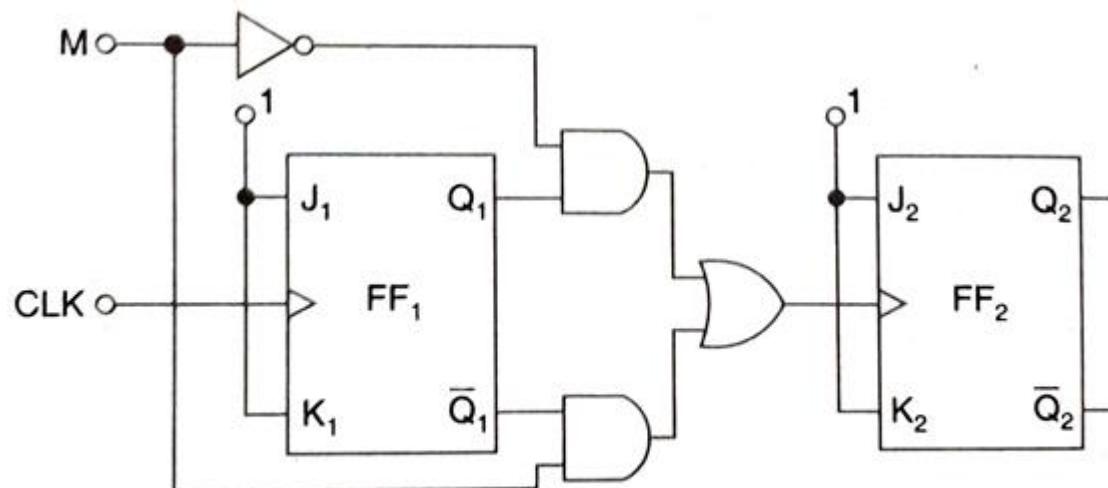


Asynchronous 2-bit down-counter using positive edge-triggered J-K flip-flops.

2-bit Ripple Up-down Counter using Positive Edge-triggered Flip-Flops

- Figure shows a 2-bit up-down counter using positive edge-triggered J-K FFs.
- When $M = 1$ for up counting, \overline{Q}_1 is transmitted to clock of FF_2 and when $M = 0$ for down counting, Q_1 is transmitted to clock of FF_2 .
- This is achieved by using two AND gates and one OR gate as shown in Figure.

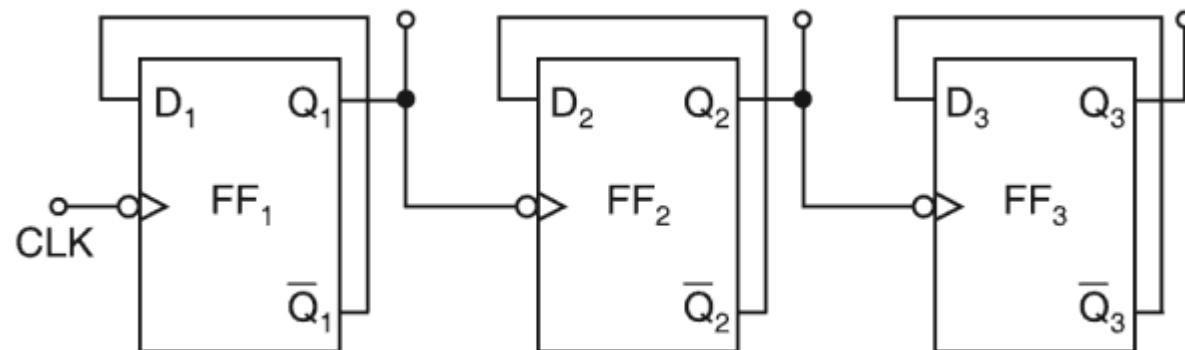
- Clock signal to $\text{FF}_2 = (\overline{Q}_1 \cdot \text{Up}) + (Q_1 \cdot \text{Down})$
 $= \overline{Q}_1 M + Q_1 \overline{M}$



Logic diagram of a two-bit ripple up/down counter
using positive edge-triggered flip-flops.

3-bit Ripple Counter using D FFs

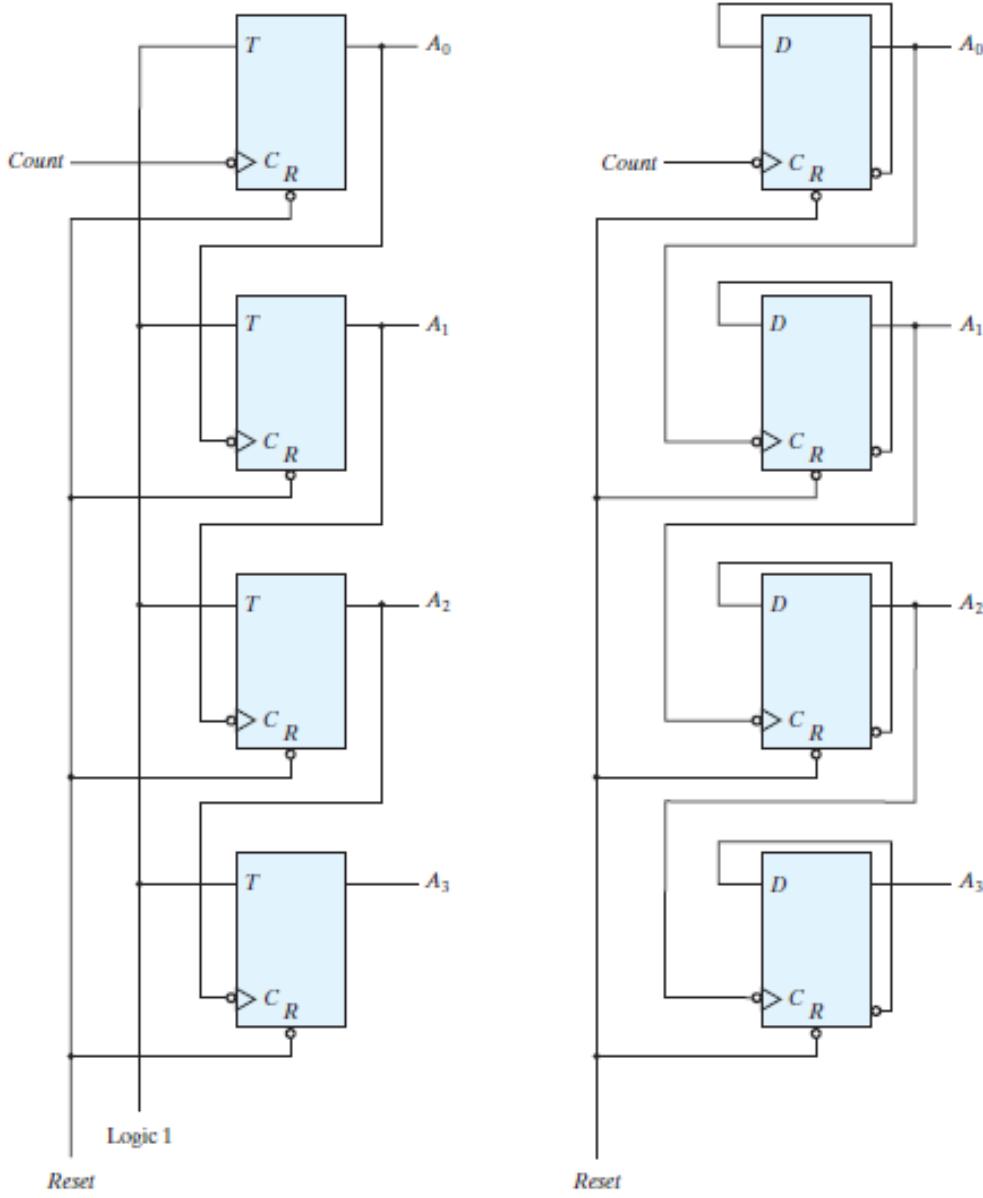
- For ripple counters, the FFs used must be in toggle mode. The D FFs may be used in toggle mode by connecting the \bar{Q} , of each FF to its D terminal.
- The 3-bit ripple counter using D FFs is shown in Figure.



Logic diagram of a 3-bit asynchronous counter using D flip-flops.

4-bit Binary Ripple Counters

- The logic diagram of two 4-bit binary ripple counters is shown in Figure.
- The counter is constructed with complementing flip-flops of the *T* type in part (a) and *D* type in part (b).
- The output of each flip-flop is connected to the *C* input of the next flip-flop in sequence.
- The flip-flop holding the least significant bit receives the incoming count pulses.



Four-bit binary ripple counter

Design of Asynchronous Counters

- To design an asynchronous counter, first write the counting sequence, then tabulate the values of reset signal R for various states of the counter and obtain the minimal expression for R or \bar{R} using K-map or any method.
- Provide a feedback such that R or \bar{R} resets all the FFs after the desired count.

Design of a Mod-6 Asynchronous Counter using T FFs.

- A mod-6 counter has six stable states 000, 001, 010, 011, 100, and 101.
- It is a ‘divide-by-6 counter’, in the sense that it divides the input clock frequency by 6.
- It requires three FFs, because the smallest value of n satisfying the condition $N \leq 2^n$ is $n = 3$; three FFs can have eight possible states out of which only six are utilized and the remaining two states 110 and 111, are invalid.

- For the design, write a truth table with the present state Q_3 , Q_2 , and Q_1 as the variables and reset R as the output and obtain an expression for R in terms of Q_3 , Q_2 , and Q_1 .

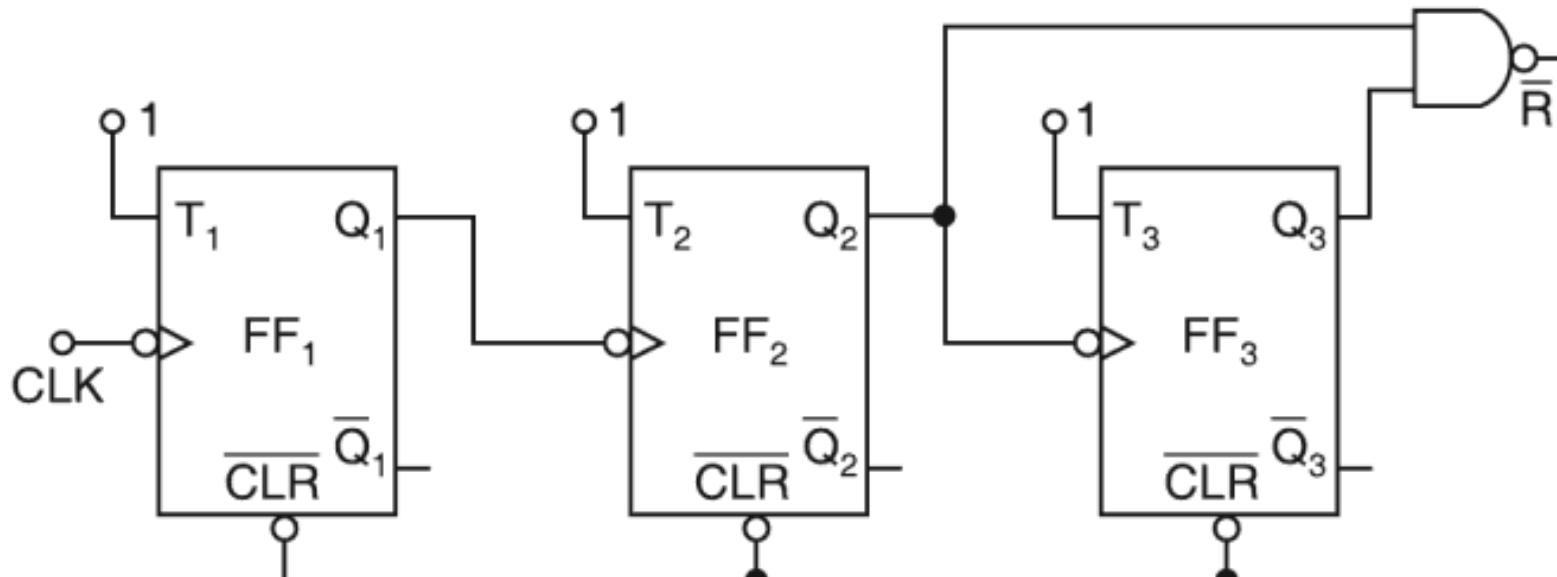
| After pulses | State | | | R |
|-----------------|-------|-------|-------|---|
| | Q_3 | Q_2 | Q_1 | |
| 0 | 0 | 0 | 0 | 0 |
| 1 | 0 | 0 | 1 | 0 |
| 2 | 0 | 1 | 0 | 0 |
| 3 | 0 | 1 | 1 | 0 |
| 4 | 1 | 0 | 0 | 0 |
| 5 | 1 | 0 | 1 | 0 |
| 6 | 1 | 1 | 0 | 1 |
| | ↓ | ↓ | ↓ | |
| | 0 | 0 | 0 | 0 |
| 7 | 0 | 0 | 1 | 0 |

(a) Table for R

- The expression for R can also be determined as follows.

$R = 0$ for 000 to 101, $R = 1$ for 110, and $R = X$ for 111

$$\text{Therefore, } R = Q_3 Q_2 \bar{Q}_1 + Q_3 Q_2 Q_1 = Q_3 Q_2$$



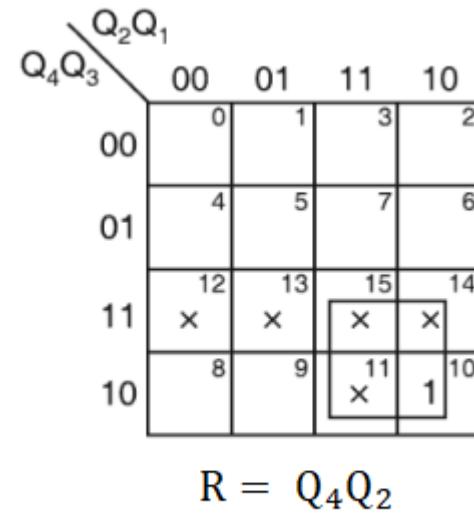
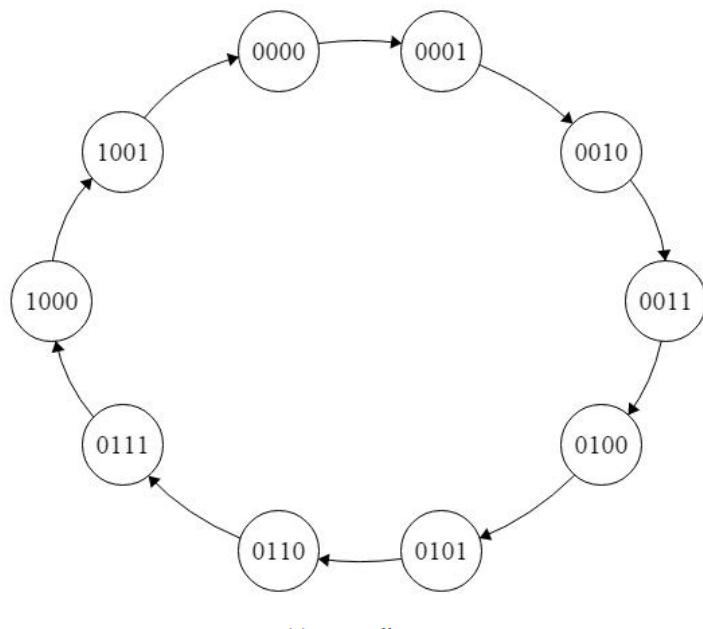
(b) Logic diagram

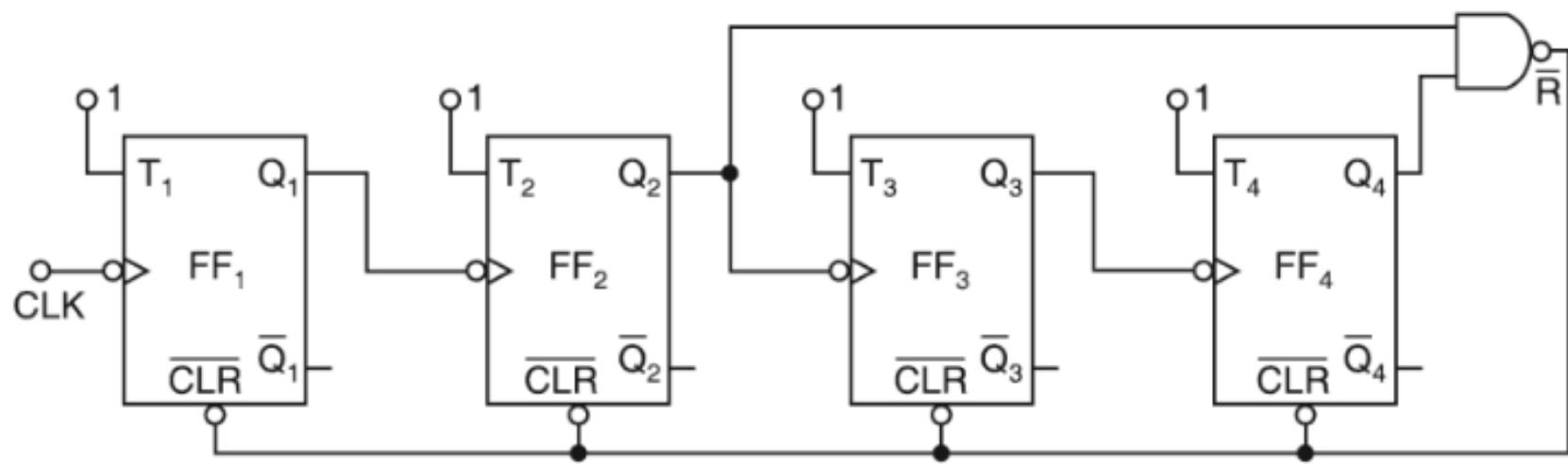
Asynchronous mod-6 counter using T flip-flops.

Design of a Mod-10 Asynchronous Counter using T FFs.

- A mod-10 counter is a decade counter. It is also called a BCD counter or a divide-by-10 counter.
- It requires four FFs (the smallest value of n satisfying the condition $10 \leq 2^n$, is $n = 4$). So, there are 16 possible states, out of which ten are valid and the remaining six are invalid.
- The counter has ten stable states, 0000 through 1001, i.e. it counts from 0 to 9.

- The state 1010 is a temporary state for which the reset signal $R = 1$, $R = 0$ for 0000 to 1001, and $R = X$ (don't care) for 1011 to 1111.





(c) Logic diagram

Asynchronous mod-10 counter using T flip-flops.

Synchronous Counters

- Synchronous counters are different from ripple counters in that clock pulses are applied to the inputs of all flip-flops.
- A common clock triggers all flip-flops simultaneously, rather than one at a time in succession as in a ripple counter.

Design of Synchronous Counters

- For a systematic design of synchronous counters, the following procedure is used.
- *Step 1. Number of flip-flops:* Based on the description of the problem, determine the required number n of the FFs – the smallest value of n is such that the number of states $N \leq 2^n$ – and the desired counting sequence.
- *Step 2. State diagram:* Draw the state diagram showing all the possible states.

- *Step 3. Choice of flip-flops and excitation table:*
Select the type of flip-flops to be used and write the excitation table that lists the present state (PS), the next state (NS) and the required excitation.
- *Step 4. Minimal expressions for the excitations:*
Obtain the minimal expressions for the excitations of the FFs using K-maps drawn for the excitations of the flip-flops in terms of present state and inputs.
- *Step 5. Logic diagram:* Draw a logic diagram based on the minimal expressions.

Excitation tables

| PS | NS | | Required inputs | |
|----|-------|-----------|-----------------|---|
| | Q_n | Q_{n+1} | S | R |
| 0 | 0 | 0 | 0 | x |
| 0 | 1 | 1 | 0 | |
| 1 | 0 | 0 | 1 | |
| 1 | 1 | x | 0 | |

(a) S-R FF excitation table

| PS | NS | | Required inputs | |
|----|-------|-----------|-----------------|---|
| | Q_n | Q_{n+1} | J | K |
| 0 | 0 | 0 | 0 | x |
| 0 | 1 | 1 | 1 | x |
| 1 | 0 | x | x | 1 |
| 1 | 1 | x | x | 0 |

(b) J-K FF excitation table

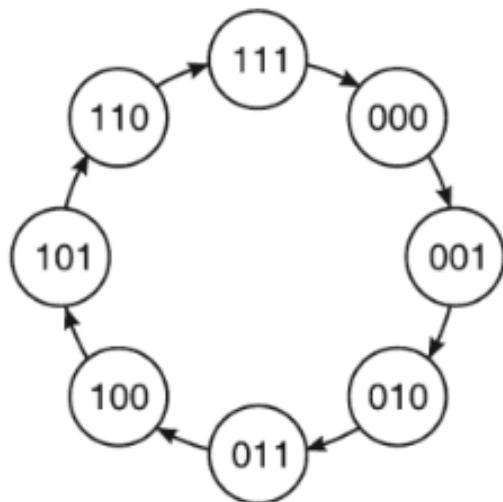
| PS | NS | | Required input |
|----|-------|-----------|----------------|
| | Q_n | Q_{n+1} | D |
| 0 | 0 | 0 | 0 |
| 0 | 1 | 1 | 1 |
| 1 | 0 | 0 | 0 |
| 1 | 1 | 1 | 1 |

(c) D FF excitation table

| PS | NS | | Required input |
|----|-------|-----------|----------------|
| | Q_n | Q_{n+1} | T |
| 0 | 0 | 0 | 0 |
| 0 | 1 | 1 | 1 |
| 1 | 0 | 0 | 1 |
| 1 | 1 | 1 | 0 |

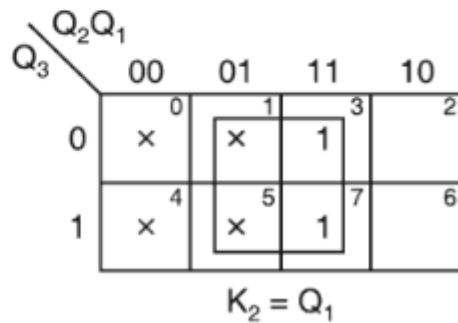
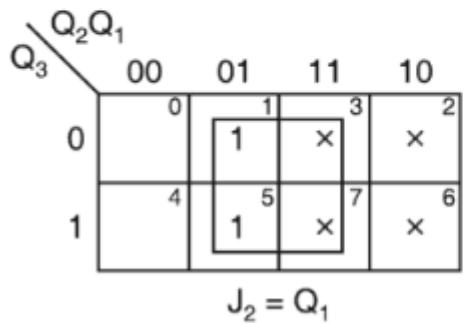
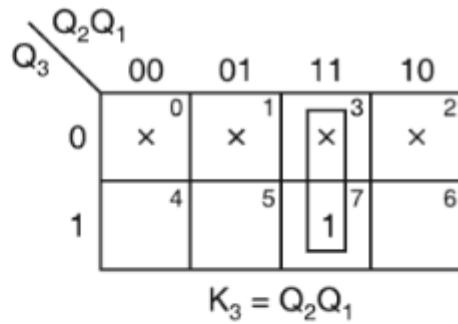
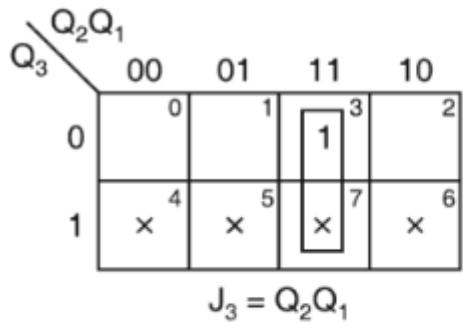
(d) T FF excitation table

Design of a Synchronous 3-bit Up Counter using J-K FFs.

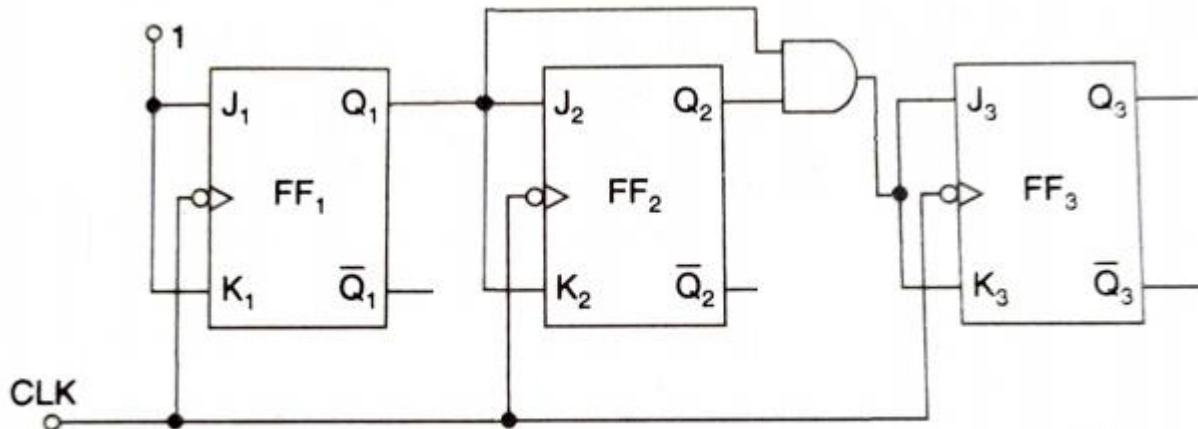


| PS | | | NS | | | Required excitations | | | | | |
|-------|-------|-------|-------|-------|-------|----------------------|-------|-------|-------|-------|-------|
| Q_3 | Q_2 | Q_1 | Q_3 | Q_2 | Q_1 | J_3 | K_3 | J_2 | K_2 | J_1 | K_1 |
| 0 | 0 | 0 | 0 | 0 | 1 | 0 | x | 0 | x | 1 | x |
| 0 | 0 | 1 | 0 | 1 | 0 | 0 | x | 1 | x | x | 1 |
| 0 | 1 | 0 | 0 | 1 | 1 | 0 | x | x | 0 | 1 | x |
| 0 | 1 | 1 | 1 | 0 | 0 | 1 | x | x | 1 | x | 1 |
| 1 | 0 | 0 | 1 | 0 | 1 | x | 0 | 0 | x | 1 | x |
| 1 | 0 | 1 | 1 | 1 | 0 | x | 0 | 1 | x | x | 1 |
| 1 | 1 | 0 | 1 | 1 | 1 | x | 0 | x | 0 | 1 | x |
| 1 | 1 | 1 | 0 | 0 | 0 | x | 1 | x | 1 | x | 1 |

(b) Excitation table

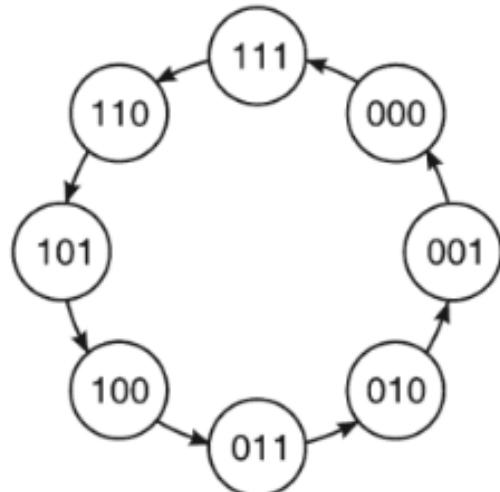


(c) Karnaugh maps for a 3-bit up counter using J-K FFs.



Logic diagram of the synchronous 3-bit up counter using J-K FFs.

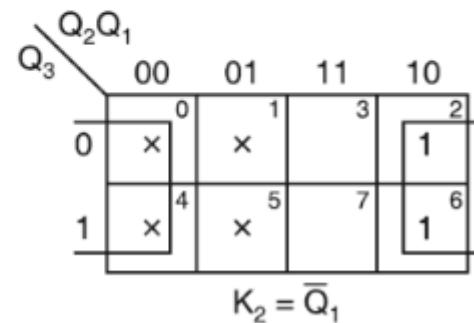
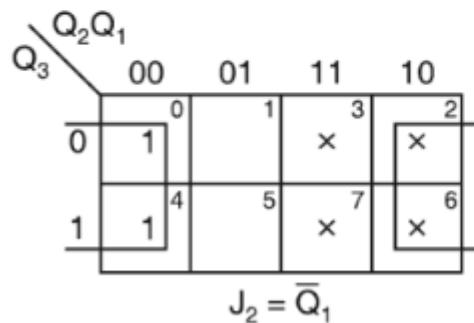
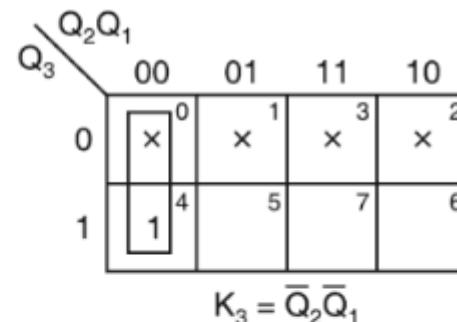
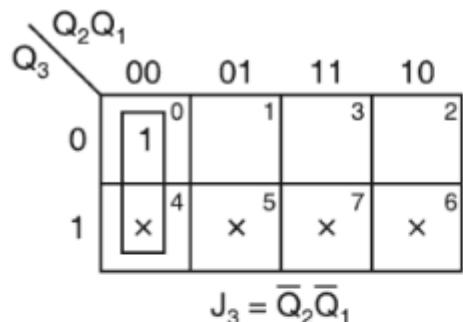
Design of a Synchronous 3-bit Down Counter using J-K FFs.



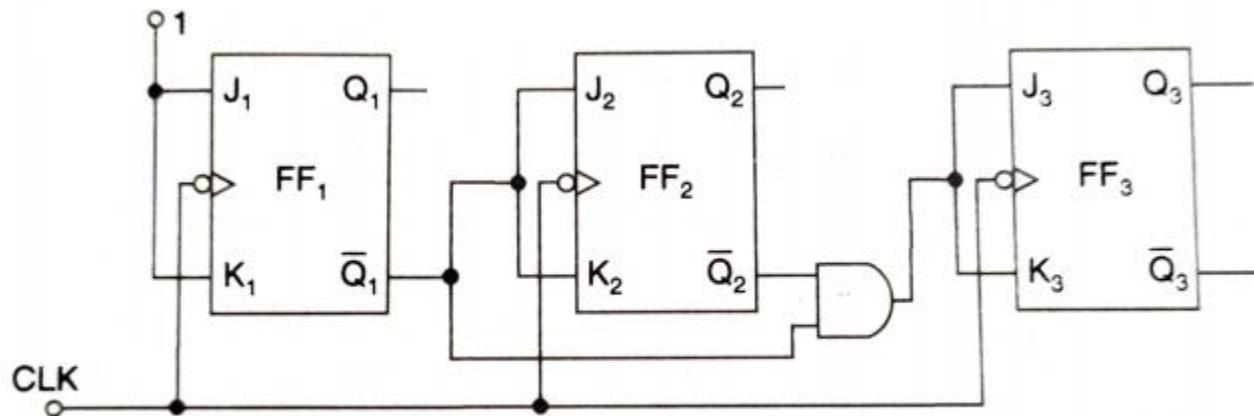
(a) State diagram

| PS | | | NS | | | Required excitations | | | | | |
|-------|-------|-------|-------|-------|-------|----------------------|-------|-------|-------|-------|-------|
| Q_3 | Q_2 | Q_1 | Q_3 | Q_2 | Q_1 | J_3 | K_3 | J_2 | K_2 | J_1 | K_1 |
| 0 | 0 | 0 | 1 | 1 | 1 | 1 | x | 1 | x | 1 | x |
| 1 | 1 | 1 | 1 | 1 | 0 | x | 0 | x | 0 | x | 1 |
| 1 | 1 | 0 | 1 | 0 | 1 | x | 0 | x | 1 | 1 | x |
| 1 | 0 | 1 | 1 | 0 | 0 | x | 0 | 0 | x | x | 1 |
| 1 | 0 | 0 | 0 | 1 | 1 | x | 1 | 1 | x | 1 | x |
| 0 | 1 | 1 | 0 | 1 | 0 | 0 | x | x | 0 | x | 1 |
| 0 | 1 | 0 | 0 | 0 | 1 | 0 | x | x | 1 | 1 | x |
| 0 | 0 | 1 | 0 | 0 | 0 | 0 | x | 0 | x | x | 1 |

(b) Excitation table

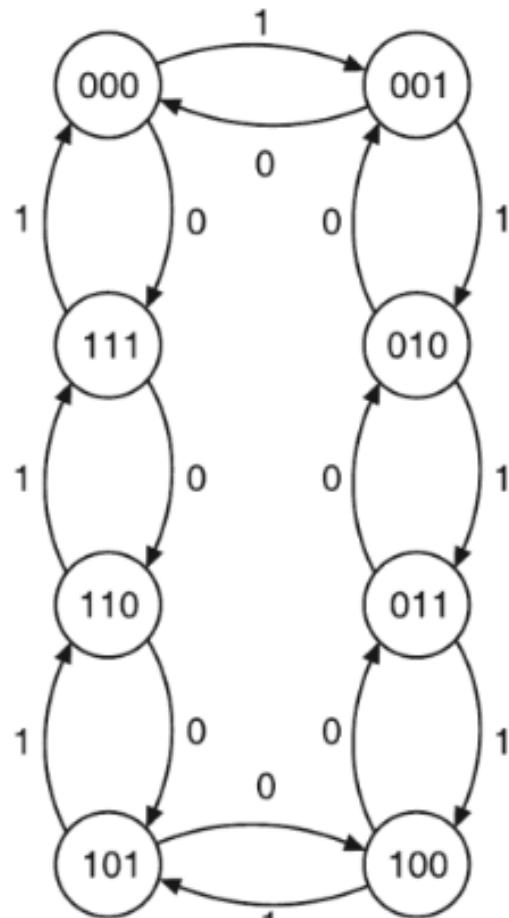


(c) Karnaugh maps for a 3-bit down counter using J-K FFs.



Logic diagram of the synchronous 3-bit down counter using J-K FFs.

Design of a Synchronous 3-bit Up-down Counter using J-K FFs.



(a) State diagram

| PS Q_3 Q_2 Q_1 | Mode M | NS | | | Required excitations | | | | | |
|-------------------------------|-----------|-------|-------|-------|----------------------|-------|-------|-------|-------|-------|
| | | Q_3 | Q_2 | Q_1 | J_3 | K_3 | J_2 | K_2 | J_1 | K_1 |
| 0 0 0 | 0 | 1 | 1 | 1 | 1 | x | 1 | x | 1 | x |
| 0 0 0 | 1 | 0 | 0 | 1 | 0 | x | 0 | x | 1 | x |
| 0 0 1 | 0 | 0 | 0 | 0 | 0 | x | 0 | x | x | 1 |
| 0 0 1 | 1 | 0 | 1 | 0 | 0 | x | 1 | x | x | 1 |
| 0 1 0 | 0 | 0 | 0 | 1 | 0 | x | x | 1 | 1 | x |
| 0 1 0 | 1 | 0 | 1 | 1 | 0 | x | x | 0 | 1 | x |
| 0 1 1 | 0 | 0 | 1 | 0 | 0 | x | x | 0 | x | 1 |
| 0 1 1 | 1 | 1 | 0 | 0 | 1 | x | x | 1 | x | 1 |
| 1 0 0 | 0 | 0 | 1 | 1 | x | 1 | 1 | x | 1 | x |
| 1 0 0 | 1 | 1 | 0 | 1 | x | 0 | 0 | x | 1 | x |
| 1 0 1 | 0 | 1 | 0 | 0 | x | 0 | 0 | x | x | 1 |
| 1 0 1 | 1 | 1 | 1 | 0 | x | 0 | 1 | x | x | 1 |
| 1 1 0 | 0 | 1 | 0 | 1 | x | 0 | x | 1 | 1 | x |
| 1 1 0 | 1 | 1 | 1 | 1 | x | 0 | x | 0 | 1 | x |
| 1 1 1 | 0 | 1 | 1 | 0 | x | 0 | x | 0 | x | 1 |
| 1 1 1 | 1 | 0 | 0 | 0 | x | 1 | x | 1 | x | 1 |

(b) Excitation table

| | | Q ₁ M | | |
|----|---|-------------------------------|----|----|
| | | Q ₃ Q ₂ | | |
| | | 00 01 11 10 | | |
| 00 | 0 | 1 | | 3 |
| 01 | 4 | 5 | 7 | 6 |
| 11 | x | x | x | 14 |
| 10 | 8 | x | 11 | x |

$$J_3 = \bar{Q}_2 \bar{Q}_1 \bar{M} + Q_2 Q_1 M$$

| | | Q ₁ M | | |
|----|---|-------------------------------|----|----|
| | | Q ₃ Q ₂ | | |
| | | 00 01 11 10 | | |
| 00 | 1 | 0 | 1 | 3 |
| 01 | x | 4 | 5 | x |
| 11 | x | 12 | 13 | x |
| 10 | 1 | 8 | 9 | 11 |

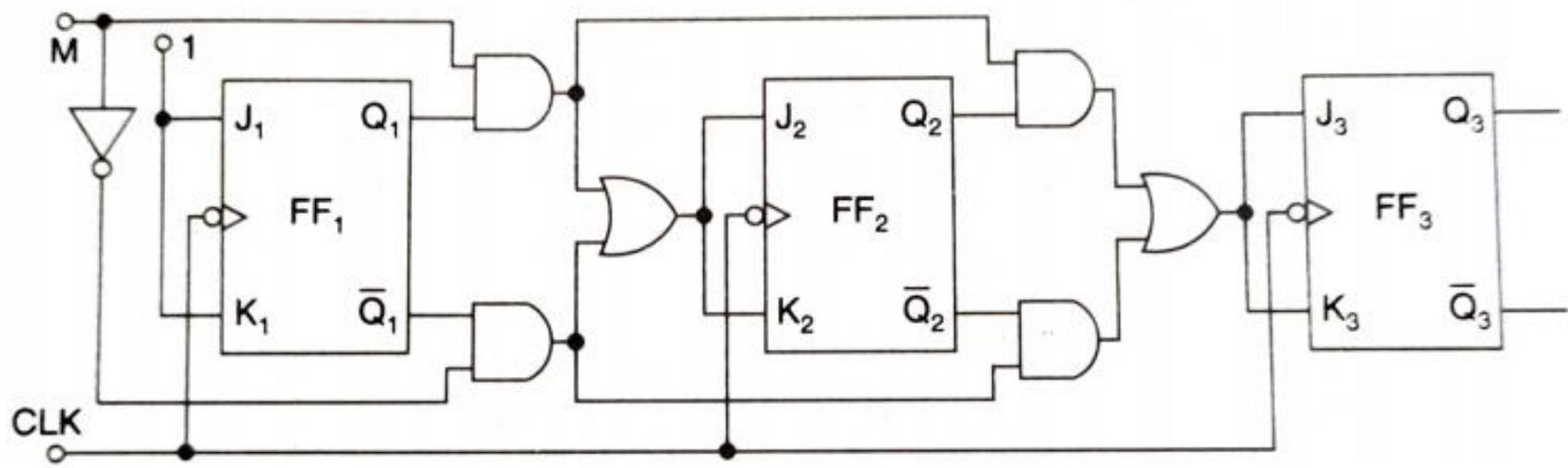
$$J_2 = \bar{Q}_1 \bar{M} + Q_1 M$$

| | | Q ₁ M | | |
|----|---|-------------------------------|----|----|
| | | Q ₃ Q ₂ | | |
| | | 00 01 11 10 | | |
| 00 | x | 0 | x | 3 |
| 01 | x | 4 | 5 | x |
| 11 | | 12 | 13 | 15 |
| 10 | 1 | 8 | 9 | 11 |

$$K_3 = \bar{Q}_2 \bar{Q}_1 \bar{M} + Q_2 Q_1 M$$

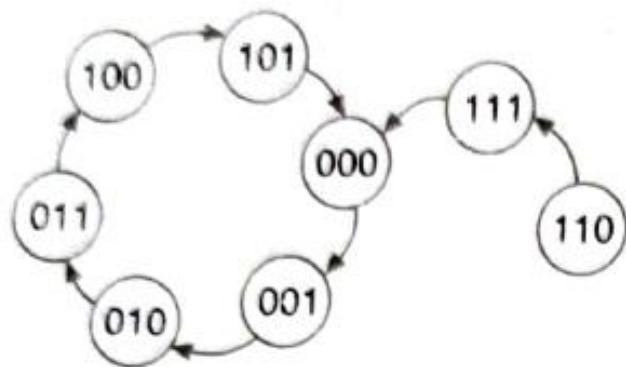
| | | Q ₁ M | | |
|----|---|-------------------------------|----|----|
| | | Q ₃ Q ₂ | | |
| | | 00 01 11 10 | | |
| 00 | x | 0 | x | 3 |
| 01 | 1 | 4 | 5 | 7 |
| 11 | 1 | 12 | 13 | 15 |
| 10 | x | 8 | x | x |

$$K_2 = \bar{Q}_1 \bar{M} + Q_1 M$$



Logic diagram of the synchronous 3-bit up-down counter using J-K FFs.

Design of a Synchronous Mod-6 Counter using J-K FFs.



(a) State diagram

| PS | | | NS | | | Required excitations | | | | | |
|-------|-------|-------|-------|-------|-------|----------------------|-------|-------|-------|-------|-------|
| Q_3 | Q_2 | Q_1 | Q_3 | Q_2 | Q_1 | J_3 | K_3 | J_2 | K_2 | J_1 | K_1 |
| 0 | 0 | 0 | 0 | 0 | 1 | 0 | x | 0 | x | 1 | x |
| 0 | 0 | 1 | 0 | 1 | 0 | 0 | x | 1 | x | x | 1 |
| 0 | 1 | 0 | 0 | 1 | 1 | 0 | x | x | 0 | 1 | x |
| 0 | 1 | 1 | 1 | 0 | 0 | 1 | x | x | 1 | x | 1 |
| 1 | 0 | 0 | 1 | 0 | 1 | x | 0 | 0 | x | 1 | x |
| 1 | 0 | 1 | 0 | 0 | 0 | x | 1 | 0 | x | x | 1 |

(b) Excitation table

| $Q_3 \backslash Q_2Q_1$ | 00 | 01 | 11 | 10 |
|-------------------------|----|----|----|----|
| 0 | 0 | 1 | 3 | 2 |
| 1 | x | 5 | x | x |

$J_3 = Q_2Q_1$

| $Q_3 \backslash Q_2Q_1$ | 00 | 01 | 11 | 10 |
|-------------------------|----|----|----|----|
| 0 | 0 | 1 | 3 | 2 |
| 1 | x | 5 | x | x |

$K_3 = Q_1$

| $Q_3 \backslash Q_2Q_1$ | 00 | 01 | 11 | 10 |
|-------------------------|----|----|----|----|
| 0 | 0 | 1 | 3 | 2 |
| 1 | x | 5 | x | x |

$J_2 = \bar{Q}_3Q_1$

| $Q_3 \backslash Q_2Q_1$ | 00 | 01 | 11 | 10 |
|-------------------------|----|----|----|----|
| 0 | x | x | 1 | 2 |
| 1 | x | 5 | x | x |

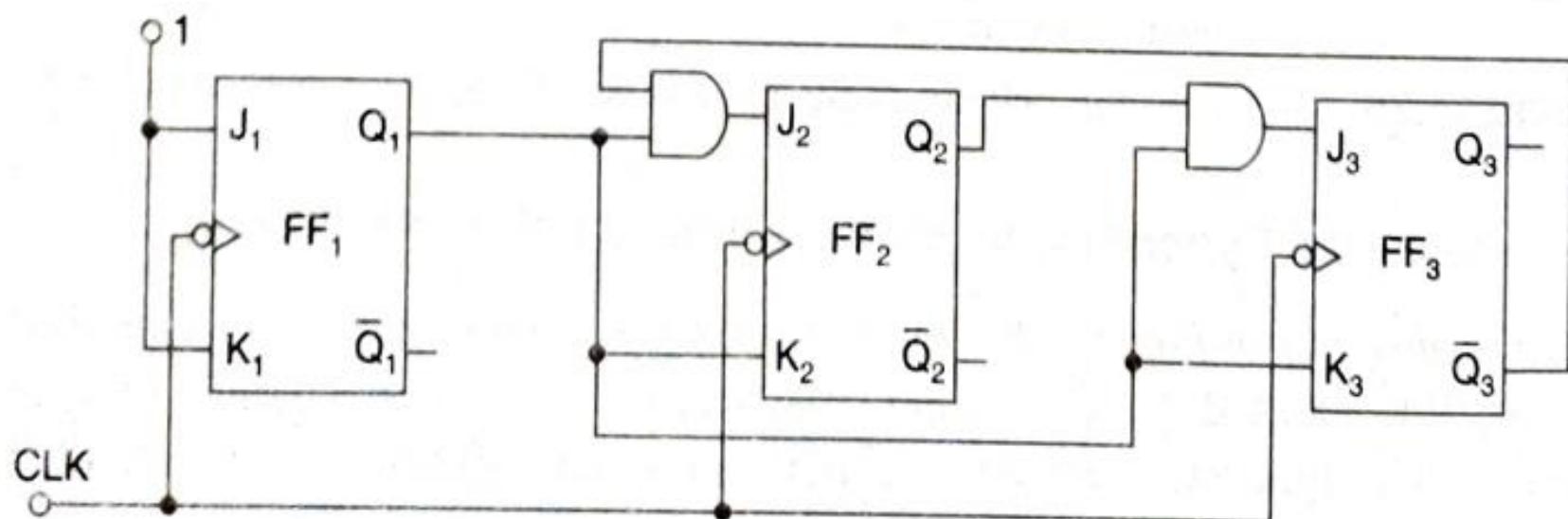
$K_2 = Q_1$

| $Q_3 \backslash Q_2Q_1$ | 00 | 01 | 11 | 10 |
|-------------------------|----|----|----|----|
| 0 | 0 | 1 | 3 | 2 |
| 1 | 1 | x | x | 1 |

$J_1 = 1$

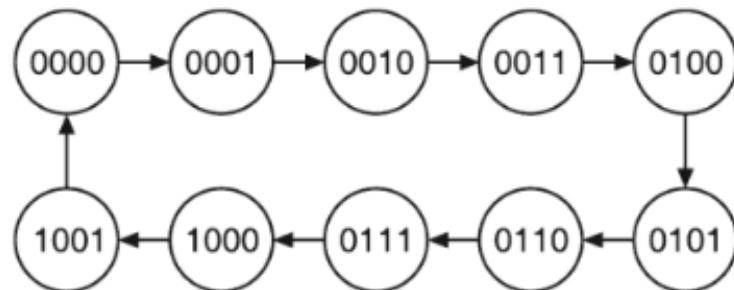
| $Q_3 \backslash Q_2Q_1$ | 00 | 01 | 11 | 10 |
|-------------------------|----|----|----|----|
| 0 | x | 1 | 1 | x |
| 1 | x | 1 | x | x |

$K_1 = 1$



Logic diagram of synchronous mod-6 counter using J-K flip-flops.

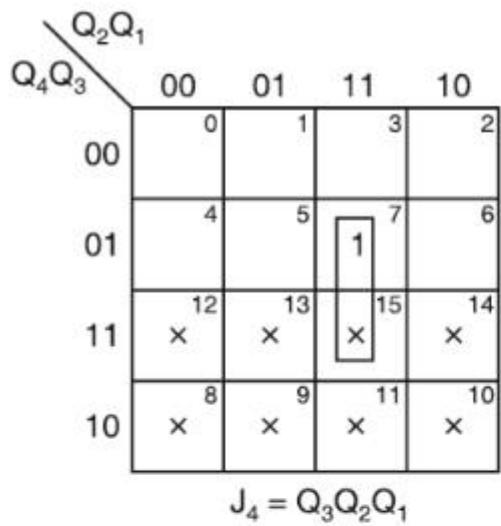
Design of a Synchronous BCD Counter using J-K FFs.



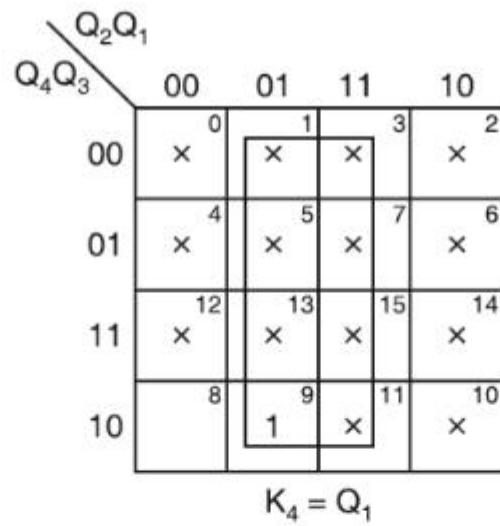
(a) State diagram

| PS | | | | NS | | | | Required excitations | | | | | | | |
|-------|-------|-------|-------|-------|-------|-------|-------|----------------------|-------|-------|-------|-------|-------|-------|-------|
| Q_4 | Q_3 | Q_2 | Q_1 | Q_4 | Q_3 | Q_2 | Q_1 | J_4 | K_4 | J_3 | K_3 | J_2 | K_2 | J_1 | K_1 |
| 0 | 0 | 0 | 0 | 0 | 0 | 0 | 1 | 0 | x | 0 | x | 0 | x | 1 | x |
| 0 | 0 | 0 | 1 | 0 | 0 | 1 | 0 | 0 | x | 0 | x | 1 | x | x | 1 |
| 0 | 0 | 1 | 0 | 0 | 0 | 1 | 1 | 0 | x | 0 | x | x | 0 | 1 | x |
| 0 | 0 | 1 | 1 | 0 | 1 | 0 | 0 | 0 | x | 1 | x | x | 1 | x | 1 |
| 0 | 1 | 0 | 0 | 0 | 1 | 0 | 1 | 0 | x | x | 0 | 0 | x | 1 | x |
| 0 | 1 | 0 | 1 | 0 | 1 | 1 | 0 | 0 | x | x | 0 | 1 | x | x | 1 |
| 0 | 1 | 1 | 0 | 0 | 1 | 1 | 1 | 0 | x | x | 0 | x | 0 | 1 | x |
| 0 | 1 | 1 | 1 | 1 | 0 | 0 | 0 | 1 | x | x | 1 | x | 1 | x | 1 |
| 1 | 0 | 0 | 0 | 1 | 0 | 0 | 1 | x | 0 | 0 | x | 0 | x | 1 | x |
| 1 | 0 | 0 | 1 | 0 | 0 | 0 | 0 | x | 1 | 0 | x | 0 | x | x | 1 |

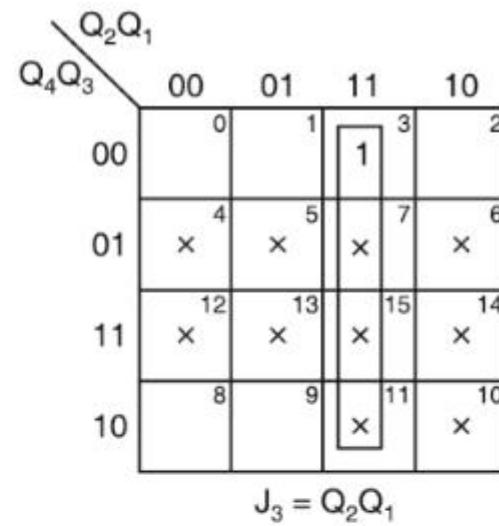
(b) Excitation table



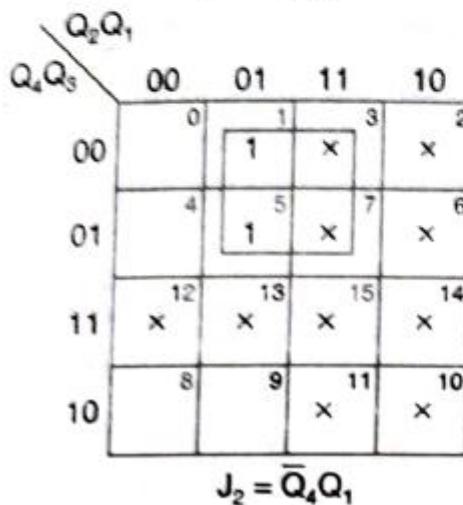
$$J_4 = Q_3Q_2Q_1$$



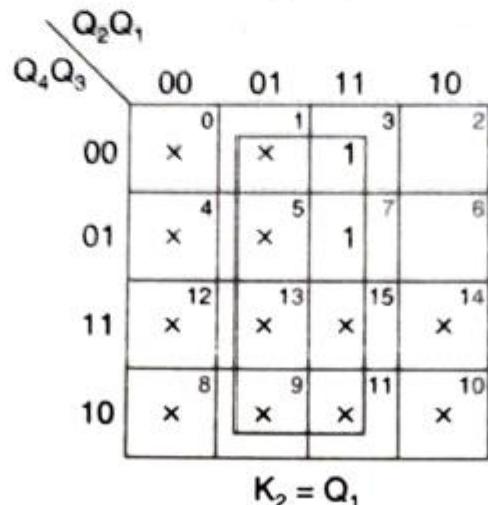
$$K_4 = Q_1$$



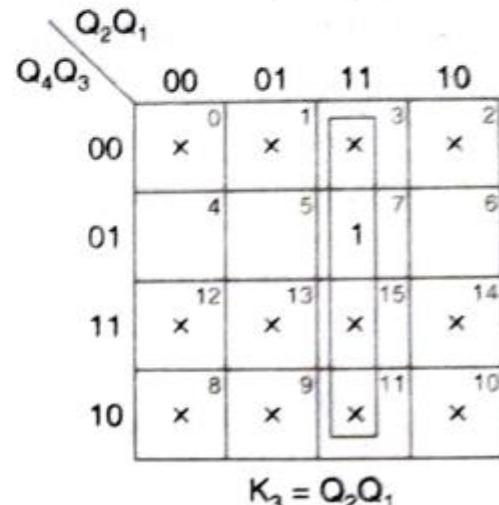
$$J_3 = Q_2Q_1$$



$$J_2 = \overline{Q}_4Q_1$$

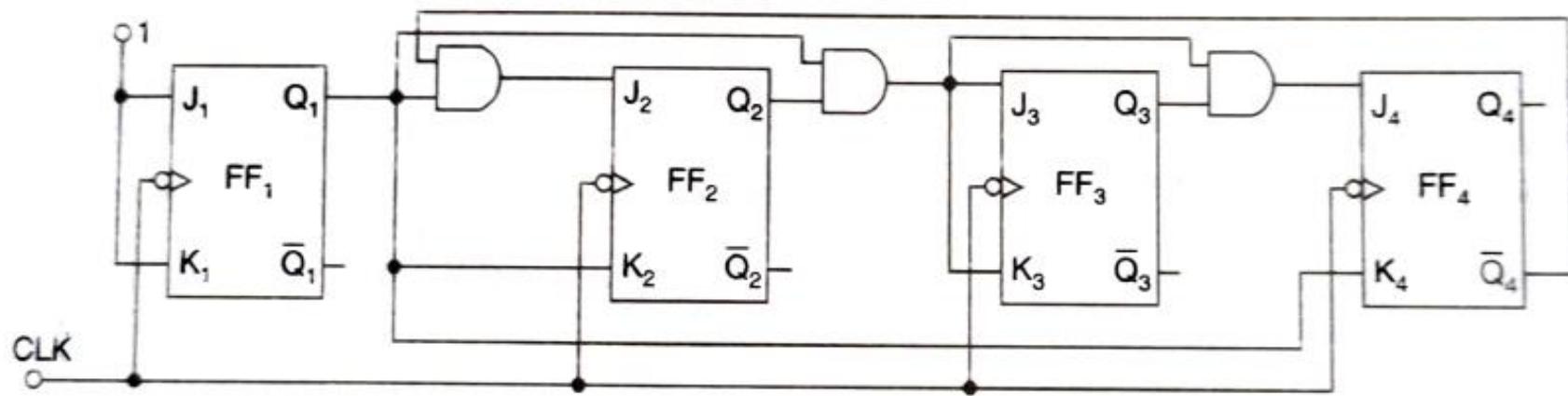


$$K_2 = Q_1$$



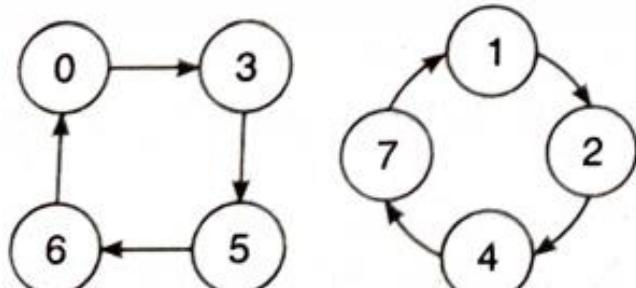
$$K_3 = Q_2Q_1$$

K-maps for excitations of synchronous BCD counter using J-K flip-flops



Logic diagram of synchronous BCD counter using J-K flip-flops.

Example: Design a type T counter that goes through states 0, 3, 5, 6, 0... .



(a) State diagram

| PS | | | NS | | | Required excitations | | |
|-------|-------|-------|-------|-------|-------|----------------------|-------|-------|
| Q_3 | Q_2 | Q_1 | Q_3 | Q_2 | Q_1 | T_3 | T_2 | T_1 |
| 0 | 0 | 0 | 0 | 1 | 1 | 0 | 1 | 1 |
| 0 | 1 | 1 | 1 | 0 | 1 | 1 | 1 | 0 |
| 1 | 0 | 1 | 1 | 1 | 0 | 0 | 1 | 1 |
| 1 | 1 | 0 | 0 | 0 | 0 | 1 | 1 | 0 |

(b) Excitation table

| Q_3 | $Q_2 Q_1$ | 00 | 01 | 11 | 10 |
|-------|-----------|----|----|----|----|
| 0 | 0 | x | 1 | 3 | 2 |
| 1 | x | 4 | 5 | 7 | 6 |
| | | | | | |

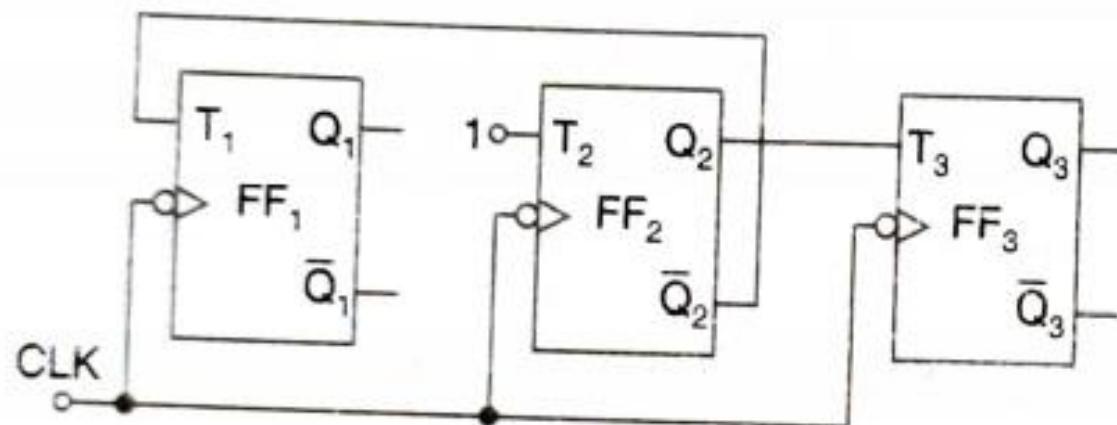
$T_3 = Q_2$

| Q_3 | $Q_2 Q_1$ | 00 | 01 | 11 | 10 |
|-------|-----------|----|----|----|----|
| 0 | 0 | 0 | x | 1 | 3 |
| 1 | x | 4 | 5 | 1 | 7 |
| | | | | | |

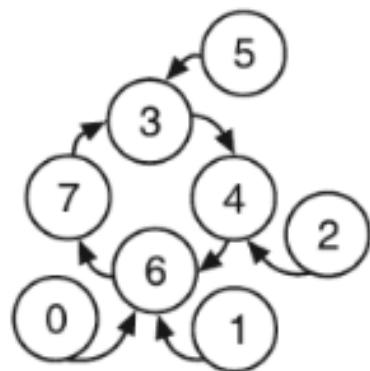
$T_2 = 1$

| Q_3 | $Q_2 Q_1$ | 00 | 01 | 11 | 10 |
|-------|-----------|----|----|----|----|
| 0 | 0 | 0 | 1 | 3 | 2 |
| 1 | x | 4 | 1 | 5 | 6 |
| | | | | | |

$T_1 = \bar{Q}_2$



Example: Design a J-K counter that goes through states 3, 4, 6, 7 and 3... .



(a) State diagram

| PS | | | NS | | | Required excitations | | | | | |
|-------|-------|-------|-------|-------|-------|----------------------|-------|-------|-------|-------|-------|
| Q_3 | Q_2 | Q_1 | Q_3 | Q_2 | Q_1 | J_3 | K_3 | J_2 | K_2 | J_1 | K_1 |
| 0 | 1 | 1 | 1 | 0 | 0 | 1 | x | x | 1 | x | 1 |
| 1 | 0 | 0 | 1 | 1 | 0 | x | 0 | 1 | x | 0 | x |
| 1 | 1 | 0 | 1 | 1 | 1 | x | 0 | x | 0 | 1 | x |
| 1 | 1 | 1 | 0 | 1 | 1 | x | 1 | x | 0 | x | 0 |

(b) Excitation table

| | $Q_2 Q_1$ | 00 | 01 | 11 | 10 |
|-------|-----------|----|----|----|----|
| Q_3 | 0 | x | 0 | 3 | 2 |
| | 1 | 4 | 5 | 7 | 6 |

$$K_3 = Q_1$$

| | $Q_2 Q_1$ | 00 | 01 | 11 | 10 |
|-------|-----------|----|----|----|----|
| Q_3 | 0 | 0 | 1 | 3 | 2 |
| | 1 | x | x | 1 | 6 |

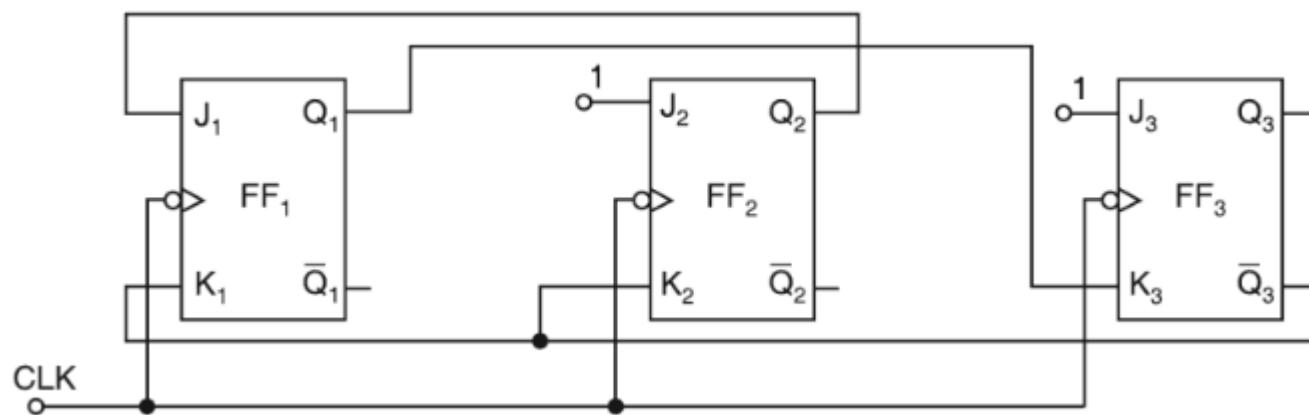
$$K_2 = \bar{Q}_3$$

| | $Q_2 Q_1$ | 00 | 01 | 11 | 10 |
|-------|-----------|----|----|----|----|
| Q_3 | 0 | x | x | 3 | 2 |
| | 1 | 4 | 5 | 7 | 6 |

$$J_1 = Q_2$$

| | $Q_2 Q_1$ | 00 | 01 | 11 | 10 |
|-------|-----------|----|----|----|----|
| Q_3 | 0 | 0 | 1 | 3 | 2 |
| | 1 | x | x | 7 | 6 |

$$K_1 = \bar{Q}_3$$

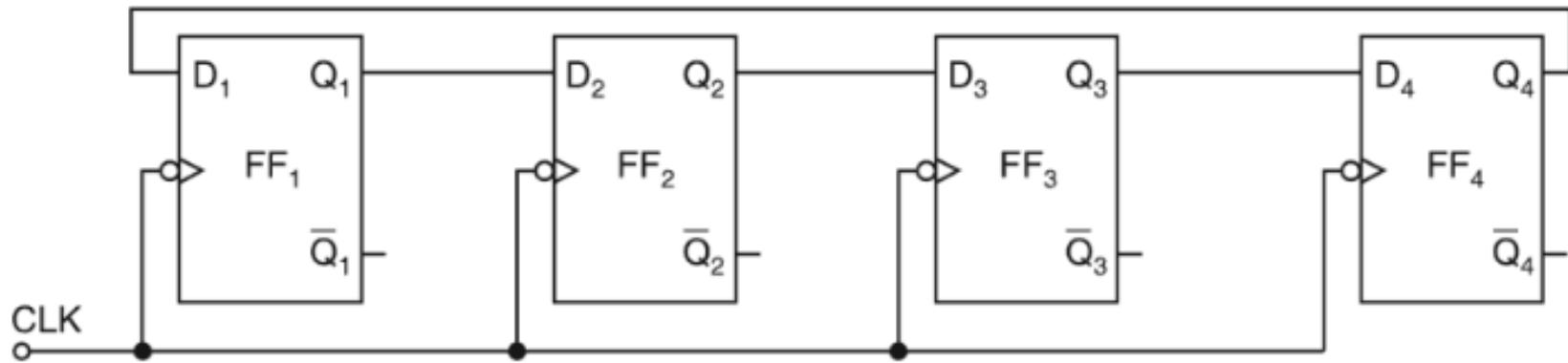


Shift Register Counters

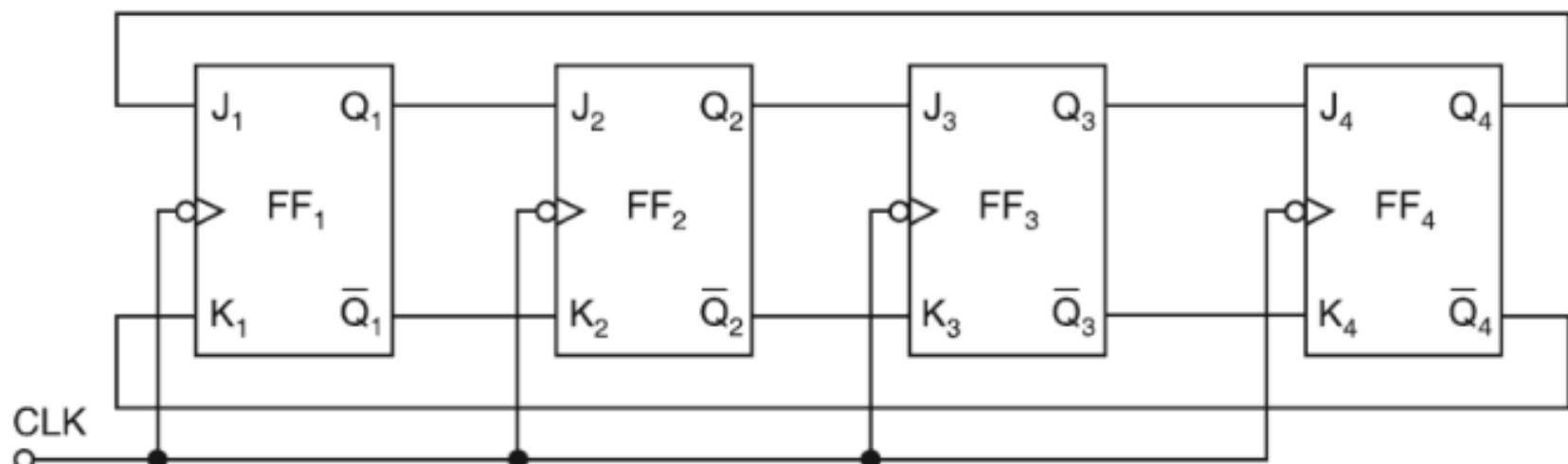
- Shift register counters are obtained from serial-in, serial-out shift registers by providing feedback from the output of the last FF to the input of the first FF.
- These devices are called counters because they exhibit a specified sequence of states.
- The most widely used shift register counter is the **ring counter** as well as the **switch-tail ring counter** or **Johnson counter**.

Ring Counter

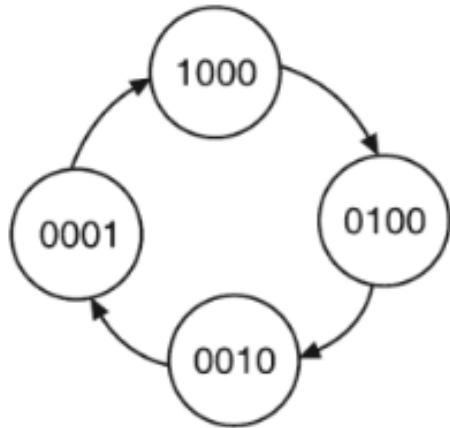
- A *ring counter* is a circular shift register with only one flip-flop being set at any particular time; all others are cleared.
- The single bit is shifted from one flip-flop to the next to produce the sequence of timing signals.
- The FFs are arranged as in a normal shift register, i.e. the Q output of each stage is connected to the D input of the next stage, but the Q output of the last FF is connected back to the D input of the first FF.



Logic diagram of a 4-bit ring counter using D flip-flops.



Logic diagram of a 4-bit ring counter using J-K flip-flops.



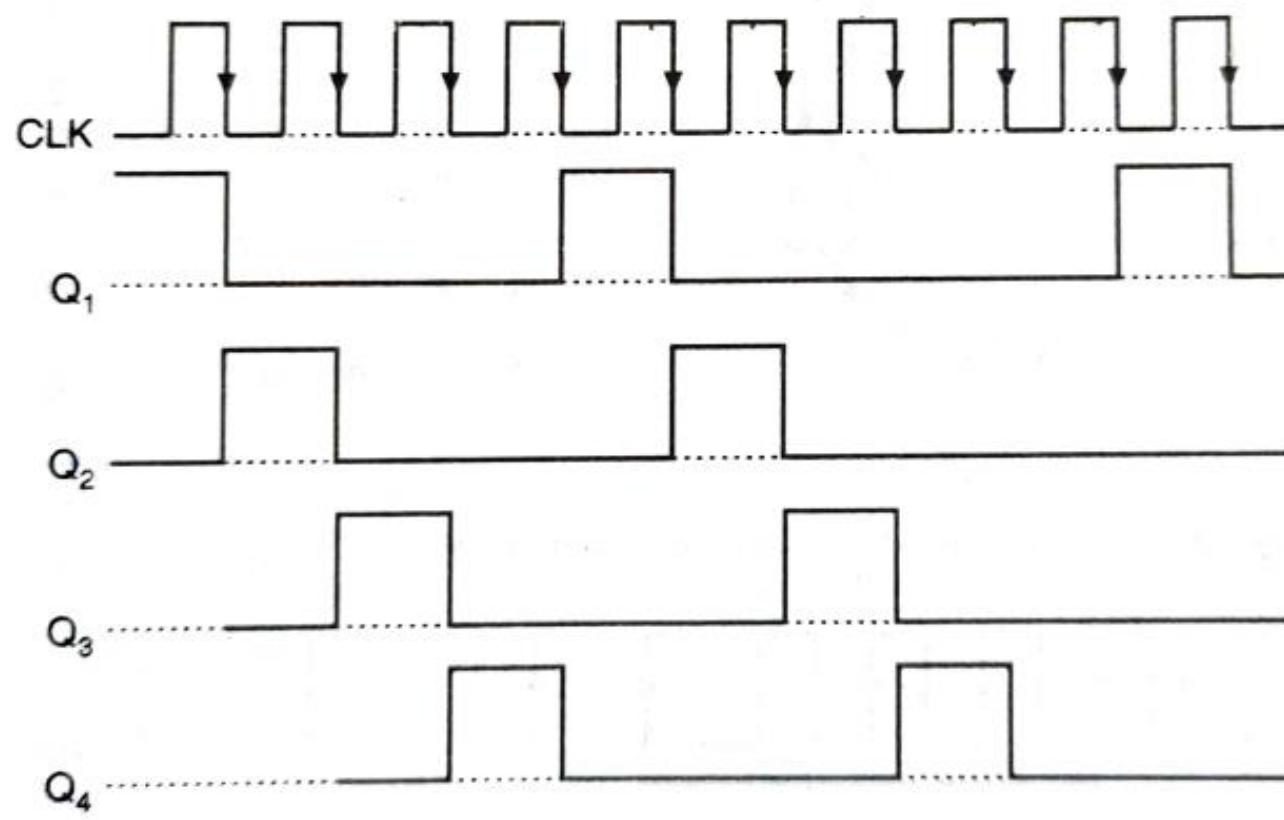
(a) State diagram

| Q_1 | Q_2 | Q_3 | Q_4 | After clock pulse |
|-------|-------|-------|-------|-------------------|
| 1 | 0 | 0 | 0 | 0 |
| 0 | 1 | 0 | 0 | 1 |
| 0 | 0 | 1 | 0 | 2 |
| 0 | 0 | 0 | 1 | 3 |
| 1 | 0 | 0 | 0 | 4 |
| 0 | 1 | 0 | 0 | 5 |
| 0 | 0 | 1 | 0 | 6 |
| 0 | 0 | 0 | 1 | 7 |

(b) Sequence table

State diagram and sequence table of a 4-bit ring counter.

- The number of distinct states in the ring counter, i.e. the mod of the ring counter is equal to the number of FFs used in the counter.
- An n -bit ring counter can count only n states, whereas an n -bit ripple counter can count 2^n states.

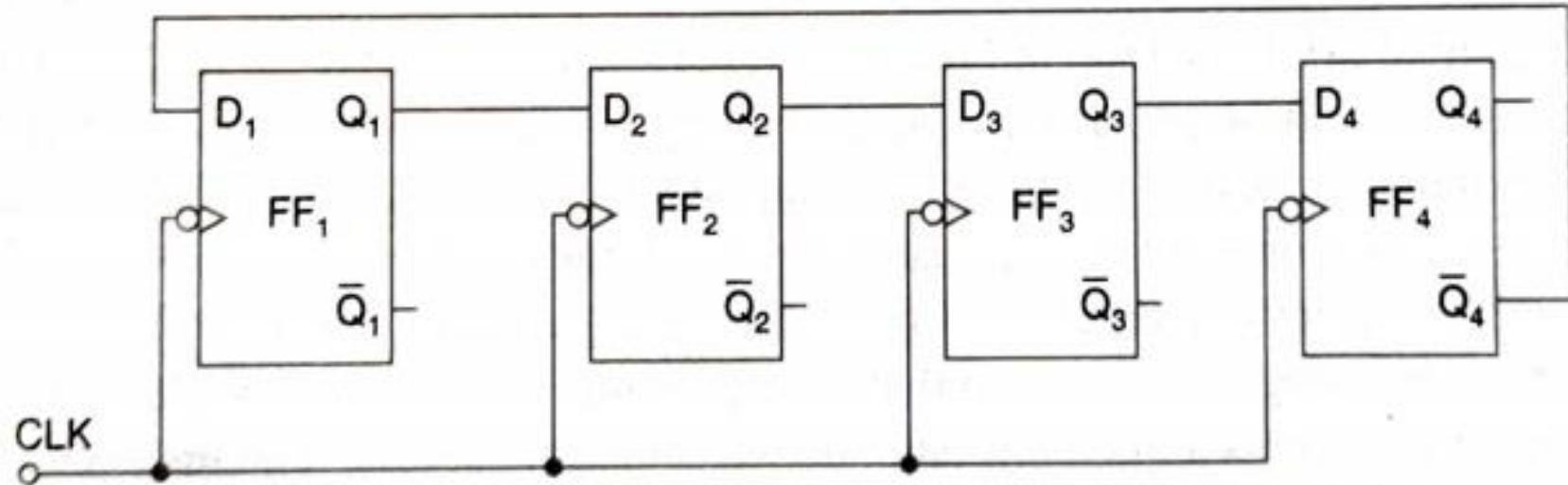


Timing diagram of a 4-bit ring counter.

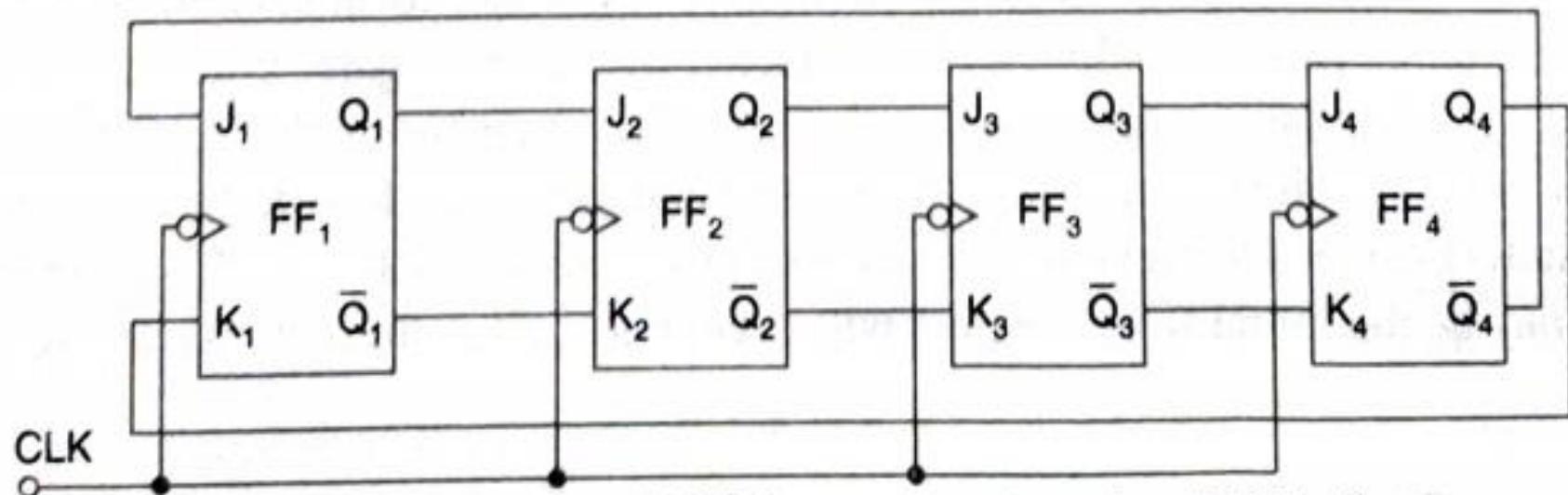
Johnson Counter

- A k -bit ring counter circulates a single bit among the flip-flops to provide k distinguishable states.
- The number of states can be doubled if the shift register is connected as a *switch-tail* ring counter / Johnson counter.
- A **switch-tail ring counter** is a circular shift register with the complemented output of the last flip-flop connected to the input of the first flip-flop.

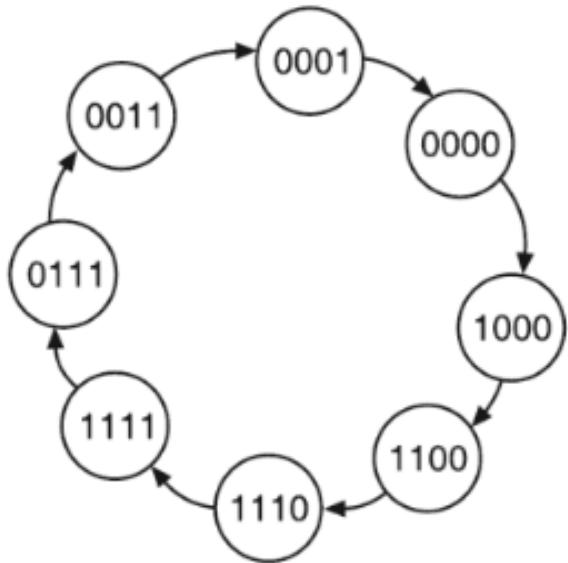
- This counter is obtained from a serial-in, serial-out shift register by providing feedback from the inverted output of the last FF to the D input of the first FF.
- The Q output of each stage is connected to the D input of the next stage, but the \bar{Q} output of the last stage is connected to the D input of first stage.



Logic diagram of a 4-bit Johnson counter using D flip-flops.



Logic diagram of a 4-bit Johnson counter using J-K flip-flops.



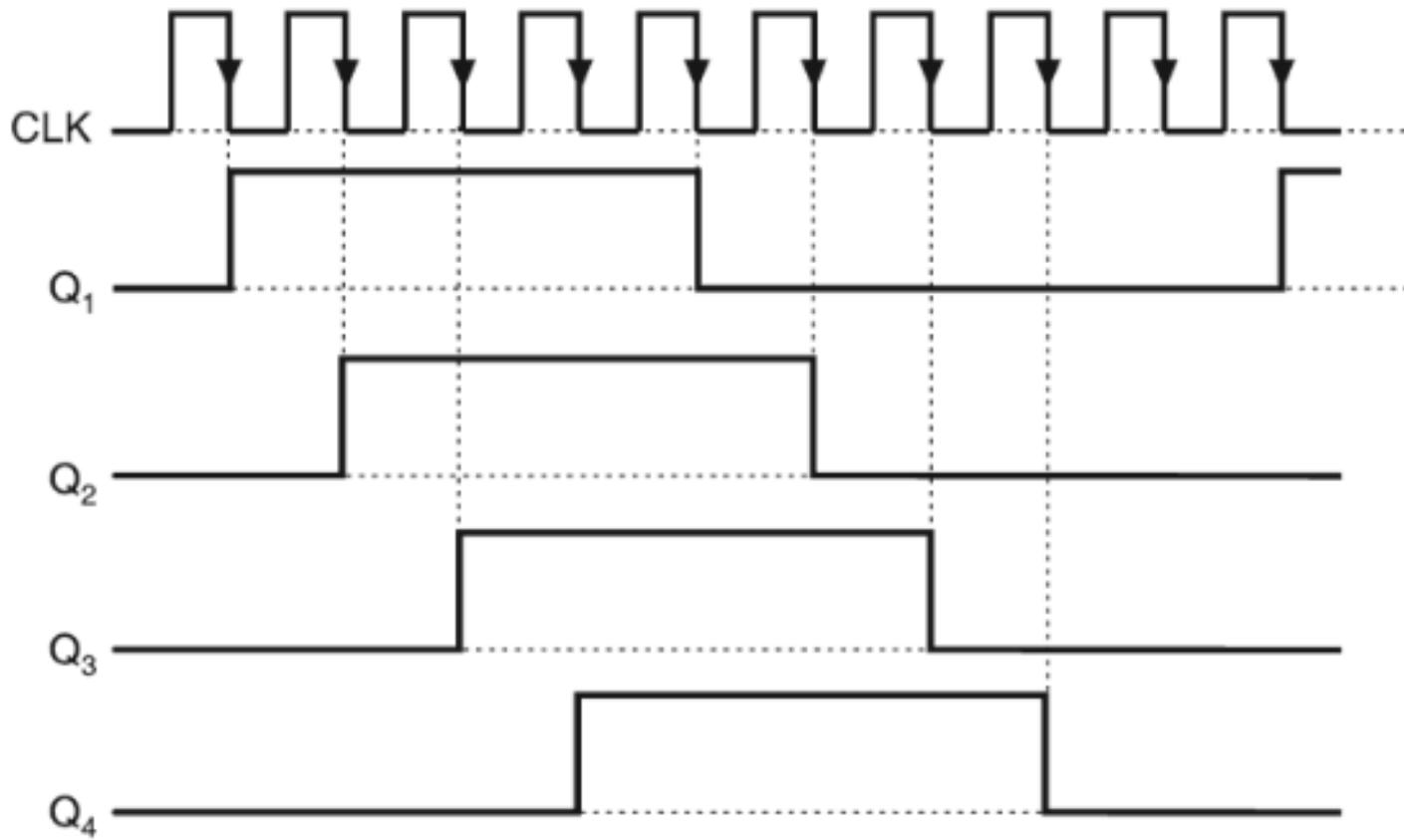
(a) State diagram

| Q_1 | Q_2 | Q_3 | Q_4 | After clock pulse |
|-------|-------|-------|-------|-------------------|
| 0 | 0 | 0 | 0 | 0 |
| 1 | 0 | 0 | 0 | 1 |
| 1 | 1 | 0 | 0 | 2 |
| 1 | 1 | 1 | 0 | 3 |
| 1 | 1 | 1 | 1 | 4 |
| 0 | 1 | 1 | 1 | 5 |
| 0 | 0 | 1 | 1 | 6 |
| 0 | 0 | 0 | 1 | 7 |
| 0 | 0 | 0 | 0 | 8 |
| 1 | 0 | 0 | 0 | 9 |

(b) Sequence table

State diagram and sequence table of a Johnson counter

- An n FF Johnson counter can have $2n$ unique states and can count up to $2n$ pulses. So it is a mod- $2n$ counter.



Timing diagram of a 4-bit Johnson counter

DIGITAL ELECTRONICS

III Semester CSE & CST



Mr. Srikanta Patnaik Mr. Ajit Kumar Patro Mr. Ami Kumar Parida

DEPARTMENT OF ELECTRONICS AND COMMUNICATION ENGINEERING

SCHOOL OF ENGINEERING & TECHNOLOGY

GIET UNIVERSITY, GUNUPUR, ODISHA

Memory and Programmable Logic

Introduction;
Random-Access Memory;
Memory Decoding;
Error Detection and Correction

Introduction

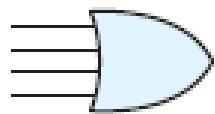
- A **memory unit** is a device to which binary information is transferred for storage and from which information is retrieved when needed for processing.
- Binary information received from an input device is stored in memory and information transferred to an output device is taken from memory.
- A **memory unit** is a collection of cells capable of storing a large quantity of binary information.

- There are two types of memories that are used in digital systems: *random-access memory* (RAM) and *read-only memory* (ROM).
- RAM stores new information for later use.
- The process of storing new information into memory is referred to as a memory *write* operation.
- The process of transferring the stored information out of memory is referred to as a memory *read* operation.
- RAM can perform both write and read operations.

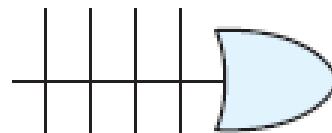
- ROM can perform only the read operation.
- This means that suitable binary information is already stored inside memory and can be retrieved or read at any time. However, that information cannot be altered by writing.
- ROM is a *programmable logic device* (PLD).
- The binary information that is stored within such a device is specified in some fashion and then embedded within the hardware in a process is referred to as *programming* the device.

- ROM is one example of a PLD. Other such units are the programmable logic array (PLA), programmable array logic (PAL), and the field-programmable gate array (FPGA).
- A PLD is an integrated circuit with internal logic gates connected through electronic paths that behave similarly to fuses.
- In the original state of the device, all the fuses are intact.

- Programming the device involves blowing those fuses along the paths that must be removed in order to obtain the particular configuration of the desired logic function.



(a) Conventional symbol



(b) Array logic symbol

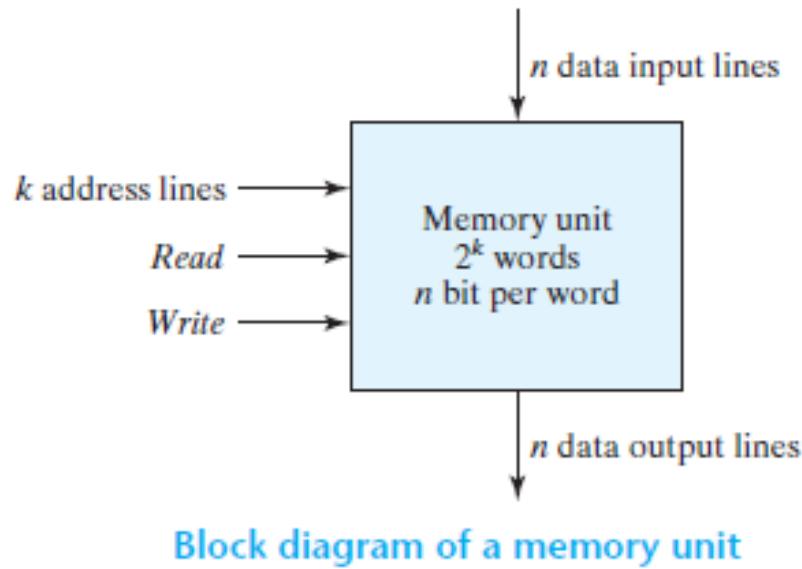
Conventional and array logic diagrams for OR gate

Random-Access Memory

- A memory unit is a collection of storage cells, together with associated circuits needed to transfer information into and out of a device.
- A memory unit stores binary information in groups of bits called *words*.
- A memory word is a group of 1's and 0's and may represent a number, an instruction, one or more alphanumeric characters, or any other binary-coded information.

- A group of 8 bits is called a *byte*.
- Most computer memories use words that are multiples of 8 bits in length.
- Thus, a 16-bit word contains two bytes, and a 32-bit word is made up of four bytes.
- The capacity of a memory unit is usually stated as the total number of bytes that the unit can store.

- Communication between memory and its environment is achieved through data input and output lines, address selection lines, and control lines that specify the direction of transfer.
- A block diagram of a memory unit is shown in Figure.



- The n data input lines provide the information to be stored in memory, and the n data output lines supply the information coming out of memory.
- The k address lines specify the particular word chosen among the many available.
- The two control inputs specify the direction of transfer desired: the *Write* input causes binary data to be transferred into the memory, and the *Read* input causes binary data to be transferred out of memory.

- The memory unit is specified by the number of words it contains and the number of bits in each word.
- It is customary to refer to the number of words (or bytes) in memory with one of the letters K (kilo), M (mega), and G (giga).
- K is equal to 2^{10} , M is equal to 2^{20} , and G is equal to 2^{30} .
- Thus, $64K = 2^6 \times 2^{10} = 2^{16}$, $2M = 2^1 \times 2^{20} = 2^{21}$, and $4G = 2^2 \times 2^{30} = 2^{32}$.

- Consider, for example, a memory unit with a capacity of 1K words of 16 bits each.
- Since $1K = 1,024 = 2^{10}$ and 16 bits constitute two bytes, we can say that the memory can accommodate $2,048 = 2K$ bytes.
- The $1K \times 16$ memory has 10 bits in the address and 16 bits in each word.
- As another example, a $64K \times 10$ memory will have 16 bits in the address (since $64K = 2^{16}$) and each word will consist of 10 bits.

- The number of address bits needed in a memory is dependent on the total number of words that can be stored in the memory and is independent of the number of bits in each word.
- The number of bits in the address is determined from the relationship $2^k \geq m$, where m is the total number of words and k is the number of address bits needed to satisfy the relationship.

Problem

- The memory units that follow are specified by the number of words times the number of bits per word. How many address lines and input-output data lines are needed in each case?
 - (a) $8K \times 16$
 - (b) $2G \times 8$
 - (c) $16M \times 32$
 - (d) $256K \times 64$

Solution:

- (a) $8K \times 16 = 2^3 \times 2^{10} \times 16 = 2^{13} \times 16$
Address lines = 13 and
Input-output data lines = 16
- (b) $2G \times 8 = 2^1 \times 2^{30} \times 8 = 2^{31} \times 8$
Address lines = 31 and
Input-output data lines = 8
- (c) $16M \times 32 = 2^4 \times 2^{20} \times 32 = 2^{24} \times 32$
Address lines = 24 and
Input-output data lines = 32
- (d) $256K \times 64 = 2^8 \times 2^{10} \times 64 = 2^{18} \times 64$
Address lines = 18 and
Input-output data lines = 64

Write and Read Operations

- The steps that must be taken for the purpose of transferring a new word to be stored into memory are as follows:
 1. Apply the binary address of the desired word to the address lines.
 2. Apply the data bits that must be stored in memory to the data input lines.
 3. Activate the *write* input.
- The memory unit will then take the bits from the input data lines and store them in the word specified by the address lines.

- The steps that must be taken for the purpose of transferring a stored word out of memory are as follows:
 1. Apply the binary address of the desired word to the address lines.
 2. Activate the *read* input.
- The memory unit will then take the bits from the word that has been selected by the address and apply them to the output data lines.

- Instead of having separate read and write inputs to control the two operations, most integrated circuits provide two other control inputs: one input selects the unit and the other determines the operation.

| Memory Enable | Read/Write | Memory Operation |
|----------------------|-------------------|-------------------------|
| 0 | X | None |
| 1 | 0 | Write to selected word |
| 1 | 1 | Read from selected word |

- The memory enable (sometimes called the chip select) is used to enable the particular memory chip in a multichip implementation of a large memory.

Types of Memories

- Integrated circuit RAM units are available in two operating modes: *static* and *dynamic*.
- Static RAM (SRAM) consists essentially of internal latches that store the binary information. The stored information remains valid as long as power is applied to the unit.
- Dynamic RAM (DRAM) stores the binary information in the form of electric charges on capacitors provided inside the chip by MOS transistors.

- The stored charge on the capacitors tends to discharge with time, and the capacitors must be periodically recharged by *refreshing* the dynamic memory.
- DRAM offers reduced power consumption and larger storage capacity in a single memory chip.
- SRAM is easier to use and has shorter read and write cycles.

- Memory units that lose stored information when power is turned off are said to be *volatile*.
- CMOS integrated circuit RAMs, both static and dynamic, are volatile memory, since the binary cells need external power to maintain the stored information.
- In contrast, a nonvolatile memory, such as magnetic disk, retains its stored information after the removal of power.

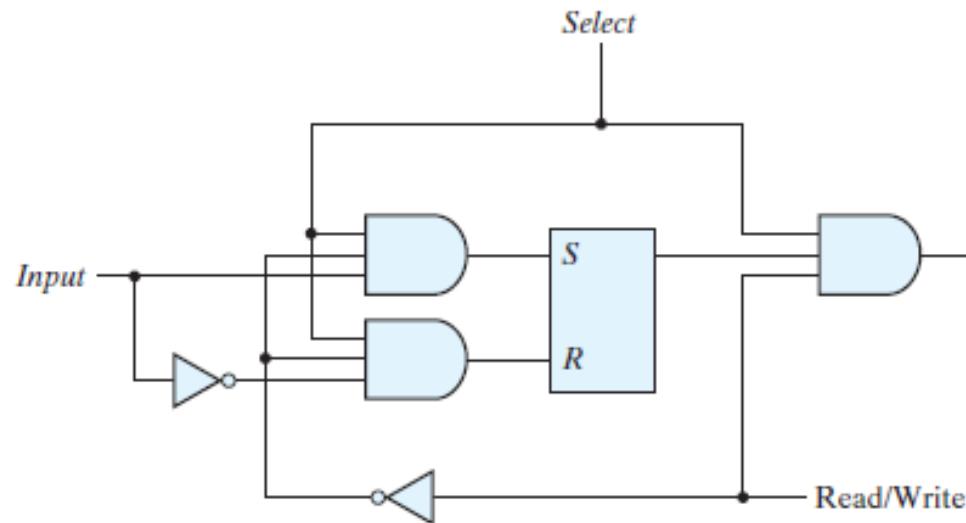
Memory Decoding

- In addition to requiring storage components in a memory unit, there is a need for decoding circuits to select the memory word specified by the input address.

Internal Construction

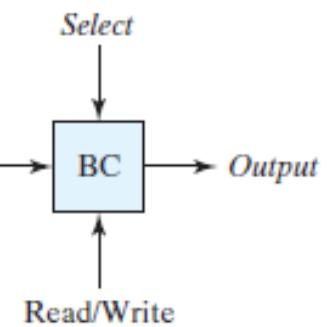
- The internal construction of a RAM of m words and n bits per word consists of $m \times n$ binary storage cells and associated decoding circuits for selecting individual words.
- The binary storage cell is the basic building block of a memory unit.
- The equivalent logic of a binary cell that stores one bit of information is shown in Figure.

- The storage part of the cell is modeled by an *SR* latch with associated gates to form a *D* latch.



(a) Logic diagram

Memory cell



(b) Block diagram

- The binary cell stores one bit in its internal latch.
- The select input enables the cell for reading or writing, and the read/write input determines the operation of the cell when it is selected.
- A **1** in the read/write input provides the read operation by forming a path from the latch to the output terminal.
- A **0** in the read/write input provides the write operation by forming a path from the input terminal to the latch.

- The logical construction of a 4×4 RAM is shown in Figure.
- The RAM consists of four words of four bits each and has a total of 16 binary cells.
- The small blocks labeled BC represent the binary cell with its three inputs and one output.
- A memory with four words needs two address lines. The two address inputs go through a 2×4 decoder to select one of the four words.
- The decoder is enabled with the memory-enable input.

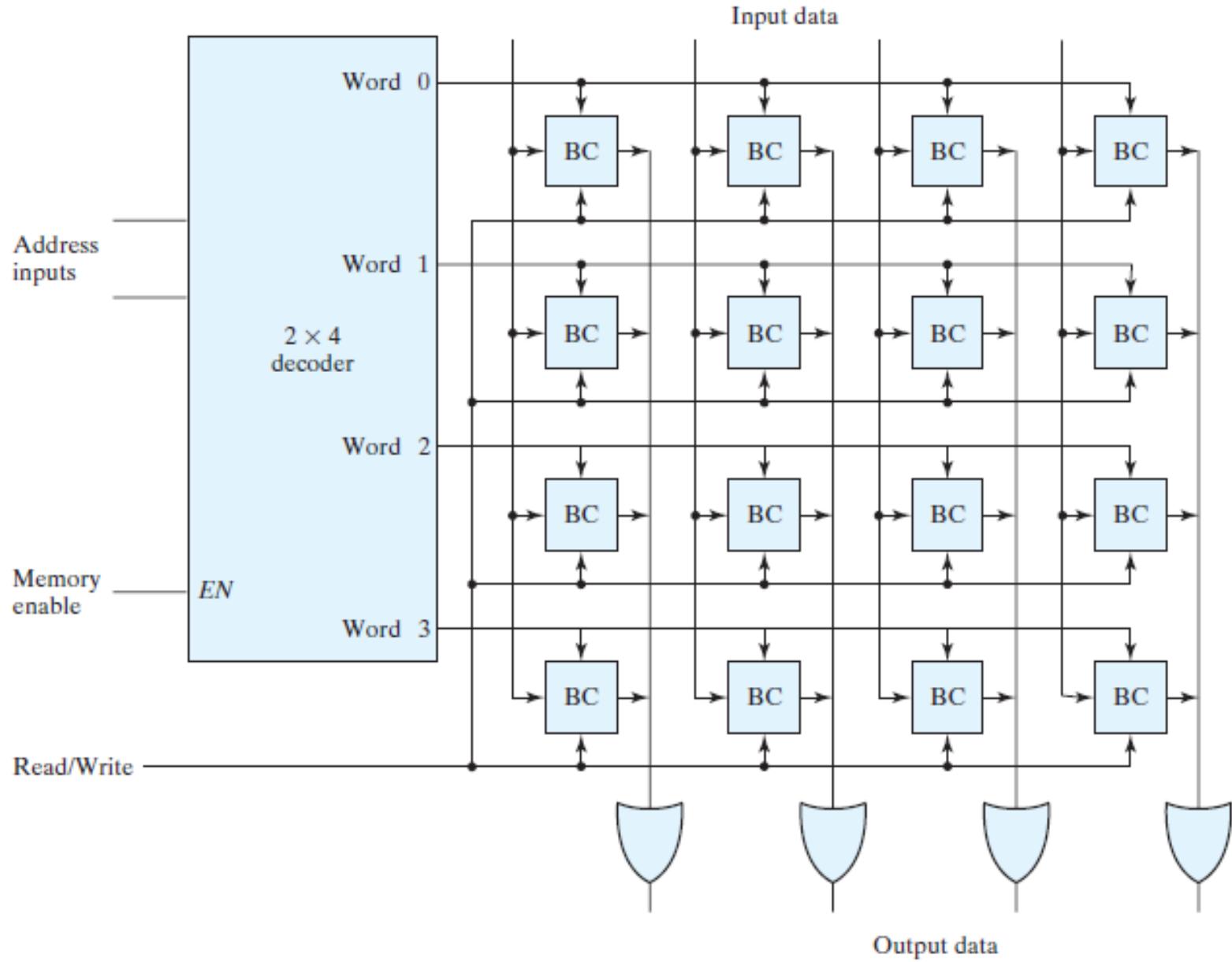


Diagram of a 4×4 RAM

- When the memory enable is 0, all outputs of the decoder are 0 and none of the memory words are selected.
- With the memory select at 1, one of the four words is selected, dictated by the value in the two address lines.
- Once a word has been selected, the read/write input determines the operation.
- During the read operation, the four bits of the selected word go through OR gates to the output terminals.

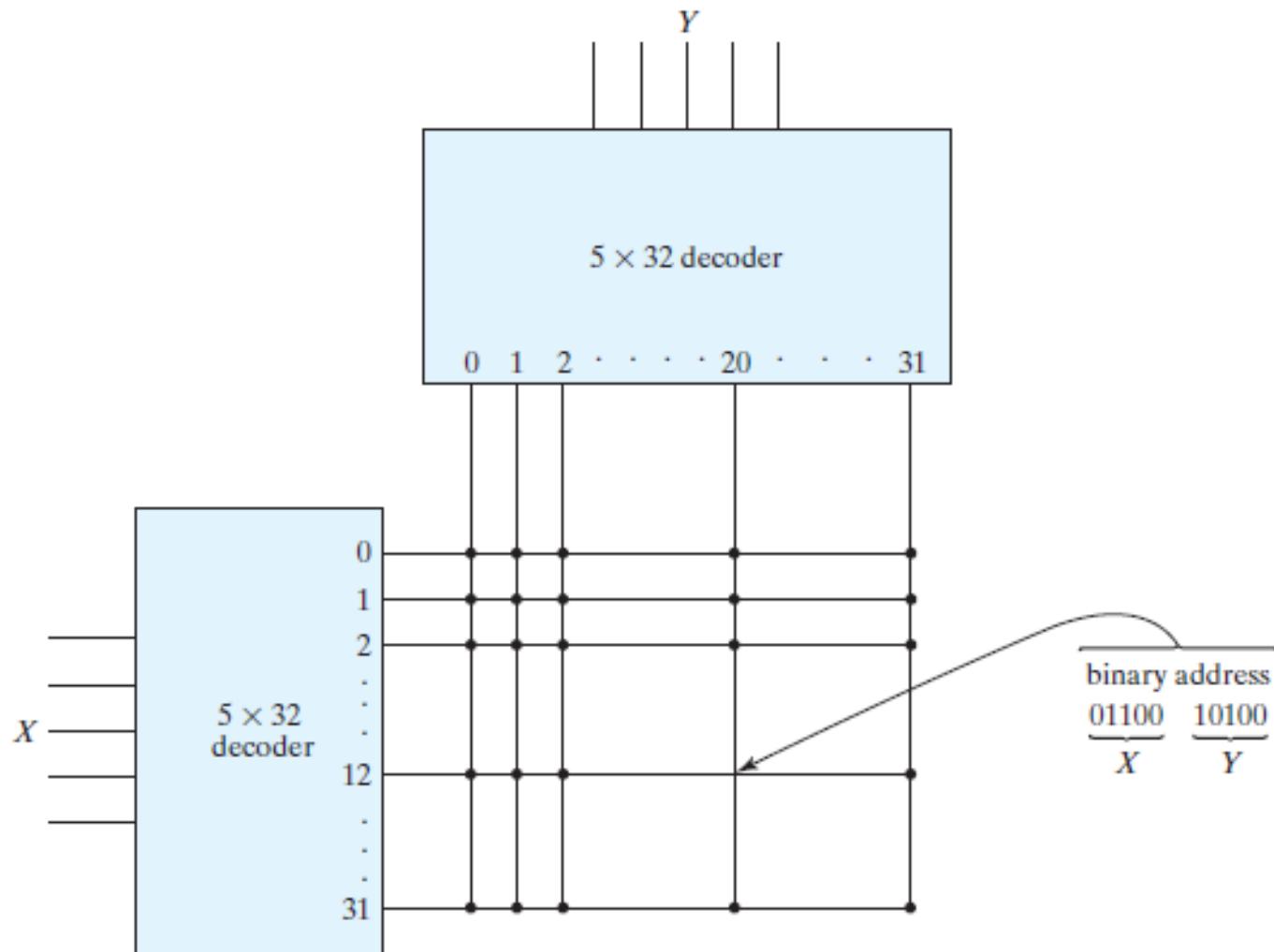
- During the write operation, the data available in the input lines are transferred into the four binary cells of the selected word.
- The binary cells that are not selected are disabled, and their previous binary values remain unchanged.
- When the memory select input that goes into the decoder is equal to 0, none of the words are selected and the contents of all cells remain unchanged regardless of the value of the read/write input.

- A memory with 2^k words of n bits per word requires k address lines that go into a $k \times 2^k$ decoder.
- Each one of the decoder outputs selects one word of n bits for reading or writing.

Coincident Decoding

- A decoder with k inputs and 2^k outputs requires 2^k AND gates with k inputs per gate.
- The total number of gates and the number of inputs per gate can be reduced by employing two decoders in a two-dimensional selection scheme.
- The basic idea in two-dimensional decoding is to arrange the memory cells in an array that is close as possible to square.
- In this configuration, two $k/2$ -input decoders are used instead of one k -input decoder.

- One decoder performs the row selection and the other the column selection in a two-dimensional matrix configuration.
- The two-dimensional selection pattern is demonstrated in Figure for a 1K-word memory. Instead of using a single $10 \times 1,024$ decoder, we use two 5×32 decoders.
- With the single decoder, we would need 1,024 AND gates with 10 inputs in each. In the two-decoder case, we need 64 AND gates with 5 inputs in each.



Two-dimensional decoding structure for a 1K-word memory

- The five most significant bits of the address go to input X and the five least significant bits go to input Y .
- Each word within the memory array is selected by the coincidence of one X line and one Y line.
- Thus, each word in memory is selected by the coincidence between 1 of 32 rows and 1 of 32 columns, for a total of 1,024 words.
- Note that each intersection represents a word that may have any number of bits.

- As an example, consider the word whose address is 404.
- The 10-bit binary equivalent of 404 is 01100 10100.
- This makes $X = 01100$ (binary 12) and $Y = 10100$ (binary 20).
- The n -bit word that is selected lies in the X decoder output number 12 and the Y decoder output number 20.
- All the bits of the word are selected for reading or writing.

Problem

- (a) How many $32K \times 8$ RAM chips are needed to provide a memory capacity of 256K bytes?
- (b) How many lines of the address must be used to access 256K bytes? How many of these lines are connected to the address inputs of all chips?
- (c) How many lines must be decoded for the chip select inputs? Specify the size of the decoder.

Solution:

- $32K \times 8 = 2^5 \times 2^{10} \times 8 = 2^{15} \times 8$
 - $256K \text{ bytes} = 256K \times 8 = 2^8 \times 2^{10} \times 8 = 2^{18} \times 8$
- (a) $2^{18} \times 8 / 2^{15} \times 8 = 2^{18-15} = 2^3 = 8$

Eight $32K \times 8$ RAM chips are needed to provide a memory capacity of $256K$ bytes.

- (b) 18 lines of the address must be used to access $256K$ bytes. 15 of these lines are connected to the address inputs of all chips.
- (c) $18 - 15 = 3$ lines must be decoded for the chip select inputs. The size of the decoder is $3 \times 2^3 = 3 \times 8$.

Error Detection and Correction

- The dynamic physical interaction of the electrical signals affecting the data path of a memory unit may cause occasional errors in storing and retrieving the binary information.
- The reliability of a memory unit may be improved by employing error-detecting and error-correcting codes.

- The most common error detection scheme is the parity bit. A parity bit is generated and stored along with the data word in memory. The parity of the word is checked after reading it from memory.
- An error-correcting code generates multiple parity check bits that are stored with the data word in memory. Each check bit is a parity over a group of bits in the data word.

Hamming Code

- One of the most common error-correcting codes used in RAMs was devised by R. W. Hamming.
- In the Hamming code, k parity bits are added to an n -bit data word, forming a new word of $n + k$ bits.
- The bit positions are numbered in sequence from 1 to $n + k$.
- Those positions numbered as a power of 2 are reserved for the parity bits. The remaining bits are the data bits.
- The code can be used with words of any length.

- Consider, for example, the 8-bit data word 11000100.
- We include 4 parity bits with the 8-bit word and arrange the 12 bits as follows:

| Bit position: | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | 10 | 11 | 12 |
|---------------|-------|-------|---|-------|---|---|---|-------|---|----|----|----|
| | P_1 | P_2 | 1 | P_4 | 1 | 0 | 0 | P_8 | 0 | 1 | 0 | 0 |

- The 4 parity bits, P_1 , P_2 , P_4 , and P_8 , are in positions 1, 2, 4, and 8, respectively.
- The 8 bits of the data word are in the remaining positions.

- Each parity bit is calculated as follows:

$$P_1 = \text{XOR of bits } (3, 5, 7, 9, 11)$$

$$= 1 \oplus 1 \oplus 0 \oplus 0 \oplus 0 = 0$$

$$P_2 = \text{XOR of bits } (3, 5, 7, 10, 11)$$

$$= 1 \oplus 0 \oplus 0 \oplus 1 \oplus 0 = 0$$

$$P_4 = \text{XOR of bits } (5, 6, 7, 12)$$

$$= 1 \oplus 0 \oplus 0 \oplus 0 = 1$$

$$P_8 = \text{XOR of bits } (9, 10, 11, 12)$$

$$= 0 \oplus 1 \oplus 0 \oplus 0 = 1$$

- The 8-bit data word is stored in memory together with the 4 parity bits as a 12-bit composite word.
- Substituting the 4 P bits in their proper positions, we obtain the 12-bit composite word stored in memory:

| Bit position: | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | 10 | 11 | 12 |
|---------------|---|---|---|---|---|---|---|---|---|----|----|----|
| | 0 | 0 | 1 | 1 | 1 | 0 | 0 | 1 | 0 | 1 | 0 | 0 |

- When the 12 bits are read from memory, they are checked again for errors.
- The parity is checked over the same combination of bits, including the parity bit.
- The 4 check bits are evaluated as follows:

$$C_1 = \text{XOR of bits } (1, 3, 5, 7, 9, 11)$$

$$C_2 = \text{XOR of bits } (2, 3, 5, 7, 10, 11)$$

$$C_4 = \text{XOR of bits } (4, 5, 6, 7, 12)$$

$$C_8 = \text{XOR of bits } (8, 9, 10, 11, 12)$$

- A 0 check bit designates even parity over the checked bits and a 1 designates odd parity.
- Since the bits were stored with even parity, the result, $C = C_8C_4C_2C_1 = 0000$, indicates that no error has occurred.
- However, if $C \neq 0$, then the 4-bit binary number formed by the check bits gives the position of the erroneous bit.

- For example, consider the following three cases:

| Bit position: | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | 10 | 11 | 12 | |
|---------------|---|---|---|---|---|---|---|---|---|----|----|----|----------------|
| | 0 | 0 | 1 | 1 | 1 | 0 | 0 | 1 | 0 | 1 | 0 | 0 | No error |
| | 1 | 0 | 1 | 1 | 1 | 0 | 0 | 1 | 0 | 1 | 0 | 0 | Error in bit 1 |
| | 0 | 0 | 1 | 1 | 0 | 0 | 0 | 1 | 0 | 1 | 0 | 0 | Error in bit 5 |

- In the first case, there is no error in the 12-bit word.
- In the second case, there is an error in bit position number 1 because it changed from 0 to 1.
- The third case shows an error in bit position 5, with a change from 1 to 0.

- Evaluating the XOR of the corresponding bits, we determine the 4 check bits to be as follows:

$C_8 \ C_4 \ C_2 \ C_1$

For no error: 0 0 0 0

With error in bit 1: 0 0 0 1

With error in bit 5: 0 1 0 1

- Thus, for no error, we have $C = 0000$; with an error in bit 1, we obtain $C = 0001$; and with an error in bit 5, we get $C = 0101$.

- When the binary number C is not equal to 0000, it gives the position of the bit in error. The error can be corrected by complementing the corresponding bit.
- Note that an error can occur in the data word or in one of the parity bits.

- The Hamming code can be used for data words of any length.
- In general, the Hamming code consists of k check bits and n data bits, for a total of $n + k$ bits.
- The range of k must be equal to or greater than $n + k$, giving the relationship $2^k - 1 \geq n + k$
- Solving for n in terms of k , we obtain $2^k - 1 - k \geq n$
- This relationship gives a formula for establishing the number of data bits that can be used in conjunction with k check bits.

Range of Data Bits for k Check Bits

| Number of Check Bits, k | Range of Data Bits, n |
|---|---|
| 3 | 2–4 |
| 4 | 5–11 |
| 5 | 12–26 |
| 6 | 27–57 |
| 7 | 58–120 |

- The grouping of bits for parity generation and checking can be determined from a list of the binary numbers from 0 through $2^k - 1$.
- The least significant bit is a 1 in the binary numbers 1, 3, 5, 7, and so on. The second significant bit is a 1 in the binary numbers 2, 3, 6, 7, and so on.
- Note that each group of bits starts with a number that is a power of 2: 1, 2, 4, 8, 16, etc. These numbers are also the position numbers for the parity bits.

Single-Error Correction, Double-Error Detection

- The Hamming code can detect and correct only a single error.
- By adding another parity bit to the coded word, the Hamming code can be used to correct a single error and detect double errors.
- If we include this additional parity bit, then the previous 12-bit coded word becomes $001110010100P_{13}$, where P_{13} is evaluated from the exclusive-OR of the other 12 bits.

- This produces the 13-bit word 0011100101001 (even parity).
- When the 13-bit word is read from memory, the check bits are evaluated, as is the parity P over the entire 13 bits.
- If $P = 0$, the parity is correct (even parity), but if $P = 1$, then the parity over the 13 bits is incorrect (odd parity).

- The following four cases can arise:
 - If $C = 0$ and $P = 0$, no error occurred.
 - If $C \neq 0$ and $P = 1$, a single error occurred that can be corrected.
 - If $C \neq 0$ and $P = 0$, a double error occurred that is detected, but that cannot be corrected.
 - If $C = 0$ and $P = 1$, an error occurred in the P_{13} bit.
- This scheme may detect more than two errors, but is not guaranteed to detect all such errors.
- Integrated circuits use a modified Hamming code to generate and check parity bits for single-error correction and double-error detection.

DIGITAL ELECTRONICS

III Semester CSE & CST



Mr. Srikanta Patnaik

Mr. Ajit Kumar Patro

Mr. Ami Kumar Parida

DEPARTMENT OF ELECTRONICS AND COMMUNICATION ENGINEERING

SCHOOL OF ENGINEERING & TECHNOLOGY

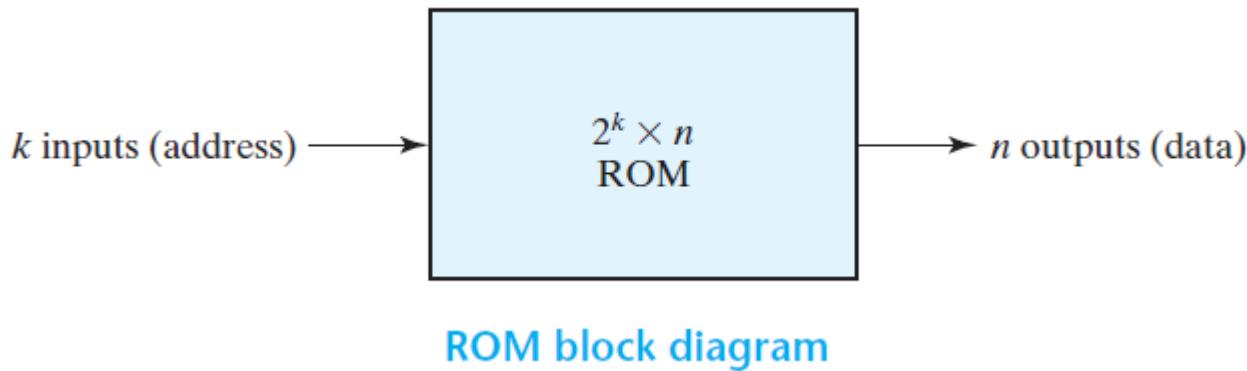
GIET UNIVERSITY, GUNUPUR, ODISHA

**Read-Only Memory;
Programmable Logic Array;
Programmable Array Logic;
Sequential Programmable Devices**

Read-Only Memory

- A read-only memory (ROM) is essentially a memory device in which permanent binary information is stored.
- The binary information must be specified by the designer and is then embedded in the unit to form the required interconnection pattern.
- Once the pattern is established, it stays within the unit even when power is turned off and on again.

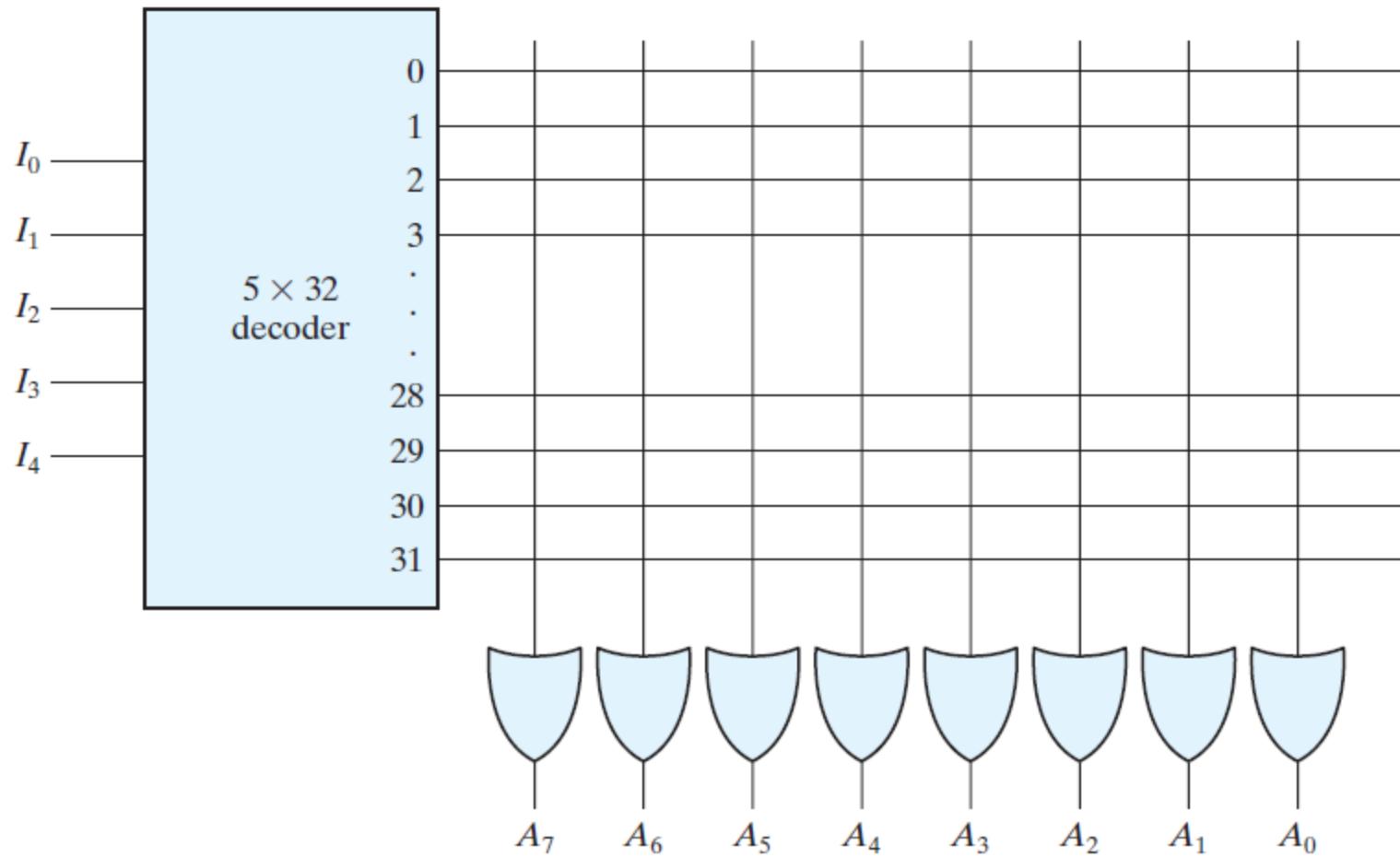
- A block diagram of a ROM consisting of k inputs and n outputs is shown in Figure.



- The inputs provide the address for memory, and the outputs give the data bits of the stored word that is selected by the address.

- The number of words in a ROM is determined from the fact that k address input lines are needed to specify 2^k words.
- Note that ROM does not have data inputs, because it does not have a write operation.
- Integrated circuit ROM chips have one or more enable inputs and sometimes come with three-state outputs to facilitate the construction of large arrays of ROM.

- Consider, for example, a 32×8 ROM. The unit consists of 32 words of 8 bits each.
- There are five input lines that form the binary numbers from 0 through 31 for the address.
- Figure shows the internal logic construction of 32×8 ROM.
- The five inputs are decoded into 32 distinct outputs by means of a 5×32 decoder. Each output of the decoder represents a memory address.
- The 32 outputs of the decoder are connected to each of the eight OR gates.



Internal logic of a 32×8 ROM

- Since each OR gate has 32 input connections and there are 8 OR gates, the ROM contains $32 \times 8 = 256$ internal connections.
- In general, a $2^k \times n$ ROM will have an internal $k \times 2^k$ decoder and n OR gates. Each OR gate has 2^k inputs, which are connected to each of the outputs of the decoder.

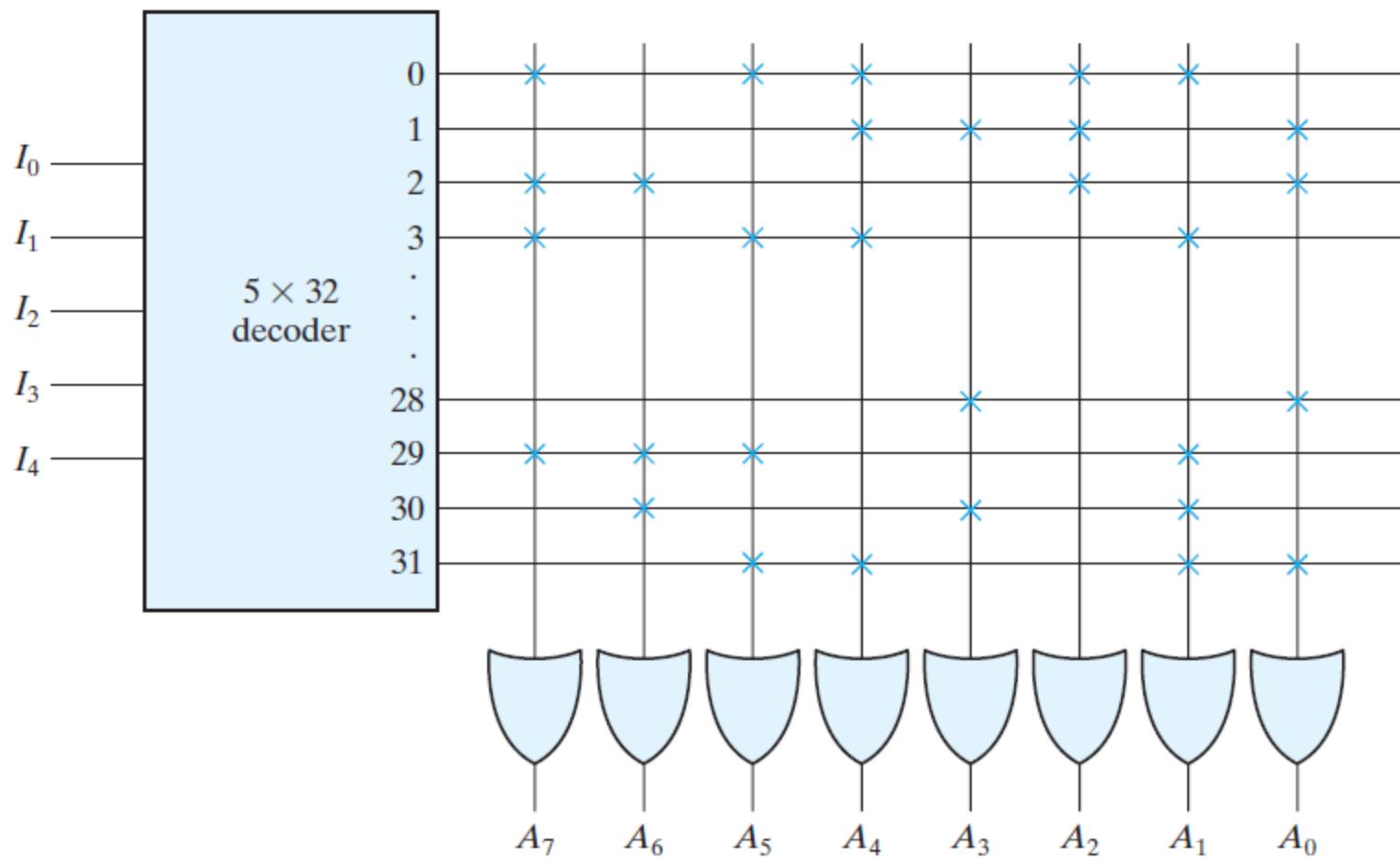
- The 256 intersections are programmable.
- A programmable connection between two lines is logically equivalent to a switch that can be altered to be either closed (meaning that the two lines are connected) or open (meaning that the two lines are disconnected).
- The programmable intersection between two lines is sometimes called a *crosspoint*.
- Various physical devices are used to implement crosspoint switches.

- One of the simplest technologies employs a fuse that normally connects the two points, but is opened or “blown” by the application of a high-voltage pulse into the fuse.
- The internal binary storage of a ROM is specified by a truth table that shows the word content in each address.
- For example, the content of a 32×8 ROM may be specified with a truth table similar to the one shown in Table.

ROM Truth Table (Partial)

| Inputs | | | | | Outputs | | | | | | | |
|--------|-------|-------|-------|-------|---------|-------|-------|-------|-------|-------|-------|-------|
| I_4 | I_3 | I_2 | I_1 | I_0 | A_7 | A_6 | A_5 | A_4 | A_3 | A_2 | A_1 | A_0 |
| 0 | 0 | 0 | 0 | 0 | 1 | 0 | 1 | 1 | 0 | 1 | 1 | 0 |
| 0 | 0 | 0 | 0 | 1 | 0 | 0 | 0 | 1 | 1 | 1 | 0 | 1 |
| 0 | 0 | 0 | 1 | 0 | 1 | 1 | 0 | 0 | 0 | 1 | 0 | 1 |
| 0 | 0 | 0 | 1 | 1 | 1 | 0 | 1 | 1 | 0 | 0 | 1 | 0 |
| ⋮ | | | | | ⋮ | | | | | | | |
| 1 | 1 | 1 | 0 | 0 | 0 | 0 | 0 | 0 | 1 | 0 | 0 | 1 |
| 1 | 1 | 1 | 0 | 1 | 1 | 1 | 1 | 0 | 0 | 0 | 1 | 0 |
| 1 | 1 | 1 | 1 | 0 | 0 | 1 | 0 | 0 | 1 | 0 | 1 | 0 |
| 1 | 1 | 1 | 1 | 1 | 0 | 0 | 1 | 1 | 0 | 0 | 1 | 1 |

- The hardware procedure that programs the ROM blows fuse links in accordance with a given truth table.
- For example, programming the ROM according to the truth table given by Table results in the configuration shown in Figure.
- Every 0 listed in the truth table specifies the absence of a connection, and every 1 listed specifies a path that is obtained by a connection.



Programming the ROM according to Table

Combinational Circuit Implementation

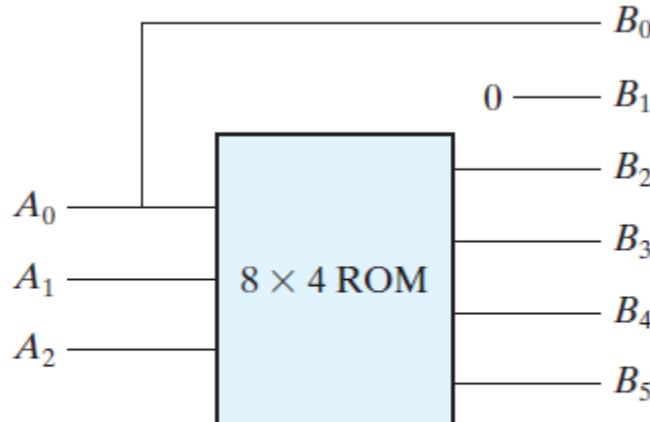
- The ROM is essentially a device that includes both the decoder and the OR gates within a single device to form a minterm generator.
- By choosing connections for those minterms which are included in the function, the ROM outputs can be programmed to represent the Boolean functions of the output variables in a combinational circuit.

Example

- Design a combinational circuit using a ROM. The circuit accepts a three-bit number and outputs a binary number equal to the square of the input number.

Truth Table for Circuit

| Inputs | | | Outputs | | | | | | Decimal |
|--------|-------|-------|---------|-------|-------|-------|-------|-------|---------|
| A_2 | A_1 | A_0 | B_5 | B_4 | B_3 | B_2 | B_1 | B_0 | |
| 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 |
| 0 | 0 | 1 | 0 | 0 | 0 | 0 | 0 | 1 | 1 |
| 0 | 1 | 0 | 0 | 0 | 0 | 1 | 0 | 0 | 4 |
| 0 | 1 | 1 | 0 | 0 | 1 | 0 | 0 | 1 | 9 |
| 1 | 0 | 0 | 0 | 1 | 0 | 0 | 0 | 0 | 16 |
| 1 | 0 | 1 | 0 | 1 | 1 | 0 | 0 | 1 | 25 |
| 1 | 1 | 0 | 1 | 0 | 0 | 1 | 0 | 0 | 36 |
| 1 | 1 | 1 | 1 | 1 | 0 | 0 | 0 | 1 | 49 |



(a) Block diagram

| A_2 | A_1 | A_0 | B_5 | B_4 | B_3 | B_2 |
|-------|-------|-------|-------|-------|-------|-------|
| 0 | 0 | 0 | 0 | 0 | 0 | 0 |
| 0 | 0 | 1 | 0 | 0 | 0 | 0 |
| 0 | 1 | 0 | 0 | 0 | 0 | 1 |
| 0 | 1 | 1 | 0 | 0 | 1 | 0 |
| 1 | 0 | 0 | 0 | 1 | 0 | 0 |
| 1 | 0 | 1 | 0 | 1 | 1 | 0 |
| 1 | 1 | 0 | 1 | 0 | 0 | 0 |
| 1 | 1 | 1 | 1 | 1 | 0 | 0 |

(b) ROM truth table

ROM implementation

Problems

- Implement the following Boolean function using ROM.

$$F_1(A, B) = \Sigma(1, 3)$$

$$F_2(A, B) = \Sigma(0, 1, 2)$$

$$F_3(A, B) = \Sigma(0, 3)$$

- Design a full-adder using ROM.
- Design a combinational circuit that accepts 3-input and produces its 1's complement as output.

Types of ROMs

- The required paths in a ROM may be programmed in four different ways.
- The first is called *mask programming* and is done by the semiconductor company during the last fabrication process of the unit.
- The procedure for fabricating a ROM requires that the customer fill out the truth table he or she wishes the ROM to satisfy.

- The truth table may be submitted in a special form provided by the manufacturer or in a specified format on a computer output medium.
- The manufacturer makes the corresponding mask for the paths to produce the 1's and 0's according to the customer's truth table.
- This procedure is costly because the vendor charges the customer a special fee for custom masking the particular ROM.
- For this reason, mask programming is economical only if a large quantity of the same ROM configuration is to be ordered.

- For small quantities, it is more economical to use a second type of ROM called *programmable read-only memory*, or PROM.
- When ordered, PROM units contain all the fuses intact, giving all 1's in the bits of the stored words.
- The fuses in the PROM are blown by the application of a high-voltage pulse to the device through a special pin.
- A blown fuse defines a binary 0 state and an intact fuse gives a binary 1 state.

- This procedure allows the user to program the PROM in the laboratory to achieve the desired relationship between input addresses and stored words.
- Special instruments called PROM programmers are available commercially to facilitate the procedure.
- In any case, all procedures for programming ROMs are hardware procedures, even though the word *programming* is used.

- The hardware procedure for programming ROMs or PROMs is irreversible, and once programmed, the fixed pattern is permanent and cannot be altered.
- Once a bit pattern has been established, the unit must be discarded if the bit pattern is to be changed.
- A third type of ROM is the *erasable PROM*, or EPROM, which can be restructured to the initial state even though it has been programmed previously.

- When the EPROM is placed under a special ultraviolet light for a given length of time, the shortwave radiation discharges the internal floating gates that serve as the programmed connections.
- After erasure, the EPROM returns to its initial state and can be reprogrammed to a new set of values.

- The fourth type of ROM is the *electrically erasable PROM* (EEPROM or E²PROM).
- This device is like the EEPROM, except that the previously programmed connections can be erased with an electrical signal instead of ultraviolet light.
- The advantage is that the device can be erased without removing it from its socket.

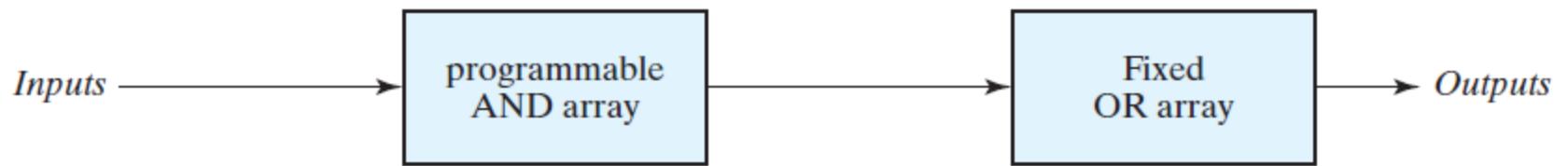
- Flash memory devices are similar to EEPROMs, but have additional built-in circuitry to selectively program and erase the device in-circuit, without the need for a special programmer.
- They have widespread application in modern technology in cell phones, digital cameras, set-top boxes, digital TV, telecommunications, non-volatile data storage, and microcontrollers.
- Their low consumption of power makes them an attractive storage medium for laptop and notebook computers.

Combinational PLDs

- The PROM is a combinational programmable logic device (PLD) - an integrated circuit with programmable gates divided into an AND array and an OR array to provide an AND-OR sum-of-product implementation.
- There are three major types of combinational PLDs, differing in the placement of the programmable connections in the AND-OR array.
- Figure shows the configuration of the three PLDs.



(a) Programmable read-only memory (PROM)



(b) Programmable array logic (PAL)



(c) Programmable logic array (PLA)

Basic configuration of three PLDs

- The **PROM** has a fixed AND array constructed as a decoder and a programmable OR array. The programmable OR gates implement the Boolean functions in sum-of-minterms form.
- The **PAL** has a programmable AND array and a fixed OR array. The AND gates are programmed to provide the product terms for the Boolean functions, which are logically summed in each OR gate.

- The most flexible PLD is the **PLA**, in which both the AND and OR arrays can be programmed. The product terms in the AND array may be shared by any OR gate to provide the required sum-of-products implementation.
- The names PAL and PLA emerged from different vendors during the development of PLDs.

Programmable Logic Array

- The PLA is similar in concept to the PROM, except that the PLA does not provide full decoding of the variables and does not generate all the minterms.
- The decoder is replaced by an array of AND gates that can be programmed to generate any product term of the input variables.
- The product terms are then connected to OR gates to provide the sum of products for the required Boolean functions.

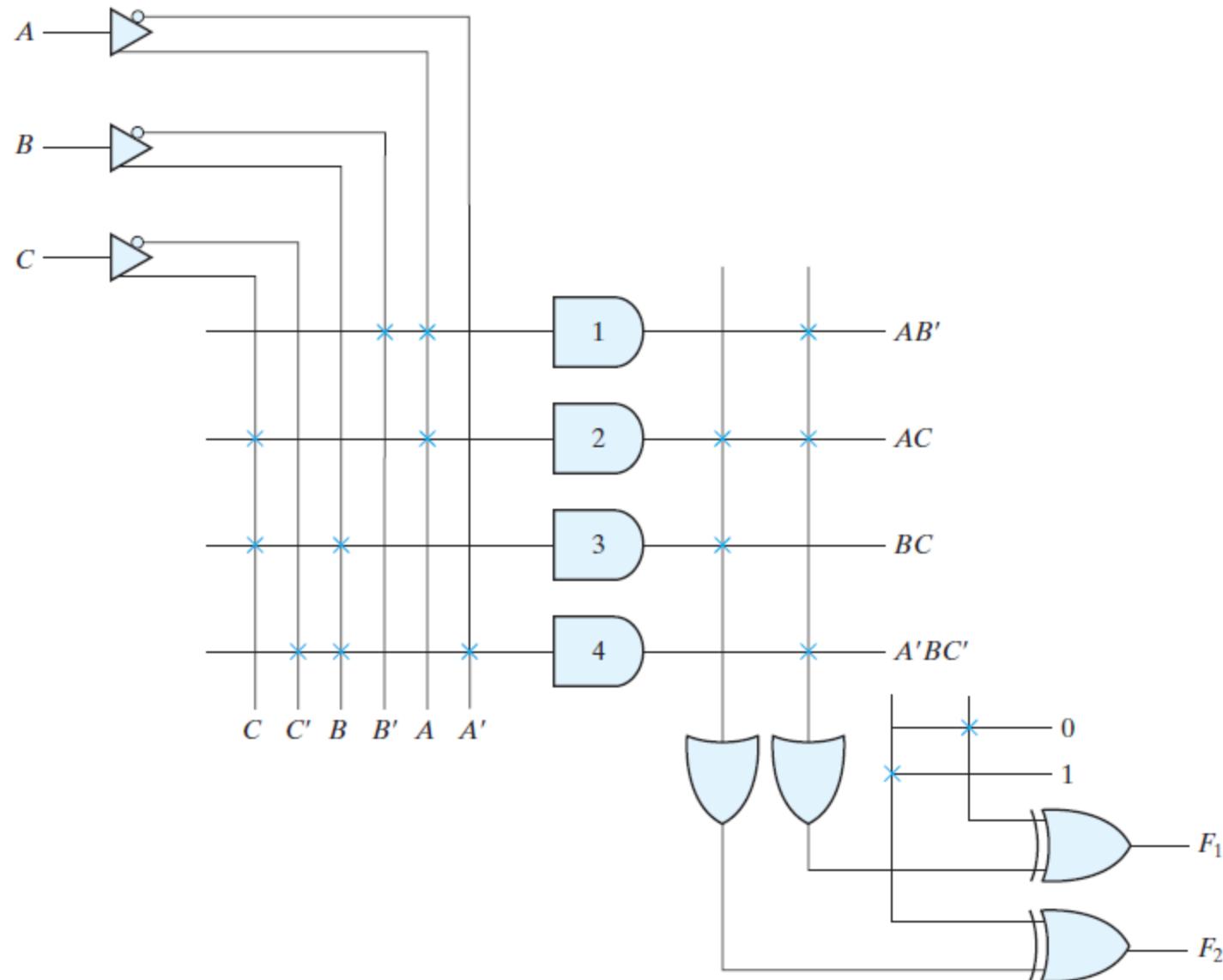
- The output of the OR gate goes to an XOR gate, where the other input can be programmed to receive a signal equal to either logic 1 or logic 0.
- The output is inverted when the XOR input is connected to 1 (since $x \oplus 1 = x'$).
- The output does not change when the XOR input is connected to 0 (since $x \oplus 0 = x$).

Example

- Implement the following two Boolean functions with a PLA:

$$F_1 = AB' + AC + A'BC'$$

$$F_2 = (AC + BC)'$$



PLA with three inputs, four product terms, and two outputs

PLA Programming Table

| Product Term | Inputs | | | Outputs | |
|---------------------|----------------------|----------|----------|----------------------|------------|
| | A | B | C | (T) | (C) |
| | F₁ | | | F₂ | |
| AB' | 1 | 1 | 0 | — | 1 |
| AC | 2 | 1 | — | 1 | 1 |
| BC | 3 | — | 1 | 1 | — |
| $A'BC'$ | 4 | 0 | 1 | 0 | 1 |

Example

- Implement the following two Boolean functions with a PLA:

$$F_1(A, B, C) = \sum(0, 1, 2, 4)$$

$$F_2(A, B, C) = \sum(0, 5, 6, 7)$$

- The two functions are simplified by the maps. Both the true value and the complement of the functions are simplified into sum-of-products form.

| | | BC | 00 | 01 | 11 | 10 |
|---|---|-------|----|----|----|----|
| | | A | 0 | 1 | 0 | 1 |
| 0 | 0 | m_0 | 1 | 1 | 0 | 1 |
| | 1 | m_4 | 1 | 0 | 0 | 0 |
| 1 | 0 | m_1 | 1 | 0 | 1 | 0 |
| | 1 | m_5 | 0 | 0 | 1 | 1 |

$$F_1 = A'B' + A'C' + B'C'$$

$$F_1 = (AB + AC + BC)'$$

| | | BC | 00 | 01 | 11 | 10 |
|---|---|-------|----|----|----|----|
| | | A | 0 | 1 | 0 | 1 |
| 0 | 0 | m_0 | 1 | 0 | 0 | 0 |
| | 1 | m_4 | 0 | 1 | 1 | 1 |
| 1 | 0 | m_1 | 0 | 1 | 0 | 1 |
| | 1 | m_5 | 1 | 0 | 1 | 0 |

$$F_2 = AB + AC + A'B'C'$$

$$F_2 = (A'B + A'C + AB'C)'$$

- The combination that gives the minimum number of product terms is

$$F_1 = (AB + AC + BC)'$$

and

$$F_2 = AB + AC + A'B'C'$$

PLA programming table

| Product term | Inputs | | | Outputs | |
|--------------|--------|---|---|---------|-------|
| | A | B | C | (C) | (T) |
| | | | | F_1 | F_2 |
| AB | 1 | 1 | 1 | – | 1 |
| AC | 2 | 1 | – | 1 | 1 |
| BC | 3 | – | 1 | 1 | – |
| $A'B'C'$ | 4 | 0 | 0 | 0 | – |

Programmable Array Logic

- The PAL is a programmable logic device with a fixed OR array and a programmable AND array.
- Because only the AND gates are programmable, the PAL is easier to program than, but is not as flexible as, the PLA.
- A typical PAL integrated circuit may have eight inputs, eight outputs, and eight sections, each consisting of an eight-wide AND-OR array. The output terminals are sometimes driven by three-state buffers or inverters.

Example

- As an example of using a PAL in the design of a combinational circuit, consider the following Boolean functions, given in sum-of-minterms form:

$$w(A, B, C, D) = \sum(2, 12, 13)$$

$$x(A, B, C, D) = \sum(7, 8, 9, 10, 11, 12, 13, 14, 15)$$

$$y(A, B, C, D) = \sum(0, 2, 3, 4, 5, 6, 7, 8, 10, 11, 15)$$

$$z(A, B, C, D) = \sum(1, 2, 8, 12, 13)$$

- Simplifying the four functions to a minimum number of terms results in the following Boolean functions:

$$w = ABC' + A'B'CD'$$

$$x = A + BCD$$

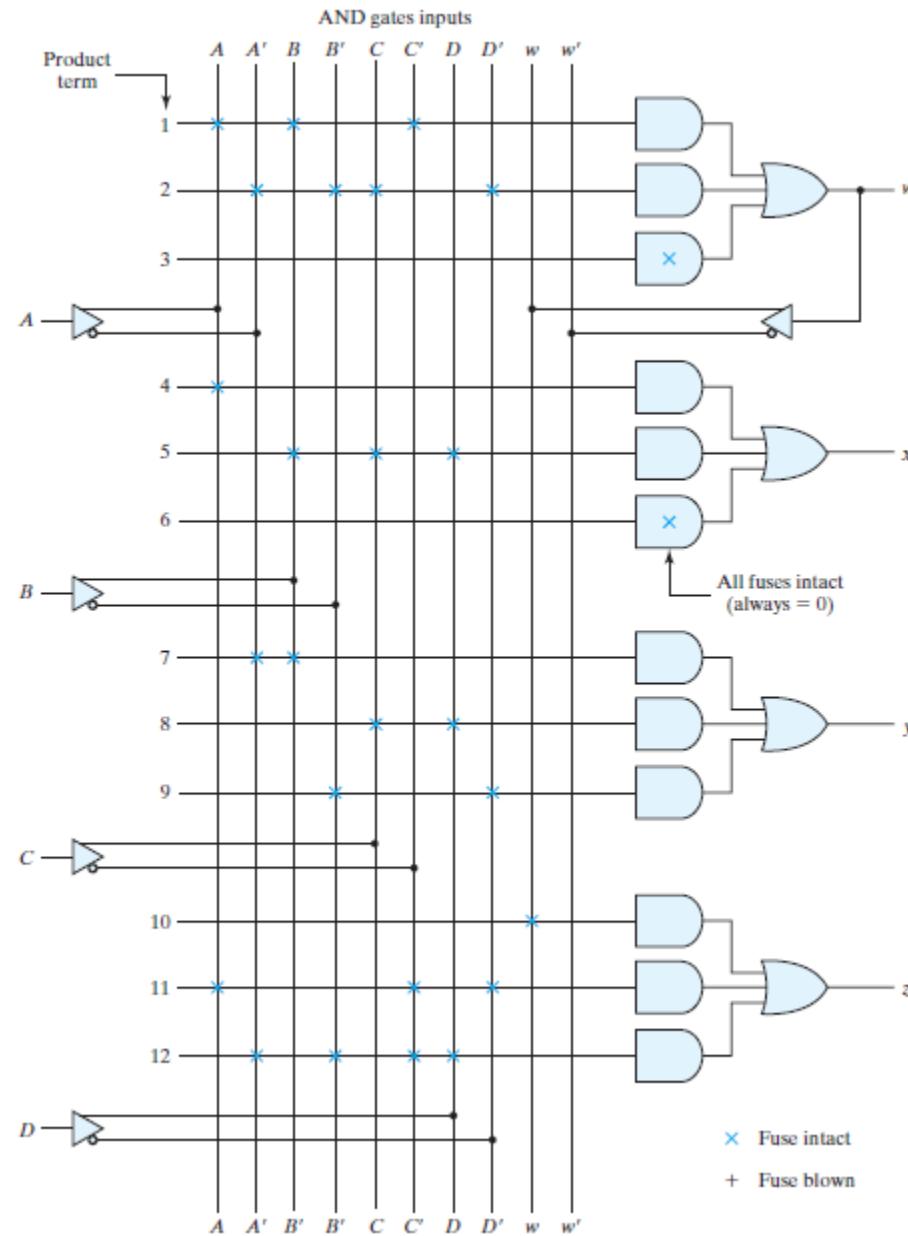
$$y = A'B + CD + B'D'$$

$$\begin{aligned}z &= ABC' + A'B'CD' + AC'D' + A'B'C'D \\&= w + AC'D' + A'B'C'D\end{aligned}$$

- Note that the function for z has four product terms. The logical sum of two of these terms is equal to w . By using w , it is possible to reduce the number of terms for z from four to three.
- The PAL programming table is similar to the one used for the PLA, except that only the inputs of the AND gates need to be programmed. Table lists the PAL programming table for the four Boolean functions.

PAL Programming Table

| Product Term | AND Inputs | | | | | Outputs |
|---------------------|-------------------|----------|----------|----------|----------|---------------------------|
| | A | B | C | D | w | |
| 1 | 1 | 1 | 0 | — | — | $w = ABC' + A'B'CD'$ |
| 2 | 0 | 0 | 1 | 0 | — | |
| 3 | — | — | — | — | — | |
| 4 | 1 | — | — | — | — | $x = A + BCD$ |
| 5 | — | 1 | 1 | 1 | — | |
| 6 | — | — | — | — | — | |
| 7 | 0 | 1 | — | — | — | $y = A'B + CD + B'D'$ |
| 8 | — | — | 1 | 1 | — | |
| 9 | — | 0 | — | 0 | — | |
| 10 | — | — | — | — | 1 | $z = w + AC'D' + A'B'C'D$ |
| 11 | 1 | — | 0 | 0 | — | |
| 12 | 0 | 0 | 0 | 1 | — | |



PAL with four inputs, four outputs, and a three-wide AND-OR structure

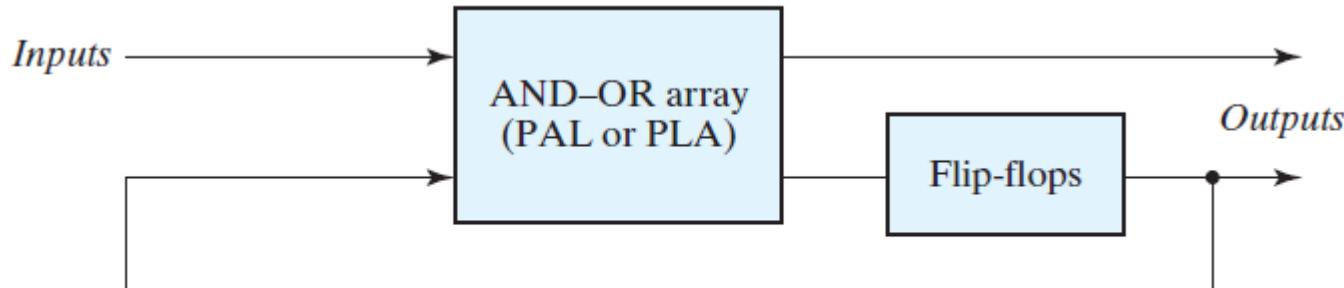
Sequential Programmable Devices

- The combinational PLD consists of only gates, but the sequential programmable devices include both gates and flip-flops.
- In this way, the device can be programmed to perform a variety of sequential-circuit functions.
- There are several types of sequential programmable devices available commercially, and each device has vendor-specific variants within each type.

- The three major types sequential programmable devices are
 1. Sequential (or simple) Programmable Logic Device (SPLD)
 2. Complex Programmable Logic Device (CPLD)
 3. Field-Programmable Gate Array (FPGA)

Sequential Programmable Logic Device (SPLD)

- The sequential PLD is sometimes referred to as a simple PLD to differentiate it from the complex PLD.
- The SPLD includes flip-flops, in addition to the AND-OR array, within the integrated circuit chip.
- The result is a sequential circuit as shown in Figure.
- A PAL or PLA is modified by including a number of flip-flops connected to form a register.



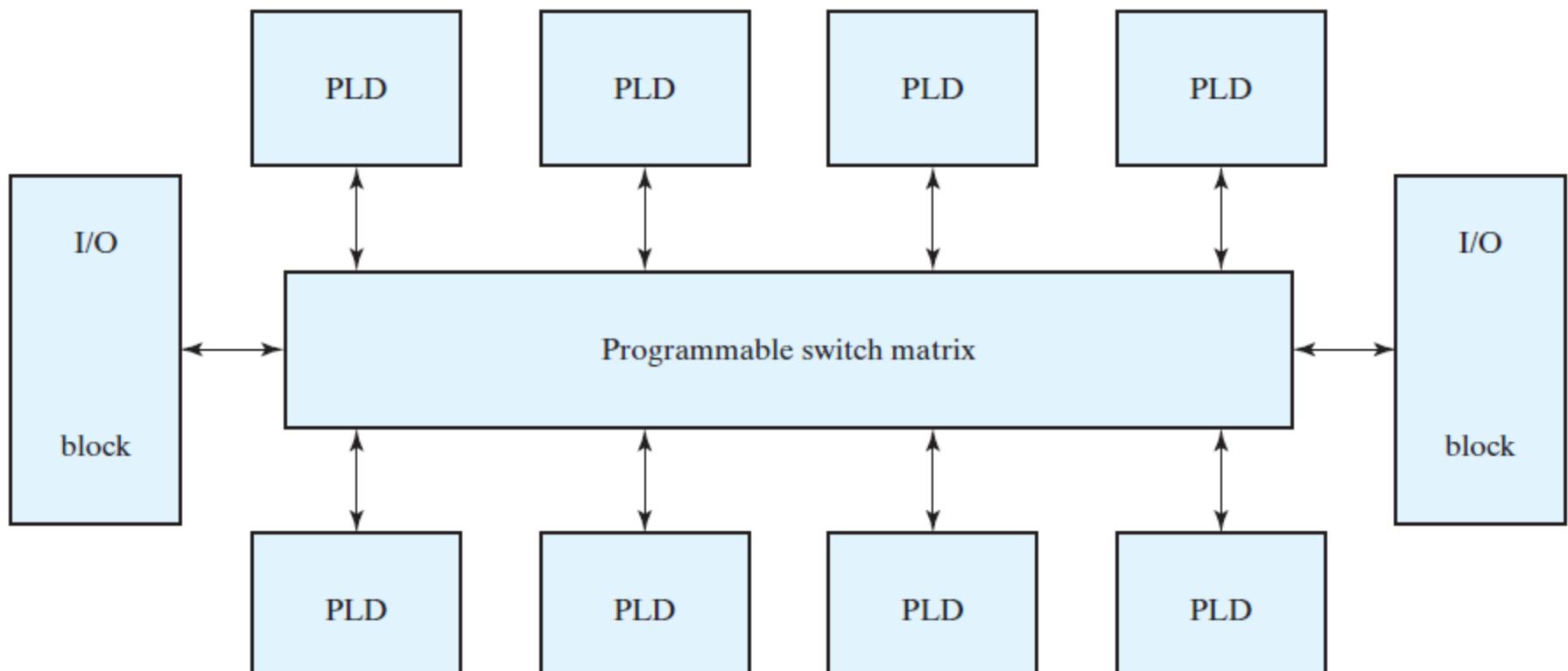
Sequential programmable logic device

- The circuit outputs can be taken from the OR gates or from the outputs of the flip-flops.
- Additional programmable connections are available to include the flip-flop outputs in the product terms formed with the AND array.
- The flip-flops may be of the D or the JK type.

Complex Programmable Logic Device (CPLD)

- The design of a digital system using PLDs often requires the connection of several devices to produce the complete specification.
- For this type of application, it is more economical to use a complex programmable logic device (CPLD), which is a collection of individual PLDs on a single integrated circuit.
- A programmable interconnection structure allows the PLDs to be connected to each other in the same way that can be done with individual PLDs.

- Figure shows the general configuration of a CPLD.
- The device consists of multiple PLDs interconnected through a programmable switch matrix.
- The input-output (I/O) blocks provide the connections to the IC pins. Each I/O pin is driven by a three state buffer and can be programmed to act as input or output.



General CPLD configuration

- The switch matrix receives inputs from the I/O block and directs them to the individual macrocells. Similarly, selected outputs from macrocells are sent to the outputs as needed.
- Each PLD typically contains from 8 to 16 macrocells, usually fully connected. If a macrocell has unused product terms, they can be used by other nearby macrocells.
- In some cases the macrocell flip-flop is programmed to act as a D , JK , or T flip-flop.

Field-Programmable Gate Array (FPGA)

- A field-programmable gate array (FPGA) is a VLSI circuit that can be programmed at the user's location.
- A typical FPGA consists of an array of millions of logic blocks, surrounded by programmable input and output blocks and connected together via programmable interconnections.
- There is a wide variety of internal configurations within this group of devices.

- The performance of each type of device depends on the circuit contained in its logic blocks and the efficiency of its programmed interconnections.
- A typical FPGA logic block consists of lookup tables, multiplexers, gates, and flip-flops.
- A lookup table is a truth table stored in an SRAM and provides the combinational circuit functions for the logic block.

- The design with PLD, CPLD, or FPGA requires extensive computer-aided design (CAD) tools to facilitate the synthesis procedure.
- Among the tools that are available are schematic entry packages and hardware description languages (HDLs), such as ABEL, VHDL, and Verilog. Synthesis tools are available that allocate, configure, and connect logic blocks to match a high-level design description written in HDL.

DIGITAL ELECTRONICS

III Semester CSE & CST



Mr. Srikanta Patnaik

Mr. Ajit Kumar Patro

Mr. Ami Kumar Parida

DEPARTMENT OF ELECTRONICS AND COMMUNICATION ENGINEERING

SCHOOL OF ENGINEERING & TECHNOLOGY

GIET UNIVERSITY, GUNUPUR, ODISHA

ANALOG-TO-DIGITAL AND DIGITAL-TO-ANALOG CONVERTERS

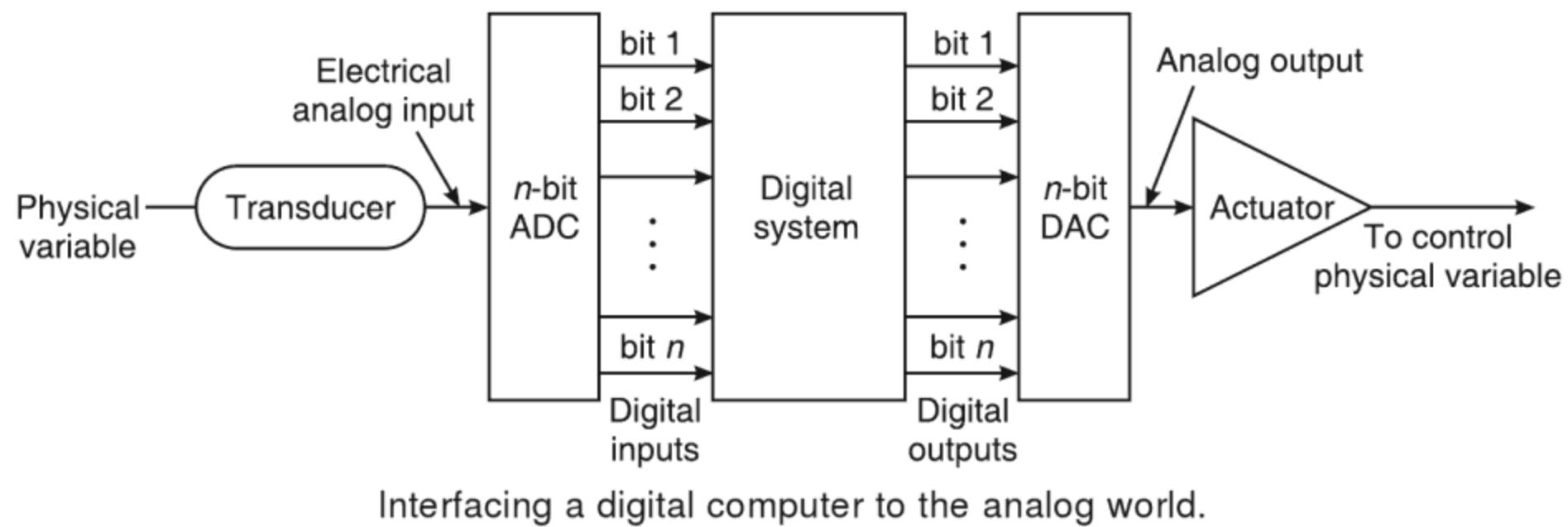
INTRODUCTION

- An analog quantity is one that can take on any value over a continuous range of values. It represents an exact value.
- Most physical variables are analog in nature.
- Example: Temperature, pressure, light and sound intensity, position, rotation, speed, etc.
- A digital quantity takes on only discrete values. The value is expressed in digital code such as a binary or BCD number.

- When a physical process is monitored or controlled by a digital system such as a digital computer, the physical variables are first converted into electrical signals using transducers, and then these electrical analog signals are converted into digital signals using analog-to-digital converters (ADCs).

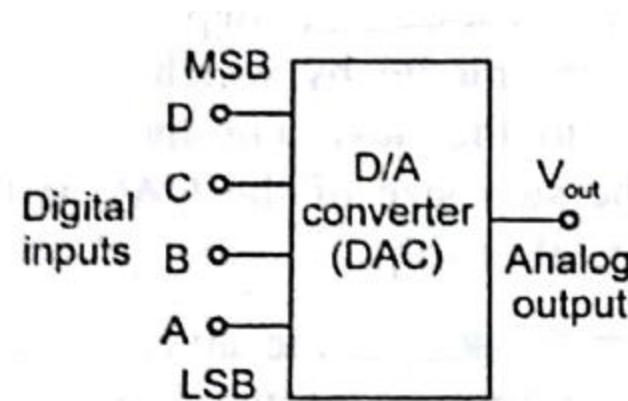
- These digital signals are processed by a digital computer and the output of the digital computer is converted into analog signals using digital-to-analog-converters (DACs).
- The output of the DAC is modified by an actuator and the output of the actuator is applied as the control variable.

- Figure shows how ADCs and DACs function as interfaces between a completely digital system such as digital computer and the analog world.



DIGITAL-TO-ANALOG (D/A) CONVERSION

- Basically, D/A conversion is the process of converting a value represented in digital code, such as straight binary or BCD, into a voltage or current which is proportional to the digital value.
- Figure shows the symbol for a typical 4-bit D/A converter.

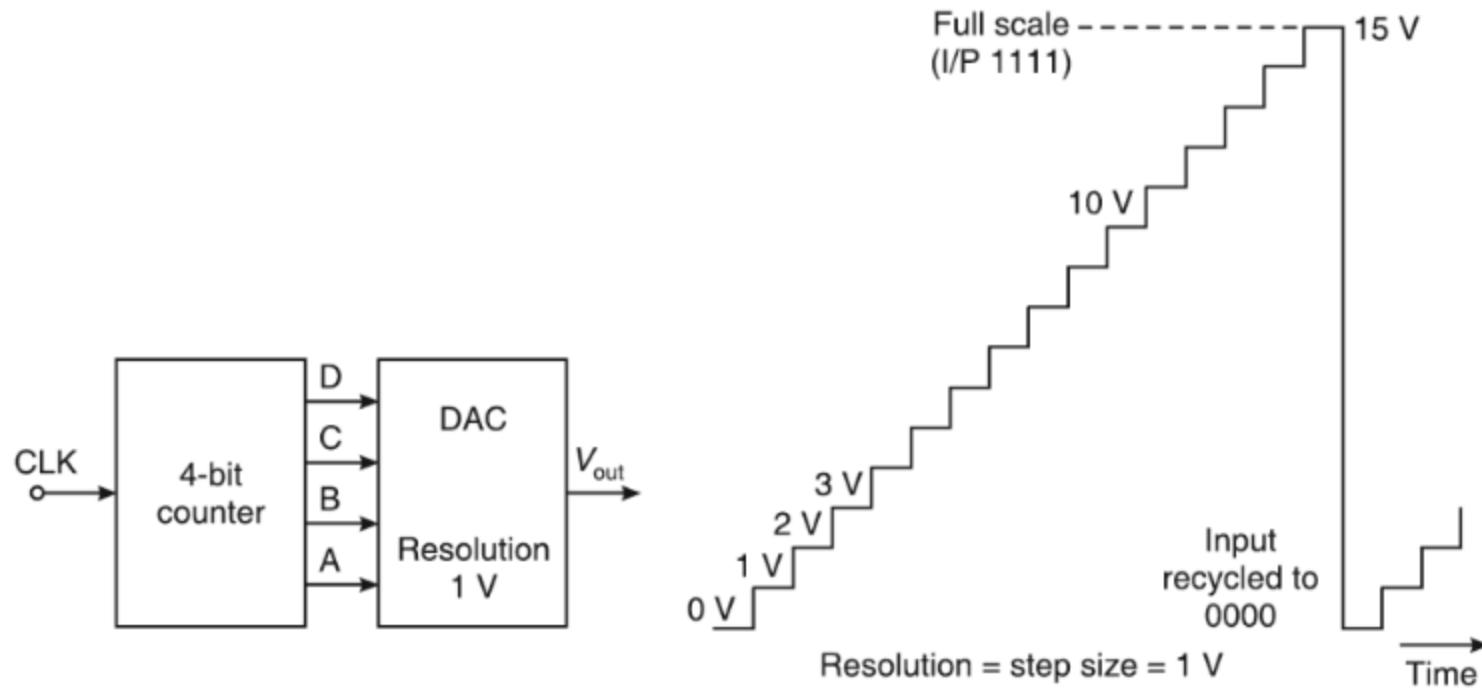


Block diagram of a 4-bit DAC.

- The analog output voltage V_{out} is proportional to the input binary number is,

$$\text{Analog output} = K \times \text{digital input}$$

Where K is the proportionality factor and is a constant value for a given DAC.



Output waveform of a DAC fed by a binary counter.

Parameters of DAC

- Resolution (step size)
- Speed
- Linearity
- Settling Time
- Reference Voltages
- Errors

Resolution (step size)

- The resolution of a DAC is defined as the smallest change that can occur in an analog output as a result of a change in the digital input.
- The resolution of a DAC is also defined as the reciprocal of the number of discrete steps in the full-scale output of the DAC.
- The resolution is always equal to the weight of the LSB and is also referred to as the *step size*.

- Although resolution can be expressed as the amount of voltage or current per step, it is also useful to express it as percentage of the full-scale output as

$$\% \text{ resolution} = \frac{\text{step size}}{\text{full scale}} \times 100\%$$

- Since, full scale = number of steps × step size, resolution can be expressed as

$$\% \text{ resolution} = \frac{1}{\text{total number of steps}} \times 100\%$$

- In general, for an N -bit DAC, the number of different levels will be 2^N and the number of steps will be $2^N - 1$.

EXAMPLE Determine the resolution of (a) a 6-bit DAC and that of (b) a 12-bit DAC in terms of percentage.

Solution

(a) For the 6-bit DAC,

$$\% \text{ resolution} = \frac{1}{2^N - 1} \times 100 = \frac{1}{2^6 - 1} \times 100 = \frac{1}{63} \times 100 = 1.587\%$$

(b) For the 12-bit DAC,

$$\% \text{ resolution} = \frac{1}{2^N - 1} \times 100 = \frac{1}{2^{12} - 1} \times 100 = \frac{1}{4095} \times 100 = 0.0244\%$$

EXAMPLE A 6-bit DAC has a step size of 50 mV. Determine the full-scale output voltage and the percentage resolution.

Solution

With 6 bits, there will be $2^6 - 1 = 63$ steps of size 50 mV each.

The full-scale output will, therefore, be

$$63 \times 50 \text{ mV} = 3.15 \text{ V}$$

$$\% \text{ resolution} = \frac{50 \text{ mV}}{3.15 \text{ V}} \times 100 = \frac{1}{63} \times 100 = 1.587\%$$

EXAMPLE An 8-bit DAC produces $V_{\text{out}} = 0.05 \text{ V}$ for a digital input of 00000001. Find the full-scale output. What is the resolution? What is V_{out} for an input of 00101010?

Solution

$$\begin{aligned}\text{Full-scale output} &= \text{Step size} \times \text{Number of steps} \\ &= 0.05(2^8 - 1) = 0.05 \times 255 = 12.75 \text{ V}\end{aligned}$$

$$\% \text{ resolution} = \frac{1}{2^N - 1} \times 100 = \frac{1}{255} \times 100 = 0.392\%$$

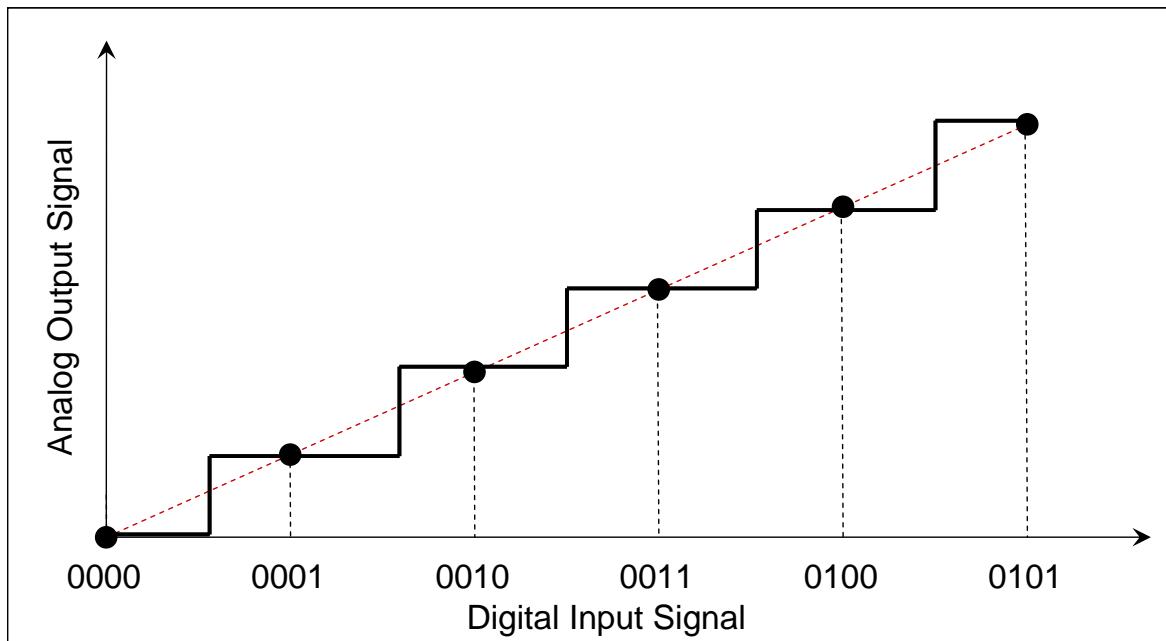
$$V_{\text{out}} \text{ for an input of } 00101010 = 42 \times 0.05 = 2.10 \text{ V}$$

Speed

- Rate of conversion of a single digital input to its analog equivalent.
- Conversion rate depends on
 - clock speed of input signal
 - settling time of converter
- When the input changes rapidly, the DAC conversion speed must be high.

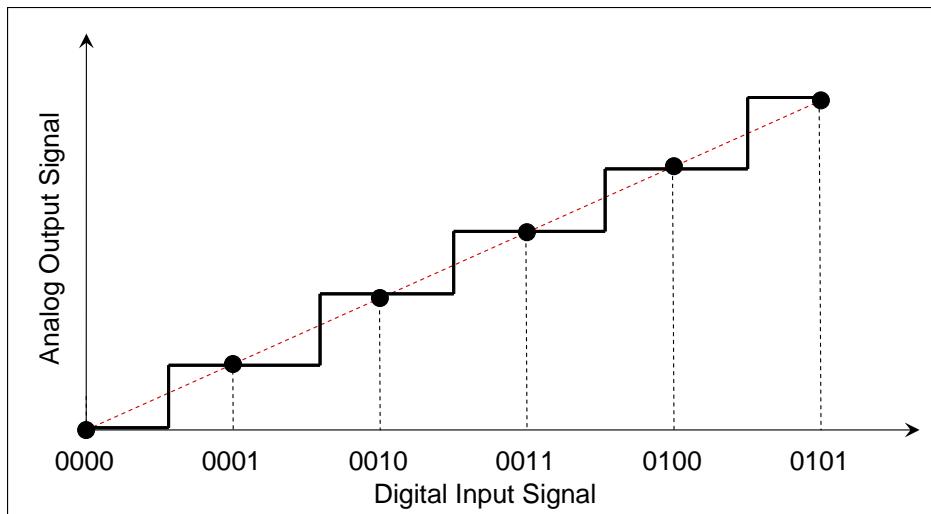
Linearity

- The difference between the desired analog output and the actual output over the full range of expected values.

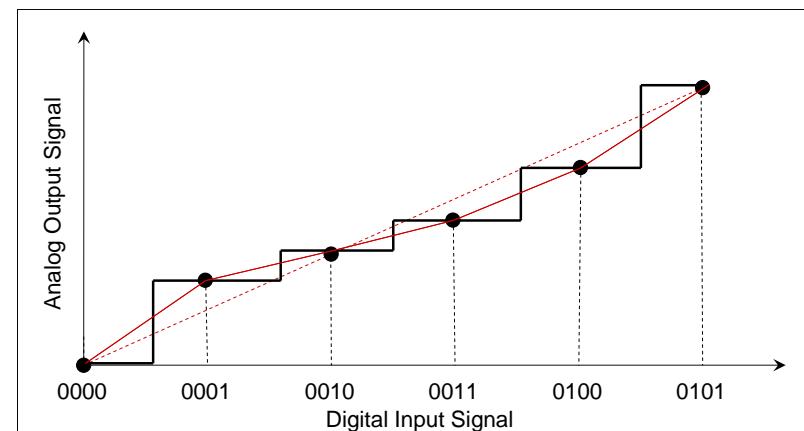


- Ideally, a DAC should produce a linear relationship between the digital input and analog output.

Linearity (Ideal)

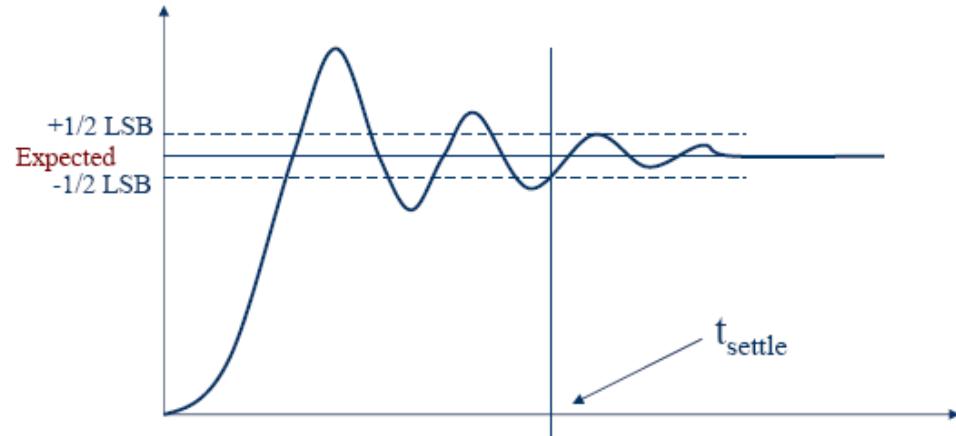


Non-Linearity



Settling Time

- Time required for the output signal to settle within $\pm \frac{1}{2}$ LSB of its final value after a given change in input scale.
- Limited by slew rate of output amplifier.
- Ideally, an instantaneous change in analog voltage would occur when a new binary word enters into DAC.



Reference Voltages

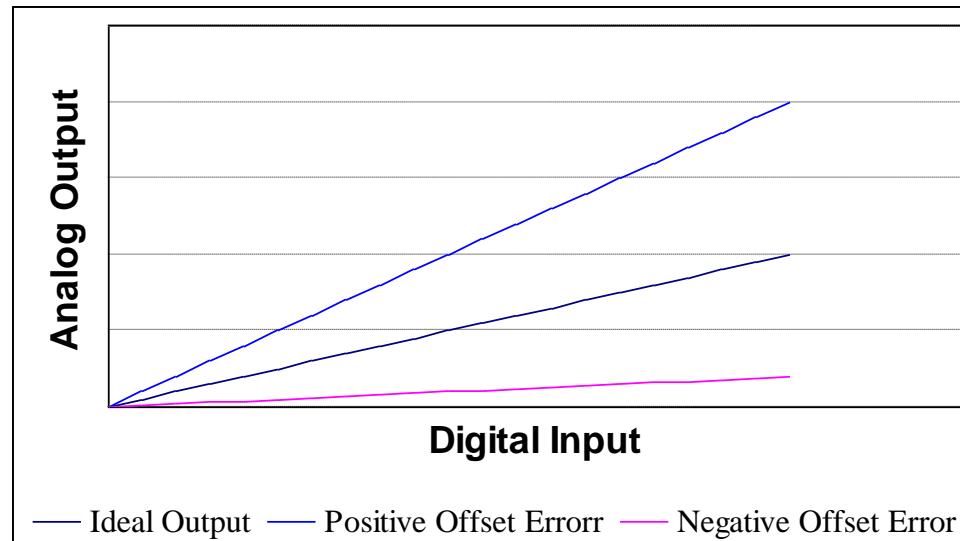
- Used to determine how each digital input will be assigned to each voltage division
- Types:
 - Non-multiplier DAC: V_{ref} is fixed
 - Multiplier DAC: V_{ref} provided by external source

Types of Errors Associated with DACs

- Gain
- Offset
- Full Scale
- Resolution
- Non-Linearity
- Non-Monotonic
- Settling Time and Overshoot

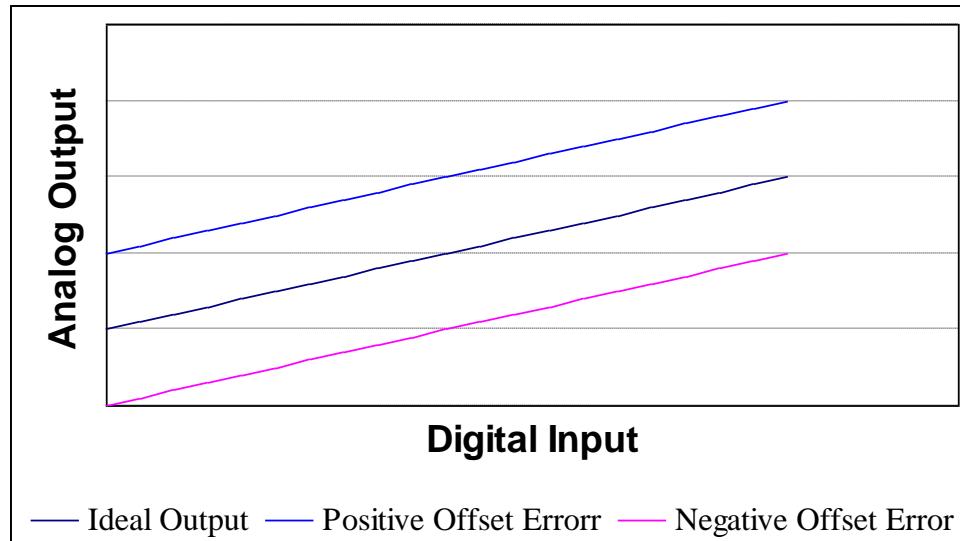
Gain Error

- Occurs when the slope of the actual output deviates from the ideal output.



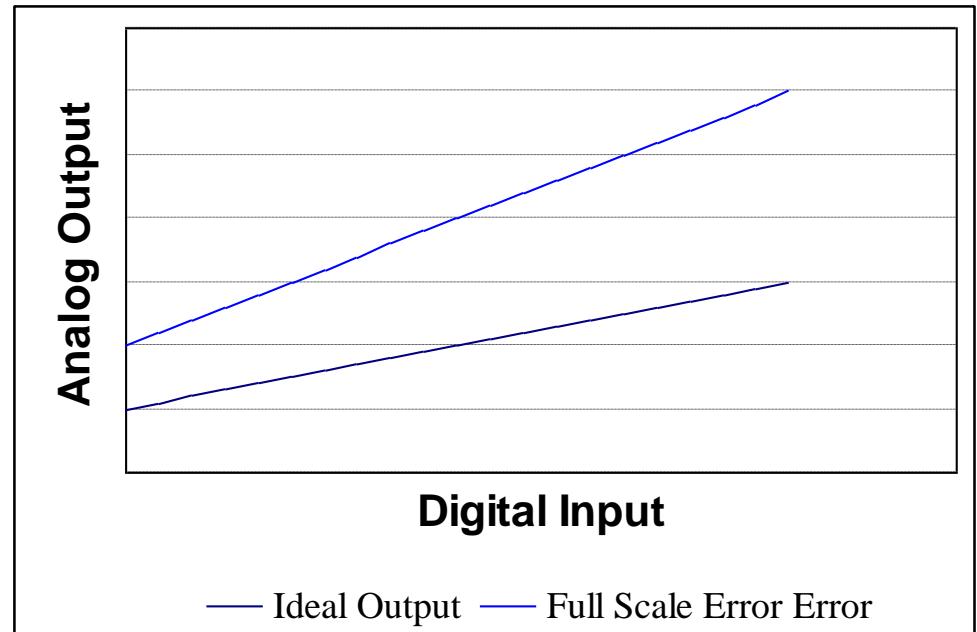
Offset Error

- Occurs when there is a constant offset between the actual output and the ideal output.



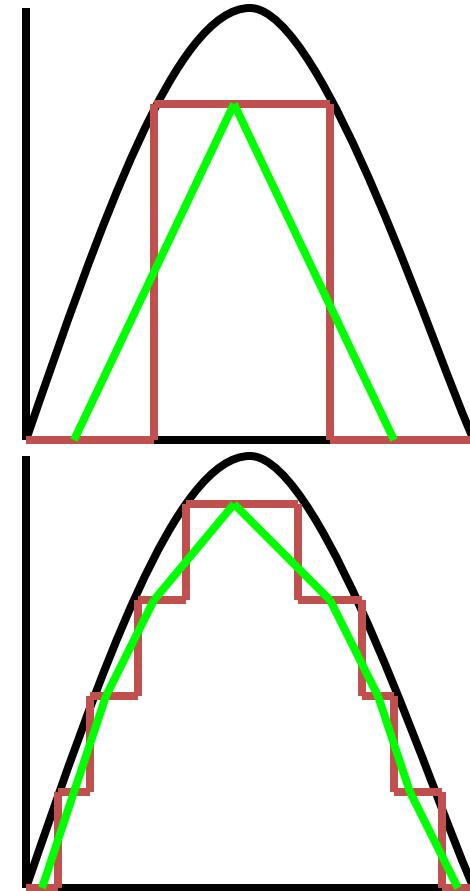
Full Scale Error

- Occurs when the actual signal has both gain and offset errors.



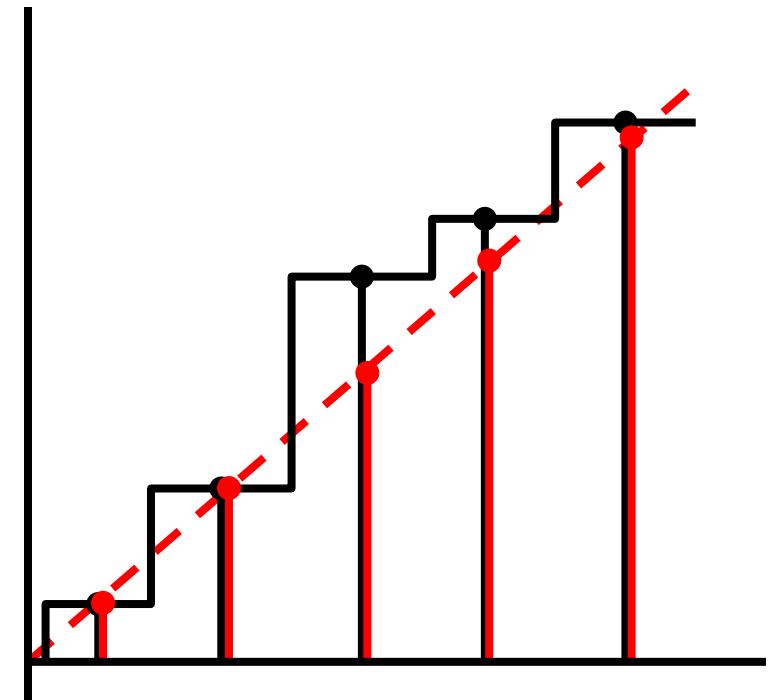
Resolution Error

- Poor representation of ideal output due to poor resolution.
- Size of voltage divisions affect the resolution.



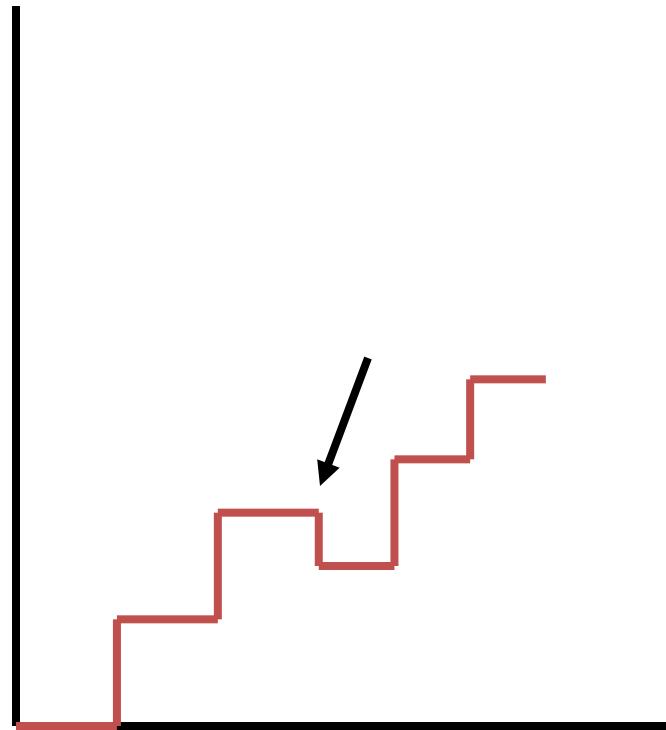
Non-Linearity Error

- Occurs when analog output of signal is non-linear.
- Two Types
 - Differential – analog step-sizes changes with increasing digital input (measure of largest deviation; between successive bits).
 - Integral – amount of deviation from a straight line after offset and gain errors removed; on concurrent bits.



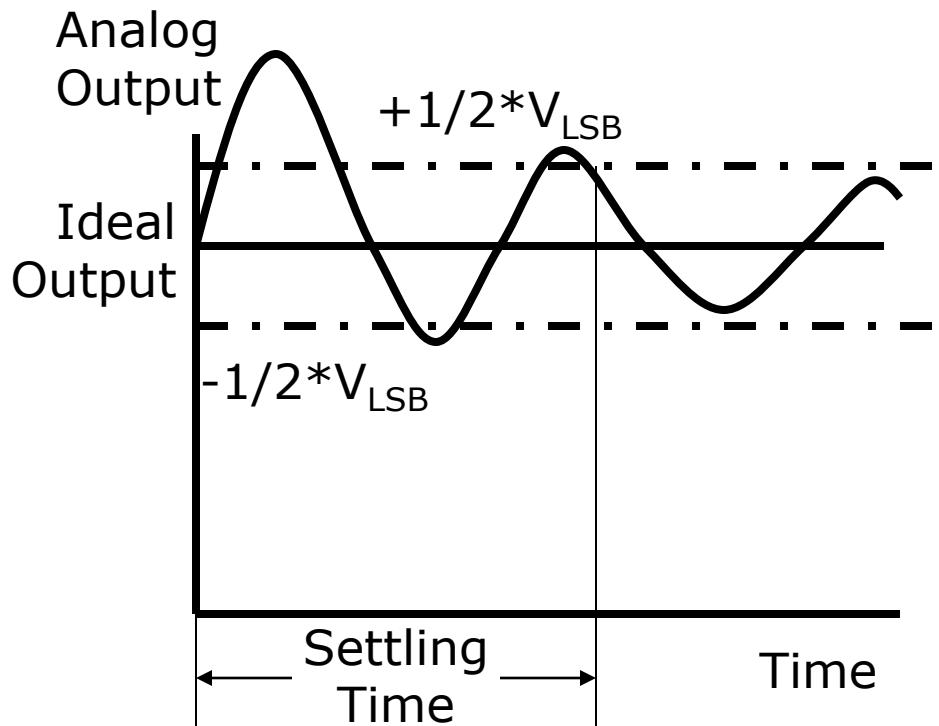
Non-Monotonic Error

- Occurs when an increase in digital input results in a decrease in the analog output.



Settling Time and Overshoot Error

- Settling Time – time required for the output to fall within $\pm \frac{1}{2} V_{LSB}$.
- Overshoot – occurs when analog output overshoots the ideal output.



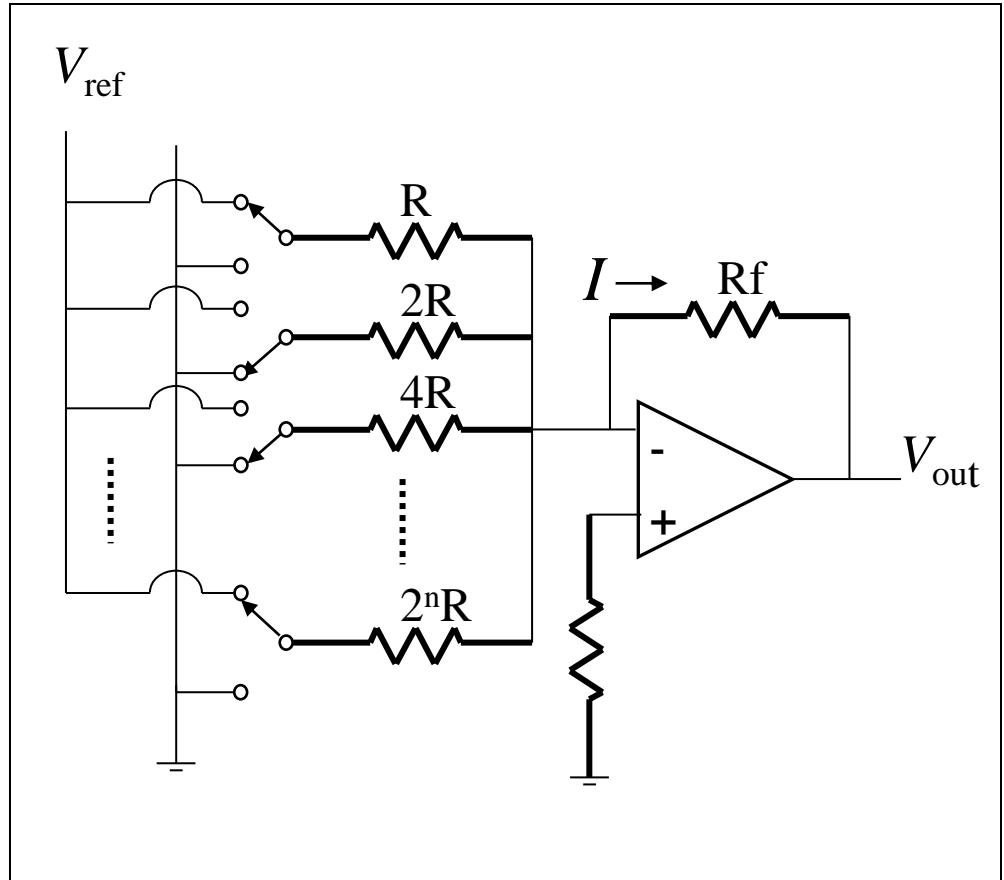
Types of DACs

- Many types of DACs available.
- Usually switches, resistors, and op-amps used to implement conversion.
- Two Types:
 - Binary Weighted Resistor Type DAC
 - R-2R Ladder Type DAC

Binary Weighted Resistor Type DAC

- Utilizes a summing op-amp circuit.
- Weighted resistors are used to distinguish each bit from the most significant to the least significant.
- Transistors are used to switch between V_{ref} and ground (bit high or low).

- Assume Ideal Op-amp
- No current into Op-amp
- Virtual ground at inverting input
- $V_{\text{out}} = -IR_f$



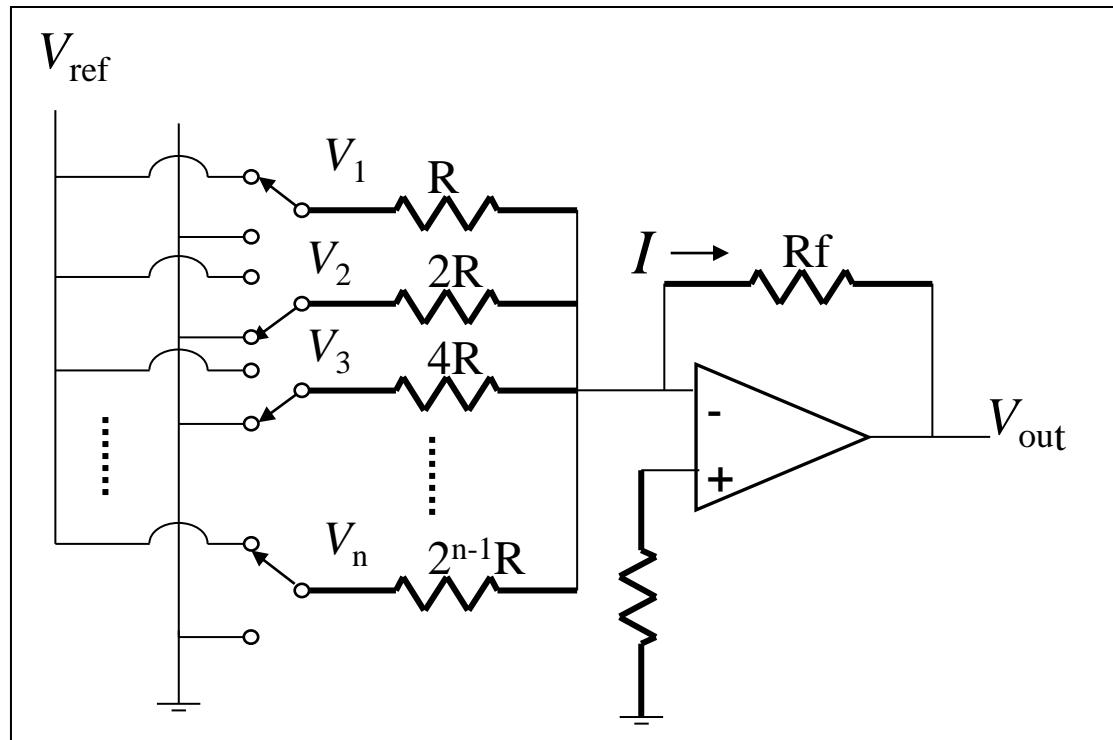
Voltages V_1 through V_n are either V_{ref} if corresponding bit is high or ground if corresponding bit is low

V_1 is most significant bit

V_n is least significant bit
MSB

$$V_{\text{out}} = -IR_f = -R_f \left(\frac{V_1}{R} + \frac{V_2}{2R} + \frac{V_3}{4R} + \dots + \frac{V_n}{2^{n-1}R} \right)$$

LSB



If $R_f = R/2$

$$V_{\text{out}} = -IR_f = -\left(\frac{V_1}{2} + \frac{V_2}{4} + \frac{V_3}{8} + \dots + \frac{V_n}{2^n}\right)$$

For example, a 4-Bit converter yields

$$V_{\text{out}} = -V_{\text{ref}} \left(b_3 \frac{1}{2} + b_2 \frac{1}{4} + b_1 \frac{1}{8} + b_0 \frac{1}{16} \right)$$

Where b_3 corresponds to Bit-3, b_2 to Bit-2, etc.

- Advantages
 - Simple Construction/Analysis
 - Fast Conversion
- Disadvantages
 - Requires large range of resistors (2000:1 for 12-bit DAC) with necessary high precision for low resistors
 - Requires low switch resistances in transistors
 - Can be expensive. Therefore, usually limited to 8-bit resolution.

EXAMPLE For the 4-bit weighted-resistor DAC shown in Figure, determine
 (a) the weight of each input bit if the inputs are 0 V and 5 V, (b) the full-scale output, if $R_f = R = 1 \text{ k}\Omega$. Also, find the full-scale output if R_f is changed to 500 Ω .

Solution

- (a) The MSB passes with a gain of 1, so, its weight = 5 V; the next bit passes with a gain of 1/2, so, its weight = $5/2 = 2.5$ V; the following bit passes with a gain of 1/4, so, its weight = $5/4 = 1.25$ V; the LSB passes with a gain of 1/8, so, its weight = $5/8 = 0.625$ V.
 (b) Therefore, the full scale output

$$\begin{aligned} (\text{when } R_f = R = 1 \text{ k}\Omega) &= -\left(1 + \frac{1}{2} + \frac{1}{4} + \frac{1}{8}\right) \times 5 \\ &= -9.375 \text{ V} \end{aligned}$$

The full-scale output, when R_f is changed to 500 Ω is

$$V_{\text{out}} = -\left(1 + \frac{1}{2} + \frac{1}{4} + \frac{1}{8}\right) \times \left(\frac{5}{2}\right) = -4.6875 \text{ V}$$

EXAMPLE Design a 4-bit weighted-resistor DAC whose full-scale output voltage is -5 V. The logic levels are $1 = +5$ V and $0 = 0$ V. What is the output voltage when the input is 1101?

Solution

The full-scale output voltage is the output voltage when the input voltage is maximum, i.e. 1111. Thus,

$$\text{The full-scale output} = -\left(5 \text{ V} + \frac{5 \text{ V}}{2} + \frac{5 \text{ V}}{4} + \frac{5 \text{ V}}{8}\right) \times \frac{R_f}{R} = -5 \text{ V}$$

Therefore,

$$9.375 \times \frac{R_f}{R} = 5$$

Choosing $R_f = 10 \text{ k}\Omega$, and since $\frac{R_f}{R} = \frac{5}{9.375}$, we have

$$R = R_f \times \frac{9.375}{5} = 18.75 \text{ k}\Omega$$

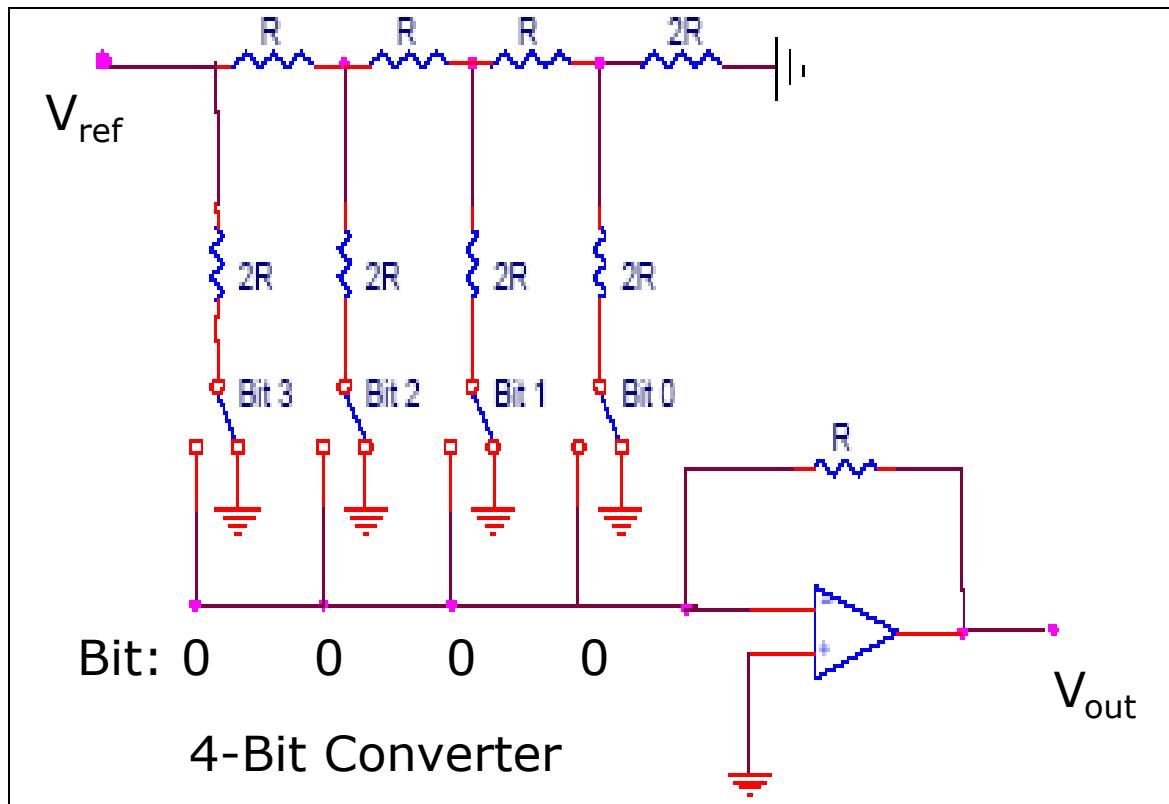
Thus,

$$2R = 37.5 \text{ k}\Omega; \quad 4R = 75 \text{ k}\Omega; \quad 8R = 150 \text{ k}\Omega$$

When the input is 1101, the output voltage is

$$V_{\text{out}} = -\left(5 \text{ V} + \frac{5 \text{ V}}{2} + 0 + \frac{5 \text{ V}}{8}\right) \times \frac{R_f}{R} = -8.125 \times \frac{10}{18.75} = -4.333 \text{ V}$$

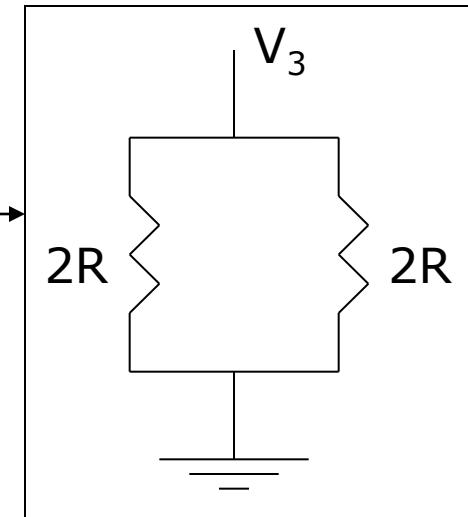
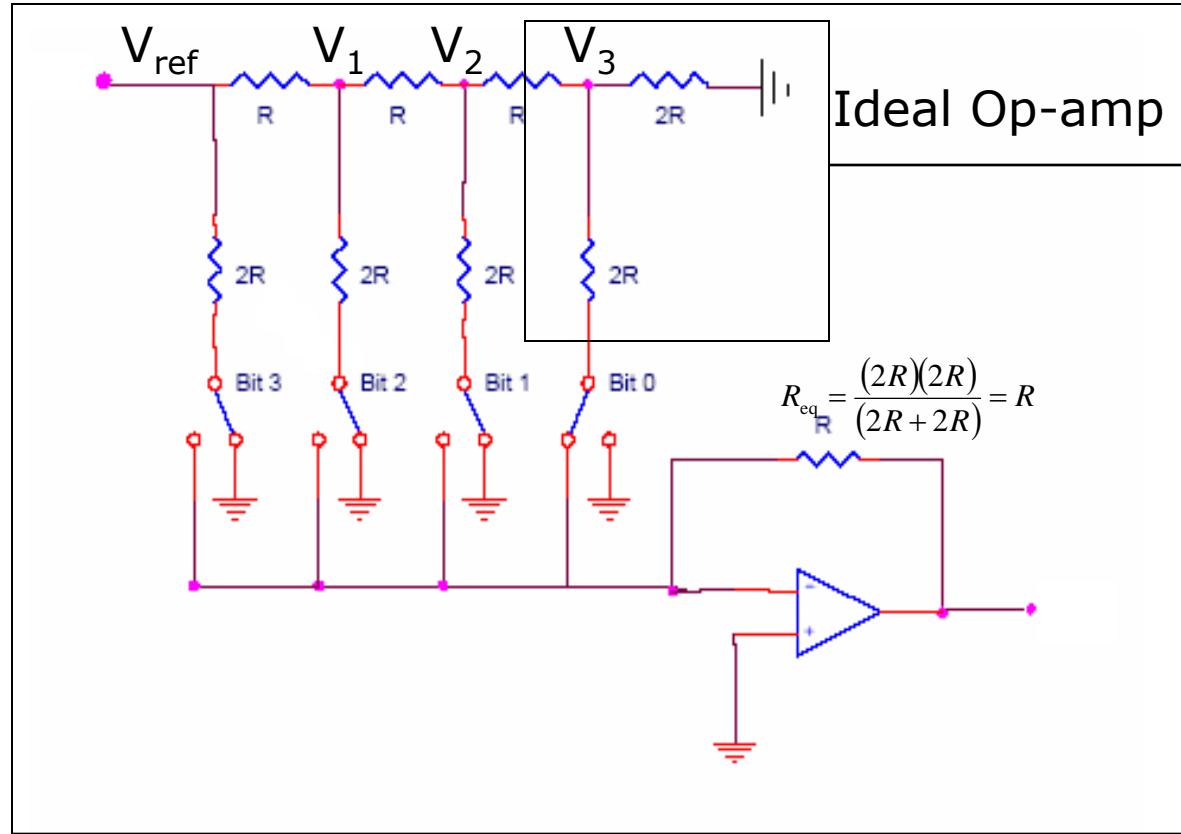
R-2R Ladder Type DAC

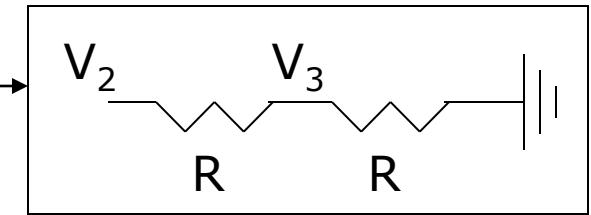
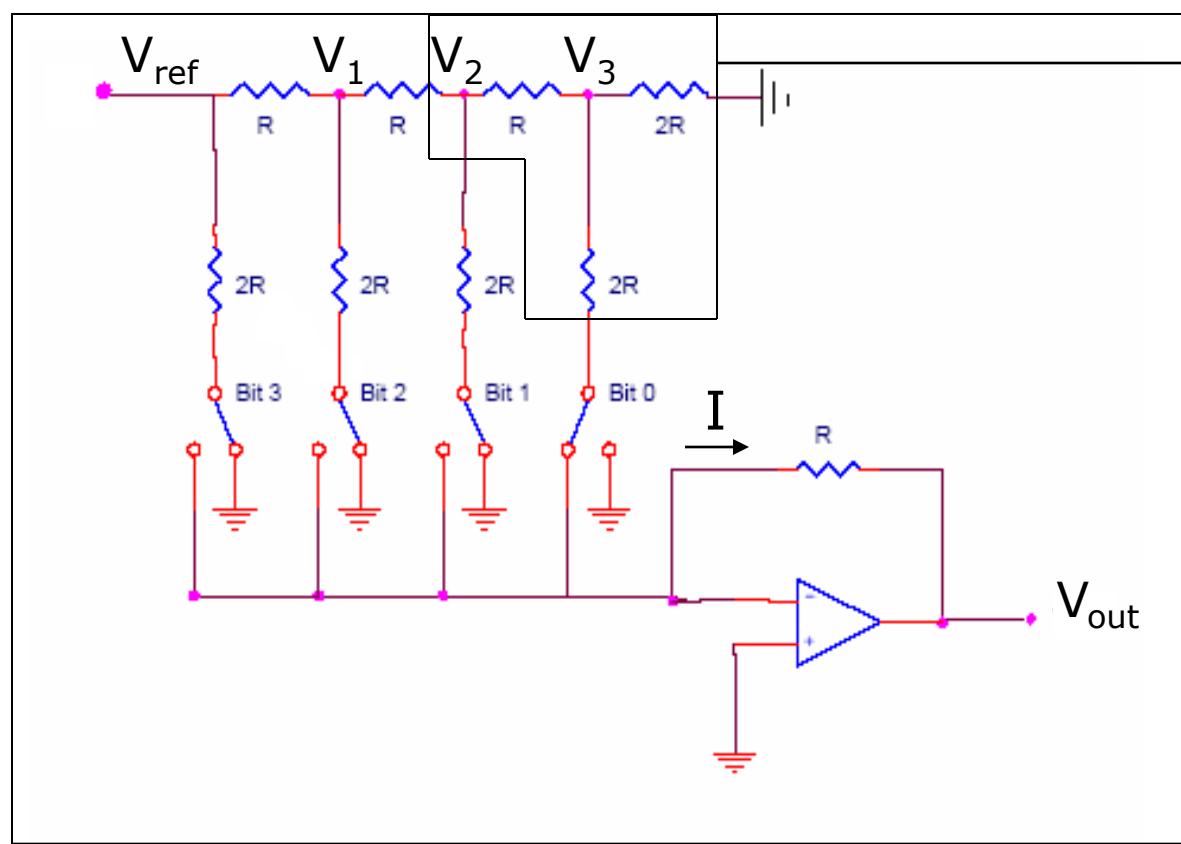


Each bit corresponds to a switch:

If the bit is high, the corresponding switch is connected to the inverting input of the op-amp.

If the bit is low, the corresponding switch is connected to ground.





$$V_3 = \left(\frac{R}{R+R} \right) V_2 = \frac{1}{2} V_2$$

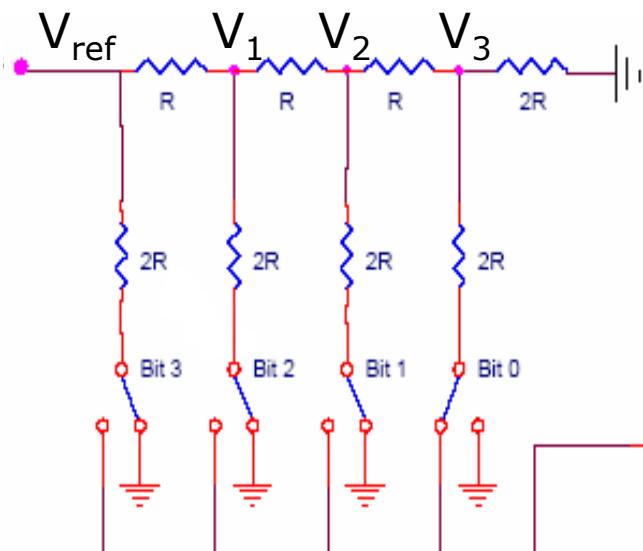
Likewise,

$$V_2 = \frac{1}{2} V_1$$

$$V_1 = \frac{1}{2} V_{\text{ref}}$$

$$V_{\text{out}} = -IR$$

Results:



$$V_3 = \frac{1}{8}V_{\text{ref}}, V_2 = \frac{1}{4}V_{\text{ref}}, V_1 = \frac{1}{2}V_{\text{ref}}$$

$$V_{\text{out}} = -R \left(b_3 \frac{V_{\text{ref}}}{2R} + b_2 \frac{V_{\text{ref}}}{4R} + b_1 \frac{V_{\text{ref}}}{8R} + b_0 \frac{V_{\text{ref}}}{16R} \right)$$

Where b_3 corresponds to bit 3,
 b_2 to bit 2, etc.

If bit n is set, $b_n=1$

If bit n is clear, $b_n=0$

For a 4-Bit R-2R Ladder

$$V_{\text{out}} = -V_{\text{ref}} \left(b_3 \frac{1}{2} + b_2 \frac{1}{4} + b_1 \frac{1}{8} + b_0 \frac{1}{16} \right)$$

For general n-Bit R-2R Ladder or Binary Weighted Resister DAC

$$V_{\text{out}} = -V_{\text{ref}} \sum_{i=1}^n b_{n-i} \frac{1}{2^i}$$

- Advantages
 - Only two resistor values (R and $2R$)
 - Does not require high precision resistors
- Disadvantage
 - Lower conversion speed than binary weighted DAC

EXAMPLE What are the output voltages caused by logic 1 in each bit position in an 8-bit ladder if the input level for 0 is 0 V and that for 1 is + 10 V?

Solution

$$\text{The output voltage level caused by the } D_n \text{ bit} = \frac{E}{2^{N-n}}$$

$$\text{The output voltage level caused by the } D_7 \text{ bit (MSB)} = \frac{E}{2^{8-7}} = \frac{E}{2^1} = \frac{10}{2} = 5 \text{ V}$$

$$\text{The output voltage level caused by the } D_6 \text{ bit} = \frac{E}{2^2} = \frac{10}{4} = 2.5 \text{ V}$$

$$\text{The output voltage level caused by the } D_5 \text{ bit} = \frac{E}{2^3} = \frac{10}{8} = 1.25 \text{ V}$$

$$\text{The output voltage level caused by the } D_4 \text{ bit} = \frac{E}{2^4} = \frac{10}{16} = 0.625 \text{ V}$$

⋮

$$\text{The output voltage level caused by the } D_0 \text{ bit (LSB)} = \frac{E}{2^8} = \frac{10}{256} = 0.03906 \text{ V}$$

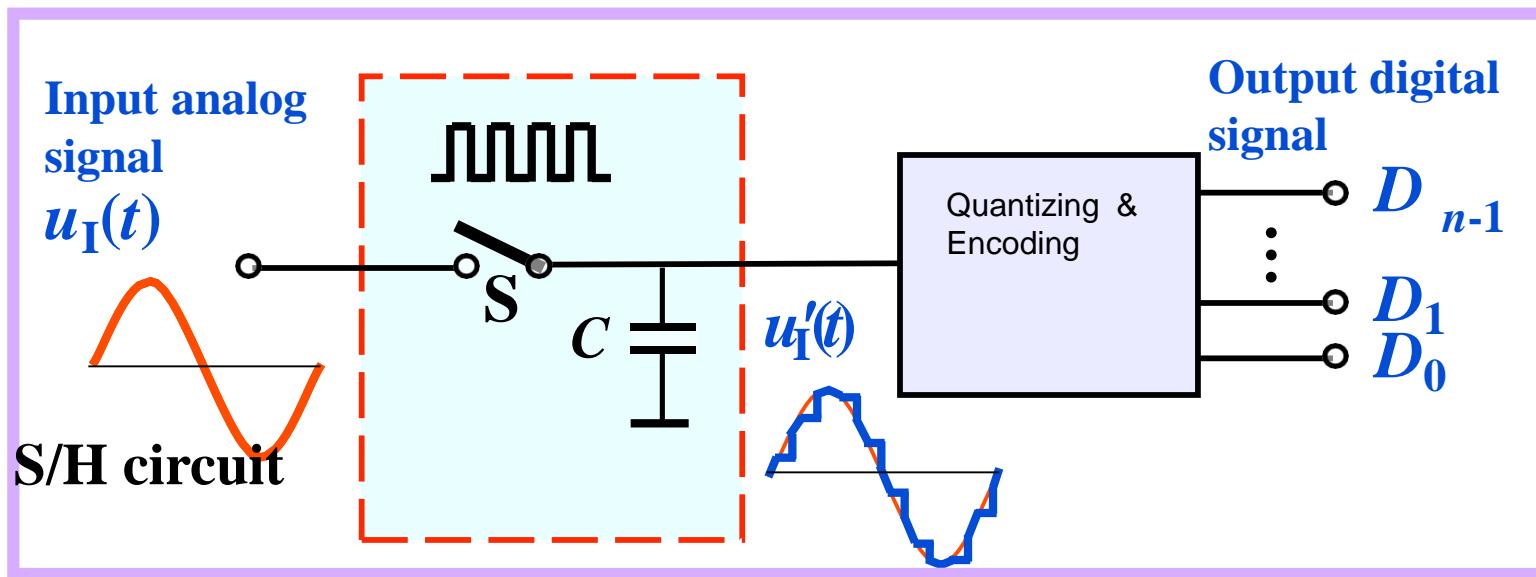
Applications of DAC

- Digital Motor Control
- Computer Printers
- Sound Equipment (e.g. CD/MP3 Players, etc.)
- Electronic Cruise Control
- Digital Thermostat

ANALOG-TO-DIGITAL (A/D) CONVERSION

- An electronic integrated circuit which transforms a signal from analog (continuous) to digital (discrete) form.
- Analog signals are directly measurable quantities.
- Digital Signals – have only two states. For digital computers, we refer to binary states, 0 and 1. “1” can be on, “0” can be off.

ADC process

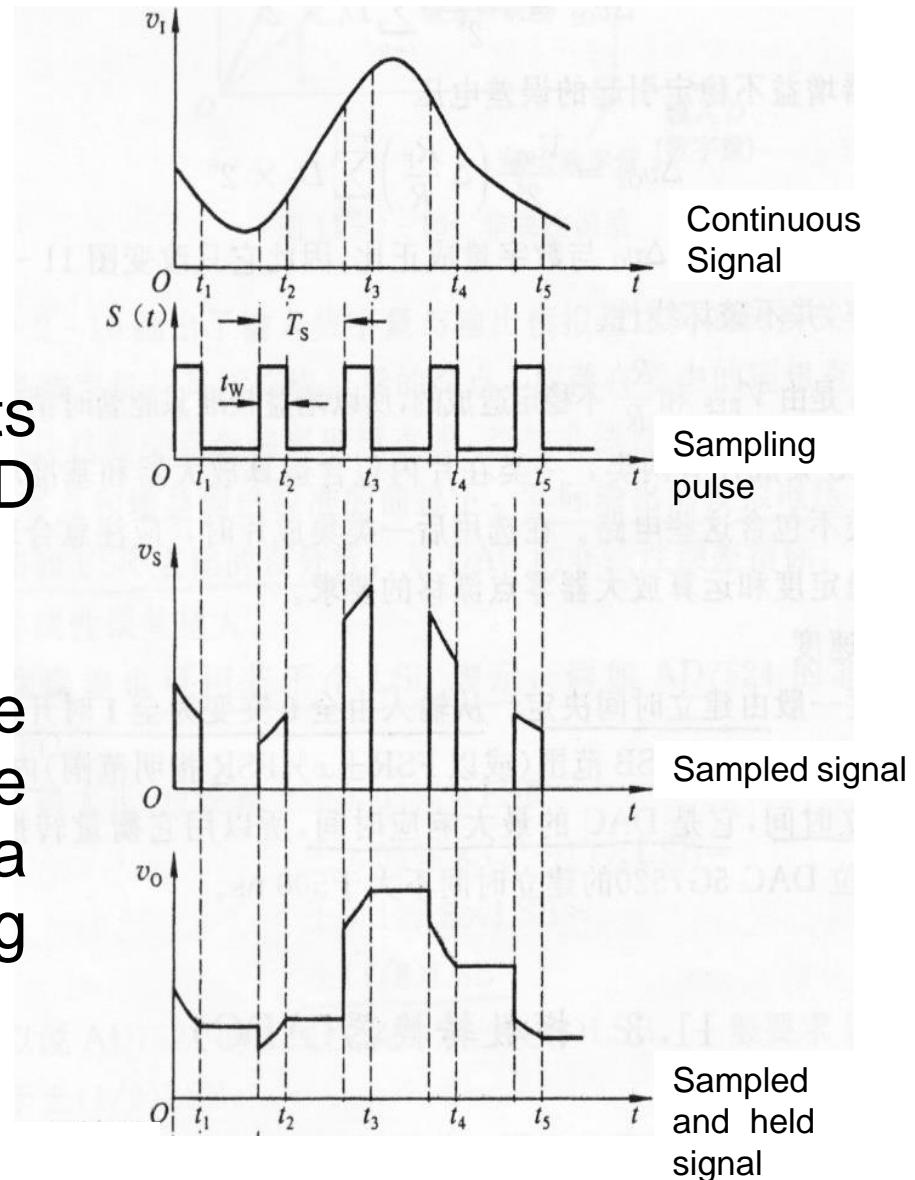


2 steps

- Sampling and Holding (S/H)
- Quantizing and Encoding (Q/E)

Sampling and Holding

- Holding signal benefits the accuracy of the A/D conversion
- Minimum sampling rate should be at least twice the highest data frequency of the analog signal



Quantizing and Encoding

Resolution:

The smallest change in analog signal that will result in a change in the digital output.

$$\Delta V = \frac{V_r}{2^N}$$

V_r = Reference voltage range

N = Number of bits in digital output. 2^N = Number of states.

ΔV = Resolution

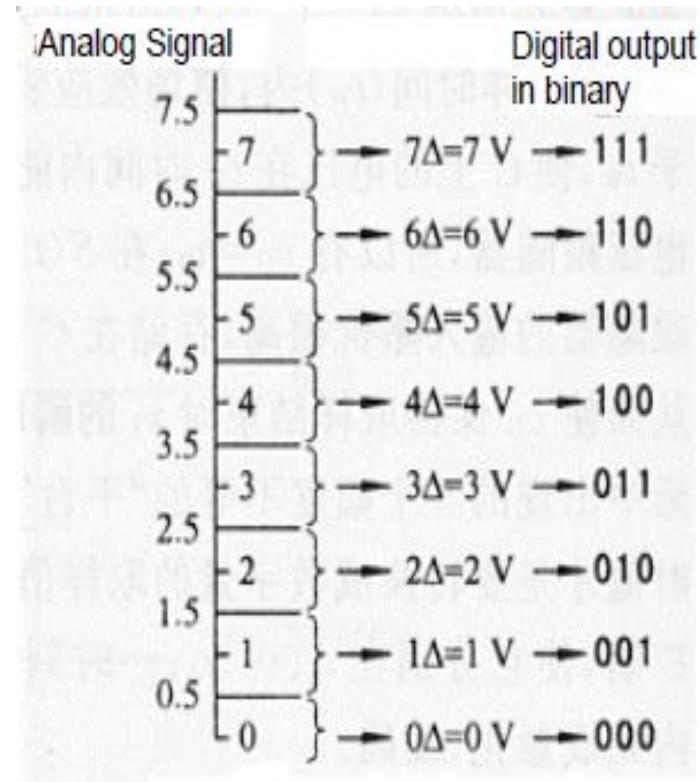
The resolution represents the quantization error inherent in the conversion of the signal to digital form

- **Quantizing:**

Partitioning the reference signal range into a number of discrete quanta, then matching the input signal to the correct quantum.

- **Encoding:**

Assigning a unique digital code to each quantum, then allocating the digital code to the input signal.



$$\Delta V = 1 V$$

$$\text{Maximum Quantization error} = \pm \frac{1}{2} \Delta V = \pm 0.5 V$$

The number of possible states that the converter can output is:

$$N=2^n$$

where n is the number of bits in the AD converter

Example: For a 3 bit A/D converter, $N=2^3=8$.

Analog quantization size:

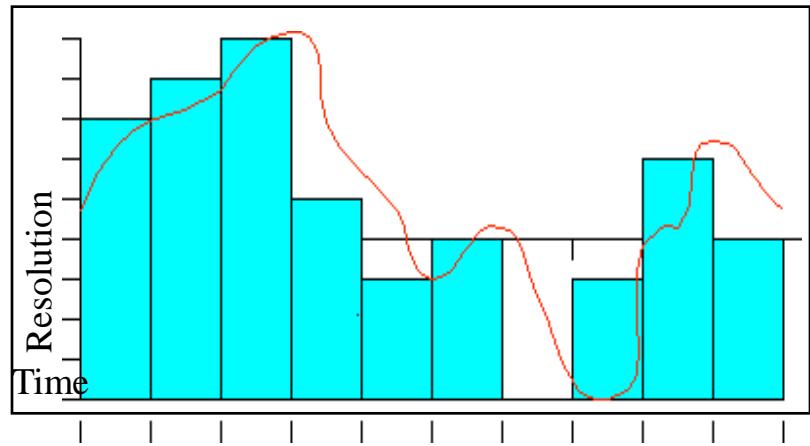
$$Q=(V_{\max}-V_{\min})/N = (10V - 0V)/8 = \textcolor{red}{1.25V}$$

Accuracy of A/D Conversion

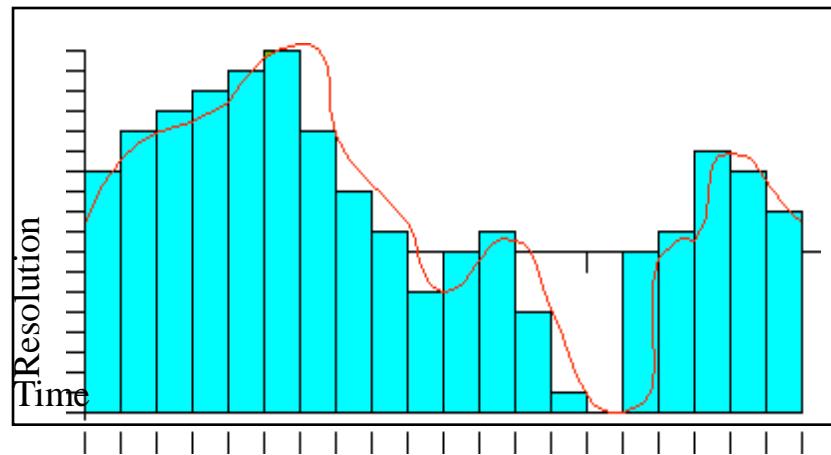
There are two ways to best improve the accuracy of A/D conversion:

- increasing the resolution which improves the accuracy in measuring the amplitude of the analog signal.
- increasing the sampling rate which increases the maximum frequency that can be measured.

- Low Accuracy



- Improved

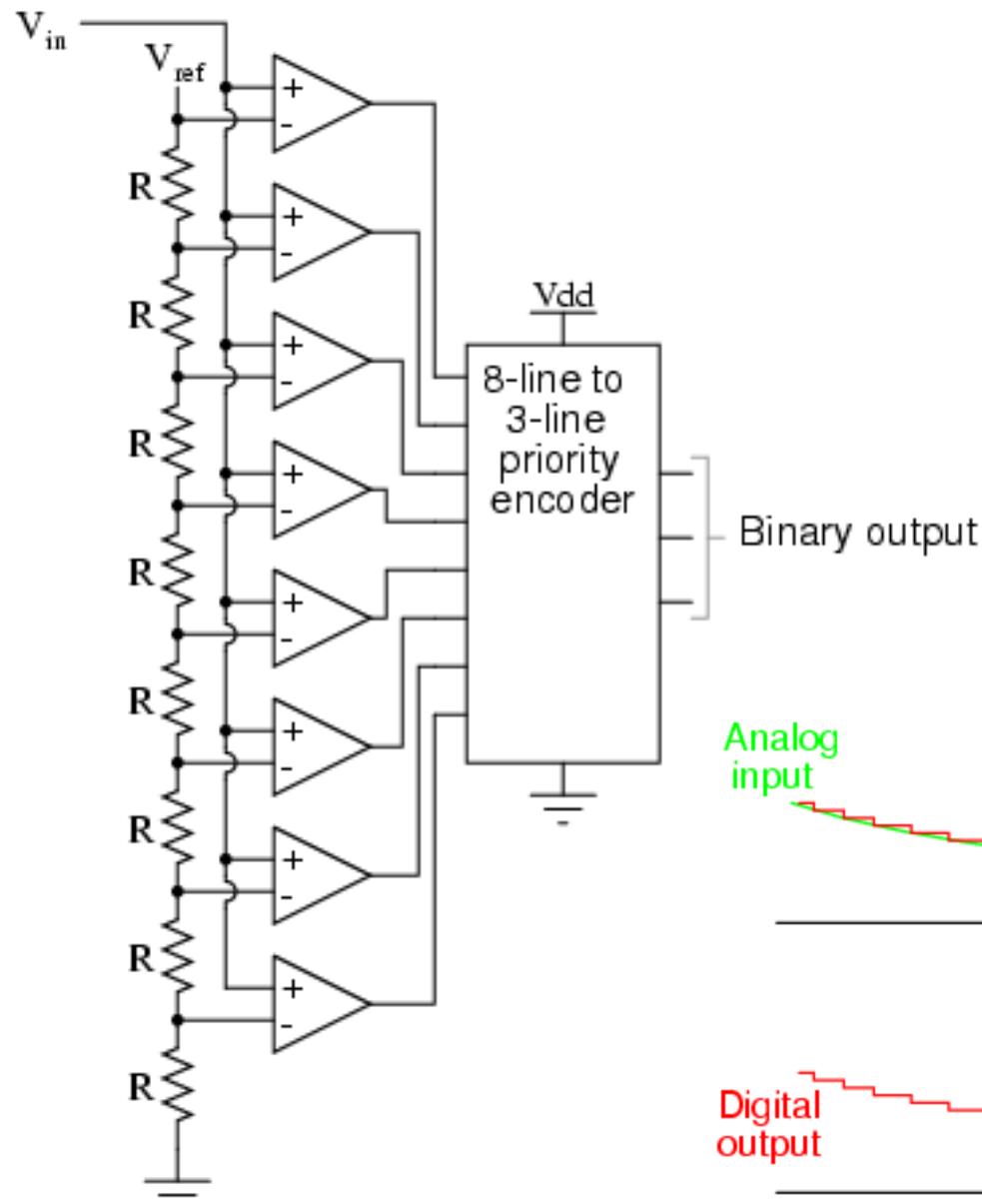


A/D Converter Types

- Flash type A/D Converter
- Delta-Sigma type A/D Converter
- Dual-Slope type A/D Converter
- Successive-Approximation type A/D Converter

Flash type A/D Converter

- Consists of a series of comparators, each one comparing the input signal to a unique reference voltage.
- The comparator outputs connect to the inputs of a priority encoder circuit, which produces a binary output



Analog
input

Time →

Digital
output

Time →

- As the analog input voltage exceeds the reference voltage at each comparator, the comparator outputs will sequentially saturate to a high state.
- The priority encoder generates a binary number based on the highest-order active input, ignoring all other active inputs.

Advantages

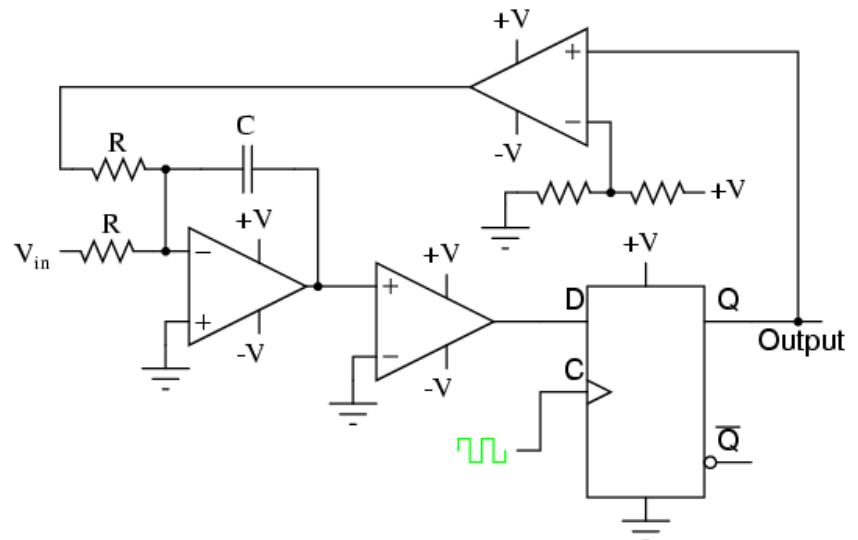
- Simplest in terms of operational theory
- Most efficient in terms of speed, very fast
 - limited only in terms of comparator and gate propagation delays

Disadvantages

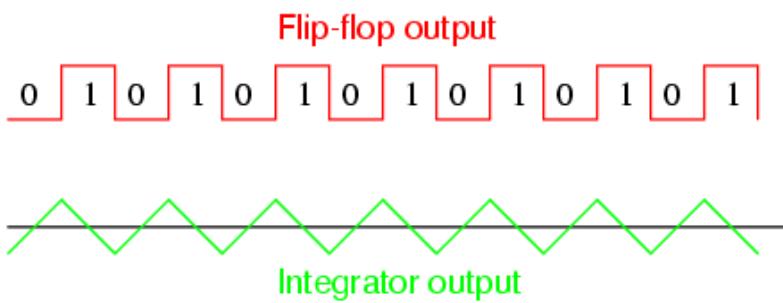
- Lower resolution
- Expensive
- For each additional output bit, the number of comparators is doubled
 - i.e. for 8 bits, 256 comparators needed

Sigma-Delta type A/D Converter

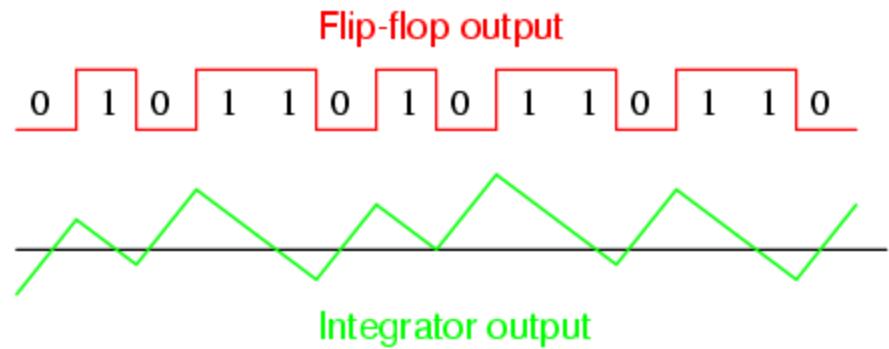
- Over sampled input signal goes to the integrator
- Output of integration is compared to GND
- Iterates to produce a serial bit stream
- Output is serial bit stream with # of 1's proportional to V_{in}



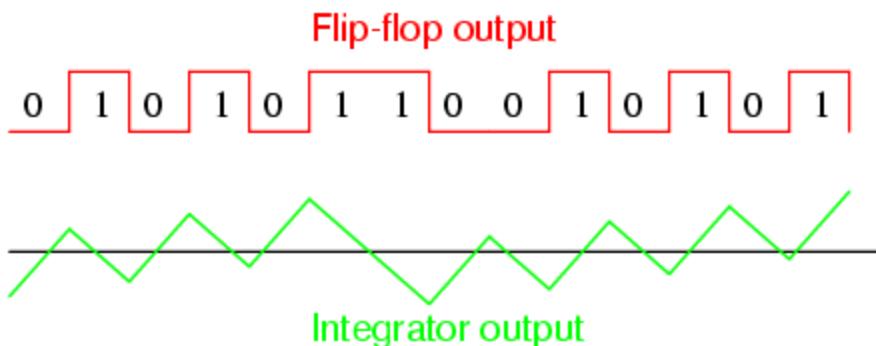
$\Delta\Sigma$ converter operation with
0 volt analog input



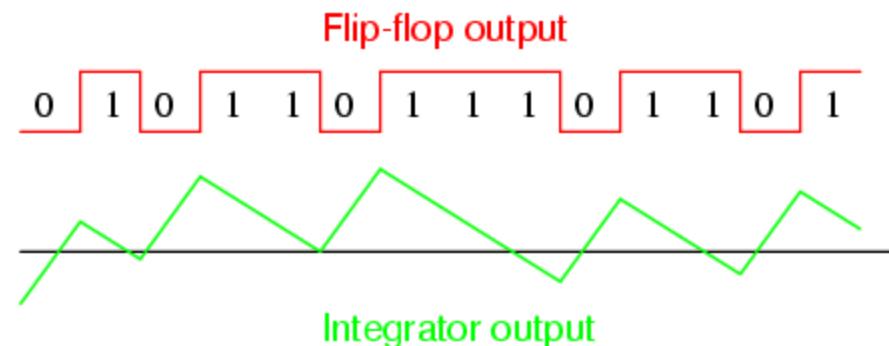
$\Delta\Sigma$ converter operation with
medium negative analog input



$\Delta\Sigma$ converter operation with
small negative analog input



$\Delta\Sigma$ converter operation with
large negative analog input



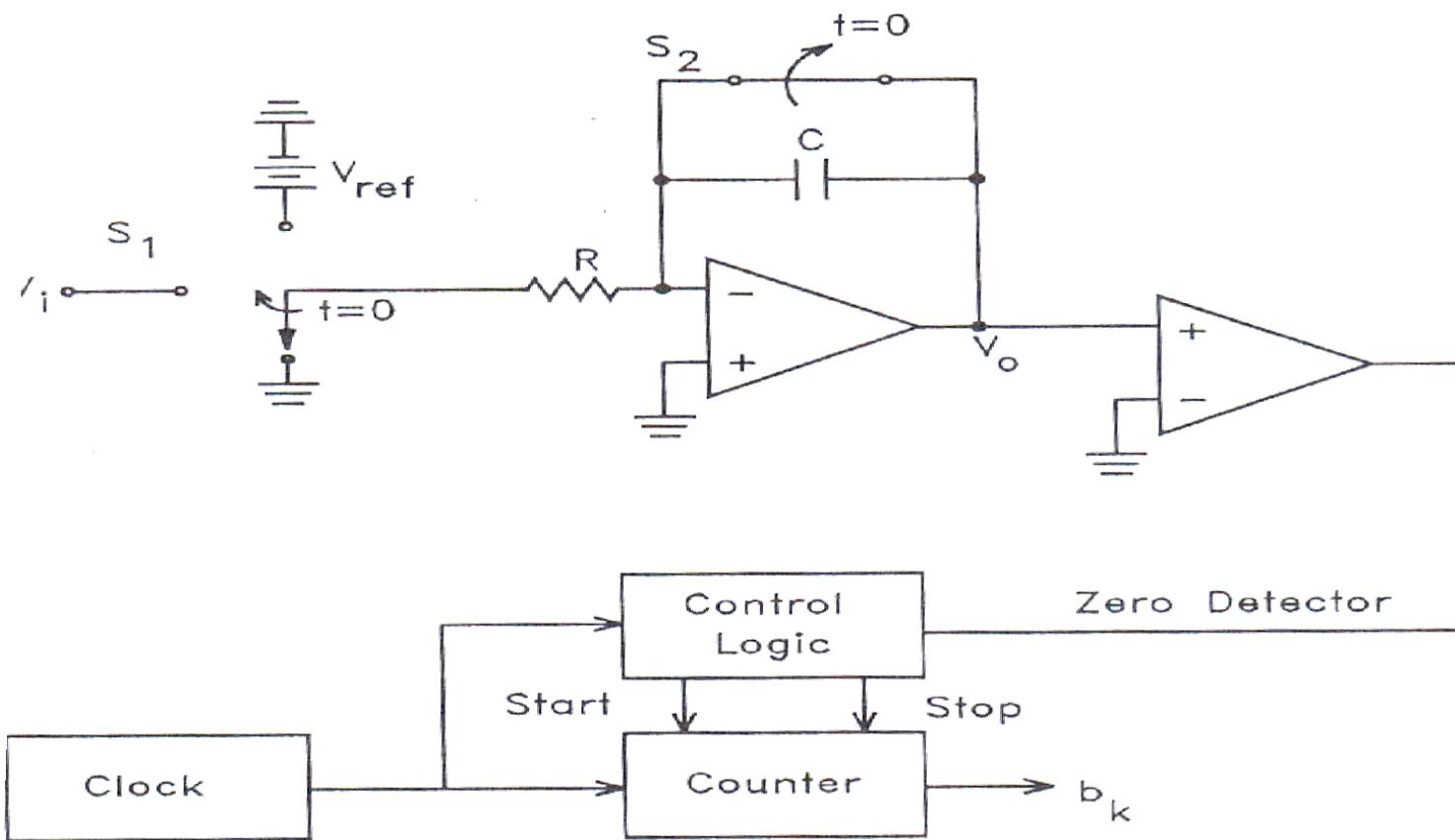
Advantages

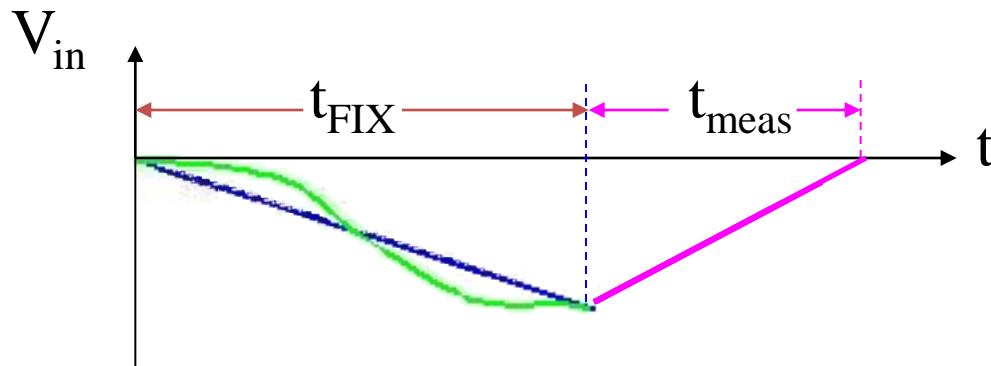
- High resolution
- No precision external components needed

Disadvantages

- Slow due to oversampling

Dual-Slope type A/D Converter





- The sampled signal charges a capacitor for a fixed amount of time
- By integrating over time, noise integrates out of the conversion
- Then the ADC discharges the capacitor at a fixed rate with the counter counts the ADC's output bits. A longer discharge time results in a higher count

Advantages

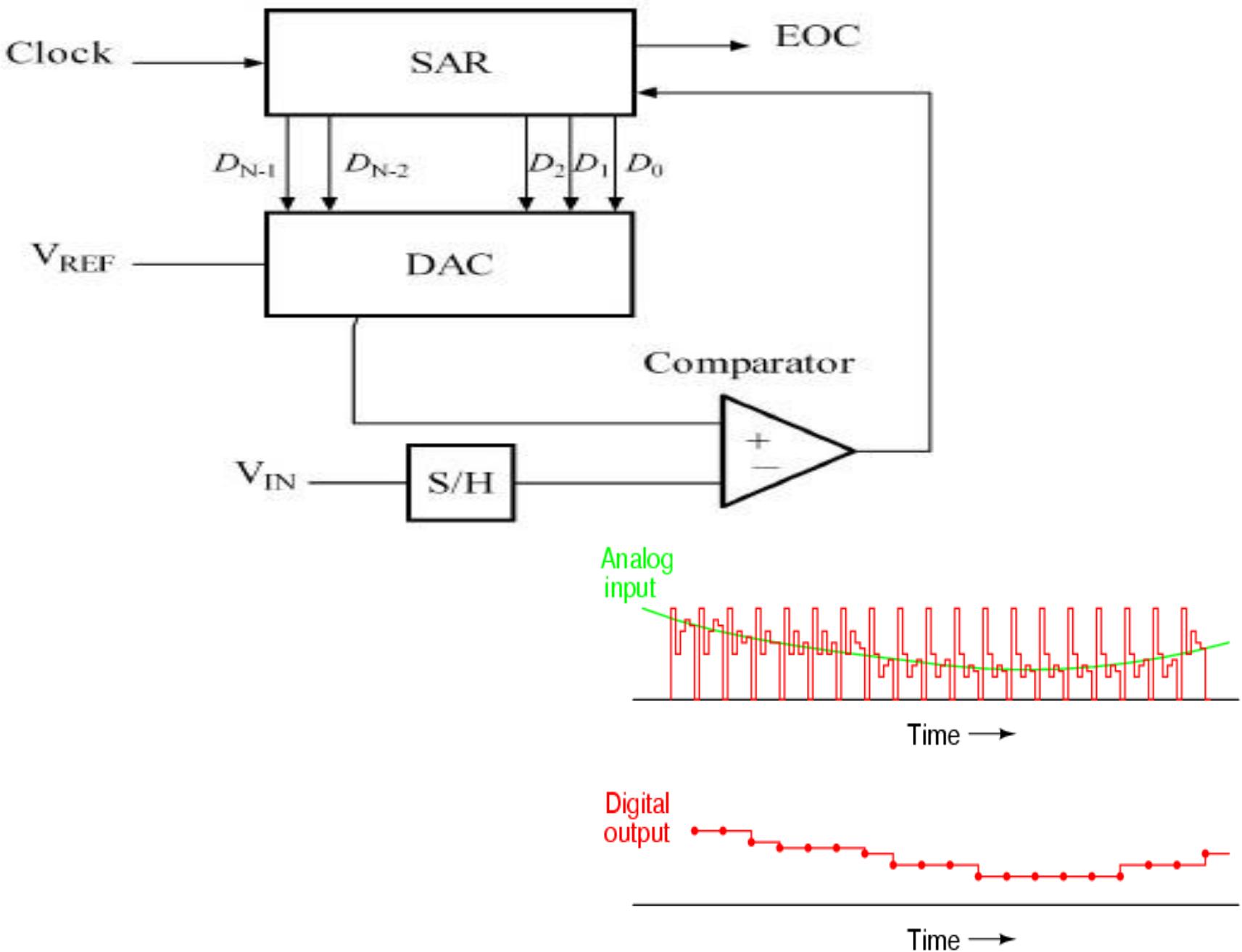
- Input signal is averaged
- Greater noise immunity than other ADC types
- High accuracy

Disadvantages

- Slow
- High precision external components required to achieve accuracy

Successive-Approximation type A/D Converter

- A Successive Approximation Register (SAR) is added to the circuit.
- Instead of counting up in binary sequence, this register counts by trying all values of bits starting with the MSB and finishing at the LSB.
- The register monitors the comparators output to see if the binary count is greater or less than the analog signal input and adjusts the bits accordingly.



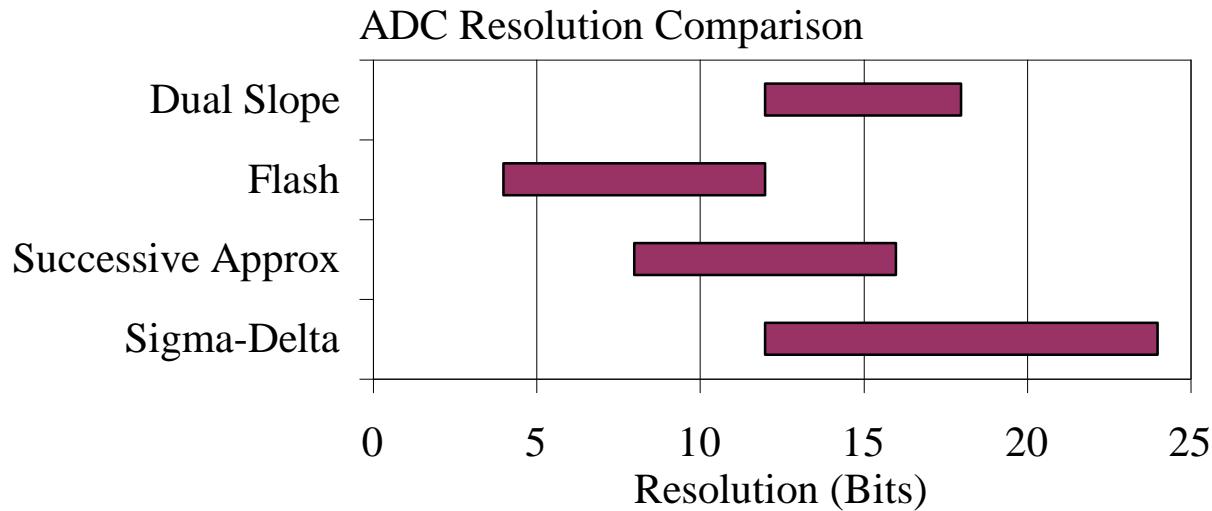
Advantages

- Capable of high speed and reliable
- Medium accuracy compared to other ADC types
- Good tradeoff between speed and cost
- Capable of outputting the binary number in serial (one bit at a time) format.

Disadvantages

- Higher resolution successive approximation ADC's will be slower
- Speed limited to ~5Msps

ADC Types Comparison



| Type | Speed (relative) | Cost (relative) |
|------------------|------------------|-----------------|
| Dual Slope | Slow | Med |
| Flash | Very Fast | High |
| Successive Appox | Medium – Fast | Low |
| Sigma-Delta | Slow | Low |

Application of ADC

- ADC are used virtually everywhere where an analog signal has to be processed, stored, or transported in digital form.
- Some examples of ADC usage are digital voltmeters, cell phone, thermocouples, and digital oscilloscope.
- Microcontrollers commonly use 8, 10, 12, or 16 bit ADCs, our micro controller uses an 8 or 10 bit ADC.

DIGITAL ELECTRONICS

III Semester CSE & CST



Mr. Srikanta Patnaik Mr. Ajit Kumar Patro Mr. Ami Kumar Parida

DEPARTMENT OF ELECTRONICS AND COMMUNICATION ENGINEERING

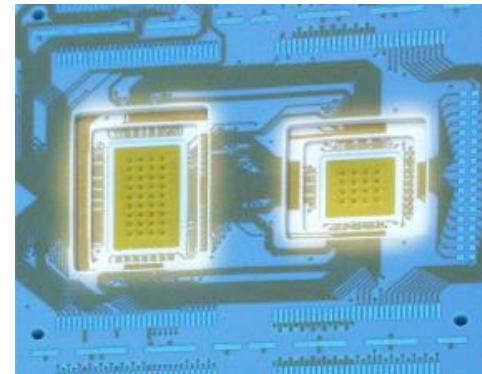
SCHOOL OF ENGINEERING & TECHNOLOGY

GIET UNIVERSITY, GUNUPUR, ODISHA

IC Logic Families

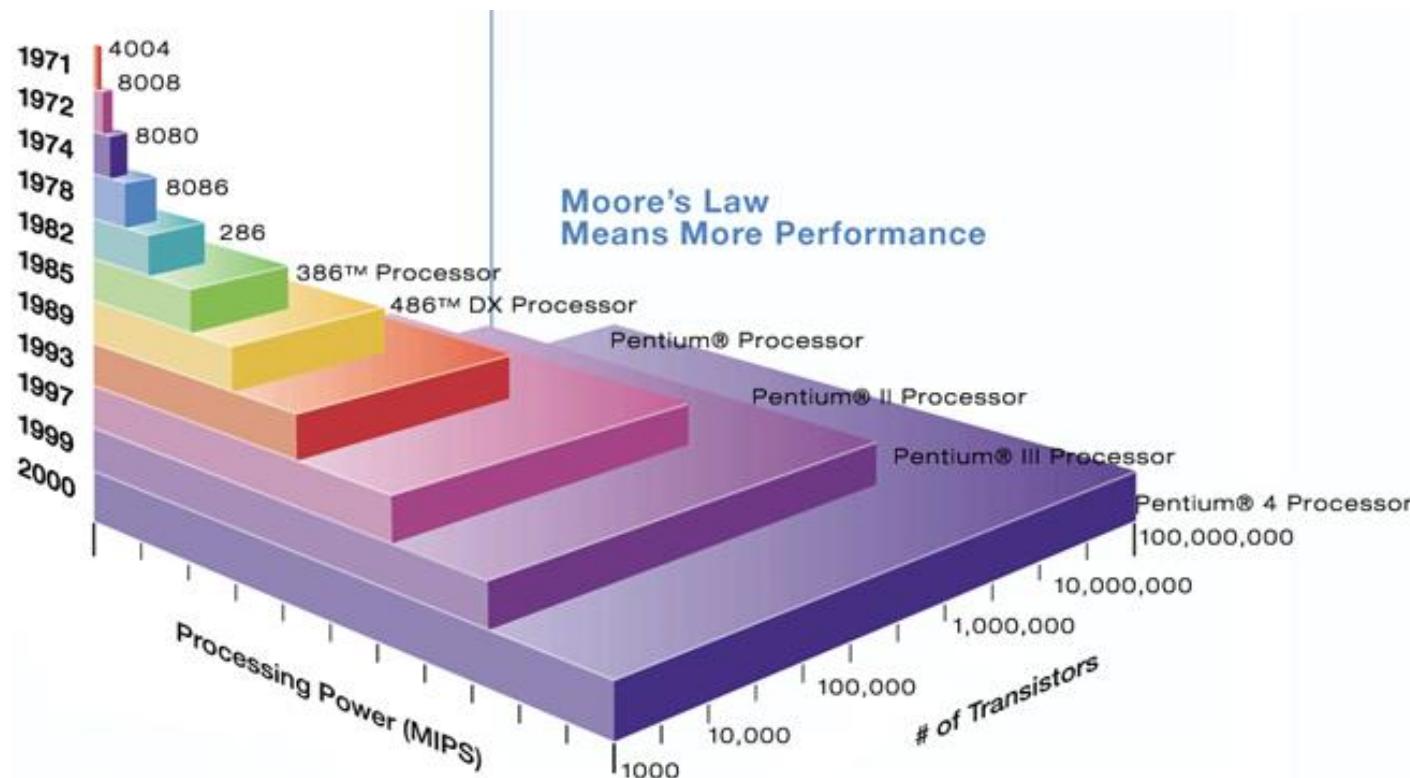
Integration Levels

- Gate/transistor ratio is roughly 1/10
 - SSI < 12 gates/chip
 - MSI < 100 gates/chip
 - LSI ...1K gates/chip
 - VLSI ...10K gates/chip
 - ULSI ...100K gates/chip
 - GSI ...1Meg gates/chip



Moore's law

- A prediction made by Moore (a co-founder of Intel) in 1965: "... a number of transistors to double every 2 years."



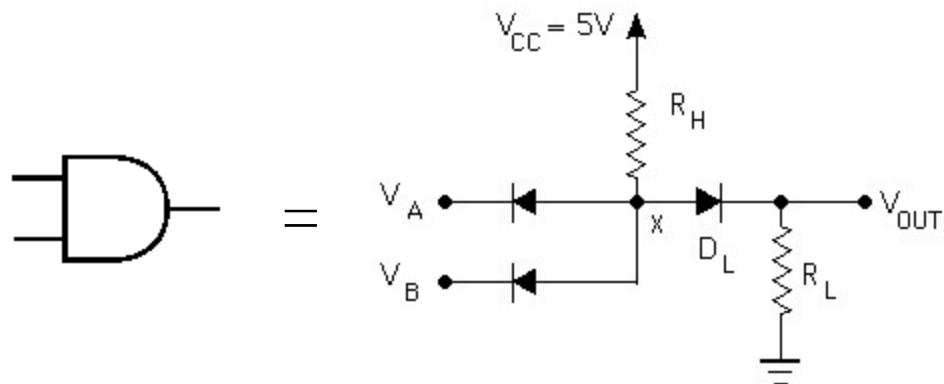
Introduction

- ICs are fabricated using various technologies such as TTL, ECL, and IIL which use bipolar transistors, whereas the MOS and CMOS technologies use unipolar MOSFETs.

In the beginning...

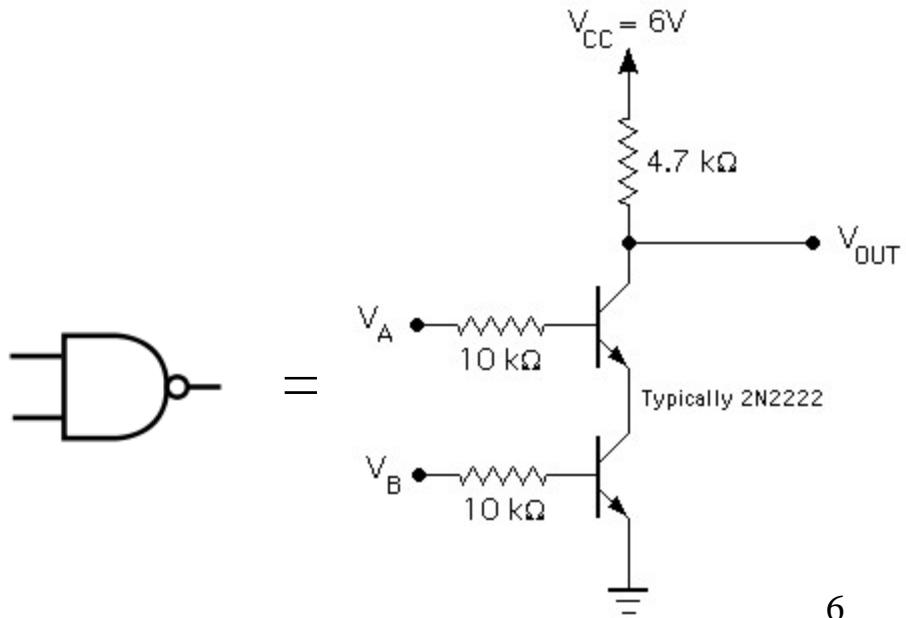
Diode Logic (DL)

- simplest; does not scale
- NOT not possible (need an active element)



Resistor-Transistor Logic (RTL)

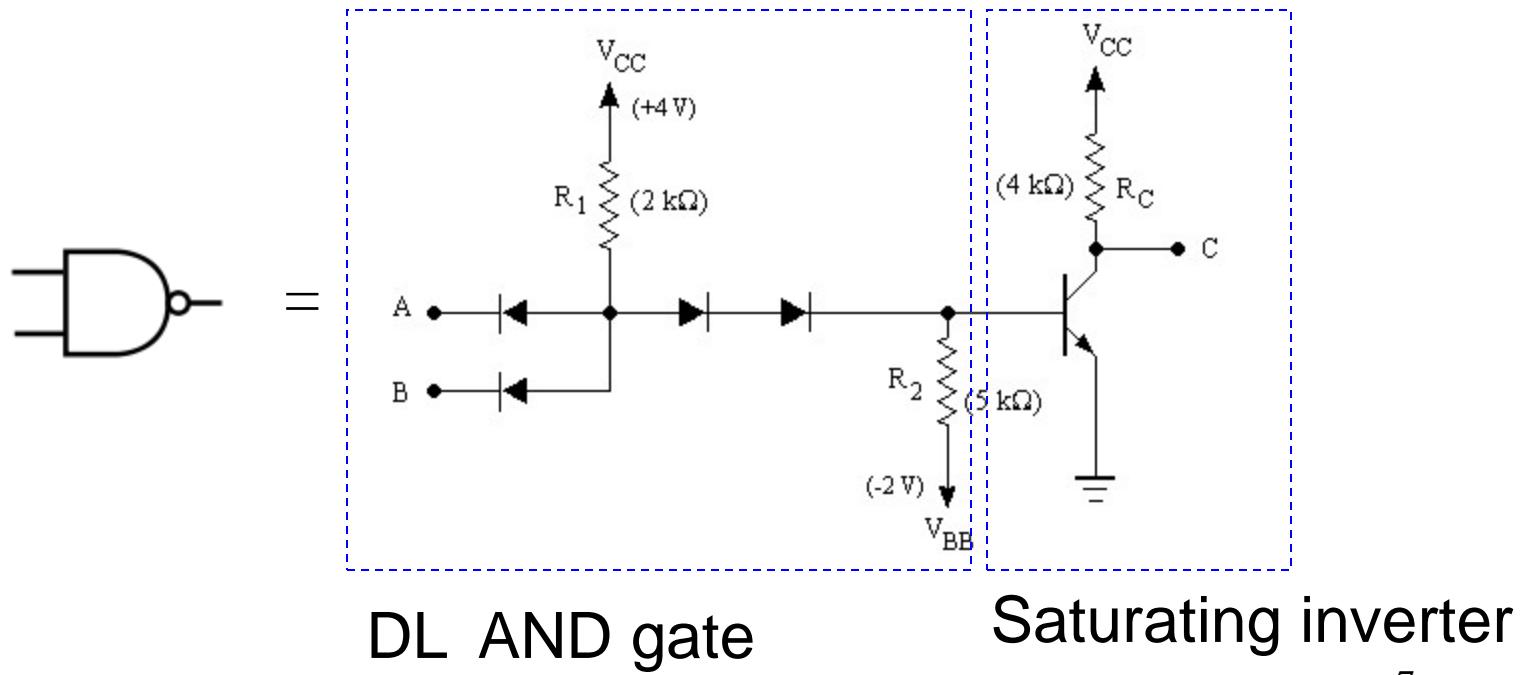
- replace diode switch with a transistor switch
- can be cascaded
- large power draw



was...

Diode-Transistor Logic (DTL)

- essentially diode logic with transistor amplification
- reduced power consumption
- faster than RTL



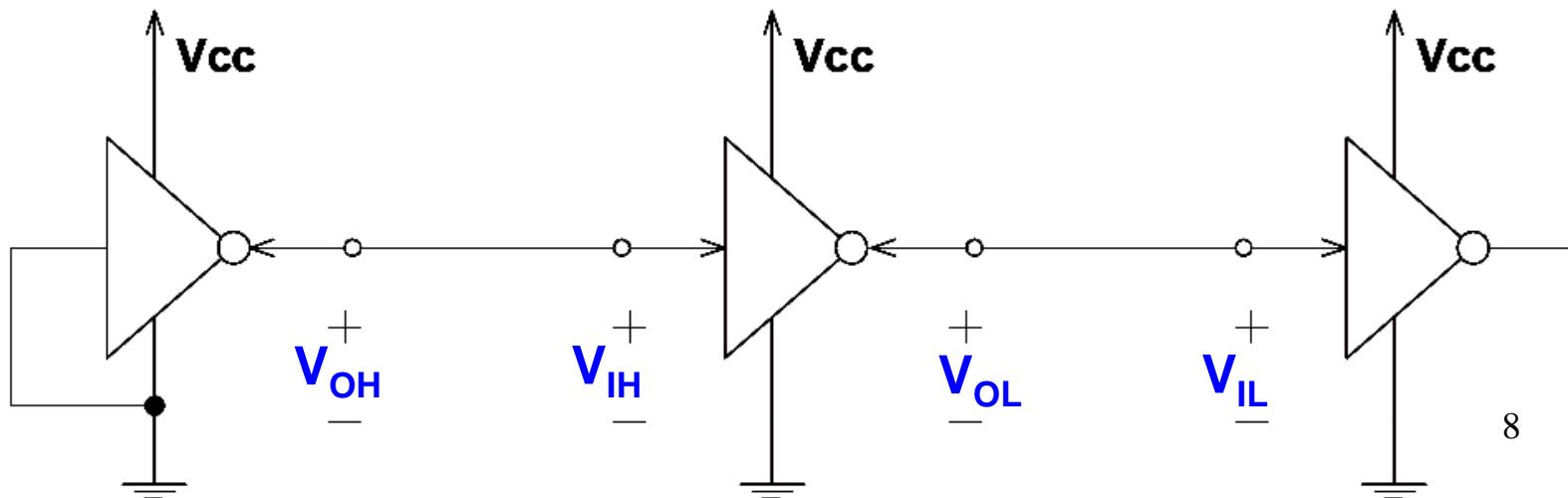
Logic families: V levels

$V_{OH}(\min)$ – The minimum voltage level at an output in the logical “1” state under defined load conditions

$V_{OL}(\max)$ – The maximum voltage level at an output in the logical “0” state under defined load conditions

$V_{IH}(\min)$ – The minimum voltage required at an input to be recognized as “1” logical state

$V_{IL}(\max)$ – The maximum voltage required at an input that still will be recognized as “0” logical state



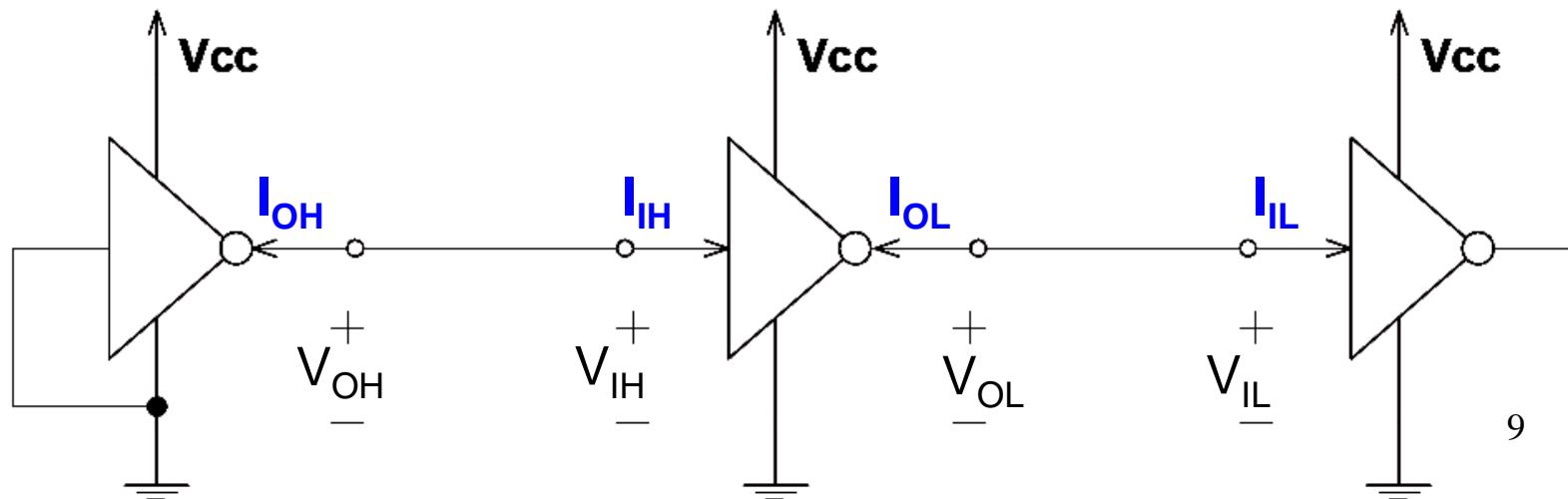
Logic families: I requirements

I_{OH} – Current flowing into an output in the logical “1” state under specified load conditions

I_{OL} – Current flowing into an output in the logical “0” state under specified load conditions

I_{IH} – Current flowing into an input when a specified HI level is applied to that input

I_{IL} – Current flowing into an input when a specified LO level is applied to that input

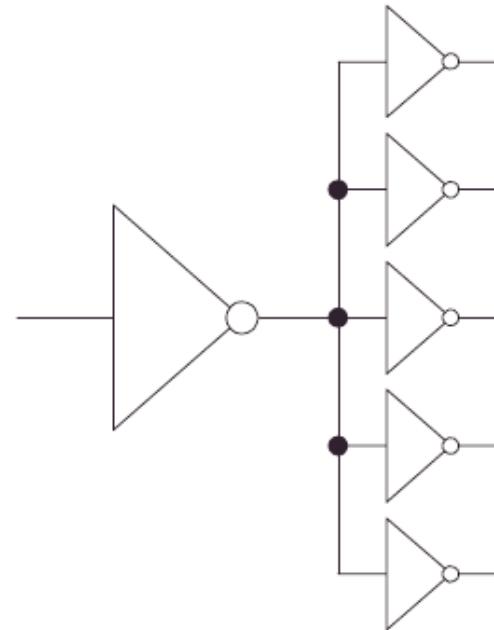


Logic families: Fan-in and Fan-out

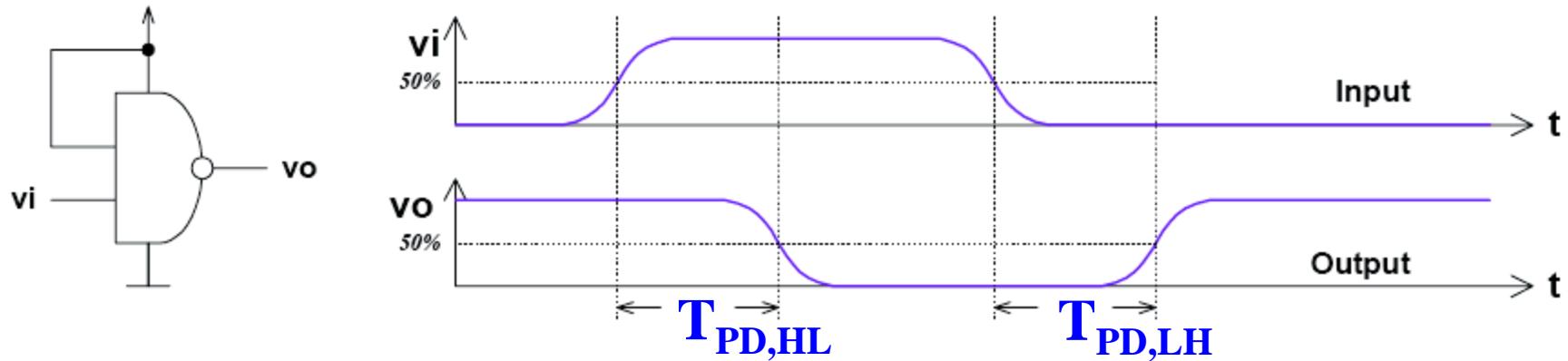
Fan-in: The *fan-in* of a logic gate is defined as the number of inputs that the gate is designed to handle.

Fan-out: The *fan-out* of a logic gate is defined as the maximum number standard loads that the output of the gate can drive without impairing its normal operation.

$$\text{DC fanout} = \min\left(\frac{I_{OH}}{I_{IH}}, \frac{I_{OL}}{I_{IL}}\right)$$



Logic families: propagation delay

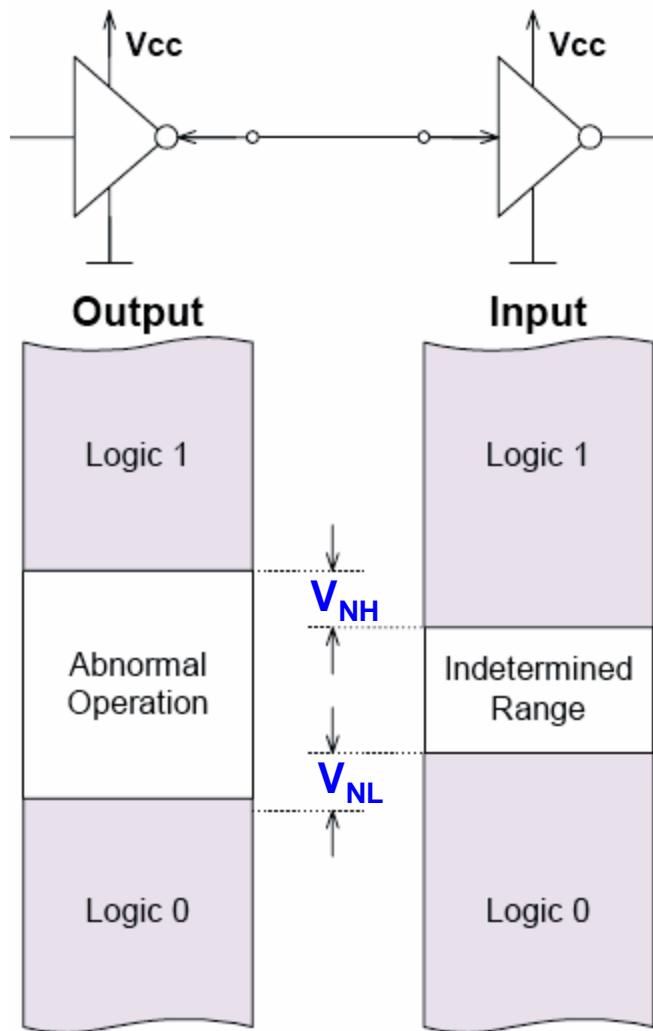


$T_{PD,HL}$ – input-to-output propagation delay from HI to LO output

$T_{PD,LH}$ – input-to-output propagation delay from LO to HI output

Speed-power product: $T_{PD} \times P_{avg}$

Logic families: noise margin



HI state noise margin:

$$V_{NH} = V_{OH}(\min) - V_{IH}(\min)$$

LO state noise margin:

$$V_{NL} = V_{IL}(\max) - V_{OL}(\max)$$

Noise margin:

$$V_N = \min(V_{NH}, V_{NL})$$

Comparison of logic families

| Logic family | Propagation delay time (ns) | Power dissipation per gate (mW) | Noise margin (V) | Fan-in | Fan-out | Cost |
|--------------|-----------------------------|---------------------------------|------------------|--------|---------|----------|
| TTL | 9 | 10 | 0.4 | 8 | 10 | Low |
| ECL | 1 | 50 | 0.25 | 5 | 10 | High |
| MOS | 50 | 0.1 | 1.5 | | 10 | Low |
| CMOS | < 50 | 0.01 | 5 | 10 | 50 | Low |
| III | 1 | 0.1 | 0.35 | 5 | 8 | Very low |

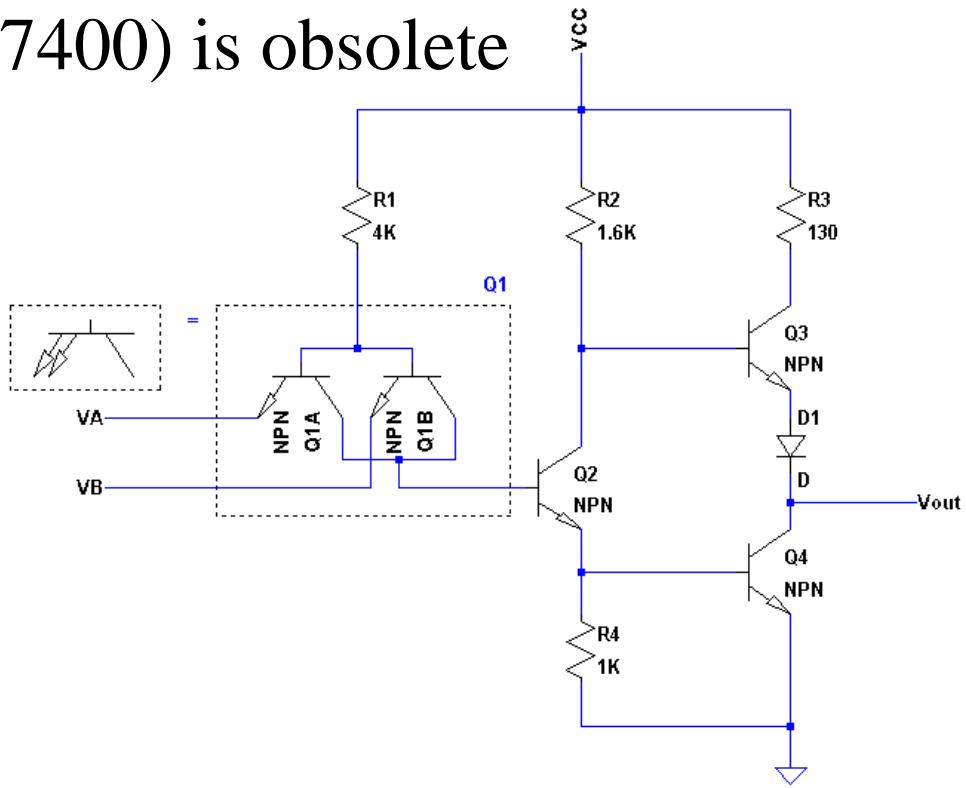
TTL

Bipolar Transistor-Transistor Logic (TTL)

- first introduced by in 1964 (Texas Instruments)
- TTL has shaped digital technology in many ways
- Standard TTL family (e.g. 7400) is obsolete
- Newer TTL families still used (e.g. 74ALS00)

Distinct features

- Multi-emitter transistors
- Totem-pole transistor arrangement
- Open LTspice example:
TTL NAND...

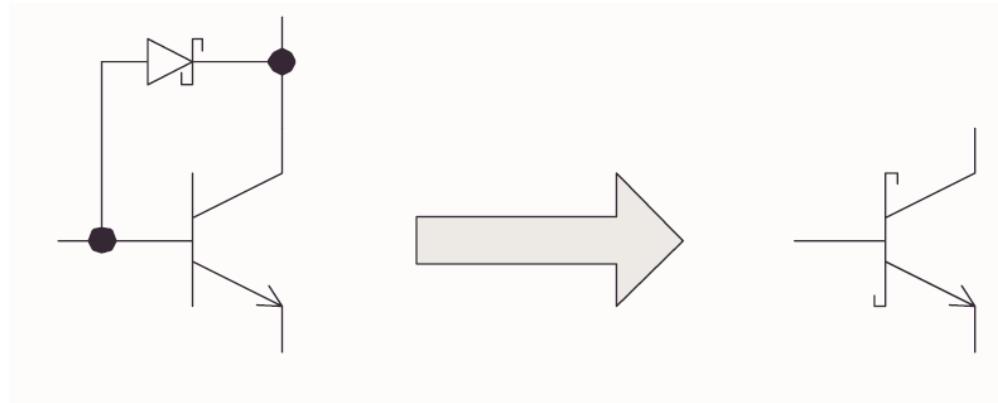


2-input NAND

TTL evolution

Schottky series (74LS00) TTL

- A major slowdown factor in BJTs is due to transistors going in/out of saturation
- Shottky diode has a lower forward bias (0.25V)
- When BC junction would become forward biased, the Schottky diode bypasses the current preventing the transistor from going into saturation



ECL

Emitter-Coupled Logic (ECL)

- PROS: Fastest logic family available ($\sim 1\text{ns}$)
- CONS: low noise margin and high power dissipation
- Operated in emitter coupled geometry (recall differential amplifier or emitter-follower), transistors are biased and operate near their Q-point (never near saturation!)
- Logic levels. “0”: -1.7V . “1”: -0.8V
- Such strange logic levels require extra effort when interfacing to TTL/CMOS logic families.
- Open LTspice example: ECL inverter...

CMOS

Complimentary MOS (CMOS)

- Other variants: NMOS, PMOS (obsolete)
- Very low static power consumption
- Scaling capabilities (large integration all MOS)
- Full swing: rail-to-rail output
- Things to watch out for:
 - don't leave inputs floating (in TTL these will float to HI, in CMOS you get undefined behaviour)
 - susceptible to electrostatic damage (finger of death)
- Open LTspice example: CMOS NOT and NAND...

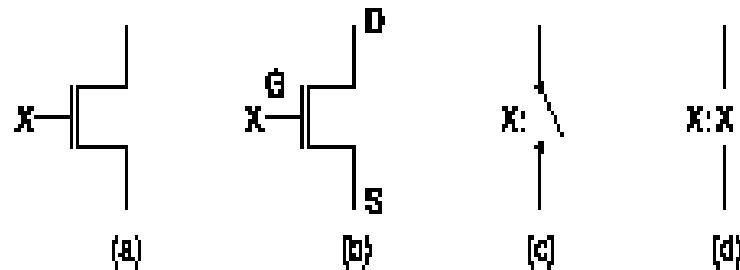
CMOS Combinational Circuits

- Implementation of logic gates and other structures using CMOS technology.
- Basic element: *transistor*
- 2 types of transistors:
 - *n-channel (nMOS)* and *p-channel (pMOS)*
 - Type depends on the semiconductor materials used to implement the transistor.
 - We want to model transistor behavior at the logic level in order to study the behavior of CMOS circuits → view pMOS and nMOS transistors as switches.

CMOS transistors as Switches

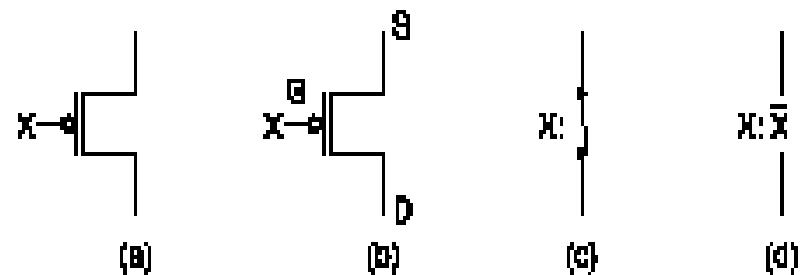
3 terminals in CMOS transistors:

- G : Gate
- D : Drain
- S : Source



□ **FIGURE 1**
Symbol and Switch Model for n-Channel Transistor

nMOS transistor/switch
 $X=1$ switch closes (ON)
 $X=0$ switch opens (OFF)



□ **FIGURE 2**
Symbol and Switch Model for p-Channel Transistor

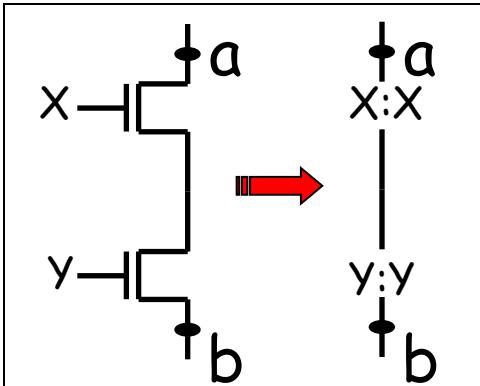
pMOS transistor/switch
 $X=1$ switch opens (OFF)
 $X=0$ switch closes (ON)

Networks of Switches

- Use switches to create networks that represent CMOS logic circuits.
- To implement a function F , create a network s.t. there is a path through the network whenever $F=1$ and no path when $F=0$.
- Two basic structures:
 - Transistors in Series
 - Transistors in Parallel

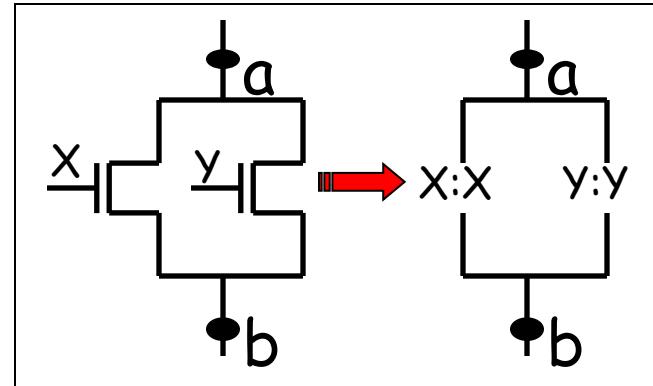
Transistors in Series/Parallel

nMOS in Series



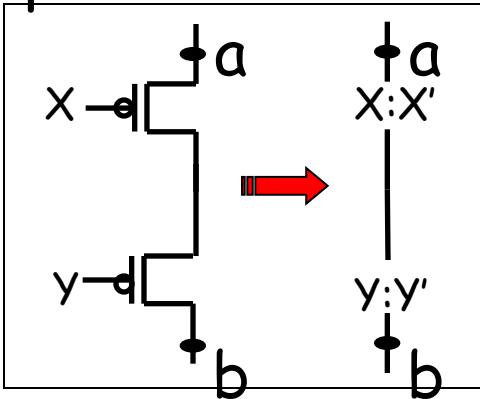
Path between points a and b exists if both X and Y are 1
 $\rightarrow X \cdot Y$

nMOS in Parallel



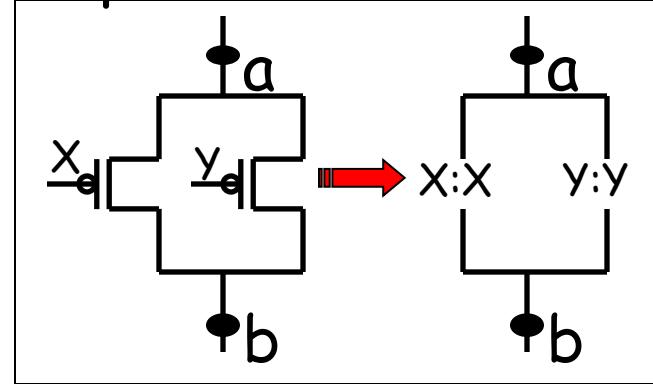
Path between points a and b exists if either X or Y are 1
 $\rightarrow X + Y$

pMOS in Series



Path between points a and b exists if both X and Y are 0
 $\rightarrow X' \cdot Y'$

pMOS in Parallel

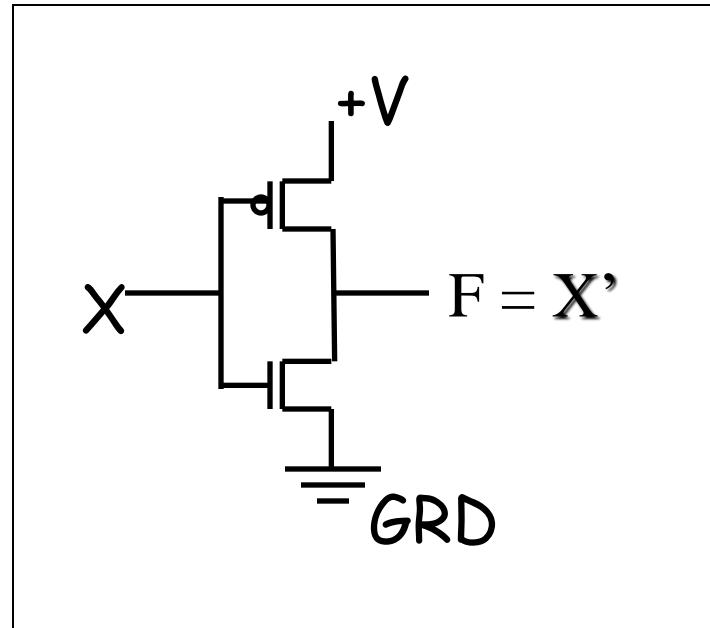
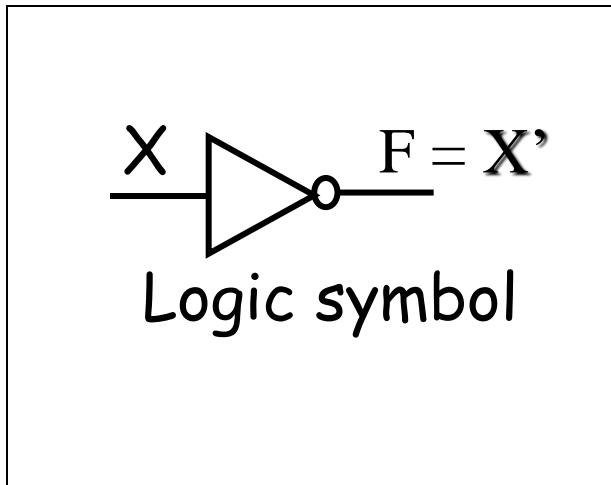


Path between points a and b exists if either X or Y are 0
 $\rightarrow X' + Y'$

Networks of Switches (cont.)

- In general:
 1. nMOS in series is used to implement AND logic
 2. pMOS in series is used to implement NOR logic
 3. nMOS in parallel is used to implement OR logic
 4. pMOS in parallel is used to implement NAND logic
- Observe that:
 - 1 is the complement of 4, and vice-versa
 - 2 is the complement of 3, and vice-versa

CMOS Inverter



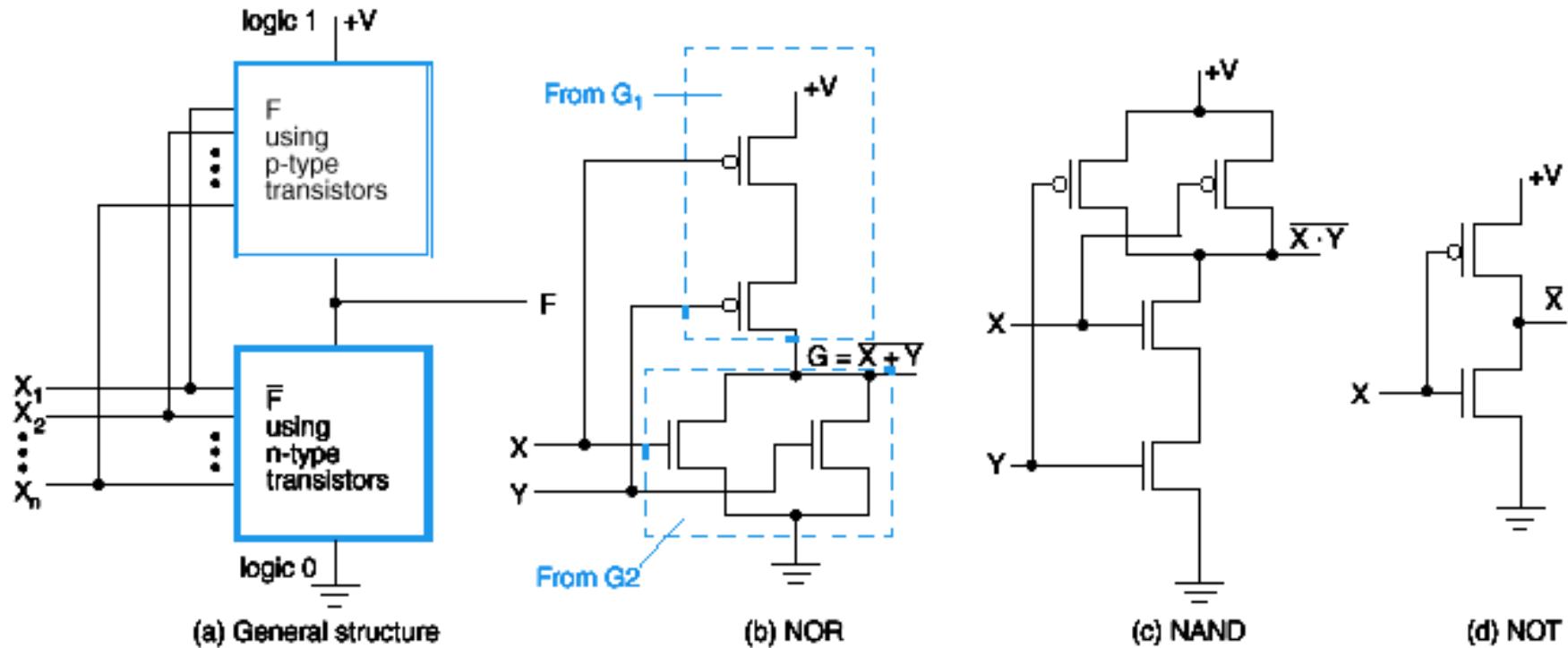
Operation:

- $X=1 \rightarrow$ nMOS switch conducts (pMOS is open)
and draws from $GRD \rightarrow F=0$
- $X=0 \rightarrow$ pMOS switch conducts (nMOS is open)
and draws from $+V \rightarrow F=1$

Transistor-level schematic

Fully Complementary CMOS Networks

Basic Gates

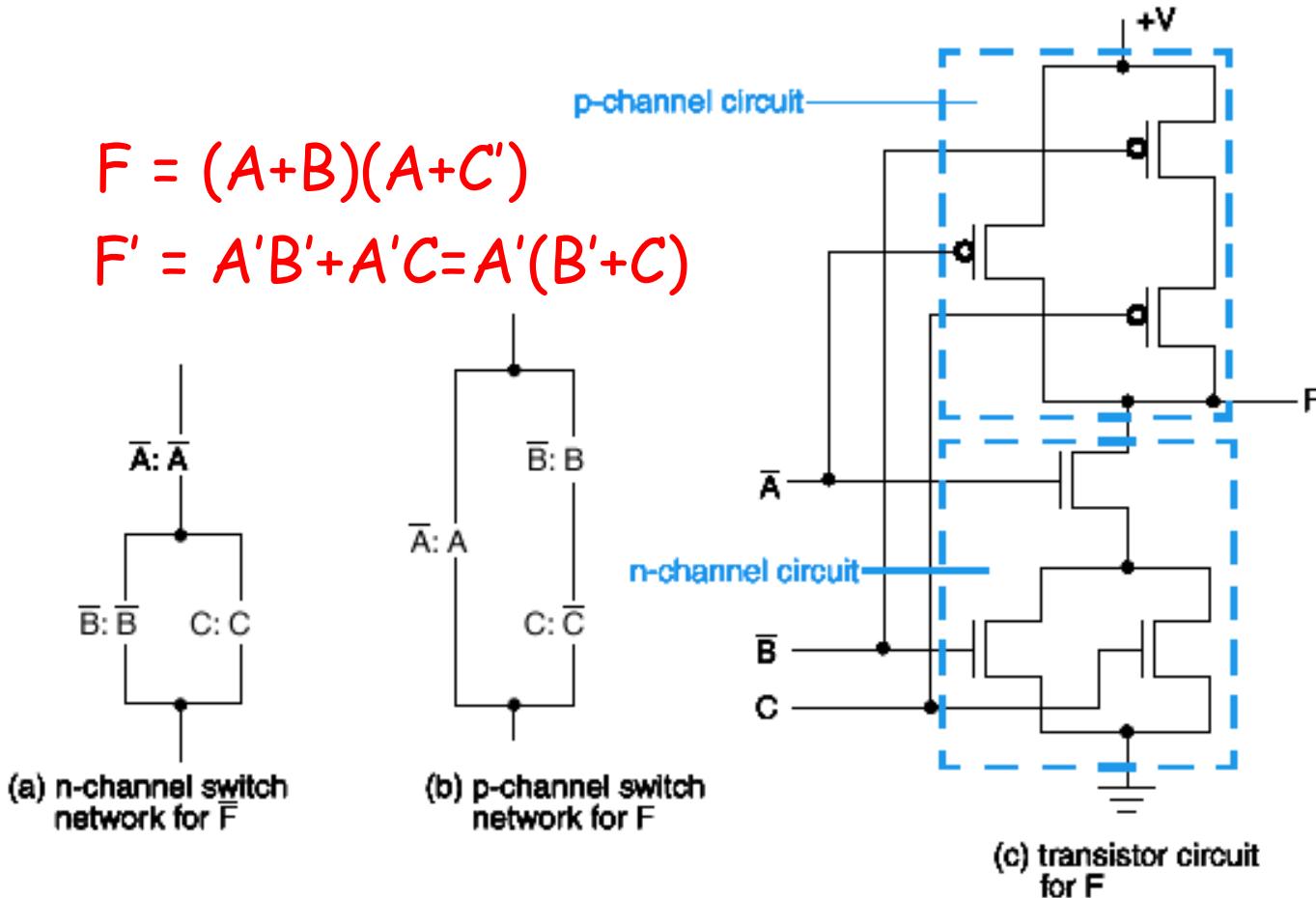


Fully Complementary CMOS Complex Gates

Given a function F:

1. First take the complement of F to form F'
2. Implement F' as an nMOS net and connect it to GRD (pull-down net) and F.
3. Find dual of F' , implement it as a pMOS net and connect it to +V (pull-up net) and F.
4. Connect switch inputs.

Fully Complementary CMOS Networks



- $F = (AB + CD)'$

