Module 1

Fundamentals of Computer Structure and Instruction Set

Learning Objectives

By the end of this module, learners will be able to:

1. Explain the basic types and characteristics of computers.
2. Identify and describe the main functional units of a computer system.
3. Understand and illustrate the fundamental operational concepts of a computer.
4. Describe the different bus structures and their roles in data transfer.
5. Interpret instruction formats and explain how instructions are represented.
6. Perform arithmetic operations on signed and unsigned numerical data.
7. Explain memory addressing methods and demonstrate read/write operations in memory.

Structure

1.1 Computer Types

Computers can be classified based on their data processing methods, size and performance, application scope, and integration within other systems. These classifications help in understanding the roles and limitations of each computer type in various contexts, from scientific research to household appliances and industrial machinery.

1.1.1 Classification of Computers: Analog, Digital, and Hybrid

Analog Computers:

Analog computers process data in a continuous form. They represent information using physical quantities such as voltage, pressure, or temperature. These computers are primarily used in engineering and scientific applications where data is measured in real-world terms. Analog computers are best suited for simulating systems like aircraft flight, nuclear power plant operations, and weather prediction. They are not precise in terms of exact numbers but are excellent for solving differential equations and performing real-time computations. The outputs are typically displayed using dials or graphs rather than numerical values. An example is a speedometer or the old-style flight simulator systems.

Digital Computers:

Digital computers process information in discrete binary form (0s and 1s). Unlike analog computers, they perform arithmetic and logical operations on data represented in numerical format. These are the most common types of computers used today, including desktops, laptops, and smartphones. Digital computers excel in accuracy and speed for general-purpose tasks such as word processing, data management, internet browsing, and complex scientific simulations. They use digital circuits and rely on microprocessors and memory units. Their flexibility and programmability make them suitable for both business and personal applications, across diverse sectors including finance, healthcare, education, and entertainment.

Hybrid Computers:

Hybrid computers combine the features of both analog and digital systems. They use analog components for processing continuous data and digital components for logical operations and memory storage. These systems are particularly valuable in specialized fields where both real-time data measurement and complex calculations are required. For instance, in hospitals, hybrid computers are used in intensive care units (ICUs) to monitor vital signs (analog input) and compute statistics or generate

alerts (digital processing). They are also used in industrial process control systems, scientific simulations, and aircraft control systems where both precision and real-time input/output handling are essential.

1.1.2 Micro, Mini, Mainframe, and Supercomputers

Microcomputers:

Also known as personal computers (PCs), microcomputers are designed for individual users. They are powered by a single microprocessor and include components like a monitor, keyboard, storage, and RAM. Microcomputers are highly versatile and are used for word processing, internet browsing, multimedia, and even programming. Their affordability and ease of use have made them ubiquitous in homes, offices, and schools. Variants include desktops, laptops, netbooks, and tablets. Despite their small size, modern microcomputers offer significant computing power, capable of handling multitasking environments, cloud applications, and even advanced software like video editing or coding platforms.

Minicomputers:

Minicomputers, or midrange computers, are more powerful than microcomputers but less powerful than mainframes. They are multi-user systems, capable of supporting dozens to hundreds of users simultaneously. Minicomputers are used in medium-sized businesses, research laboratories, and manufacturing plants. They are ideal for tasks requiring moderate computing power, such as database management, accounting, and process control. Unlike microcomputers, minicomputers often serve as servers in networked environments. Though the term is less commonly used today, their functionality has evolved into modern enterprise-grade servers and cloud computing nodes, bridging the gap between personal computers and large-scale computing systems.

Mainframe Computers:

Mainframe computers are high-performance machines designed to handle vast amounts of data and support hundreds or thousands of users at once. They are commonly used by large organizations such as banks, government agencies, and insurance companies for critical applications like transaction processing, bulk data handling, and enterprise resource planning. Mainframes offer exceptional reliability, security, and uptime, making them indispensable for mission-critical operations. They support multiple operating systems and can run thousands of virtual machines simultaneously. Their architecture focuses on I/O throughput rather than raw computational speed, distinguishing them from supercomputers.

Supercomputers:

Supercomputers are the fastest and most powerful type of computers, used for solving extremely complex scientific and mathematical problems. They are capable of performing trillions of calculations per second and are utilized in fields like climate modeling, quantum mechanics, nuclear research, and aerospace simulations. Supercomputers use thousands of interconnected processors and massive parallel processing techniques to achieve high performance. They often occupy entire rooms and require sophisticated cooling systems. Governments and research institutions invest heavily in supercomputers to advance scientific knowledge and national interests. Examples include systems like IBM’s Summit or China’s Sunway TaihuLight.

1.1.3 General-Purpose vs. Special-Purpose Computers

General-Purpose Computers:

General-purpose computers are designed to perform a wide variety of tasks. These machines can be programmed to carry out numerous operations such as word processing, browsing the internet, playing games, or running business software. Most personal computers and laptops fall into this category. Their versatility lies in their software-driven nature, allowing users to install and run different applications as needed. Educational institutions, businesses, and households rely on general-purpose computers due to their adaptability and cost-effectiveness. They typically use operating systems like Windows, macOS, or Linux, and their functionality can be extended through software and peripheral devices.

Special-Purpose Computers:

Special-purpose computers are built to perform a specific task efficiently. Unlike general-purpose computers, they are optimized for dedicated functions and cannot be easily reprogrammed for other uses. Examples include ATMs, GPS navigation systems, industrial robots, and digital watches. These systems are usually embedded within larger mechanical or electrical systems and are highly reliable, with performance tailored to the intended application. Because of their narrow scope, special-purpose computers are more efficient and faster in executing their assigned tasks than general-purpose systems. They are often used in real-time computing environments where accuracy and quick response are crucial.

1.1.4 Embedded Systems

Embedded systems are specialized computing systems that are integrated into larger mechanical or electrical systems to perform specific control functions. They consist of a processor, memory, and input/output interfaces, all embedded within a larger device. These systems are typically designed to perform a single or limited set of tasks with real-time computing constraints. Common examples include microwave ovens, washing machines, medical devices, automobiles, and industrial control systems. Unlike general-purpose computers, embedded systems are optimized for efficiency, reliability, and low power consumption. They often run on real-time operating systems (RTOS) and are critical for automation and control in a wide range of applications. Depending on complexity, embedded systems can be small and simple (like those in digital thermostats) or complex and powerful (like those in autonomous vehicles). Their importance continues to grow with the proliferation of the Internet of Things (IoT), where countless devices are interconnected through embedded computing.

1.2 Functional Units

A computer system is built upon a number of **functional units** that work together in a coordinated manner to execute tasks. These units are logically divided based on their roles, and each performs a distinct function in the **information processing cycle**: input, processing, storage, and output. Understanding these units is essential for comprehending how computers function at a fundamental level. The main functional units of a computer include:

- Input Unit
- Output Unit

- Memory Unit
- Arithmetic and Logic Unit (ALU)
- Control Unit
- Communication among Units

1.2.1 Input Unit

The **input unit** serves as the entry point through which data and instructions are fed into the computer system. It enables users to interact with the computer and provide the necessary information required for processing.

Key Functions of the Input Unit:

- **Data Acquisition:** Accepts raw data and user instructions from input devices.
- **Conversion:** Transforms input into a format (typically binary) suitable for internal processing.
- **Transmission:** Sends the converted data to the memory unit or processor.
- **Buffering:** Temporarily holds input data before it's processed, ensuring smooth data flow.

Common Input Devices Include:

- Keyboard
- Mouse
- Touchscreen
- Scanner
- Microphone
- Digital cameras
- Sensors (in IoT devices or embedded systems)

This unit is critical because the quality and format of input directly impact the accuracy of output. Moreover, advanced systems may use **speech recognition**, **gesture controls**, or **biometric input** for interaction, significantly extending the input capabilities of modern computing systems.

1.2.2 Output Unit

The **output unit** communicates the results of processed data from the computer to the external environment in a human-readable or usable form.

Main Roles of the Output Unit:

- **Data Conversion:** Converts binary or machine-readable results into forms understandable by humans (text, audio, visuals).
- **Presentation:** Displays or produces the result in physical or visual form.
- **Storage & Exporting:** Some output devices (like plotters or printers) can also permanently record outputs for physical use.

Common Output Devices Include:

- Monitor (Visual display)
- Printer (Hard copy output)
- Speakers (Audio output)
- Projectors
- Haptic devices (e.g., vibrating controllers)

Output devices are not just limited to showing the end result; they are also critical for real-time systems, such as **industrial monitoring**, **medical diagnostics**, and **control applications**, where rapid, clear, and accurate output is vital.

1.2.3 Memory Unit

The **memory unit** is responsible for **storing data**, both temporarily and permanently, and plays a central role in the computer's processing activities. All data, whether input, intermediate, or final results, must pass through memory.

Memory Hierarchy Includes:

1. **Primary Memory:**
   - **RAM (Random Access Memory):** Volatile memory used to store active data and instructions.
   - **ROM (Read-Only Memory):** Non-volatile, contains boot-up instructions and system firmware.
2. **Secondary Memory:**
   - Hard Disk Drives (HDDs)
   - Solid State Drives (SSDs)
   - Optical Disks (CDs, DVDs)
   - USB Drives and SD Cards
3. **Cache Memory:**
   - Located closer to the CPU than RAM.

- Stores frequently accessed data and instructions to speed up processing.

4. **Registers:**

- High-speed temporary memory inside the CPU.
- Hold data during instruction execution (operands, addresses, etc.).

Functions of the Memory Unit:

- Stores data and instructions before processing.
- Holds intermediate results of computations.
- Saves final output before sending it to the output unit.
- Retains essential system instructions (in ROM).

Efficient memory management is crucial for system performance, especially in **multitasking** and **real-time systems** where time-critical data access is required.

1.2.4 Arithmetic and Logic Unit (ALU)

The **Arithmetic and Logic Unit (ALU)** is the core of the **CPU (Central Processing Unit)** responsible for performing mathematical and logical operations.

Two Main Operations Performed by the ALU:

1. **Arithmetic Operations:**

- Addition, subtraction, multiplication, division.
- Extended functions like incrementing, decrementing, shifting, etc.

2. **Logical Operations:**

- Comparisons: equal to, less than, greater than.
- Boolean logic: AND, OR, NOT, XOR operations.

Internal Components of the ALU:

- **Accumulator Register:** Temporarily stores results of operations.
- **Flag Register:** Maintains status of the last operation (e.g., zero result, overflow, carry, etc.).

Functions and Significance:

- Receives data and instructions from memory and control unit.
- Processes the data based on specified operations.
- Sends results back to memory or registers.
- Essential for decision-making operations (branching, looping).

In modern processors, the ALU may be accompanied by **Floating Point Units (FPUs)** for high-precision arithmetic and **Vector Processing Units (VPUs)** in graphics and AI-intensive systems.

1.2.5 Control Unit

The **Control Unit (CU)** is the **coordinator and manager** of the computer system. Though it does not process or store data itself, it is responsible for **directing all operations** within the computer by interpreting and executing instructions.

Primary Responsibilities:

- **Instruction Fetching:** Retrieves instructions from memory.
- **Decoding:** Interprets the instruction type and operands.
- **Execution Control:** Directs ALU, memory, and I/O units to perform operations.
- **Timing and Sequencing:** Maintains execution order and timing to avoid conflicts.

Key Functions:

- Sends **control signals** to other components (e.g., "Read from memory", "Write to output").
- Coordinates the **fetch-decode-execute cycle**, which is fundamental to instruction processing.
- Manages **interrupts** and handles errors or unexpected events during execution.
- Ensures **synchronization** across all hardware components.

Types of Control Units:

- **Hardwired CU:** Uses fixed logic circuits; faster but less flexible.
- **Microprogrammed CU:** Uses firmware-level control; easier to update or modify.

Without the control unit, other components would not know when or how to operate, making it the brain that organizes the system's workflow.

1.2.6 Communication Between Units

The **communication among functional units** in a computer system is made possible through **interconnection systems** called **buses**. This communication allows data and control signals to flow between the processor, memory, and I/O devices efficiently.

Types of System Buses:

- **Data Bus:** Transfers actual data between components.
- **Address Bus:** Carries memory addresses from the processor to other components.

- **Control Bus:** Sends control signals (like read/write, interrupt, halt) from the CU.

Additional Concepts:

- **Bus Width:** Determines how much data can be transferred at once (e.g., 32-bit, 64-bit).
- **Clock Signals:** Synchronize data flow between units using a system clock.
- **Direct Memory Access (DMA):** Allows certain hardware subsystems to access memory independently of the CPU for faster data transfers.

Flow Example:

1. Input data is sent from the input unit via the data bus.
2. Control unit decodes the instruction and signals the ALU.
3. ALU performs the operation, using data fetched from memory.
4. Result is stored back in memory or sent to output.
5. All transfers occur over the bus system in a synchronized manner.

Efficient communication is essential for ensuring **data integrity**, **minimizing latency**, and **maximizing performance**, especially in modern high-speed computing environments.

1.3 Basic Operational Concepts

At the heart of every computer system lies a set of fundamental operational principles that govern how instructions are executed and how data flows between various units. These principles ensure that a computer performs its functions in a **systematic, synchronized, and predictable** manner. This section explores the **instruction processing cycle**, **data handling**, **execution steps**, and the **role of control signals** in enabling smooth system operation.

1.3.1 Fetch–Decode–Execute Cycle

The **Fetch–Decode–Execute Cycle**, also known as the **instruction cycle**, is the fundamental process through which a computer carries out program instructions stored in memory. It is a continuous loop that drives the entire functioning of the CPU.

1. Fetch Phase:

- The **Control Unit (CU)** retrieves the next instruction to be executed from **main memory (RAM)**.
- The location of the instruction is specified by the **Program Counter (PC)**, which keeps track of the memory address of the current instruction.
- The instruction is transferred to the **Instruction Register (IR)** for decoding.

2. Decode Phase:

- The CU analyzes the instruction stored in the Instruction Register.

- It identifies the **operation type**, **operands**, and the **location** of the required data (either from memory or registers).
- It sets up control signals for the required components (ALU, memory, I/O).

3. Execute Phase:

- The relevant component (e.g., the ALU) performs the operation—arithmetic, logical, data transfer, or branch/jump.
- Intermediate or final results may be stored in registers or memory.
- The PC is updated to point to the next instruction.

4. Repeat Cycle:

- The process repeats continuously until the program ends or a **halt instruction** is encountered.

This cycle is the **core of all computer processing**, allowing programs to run instruction-by-instruction in a well-defined sequence.

1.3.2 Data Flow between Units

In a functioning computer system, data constantly flows between input/output devices, memory, and processing units. This movement is **regulated and optimized** to ensure smooth instruction execution and system responsiveness.

Key Paths of Data Flow:

- **From Input Unit to Memory:**
  - Raw data entered by the user is first converted into binary and sent to **RAM** for temporary storage.
  - This data may later be accessed by the CPU for processing.
- **From Memory to CPU:**
  - The **CPU fetches instructions** and associated data from RAM.
  - Instructions go to the **Instruction Register**; data goes to CPU **registers** or **ALU** for processing.
- **Within the CPU:**
  - Data may flow between **registers**, **ALU**, and **Control Unit** as operations are performed.
  - Processed results are stored temporarily in CPU registers or sent back to memory.
- **From CPU to Memory/Output:**
  - Final results are sent to memory for storage or directed to the **Output Unit** (monitor, printer, etc.) for display or usage.

Mechanisms Facilitating Data Flow:

- **System Buses:** Used to carry data, addresses, and control signals.
    - Data Bus – transfers actual data.
    - Address Bus – sends location of data.
    - Control Bus – manages signal flow.
- **Registers:** Temporary storage within the CPU used for quick access.

Efficient data flow is crucial in high-performance systems and is enhanced through concepts like **pipelining**, **caching**, and **direct memory access (DMA)**.

1.3.3 Instruction Execution Steps

Executing an instruction involves multiple precise steps, beyond just the basic fetch-decode-execute cycle. This breakdown provides a closer look at the internal handling of each instruction.

Detailed Steps in Instruction Execution:

1. **Instruction Fetch:**
    - The **Program Counter (PC)** sends the address of the instruction to the memory.
    - The instruction is loaded into the **Instruction Register (IR)**.
2. **Instruction Decode:**
    - The CU interprets the opcode (operation code) and identifies what needs to be done.
    - It determines the operation type, operand sources, and destination.
3. **Operand Fetch (if needed):**
    - If the instruction requires data (operands), it is fetched from:
        - **Memory (RAM)**
        - **Registers**
        - **Input devices**
4. **Execution:**
    - The ALU performs the operation (e.g., ADD, SUB, COMPARE).
    - For non-arithmetic instructions, other components may be involved (e.g., data transfer, branching).
5. **Result Storage:**
    - The result is stored in a **register** or written back to **memory**.

- Output instructions may send data to an **output device**.

6. **Update Program Counter:**
    - The PC is incremented to point to the next instruction, unless altered by a branch/jump instruction.

Note: For instructions involving **interrupts** or **conditional branching**, special steps are inserted to check conditions or handle external signals.

1.3.4 Control Signals and Timing

Control signals and precise timing mechanisms are essential for orchestrating the activities of all computer components. These signals ensure that each operation happens at the right moment and in the right order.

What Are Control Signals?

- Binary signals generated by the **Control Unit** to direct the operations of the CPU and other components.
- Control signals manage operations like:
    - Read/Write (to/from memory or I/O)
    - Load (registers)
    - ALU Operation (Add, Subtract, etc.)
    - Increment Program Counter
    - Enable or disable buses

Types of Control Signals:

1. **Internal Control Signals:**
    - Direct the internal movement of data within the CPU (e.g., from register to ALU).
2. **External Control Signals:**
    - Interact with memory and I/O devices (e.g., memory read/write, I/O enable).
3. **Status Signals:**
    - Indicate the condition of devices or operations (e.g., ready, busy, error).

Timing and Synchronization:

- **Clock Signals:** Generated by a system clock to synchronize all actions in a computer.

    - The system operates in **clock cycles**, with each instruction potentially taking multiple cycles.
    - Faster clocks = faster instruction execution, but also more heat/power.
- **Instruction Timing:**
    - Simple instructions (like register transfer) may execute in one cycle.
    - Complex instructions (like memory access or floating-point operations) take multiple cycles.
- **Synchronous vs. Asynchronous Operations:**
    - **Synchronous:** Timed with the system clock (most modern CPUs).
    - **Asynchronous:** Operate based on events or readiness signals rather than clock pulses (used in I/O operations).

Precise control and timing are **critical for performance, accuracy, and reliability**, especially in modern multi-core processors and real-time computing systems.

1.4 Bus Structures

A **bus** is a communication pathway through which data and control signals are transmitted between various components of a computer system. The bus structure serves as the **backbone** of the system, enabling communication among the **CPU, memory, and input/output devices**. The efficiency and design of the bus directly affect the system's overall performance.

1.4.1 Data Bus, Address Bus, and Control Bus

The system bus is divided into three distinct parts, each with a specific role:

1. Data Bus

- **Function:** Carries actual data between the processor, memory, and I/O devices.
- **Bidirectional:** Allows data to flow both to and from the CPU.
- **Bus Width:** Determines the volume of data that can be transferred at once (e.g., 8-bit, 16-bit, 32-bit, 64-bit buses).
- **Impact:** Wider data buses allow faster and more efficient data handling.

2. Address Bus

- **Function:** Carries the memory addresses of data or instructions to be fetched or stored.
- **Unidirectional:** The CPU sends addresses to memory or I/O devices.
- **Address Space:** The width of the address bus determines the maximum addressable memory.

- E.g., a 32-bit address bus can address up to 2^{32}= 4 GB of memory.

3. Control Bus

- **Function:** Transmits control signals that regulate system operations.
- **Examples of Control Signals:**
  - Memory Read/Write
  - I/O Read/Write
  - Clock signals
  - Interrupt requests
  - Bus request and grant signals
- **Bidirectional or Mixed:** Some control signals are sent from CPU, others return as device responses.

Together, these buses form the communication infrastructure of the system.

1.4.2 Single Bus and Multiple Bus Organization

The **organization of buses** affects system throughput, cost, and scalability.

1. Single Bus Organization

- **Structure:** All major components share one common bus.
- **Advantages:**
  - Simple design
  - Lower cost
  - Easier to implement in small systems
- **Disadvantages:**
  - Performance bottlenecks due to contention
  - Slower data transfer, especially in high-traffic systems

2. Multiple Bus Organization

- **Structure:** Uses separate buses for different components or operations.
  - Examples: Dedicated memory bus, I/O bus, processor bus.
- **Advantages:**
  - Parallel data flow reduces congestion
  - Improved performance and scalability
- **Disadvantages:**

- More complex design
- Higher cost and power consumption

Multiple bus systems are common in high-performance computing and multi-core processors, where simultaneous data flows are critical.

1.4.3 Bus Arbitration and Control

In systems where multiple devices can initiate communication, **bus arbitration** is needed to decide **which device gets control** of the bus at a given time.

Why Bus Arbitration is Needed:

- Avoids **data collisions**
- Ensures **fairness and efficiency**
- Allows **multiple masters** (e.g., CPU, DMA controller) to share the bus

Types of Bus Arbitration:

1. **Centralized Arbitration**
   - A single controller (often part of the CPU) manages bus access.
   - Uses priority schemes (fixed, rotating, etc.)
   - Simpler to design but can be a bottleneck.
2. **Distributed Arbitration**
   - All devices participate in the decision-making.
   - Uses logic circuits to resolve conflicts.
   - More complex but avoids single point of failure.

Arbitration Techniques:

- **Fixed Priority:** Higher-priority devices always win.
- **Round-Robin:** Rotates priority in a circular fashion.
- **Dynamic Priority:** Adjusts priority based on context or history.

1.4.4 Bus Performance Considerations

Bus performance significantly affects the overall **speed and efficiency** of the computer system.

Key Factors Influencing Bus Performance:

1. **Bus Width**
   - Wider buses (e.g., 64-bit) transfer more data per cycle.
2. **Bus Clock Speed**
   - Faster clock rates increase data transfer speed.

3. **Data Transfer Protocol**
    - Determines how data is packaged and transmitted (synchronous vs. asynchronous).
4. **Bus Contention**
    - Too many devices requesting the bus simultaneously can lead to delays.
5. **Latency and Throughput**
    - **Latency:** Delay between request and transfer.
    - **Throughput:** Total volume of data transferred per unit time.
6. **Use of Buffers and Caches**
    - Reduces bus traffic by storing temporary data near the CPU.

Optimization Strategies:

- Implementing **dedicated buses** for high-traffic components
- Using **DMA (Direct Memory Access)** for fast transfers without CPU intervention
- Employing **bus multiplexing** to share lines for multiple purposes

1.5 Instruction Formats

Instruction formats define how instructions are represented in binary within a system's architecture. They determine **how the CPU interprets instructions**, extracts opcodes, and identifies operands. Proper formatting is essential for efficient decoding and execution of machine-level instructions.

1.5.1 Components of an Instruction

Each machine-level instruction typically contains the following fields:

- **Opcode (Operation Code):** Specifies the operation to be performed (e.g., ADD, SUB, LOAD).
- **Operand(s):** Data to be operated on, which may be immediate values, memory addresses, or register numbers.
- **Addressing Mode:** Specifies how to interpret the operand field (direct, indirect, immediate, etc.).
- **Instruction Length Indicator (in some architectures):** Specifies total bits/bytes used.

Example Breakdown:

Instruction: LOAD R1, 2000

- Opcode: LOAD
- Operand 1: R1 (destination register)
- Operand 2: 2000 (memory address to load data from)

1.5.2 Operation Code (Opcode) and Operand Specification

Opcode:

- The binary code that specifies the operation the CPU must perform.
- Each operation (add, subtract, move, etc.) has a unique code.

Operand Specification:

- Operand fields indicate the **data source**, **destination**, or **both**.
- Can refer to:
    - **Registers**
    - **Memory addresses**
    - **Immediate values (constants embedded in the instruction)**

Types of Operand Formats:

- **Zero-address:** Uses a stack for operands.
- **One-address:** Uses accumulator-based architecture.
- **Two-address:** Allows source and destination specification.
- **Three-address:** Fully specifies both sources and destination.

1.5.3 Types of Instruction Formats (Fixed, Variable, Hybrid)

1. Fixed-Length Format:

- All instructions have the same size (e.g., 32 bits).
- **Advantages:**
    - Simplifies decoding and pipelining
    - Fast instruction fetch
- **Disadvantages:**
    - May waste space if instruction is smaller than the fixed size

2. Variable-Length Format:

- Instruction size depends on operation and number/type of operands.
- **Advantages:**
    - More flexible
    - Space-efficient

- **Disadvantages:**
    - Slower decoding
    - Complicates pipeline design

3. Hybrid Format:

- Combines both fixed and variable instruction types.
- Used in modern architectures like x86.
- Offers flexibility while retaining performance for common instructions.

1.5.4 Example of Instruction Representation

Consider a **32-bit instruction format** used in a simplified architecture:

6 bits	5 bits	5 bits	16 bits
Opcode	Reg1	Reg2	Address/Immediate

Example Instruction:

Opcode: 000001 (LOAD)
Reg1: 00010 (R2)
Reg2: 00000 (Unused)
Address: 0000000000110100 (decimal 52)
Meaning: Load data from memory address 52 into register R2.
Another example in **assembly language:**
ADD R1, R2, R3
This means: Add contents of R2 and R3, store the result in R1.

- Opcode: ADD
- Operands: R2 and R3 (sources), R1 (destination)

1.6 Number Representation

Computers internally represent numbers using **binary systems**, which are efficient for digital processing. Various representations are used depending on the type of data, sign representation, and arithmetic requirements.

1.6.1 Binary, Octal, Decimal, and Hexadecimal Systems

Binary (Base-2):

- Uses two digits: 0 and 1.
- Each binary digit is called a **bit**.
- Fundamental to all digital computing.

Octal (Base-8):

- Digits range from 0 to 7.
- Used as a shorthand for binary (3 bits = 1 octal digit).

Decimal (Base-10):

- Digits from 0 to 9.
- Human-readable system; not used internally by computers but common for input/output.

Hexadecimal (Base-16):
- Digits: 0–9 and letters A–F (for 10–15).
- Compact representation of binary (4 bits = 1 hex digit).

Conversions:
- Binary to Decimal: Multiply each bit by 2^position.
- Binary to Hex: Group bits in 4s.
- Binary to Octal: Group bits in 3s.

1.6.2 Signed Magnitude Representation
- Used to represent both **positive and negative numbers**.
- The **most significant bit (MSB)** is reserved for the **sign**:
    - 0 for positive
    - 1 for negative
- The remaining bits represent the magnitude (absolute value).

Example (8-bit):
- +5 → 00000101
- -5 → 10000101

Pros:
- Simple and intuitive

Cons:
- Arithmetic operations are complex
- Two representations for zero: +0 and -0

1.6.3 1's and 2's Complement Representation
1's Complement:
- Inverts all bits of the binary number.
- MSB indicates sign: 0 for positive, 1 for negative.

Example (8-bit):
- +5 → 00000101
- -5 → 11111010 (1's complement of +5)

Drawbacks:

- Still has two representations of zero.
- Arithmetic needs end-around carry correction.

2's Complement:

- Take 1's complement and **add 1**.
- Most widely used in modern systems.

Example (8-bit):

- +5 → 00000101
- -5 → 11111011 (2's complement)

Advantages:

- Single representation for 0
- Simplifies subtraction and addition using same logic

1.6.4 Floating-Point Representation

Used to represent **real numbers** (fractions, very large/small numbers).
Standard: IEEE 754 (Single and Double Precision)
Format:

- **Sign Bit (1 bit)**
- **Exponent (8 bits for single precision)**
- **Mantissa or Significand (23 bits)**

Value = (-1)^Sign × 1.Mantissa × 2^(Exponent - Bias)
Example:

- 32-bit representation of a real number like 3.14

Pros:

- Large dynamic range
- Useful for scientific calculations

Cons:

- Rounding errors
- Complexity in hardware implementation

1.7 Arithmetic Operations on Signed and Unsigned Data

Understanding arithmetic operations is essential for both **low-level programming** and **hardware design**.

1.7.1 Addition and Subtraction

Unsigned Addition:

- Add binary values directly.
- Carry out if result exceeds bit width.

Signed Addition:
- Use **2's complement** for negative numbers.
- Add normally; interpret result accordingly.

Subtraction:
- Achieved by **adding the 2's complement** of the subtrahend.

Example (8-bit):
- 5 – 3 → Add 5 and 2's complement of 3
- 5 + (−3) = 2

1.7.2 Overflow and Underflow Conditions
Occurs when the result of an operation **exceeds** the range representable by the system.

Overflow (Addition/Subtraction):
- In unsigned: Carry out of MSB
- In signed: When signs of operands are same but result has opposite sign

Underflow (Floating-point):
- Result is too close to 0 to be represented

Detection:
- Hardware flags are set to signal overflow/underflow conditions.

1.7.3 Multiplication and Division
Unsigned Multiplication:
- Performed using repeated addition or shift-add algorithms.
- Result may be **double the operand size**.

Signed Multiplication:
- Operands are converted to 2's complement before operation.

Division:
- Involves repeated subtraction or long division method.
- Includes quotient and remainder.

Hardware Methods:
- Booth's algorithm (for efficient signed multiplication)
- Restoring and non-restoring division algorithms

1.7.4 Arithmetic Logic Unit Operations

The **ALU (Arithmetic Logic Unit)** is responsible for all arithmetic and logical operations:

Supported Operations:

- Addition, Subtraction
- Logical AND, OR, NOT, XOR
- Comparisons (equal, greater, less)
- Shifts (logical, arithmetic)

Flags Set by ALU:

- Zero Flag
- Sign Flag
- Overflow Flag
- Carry Flag

These operations are **central to instruction execution** and are optimized for speed and precision in modern processors.

1.8 Memory Locations and Addressing

Understanding how memory is organized and accessed is critical in programming and system architecture.

1.8.1 Concept of Memory Cells and Words

- Memory is organized into **cells** (smallest unit of storage).
- Each cell has a **unique address**.
- A **word** is a group of bits treated as a unit (commonly 8, 16, 32, or 64 bits).

Example:

- 32-bit word = 4 bytes

Memory Word Storage:

- Can be **byte-addressable** or **word-addressable**

1.8.2 Addressing Techniques (Immediate, Direct, Indirect)

1. Immediate Addressing:

- Operand is part of the instruction.
- Example: MOV R1, #5 (Load 5 directly)

2. Direct Addressing:

- Instruction specifies the memory address of the operand.

- Example: MOV R1, [1000]

3. Indirect Addressing:

- Address of the operand is stored in a register or memory location.
- Example: MOV R1, [R2] (R2 contains address)

Each addressing mode offers a trade-off between **execution speed**, **memory usage**, and **instruction complexity**.

1.8.3 Register and Indexed Addressing

Register Addressing:

- Operand is located in a **CPU register**.
- Fastest access method.
- Example: ADD R1, R2

Indexed Addressing:

- Combines a base address and an index value.
- Common in **array processing**.
- Example: MOV R1, [Base + Index]

Used extensively in **looping** and **data structure traversal**.

1.8.4 Effective Address Calculation

Effective Address (EA): The **actual memory address** used during instruction execution.
EA = Base Address + Offset (or Index)
Used in:

- Indirect and indexed addressing
- Dynamic memory access
- Array and pointer operations

Correct calculation ensures **accurate data fetching** during execution.

1.9 Memory Read/Write Operations

Reading and writing data to memory is a key aspect of computer functioning, involving precise control and timing.

1.9.1 Memory Read Cycle

Steps in a **Read Cycle**:

1. CPU places **address** on the address bus.
2. Sends a **Read** signal via control bus.
3. Memory places data on the data bus.

4. CPU reads the data and stores it in a register.

Read timing is critical to ensure valid data is captured.

1.9.2 Memory Write Cycle

Steps in a **Write Cycle**:

1. CPU places address and data on the buses.
2. Sends **Write** signal via control bus.
3. Memory writes the data into the specified location.

Memory write must be synchronized to avoid incorrect or partial writes.

1.9.3 Memory Access Time and Cycle Time

Memory Access Time:

- Time between the CPU requesting data and receiving it.

Memory Cycle Time:

- Time between two successive memory operations (read/write).

Lower times = faster performance.

Modern systems use **caches**, **pipelining**, and **burst access** to optimize these times.

1.9.4 Cache Memory and Performance

Cache Memory: Small, high-speed memory located between CPU and main memory.

Levels of Cache:

- **L1 Cache:** Closest to CPU core, fastest but smallest
- **L2 Cache:** Slightly slower, larger
- **L3 Cache:** Shared among cores, even larger

Purpose:

- Stores frequently used instructions and data
- Reduces memory access time
- Increases system throughput

Cache Strategies:

- Write-back vs. write-through
- Least Recently Used (LRU) replacement

Effective caching greatly enhances **CPU performance** and **overall system efficiency**.

1.10 Summary

This module offers a comprehensive overview of the fundamental concepts that define computer organization and architecture. It begins with the classification of computers into analog, digital, and hybrid types, and further categorizes them based on size and purpose—ranging from microcomputers to supercomputers, as well as

general-purpose, special-purpose, and embedded systems. Understanding the types of computers lays the foundation for exploring how computers function internally.
The functional units of a computer are at the heart of its operation. These include the input unit, output unit, memory unit, arithmetic and logic unit (ALU), and control unit. Together, these components facilitate the process of receiving data, processing it, storing it, and generating results. Their interaction is governed by precise control and data signals that flow through the system buses—namely the data bus, address bus, and control bus. Depending on the system architecture, computers may use single or multiple bus organizations, with arbitration mechanisms to manage access and ensure conflict-free communication.
Instruction execution is explained through the fetch-decode-execute cycle, which systematically processes program instructions stored in memory. The data flow between units during instruction execution is governed by control signals and timing logic. The instruction format itself is a key design element, involving opcodes, operands, and addressing modes. These can vary in length (fixed, variable, hybrid) and dictate how the CPU interprets and executes commands.
A critical aspect of computer design is the representation of data. The module discusses number systems such as binary, octal, decimal, and hexadecimal, and various ways to represent signed numbers, including signed magnitude, 1's and 2's complement, and floating-point notation. These representations are essential for enabling arithmetic operations, especially on signed and unsigned data. Arithmetic units perform operations like addition, subtraction, multiplication, and division while accounting for issues like overflow and underflow.
Memory organization is also thoroughly explored. Concepts like memory cells, words, and addressing techniques (immediate, direct, indirect, register, and indexed) are crucial for understanding how data is stored and accessed. Memory access involves specific read and write cycles, and performance is influenced by access time, cycle time, and the use of high-speed cache memory. The inclusion of cache significantly enhances system performance by reducing the latency involved in frequent memory accesses.
By providing a detailed examination of both hardware-level functions and data-level operations, this module equips learners with the foundational knowledge necessary for understanding advanced computer architecture, system programming, and hardware design principles. It builds a strong conceptual base for further studies in operating systems, digital logic design, and microprocessor architecture.

1.11 Keywords

1. **Bus:** A communication system that transfers data between components in a computer.
2. **Instruction Cycle:** The process by which a CPU fetches, decodes, and executes instructions.
3. **ALU (Arithmetic Logic Unit):** A digital circuit that performs arithmetic and logical operations.

4. **Addressing Mode:** The method used to specify operands for instructions in assembly or machine language.
5. **Cache Memory:** A small, fast memory used to store frequently accessed data and instructions.
6. **2's Complement:** A method of representing negative numbers in binary that simplifies arithmetic operations.

1.12 Self-Assessment Questions

1. Differentiate between digital and analog computers with examples.
2. Explain the function of the control unit in instruction execution.
3. What are the main differences between a data bus, address bus, and control bus?
4. Describe the steps involved in the fetch-decode-execute cycle.
5. How is a negative number represented using 2's complement?
6. What are the advantages of using cache memory in computer systems?

1.13 Case Study

Case Study: Hospital Monitoring System

A modern hospital ICU uses embedded systems to monitor patient vitals. Each monitoring device has a microcontroller that collects data like heart rate, blood pressure, and oxygen levels in real-time. This data is processed locally and sent to a central system for storage, alert generation, and long-term analysis. The system uses floating-point representation for precise measurement and applies 2's complement for signal processing. It includes multiple functional units, a dedicated bus structure for rapid communication, and relies heavily on memory management to store historical data.

Questions:

1. Identify and explain which computer types and number representations are used in the ICU monitoring system.
2. Describe how data flows between different functional units and memory in this scenario.

1.14 References

1. Mano, M. Morris. *Computer System Architecture*. Pearson Education.
2. Stallings, William. *Computer Organization and Architecture: Designing for Performance*. Pearson.
3. Hamacher, Vranesic, Zaky. *Computer Organization*. McGraw-Hill Education.

4. Patterson, David A., and Hennessy, John L. *Computer Organization and Design*. Morgan Kaufmann.
5. Tanenbaum, Andrew S. *Structured Computer Organization*. Pearson.
6. IEEE Standard 754-2019 for Floating-Point Arithmetic.
7. Hayes, John P. *Computer Architecture and Organization*. McGraw-Hill.