CARDAMOM PLANTERS' ASSOCIATION COLLEGE
(Re-Accredited With 'A' Grade By NAAC)
Pankajam Nagar, Bodinayakanur - 625 582.

DEPARTMENT OF CS & IT

Digital Computer Fundamentals

UNIT – I

Number Systems and Codes: Number System – Base Conversion – Binary Codes – Code Conversion.
Digital Logic: Logic Gates – Truth Tables – Universal Gates

Number Systems and Codes

1.1. Number System

A **number system** is a system of writing used to represent numbers. It is a mathematical system with **base (radix) n**, where *n* represents the total number of digits used in that system.

Example:

- Decimal number system uses **10 digits (0–9)**

Radix Point

The **radix point** is a point used to separate the **integer part** and the **fractional part** of a number.

Example:

- Decimal: 123.45
- Binary: 101.101

Base or Radix

The **base (radix)** of a number system is the total number of symbols (digits) used.

Examples:

- Decimal number system → Base = 10
- Binary number system → Base = 2
- Octal number system → Base = 8
- Hexadecimal number system → Base = 16

Most Significant Bit (MSB)

The **leftmost bit** of a binary number having the **highest place value** is called the **Most Significant Bit (MSB)**.

Example:
Binary number: **1101**
→ MSB = **1**

Least Significant Bit (LSB)

The **rightmost bit** of a binary number having the **lowest place value** is called the **Least Significant Bit (LSB)**.

Example:
Binary number: **1101**
→ LSB = **1**

Bit

A **bit (Binary Digit)** is the smallest unit of data in a computer.
It can have only **two values: 0 or 1**.

Nibble

A group of **4 bits** is called a **nibble**.

Examples:

- 0110
- 1110

Byte

A group of **8 bits** is called a **byte**.

Examples:

- 01101101
- 11010011

1.2. Types of Number Systems

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

1.2.1. Binary Number System

- **Base:** 2
- **Digits used:** 0 and 1
- Widely used in **digital computers and electronics**

Examples:

- 11001_2
- 0101_2

1.2.2. Octal Number System

- **Base:** 8
- **Digits used:** 0 to 7
- Used as a **shorthand representation of binary numbers**

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

Examples:

- 157_8
- 45_8

1.2.3. Decimal Number System

- **Base:** 10
- **Digits used:** 0 to 9
- Most commonly used number system in daily life

Examples:

- 256_{10}
- 98_{10}

1.2.4. Hexadecimal Number System

- **Base:** 16
- **Digits used:** 0–9 and A–F
    - A = 10
    - B = 11
    - C = 12
    - D = 13
    - E = 14
    - F = 15
- Used in **computer memory addressing and programming**

Decimal Number	Hexadecimal Number	Binary Number
0	0	0000
1	1	0001
2	2	0010
3	3	0011
4	4	0100
5	5	0101
6	6	0110
7	7	0111
8	8	1000
9	9	1001
10	A	1010
11	B	1011
12	C	1100
13	D	1101
14	E	1110
15	F	1111

Examples:

- $2F_{16}$
- $A9_{16}$

1.3. Why Number Systems Matter

- **Binary Number System**
  - The **binary number system** is the internal language of computers.
  - All digital systems operate using only **two states: 0 and 1**, which represent **OFF and ON** conditions in electronic circuits.
  - Binary numbers are used in **logic gates, data storage, and processing operations** inside the computer.
- **Octal and Hexadecimal Number Systems**
  - The **octal** and **hexadecimal** number systems are used to simplify the representation of long binary numbers.
  - Writing and reading large binary strings is difficult and error-prone.
  - Octal and hexadecimal provide **shorter, more readable forms**, making them essential for **programmers, engineers, and system designers**, especially in debugging and memory addressing.
- **Number System Conversions**
  - Digital systems constantly perform **number system conversions**.
  - Human beings understand numbers in **decimal form**, while computers work in **binary form**.
  - Conversions help translate **human-readable decimal data into machine-readable binary**, and convert results back into decimal for user display.

1.4. Number System Conversions

Number system conversion is the process of converting a number from one base to another. In digital systems, conversions are required to translate **human-readable decimal numbers** into **machine-readable binary numbers** and vice versa.

Types of Number System Conversions

1. Decimal → Binary / Octal / Hexadecimal
2. Binary → Decimal
3. Binary → Octal
4. Binary → Hexadecimal
5. Octal → Binary
6. Hexadecimal → Binary
7. Octal / Hexadecimal → Decimal

1.4.1. Decimal → Binary

Method: Repeated Division by 2

Steps:

For Integer Part

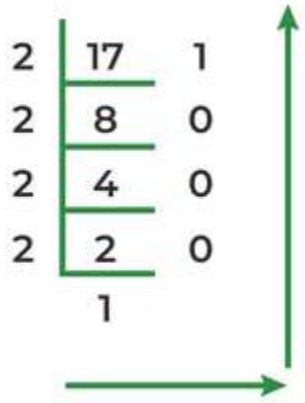
1. Divide the decimal number repeatedly by **2**.
2. Note the remainder after each division.
3. Continue until the quotient becomes **0**.
4. Write the remainders in **reverse order** (bottom to top).

For Fractional Part

1. Multiply the fractional part by **2**.
2. Take the **integer part** of the result.
3. Repeat with the remaining fraction.
4. Stop when the fraction becomes **0** or after required precision.
5. Write the integer parts in **same order**.

Example:

Convert 17_{10} to binary



Answer:10001

1.4.2. Binary → Decimal

Method: Positional Value Method

Formula:

$$\sum(bit \times 2^{position})$$

Procedure:

1. Write positional values of base 2.
2. For integer part → powers of **2^0, 2^1, 2^2,** … from right to left.
3. For fractional part → powers of **2^{-1}, 2^{-2},** … from left to right.
4. Multiply each bit with its positional value.
5. Add all values to get the decimal number.

Example:

1. Convert 11001_2 to decimal

$$= (1 \times 2^4) + (1 \times 2^3) + (0 \times 2^2) + (0 \times 2^1) + (1 \times 2^0)$$
$$= 16 + 8 + 0 + 0 + 1 = 25$$

Final Answer:

$11001_2 = 25_{10}$

2. Convert 1011.101_2 to decimal

Integer part:

$$= (1 \times 2^3) + (0 \times 2^2) + (1 \times 2^1) + (1 \times 2^0) = 8 + 0 + 2 + 1 = 11$$

Fractional part:

$$= (1 \times 2^{-1}) + (0 \times 2^{-2}) + (1 \times 2^{-3}) = 0.5 + 0 + 0.125 = 0.625$$

Final Answer:

$1011.101_2 = 11.625_{10}$

1.4.3. Decimal → Octal

Method: Repeated Division by 8

Procedure:

For Integer Part

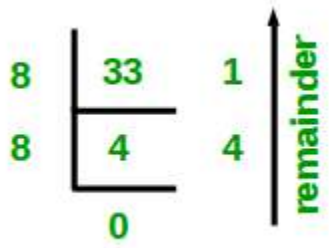
1. Divide the decimal number repeatedly by **8**.
2. Record the remainders.
3. Read remainders from **bottom to top**.

For Fractional Part

1. Multiply the fractional part by **8**.
2. Note the integer part.
3. Repeat until fraction becomes zero or precision is reached.
4. Write integer parts in **same order**.

Example:

1. **Convert 33_{10} to octal**



Final Answer:

$33_{10} = 41_8$

2. Convert 83.75_{10} to octal

Final Answer:

$83.75_{10} = 123.6_8$

1.4.4. Decimal → Hexadecimal

Method: Repeated Division by 16

Procedure:

For Integer Part

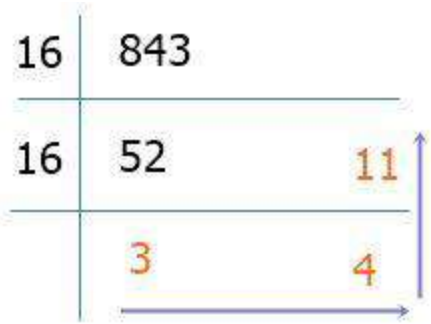
1. Divide the decimal number repeatedly by **16**.
2. Note the remainders.
3. Convert remainders **10–15** to **A–F**.
4. Read from bottom to top.

For Fractional Part

1. Multiply the fractional part by **16**.
2. Note the integer part (convert to A–F if needed).
3. Continue until fraction becomes zero or required precision.

Example:

1. Convert 843_{10} to hexadecimal



Final Answer:

$843_{10} = 34B_{16}$

2. Convert 26.5_{10} to hexadecimal

Final Answer:

$26.5_{10} = 1A.8_{16}$

1.4.5. Binary → Octal

Method: Grouping (3 bits)

Always start grouping from the RIGHT (LSB side), not from the left.

Why Group from the Right?

- The **rightmost bit is the LSB**
- Positional values are calculated from right to left
- Grouping from the left gives **wrong place values**

Procedure:

1. Separate integer and fractional parts.
2. Group 3 bits:
    - Integer part → group from right
    - Fractional part → group from left
3. Add leading or trailing zeros if needed.
4. Convert each 3-bit group to its octal equivalent.
5. Combine the results.

Example:

1. **Convert 101101_2 to octal**

Group from right:
101 101
←

$$101_2 = 5, 101_2 = 5$$

Final Answer:

$101101_2 = 55_8$

2. Convert 1100111_2 to octal

Group from right:

1 100 111

(Add leading zero if needed)

001 100 111

Convert:

- 001 → 1
- 100 → 4
- 111 → 7

Final Answer:

$1100111_2 = 147_8$

3. Convert 1101.101_2 to octal

Step 1: Group integer part (3 bits from right)

001 101

←

001 = 1, 101 = 5

Step 2: Group fractional part (3 bits from left)

.101

→

101 = 5

Final Answer:

$1101.101_2 = 15.5_8$

1.4.6. Binary → Hexadecimal

Method: Grouping (4 bits)

Procedure:

1. Separate integer and fractional parts.
2. Group **4 bits**:
    - Integer part → from **right**

- Fractional part → from **left**

3. Add zeros if necessary.
4. Convert each 4-bit group to hexadecimal digit.
5. Write the final number.

Example:

1. Convert 11101010_2 to hexadecimal

Group from right:
1110 1010
←

$$1110_2 = E, 1010_2 = A$$

Final Answer:

$11101010_2 = EA_{16}$

2. Convert 1110.1011_2 to hexadecimal

Integer part (4 bits):

1110 → E
←
Fractional part (4 bits):

.1011 → B
→
Final Answer:

$1110.1011_2 = E.B_{16}$

1.4.7. Octal → Binary

Rule:

Each octal digit is replaced by **3-bit binary**

Procedure:

1. Replace each octal digit with its **3-bit binary equivalent**.
2. Convert digits **before and after radix point separately**.
3. Combine all bits in the same order.

Example:

1. Convert 57_8 to binary

Octal	Binary
5	101
7	111

Final Answer:

$57_8 = 101111_2$

2. Convert 12.6_8 to binary

Octal	Binary
1	001
2	010
.	.
6	110

Final Answer:

$12.6_8 = 001010.110_2$

1.4.8. Hexadecimal → Binary

Rule:

Each hexadecimal digit is replaced by **4-bit binary**

Procedure:

1. Replace each hexadecimal digit with its 4-bit binary equivalent.
2. Convert digits on both sides of radix point.
3. Combine the binary digits to get final answer.

Example:

1. Convert $9C_{16}$ to binary

Hex	Binary
9	1001
C	1100

Final Answer:

$9C_{16} = 10011100_2$

2. Convert $B.A_{16}$ to binary

Hex	Binary
B	1011
.	.
A	1010

Final Answer:

$B.A_{16} = 1011.1010_2$

1.4.9. Octal → Decimal

Method: Positional Value

Procedure:

1. Assign powers of 8 to each digit.
2. Integer part → **8^0, 8^1, 8^2…** from right.
3. Fractional part → **8^{-1}, 8^{-2}…** from left.
4. Multiply each digit by its positional value.
5. Add all results to obtain decimal value.

Example:

1. Convert 246_8 to decimal

$$= (2 \times 8^2) + (4 \times 8^1) + (6 \times 8^0)$$
$$= 128 + 32 + 6 = 166$$

Final Answer:

$246_8 = 166_{10}$

2. Convert 25.4_8 to decimal

$$= (2 \times 8^1) + (5 \times 8^0) + (4 \times 8^{-1})$$
$$= 16 + 5 + 0.5 = 21.5$$

Final Answer:

$25.4_8 = 21.5_{10}$

1.4.10. Hexadecimal → Decimal

Method: Positional Value

Procedure:

1. Replace letters A–F with values **10–15**.
2. Assign powers of **16** to each digit.
3. Integer part → **16^0, 16^1, 16^2…**
4. Fractional part → **16^{-1}, 16^{-2}…**
5. Multiply each digit with its power and add all values.

Example:

Convert **$B4_{16}$** to decimal

$$= (11 \times 16^1) + (4 \times 16^0)$$
$$= 176 + 4 = 180$$

$B4_{16} = 180_{10}$

2. Convert $A.F_{16}$ to decimal

$$(10 \times 16^0) + (15 \times 16^{-1})$$

$$= 10 + 0.9375 = 10.9375$$

$A.F_{16} = 10.9375_{10}$

Shortcut (Must remember)

Conversion	Rule
Binary → Octal	Group 3 bits
Binary → Hex	Group 4 bits
Octal → Binary	Each digit → 3 bits
Hex → Binary	Each digit → 4 bits

✓ Integer grouping → Right to Left
✓ Fraction grouping → Left to Right
✓ Add zeros if grouping is incomplete
✓ Always mention base as subscript

1.5. Binary Codes

A binary code is a system of representing **numbers, letters, or symbols** using binary digits (0 and 1). Binary codes are widely used in digital computers, communication systems, and electronic circuits.

1.5.1. Classification of Codes

Binary codes are mainly classified into:

1. Weighted Codes
2. Non - Weighted Codes
3. Alphanumeric Binary Codes
4. Error Detecting and Correcting Codes

1.5.1.1. Weighted Codes

The type of binary code in which the **value of the digit depends on its position** in the number is called a **weighted code**. Hence, in weighted codes, **each position in a binary number has a specific weight** associated with it.

Characteristics:

- Each bit has a **positional weight**
- Easy to perform **arithmetic operations**
- Decimal value can be calculated directly
- Commonly used in **digital systems**

Example:

Consider the binary number **1001**

Bit Position	8	4	2	1
Binary Digit	1	0	0	1

Value = (1×8) + (0×4) + (0×2) + (1×1)
= 8 + 1 = 9

Hence, the positional weight of the **LSB** is **1**, and the positional weight of the **MSB** is **8**.

Examples of Weighted Codes:

(a) 8421 BCD Code

BCD (Binary Coded Decimal) is a weighted binary code used to represent **decimal digits (0–9)** in binary form. In BCD code, **each decimal digit is represented separately using 4 bits**.

Weights: **8, 4, 2, 1**

BCD (8421) Code Table

Decimal	BCD (8421)
0	0000
1	0001
2	0010
3	0011
4	0100
5	0101
6	0110
7	0111
8	1000
9	1001

Example:
1. Find the **BCD equivalent of 59**

- 5 → 0101
- 9 → 1001

BCD of 59 = 0101 1001

2. Find the decimal value of BCD **0100 0011**

- 0100 → 4
- 0011 → 3

Decimal number = 43

Important Points of BCD Code

✓ It is a weighted code
✓ Easy conversion between decimal and BCD

✓ Used in digital displays, calculators, and clocks

✕ Requires more bits compared to pure binary representation

(b) 2421 Code

The **2421 code** is a **weighted binary code**.
Its positional weights are **2, 4, 2 and 1**.

A decimal digit is represented using **4 bits**, and the **sum of the weights** is:

$$2 + 4 + 2 + 1 = 9$$

Hence, the **2421 code represents decimal digits from 0 to 9**.
It is also a **self-complementing code**.

Weights: **2, 4, 2, 1**

Decimal Digit	2421 Code
0	0000
1	0001
2	0010
3	0011
4	0100
5	1011
6	1100
7	1101
8	1110
9	1111

Example

Find the **2421 code of decimal 6**

1100 → 2 + 4 = 6

2421 code of 6 = 1100

Advantages:

✓ Simple to understand

✓ Suitable for arithmetic calculations

Disadvantages:

✕ Not efficient for error reduction

c) 5211 Code

The **5211 code** is also a **weighted binary code**.
Its positional weights are **5, 2, 1 and 1**.

A decimal digit is represented using **4 bits**, and the **sum of the weights** is:

$$5 + 2 + 1 + 1 = 9$$

Hence, the **5211 code represents decimal digits from 0 to 9**.
It is also a **self-complementing code**.

5211 Code Table

Decimal Digit	5211 Code
0	0000
1	0001
2	0011
3	0100
4	0101
5	0111
6	1000
7	1001
8	1011
9	1100

Example

Find the **5211 code of decimal 7**

1001 → 5 + 1 + 1 = 7

5211 code of 7 = 1001

1.5.1.2. Non-Weighted Codes

The type of binary code in which the **value of the digit does NOT depend on its position** in the number is called a **non-weighted code**.
In non-weighted codes, **no fixed positional weights** are assigned to the bit positions.

These codes are mainly used to **reduce errors**, **simplify circuit design**, and **improve reliability** rather than for arithmetic calculations.

Key Points:

- Value cannot be calculated directly
- Used for error reduction
- Arithmetic is difficult

(a) Excess-3 Code

The **Excess-3 code** is a non-weighted binary code obtained by **adding 3 to the decimal digit** and then converting the result into binary.

It is also called a **self-complementing code**.

Procedure to Generate Excess-3 Code

1. Take the given decimal digit
2. Add **3** to it

3. Convert the result into **4-bit binary**

Excess-3 Code Table

Decimal Digit	Excess-3 Code
0	0011
1	0100
2	0101
3	0110
4	0111
5	1000
6	1001
7	1010
8	1011
9	1100

Example 1:

Find the Excess-3 code of **5**

5 + 3 = 8
Binary of 8 = 1000

✅ **Excess-3 code of 5 = 1000**

Example 2:

Convert Excess-3 code **1010** to decimal

$1010_2 = 10$
10 − 3 = **7**

✅ **Decimal number = 7**

Important Points of Excess-3 Code

✓ It is a **non-weighted code**
✓ It is **self-complementing**
✓ Used in **digital arithmetic circuits**
✗ Not suitable for direct arithmetic operations

(b) Gray Code

Gray code is a non-weighted binary code in which **only one-bit changes between successive numbers**. It is also known as the **Reflected Binary Code**.

This single-bit change makes Gray code very useful in:

- Digital circuits
- Error reduction
- Rotary encoders

- Analog-to-digital conversion

Why is it used?

Because only **one-bit changes at a time**, Gray code **reduces errors** during transitions. **Example**: Between numbers **3 (011)** and **4 (100)**, switching 3 bits can cause glitches. Using Gray code fixes this.

Property of Gray Code

✓ Two consecutive Gray codes differ by **only one bit**

✓ Reduces **transition errors** in digital systems

Decimal, Binary and Gray Code Table

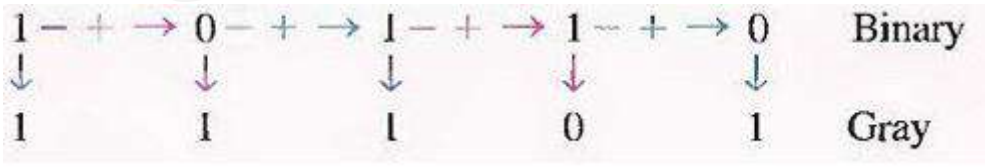
Decimal	Binary	Gray
0	0000	0000
1	0001	0001
2	0010	0011
3	0011	0010
4	0100	0110
5	0101	0111
6	0110	0101
7	0111	0100

Gray Code Conversion Procedures

Binary to Gray Code Conversion

Procedure

1. **Write the binary number.**
2. **Copy the MSB** (Most Significant Bit) directly as the first Gray bit.
3. For the remaining bits:
    - **XOR each binary bit with the bit before it**.
    - Formula:
      Gray[i] = Binary[i] XOR Binary[i-1]



Example

Convert **Binary 1011** to Gray:

Step	Binary Bit	Operation	Gray Bit
1	MSB = 1	Copy	1

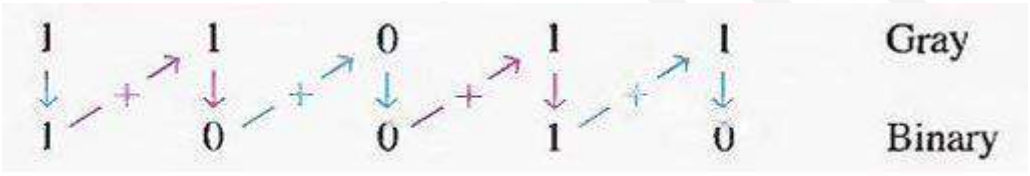
2	0	1 XOR 0 = 1	1
3	1	0 XOR 1 = 1	1
4	1	1 XOR 1 = 0	0

Gray Code = 1110

Gray Code to Binary Conversion

Procedure

1. **Write the Gray code.**
2. **Copy the MSB** of Gray as the first Binary bit.
3. For the remaining bits:
    - **XOR the previous Binary bit with the current Gray bit**
    - Formula:
      Binary[i] = Binary[i-1] XOR Gray[i]



Example

Convert **Gray 1110** to Binary:

Step	Gray Bit	Operation	Binary Bit
1	MSB = 1	Copy	1
2	1	1 XOR 1 = 0	0
3	1	0 XOR 1 = 1	1
4	0	1 XOR 0 = 1	1

Binary = 1011

Applications of Gray Code

✓ Shaft encoders
✓ Analog-to-Digital Converters (ADC)
✓ Error reduction in digital communication

Advantages of Non-Weighted Codes

✓ Reduce transition errors
✓ Useful in error-sensitive systems
✓ Improve reliability

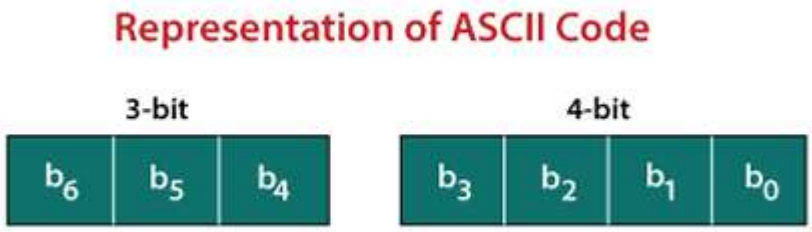
Disadvantages of Non-Weighted Codes

✕ Arithmetic operations are difficult

✕ Conversion to decimal is not direct

1.5.1.3. Alphanumeric Codes

- Alphanumeric codes represent **letters, numbers, and special symbols**.
- The ASCII stands for American Standard Code for Information Interchange.
- The ASCII code is an alphanumeric code used for data communication in digital computers.
- The ASCII is a 7-bit code capable of representing 2^7 or 128 number of different characters.
- The ASCII code is made up of a three-bit group, which is followed by a four-bit code.



- The ASCII Code is a 7 or 8-bit alphanumeric code.
- This code can represent 127 unique characters.
- The ASCII code starts from 00h to 7Fh. In this, the code from **00h to 1Fh** is used for **control characters,** and the code from **20h to 7Fh** is used for **graphic symbols**.
- The 8-bit code holds ASCII, which supports 256 symbols where math and graphic symbols are added.
- The range of the extended ASCII is 80h to FFh.

How Computers Use ASCII to Understand Human Text

When a user types text, each character is converted into its corresponding ASCII value.

Example: "Hello!"

Character	ASCII Value
H	72
e	101
l	108
l	108
o	111
!	33

Process Flow

1. Each character is converted into its **ASCII decimal value**
2. The ASCII value is converted into **binary (0s and 1s)**
3. Binary data is transmitted through **internet, cables, or wireless media**
4. The receiving system decodes the binary back into **ASCII values**
5. ASCII values are converted back into **readable characters**

Thus, the original message is displayed exactly as sent.

The ASCII characters are classified into the following groups:

1. CONTROL CHARACTERS

- The non-printable characters used for sending commands to the PC or printer are known as control characters.
- We can set tabs, and line breaks functionality by this code. The control characters are based on telex technology. Nowadays, it's not so much popular in use.
- The character from 0 to 31 and 127 comes under control characters.

Common Control Characters

ASCII Value	Name	Function
9	TAB	Horizontal tab
10	LF	Line Feed
13	CR	Carriage Return
127	DEL	Delete

2. SPECIAL CHARACTERS

- All printable characters that are neither numbers nor letters come under the special characters.
- These characters contain technical, punctuation, and mathematical characters with space also.
- The character from 32 to 47, 58 to 64, 91 to 96, and 123 to 126 comes under this category.

3. NUMBERS CHARACTERS

- This category of ASCII code contains Ten Arabic numerals from 0 to 9.

4. LETTERS CHARACTERS

- In this category, two groups of letters are contained, i.e., the group of uppercase letters and the group of lowercase letters.
- The range from 65 to 90 and 97 to 122 comes under this category.

Advantages of ASCII

✓ Simple and widely used
✓ Easy conversion between characters and binary
✓ Compatible with most computer systems
✓ Efficient for English text representation

Limitations of ASCII

✕ Limited to 128 characters
✕ Not suitable for multilingual text
✕ Extended ASCII lacks standardization

1.5.1.4. Error Detecting and Correcting Codes

We know that binary data consists of **0s and 1s**, which correspond to two different ranges of **analog voltages**.
During transmission of data from one system to another, **noise and interference** may get added to the signal.

Due to this noise:

- A bit **0 may change to 1**
- A bit **1 may change to 0**

Such changes lead to **errors** in the received data.
Although we cannot completely avoid noise, we can **detect** and sometimes **correct** these errors by using **error control codes**.

Classification of Error Control Codes

Error control codes are broadly classified into:

1. **Error Detection Codes**
2. **Error Correction Codes**

Error Detection Codes

Definition

Error detection codes are used to **detect whether an error has occurred** in the received data during transmission.

To achieve this:

- One or more **extra bits** are appended to the original data bits before transmission.
- These extra bits help the receiver to identify errors.

Important Points

- Can **detect errors**
- Cannot locate or correct the error
- Simple and easy to implement

Examples

- Parity Code
- Hamming Code (also used for correction)

Error Correction Codes

Definition

Error correction codes are used to **detect and correct errors** in the received data, so that the **original data** can be recovered.

Important Points

- Can **detect and correct errors**
- Require more additional bits than detection codes
- Slightly complex compared to parity codes

Example

- Hamming Code

Need for Additional Bits

To detect and correct errors:

- **Redundant bits (parity bits)** are appended to data bits
- These bits carry error-checking information
- Used by the receiver to verify correctness

Parity Code

Definition

Parity code is a **simple error detection code** in which **one parity bit** is added to the data bits.

The parity bit may be:

- Added to the **left of MSB**, or
- Added to the **right of LSB**

Types of Parity Codes

1. **Even Parity Code**
2. **Odd Parity Code**

Even Parity Code

Definition

In even parity:

- The parity bit is chosen such that the **total number of 1s becomes even**

Rule

- If number of 1s in data is **even → parity bit = 0**
- If number of 1s in data is **odd → parity bit = 1**

Even Parity Code Table (3-bit Data)

Binary Code	Even Parity Bit	Even Parity Code
000	0	0000
001	1	0011
010	1	0101
011	0	0110
100	1	1001
101	0	1010
110	0	1100
111	1	1111

Observations

- Total bits = **4**
- Possible even number of 1s = **0, 2, 4**

Limitation

- Can **detect errors**
- Cannot **correct errors**
- Cannot identify error position

Odd Parity Code

Definition

In odd parity:

- The parity bit is chosen such that the **total number of 1s becomes odd**

Rule

- If number of 1s in data is **odd → parity bit = 0**
- If number of 1s in data is **even → parity bit = 1**

Odd Parity Code Table (3-bit Data)

Binary Code	Odd Parity Bit	Odd Parity Code
000	1	0001
001	0	0010
010	0	0100
011	1	0111
100	0	1000
101	1	1011
110	1	1101
111	0	1110

Observations

- Total bits = **4**
- Possible odd number of 1s = **1, 3**

Limitation

- Detects error only
- Cannot correct error

Hamming Code

Definition

Hamming code is a powerful error control code used for **both error detection and error correction**.

It uses:

- **Multiple parity bits**
- Parity bits are placed at positions that are **powers of 2**

Number of Parity Bits Required

The number of parity bits **k** is calculated using:

$$2^k \geq n + k + 1$$

Where:

- **n** = number of data bits
- **k** = number of parity bits

Total bits in Hamming code = **n + k**

Placement of Parity Bits

Parity bits are placed at positions:

$$1,2,4,8,16, \ldots$$

Remaining positions are filled with data bits.

Parity Bit Calculation

Using **even parity** (commonly used):

- $\mathbf{p_1}$ → checks positions with binary LSB = 1
- $\mathbf{p_2}$ → checks positions with second LSB = 1
- $\mathbf{p_3}$ → checks positions with third LSB = 1

Parity bits are calculated using **XOR (⊕)** operation.

Example 1: Hamming Code Generation

Given:

Data bits:

$$d_4 d_3 d_2 d_1 = 1000$$

Step 1: Find parity bits

$$2^k \geq 4 + k + 1 \Rightarrow k = 3$$

Step 2: Arrange bits

Position	**7**	**6**	**5**	**4**	**3**	**2**	**1**
Bit	d_4	d_3	d_2	p_3	d_1	p_2	p_1

Substitute data bits:

$$100p_3 0p_2p_1$$

Step 3: Find parity bits

$$p_1 = b_7 \oplus b_5 \oplus b_3 = 1 \oplus 0 \oplus 0 = 1$$
$$p_2 = b_7 \oplus b_6 \oplus b_3 = 1 \oplus 0 \oplus 0 = 1$$
$$p_3 = b_7 \oplus b_6 \oplus b_5 = 1 \oplus 0 \oplus 0 = 1$$

Final Hamming Code

$$\boxed{1001011}$$

Example 2: Error Detection and Correction

Received Code:

$$1001111$$

Step 1: Calculate Check Bits

$$c_1 = b_7 \oplus b_5 \oplus b_3 \oplus b_1 = 1 \oplus 0 \oplus 1 \oplus 1 = 1$$
$$c_2 = b_7 \oplus b_6 \oplus b_3 \oplus b_2 = 1 \oplus 0 \oplus 1 \oplus 1 = 1$$
$$c_3 = b_7 \oplus b_6 \oplus b_5 \oplus b_4 = 1 \oplus 0 \oplus 0 \oplus 1 = 0$$

Step 2: Find Error Position

$$c_3c_2c_1 = (011)_2 = 3$$

👉 Error is at **bit position 3**

Step 3: Correct Error

- Complement bit 3
- Remove parity bits

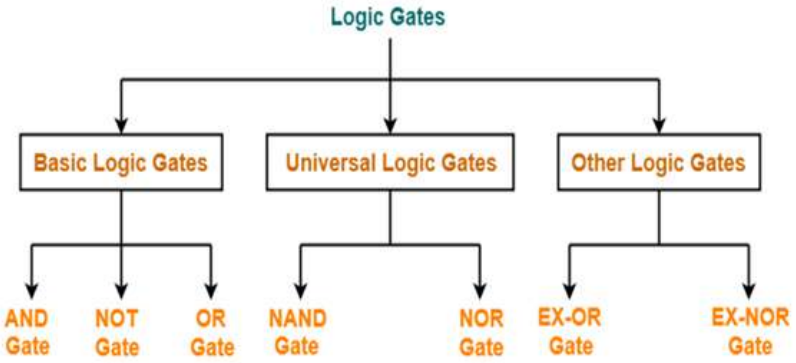
✅ Original data recovered successfully

1.16. Logic Gates:

- Logic Gates may be defined as Logic gates are the digital circuits capable of performing a particular logic function by operating on a number of binary inputs. **(OR)**
- Logic gates are the basic building blocks of any digital circuit.

Types Of Logic Gates:

Logic gates can be broadly classified as



1.16.1. Basic Logic Gates:

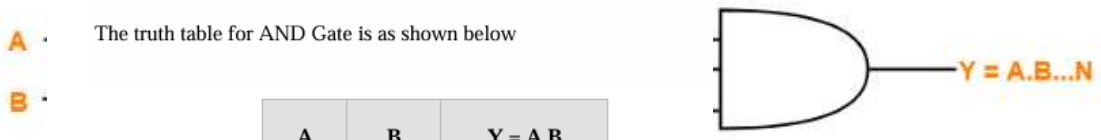
1. AND Gate

- The output of AND gate is high (‘1’) if all of its inputs are high (‘1’).
- The output of AND gate is low (‘0’) if any one of its inputs is low (‘0’).

The logic symbol for AND Gate is as shown below

Truth Table

The truth table for AND Gate is as shown below



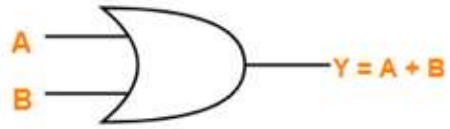
N-Input AND Gate

A	B	Y = A.B
0	0	0
0	1	0
1	0	0
1	1	1

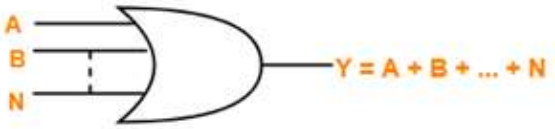
2. OR Gate

- The output of OR gate is high (‘1’) if any one of its inputs is high (‘1’).
- The output of OR gate is low (‘0’) if all of its inputs are low (‘0’).

The logic symbol for OR Gate is as shown below



2-Input OR Gate



N-Input OR Gate

Truth Table

The truth table for OR Gate is as shown below

A	B	Y = A + B
0	0	0
0	1	1
1	0	1
1	1	1

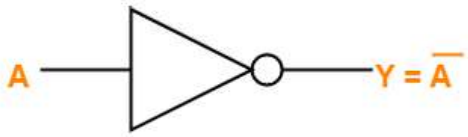
3. NOT Gate

- The output of NOT gate is high (‘1’) if its input is low (‘0’).
- The output of NOT gate is low (‘0’) if its input is high (‘1’).

From here

- It is clear that NOT gate simply inverts the given input.
- Since NOT gate simply inverts the given input, therefore it is also known as **Inverter Gate.**

The logic symbol for NOT Gate is as shown below



NOT Gate

Truth Table

The truth table for NOT Gate is as shown below

A	Y = A'
0	1
1	0

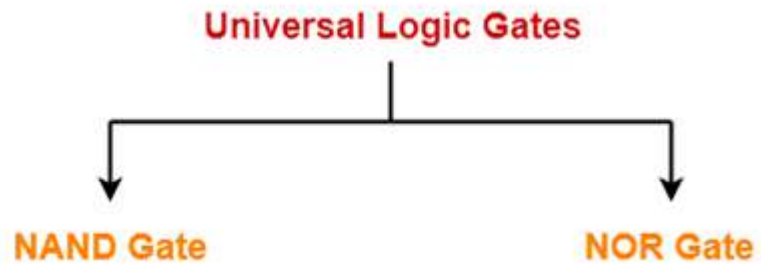
1.16.2. Universal Logic Gates:

Universal logic gates are the logic gates that are capable of implementing any Boolean function without requiring any other type of gate.

They have the following properties-

- Universal gates are not associative in nature.
- Universal gates are commutative in nature.

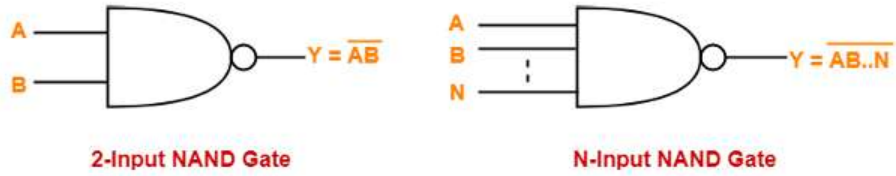
There are following two universal logic gates



1. NAND Gate

- A NAND Gate is constructed by connecting a NOT Gate at the output terminal of the AND Gate.
- The output of NAND gate is high ('1') if at least one of its inputs is low ('0').
- The output of NAND gate is low ('0') if all of its inputs are high ('1').

The logic symbol for NAND Gate is as shown below



2-Input NAND Gate N-Input NAND Gate

Truth Table

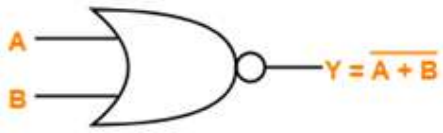
The truth table for NAND Gate is as shown below

A	B	Y = (A.B)' **OR** Y = $\overline{(A.B)}$
0	0	1
0	1	1
1	0	1
1	1	0

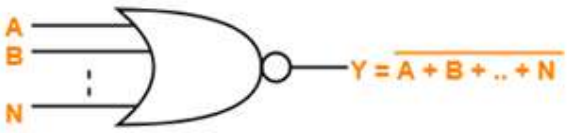
2. NOR Gate

- A NOR Gate is constructed by connecting a NOT Gate at the output terminal of the OR Gate.
- The output of OR gate is high (‘1’) if all of its inputs are low (‘0’).
- The output of OR gate is low (‘0’) if any of its inputs is high (‘1’).

The logic symbol for NOR Gate is as shown below



2-Input NOR Gate



N-Input NOR Gate

Truth Table

The truth table for NOR Gate is as shown below

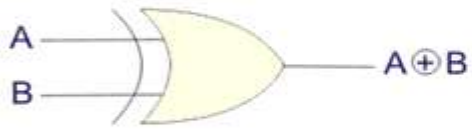
A	B	Y = (A+B)' OR Y = $\overline{(A+B)}$
0	0	1
0	1	0
1	0	0
1	1	0

1.16.3. OTHER LOGIC GATES:

1. XOR Gate

- The Exclusive-OR or 'Ex-OR' gate is a digital logic gate with more than two inputs and gives only one output.
- The output of XOR Gate is 'High' if either input is 'High'.
- The output is 'Low' if both the inputs are 'High' or if both the inputs are 'Low'.

The symbol and truth table of the XOR gate can be shown as:

<table>
<tr><th>Symbol</th><th colspan="3">Truth Table</th></tr>
<tr><td rowspan="5"> 

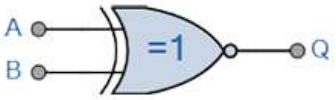
2-input Ex-OR Gate</td><td>A</td><td>B</td><td>Y</td></tr>
<tr><td>0</td><td>0</td><td>0</td></tr>
<tr><td>1</td><td>0</td><td>1</td></tr>
<tr><td>0</td><td>1</td><td>1</td></tr>
<tr><td>1</td><td>1</td><td>0</td></tr>
<tr><td>Boolean Expression Y = A ⊕ B</td><td colspan="3">A OR B but NOT BOTH gives Y</td></tr>
</table>

2.

XNOR Gate

- The Exclusive-NOR or 'EX-NOR' gate is a digital logic gate with more than two inputs and gives only one output.
- The output of XNOR Gate is 'High' if both the inputs are 'High' or if both the inputs are 'Low.'
- The output is 'Low' if either of the input is 'Low'.

The symbol and truth table of an XNOR gate can be given as:

Symbol	Truth Table		
 2-input Ex-NOR Gate	A	B	Y
	0	0	1
	0	1	0
	1	0	0
	1	1	1
Boolean Expression Y = $\overline{A \oplus B}$	If **A** AND **B** the **SAME** gives Y		

1.16.4. Universal Property of NAND and NOR Gates

- In digital electronics, **logic gates** are the basic building blocks of digital systems.
- Among all logic gates, **NAND** and **NOR** gates are called **Universal Gates** because **any Boolean function or logic circuit can be implemented using only NAND gates or only NOR gates**.
- This special capability is known as the **Universal Property** of NAND and NOR gates.
- A **universal gate** is a logic gate that can be used to **implement all basic logic gates** such as:
  - NOT
  - AND
  - OR
- and hence **any digital logic circuit**. **NAND and NOR gates are universal gates.**

Universal Property of NAND Gate

NAND Gate

A NAND gate is the **combination of AND gate followed by a NOT gate**.
Boolean Expression

$$Y = (A \cdot B)'$$

1. NOT Gate Using NAND Gate

- Connect **both inputs together** (A = B)

Operation

$$Y = (A \cdot A)' = A'$$

Hence, NAND gate acts as a **NOT gate**

2. AND Gate Using NAND Gates

Step 1: First NAND gate

$$Y_1 = (A \cdot B)'$$

Step 2: Second NAND gate (as NOT)

$$Y = (Y_1 \cdot Y_1)' = ((A \cdot B)')' = A \cdot B$$

AND gate is implemented using **two NAND gates**

3. OR Gate Using NAND Gates

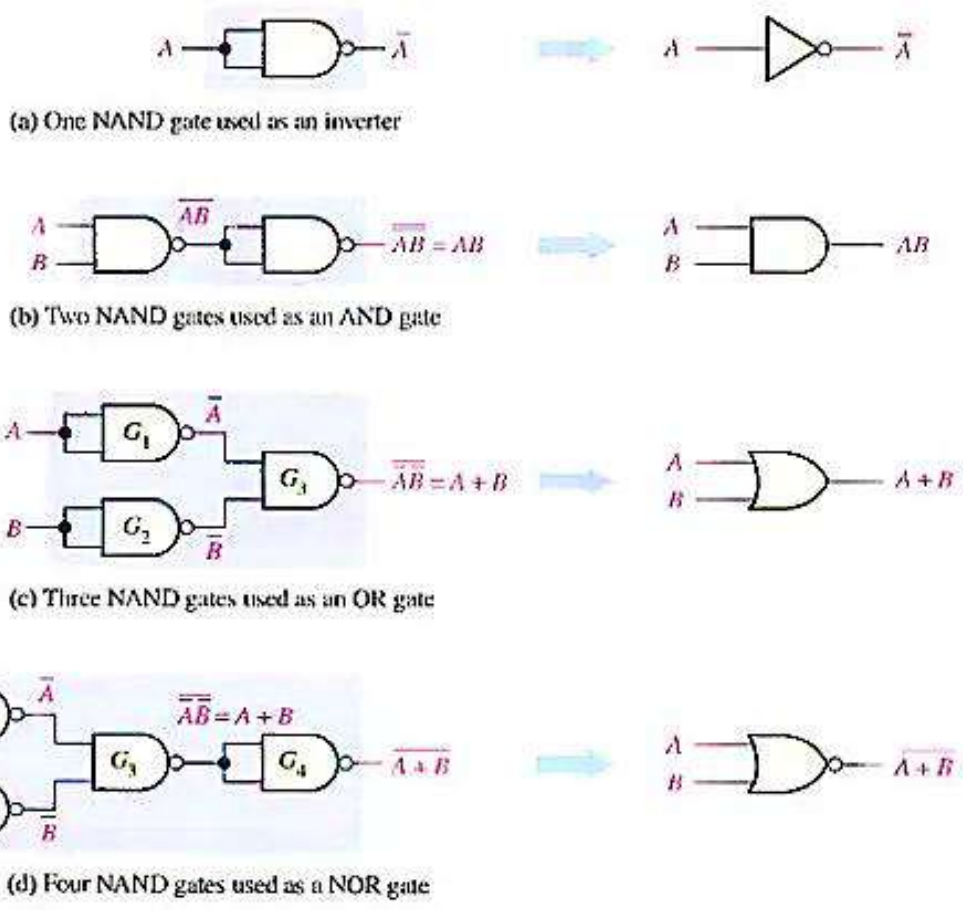
Using **De Morgan's Theorem**:

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

Implementation

- Use two NAND gates as NOT gates to get $A'$and B'
- Use one NAND gate to combine them

👉 OR gate is implemented using **three NAND gates**



(a) One NAND gate used as an inverter

(b) Two NAND gates used as an AND gate

(c) Three NAND gates used as an OR gate

(d) Four NAND gates used as a NOR gate

Universal Property of NOR Gate

NOR Gate

A NOR gate is the **combination of OR gate followed by a NOT gate**.

Boolean Expression

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

1. NOT Gate Using NOR Gate

- Connect **both inputs together** (A = B)

Operation

$$Y = (A + A)' = A'$$

NOR gate acts as a **NOT gate**

2. OR Gate Using NOR Gates

Step 1: First NOR gate

$$Y_1 = (A + B)'$$

Step 2: Second NOR gate (as NOT)

$$Y = (Y_1 + Y_1)' = ((A + B)')' = A + B$$

OR gate is implemented using **two NOR gates**

3. AND Gate Using NOR Gates

Using **De Morgan's Theorem**:

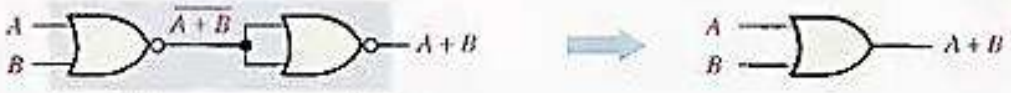
$$A \cdot B = (A' + B')'$$

Implementation

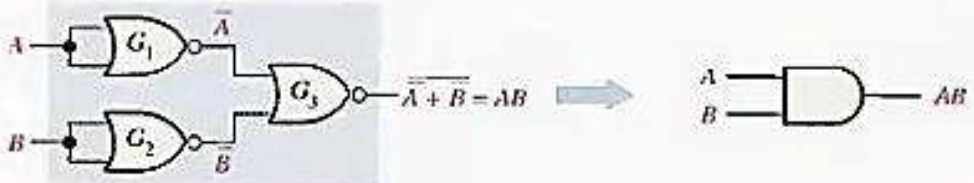
- Use two NOR gates as NOT gates to get $A'$and B'
- Use one NOR gate to combine them

AND gate is implemented using **three NOR gates**

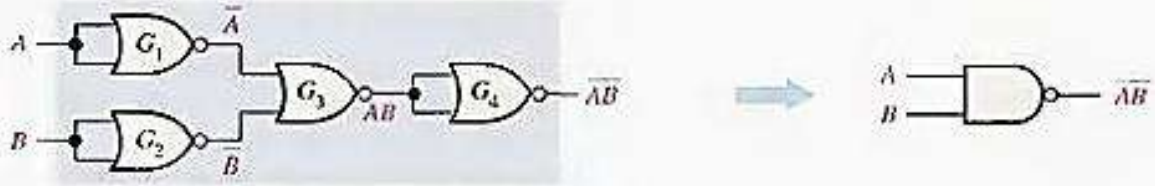
(a) One NOR gate used as an inverter



(b) Two NOR gates used as an OR gate



(c) Three NOR gates used as an AND gate



(d) Four NOR gates used as a NAND gate