REPUBLIQUE ALGERIENNE DEMOCRATIQUE ET POPULAIRE
الجمهورية الجزائرية الديمقراطية الشعبية

MINISTERE DE L'ENSEIGNEMENT SUPERIEUR
ET DE LA RECHERCHE SCIENTIFIQUE

ECOLE SUPERIEUREEN SCIENCES APPLIQUEES
--TLEMCEN--

وزارة التعليم العالي والبحث العلمي

المدرسة العليا في العلوم التطبيقية
-تلمسان-

Higher School of Applied Sciences of Tlemcen

First-Cycle Department

Course Handout

Computer Science 1

Prepared by:

Dr Mohammed M'HAMEDI

© Copyright by Dr. Mohammed M’HAMEDI, 2026

All Rights Reserved

PREFACE

This handout is for the detailed course intended for first-year engineering students in the core curriculum, to teach them the basic principles of computer architecture, Data representation and encoding, combinatorial logic and Boolean algebra, and algorithms, with an introduction to programming in the C language. The content of this handout is adapted to the preparatory class syllabus in science and technology.

This handout is organized into two parts, the first of which is devoted to number systems and coding, Boolean algebra, and logic gate circuits. The second part is designed to teach students how to design a programme using advanced structured development techniques in algorithmic and C programming. Each part includes an activity.

In order to ensure the smooth running of these assignments (both supervised and practical), students must attend their classes to cover all the prerequisites, which are based on basic computer science knowledge.

Finally, I hope that this handout will be of additional value and provide educational assistance to teachers in charge of tutorials or practical work, as well as to students in their computer science training.

List of Table

Table	Page
Table 2.2: Rules of Binary Addition	**20**
Table 4.1: Operators in Algorithms	**44**
Table 5.2: Assignment Operators in C	**58**
Table 5.3: printf Function Format Specifiers	**58**

List of Figure

Figure	Page
Figure 1.1: The first PC of IBM, The Macintosh PC	**3**
Figure 1.2: Computer	**3**
Figure 1.3:– Intel 80486DX2 (top, bottom)	**5**
Figure 1.5: PIC16F871 Microchip microcontroller	**10**
Figure 1.6: Arduino UNO card	**11**
Figure 4.1: Computer Software components	**39**
Figure 4.2: Programming Steps	**39**
Figure 4.2: Steps to Build an Algorithm Flowchart	**47**
Figure 4.3: Building an Algorithm model	**47**
Figure 5.1: The general structure of a C program	**54**

Table of Contents

INTRODUCTION 1

Chapter 1: Computer Science, Definition 3

I. What is a computer? 3

II. Operations of the computer 4

III. Computer architecture 4

III.1. Introduction 5
III.2 CPU 5
III.2.1 Von Neumann architecture 6
III.2.2. Harvard architecture 6
III.2.3.Main Memory 7
III.2.4. Input Output 7

IV. The microcontrollers 9

IV.1. The microcontroller PIC 10
IV.2. The Microcontroller card (arduino) 11
IV.3. The Microprocessor cards (Raspberry Pi) 12

Chapter 2: Data representation and encoding 13

I. Information Encoding Definition 13

II. The Number Bases 14

III. Encoding unsigned number (natural numbers) 16

III.1. Converting from base B to Decimal 16

III.2. Converting from Decimal to base B 17

III.3. Converting from base B to another base B' 18

IV. Binary arithmetic 20

IV.1. Binary Addition 20

IV.2. Binary Subtraction 21

V. Encoding signed numbers (integers) 22

V.1. Representation methods 22

V.2. Signed Magnitude Method (SM) 23

V.3. One's complement Method (1cp) 23

V.4. Two's complement Method (2cp) 24

VI. ASCII encoding 25

VII.Activity 26

Chapter 3: Boolean algebra and logic Gates 27

I. Introduction 27
II. Boolean algebra and logic functions 28
II.1. Logic operators 29
II.2. logical variable 30
II.3. Logic Functions 30
II.3.1. **Algebraic Expression** representation 30
II.3.2 Truth tables representation 31
II.3.3. **Logic Circuit** representation **(Gate Diagram)** 31
III. Postulates of boolean algebra 32
IV. Theorems of Boolean Algebra 32
V. Simplifying logic functions 33
V.1. Boolean Algebra method 34
V.1. Karnaugh Maps method (K-Maps) 35
VI. Logic Gates and circuits. 36
VI.1.VI.1. Simulation Tools 37
VII. Activity 37

Chapter 4: Programming Algorithm 38

I. Introduction
II. Algorithm Definition & Meaning 38
III. Variables in an algorithm 39
III.1. Variables Name 40
III.2. Variables Type 41
III.3. Variables designation 43
IV. Expressions and operators in an algorithm 44
V. Basic Instructions 45
VI. 1. Assignment 46
VI.2. Input Instructions 46
VI.3. Output Instructions 47
VI. Building an Algorithm (My First Algorithm) 47
VII. Conditional statements/control structures 48
VIII.1. if Statements 49
VIII.2. If-else Statement 49
VIII.3. else-if Statement 50
VIII.4. Switch Statement 50
VIII. Iteration statements/Loop structures 50
VIII.1. for loop 51
VIII.2. while loop 52
VIII.3. repeat loop 52
IX. Activity 53

Chapter 5: C programming language 54

I. Introduction 54
II. C library 54
III. C variables declaration 55
IV. Basic statements 56
V. My First c program 57
VI. Conditional statements (if..else) 58
VI.1. if Statements 59
VI.2. If-else Statement 59
VI.3. else-if Statement 60
VI.4. Switch Statement 60
VII. Iteration statement/Loop 61
VII.1. for loop 62
VII.2. while loop 63
VII.3. repeat loop 65
VII.4. Break and Continue statement 67
IX. Activity 69

CONCLUSION 70

BIBLIOGRAPHY 71

Activity Exercises Solution 72

INTRODUCTION

Computer Science has emerged as one of the most influential and transformative fields in the modern world, shaping industries, education, healthcare, communication, and nearly every aspect of daily life. At its core, it is the study of computers, information processing systems, algorithms, and programming languages, providing the tools and frameworks needed to solve complex problems, automate processes, and innovate across a wide range of disciplines.

The Computer Science 1 course is designed to offer students a thorough understanding of fundamental computing concepts, bridging both theoretical knowledge and practical skills. The initial chapters introduce the concept of a computer, detailing its essential operations, architecture, and key components such as the central processing unit (CPU), main memory, and input/output systems. The course also explores different computer architectures, including Von Neumann and Harvard models, and presents microcontrollers and microprocessor boards, such as PIC, Arduino, and Raspberry Pi.

A critical aspect of computing is the representation and encoding of data, which is addressed in the course through the study of number systems, unsigned and signed numbers, binary arithmetic, and encoding standards such as ASCII. Understanding these principles allows students to grasp how information is stored, processed, and transmitted within digital systems, forming the backbone of all computer operations.

Another fundamental component is Boolean algebra and logic gates, which underpin digital circuits and decision-making in computing systems. Students will learn how to represent logical functions using algebraic expressions, truth tables, and circuit diagrams, as well as methods to simplify these functions, including Boolean algebra laws and Karnaugh maps. This knowledge equips learners with the skills to design, analyze, and optimize logical circuits effectively.

Finally, the course introduces algorithm design and programming using the C language, enabling students to develop structured, efficient, and robust solutions to computational problems. Through hands-on exercises and programming projects, students will gain practical experience in translating theoretical concepts into functional programs, preparing them for more advanced studies in computer science.

Overall, Computer Science 1 provides a solid foundation in both the theory and practice of computing. It equips students with the essential knowledge and technical skills required to pursue advanced topics in programming, computer architecture, embedded systems, and digital logic design, fostering critical thinking, problem-solving, and computational reasoning that are vital in today's technology-driven world.

Chapter 1: Computer Science, Definition

I. What is a computer?

A computer is an electronic device that processes data according to a set of instructions called a program. It can perform a wide variety of tasks, such as calculations, data storage, and complex operations, by following logical and mathematical rules. A computer generally consists of several key components:

— **Central Processing Unit (CPU)**: The machine's brain, which performs instructions from software and processes data.

— **Memory (RAM)**: Temporarily stores data the CPU uses while performing tasks.

— **Storage**: Holds data permanently or semi-permanently, like a hard drive or solid-state drive (SSD).

— **Input Devices**: Hardware that allows users to interact with the computer, such as a keyboard or mouse.

— **Output Devices**: Devices like monitors or printers that display or output the results of the computer's processes.

Figure 1.1: The first PC of IBM, The Macintosh PC

It's essentially a programmable machine that can perform tasks automatically, following a set of instructions. Computers can be used for a wide variety of tasks, including

- **Communication:** Sending and receiving emails, video conferencing, and social networking.
- **Entertainment:** Playing games, watching movies, and listening to music.
- **Education:** Learning new skills, researching information, and completing assignments.
- **Work:** Creating documents, spreadsheets, and presentations, as well as managing databases and analyzing data.
- **Automation:** Controlling machines and systems, such as factories and traffic lights.

Figure 1.2: Computer

II. Operations of the computer

The operation of a computer involves several key steps that allow it to process data and execute tasks. Here's an overview of how a computer operates:

Input: The computer receives data or instructions from input devices like a keyboard, mouse, microphone, or touch screen. The input can also come from other sources, such as a network or file system.

Processing: Once the input is received, the **Central Processing Unit (CPU)**, which acts as the brain of the computer, processes the data. The CPU follows instructions from software programs that tell it how to manipulate the input data. The CPU performs four key tasks during processing, the CPU retrieves instructions from the computer's memory, it decodes the instructions to determine what actions need to be taken, it executes and performs the necessary calculations or logical operations and finally stores the results sends back to memory for later use or further processing.

Memory: During processing, the computer uses two types of storage, main memory and auxiliary storage**:** Devices like hard drives (HDD) or solid-state drives (SSD) store data persistently, even when the computer is powered off. This includes system files, applications, and user data.

Output: After processing the input, the computer sends the results to output devices such as a monitor, printer, or speaker. Output can be displayed as text, images, audio, or data files, depending on the task.

The operating system is a special program that manages the computer's resources, including the CPU, memory, and storage. It also provides an interface for users to interact with the computer. In summary, a computer's operation involves the seamless interaction of its input,

processing, memory, storage, and output components, controlled by its operating system to execute various programs and perform useful work.

III. Computer architecture

III.1. Introduction

Computer architecture refers to the conceptual design and fundamental operational structure of a computer system. It defines how the computer's hardware components interact with each other and how data flows between them. Computer architecture encompasses a wide range of elements, including the CPU design, memory organization, and input/output systems.

The Key Components of Computer Architecture are the Central Processing Unit (CPU), main Memory System, Input/Output (I/O) System and System Bus.

III.2. Central Processing Unit (CPU)

The CPU, or processor, is the brain of the computer that carries out instructions of a program by performing basic arithmetic, logical, control, and input/output (I/O) operations. The CPU is typically divided into three main parts:

- **Arithmetic Logic Unit (ALU)**: Handles all arithmetic and logical operations (e.g., addition, subtraction, comparison).
- **Control Unit (CU)**: Directs the operation of the processor, telling the ALU, memory, and input/output devices how to respond to instructions.
- **Registers**: Small, fast storage locations within the CPU used to hold temporary data or instructions during execution.

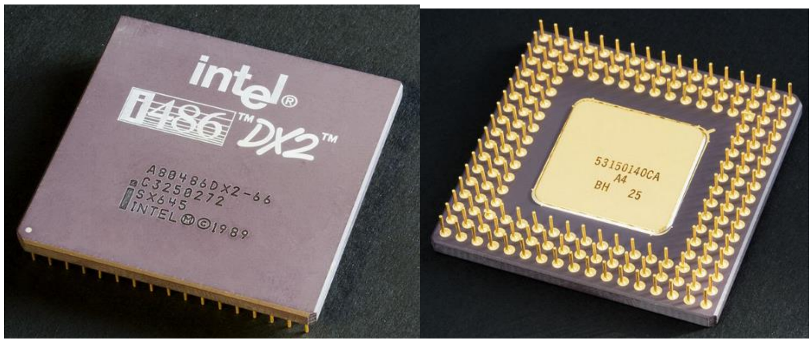


Figure 1.3: Intel 80486DX2 top, bottom

III.2.1 Von Neumann architecture

This is the traditional model of computer architecture where both data and instructions are stored in the same memory space. It follows the concept of a sequential execution model:

— Single path for data and instructions.
— Uses a single bus for data transfers.

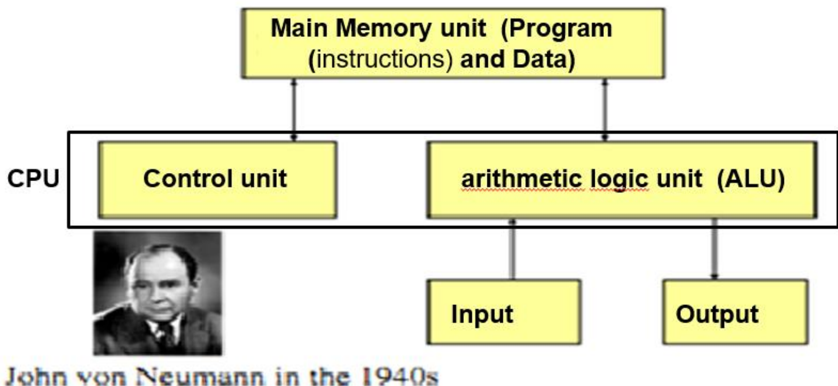


Figure 1.4: Von Neumann architecture

III.2.2. Harvard architecture

In this model, the computer has separate memory spaces for data and instructions. This allows for simultaneous access to instructions and data, improving speed and performance.

— Separate paths for data and instructions.
— Used in some modern microcontrollers and specialized systems.

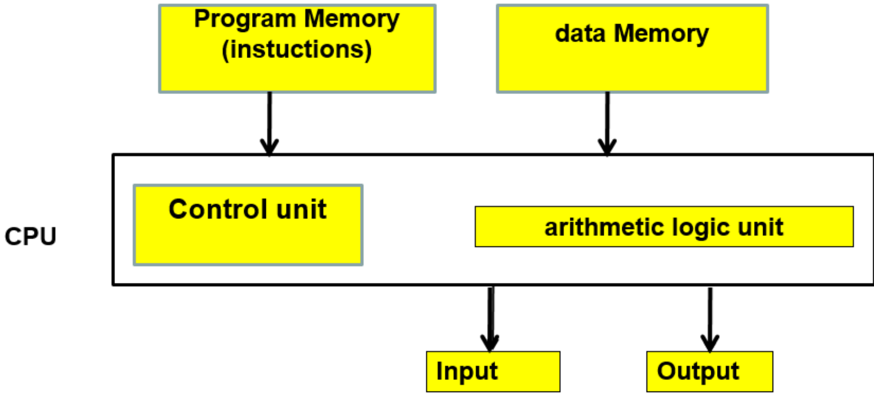


Figure 1.5: Harvard architecture

III.3. Main Memory

RAM (Random Access Memory) and **ROM (Read-Only Memory)** are two critical types of main memory in a computer, but they serve different purposes and have distinct characteristics. Let's explore each of them in detail:

A. RAM

RAM (Random Access Memory), also known as **primary memory**, is a crucial component of a computer system. It serves as the primary workspace for the CPU, storing data and instructions that the CPU needs to access and execute. The Key Characteristics of Main Memory are:

— **Volatile:** Main memory is volatile, meaning its contents are lost when the computer is turned off. This is in contrast to secondary storage devices like hard drives, which retain their data even when the power is off.

— **Random Access:** As the name suggests, any memory location can be accessed directly without having to go through other locations, making it very fast.

— **Speed:** Main memory is significantly faster than secondary storage devices, allowing the CPU to access data quickly.

— **Limited Capacity:** Main memory has a limited capacity compared to secondary storage. This is because it's designed for fast access rather than large storage.

Types of RAM:

— **DRAM (Dynamic Random Access Memory):** The most common type of main memory, using capacitors to store data. It requires periodic refreshing to maintain its contents.

— **SRAM (Static Random Access Memory):** A faster but more expensive type of main memory that uses flip-flops to store data. It doesn't require periodic refreshing.

The role of RAM in Computer Operation:

— **Storing Data and Instructions:** Main memory holds the data and instructions that the CPU needs to execute.

- **Acting as a Cache:** Some main memory can be used as a cache, storing frequently accessed data and instructions for faster retrieval.
- **Interacting with Secondary Storage:** Main memory is used to transfer data between the CPU and secondary storage devices like hard drives.

B. ROM

ROM is a type of non-volatile memory that retains its data even when the computer is turned off. Unlike RAM, ROM is primarily used to store firmware and system-level instructions that don't change often, such as the computer's BIOS.

Characteristics of ROM:

- **Non-volatility**: ROM retains its data even when the power is off, making it ideal for storing permanent instructions like firmware.
- **Pre-Programmed**: ROM is typically pre-programmed during manufacturing and cannot be easily modified. However, there are certain types of ROM that allow some modification.
- **Read-Only**: As the name suggests, the contents of ROM can generally only be read, not written to or modified by normal operations. It holds the instructions that are critical for booting up the computer.

Types of ROM:

- **PROM (Programmable ROM)**: This can be programmed once after manufacturing, but after that, the data cannot be changed.
- **EPROM (Erasable Programmable ROM)**: Can be erased and reprogrammed using UV light, making it more flexible than traditional ROM.
- **EEPROM (Electrically Erasable Programmable ROM)**: Can be erased and reprogrammed electrically, without needing UV light. It is used for firmware updates in modern devices.

Functions of ROM :

- **Boot Instructions**: ROM contains the **BIOS (Basic Input/Output System)** or **UEFI** firmware, which initializes the hardware and prepares the system to load the operating system when you turn on the computer.

— **Permanent Storage of System-Level Software**: ROM stores essential system software that should not be altered frequently, such as the firmware that controls hardware devices.

III.4. Input/Output

I/O systems are responsible for the communication between a computer and the external world. They enable the computer to receive input (e.g., from a keyboard or mouse) and send output (e.g., to a monitor or printer). I/O devices are essential for a computer to interact with the outside world, allowing data to be input (received) from devices like keyboards or sensors and output (delivered) to devices like displays or printers.

Components of the I/O System:

— I**nput Devices :** Input devices allow users to provide data or instructions to the computer, for examples, Keyboard, mouse, scanner, microphone, touchpad, sensors.

— **Output Devices:** Output devices allow the computer to send processed data to the user or other devices, for examples: Monitor, printer, speakers, projector, headphones.

— **I/O Controller:** The I/O controller acts as an interface between the CPU and the I/O devices. It manages data transfer between peripheral devices and the central system, ensuring that the CPU is not overloaded with I/O tasks.The controller interprets device commands, manages data flow, and handles interrupts when a device needs CPU attention.

— **Device Drivers:** A device driver is software that allows the operating system and applications to communicate with hardware devices. Each device (keyboard, printer, etc.) has its own driver that controls how it functions with the computer system.The operating OS uses drivers to manage hardware, sending instructions to I/O devices and interpreting input from those devices.

— **I/O Ports:** I/O ports are physical or virtual interfaces where data is transmitted between the computer and peripherals. Including physical ports, These include USB ports, HDMI ports, Ethernet ports, etc., where external devices can be plugged in. And virtual ports, as Software-defined ports that manage data flow in networking (e.g., TCP/IP ports).

The **Input/Output System** is responsible for managing how the computer interacts with external devices and the outside world. It involves input devices that send data to the computer and output devices that receive data from the computer.

IV.The microcontrollers

Microcontrollers are essentially tiny, self-contained computers on a single chip. They're designed to control specific functions within embedded systems, meaning they're often found in devices that don't require a full-fledged computer.

Key Components of a Microcontroller:

— **Processor:** The brain of the microcontroller, responsible for executing instructions.
— **Memory:** Stores instructions and data. This includes both ROM (read-only memory) and RAM (random access memory).
— **Input/Output (I/O) Peripherals:** These components allow the microcontroller to interact with the outside world, such as sensors, actuators, and communication devices.

Common Applications of Microcontrollers :

— **Automotive:** Engine control units, airbag systems, anti-lock brakes
— **Consumer Electronics:** Washing machines, refrigerators, microwave ovens, remote controls
— **Industrial Automation:** Robotics, factory control systems
— **Medical Devices:** Pacemakers, insulin pumps
— **IoT Devices:** Smart home appliances, wearable technolo

Popular Microcontroller Families :

— Arduino A popular platform for beginners and hobbyists, known for its ease of use and extensive community support.
— **Raspberry Pi:** While primarily a single-board computer, it can also be used as a microcontroller in certain applications.
— **PIC (Peripheral Interface Controller):** A family of microcontrollers from Microchip Technology, widely used in various industries.

— **AVR (Atmel RISC Architecture):** Another popular microcontroller family from Microchip

IV.1. PIC Microcontroller

PIC (Peripheral Interface Controller) microcontrollers are a popular family of devices produced by Microchip Technology. Known for their reliability, flexibility, and ease of programming, they're widely used in a variety of applications.

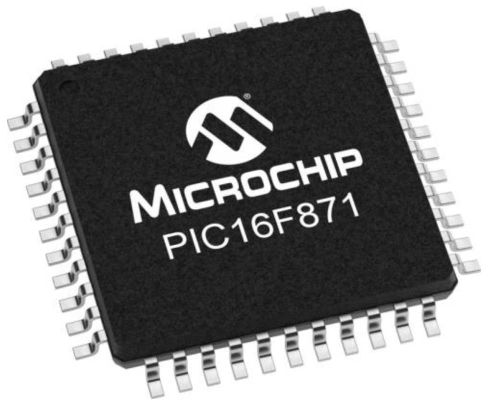


Figure 4.5: PIC16F871 Microchip microcontroller.

Common Applications of PIC:

— Home appliances (e.g., washing machines, microwaves)

— IoT devices

— Robotics

— Automotive control systems (e.g., engine control units, airbags)

— Remote controls

IV.2. Microcontroller card (arduino)

The **Arduino** is an open-source platform based on easy-to-use **hardware** and **software**, primarily designed for building electronics projects. It includes a **microcontroller card** (or board), often equipped with an **Atmel AVR** or **ARM-based microcontroller**, and a development environment for writing, compiling, and uploading code. Arduino is popular for beginners and hobbyists, as well as professionals, due to its simplicity and versatility.

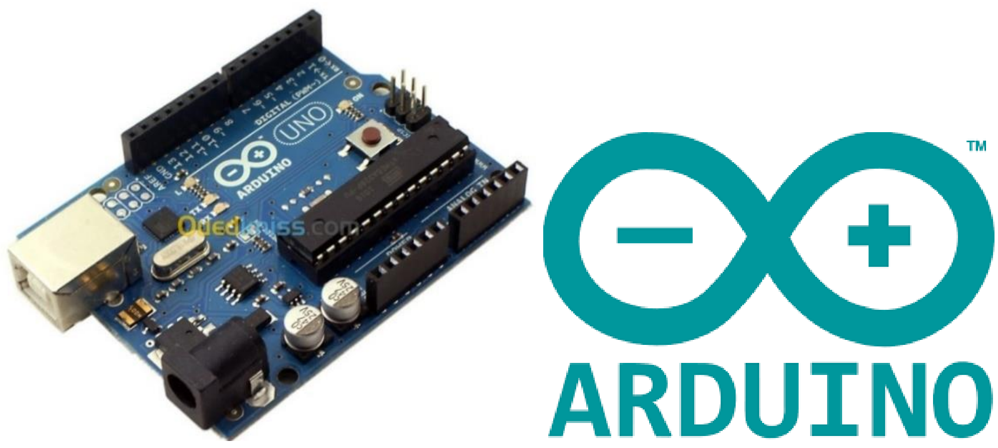


Figure 4.6: Arduino UNO card

Key Components of an Arduino Microcontroller Card:

— **Microcontroller**: The heart of the board, which processes the input and controls the output. For example, an Arduino Uno uses the **ATmega328P** microcontroller.
— **Digital I/O Pins**: Used to read inputs (like sensors) and control outputs (like LEDs or motors). Common boards have 14 digital pins.
— **Analog Input Pins**: Typically used for reading variable signals like temperature or light intensity. Most Arduino boards come with 6 analog pins.
— **Power Supply**: Arduino boards can be powered via USB or an external power source (DC adapter or battery).
— **Clock**: The clock speed defines how fast the microcontroller operates. The standard Arduino Uno runs at **16 MHz**.
— **Memory**: Divided into Flash (for storing code), SRAM (for runtime data), and EEPROM (for long-term data storage). The ATmega328P has **32 KB** of flash memory.
— **USB Port**: Used for programming the microcontroller and communication with a computer.

IV.3. Microprocessor cards (Raspberry Pi)

The **Raspberry Pi** is a series of low-cost, single-board computers developed by the Raspberry Pi Foundation. Unlike microcontroller-based boards like Arduino, Raspberry Pi uses a **microprocessor** and functions as a fully-fledged computer with an operating system, making it capable of more complex tasks such as running software, web browsing, and multimedia.

Key Features of Raspberry Pi :

- **Microprocessor**: Raspberry Pi boards are powered by a **Broadcom ARM-based microprocessor**. For instance, the Raspberry Pi 4 uses a **quad-core Cortex-A72 (ARM v8) 64-bit SoC** running at 1.5 GHz.
- **RAM**: Depending on the model, Raspberry Pi boards can come with various amounts of RAM (e.g., 2GB, 4GB, or 8GB on the Raspberry Pi 4).
- **Operating System**: Raspberry Pi typically runs a Linux-based OS called **Raspberry Pi OS** (formerly Raspbian), but it can also run other operating systems like Ubuntu, and even Windows IoT Core.
- **GPIO (General-Purpose Input/Output) Pins**: The board features a set of GPIO pins that allow it to interface with external components like sensors, motors, and LEDs.
- **Storage**: Raspberry Pi uses a **microSD card** for storage, where the operating system and all files are stored.
- **USB Ports**: It has multiple USB ports, including **USB 3.0** for faster data transfer on newer models.
- **HDMI**: Raspberry Pi includes **HDMI ports** for video output, allowing it to be connected to a monitor or TV. The Raspberry Pi 4 supports dual micro-HDMI output.
- **Networking**: Most Raspberry Pi models have built-in **Ethernet** and **Wi-Fi** for network connectivity, along with **Bluetooth**.

Figure 1.7: Raspberry Pi 3B card.

Chapter 2: Data representation and encoding

I. Information Encoding Definition

A **computer processes data** by performing a series of operations to transform raw input into meaningful output. Data processing involves inputting, storing, and manipulating data to produce results, such as calculations, images, text, or decisions.

Steps in Data Processing:

- **Input**: Data is entered into the computer system via input devices like a keyboard, mouse, sensor, or through files.
- **Processing**: The central processing unit (CPU) performs arithmetic and logical operations on the data according to the instructions provided by the software or program. This can involve: Sorting and filtering data, Performing calculations (e.g., addition, subtraction), and Running algorithms to analyze the data.
- **Storage**: During and after processing, the data can be stored temporarily in RAM or permanently in storage devices like hard drives, SSDs, or cloud storage.
- **Output**: The processed data is delivered as useful information. Output devices like monitors, printers, or speakers display the results, or data can be sent to another system.
- **Feedback/Control**: In certain cases, such as embedded systems or IoT devices, the computer controls other devices or systems based on the processed data (e.g., adjusting temperature, controlling motors).

There are various **types of data** that computers can process, depending on its structure and purpose. Data can be broadly categorized into different forms based on how it's represented and used in computation:

Numerical Data:

- **Integer**: Whole numbers without fractions (e.g., 10, -25, 500).
- **Float (Floating-point)**: Numbers with decimal points or in scientific notation (e.g., 3.14, -0.005, 1.5e3).
- **Double**: A type of floating-point with double precision, allowing for more decimal places.

Textual Data:

- **Character**: Single letters, numbers, or symbols (e.g., 'A', '5', '#').
- **String**: A sequence of characters (e.g., "Hello, World!", "12345"). Strings can represent words, sentences, or longer blocks of text.

Boolean Data:

Boolean (True/False): A type of data that represents logical values. It can have only two possible states: True or False, often used in decision-making operations (e.g., "Is the user logged in?").

Each type of data requires specific methods for storage, processing, and analysis, based on the computer's capabilities and the application's needs.

Different types of data (numbers, text, images, etc.) are encoded into binary so that computers can store and process them.

The Concepts in Binary Encoding are, **Bit (Binary Digit)**: The smallest unit of data in a computer, either **0** or **1**, **Byte**: A group of 8 bits. For example, 01001010 is 1 byte of data, **Binary Number System**: Unlike the decimal system, which uses 10 digits (0–9), the binary system only uses two digits (0 and 1). Binary numbers represent values in powers of 2.

— Binary Encoding for Numbers: Numbers can be represented directly in binary, where each digit is either 0 or 1. Each place in a binary number corresponds to a power of 2, starting from the rightmost bit.

For example, the decimal number 13 can be represented as a binary number:

13 in decimal = 1101 in binary

— Binary Encoding for Text (ASCII/UTF-8):

Text characters are converted into binary using encoding schemes like **ASCII** or **UTF-8**. Each character is assigned a specific binary value.

For example:

'A' in ASCII is 65 in decimal, which is 01000001 in binary.

— Binary Encoding for Images**:**

Images are encoded as a series of binary values representing pixel color values. Each pixel might be represented in binary based on color depth:

For example:

Black and White Images: Each pixel is either 0 (black) or 1 (white).

II. The Number Bases

Number bases refer to the different numeral systems that use various base values to represent numbers. The **base** of a numeral system specifies how many unique digits, including zero, are used in the system and how the digits' positions represent values.

The Common Number Bases:

Base 10 (Decimal):

- **Most common** numeral system in everyday use.
- Digits : 0, 1, 2, 3, 4, 5, 6, 7, 8, 9.

— Each position represents a power of 10.

Base 2 (Binary):

— Used by computers and digital systems because it represents data with only two symbols: 0 and 1.

— Digits: 0, 1.

— Each position represents a power of 2.

Base 8 (Octal):

— Often used in computing as a shorthand for binary.

— Digits: 0, 1, 2, 3, 4, 5, 6, 7.

— Each position represents a power of 8.

Base 16 (Hexadecimal):

— Widely used in computing as a compact representation of binary data.

— Digits: 0, 1, 2, 3, 4, 5, 6, 7, 8, 9, A, B, C, D, E, F (where A = 10, B = 11, ..., F = 15).

— Each position represents a power of 16.

Table 2.1: Number Base Representations (decimal numbers (0–15))

Decimal	Binary	Octal	Hexadecimal
0	0000	0	0
1	0001	1	1
2	0010	2	2
3	0011	3	3
4	0100	4	4
5	0101	5	5
6	0110	6	6
7	0111	7	7
8	1000	10	8
9	1001	11	9
10	1010	12	A
11	1011	13	B
12	1100	14	C
13	1101	15	D
14	1110	16	E
15	1111	17	F

III. Encoding unsigned number (natural numbers)

Unsigned numbers are non-negative integers (natural numbers: 0, 1, 2, 3, …). Unlike signed numbers, they do not use a sign bit; instead, the entire bit pattern is used to represent the magnitude.

In binary, an n-bit unsigned number can represent values in the range: $0 \leq N \leq 2^n - 1$.

Example:

- With **3 bits**, we can represent numbers from 0 to $2^3 - 1 = 7$.
- Values: 000 → 0, 001 → 1, …, 111 → 7.

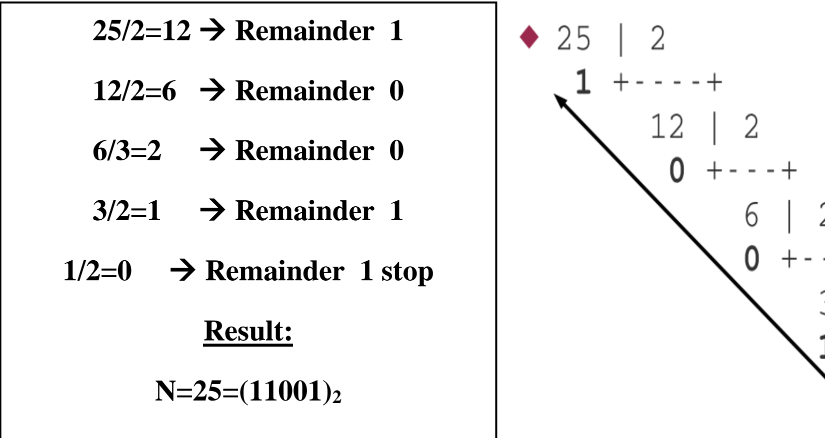
The unsigned numbers can be represented in base B, such as Binary (**base 2**), Octal (**base 8),** and Hexadecimal (**base 16**).

III.1. Converting from Decimal to base B

The process of converting a decimal number (base-10) into a different base (base-B) is a systematic mathematical procedure. Converting a decimal number to another (base-B) involves following a systematic, step-by-step procedure.

1. **Divided this number by B**
2. **Replace the original number with the result of the division and return any remainder.**
3. **Continue these steps until the original number becomes 0.**
4. **Read the recorded remainders, from right to left, to form the representation in base B.**

<u>Example 1: N=25, Convert to binary (B=2)</u>



25/2=12 → Remainder 1

12/2=6 → Remainder 0

6/3=2 → Remainder 0

3/2=1 → Remainder 1

1/2=0 → Remainder 1 stop

<u>Result:</u>

$N=25=(11001)_2$

Example 2: N=25, Convert to binary (B=8)

25/8=3 → Remainder 1

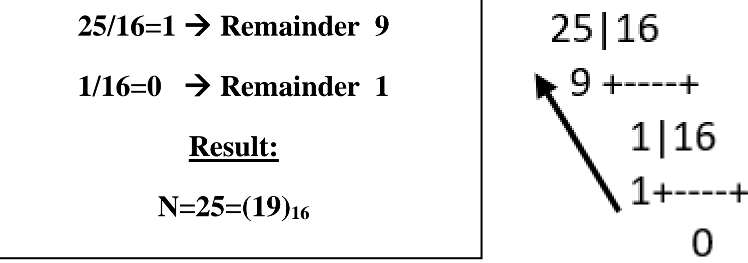
3/8=0 → Remainder 3

Result:

N=25=$(31)_8$

```
25|8
1 +----+
    3|8
    3+----+
        0
```

Example 2: N=25, Convert to binary (B=16)



III.2. Converting from base B to Decimal

The process of converting a different base (base-B) to the decimal number involves following a systematic, step-by-step procedure.

1. **Start with the rightmost digit in the base B number.**
2. **Assign a positional value of 0 to the rightmost digit, and then increase the positional value by 1 for each digit to the left.**
3. **Multiply each digit by B raised to the power of its positional value.**
4. **Sum up the results of these multiplications for all digits.**
5. **The final sum is the decimal equivalent of the base B number.**

$$a_nB^n + a_{n-1}B^{n-1} + \dots a_1B + a_0 \rightarrow \text{With } 0<a_i<B$$

Example 1: N=$(11001)_2$, Convert to Decimal

$(11001)_8 = 1 \times 2^4 + 1 \times 2^3 + 0 \times 2^2 + 0 \times 2^1 + 1 \times 2^0$

$= 16 + 8 + 0 + 0 + 1 = 25$

Result:

$(11001)_8 = 25$

Example 2: N=$(31)_8$, Convert to Decimal

$$(31)_8 = 3x\ 8^1 + 1x8^0$$
$$= 24+1 = 25$$
Result:
$$(31)_8 = 25$$

Example 2: N=$(19)_{16}$, Convert to Decimal

$$(19)_{16} = 1x\ 16^1 + 9x16^0$$
$$= 16+9 = 25$$
Result:
$$(19)_{16} = 25$$

III.3. Converting from base B to another base B’

The conversion of a numeral from a source Base B (such as 2, 8, or 16) to a target Base B' is commonly achieved through one of two approaches.

III.3.1. Method 1: Indirect Conversion via Decimal

This is the most common method because the algorithms for “base → decimal” and “decimal → base” are straightforward.

1. **Convert the given number from base** B **into its decimal equivalent.**
2. **Then, convert the obtained decimal number into the target base** B’ **using successive division by** B’ **(for integers) or multiplication by** B’ **(for fractional parts).**
3. **This method is widely used because decimal is the most familiar and straightforward base for calculations.**

Example: Convert $(237)_8$ to base 2

Step 1: $(237)_8=159$

Step 2: $159=(10011111)_2$

Result:

$$(237)_8=(10011111)_2$$

III.1.2. Method 2: Direct Conversion (Base-to-Base)

This approach avoids passing through decimal by using grouping techniques or mathematical manipulation.

It works best when the two bases are powers of each other (e.g., base 2 ↔ base 8, or base 2 ↔ base 16).

Digits are grouped and directly mapped between the two bases, making the process faster and less error-prone.

- **Binary ↔ Octal:** Group binary digits in **groups of 3**.
- **Binary ↔ Hexadecimal:** Group binary digits in **groups of 4**.

Example: Convert $(473)_8$ to base 2

Each octal digit corresponds to 3 binary digits:

4 → 100, 7 → 111, 3 → 011

So, $(473)_8 = (100111011)_2$.

Result:

$(473)_8 = (100111011)_2$.

IV. Binary arithmetic

Binary arithmetic is the foundation of all digital computer operations. Since computers use the **binary number system (base 2)**, arithmetic is performed using only two digits: **0 and 1**.

The basic operations in binary arithmetic are:

- **Addition**
- **Subtraction**

Computers use binary (base-2) because it's a system with only two digits, 0 and 1, which can be easily represented by electrical states (off/on, false/true, low voltage/high voltage). All complex calculations a computer performs boil down to these simple binary operations.

IV.1. Binary Addition

This is the most fundamental operation. The rules are much simpler than in decimal. Binary addition follows simple rules similar to decimal addition, but with base 2:

Table 2.2 : Rules of Binary Addition

A	B	Sum	Carry
0	0	0	0
0	1	1	0
1	0	1	0
1	1	0	1

Example: **$(1011)_2 + (110)_2$**

Carry: 1 1 1 (Carry row shows values carried over)

1 0 1 1 (11 in decimal)
+ 0 1 1 0 (6 in decimal)
----
1 0 0 0 1 (17 in decimal)

Step-by-step:
LSB (rightmost): 1 + 0 = 1 -> Write 1, Carry 0**.**
Next: 1 + 1 = 10 -> Write 0, Carry 1.
Next: 0 + 1 + 1 (carry) = 10 -> Write 0, Carry 1.
MSB: 1 + 0 + 1 (carry) = 10 -> Write 10. Since this is the **final column, the `10` is written down.**
Result

$(1011)_2 + (110)_2 = (10001)_2$

IV.2. Binary Subtraction

Subtraction can be done directly, but computers often use a method called **Two's Complement** to turn subtraction into addition, which is easier for circuits to handle. We'll cover the direct method first. Binary subtraction is similar to decimal subtraction, but it may require borrowing.

Table 2.3 : Rules of Binary Subtraction

A	B	Difference	Borrow
0	0	0	0
1	0	1	0
0	1	1	1
1	1	0	0

Example: $(1001)_2$ - $(110)_2$

Borrow: 0 1 1 (Borrow row shows values needed)
1 0 0 1 (9 in decimal)
- 0 1 1 0 (6 in decimal)
----
0 0 1 1 (3 in decimal)

Step-by-step:

LSB: 1 - 0 = 1 -> Write 1.
Next: 0 - 1 -> Can't do. Borrow from the left. The next left is 0, so we must borrow from the column after that (0 becomes 10, then the first 0 becomes 10, but then gives one to the right, becoming 1). It's messy. The result is that this column becomes `10 - 1 = 1`.
Next: After borrowing, this digit is now `0` (it was 0, then lent one). `0 - 1` → Can't do. Borrow from the MSB. This becomes `10 - 1 = 1`.
MSB: After lending, the `1` becomes `0`. `0 - 0 = 0`.

Result

$(1001)_2$ - $(110)_2$=$(11)_2$

V. Encoding signed numbers (integers)

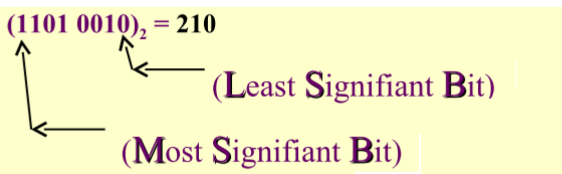
In digital systems, we often need to represent both **positive** and **negative integers**. Since the binary number system naturally represents only non-negative integers (unsigned numbers), special methods are used to encode signed integers.

The three most common representation methods are:

- Signed Magnitude (SM)
- One's Complement (1's complement, 1cp)
- Two's Complement (2's complement, 2cp)

V.1. Representation methods

A signed integer representation must include both sign information and magnitude information. Conventionally, the most significant bit (MSB) is reserved as the sign bit:



- 0 → Positive number (+)
- 1 → Negative number (-)

The remaining bits represent the magnitude of the number, but how they do this depends on the method used (SM, 1's complement, or 2's complement).

Let's assume we are working with an **n-bit** representation (e.g., 4-bit, 8-bit, 32-bit).

V.2. Signed Magnitude Method (SM)

The (SM) is the most intuitive method because it mimics how we write numbers on paper. Converting a decimal signed number to binary (Base-2) using the signed magnitude method (SM) involves following a systematic, two step procedure.

1. **The MSB is the sign bit.**
2. **The remaining n-1 bits represent the absolute value of the number in standard binary.**

Example: N=+7 and -7, Convert to binary with n=4 bits

N=4 bits→MSB = sign bit, remaining bits (3bit) = magnitude.

$|7|=(111)_2$

Adding the signed bit (MSB)

Result

$+7=(0111)_2$

$-7=$**$(1111)_2$**

Drawback: Two representations for zero: +0=0000 and −0=1000

V.3. One's complement Method (1cp)

This method aims to make subtraction easier by defining a negative number as the bitwise complement (inverse) of its positive counterpart.

1. **The MSB is still the sign bit.**
2. **For a positive number, the magnitude bits are the standard binary representation.**
3. **For a negative number, the magnitude bits are obtained by flipping all the bits (including the sign bit) of the positive number.**
4. **Process to find -X:**
    - Write the binary representation of +**X**.
    - Invert every bit **(0 becomes 1, 1 becomes 0).**

Example: N=+7 and -7, Convert to binary with n=4 bits

N=4 bits→MSB = sign bit, remaining bits (3bit) = magnitude.

$|7|=(111)_2$

Adding the signed bit (MSB) and flip all bits of (+7)

+7=0111
-7=1000
Result

$-7=(1000)_2$

Advantage:

Easier Subtraction: Subtraction can be performed using addition. To calculate A - B, you can instead calculate A + (-B), where -B is the one's complement of B.

Drawbacks:

- **Two Zeros:** The problem persists. 0000 is positive zero. Its one's complement is 1111, which is negative zero.
- **End-Around Carry:** When you add two one's complement numbers and there is a carry-out from the MSB, you must add that carry back to the result (this is called an "end-around carry"). This adds an extra step to the arithmetic operation.

V.4. Two's complement Method (2cp)

This is the universal method used in modern computers. It eliminates the problems of both Signed Magnitude and One's Complement.

1. **The MSB is the sign bit and has a negative weight.**
2. **For a positive number, the representation is identical to standard unsigned binary.**
3. **For a** negative **number -X, the representation is the result of the operation.**
4. **Process to find -X:**

    - Start with the binary representation of +X.
    - **Invert all the bits** (this gives you the one's complement).
    - **Add 1** to the result.

Example: N=+7 and -7, Convert to binary with n=4 bits

N=4 bits→MSB = sign bit, remaining bits (3bit) = magnitude.

$|7|=(111)_2$

Adding the signed bit (MSB) and flip all bits of (+7)

Step 1: One's complement

+7=0111
-7=1000
Step 2: Add +1

```
  1000   (1CP -7)
+
      1
---------
  1001 (2CP -7)
```

Result

$-7=(1001)_2$

Advantages:

- Only one representation for zero.
- Arithmetic operations (addition, subtraction) are simplified → same hardware can handle both signed and unsigned numbers.

Table 2.4: Comparison Table

Method	Zero Representations	Range (n bits)	Example (n=4, for ±7)
SM	+0, -0	−7-7 to +7+7	+7 = 0111, -7 = 1111
1cp	+0, -0	−7-7 to +7+7	+7 = 0111, -7= 1001
2cp	Only one (0000)	−8-8 to +7+7	+7 = 0111, -7 = 1001

The following table compares the three main methods for representing negative integers in binary. Its core message is that Two's Complement is the best and universally adopted method for modern computing.

Table 2.5: Comparison of Signed Integer Representation Methods

Feature	Signed Magnitude (SM)	One's Complement (1CP)	Two's Complement (2CP)
Concept	Sign + Absolute Value	Sign + Bitwise Inverse	Sign + Bitwise Inverse + 1
Positive Number	0 + Binary Value	0 + Binary Value	0 + Binary Value
Negative Number	1 + Binary Value	Bitwise Complement of +X	Bitwise Complement of +X then + 1
Number of Zeros	**Two** (+0 & -0)	**Two** (+0 & -0)	One
Hardware Complexity	High (Complex sign logic)	Medium (Requires end-around carry)	Low (Uses standard adder)
Arithmetic Operations	Complex, requires sign check	Simpler, but needs end-around carry	Simplest, uses standard adder
Range (n-bit)	$\pm(2^{(n-1)} - 1)$	$\pm(2^{(n-1)} - 1)$	$-2^{(n-1)}$ to $+(2^{(n-1)} - 1)$

VI. ASCII encoding

ASCII stands for **American Standard Code for Information Interchange**. It is one of the earliest and most widely used character encoding schemes in computers and communication systems.

- Each character (letter, digit, symbol, or control code) is represented using a unique **7-**bit binary code (values from 0 to 127).
- In modern systems, ASCII is often stored in 8 bits (1 byte), with the extra bit set to 0.

Table 2.6: ASCII Code Ranges

Range	Characters Represented
0 – 31	Control characters (non-printable, e.g., `NULL`, `BEL`, `ESC`)
32 – 47	Punctuation symbols (`space`, `!`, `"`, `#`, etc.)
48 – 57	Digits `0-9`
65 – 90	Uppercase letters `A-Z`
97 – 122	Lowercase letters `a-z`
127	`DEL` (delete)

Example: Characters

A → Decimal: 65 → Binary: 1000001
a → Decimal: 97 → Binary: 1100001
0 → Decimal: 48 → Binary: 0110000
Space → Decimal: 32 → Binary: 0100000

ASCII plays a crucial role in computing because it provides a standardized way for communication between devices and software, ensuring consistency in text representation across different systems. It also served as the basis for extended encodings, such as ISO-8859 and Unicode (UTF-8), which support a much larger set of characters and symbols used worldwide. Despite the rise of these newer standards, ASCII is still widely used in programming, networking protocols, and data representation, making it a fundamental building block in digital communication.

VII. Activity

The numbers N, M and P 8-bit signed integer: N =+10.

1) Give the 8-bit binary value of N, in three representations: "signed and magnitude", "1's complement," and "2's complement".
2) Give the 8-bit binary value of P

A) Give the binary and decimal value of the signed integer **P**=N-M, with $M=(0000\ 1100)_2$ and all numbers are represented as 8-bit and magnitude representation (SM).

B) Give the binary and decimal value of the signed integer **P** =N-M, with $M=(0000\ 1100)_2$, and all numbers are represented as 8-bit and 1's complement representation(1CP).

C) Give the binary and decimal value of the signed integer **P** =N-M, with $M=(0000\ 1100)_2$, and all numbers are represented as 8-bit and 2's complement representation(2CP)**.** ***(All operations must be performed in binary)***

Chapter 3: Boolean algebra and logic Gates

I. Introduction

Boolean algebra and logic gates form the foundation of digital electronics and computer **systems**. All modern digital devices, from simple calculators to advanced processors, rely on principles of Boolean logic to perform operations, make decisions, and process information.

Boolean algebra, introduced by George Boole in the mid-19th century, provides a mathematical framework for analyzing and simplifying logical expressions. Unlike conventional algebra, which deals with real numbers, Boolean algebra operates on binary variables that take only two values: **0 (false)** and **1 (true)**.

Logic gates are the physical implementation of Boolean functions in electronic circuits. They are the building blocks of digital hardware, used to design circuits that perform tasks such as addition, comparison, memory storage, and control operations. Common logic gates include **AND, OR, NOT, NAND, NOR, XOR, and XNOR**, each corresponding to a fundamental Boolean operation.

This chapter explores the basic rules of Boolean algebra, the properties that allow simplification of expressions, and how these principles are translated into logic gate circuits. Mastering these concepts is essential for understanding the design of digital systems, microprocessors, and computer architecture.

In this chapter, we will explore:

- **The Basics of Boolean Algebra:** The fundamental operators (AND, OR, NOT), Boolean expressions, and truth tables.
- **Logic Gates:** The symbols and behavior of the basic gates (AND, OR, NOT, NAND, NOR, XOR, XNOR) that perform Boolean operations.
- **Boolean Laws and Theorems:** The set of rules (such as Commutative, Associative, Distributive, De Morgan's Theorems) that allow us to manipulate and simplify complex logical expressions, leading to more efficient circuit designs.
- **Canonical Forms:** Standard ways of representing Boolean functions, such as the Sum-of-Products (SOP) and Product-of-Sums (POS) forms, which provide a systematic method for designing logic circuits.
- **Circuit Synthesis and Analysis:** The process of translating a real-world problem into a Boolean expression, designing its logic gate implementation, and analyzing its function.

II. Boolean algebra and logic functions

To understand the fundamental principles of Boolean algebra, its operations, and how it is used to define, analyze, and simplify logic functions that form the basis of digital circuit design.

Boolean algebra is a branch of mathematics that deals with **variables called Logical variable that can have only two possible values**: **TRUE (1)** or **FALSE (0)**. This binary nature makes it the perfect mathematical tool for analyzing and designing digital circuits, where signals are either a **high voltage (~1)** or a **low voltage (~0)**.

Boolean algebra provides the **rules**, and logic gates provide the **tools**. Together, they allow us to:

1. **Formally describe the behavior of a digital system.**
2. **Analyze complex circuits by deriving their Boolean functions.**
3. **Simplify circuits to use the fewest possible gates, optimizing for cost, power, and speed.**
4. **Synthesize new circuits from a set of requirements.**

Algebraic representation is based on three things. The first is a logical variable. The second is a logical operator. The third is a logical function.

II.1. logical variable

Boolean algebra is a mathematical system that represents **logical operations** and relationships between binary variables. Each variable can take only two possible values:

- **1** → represents **True / High / ON**
- **0** → represents **False / Low / OFF**

Boolean expressions describe how variables combine using logical operators. These expressions can be represented in different ways:

II.2. Logic operators

Logic operators, or Boolean operators, are symbols or words used in logic to combine or manipulate logic functions. There are three fundamental operations in Boolean algebra from which all others are built. Each operation has a corresponding **logic gate**.

- **The NOT operator (Inversion / Complement)**

  Symbol: $\overline{A}$

  Logic Gate: NOT gate

  Definition: The output is 1 only if the input is 0, and vice versa. It inverts the input.

Truth Table:

A	**NOT A ($\overline{A}$)**
0	**1**
1	**0**

- **The AND operator (Logical Multiplication)**

  Symbol: A.B

  Logic Gate: AND gate

  Definition: The output is 1 only if all inputs are 1. If any input is 0, the output is 0.

  Truth Table:

A	**B**	**A · B**
0	0	**0**
0	1	**0**
1	0	**0**
1	1	**1**

- **The OR operator (Logical Addition)**

  Symbol: A+B

  Logic Gate: OR gate

  Definition: The output is 1 only if all inputs are 1. If any input is 0, the output is 0.

  Truth Table:

A	**B**	**A + B**
0	0	**0**
0	1	**1**
1	0	**1**
1	1	**1**

- **Derived Logical operators (NAND, NOR, XOR, XNOR)**

  The NAND Operation (NOT AND)

  The NOR Operation (NOT OR)

  The XOR Operation ($A\oplus B = \overline{A}B + A\overline{B}$)

  The XOR Operation (NOT XOR)

II.3. Logic Functions

Logic functions describe the relationship between binary inputs and a single binary output. They form the foundation of digital systems and circuits, where every computation or decision is reduced to operations on 0s and 1s.

$$F(X_1,X_2,\ldots,X_n):\{0,1\}^n \rightarrow \{0,1\}$$

Takes n binary inputs and produces one binary output.

Logic functions can be represented in three main forms:

1. **Algebraic Expression** (Boolean formula) Example : $F(A,B) = AB + A\bar{B} + \overline{AB}$
2. **Truth Table:** Lists all possible inputs and outputs.
3. **Logic Circuit (Gate Diagram):** Graphical representation using logic gate symbols.

II.3.1. Algebraic Expression representation

In Boolean algebra, logic functions are expressed using algebraic expressions that describe the relationship between binary inputs and outputs. These expressions use Boolean operators (AND, OR, NOT, etc.) to define the function in symbolic form.

Examples of Algebraic Expressions

1. **Simple Functions**
   F(A,B)=A+B → Logical OR
   F(A,B)=ABF(A,B) = ABF(A,B)=AB → Logical AND
   F(A)=A¯F(A) = \overline{A}F(A)=A → Logical NOT
2. **Composite Functions**
   $F(A,B,C)=(A+B)\cdot C$
3. **Canonical Algebraic Forms**
   Sum of Products (SOP) (OR (+) of multiple AND terms)
   $F(A,B,C)=ABC+ABC+ABC$
   Product of Sums (POS) (AND (·) of multiple OR terms)
   $F(A,B,C)=(A+B)(A+C)(B+C)$

The algebraic expression representation is a symbolic way to describe logic functions using Boolean operators (AND, OR, NOT, etc.). It is widely used because it connects truth tables and circuit design, while also allowing simplification through Boolean algebra laws.

II.3.2 Truth tables representation

A truth table is a tabular method used to represent all possible combinations of input values and their corresponding output for a logic function. It provides a complete picture of how a Boolean expression behaves for every input case.

Structure of a Truth Table

1. **Inputs:** Each column corresponds to a binary input variable (A, B, C, …).
2. **Outputs:** One or more columns correspond to the logic function(s).
3. **Rows:** Each row represents one possible input combination. For nnn input variables, the truth table has 2^n rows.

Example with Three Variables

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

A	*B*	*C*	*A+B*	$\bar{C}$	*F(A,B,C)*
0	0	0	0	1	0
0	0	1	0	0	0
0	1	0	1	1	1
0	1	1	1	0	0
1	0	0	1	1	1
1	0	1	1	0	0
1	1	0	1	1	1
1	1	1	1	0	0

Truth tables provide a systematic and visual method to analyze Boolean functions. They show every possible input combination and the resulting output, making them a fundamental tool in digital logic design.

II.3.3. Logic Circuit representation (Gate Diagram)

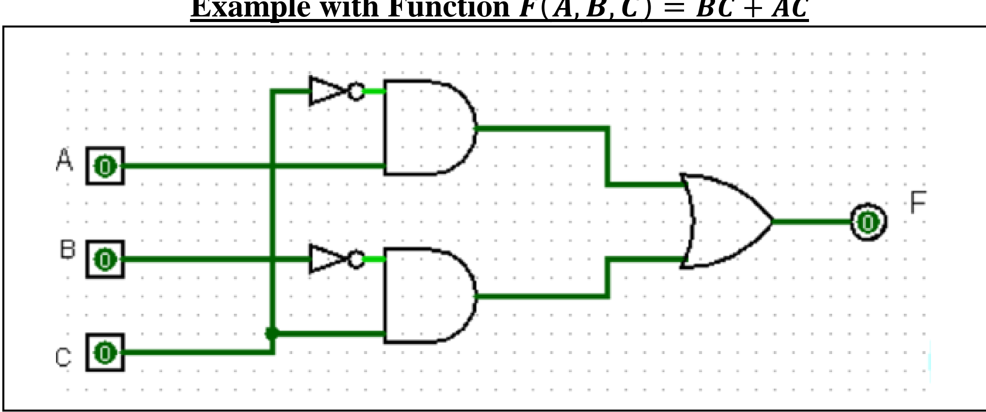
A logic circuit representation, also known as a gate diagram, is a graphical method of illustrating Boolean expressions or truth tables using logic gate symbols. It visually shows how binary inputs are processed through gates to produce the required output.

Each Boolean operator has a corresponding electronic symbol used in circuit diagrams.

The Basic Logic Gate Symbols

Gate	Symbol	Boolean Expression	Function
AND		$F = A + B$	Output = 1 if all inputs = 1
OR		$F = A.B$	Output = 1 if at least one input = 1
NOT		$F = \bar{A}$	Output is the inverse of input
NAND		$F = \overline{A.B}$	Inverse of AND
NOR		$F = \overline{A + B}$	Inverse of OR
XOR		$F = A \oplus B$	Output = 1 if inputs differ
XNOR		$F = \overline{A \oplus B}$	Output = 1 if inputs are equal

Example with Function $F(A,B,C) = \overline{B}C + A\overline{C}$



III. Postulates of boolean algebra

Boolean algebra is a mathematical system based on a set of elements, operations, and postulates (axioms). It was first introduced by George Boole in 1854 and later formalized for digital logic by Claude Shannon.

The postulates define the fundamental properties that any Boolean system must satisfy. These rules are the foundation for manipulating logic expressions and designing digital circuits.

Table 3.2: Postulates of Boolean Algebra (Summary Table)

Postulate	Law / Rule	Expression(s)	Example
Closure	Results stay in {0,1}	$A + B \in \{0,1\},\ A \cdot B \in \{0,1\}$	If A=1, B=0 → A+B=1
Identity	0 (OR), 1 (AND)	$A + 0 = A,\ A \cdot 1 = A$	$1 \cdot A = A$
Commutativity	Order doesn't matter	$A + B = B + A,\ A \cdot B = B \cdot A$	$1 + 0 = 0 + 1$
Distributivity	AND over OR, OR over AND	$A \cdot (B + C) = AB + AC$ $A + (BC) = (A + B)(A + C)$	$1 \cdot (0 + 1) = 1$
Idempotency	Repetition doesn't change value	$A + A = A,\ A \cdot A = A$	$1 + 1 = 1$
Complement	Each variable has an opposite	$A + A' = 1,\ A \cdot A' = 0$	$0 + 1 = 1, 1 \cdot 0 = 0$
Domination	Nullifying values	$A + 1 = 1,\ A \cdot 0 = 0$	$0 + 1 = 1$
Absorption	Simplifies redundant terms	$A + (AB) = A,\ A(A + B) = A$	$1 + (1 \cdot 0) = 1$

IV. Theorems of Boolean Algebra

A Boolean expression can be implemented directly as a logic circuit. However, the initial expression derived from a truth table is often not the most efficient. **Simplification** aims to find an equivalent expression that uses fewer gates and/or fewer inputs to gates. Boolean Algebra theorems provide the formal framework for this simplification.

Table 3.3: Boolean Algebra Theorems

Theorems	Expression	Explanation
Consensus Theorem	$AB + A'C + BC = AB + A'C$	Redundant term BC can be removed.
Generalized Consensus	$(A + B)(A' + C)(B + C) = (A + B)(A' + C)$	Extended form eliminating redundancy.
De Morgan's Theorem	$(A \cdot B)' = A' + B', (A + B)' = A' \cdot B'$	Negation distributes with operator switch.
Generalized De Morgan	$(A_1 \cdot A_2 \cdot ... \cdot A_n)' = A_1' + A_2' + ... + A_n'$ $(A_1 + A_2 + ... + A_n)' = A_1' \cdot A_2' \cdot ... \cdot A_n'$	Works for any number of variables.

V. Simplifying logic functions

The Power of Simplification of Boolean function, that can be directly implemented as a logic circuit, but the expression obtained from a truth table is often not the most efficient. The goal of simplification is to derive an equivalent expression that requires fewer gates and/or fewer inputs per gate. This reduction offers several advantages: lower cost, since fewer chips and components are needed; increased reliability, as fewer components mean fewer potential points of failure; and faster operation, thanks to reduced propagation delay through fewer logic levels.

Simplifying logic functions is a common task in digital design and computer science. The goal is to simplify a Boolean expression or logic circuit while preserving its functionality. There are several techniques for simplifying logic functions, including:

- **Boolean Algebra:** Use Boolean algebra laws and theorems to manipulate and simplify expressions. Common laws include the commutative, associative, distributive, identity, and complement laws.

- **Karnaugh Maps (K-Maps):** It helps simplify Boolean expressions and minimize logic circuits. Karnaugh Maps are particularly useful for functions with a small number of variables.

V.1. Boolean Algebra method

The **Boolean Algebra method** is one of the most fundamental approaches to simplifying logic functions. It relies on the application of **Boolean laws, rules, and theorems** to manipulate expressions into simpler, but logically equivalent, forms.

The objective is to reduce the number of logic gates, inputs, and overall complexity of the circuit while preserving its functionality.

Steps in the Boolean Algebra Simplification Method:

1. **Write the expression** from the truth table or problem statement.
2. **Apply Boolean laws and theorems** systematically to reduce terms.
3. **Eliminate redundant variables or terms** whenever possible
4. **Stop when no further simplification** is possible.

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

Step 1: Factor A : $F = A(B + \bar{B})$
Step 2: Apply Complement Law: $B + \bar{B} = 1$
Step 3: Simplify: $F = A.1 = A$

The Boolean algebra method is a fundamental skill for anyone working with digital logic. While visual methods like Karnaugh Maps are often faster and guarantee a minimum form, the algebraic method is indispensable for formal verification and understanding the underlying principles of logic simplification.

V.2. Karnaugh Maps method (K-Maps)

To visually simplify Boolean expressions into their minimal Sum-of-Products (SOP) or Product-of-Sums (POS) form by grouping adjacent cells in a special grid.

Boolean algebra simplification requires intuition and practice. The K-map method provides a systematic, visual technique that:

- Guarantees the simplest SOP or POS expression.
- Is much faster and less error-prone than algebraic manipulation for 2-5 variables.
- Makes it easy to identify and eliminate redundant terms.

A K-map is a graphical representation of a truth table. The key innovation is that adjacent cells differ in only one input variable. This layout visually highlights terms that can be combined. **Gray Code Order:** The row and column headers are not in binary order (00, 01, 10, 11) but in Gray code order (00, 01, 11, 10). This ensures adjacent cells are always logically adjacent (differ by one bit).

Common K-Map Sizes:

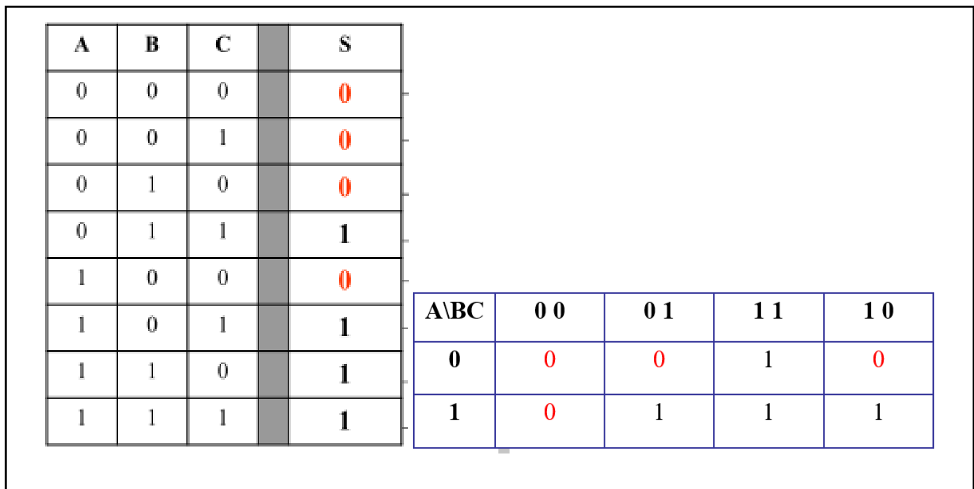
- 2-Variable: 2x2 grid (4 cells)
- 3-Variable: 2x4 grid (8 cells)
- 4-Variable: 4x4 grid (16 cells)

Example: 3-Variable amd 4-Variable K-Maps Structure

A\BC	0 0	0 1	1 1	1 0
0	m0	m1	m3	m2
1	m4	m5	m7	m6

AB\CD	00	01	11	10
00	m0	m1	m3	m2
01	m4	m5	m7	m6
11	m12	m13	m15	m14
10	m8	m9	m11	m10

Example: Function with the following truth table



A	B	C	S
0	0	0	0
0	0	1	0
0	1	0	0
0	1	1	1
1	0	0	0
1	0	1	1
1	1	0	1
1	1	1	1

A\BC	0 0	0 1	1 1	1 0
0	0	0	1	0
1	0	1	1	1

V.2.1. Methodology

The process of simplifying Boolean functions can be carried out using different techniques. The goal is to obtain an **equivalent but minimal expression** that reduces circuit complexity while preserving the same functionality. Two widely used methods are the **Boolean Algebra method** and the **Karnaugh Map (K-Map) method**.

- Once you have identified the groups of '1s', write down the simplified terms for each group. These terms can be derived from the row and column labels associated with the cells in each group.
- Combine the simplified terms to create the minimized Boolean expression. The simplified expression will have fewer terms than the original expression while preserving the same logic function.

Simplification Steps :

1. **Place 1s in cells corresponding to minterms where the function is true.**
2. **Group adjacent 1s in rectangles of size 1, 2, 4, 8, … (must be powers of 2).**
3. **Each group should be as large as possible.**
4. **Groups may overlap and wrap around the edges.**
5. **Derive the simplified Boolean expression by identifying variables that remain constant within each group.**
6. **Combine results to obtain the final minimal Boolean expression.**

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

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

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

The Boolean Algebra method provides a strong theoretical foundation, while the K-Map offers a practical and visual approach to simplification. In digital design, both methods are often used together to verify results and ensure optimal circuit design.

VI. Logic Gates and circuits

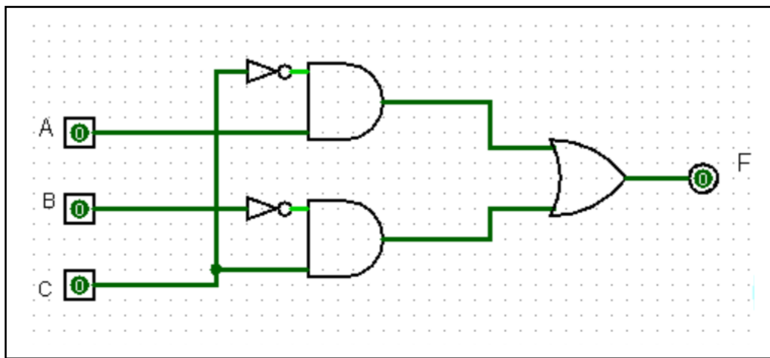
Logic gates are the fundamental building blocks of digital systems. They are electronic devices that perform logical operations on one or more binary inputs to produce a single binary output. Each gate implements a basic Boolean function such as AND, OR, NOT, NAND, NOR, XOR, and XNOR.

A logic circuit is a network of interconnected logic gates designed to perform a specific function.

- Combinational circuits: Output depends only on the current inputs (e.g., adders, multiplexers).
- Sequential circuits: Output depends on current inputs and past states (e.g., flip-flops, counters).

By combining gates, complex functions such as arithmetic operations, data storage, and control logic are implemented in digital systems, processors, and embedded devices.

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



VI.1. Software and Online Simulation Tools for Logic Gates and Circuits

Learning logic gates and circuits is greatly enhanced by the use of **simulation tools**. These tools allow students and designers to **build, test, and visualize** digital circuits without needing physical hardware. They are especially useful for experimenting with Boolean functions, combinational logic, and sequential systems.

Example: Software Tools

Offline Simulators
Logisim (open-source)
Proteus
Multisim (by NI)
Online Simulators
Logic.ly (https://logic.ly)
CircuitVerse (https://circuitverse.org)
Tinkercad Circuits (https://tinkercad.com/circuits)

VII. Activity

Consider the following Boolean logic function:

$$F(x,y,z,w) = y\bar{z}\bar{w} + \bar{y}zw + \bar{x}\bar{y}z\bar{w} + \bar{x}yzw + \bar{x}yz\bar{w}$$

1) Simplify the function *F(x,y,z,w)* and draw the equivalent circuit.
2) Which (4-variable) terms need to be added to the initial (non-simplifying) function *F(x,y,z,w),* to obtain a simplifying function with only two-variable terms.

Chapter 4: Programming Algorithm

I. Introduction

System software manages the hardware and provides a platform for running application software. Its components include:

- **System Software**

a. **Operating System (OS):** Manages hardware resources (CPU, memory, storage, I/O devices). Examples: Windows, Linux, macOS.
b. **Utility Programs:** Perform maintenance and optimization tasks. Examples: Antivirus software, disk defragmenters, backup tools, system cleaners.
c. **Device Drivers:** Enable communication between the operating system and hardware devices. Example: Printer drivers, graphics card drivers.
d. **Firmware:** Low-level software stored in hardware devices to control basic operations. Example: BIOS/UEFI in a motherboard.

- **Application Software**

Application software allows users to perform specific tasks. Its components include:

a. **Productivity Software:** Helps users perform office or creative tasks. Examples: Microsoft Office (Word, Excel, PowerPoint), Google Docs.
b. **Media Software:** For multimedia creation, editing, or playback. Examples: Adobe Photoshop, VLC Media Player.
c. **Database Software:** For storing, managing, and retrieving data efficiently. Examples: MySQL, Oracle, Microsoft SQL Server.
d. **Web Browsers and Internet Applications:** For accessing the internet and web services. Examples: Google Chrome, Mozilla Firefox, Microsoft Edge.
e. **Enterprise Software:** For business operations and management. Examples: ERP systems (SAP), CRM systems (Salesforce).

- **Development Software (Programming Tools)**

This category supports software developers in creating applications. Components include:

a. **Compilers and Interpreters:** Convert source code into executable machine code, and interpreters execute code line by line, translating and executing each line as it is encountered, while compilers translate the entire source code into machine code or an intermediate representation before execution.Example: GCC compiler for C/C++, Python interpreter.

b. **Integrated Development Environments (IDEs):** Provide tools for coding, debugging, and testing. Examples: Visual Studio, Eclipse, PyCharm.
c. **Version Control Systems:** Track code changes and collaborate among developers. Examples: Git, SVN.
d. **Debuggers:** Identify and fix software bugs.

Applications
(Word, Excel, Jeux, Maple, google chrome, etc.)

Operating Systems
(DOS, Windows, Linux, Unix, Macintosh, Android…)

programming languages
(pascal, Java, C, C++, python,….)

Hardware
(PC, server, phone, machines, etc.)

Figure 4.1: Computer Software components

Figure 4.2 : Programming Steps

Programming involves the writing of algorithms →This explains why algorithms matter.

II. Algorithm Definition & Meaning

The term "algorithm" is the technique of performing arithmetic with Arabic numerals developed by **al-Khwārizmī**.

An algorithm is a step-by-step procedure or a set of well-defined rules for solving a specific problem or completing a specific task. A high-level, human-readable way of describing an algorithm that mixes natural language with code called pseudo_code like steps, without following the rules of a specific programming language.

Pseudo-code is a method used to describe algorithms by combining natural language with a structured, code-like format. It serves as a bridge between plain language and actual

programming languages. Instead of following the strict syntax and rules of a specific language, pseudo-code focuses on clarity and logic, outlining the sequence of steps in a way that is easy for humans to read and understand. It highlights the control structures (such as loops, conditions, and procedures) while avoiding unnecessary technical details, making it an effective tool for designing, explaining, and communicating algorithms before converting them into real code.

Example: Algorithm: Find the Largest Number

```
START
  Set MAX ← first element of the list
  FOR each number in the list
    IF number > MAX THEN
      Set MAX ← number
    ENDIF
  ENDFOR
  PRINT MAX
END
```

Description:

- Uses natural language (e.g., "Set", "FOR each number")
- Includes structured code-like steps (like IF, ENDIF, PRINT)
- Focuses on **logic, not syntax**, so it can be easily adapted to any programming language later.

III. Variables in an algorithm

Variables are an essential concept in programming and algorithm design. They are used to store and manipulate data.

- **Data Storage**: Variables are like containers that hold data. They can store various types of data, such as numbers, text, or more complex structures like arrays and objects.
- **Naming**: Variables have names that programmers assign to them. These names are used to refer to the data stored in the variable.
- **Data Types**: Variables have data types that determine the kind of data they can store. Common data types include integers, floating-point numbers, strings, and boolean values.

- **Declaration**: Before using a variable, it must be declared. This informs the computer's memory system to allocate space for the variable. The syntax for declaring a variable varies from one programming language to another.

III.1. Variables Name

When defining variables in an algorithm, it is important to follow clear rules to maintain readability and avoid errors. Reserved words or programming keywords should not be used as variable names, since they already have specific meanings in programming languages.

Avoid reserved words by not using programming language keywords as variable names. For example, in C the word *for* cannot be used because it is a keyword for loops. Variable names should begin with a letter ranging from a–z or A–Z.

They can include letters, numbers, and underscores, but special characters and spaces should be avoided.

It is best to choose descriptive names that clearly indicate the purpose or content of the variable, which makes the code more readable and easier to understand. For instance, instead of using generic names like *a* or *x*, use meaningful names such as *total_sum* or *user_average*.

III.2. Variable Types

In algorithms, variables can have different data types, which determine the kind of data they can store and how that data is manipulated.

- **Integer (int):** Integer variables store whole numbers, both positive and negative, without fractional parts.
- **Floating-Point (float):** Floating-point variables are used to store real numbers with decimal points.
- **Character (char):** Character variables hold sequences of characters, such as text or words.
- **Boolean (bool):** Boolean variables have only two possible values: True and False.
- **Array:** Arrays are used to store collections of elements of the same data type.
- **Pointers:** In low-level languages like C and C++, variables can also store memory addresses (pointers or references) to other data in memory.

III.3. Variables designation

In algorithms, variables can be designated according to their role in processing data.

- **Input Variable or Data:** refer to the values or information that a program receive from external sources, such as a user, other software/hardware components, or data files.
- **Output Variable or Results:** refer to the values, data, or information that a program or function generates or produces as a result of its execution.
- **Intermediate Variable :**Typically refers to a variable that is used to store an intermediate result or temporary value during a computation or operation.

Example: Algorithm: Variable Declaration

INPUT: number1, number2 **Integer**
INTERMEDIATE: sum **Integer**
OUTPUT: average **Integer**

Description:

- number1 and number2 are **input variables** (values entered by the user).
- sum is an **intermediate variable** (stores the partial result).
- average is the **output variable** (final result to be displayed).

IV. Expressions and operators in an algorithm

In algorithms and programming, expressions and operators are fundamental concepts used for performing operations on data. Expressions are combinations of values, variables, and operators that can be evaluated to produce a result. Operators are symbols or keywords that perform specific operations on the operands.

IV.1. Expressions

The following expressions enable algorithms to process data, make decisions and control the flow of execution by using different operators.

- **Arithmetic Expressions**: These expressions involve mathematical operations, such as addition, subtraction, multiplication, and division.

- **Relational Expressions**: Relational expressions are used to compare values and produce a Boolean result (True or False).
- **Logical Expressions**: Logical expressions involve logical operators and are used for making decisions or combining conditions.
- **Bitwise Expressions**: In low-level programming or when dealing with binary data, bitwise expressions can be used to manipulate individual bits within values.

Example: Combines arithmetic, relational, and logical expressions in one Algorithm

```
INPUT: math_score, science_score, english_score Integer
INTERMEDIATE: total Integer
OUTPUT: average Integer
  BEGIN
1: total = math_score + science_score + english_score
2: average = total / 3
3: if average ≥ 50 AND math_score ≥ 40 AND science_score ≥ 40 AND english_score ≥ 40
     PRINT "Student Passed"
   else
     PRINT "Student Failed"
   end_if
  END
```

Description:

- **Arithmetic expressions**: (line 1 and 2), total = math_score + science_score + english_score, and average = total / 3
- **Relational expressions**: (line 3) average ≥ 50, math_score ≥ 40, science_score ≥ 40, ,and english_score ≥ 40
- **Logical expression**: (line 3) average ≥ 50 AND math_score ≥ 40 AND science_score ≥ 40 AND english_score ≥ 40

IV.2. Operators

Operators are special symbols or keywords used in algorithms to perform operations on data. They allow variables, constants, and expressions to be manipulated. Operators can be grouped into several categories**:**

Table 4.1. Operators in Algorithms

<table>
<tr><th>Type</th><th>Operator</th><th>Meaning / Use</th><th>Example</th></tr>
<tr><td>Arithmetic Operators</td><td>+</td><td>Addition</td><td>a + b → adds two numbers</td></tr>
<tr><td></td><td>-</td><td>Subtraction</td><td>a - b → subtracts b from a</td></tr>
<tr><td></td><td>*</td><td>Multiplication</td><td>a * b → multiplies numbers</td></tr>
<tr><td></td><td>/</td><td>Division</td><td>a / b → divides a by b</td></tr>
<tr><td></td><td>%</td><td>Modulo (remainder)</td><td>a % b → remainder of a ÷ b</td></tr>
<tr><td>Assignment Operator</td><td>=</td><td>Assigns a value to a variable</td><td>x = 10</td></tr>
<tr><td>Relational Operators</td><td>= =</td><td>Equal to</td><td>a == b</td></tr>
<tr><td></td><td>!=</td><td>Not equal to</td><td>a != b</td></tr>
<tr><td></td><td><</td><td>Less than</td><td>a < b</td></tr>
<tr><td></td><td>></td><td>Greater than</td><td>a > b</td></tr>
<tr><td></td><td><=</td><td>Less than or equal to</td><td>a <= b</td></tr>
<tr><td></td><td>>=</td><td>Greater than or equal to</td><td>a >= b</td></tr>
<tr><td>Logical Operators</td><td>&&</td><td>Logical AND (true if both true)</td><td>(a > 0 && b > 0)</td></tr>
<tr><td></td><td>||</td><td>Logical OR (true if at least one true)</td><td>(a < 0 || b >0)</td></tr>
<tr><td></td><td>!</td><td>Logical NOT (reverses condition)</td><td>!(a > b)</td></tr>
<tr><td>Bitwise Operators</td><td>&</td><td>Bitwise AND</td><td>a & b</td></tr>
<tr><td></td><td>|</td><td>Bitwise AND</td><td>a | b</td></tr>
<tr><td></td><td>^</td><td>Bitwise XOR</td><td>a ^ b</td></tr>
<tr><td></td><td>~</td><td>Bitwise NOT (inverts all bits)</td><td>~a</td></tr>
<tr><td></td><td><<</td><td>Left shift</td><td>a << 1</td></tr>
<tr><td></td><td>>></td><td>Right shift</td><td>a >> 1</td></tr>
<tr><td>Increment Operators</td><td>++</td><td>Increment (increase by 1)</td><td>a++ or ++a</td></tr>
<tr><td>Decrement Operators</td><td>- -</td><td>Decrement (decrease by 1)</td><td>a-- or --a</td></tr>
</table>

V. Basic Instructions

Basic instructions are the fundamental building blocks of algorithms. They describe the simple actions that an algorithm or program can perform. These instructions can be grouped into several categories:

Assignment Instruction, Input Instruction, Output Instruction, Processing Instruction, Control Instructions (basic flow control), and Looping Instruction (Repetition).

V.1. Assignment Instruction

An assignment is a fundamental operation that involves storing a value in a variable. It is used to initialize a variable, update its value, or modify the content of a data structure. Assignment is typically represented by the '=' operator, which assigns a value to a variable.

Variables can be updated by assigning new values to them. This is one of the most common operations in algorithms and programming. When a new value is assigned, it replaces the current value stored in the variable.

Example: Shows how assignment replaces old values

```
INPUT: A, B Float
INTERMEDIATE: ------
OUTPUT: S Integer
  BEGIN
        Read (A)
        Read (B)
        S = A + B
  END
```

Step-by-step explanation:

- Step 1: The program asks the user to enter a value for **A**.
- Step 2: The program asks the user to enter a value for **B**.
- Step 3: The program computes **S** = **A** + **B**.

V.2. Input Instructions

Rading input is a common operation that allows a program to obtain data from an external source, typically a numerical date, a user, a file, or another system. The specific way to read input depends on the programming language and the source of the input. Input instructions make the program interactive by asking the user to enter values, which are then stored in variables for further processing. In pseudocode: Read (variable)

Example: Shows how using Read statement

```
INPUT: A, B Float
INTERMEDIATE: ------
OUTPUT: S Integer
  BEGIN
        Read (A)
        Read (B)
        S = A + B
  END
```

Step-by-step explanation:

- Step 1: The program asks the user to enter a value for **A**.
- Step 2: The program asks the user to enter a value for **B**.
- Step 3: The program computes **S** = **A** + **B**.

Input instructions bring external values into the program, making it adaptable to different user data instead of using only fixed numbers.

V.3. Output Instructions

Is instruction refer to operations that output or display information, typically to the user via a screen, console, file, or another output medium. Writing instructions allow a program to communicate results, messages, and data to the user or store information for future reference. The specific way to write output depends on the programming language and the output destination.

Example: Shows how using Write statment

```
INPUT: A, B Float
INTERMEDIATE: ------
OUTPUT: S Integer
  BEGIN
        Read (A)
        Read (B)
        S = A + B
        Write (S)
  END
```

Step-by-step explanation:

- Step 1: The program asks the user to enter a value for **A**.
- Step 2: The program asks the user to enter a value for **B**.
- Step 3: The program computes **S = A + B**.
- Step 4: The program displays the result **S** on the screen.

VI. Building an Algorithm (My First Algorithm)

An algorithm is a step-by-step set of instructions designed to solve a specific problem or perform a task. When building an algorithm, the process usually follows a clear structure to ensure the solution is correct, efficient, and easy to understand.

When creating an algorithm, it is important to follow a clear and logical sequence. The process usually involves the following steps:

a. **Problem Definition:** Clearly state the problem that needs to be solved, and ask: *What is the goal?*

b. **Identify Inputs and Outputs:** The Inputs are the information the algorithm will receive. And Outputs: are the results the algorithm should produce.

c. **Develop the Procedure (Logical Steps):** Break down the solution into simple, ordered steps**,** and Use operations like *input, assignment, output, decision, or loop*.

d. **Write the Algorithm Representation:** Express the steps either in the **Pseudo-cod** or a human-readable way of writing instructions.

e. **Test the Algorithm:** Run through the algorithm with sample values to ensure correctness.

f. **Optimize (Optional):** Simplify or improve the algorithm to make it more efficient. For example, Remove redundant steps, or use shorter logic.

In summarize these steps into a visual diagram (flow-style) so your students can quickly memorize the process

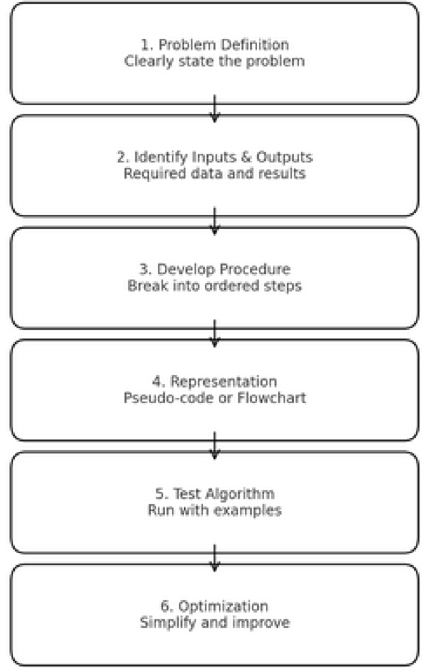


Figure 4.2: Steps to Build an Algorithm Flowchart

It appears you're describing the general structure of an algorithm as consisting of three main parts: the header, declaration, and block of instructions.

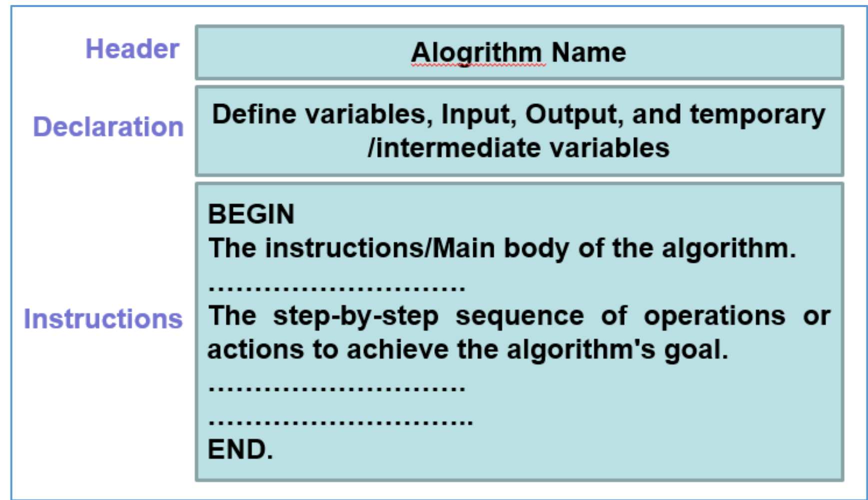


Figure 4.3: Building an Algorithm model

Example: My First Algorithm

```
Name:  Quadratic equation (second-degree equation)
INPUT: A,B,C Float
INTERMEDIATE: D Float
OUTPUT: X1,X2,X Float
  BEGIN
          Read (A), Read (B), Read (C)
          D = b*b – 4*a*c
          if (D > 0) THEN
               x1 = (-b + sqrt(D)) / (2*a)
               x2 = (-b - sqrt(D)) / (2*a)
               Write("Two distinct real roots:", x1, x2)
          End_if
          If (D == 0) THEN
               x = -b / (2*a)
               Write ("One real root (double root):", x)
          End_if
          If (D < 0) THEN
                Write ("No real roots (complex solutions)")
          End_if
END
```

VII. Conditional statements/control structures

Conditional statements (also called control structures) allow an algorithm or program to make decisions. They let the program choose different actions depending on whether a condition is true or false. They are essential for controlling the flow of execution.

VII.1. Simple if Statements

A Simple "if" statement is used to execute a block of code if a condition is true.

Example: if Statements

```
Name:  Example if_1
INPUT: A,B Float
INTERMEDIATE: --
OUTPUT: D Float
  BEGIN
          Read (A)
          Read (B)
          D=A-B
          if (D > 0) THEN
               Write("Result:",D)
          End_if
END
```

VII.2. if-else Statements

An "if-else" statement allows to execute one block of code if a condition is true and another block if the same condition is false.

Example: if-else Statements

```
Name:  Example if_2
INPUT: A,B Float
INTERMEDIATE: --
OUTPUT: D Float
  BEGIN
        Read (A)
        Read (B)
        D=A-B
        if (D > 0) Then
            Write("Result:",D)
        End_if
        else
            Write("Result: ",-D)
        End_else
END
```

VII.3. if else-if else Statements

These are used when you have multiple conditions to check and execute different code blocks based on which condition is true.

Example: if else-if else statements

```
Name:  Example if_3
INPUT: A,B Float
INTERMEDIATE: --
OUTPUT: D Float
  BEGIN
        Read (A)
        Read (B)
        D=A-B
        if (D > 0) THEN
            Write("Result:",D)
        End_if
        else
           if(D>0) THAN
               Write("Result:",-D)
           End_if
           else
           Write("Result:",0)
            End_else
        End_else
END
```

VII.4. Switch Statements

The switch statement is a control structure used to select one of many possible actions depending on the value of a given expression.

It is often used as an alternative to multiple **if … else if … else** statements, making the code cleaner and easier to read.

Example: Switch Statements

```
Name:  Example if_4
INPUT: A,B Float
INTERMEDIATE: --
OUTPUT: D Float
  BEGIN
        Read (A)
        Read (B)
        D=A-B
        Switch (D)
           Case (D<0): Write("Result:",D)
           Case (D<0): Write("Result:",-D)
           Default: Write("Result:",0) //D=0
        End_Switch
END
```

VIII. Iteration statements/Loop structures

Iteration statements, also called loops**,** allow an algorithm to repeat a set of instructions multiple times until a certain condition is met. They are used when we need to perform repetitive tasks without writing the same code several times.

Iteration is essential for automating repetitive tasks, processing collections of data, and implementing algorithms that involve repeating actions.

VIII.1. for loop

A **for loop** is a fundamental control structure in programming that allows you to execute a block of code a specific number of times. It is commonly used when you know in advance how many times you want to repeat a certain operation.

Used when the number of iterations is known in advance. Contains initialization, condition, and update in one structure.

for (initialization/start; stop; iteration/step)

Example: For loop

```
Name:  Example loop_1
INPUT: A integer
INTERMEDIATE: i integer
OUTPUT: --
  BEGIN
          Read (A)
          for (i=0 to 10 with step 1)
                Write(A+1)
                 End_for
END
```

VIII. 2. while loop

A **while loop** is a control structure in programming that allows you to repeatedly execute a block of code as long as a specified condition remains true. It continues iterating as long as the condition is true.

The condition is checked before executing the loop body, and the loop continues as long as the condition is true.If the condition is false from the start, the loop body may not execute at all.

Example: While loop

```
Name:  Example loop_2
INPUT: A integer
INTERMEDIATE: i integer
OUTPUT: --
  BEGIN
          Read (A)
          i=0
          while (i<10)
                Write(A+1)
                i=1+1
          End_while
END
```

VIII.3. repeat loop

That is similar to a "while" loop, but it guarantees that the code block will be executed at least once, even if the condition is initially false.

The loop body is executed at least once, and the condition is checked after execution. Useful when you want the instructions to run first before testing the condition.

Example: Repeat loop

```
Name:  Example loop_3
INPUT: A integer
INTERMEDIATE: i integer
OUTPUT: --
  BEGIN
          Read (A)
          i=0
          Do
             Write(A+1)
             i=1+1
          while (i<10) End_repeat
END
```

IX. Activity

Write an algorithm that prompts the user to enter any time in the form of three variables: Hour, Minute, and Second, and then displays a message indicating whether the time entered is valid or not. (Assuming all three variables are non-negative).

Examples:

Hour =**28** Minute = **31** Second =**51** is an invalid time.

Hour = **16** Minute= **3** Second = **55** is a valid time.

Chapter 5: C programming language

I. Introduction

C programming is a widely used and influential programming language that was developed in the early 1970s by Dennis Ritchie at Bell Labs. It is a high-level, general-purpose programming language known for its flexibility, efficiency, and low-level programming capabilities.

C programming has had a significant impact on the development of modern programming languages and systems. It's commonly used in areas like operating system development, device drivers, and embedded systems, where fine-grained control over hardware and resources is essential.

The C programming language is simple and easy to learn, with only a small number of keywords, making it a good starting point for beginners. It is highly portable, allowing programs to run on different platforms with little or no modification. C is efficient and produces fast machine-level code, which makes it suitable for system-level applications. It follows a structured and modular programming approach, where programs can be divided into functions for better organization and maintenance. The language provides a rich library of built-in functions to handle input, output, mathematics, and strings. One of its powerful features is low-level memory access using pointers, which gives programmers direct control over hardware. C is also extensible, allowing users to create and reuse their own functions. Being a middle-level language, it combines the features of both high-level and low-level programming. Furthermore, it supports dynamic memory allocation, giving flexibility in managing memory during execution. These features have made C widely used in developing operating systems, embedded systems, and various applications.

- **Structured Language**: Programs are divided into functions/modules.
- **Portability**: Code can run on different machines with little or no modification.
- **Efficiency**: Produces fast and optimized machine code.
- **Low-Level Access**: Direct access to memory through pointers.
- **Rich Library**: Standard library functions (I/O, string handling, math, etc.).

The general structure of a C program consists of various components and follows a specific format. A C program usually begins with **preprocessor directives** such as #include, which include header files containing standard library functions. Next, **global declarations** may be defined, such as constants, macros, or global variables. The main part of every C program is

the **main() function**, which acts as the entry point of execution. Inside main(), the program contains **declarations** of variables and **statements** that perform the required tasks. In addition to the main function, a program can also include **user-defined functions** to make the code modular, organized, and reusable. Together, these components—preprocessor directives, global declarations, the main() function, variable declarations, executable statements, and user-defined functions—form the standard structure of a C program.

The general structure of a C program consists of various components and follows a specific format.

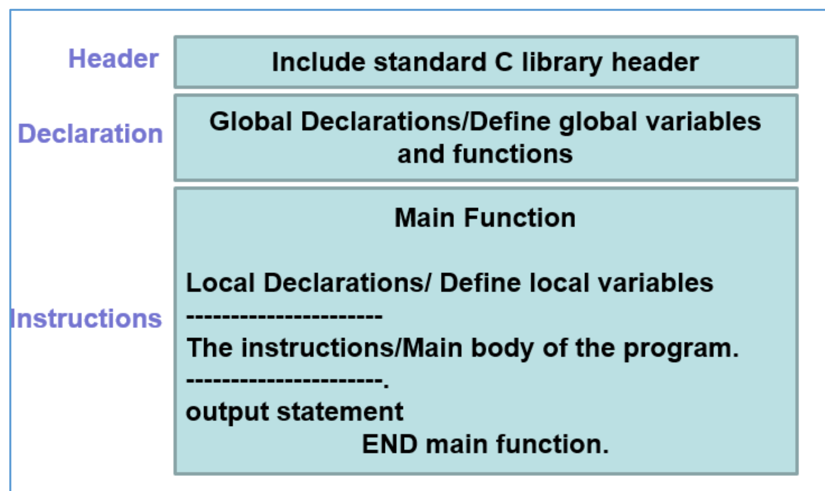


Figure 5.1: The general structure of a C program

In C, you can insert text as comments to provide explanations, notes, or documentation within your code. Comments are ignored by the C compiler and are intended solely for human readers.

- **Single-Line Comments:** Single-line comments are used for brief comments on a single line. You can start a single-line comment using //, and it continues until the end of the line. **// This is a single-line comment**
- **Multi-Line Comments:** Multi-line comments are used for longer comments that span multiple lines. You begin a multi-line comment with /* and end it with */.

/* This is a multi-line comment

It can span multiple lines

Useful for longer explanations */

II. C library

The C Standard Library is a collection of precompiled functions and header files that provide essential functionalities to C programs. Instead of writing code from scratch, programmers can

use these ready-made functions to handle common tasks such as input/output, string handling, memory management, mathematical operations, and file processing. The C Standard Library is a set of functions, macros, and constants that provide core functionality for the C programming language.

- **<stdio.h>**: Input and output operations. Functions like printf, scanf, fopen, and others are part of this header.
- **<stdlib.h>**: General utilities. Functions such as malloc, free, exit, and others for dynamic memory allocation, random number generation, and program termination.
- **<string.h>**: chain of characters or text manipulation functions like strcpy, strlen, etc.
- **<math.h>**: Mathematical functions, such as sqrt, sin, cos, and others.

A summary Table 5.1 listing the C Standard Library header files, their purposes, and example functions.

Table 5.1: Summary of Common C Library Header Files

Header File	Purpose	Example Functions
<stdio.h>	Standard input/output	printf(), scanf(), getchar(), puts()
<stdlib.h>	General utilities, memory management	malloc(), free(), exit(), rand()
<string.h>	String handling	strlen(), strcpy(), strcmp(), strcat()
<math.h>	Mathematical operations	sqrt(), pow(), sin(), cos()
<ctype.h>	Character classification/conversion	isalpha(), isdigit(), toupper(), tolower()
<time.h>	Date and time functions	time(), clock(), difftime(), strftime()
<stdbool.h>	Boolean data type support	true, false
<assert.h>	Diagnostics and debugging	assert()
<errno.h>	Error handling	errno variable, error codes
<limits.h>	Defines integer limits	INT_MAX, INT_MIN
<float.h>	Defines floating-point limits	FLT_MAX, FLT_MIN
<signal.h>	Signal handling	signal(), raise()
<setjmp.h>	Jump functions (non-local goto)	setjmp(), longjmp()
<locale.h>	Localization	setlocale(), localeconv()

III. C variables declaration

A variable in C is a named storage location in memory that holds a value, which can be modified during program execution. Each variable has:

- A **name** (identifier).
- A **type** (defines the kind of data stored).
- A **memory location** (address).
- A **value** (the actual data stored).

Declaring a variable means telling the compiler what kind of data the variable will hold.

Examples: Syntax of variable declaration in C

Integer Variables (int)

```
int age;
int x, y, z; // You can declare multiple int variables in one line.
```

Floating-Point Variables (float or double)

```
float price;
double pi = 3.14159; /* Initialization is optional but can be
                        done at declaration.*/
```

Character Variables (char)

```
char Letter;
char symbol = '$'; /*You can initialize char variables at the
                     time of declaration.*/
```

The Rules for Naming Variable

- Must begin with a letter or underscore (_).
- Can contain letters, digits, and underscores.
- Case-sensitive (Age and age are different).
- Cannot use reserved keywords (int, float, if, etc.).

Examples: Rules for Naming Variable

Valid examples: int total, number1, value_1

Invalid examples: int 3total, float, my-number, my name, A+

Types of Variable Declaration:

a. **Local Variables**: Declared inside a function/block, and accessible only within that function/block.
b. **Global Variables:** Declared outside all functions, and accessible by all functions in the program.
c. **Static Variables**: Preserve their value even after the function exits.
d. **Extern Variables:** Declared with extern keyword, defined in another file or scope.

IV. Basic statements in C

In C programming, a statement is an instruction that tells the computer to perform a specific action. A program is essentially a sequence of such statements executed in order.

IV.1. Assignment in C

The assignment statement in C is used to store a value in a variable. It uses the assignment operator (=).

Examples: Assignment statement in C

Valid examples

```
int x;
x = 10;        // assign 10 to x
int y = 5;     // declare and assign
x = y + 3;     // assign result of expression
```

invalid examples

```
10 = x;    // Error: constant cannot be on left-hand side LHS
x + y = 5; // Error: expression cannot be on left-hand side LHS
```

In Table 5.2, the compound assignment operators in C offer convenient shortcuts for updating the value of a variable.

Table 5.2: Assignment Operators

Operator	Description	Example	Equivalent To
a += b	Adds b to a	a += b;	a = a + b;
a -= b	Subtracts b from a	a -= b;	a = a - b;
a *= b	Multiplies a by b	a *= b;	a = a * b;
a /= b	Divides a by b	a /= b;	a = a / b;
a %= b	Stores remainder of a / b	a %= b;	a = a % b;
a++	Post-increment: increases a by 1 (after use)	a++;	a = a + 1;
++a	Pre-increment: increases a by 1 (before use)	++a;	a = a + 1;

IV.2. Input statement (scanf)

The scanf function is a standard input function in the C programming language, and it is used to read formatted data from the standard input (usually the keyboard) into variables.

It is part of the standard input/output library (stdio.h).

Basic Syntax of input statement in C program is: scanf("%f", &Variable);

This is a C programming statement that reads a float value from standard input. Here's a detailed explanation:

a. **scanf() :** Standard input function that reads formatted input
b. **"%f"**: Format specifier for reading a **float** value :
   - %d: Reads an integer.
   - %f: Reads a floating-point number.

- %lf: Reads a double-precision floating-point number.
- %c: Reads a character.
- %s: Reads a string (sequence of characters excluding whitespaces).

c. **&Variable** : Address of operator (&) followed by the variable name. The & is crucial, it passes the memory address where the value should be stored.

IV.3. Output statement (printf)

The printf function in C is used for formatted output. It allows you to print data to the standard output with a specified format, basic syntax: printf("The value= %d",S);

a. "%d" is the format specifier for an integer (int).

b. If S is not an integer variable, this will cause undefined behavior. The program might crash, output garbage values, or behave unpredictably.

This is the most powerful feature of printf. You use **format specifiers** as placeholders in the string, and then provide the variables as arguments.

Table 5.3: printf function **format specifiers**

Format Specifier	Data Type	Example Code	Example Output
%d or %i	`int` (integer)	printf("%d", 42);	42
%f	`float`	printf("%f", 3.14);	3.140000
%lf	`double` (long float)	printf("%lf", 3.14);	3.140000
%c	`char` (single character)	printf("%c", 'A');	A
%s	`char*` (string)	printf("%s", "Hi");	Hi
%p	Pointer address	printf("%p", &var);	0x7ffc7a7c3a4
%x or %X	Integer as hexadecimal	printf("%X", 255);	FF
%o	Integer as octal	printf("%o", 8);	10
%%	To print a literal `%` sign	printf("100%%");	100%

Examples 1: Input/Output statement in C program

Code c

```
#include <stdio.h>

int main() {
   int age;
   printf("Enter your age: ");
   scanf("%d", &age);
   printf("You are %d years old.\n", age);
return 0;}
```

Output

Enter your age: **25**

You are **25** years old.

Examples 2: Input/Output statement in C program

Code c

```
#include <stdio.h>
int main() {
   float price;
  printf("Enter the price: ");
   scanf("%f", &price);
   printf("Price: $%.2f\n", price); // 2 decimal places
return 0;}
```

Output

Enter the price: **19.99334**

Price: $19.99

V. My First C Program

Let’s make your first C program more advanced by adding user input, calculations, and conditions. Here’s an example where the program asks the user for two numbers, performs arithmetic, and checks which number is larger.

Examples: First C program

Code c

```
#include <stdio.h>
int num1, num2;
int main() {
   // Ask the user to enter two integers
   printf("Enter the first number: ");
   scanf("%d", &num1);
   printf("Enter the second number: ");
   scanf("%d", &num2);

   // Display the sum, difference, product, and division
   printf("Sum: %d\n", num1 + num2);
   printf("Difference: %d\n", num1 - num2);
   printf("Product: %d\n", num1 * num2);
   if (num2 != 0) {  // Check to prevent division by zero
      printf("Division: %.2f\n", (float)num1 / num2); }
  else {
      printf("Division by zero is not allowed.\n"); }
   // Compare numbers
   if (num1 > num2) {
      printf("%d is greater than %d\n", num1, num2);}
   else{
      if (num1 < num2) {
        printf("%d is smaller than %d\n", num1, num2);}
      else {
          printf("Both numbers are equal.\n");}}
return 0;}
```

Output

```
Enter the first number: 10
Enter the second number: 5
Sum: 15
Difference: 5
Product: 50
Division: 2.00
10 is greater than 5
```

The programme teaches the following:

a. scanf : How to read user input.

b. **Type casting** : (float) num1 / num2 ensures division gives a decimal value.

c. **Conditionals** : if, else if, else to compare numbers.

d. **Arithmetic operations** : Addition, subtraction, multiplication, and division.

e. **Error handling**: Prevents division by zero.

VI. Conditional statements (if..else)

Conditional statements allow your program to **make decisions** based on certain conditions. In C, the most common ones are:

- if
- if...else
- if...else if...else
- Switch statement

VI.1. if statement

The `if` statement is a fundamental building block of decision-making in C. It allows your program to execute certain code only if a specified condition is true.

Syntax

```
if (condition) {
   // code to execute if condition is true
                 }
```

- if: The keyword that starts the statement.
- (condition): This is a logical or relational expression that is evaluated. If the result is **non-zero** (which means true in C), the code inside the block `{}` runs. If the result is **zero** (false), the code block is skipped.
- { }: The curly braces define a "block" of code. If your block contains only one statement, the braces are optional (but highly recommended for clarity and to avoid future bugs).

Examples: if statement

Code c

```
#include <stdio.h>
int num;
int main() {
num = 10;
   if (num > 5) {
      printf("Number is greater than 5\n");
                }
return 0;}
```

Output

```
Number is greater than 5
```

VI.2. if…else statement

Use this when you want to choose one of two paths.

Syntax

```
if (condition) {
   // code executes if condition is true
} else {
   // code executes if condition is false
}
```

Examples: if…else statement

Code c

```
#include <stdio.h>
int num
int main() {
   int num;
   if (num % 2 == 0) {
      printf("Number is even\n");
   } else {
      printf("Number is odd\n");
   }
 return 0;}
```

Output

Number is odd

VI.3. if…else if…else statement

Use this when you have multiple, mutually exclusive conditions to check.

Syntax

```
if (condition1) {
   // code executes if condition1 is true
   }
else{
    if (condition2) {
   // code executes if condition2 is true
                    }
   else {
   // code executes if none of the above are true
         } }
```

Examples: if…else if…else statement

Code c

```
#include <stdio.h>
Int num;
int main() {
   int num = 0;
 if (num > 0) {
       printf("Number is positive\n");
            }
Else{
      if (num < 0) {
       printf("Number is negative\n");
             }
      else {
       printf("Number is zero\n");
      }}
return 0;}
```

Output

Number is zero

VI.4. Switch statement

The switch statement is used when you want to choose one action out of many possible options based on the value of a variable or expression.
It's often cleaner than writing multiple if...else if statements.

Syntax

```
switch (expression) {
    case value1:
        // code to execute if expression == value1
        break;
    case value2:
        // code to execute if expression == value2
        break;
    ...
    default:
        // code to execute if no cases match
}
```

Examples: switch statement

Code c

```
#include <stdio.h>
int choice, a, b;
int main() {
   printf("Enter two numbers: ");
   scanf("%d %d", &a, &b);

   printf("Choose operation:\n");
   printf("1. Addition\n2. Subtraction\n3. Multiplication\n4.
Division\n");
   scanf("%d", &choice);

   switch (choice) {
      case 1:
         printf("Result: %d\n", a + b);
         break;
      case 2:
         printf("Result: %d\n", a - b);
         break;
      case 3:
         printf("Result: %d\n", a * b);
         break;
      case 4:
         if (b != 0)
            printf("Result: %.2f\n", (float)a / b);
         else
            printf("Error: Division by zero not allowed.\n");
         break;
      default:
         printf("Invalid choice!\n");
   }

   return 0;
}
```

Output

```
Enter two numbers: 10 5
Choose operation:
1. Addition
2. Subtraction
3. Multiplication
4. Division
3
Result: 50
```

VII. Iteration statement/Loop

Iteration statements, or loops, are fundamental to programming. They allow you to execute a block of code repeatedly until a specific condition is met.

C provides three primary loop constructs:

a. **for loop** : used when you know the exact number of iterations.
b. **while loop** : used when the number of iterations is unknown, depends on a condition.
c. **do...while loop :** similar to while, but executes at least once before checking the condition.

VII.1. for loop

The for loop is ideal when you know in advance how many times you need to iterate. It combines initialization, condition checking, and increment/decrement in a single line.

Syntax

```
for (initialization; condition; increment/decrement) {
    // Code to be executed repeatedly
}
```

Examples: for statement

Code c

```
#include <stdio.h>
Int i;
int main() {
  // i is the loop counter
  for ( i = 1; i <= 5; i++) {
     printf("%d\n", i);
    }
return 0;}
```

Output

```
1
2
3
4
5
```

In for loop we can using a multiple variables, can use commas to handle multiple variables (though it can reduce readability).

Syntax

```
for (initialization of multiple variables ; condition of multiple
variables ; increment/decrement of multiple variables) {
    // Code to be executed repeatedly
```

Examples: for multiple variables statement

Code c

```
#include <stdio.h>
int i;
int main() {
   for (int i = 0, j = 5; i < j; i++, j--) {
     printf("i = %d, j = %d\n", i, j);}
return 0;}
```

Output

```
0 5
1 4
2 3
```

VII.2. while loop

The while loop is used when you want to repeat a block of code an unknown number of times, as long as a condition remains true. The condition is checked before the loop body executes.

Syntax

```
while (condition) {
   // Code to be executed repeatedly
}
```

Examples: while statement

Code c

```
#include <stdio.h>
int number, sum
int main() {
   printf("Enter a number (0 to quit): ");
   scanf("%d", &number);
   sum = 0;
   while (number != 0) {
      sum += number;
      printf("Enter another number (0 to quit): ");
      scanf("%d", &number);}
   printf("The total sum is %d\n", sum);
   return 0;}
}
```

Output

```
Enter a number (0 to quit): 5
Enter another number (0 to quit): 3
Enter another number (0 to quit): 0
The total sum is 8
```

VII.3. do...while loop

The do-while loop is similar to the while loop, but with a crucial difference: it checks the condition after the loop body executes. This guarantees that the loop body will run at least once.

Syntax

```
do {
   // Code to be executed repeatedly
      } while (condition);
```

Examples: do..while statement

Code c

```
#include <stdio.h>
char choice;
int main() {
   do {
      printf("\nMenu:\n");
      printf("1. Play Game\n");
      printf("2. Load Game\n");
      printf("3. Quit\n");
      printf("Enter your choice: ");
      scanf(" %c", &choice); // Note the space before %c to skip whitespace
// Process the choice (simplified)
      switch (choice) {
         case '1': printf("Starting game...\n"); break;
         case '2': printf("Goodbye!\n"); break;
         default: printf("Invalid choice!\n"); }
  } while (choice != '3'); // Keep showing the menu until user chooses '3'
return 0;}
```

Output

```
Menu:
1. Play Game
2. Load Game
3. Quit
Enter your choice: 1
Starting game...
Menu:
1. Play Game
2. Load Game
3. Quit
Enter your choice: 5
Invalid choice!
Menu:
1. Play Game
2. Load Game
3. Quit
Enter your choice: 2
Goodbye!
```

VII.4. Break and Continue statement

These statements give you more control over the flow of a loop. Break, immediately terminates the innermost loop or switch statement it is in. Control passes to the statement following the loop.

Syntax

```
do {
   // Code to be executed repeatedly
Break ;
       } while (condition);
```

Examples: Brak statement

Code c

```
#include <stdio.h>
Int i;
int main() {
   for (int i = 1; i <= 10; i++) {
   if (i == 5) {
      break; // Exit the loop when i is 5}
   printf("%d\n", i);
  }
return 0;}
```

Output

```
1 2 3 4
```

These statements continue; Skips the rest of the current iteration of the loop and immediately proceeds to the next iteration (checking the condition in while/for, or updating in for).

yntax

```
do {
   // Code to be executed repeatedly
continue;
       } while (condition);
```

Examples: continue statement

Code c

```
#include <stdio.h>
int i;
int main() {
   for (int i = 1; i <= 5; i++) {
   if (i == 3) {
      continue; // Skip the print statement when i is 3
   }
   printf("%d\n", i);
}
return 0;}
```

Output

```
1 2 4 5
```

Table 5.4: Summary: Which Loop to Use

Loop	When to Use
for	When the number of iterations is known beforehand (e.g., iterating over an array, counting).
while	When the number of iterations is unknown and depends on a condition that is checked before entering the loop (e.g., reading input until a sentinel value is found).
do-while	When the number of iterations is unknown, but you need to ensure the loop body executes at least once (e.g., displaying a menu, validating user input).

VII. Activity

Establish an algorithm or a C program that checks whether a positive non-zero integer entered by the keyboard is a perfect number or not. A number is considered perfect if it equals the sum of its proper divisors.

Example 1:

N = 14, the proper divisors of 14 are 1, 2, 7.

The sum of divisors = 1 + 2 + 7 = 10 → so N is **not a perfect number**.

Example 2:

N = 28, the proper divisors of 28 are 1, 2, 4, 7, 14.

The sum of divisors = 1 + 2 + 4 + 7 + 14 = 28 → so N **is a perfect number**.

CONCLUSION

The Computer Science 1 module provides a solid foundation in the essential concepts of computer science. Through this course, students have learned the fundamental principles of computer operations, architecture, and the role of microcontrollers and microprocessors in modern computing.

The course also introduced students to data representation and encoding, including binary arithmetic, number systems, signed and unsigned numbers, and ASCII encoding, which are critical for understanding how computers process and store information.

Additionally, the study of Boolean algebra and logic gates equips students with the skills to design, analyze, and simplify digital logic circuits—an essential aspect of computer engineering and programming.

The module further familiarizes students with the basics of algorithm design and programming, emphasizing problem-solving techniques and the fundamentals of the C programming language. Practical activities throughout the course reinforce theoretical knowledge and develop students' analytical and technical skills.

By completing this course, students gain the ability to understand the inner workings of computers, represent and manipulate data efficiently, design logical circuits, and write basic programs. These competencies form a crucial foundation for advanced studies in computer science, digital systems, and software development.

Bibliography

[1] Shannon, C., & Weaver, W. (1949). *The mathematical theory of communication*. University of Illinois Press.

[2] Stallings, W. (2017). *Cryptography and network security: Principles and practice* (7th ed.). Pearson.

[3] Mano, M. M., & Kime, C. R. (2007). *Logic and computer design fundamentals* (4th ed.). Prentice Hall.

[4] Malgouyres, R., Zrour, R., & Feschet, F. (2012). *Initiation à l'algorithmique et à la programmation en C* (2e éd.). Dunod.

[6] Léry, J.-M. (2003). *Algorithmique en C* (2e éd.). Dunod.

[6] Tondo, C. L., & Gimpel, S. E. (2000). *Exercices corrigés sur le langage C* (2e éd.). Eyrolles.

Activity for chapter No. 2:

The numbers N=+10 is 8-bit signed integer

1) The 8-bit binary value of positive number N, in three representations: “signed and magnitude", "1's complement," and "2's complement" are given by:

$|N| = (1010)_2$

SM on 8bits(N) → N= $(0000\ 1010)_2$

1CP on 8bits (N) → N= $(0000\ 1010)_2$2

2CP on 8bits (N) → N= $(0000\ 1010)_2$

Result: N= $(0000\ 1010)_2$. In SM, 1CP and 2CP

2) The numbers **M= $(0000\ 1100)_2$** is 8-bit binary signed integer

A) Signed and magnitude of P=N-M

Calcule de P en SM 0000 1010 SM(N)

\-

0000 1100 SM (M)

P=$(1111\ 1110)_2$ **SM (P is a negative number)**

So: **|P|** =$(0111\ 1110)_2$

$P=0.2^0+ 1.2^1+1.2^2+1.2^3+1.2^4+1.2^5+1.2^6=2+4+8+16+32+64$

Result: P= - 126

B) 1's complement of P=N-M

0000 1010 **1 CP (N)**

\-

0000 1100 **1CP(M)**

P=$(1111\ 1110)_2$ 1 **CP (P is a negative number)**

So: |P| = $(0000\ 0001)_2$

$P=1.2^0$

Result: P= - 1

C) 2’s complement of P=N-M

0000 1010 **2CP (N)**

\-

0000 1100 **2CP (M)**

P=$(1111\ 1110)_2$ **2 CP (P is a negative number)**

$(1111\ 1110)_2$

\-

1

1111 1101 **CP1 (P)**

0000 0010 **SM (P)**

So : |P| est $(0000\ 0010)_2$

$P= 1.2^1$

Result: P= - 1

Activity for chapter No. 3:

$$F(x,y,z,w) = y\bar{z}\bar{w} + \bar{y}zw + \bar{x}\bar{y}z\bar{w} + \bar{x}yzw + \bar{x}yz\bar{w}$$

1) Simplification of the function *F(x,y,z,w)* :

A) Method of the laws and theorems of the Boolean algebra :

$F(x,y,z,w) = y\bar{z}\bar{w} + \bar{y}zw + \bar{x}\bar{y}z\bar{w} + \bar{x}yzw + \bar{x}yz\bar{w}$
$F(x,y,z,w) = y\bar{z}\bar{w} + \bar{y}z(w + \bar{x}\bar{w}) + \bar{x}yz(w + \bar{w})$
$F(x,y,z,w) = y\bar{z}\bar{w} + \bar{y}z(w + \bar{x}) + \bar{x}yz$
$F(x,y,z,w) = y\bar{z}\bar{w} + \bar{y}zw + \bar{x}\bar{y}z + \bar{x}yz$
$F(x,y,z,w) = y\bar{z}\bar{w} + \bar{y}zw + \bar{x}z(\bar{y} + y)$

Result:

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

B) Karnaugh Map method:

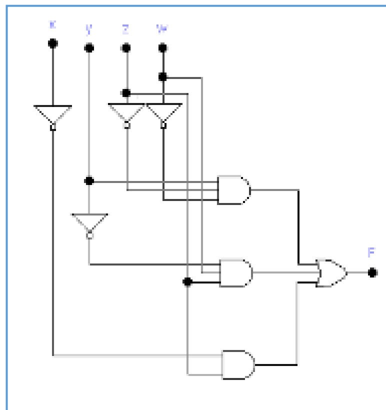
K-Map Table :

xy \ zw	00	01	11	10
00	0	0	1	1
01	1	0	1	1
11	1	0	0	0
10	0	0	1	0

Result

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

- **Logic circuits**



2) The 4-variable terms, when added to the initial function *F(x,y,z,w)* (do not simplify), In order to obtain terms with only two variables in the simplified function of G:

Adding two terms: $\boldsymbol{xyzw}$ **and** $\boldsymbol{xyz\bar{w}}$

Result:

$$\boldsymbol{G(x,y,z,w) = y\bar{w} + zw + \bar{x}z}$$

Activity for chapter No. 4:

Algorithm

Name: validated time

Input variables: m, h, s : Integer

Output variables:

Intermediate variables :

```
BEGIN
   read(h) ;
   read(m) ;
   read(s) ;
if(h<24 ) begin-if
            if(m<60) begin-if
                              if(s<60) begin-if
                              Ecrire ("is a valid time ")
                               end_si
                              else begin_else
                              Ecrire("is a valid time ")
                              end_else
             end_if
             else begin-if
             Ecrire("is a valid time ")
             end-if
end-if
else begin-if
             Ecrire("is an invalid time") ;
end-if
END.
```

Activity for chapter No. 5:

Code C

```
#include <stdio.h>

int I,N,S;
int main() {
do{
    printf("Give the number N : ");
    scanf("%d",&N);
    if(N<=0) {
    printf("Erreur \n");
             }
           }while(N<=0);
S=0
For(i=1 ;i<=N/2 ;i++){
   if(N%i==0){
      S=S+i
   }
}

if(N==S) {
    Ecrire ("le nombre % is perfect ",N) ;
   }
else{
Ecrire ("le nombre %d is not perfect ",N) ;
}
return 0 ;}
```