Digital Systems

Chapter-1

Digital systems are assemblies of components-primarily electronic hardware and software-designed to generate, process, store, and transmit information in digital (discrete) form, most commonly as binary data (0s and 1s). Unlike analog systems, which handle continuous signals, digital systems work with signals that have a finite set of discrete values, enabling more reliable and accurate data manipulation.

Key Principles and Components

Binary Representation:

At the core of digital systems is binary logic, where information is encoded as sequences of 0s and 1s. This binary approach underpins all data storage, processing, and communication in digital devices.

Hardware:

- *Microprocessors and Integrated Circuits:* These are the "brains" of digital systems, executing instructions and performing calculations.
- *Memory (RAM/ROM):* Stores data temporarily or permanently.
- *Input/Output Devices:* Enable interaction with the external world (e.g., keyboards, screens, sensors).
- *Logic Circuits and Gates:* Implement basic logical operations (AND, OR, NOT) using Boolean algebra.

Software:

- *Firmware:* Low-level code embedded in hardware.
- *Operating Systems and Applications:* Manage resources and provide user interfaces.
- *Programming Languages:* Used to create software for digital systems[2].

How Digital Systems Work

Digital systems process information through a series of steps:

- **Input:** Data from the real world (often analog) is converted into digital signals using analog-to-digital converters (ADC).
- **Processing:** The system manipulates digital data using microprocessors and logic circuits.
- **Storage:** Data is held in registers or memory modules.
- **Output:** Results are presented to users or used to control other systems, sometimes converted back to analog using digital-to-analog converters (DAC)

Types and Applications

Digital systems are pervasive and can be categorized as follows:

Type	**Examples & Applications**
General Purpose Systems	Personal computers, servers, smartphones
Embedded Systems	Automotive controls, smart appliances, medical devices
Digital Signal Processing	Audio equipment, radar, telecommunications
Industrial Control	PLCs, SCADA systems, industrial robots
Real-Time Systems	Air traffic control, ABS in vehicles, nuclear plant control

Each type is tailored for specific functions but shares the fundamental principles of digital logic, modular design, and hierarchical implementation.

Advantages of Digital Systems

- **Accuracy and Reliability:** Less susceptible to noise and interference than analog systems.
- **Ease of Design and Mass Production:** Modular and hierarchical design simplifies construction and testing.
- **Data Storage and Transmission:** Digital data can be stored and transmitted without significant degradation.
- **Flexibility:** Can be reprogrammed or updated via software for new tasks.

1.2 Binary number system Hexadecimal and octal number systems.

Digital systems use several different number systems for representing and processing data. The most common are the binary, hexadecimal, and octal number systems. Here's an overview and comparison:

Binary Number System

- Base: 2
- Digits Used: 0, 1
- How It Works: Each digit (bit) represents a power of 2, starting from the rightmost digit (least significant bit). For example, the binary number 1011 equals $(1\times2^3)+(0\times2^2)+(1\times2^1)+(1\times2^0)=8+0+2+1=11$
- Usage: Fundamental to computers and digital electronics because digital circuits have two states (on/off, true/false), which map directly to 0 and 1.
- Example:
  - Binary: 1101
  - Decimal Equivalent: $(1\times2^3)+(1\times2^2)+(0\times2^1)+(1\times2^0)=8+4+0+1=13$

Hexadecimal Number System

- Base: 16
- Digits Used: 0–9 and A–F (A=10, B=11, C=12, D=13, E=14, F=15).

- How It Works: Each digit represents a power of 16. For example, the hexadecimal number 2F equals $(2\times16^1)+(15\times16^0)=32+15=47$
- Usage: Widely used in computing to represent binary numbers more compactly (since one hex digit equals four binary digits), for memory addresses, color codes in web design, etc.
- Example:
    - Hexadecimal: 1A3
    - Decimal Equivalent: $(1\times16^2)+(10\times16^1)+(3\times16^0)=256+160+3=419$
    - **Octal Number System**
- Base: 8
- Digits Used: 0–7.
- How It Works: Each digit represents a power of 8. For example, the octal number 157 equals $(1\times8^2)+(5\times8^1)+(7\times8^0)=64+40+7=111$
- Usage: Used in some computing contexts, especially older systems, to shorten binary representations (one octal digit equals three binary digits)
- Example:
    - Octal: 245
    - Decimal Equivalent: $(2\times8^2)+(4\times8^1)+(5\times8^0)=128+32+5=165$

Comparison Table

Number System	**Base**	**Digits Used**	**Example**	**Decimal Equivalent**	**Binary Grouping**
Binary	**2**	**0, 1**	**1011**	**11**	**1 bit**
Octal	**8**	**0–7**	**157**	**111**	**3 bits**
Hexadecimal	**16**	**0–9, A–F**	**2F**	**47**	**4 bits**
Decimal	**10**	**0-9**	**55**	**55**	

Key Points

- Binary is the foundation of digital systems, representing data with only two symbols.
- Hexadecimal is favored for its concise representation of binary data and is commonly used in programming and digital electronics.
- Octal also simplifies binary representation but is less common today, though still used in some legacy systems and applications.

1.3 Number system conversion

1.4 Binary codes weighted and non-weighted codes

Binary codes

Binary codes are ways of representing numbers, characters, or instructions using binary digits (0s and 1s). These codes are used in digital systems, computers, and electronics. Binary codes can be broadly categorized into **weighted codes** and **non-weighted codes**, based on whether each bit position carries a fixed weight.

1. Weighted Codes

In **weighted binary codes**, each digit position (bit) has a specific fixed weight, and the value of the code is calculated by summing the weights of the bits that are set to 1.

Key Characteristics:

- Each bit position has a predefined weight.
- Used for numerical data representation.
- Easy conversion between binary and decimal.

Common Weighted Codes:

1. **Binary Number System** (Natural Binary)
    - Weights: 8 4 2 1 (for 4-bit)
    - Example: 1010 = (1×8) + (0×4) + (1×2) + (0×1) = **10**
2. **BCD (Binary Coded Decimal)**
    - Each decimal digit (0–9) is represented using a 4-bit binary number.
    - Example: Decimal 25 = BCD 0010 0101
    - Weights: 8 4 2 1 (for each 4-bit group)

Advantages:

- Easy arithmetic operations.
- Simple to implement.

2. Non-Weighted Codes

In **non-weighted binary codes**, no specific weight is assigned to each digit position. The value does not depend on positional weights, and these codes are often used for purposes other than representing numbers directly.

Key Characteristics:

- No positional weights.
- Mainly used for error detection/correction or logical operations.
- May not follow simple binary-to-decimal conversion.

Common Non-Weighted Codes:

1. **Gray Code**
   - Only one-bit changes at a time between successive numbers (minimizes errors).
   - Example: Binary 011 = Gray 010
2. **Excess-3 Code**
   - Decimal digit + 3, then converted to binary.
   - Example: Decimal 2 → 2+3=5 → Binary 0101
3. **ASCII Code** (American Standard Code for Information Interchange)
   - Used to represent characters.
   - Example: 'A' = 01000001

Advantages:

- Useful in analog to digital conversion (Gray Code).
- Helps in error correction and data transmission (e.g., Parity, ASCII).

Comparison Table:

Feature	**Weighted Codes**	**Non-Weighted Codes**
Bit positions have weight	Yes	No
Used for	Numeric representation	Special purposes (errors, text)

Feature	Weighted Codes	Non-Weighted Codes
Example	Binary, BCD	Gray, Excess-3, ASCII
Arithmetic operations	Easy	Not always applicable

1.5 Alphanumeric Codes: ASCII and EBCDIC

Alphanumeric codes are **binary representations of letters, digits, and symbols**. These codes allow computers to store and transmit text data. Two of the most well-known alphanumeric codes are **ASCII** and **EBCDIC**.

1. ASCII (American Standard Code for Information Interchange)

Features:

- Developed in the 1960s.
- **7-bit code** (standard) → can represent **128 characters** (from 0 to 127).
- **8-bit extended ASCII →** supports **256 characters** (used in modern systems).
- Includes:
  - Control characters (0–31): e.g., NULL, Backspace, Line feed
  - Printable characters (32–126): e.g., letters, digits, punctuation
  - Extended characters (128–255 in 8-bit ASCII): e.g., graphical symbols, special characters

Examples:

Character	**ASCII Code (7-bit)**
A	1000001 (65)
a	1100001 (97)
0	0110000 (48)
Space	0100000 (32)

Advantages:

- Universally accepted.
- Simple and compact.
- Compatible with most systems.

---

2. EBCDIC (Extended Binary Coded Decimal Interchange Code)

Features:

- Developed by **IBM** for mainframe and midrange systems.
- **8-bit code** → represents **256 characters**.
- Different layout than ASCII.
- Less common in modern systems, mainly used in **IBM environments**.

Examples:

Character	EBCDIC Code (Hex)	Binary
A	C1	11000001
a	81	10000001
0	F0	11110000

Character	**EBCDIC Code (Hex)**	**Binary**
Space	40	01000000

Disadvantages:

- Not compatible with ASCII.
- Complex and less intuitive.
- Limited to IBM platforms.

Comparison Table:

Feature	**ASCII**	**EBCDIC**
Bits Used	7 (standard), 8 (extended)	8
Characters Supported	128 (standard), 256 (extended)	256
Developer	ANSI/ISO	IBM
Usage	Most modern systems	IBM mainframes
Simplicity	Simple and widely used	Complex and rarely used
Compatibility	Cross-platform	IBM-only

1.6 Representations of negative numbers

Representation of Negative Numbers in Digital Systems

Digital systems must represent both positive and negative numbers for arithmetic operations. Since binary (and other digital systems) naturally only represent positive values, special methods are used to encode negative numbers. The three most common methods are:

1. Sign and Magnitude

- **How it works:**
  The most significant bit (MSB) is used as a sign bit:
  - 0 indicates a positive number
  - 1 indicates a negative number
    The remaining bits represent the magnitude (absolute value) of the number.
- **Example (8-bit):**
  - +16: 00010000
  - -16: 10010000
- **Drawbacks:**
  - Two representations of zero (positive and negative zero)
  - Arithmetic operations are more complex for hardware.

2. One's Complement

- **How it works:**
  - Positive numbers are stored as usual.
  - To represent a negative number, invert (flip) all the bits of its positive equivalent.
- **Example (8-bit):**
  - +9: 00001001
  - -9: 11110110 (all bits of 9 flipped)
- **Drawbacks:**
  - Also results in two representations of zero.
  - Slightly easier for hardware to process than sign and magnitude, but still not ideal.

3. Two's Complement

- **How it works:**
    - Positive numbers are stored as usual.
    - To represent a negative number:
        1. Invert all bits of the positive number (find the one's complement).
        2. Add 1 to the result.
- **Example (8-bit):**
    - +12: 00001100
    - -12:
        - Invert bits: 11110011
        - Add 1: 11110100
- **Advantages:**
    - Only one representation of zero.
    - Arithmetic operations (addition, subtraction) are straightforward for hardware.
    - Most widely used method in modern digital systems and computers.

Comparison Table

Method	**Sign Bit**	**Negative Representation Example (-9, 8-bit)**
Sign & Magnitude	**Yes**	**10001001**

Method	**Sign Bit**	**Negative Representation Example (-9, 8-bit)**
One's Complement	**Yes**	**11110110**
Two's Complement	**Yes**	**11110111**

Summary

- Sign and Magnitude and One's Complement are easier for humans to interpret but create complications for hardware and have two zeros.
- Two's Complement is the standard in digital systems due to its unique zero representation and simplified arithmetic operations.
- In all methods, the MSB serves as the sign indicator: 0 for positive, 1 for negative.

1.7 subtraction using complements

Subtraction Using 1's Complement

Steps:

1. Check the given number is in binary or not
2. Check whether the bits are equal or not
3. Find the 1's complement of the subtrahend (the number to subtract).
4. Add it to the minuend (the number from which you subtract).
5. If there is a carry, add it to the least significant bit (end-around carry).
6. If there is no carry, take the 1's complement of the result and mark it negative.

Example:

Subtract 1001012100101(37 in decimal) from 1100102110010 (50 in decimal).

- **Step 1:** Find 1's complement of 100101:
    - 100101→011010
- **Step 2:** Add to minuend:
    - 110010+011010=1001100 (the leftmost '1' is a carry)
- **Step 3:** Add carry to result:
    - 001100+1=001101
- **Result:** 001101=13 (which is 50 - 37 = 13) ans

2. Subtraction Using 2's Complement

Steps:

1. Check the given number is in binary or not
2. Check whether the bits are equal or not

3. Find the 2's complement of the subtrahend (invert all bits and add 1).
4. Add it to the minuend.
5. If there is a carry, discard it (the result is positive).
6. If there is no carry, the result is negative and in 2's complement form.

Example:

Subtract 1001012100101 (37 in decimal) from 110010110010 (50 in decimal).

- **Step 1:** Find 2's complement of 100101
    - 1's complement: 011010
    - Add 1: 011010+1=011011
- **Step 2:** Add to minuend:

    - 110010+011011=1001101 (leftmost '1' is a carry)
- **Step 3:** Discard carry:
    - 001101=13 ans

If the result is negative (no carry), the answer is in 2's complement form and should be converted back to binary for interpretation.