Basic Structure of Computers

In this chapter you will be introduced to:

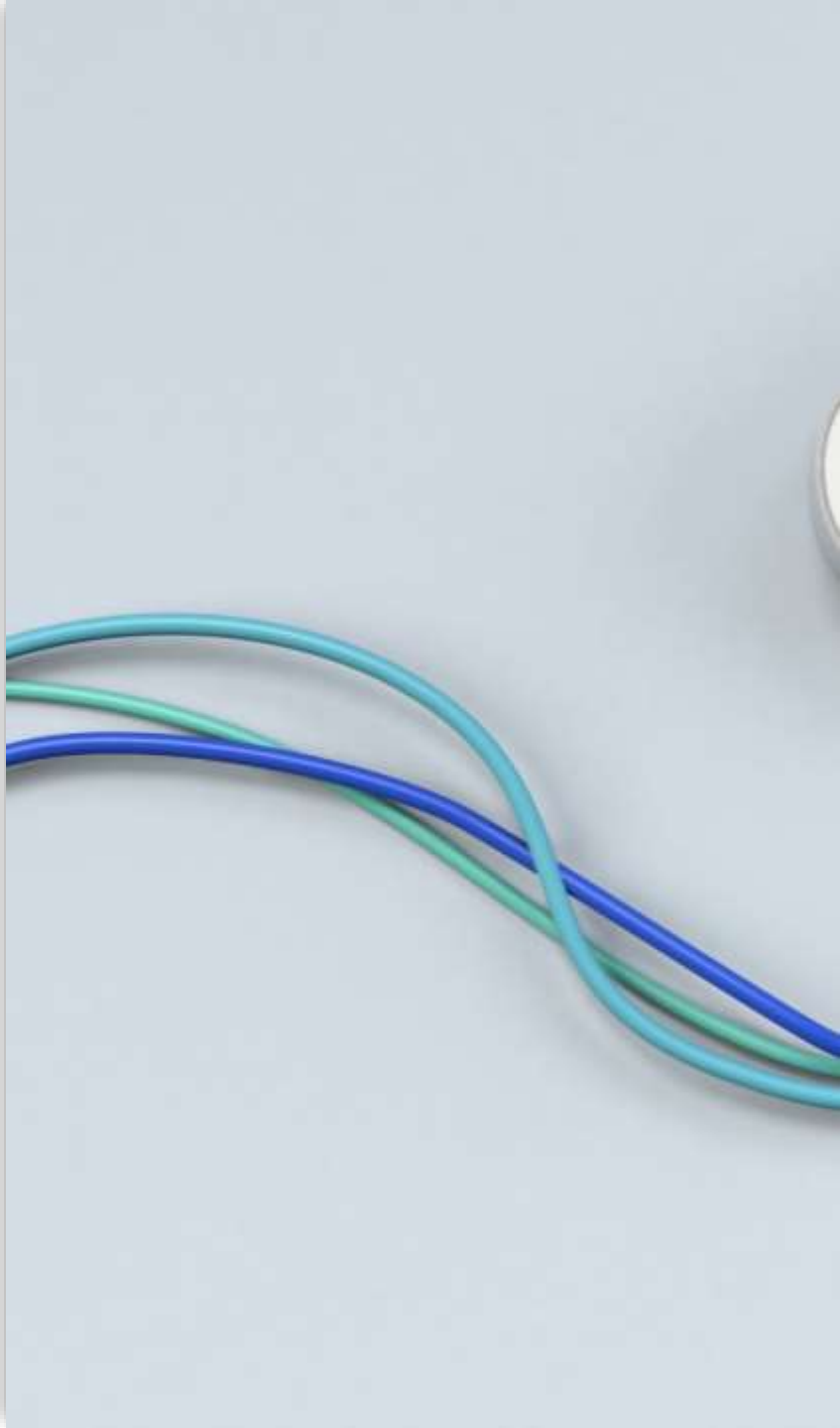
- In this chapter you will be introduced to:
- The different types of computers
- The basic structure of a computer and its operation
- Machine instructions and their execution
- Number and character representations
- Addition and subtraction of binary numbers
- Basic performance issues in computer systems
- A brief history of computer development

Detailed classification based on size, cost, performance, and purpose

Computer Types

Overview

- Since the 1940s, digital computers have evolved into various types differing in size, cost, computational power, and usage.
- Four main categories: Embedded, Personal, Servers/Enterprise, and Supercomputers/Grid computers.
- Emerging trend: Cloud computing for on-demand access to computing resources.



Embedded Computers

- Integrated into larger devices/systems to automatically monitor and control processes.
- Specific purpose rather than general computing.
- Applications: Industrial automation, home appliances, telecom products, vehicles.
- Often invisible to the user.

Personal Computers

- Widely used at home, in education, business, and engineering.
- Supports diverse applications: computation, document prep, CAD, entertainment, communication, Internet.
- Types:
- - Desktop: General use, fits in personal workspace.
- - Workstation: Higher computational and graphical power for engineering/science.
- - Portable/Notebook: Compact, battery-powered for mobility.

Servers and Enterprise Systems

- Large-scale computers shared by many users via network connections.
- Host large databases, process information for government or businesses.
- Provide centralized resources and services to connected devices.

Supercomputers and Grid Computers

- Supercomputers: Highest performance, used for complex simulations, weather forecasting, scientific research.
- Very expensive and physically large.
- Grid Computers: Cost-effective alternative combining many PCs and storage in a distributed high-speed network.
- Workload evenly distributed for large-scale applications.

Cloud Computing

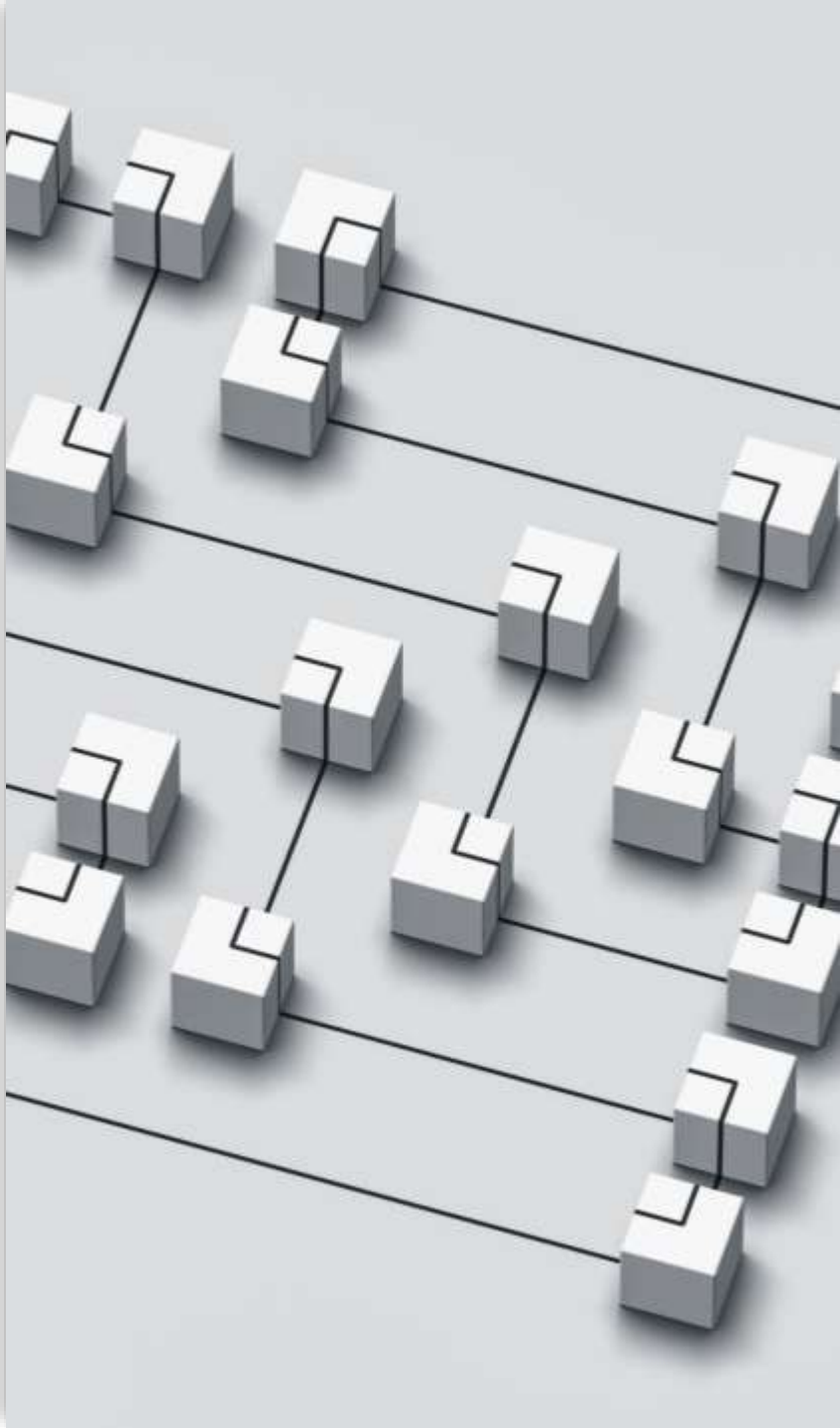
- Trend in accessing computing resources over the Internet.
- Users access distributed servers for independent computing needs.
- Operates as a utility—pay-as-you-use model.
- Examples: AWS, Microsoft Azure, Google Cloud.

Functional Units of a Computer

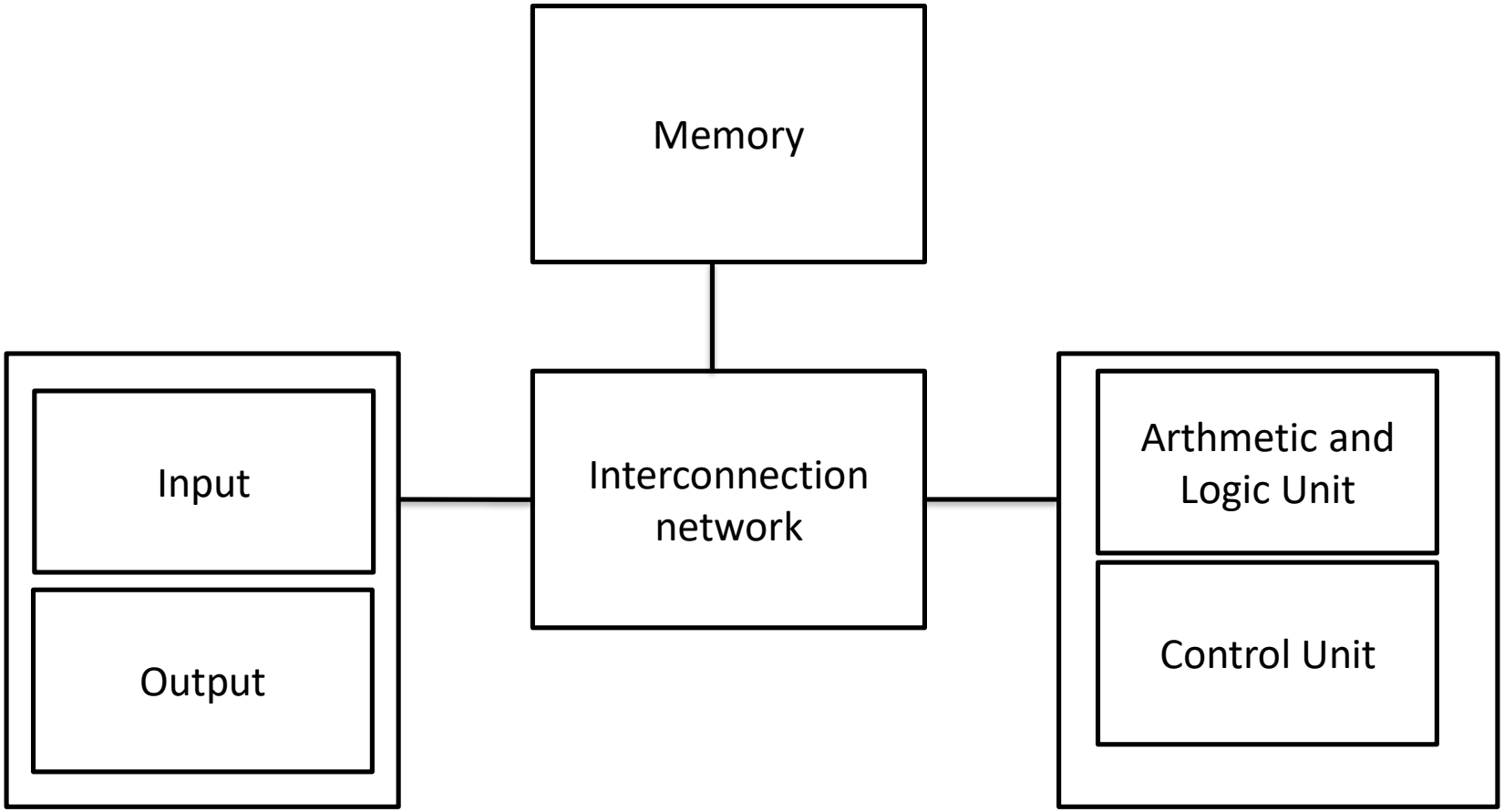
Five main parts: Input, Memory, ALU, Output, and Control Units

Overview

- Five functionally independent main parts: Input, Memory, Arithmetic & Logic, Output, and Control Units.
- Input Unit accepts coded information from users or other computers.
- Memory stores programs and data.
- ALU processes data according to program instructions.
- Output Unit sends results to the outside world.
- Control Unit coordinates all activities.
- Interconnection network allows information exchange.



Basic functional units of a computer



Input Unit

- Accepts coded information from human operators or other computers.
- Common device: Keyboard – converts key presses to binary codes.
- Other devices: Mouse, touchpad, joystick, microphone, camera.
- Can receive data via digital communication (e.g., Internet).

Memory Unit

- Stores programs and data.
- Primary Memory (RAM): Fast, volatile storage for active programs.
- Cache Memory: Smaller, faster RAM near the processor for frequently used data.
- Secondary Storage: Permanent, slower (HDD, SSD, optical disks, flash memory).
- Each memory location has a unique address for quick access.

Arithmetic & Logic Unit (ALU)

- Executes arithmetic operations (add, subtract, multiply, divide).
- Performs logical operations (comparison, AND, OR, NOT).
- Uses high-speed registers to store operands and results temporarily.
- Registers are faster than cache.

Output Unit

- Sends processed results to external devices.
- Examples: Printers, displays, speakers.
- Some devices combine input & output (touchscreens).
- Printers are slower compared to processor speed.

Control Unit

- Directs the operation of all other units.
- Sends control signals and senses states of units.
- Manages data transfers between processor, memory, and I/O devices.
- Generates timing signals for synchronization.
- Much control circuitry is distributed across the system.

Operation Summary

- Input Unit accepts data and stores it in memory.
- Memory holds both program instructions and data.
- ALU processes data as per program.
- Output Unit delivers results to the user.
- Control Unit coordinates and controls all operations.

Basic Operational Concepts

How instructions are fetched, executed, and how data flows in a computer

Instruction Execution – Simple Kitchen Analogy

- Computer as a Kitchen

- **Memory** = Fridge/Pantry – stores ingredients (data) and recipes (instructions)
- **Processor (CPU)** = Chef – reads recipes, fetches ingredients, cooks results
- **Registers** = Small bowls – temporary storage for quick access during cooking

Load Instruction *(Load R2, LOC)*

- Take ingredient from shelf **LOC** in pantry (memory) and put it into bowl **R2** (register)
- Pantry still has the ingredient, but the bowl now has a quick-access copy

Add Instruction

(Add R2, R3 get R4)

- Take ingredients from bowls **R2** and **R3**, mix them together
- Put the result into bowl **R4**
- **R2** and **R3** remain unchanged, **R4** gets the new mix

Store Instruction *(Store R4, LOC)*

- Take the ingredient from bowl **R4** and put it back on shelf **LOC** in pantry (memory)
- Bowl still keeps its copy, pantry now has the updated ingredient

How the CPU Executes Instructions

- CPU fetches instruction from memory (recipe step)
- Control unit figures out the action required
- CPU fetches required data into registers (bowls)
- ALU processes the data (cooking/mixing)
- Result stored in register or memory (fridge or pantry)

Connection between the processor and the main memory

MAR & MDR – Memory Communication Registers

- **MAR (Memory Address Register)**
- Holds the address of the memory location to be accessed.
- Used for both **read** and **write** operations.
- Works like a “pointer” telling the CPU where to look.
- **MDR (Memory Data Register)**
- Holds the actual data being transferred.
- During **read** → MDR receives data from memory.
- During **write** → MDR holds data to be stored in memory.
- Works as a temporary **data buffer** between CPU and memory.

- MAR = Where to look in Memory
- MDR = What data is being Moved

- **Instruction Register (IR)**
- Holds the instruction currently being executed.
- Passes instruction details to the **control unit** for decoding and execution.
- **Program Counter (PC)**
- Keeps track of the execution of a program.
- Stores the **memory address** of the next instruction to be fetched.
- Automatically increments after each instruction fetch.
- **Quick Tip for Remembering:**
- **IR = What to do now**
- **PC = Where to go next**

Program Execution Steps (Part 1) Program Execution – Fetch & Decode Phase

- **Step 1:** Execution starts when **PC** is set to the first instruction's address.
- **Step 2: PC** → **MAR** (Memory Address Register) → send **Read** signal to memory.
- **Step 3:** Data from memory → **MDR** (Memory Data Register).
- **Step 4:** MDR → **IR** (Instruction Register) to hold the current instruction.
- **Step 5:** Decode the instruction to understand the operation.

Program Execution Steps (Part 2) Execute & Store Phase

- **Step 6:** If instruction needs ALU operation → get operands:
- If operand is in a register → send to ALU directly.
- If operand is in memory → MAR ← address, Read cycle → MDR → ALU.
- **Step 7:** ALU performs operation.
- **Step 8:** Store result:
- To register, or
- To memory (**MAR** ← address, **MDR** ← data, Write cycle).
- **Step 9: PC** increments to point to the next instruction.

Bus Structures

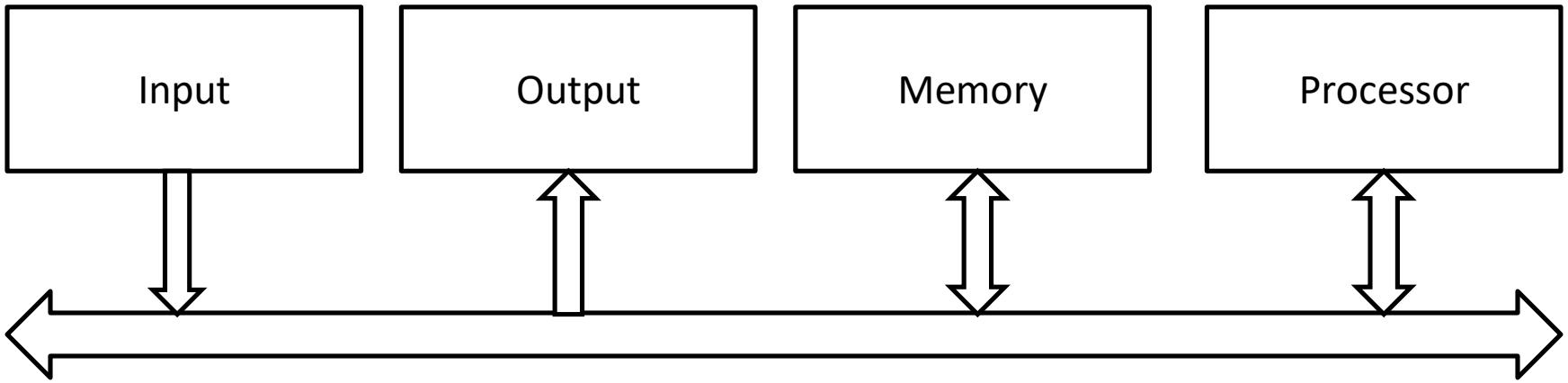
- To make a computer work, all its parts must be connected in an organized way.
- A **bus** is a group of lines that acts as a common pathway for communication between components.
- **Three types of bus lines:**

1.Data Bus – carries actual data being transferred.

2.Address Bus – carries the memory or device address involved in the transfer.

3.Control Bus – carries signals to control the operation (read/write, clock, etc.).

Bus Structures



- The simplest way to interconnect functional units is to use a single bus as shown in the above figure

Bus Structures

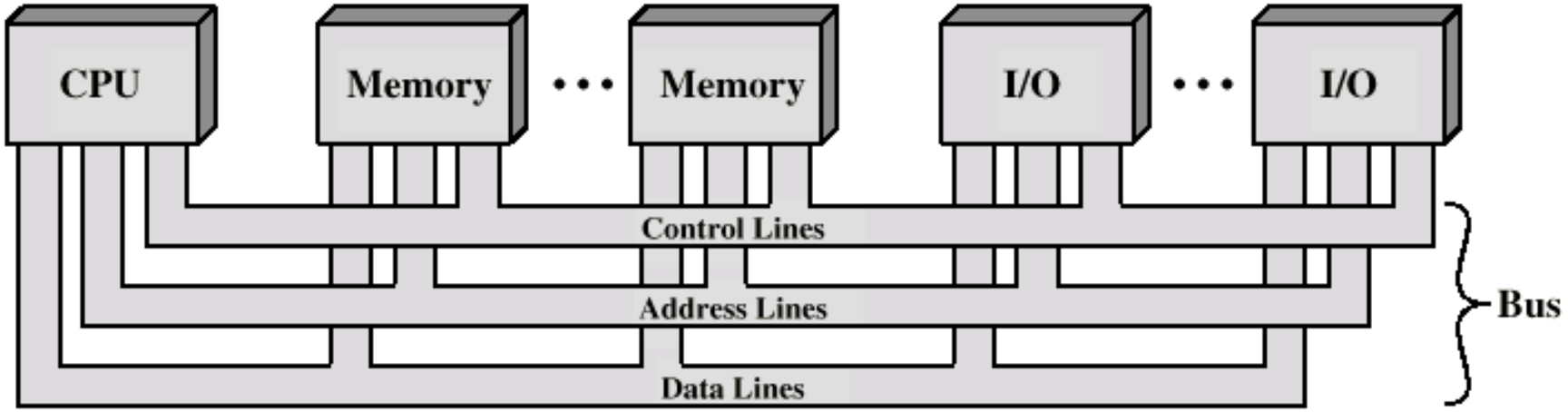
- All units (CPU, Memory, I/O) are connected to a **common bus**.
- Only **two units** can use the bus at a time because it supports only **one transfer at a time**.
- **Advantages:**
- Low cost.
- Flexible for connecting peripheral devices.
- For larger systems, **multiple buses** can be used to connect many devices.
- Devices connected to the bus vary in speed:
- **Slow devices:** keyboards, printers.
- **Fast devices:** memory, high-speed storage.

Bus Structures

•Memory and processor operate at high speeds.
•Peripheral devices (like I/O) are often slower.

Use **buffer registers** in devices to temporarily hold information during transfers.

This helps synchronize data movement between fast and slow components.



Bus Structures

- **CPU (Central Processing Unit)**
  - Executes instructions and controls the whole system.
- **Memory**
  - Stores programs (instructions) and data.
  - Multiple memory units may exist.
- **I/O (Input/Output) Devices**
  - Connect the computer to external devices like keyboards, displays, printers, etc.
- **Bus** – a shared communication path made of three sets of lines:
  - **Data Lines (Data Bus)**: Carry actual data between components.
  - **Address Lines (Address Bus)**: Specify the location (in memory or I/O) to read from or write to.
  - **Control Lines (Control Bus)**: Send control signals (e.g., read, write, clock) to coordinate actions.
- **Key point:**
- Only one transfer (CPU ↔ Memory, or CPU ↔ I/O) can happen on the bus at a time.
- This makes it simple and cost-effective, but limits speed if many devices need access simultaneously

Factors Affecting Computer Performance

- The most important measure of a computer is **how quickly it can execute programs**.

Execution speed depends on:

- Hardware design
- Machine language instruction set
- Efficiency of the compiler

Best performance is achieved when:

- Compiler, instruction set, and hardware are **designed in coordination**.

Factors Affecting Total Program Execution Time

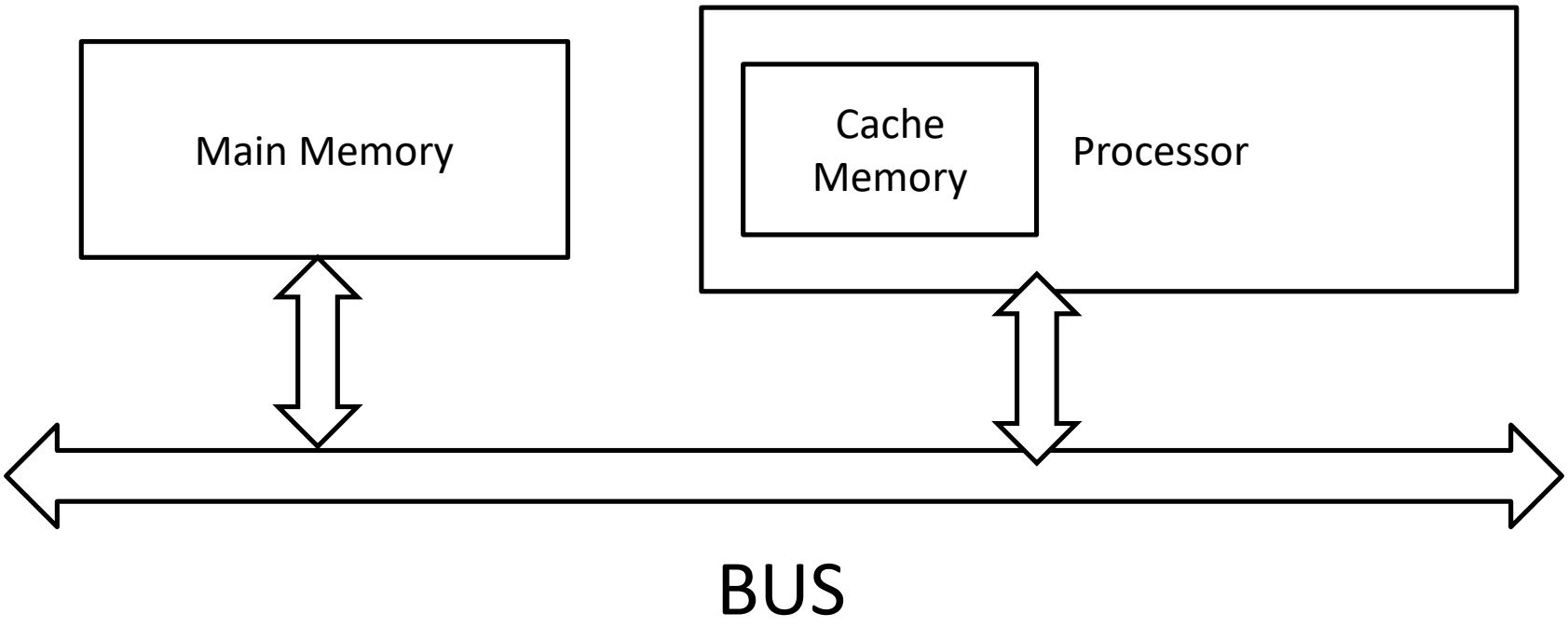
Total execution time depends on:

- Speed of the processor
- Compiler efficiency
- Disk speed
- Output device speed

Processor Time:

- Time needed to execute a single instruction.
- Depends on hardware involved in instruction execution.

Performance of Computer



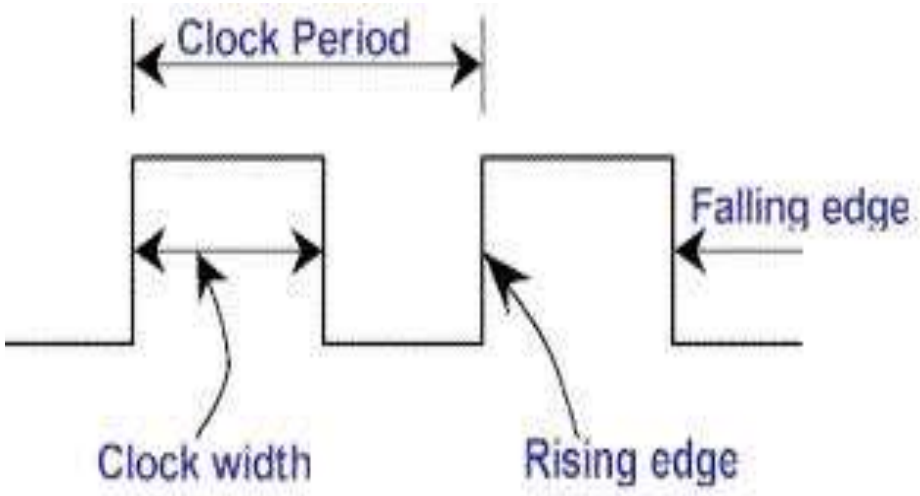
Role of Cache Memory in Program Speed

- Program execution is faster when data and instructions move less often between CPU and main memory.

Example:

- In a program loop, certain instructions are executed repeatedly in a short time.
- If these instructions are stored in **cache memory**,
- They can be fetched quickly.
- Reduces access time compared to main memory.
- Speeds up program execution.

Processor Clock



- Every electronic embedded system has a **processor circuit**.
- These are controlled by a **clock signal**.
- **Clock** defines regular time intervals → **clock cycles**.
- Each machine instruction consists of one or more actions.
- Example: ADD R0, R1
- Actions are divided into **basic steps**, each completed in **one clock cycle**.
- **Clock cycle length** = p
- **Clock rate**: R = 1 / p (cycles per second)
- Processor speed is given in **GHz** (Giga hertz).

Basic Performance Equation

$$T = \frac{N \times S}{R}$$

- **T –**time taken by the processor to execute the program written in high-level language
- **N –** number of actual machine language instructions in the program (needed to
- complete the execution (note: loop)
- **S –** average number of basic steps needed to execute one machine
- instruction. Each step completes in one clock cycle
- **R –** clock rate

Basic Performance Equation

- How to improve the performance parameter T?
- Solution: To achieve high performance. The computer designer must reduce the value of T, which means reducing N, & S and increasing R
- The value of ‘N’ is reduced if the source program is compiled into fewer machine instructions.
- The value of ‘S’ is reduced if instructions have a smaller number of basic steps to perform.
- Using higher- frequency clock increases the value of ‘R’. which means that the time required to complete a basic execution step is reduced.

$$T = \frac{N \downarrow \times S \downarrow}{R \uparrow}$$

Ways to Increase Clock Rate

- **Improve Integrating Circuits Technology**
- Faster logic circuits → shorter time to complete a basic step.
- Reduces clock period **p**, increases clock rate **R**.
- **Reduce Processing per Basic Step**
- Smaller workload per step → shorter clock cycle possible.
- Leads to higher clock rate.

Number Representation

- Computers use logic circuits operating on two-valued electrical signals: **0** and **1**.
- Numbers are represented as strings of bits → **binary numbers**.
- Text characters are represented as strings of bits → **character codes**.

Number Representation

- Computers must represent **both positive and negative numbers**.
- Three common systems:

1.Sign-and-Magnitude

2.1's Complement

3.2's Complement

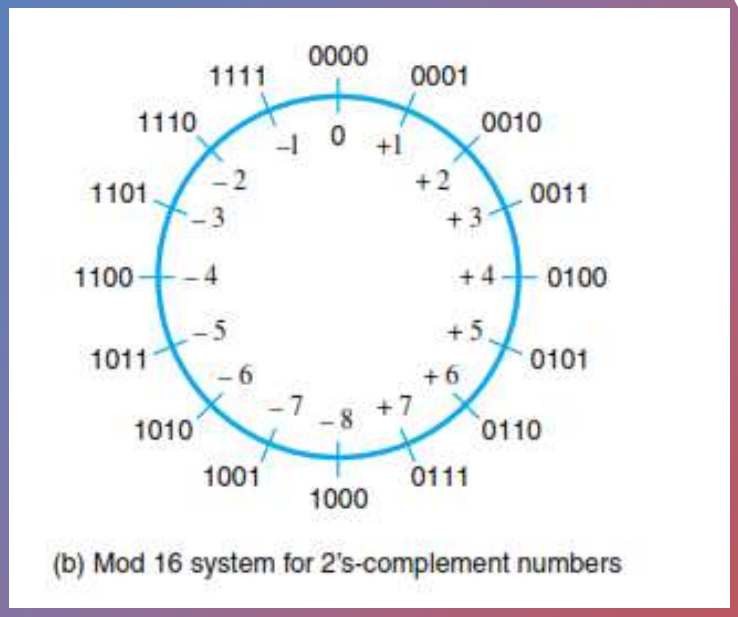
- In all three systems:
- **Leftmost bit = Sign bit**
  - 0 → Positive number
  - 1 → Negative number
- Positive numbers look the **same** in all systems.
- **Negative numbers** differ in representation between systems.

Sign-and-Magnitude Representation

- **Leftmost bit** = Sign bit
- 0 → Positive
- 1 → Negative
- **Negative number**: change the sign bit of the positive version
- Example:
- +5 → 0101
- -5 → 1101

1's Complement Representation

- **Negative number**: flip (complement) every bit of the positive version
- Change 0 → 1, and 1 → 0
- Example:
- +3 → 0011
- -3 → 1100



(b) Mod 16 system for 2's-complement numbers

2's Complement Representation

- **Negative number**: Take the 1's complement, then **add 1**
- Efficient for addition & subtraction
- Used in most modern computers
- Example:
- +3 → 0011
- 1's complement of 0011 → 1100
- Add 1 → 1101 (this is -3)

B	Values represented		
$b_3 b_2 b_1 b_0$	Sign and magnitude	1's complement	2's complement
0 1 1 1	+7	+7	+7
0 1 1 0	+6	+6	+6
0 1 0 1	+5	+5	+5
0 1 0 0	+4	+4	+4
0 0 1 1	+3	+3	+3
0 0 1 0	+2	+2	+2
0 0 0 1	+1	+1	+1
0 0 0 0	+0	+0	+0
1 0 0 0	−0	−7	−8
1 0 0 1	−1	−6	−7
1 0 1 0	−2	−5	−6
1 0 1 1	−3	−4	−5
1 1 0 0	−4	−3	−4
1 1 0 1	−5	−2	−3
1 1 1 0	−6	−1	−2
1 1 1 1	−7	−0	−1

Addition of Unsigned Integers

- 1. Addition of 1-bit numbers

•Possible cases when adding **two 1-bit numbers**:

A	B	Sum	Carry-out
0	0	0	0
0	1	1	0
1	0	1	0
1	1	0	1

Example: Adding two 3-bit numbers

- Let’s add: **101 (5 in decimal) + 011 (3 in decimal)**
- Step by step (right to left):
- **Bit 1 (rightmost):** 1 + 1 = 0 (Sum) with Carry = 1
- **Bit 2:** 0 + 1 + Carry(1) = 0 (Sum) with Carry = 1
- **Bit 3 (leftmost):** 1 + 0 + Carry(1) = 0 (Sum) with Carry = 1
- Result: **1000 (binary)** = 8 (decimal).

Special Case (All Ones with Carry)

- If both bits are 1 and the carry-in is also 1:
- 1 + 1 + 1 = **11 (binary)** → Sum = 1, Carry-out = 1.
- This equals **3 in decimal**.

Signed Number Representations

- There are **three systems** to represent positive and negative integers in binary:

Sign-and-Magnitude:

- Leftmost bit = sign (0 = positive, 1 = negative).
- Example (4 bits): +5 = 0101, −5 = 1101.

1’s Complement:

- Negative number = bitwise complement of positive number.
- Example (4 bits): +5 = 0101, −5 = 1010.

2’s Complement (used in modern computers):

- Negative number = 1’s complement + 1
- Example (4 bits): +5 = 0101, −5 = 1011.

2’s Complement Arithmetic

- For **n-bit numbers**, range = **-2^{n-1} to $+2^{n-1} - 1$**.
- Example (4 bits): –8 to +7.

Addition Rule

- To add two signed numbers:
- Add their binary representations.
- **Ignore carry-out** from the most significant bit.
- Example (4-bit):+7 + (−3)
- +7 = 0111
- −3 = 1101

0111

+ 1101

10100 (carry out ignored)

0100 → +4 ✓

Subtraction Rule

- To perform **X – Y**:
- Take **2’s complement of Y**, then add to X.
- Example (4-bit): 5 − 3
- 5 = 0101
- 3 = 0011
- 2’s complement of 3 → invert (1100) + 1 = 1101 (represents −3)

0101 (5)

+ 1101 (−3)

---

1 0010 (ignore carry out)

0010 → +2 ✓

Overflow

- Happens when the result is **outside the representable range**.
- Example (4 bits, range −8 to +7):

0100 (+4)

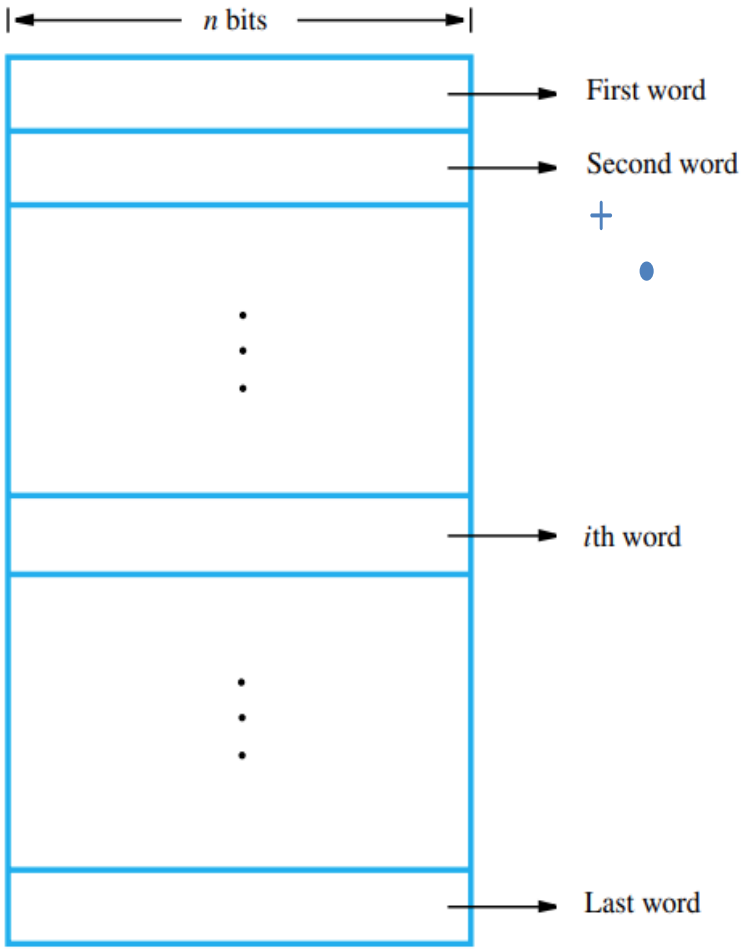
\+ 0101 (+5)

1001 (−7) ✕ Overflow

- Addition in 2’s complement = normal binary addition, ignore final carry.
- Subtraction = add 2’s complement of the second number.
- Efficient and simple → That’s why **computers use 2’s complement**.

Memory Locations and Addresses

- Memory consists of many millions of storage cells, each of which can store 1 bit.
- Data is usually accessed in *n*-bit groups. *n* is called word length.



Memory Locations and Addresses

- **Bits and Words**
- Memory is made up of millions of **storage cells**, each cell storing **1 bit** (0 or 1).
- Since a single bit is too small to be useful, computers group them into fixed-size chunks called **words**.
- The number of bits in one word is the **word length**.
  Example: A 32-bit word means each memory word is 32 bits wide.
- **Bytes**
- A **byte** is always **8 bits**.
- Words can be broken down into bytes.
  Example: A 32-bit word = 4 bytes.
- **What can fit in a word?**
- If the word length is **32 bits**, you could store:
  - One 32-bit integer, or
  - Four ASCII characters (each 8 bits = 1 byte).

Memory Location and addresses

- **Addresses**
- Each memory word (or byte) needs a unique name so the CPU knows where to find it. These names are called **addresses**.
- Addresses are usually numbers starting from **0 up to $2^k - 1$**, where *k* is the number of bits in the address.

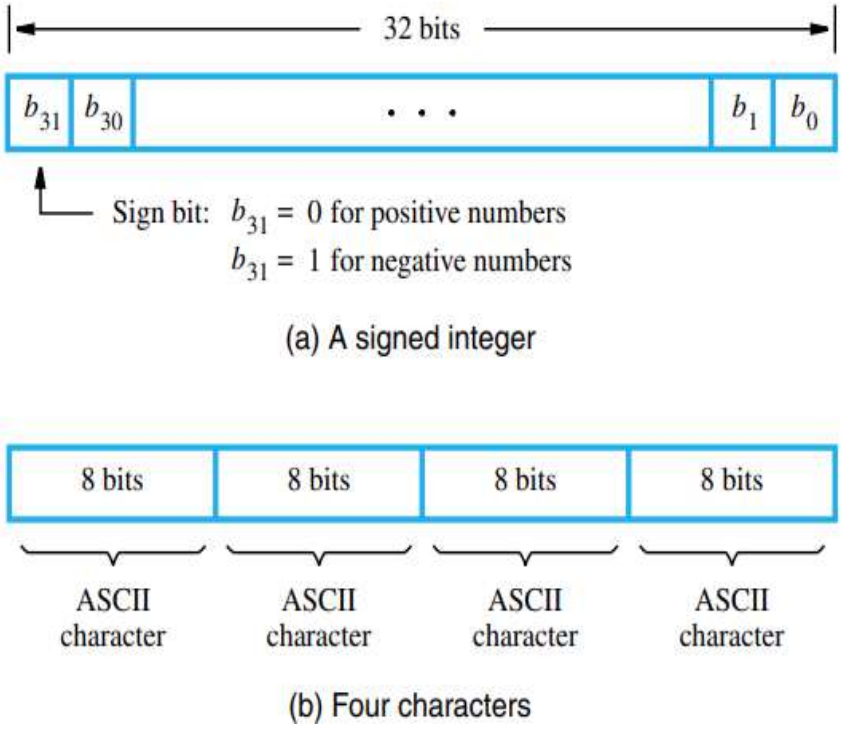
Example:

  - If k = 24 bits → address space = 2^{24} = 16,777,216 locations (16M).
  - If k = 32 bits → address space = 2^{32} = about 4 billion locations (4G).

- **Notation**
- **K (kilo)** = 2^{10} = 1024
- **M (mega)** = 2^{20} ≈ 1 million
- **G (giga)** = 2^{30} ≈ 1 billion
- **T (tera)** = 2^{40} ≈ 1 trillion

32-bit word length example

- **More detail:**
- A **bit** is too small to work with directly.
- A **byte** (8 bits) is the minimum practical unit but CPUs usually like to process more than tha at once.
- So we define a **word**:
  - On a **16-bit machine**, one word = 16 bits (2 bytes).
  - On a **32-bit machine**, one word = 32 bits (4 bytes).
  - On a **64-bit machine**, one word = 64 bits (8 bytes).
- So **the size of a word depends on the architecture** of the computer.



Byte Addressability

- **Bit** = smallest unit (0 or 1).
- **Byte** = 8 bits (always).
- **Word** = machine-dependent, usually 16–64 bits (2–8 bytes).
- Bits are **not individually addressable** (impractical).
- Modern computers use **byte-addressable memory** → each address points to 1 byte.
- Addresses: 0, 1, 2, 3, … (each one is a byte).
- Example: On a **32-bit machine** (word = 4 bytes), words start at addresses **0, 4, 8, 12, ….**

BIG-ENDIAN and LITTLE-ENDIAN Assignments

1. What does endianness mean?

- When a computer stores a **word** (e.g., 32 bits = 4 bytes) in memory, it has to decide:
- Which byte goes in the **lowest memory address**?
- Which byte goes in the **highest memory address**?
- That choice is called **endianness**.

Big-Endian

Address	Stored Byte	Meaning
1000	12	Most significant byte
1001	34	Next
1002	56	Next
1003	78	Least significant byte

- **2. Big-Endian**
- **Definition**: The **most significant byte (MSB)** of the word is stored at the **lowest address**.
- "Big end first" → you put the most important part first.

Little-Endian

Address	Stored Byte	Meaning
1000	78	Least significant byte
1001	56	Next
1002	34	Next
1003	12	Most significant byte

- **3. Little-Endian**
- **Definition**: The **least significant byte (LSB)** of the word is stored at the **lowest address**.
- "Little end first" → you put the least important part first.

Problem 1: Word alignment in byte-addressable memory

A computer has **32-bit words** (4 bytes per word) and is byte-addressable.

What addresses do the first 5 words start at?

- **Solution:**
- Word 0 → bytes at 0, 1, 2, 3 → starts at **address 0**
- Word 1 → bytes at 4, 5, 6, 7 → starts at **address 4**
- Word 2 → bytes at 8, 9, 10, 11 → starts at **address 8**
- Word 3 → starts at **12**
- Word 4 → starts at **16**

Problem 2: Big-endian vs Little-endian storage

Problem 2: Big-endian vs Little-endian storage

Store the 32-bit value 0x12345678 at starting address 1000. Show the memory contents for both schemes.

Solution:

- **Big-endian**:
  - 1000 → 12
  - 1001 → 34
  - 1002 → 56
  - 1003 → 78
- **Little-endian**:
  - 1000 → 78
  - 1001 → 56
  - 1002 → 34
  - 1003 → 12

Problem 3: Accessing a single byte

Problem 3: Accessing a single byte

A 32-bit little-endian system stores the word 0x87654321 starting at address 2000.

- What is the content at address 2000?

Solution:

- Little-endian → lowest address gets LSB (21).
- So:
  - 2000 → 21
  - 2001 → 43
  - 2002 → 65
  - 2003 → 87
- Answer: Content at address 2000 = **0x21**.

Word Alignment

- **Word alignment** is about how data (words) are placed in memory.
- A **word** is just the natural unit of data a processor handles (could be 16, 32, or 64 bits depending on the machine).
- Memory, on the other hand, is byte-addressable—each byte has its own address: 0, 1, 2, 3, …
- Now the key point:
- If a word is **aligned**, it starts at a memory address that is a multiple of its size.
  - Example: On a **32-bit machine** (4 bytes per word), aligned word addresses are 0, 4, 8, 12, … because each word needs 4 bytes.
  - On a **16-bit word system** (2 bytes per word), aligned addresses are 0, 2, 4, 6, …
  - On a **64-bit word system** (8 bytes per word), aligned addresses are 0, 8, 16, 24, …
- If a word is **unaligned**, it starts at some odd or non-multiple address.
  - Example: a 32-bit word stored starting at address 2 would span addresses 2, 3, 4, 5. That's *unaligned*.

Word Alignment

- **Why does alignment matter?**
- Hardware is optimized to fetch words starting at aligned addresses.
- If data is unaligned, the CPU may need **two memory reads** instead of one (since the word spans across boundaries). That slows things down.
- Some processors even **disallow unaligned access** completely and throw an error. Others allow it, but it's slower.
- So the short version:
- Aligned = word starts at a clean boundary (multiple of word size).
- Unaligned = word starts at an arbitrary address, less efficient.

Accessing Numbers and Characters

• In memory, **numbers** (like integers or floating-point values) are usually stored in one full **word**.

- •On a 32-bit machine, that means 4 bytes.
- •To fetch that number, you just give its **word address** (like 0, 4, 8, …).

•**Characters**, on the other hand, are typically stored in **bytes**.

- •So each character has its own **byte address** (0, 1, 2, 3, …).
- •For example, the string HELLO would take 5 consecutive byte addresses.

What this means in practice:

•If your program says “load word at address 8,” the hardware knows to grab the 4 bytes starting at memory address 8.

•If your program says “load byte at address 10,” the hardware just fetches that single character (1 byte).

And for programmers, it’s convenient to be able to use **different addressing modes**:

•Sometimes you care about the word (when working with numbers).

•Sometimes you care about the byte (when working with characters or strings).

That’s why instruction sets usually let you specify addresses at both levels.

Example

- Imagine you have a 32-bit system (so one **word = 4 bytes**) and the memory looks like this:
- Now here's how this plays out:
- If a **program wants the word at address 1008**, it gets bytes 1008–1011 together. That's one full 32-bit integer = 0x0000000A (decimal 10).
- If the program wants the **byte at address 1002**, it just pulls the value at that single address, which is 0x4C, i.e., the character 'L'.
- So in real terms:
- Your **string "HELLO"** is stored byte by byte, each character with its own byte address.
- Your **integer 10** is stored as one 32-bit word, accessed by its word address (1008).
- This is why processors let you say **load byte** or **load word** depending on what kind of data you're dealing with.

Address	Content (in hex)	Meaning if seen as...
1000	48	'H' (character)
1001	45	'E'
1002	4C	'L'
1003	4C	'L'
1004	4F	'O'
1005	00	(null terminator for string)
1006	00	—
1007	00	—
1008	00	First byte of an integer
1009	00	Second byte
1010	00	Third byte
1011	0A	Fourth byte (together these 4 bytes = integer 10)

Memory Operations

The basics

Memory really only supports **two fundamental operations**:

1.Read – copy data from memory → processor (nothing changes in memory).

2.Write – copy data from processor → memory (old contents are overwritten).

How it actually happens

•**Read example**: Suppose the processor wants to add two numbers stored in memory.

1. It sends the **address** (say 2000) to memory.
2. Memory fetches the content at address 2000 (let's say the number 25).
3. That value is sent to the processor's registers.
4. The original 25 is still in memory — memory wasn't modified.

•**Write example**: After the processor computes, say 25 + 5 = 30, it may want to store the result back.

1. It sends the **address** (say 3000) along with the **data** (30).
2. Memory writes 30 into location 3000, overwriting whatever was there before.

Memory Operations

- **Real-life analogy**
- Think of memory as a big **notebook**:
- **Read** = you look at page 200 and copy what's written there onto your scratchpad (the scratchpad = processor registers). The notebook itself doesn't change.
- **Write** = you take your pen and overwrite what's on page 200 with something new. Now the notebook has changed.

Instructions and Instruction Sequencing

- A **program** is really nothing more than a long list of **instructions**. Each instruction tells the processor to do a very small, specific step. When you chain thousands or millions of these steps, you get meaningful work done (like running Excel, playing a video, or training a model).
- Now, instructions typically fall into four big categories:

1. Data transfer

- Moving data between **memory** and **registers** (the small, super-fast storage inside the CPU).
- Example:
  - LOAD R1, 2000 → fetch data from memory location 2000 into register R1.
  - STORE R1, 3000 → send the contents of R1 back into memory at address 3000.

This is the “Read/Write” stuff we discussed earlier.

2. Arithmetic and logic

- Once the data is in registers, the CPU can actually compute.
- Examples:
  - ADD R1, R2, R3 → R1 = R2 + R3.
  - AND R1, R2, R3 → R1 = bitwise AND of R2 and R3.
  - NOT R1, R2 → R1 = logical NOT of R2.

These are the math and decision-making tools.

3. **Program sequencing and control**

- This is about deciding **what instruction comes next**.
- Normally, instructions run one after another (sequential execution).
- But sometimes you need to **branch, loop, or make decisions**:
  - JUMP 4000 → next instruction is at address 4000.
  - BEQ R1, R2, 5000 → if R1 == R2, branch to 5000; otherwise continue normally.

This is how you get **if-else, loops, and function calls** in higher-level languages.

4. Input/Output transfers

- Computers also need to talk to the outside world (keyboard, screen, disks, network).
- Special instructions handle data transfer between CPU and **I/O devices**.
- Example:
- IN R1, Keyboard → read a character from keyboard into R1.
- OUT Display, R1 → send contents of R1 to the display.

Register Transfer Notation

- **Locations where data can live**:
- **Memory locations** → identified by symbolic names like LOC, PLACE, VAR2.
- **Processor registers** → identified by names like R0, R1, R5.
- **I/O registers** → identified by names like DATAIN (keyboard input), OUTSTATUS (screen output status).
- **Square brackets [ ] mean contents**
- [LOC] → the value stored at memory location LOC.
- [R2] → the value currently inside register R2.
- **Left-hand side = destination**
- Example:

R2 ← [LOC]

→ Copy the contents of memory location LOC into register R2.

→ After this, R2 has new data, but LOC is unchanged.

Register Transfer Notation

- **Operations can be combined**
- Example:
  R4 ← [R2] + [R3]
  → Add the contents of registers R2 and R3, then store the result into R4.
  → Old contents of R4 are overwritten.
- **Important nuance**
- "Transfer" or "Move" in computer jargon **does not destroy the source**.
- It's always a **copy**. Source stays the same; destination gets overwritten.

Assembly-Language Notation

- **Assembly language is notation for machine instructions**
- Every CPU instruction has an equivalent assembly form.
- Instead of binary codes, we use words and symbols.
- **Example 1: Load instruction**
- **Load R2, LOC**
- This tells the CPU: *"Take the contents of memory location LOC and copy them into register R2."*
- LOC (memory) is unchanged.
- The old value in R2 is overwritten.
- The word **Load** makes sense because data is being loaded *into* the CPU register.

Assembly Language Notation

- Example 2: Add instruction
- Add R4, R2, R3
- This means: *"Take the contents of R2 and R3, add them, and put the result into R4."*
- Here, R2 and R3 are **source operands**, and R4 is the **destination register**.

Assembly Language Notation

- **Instruction = Operation + Operands**
- Every instruction has:
  - **Operation** (what to do → Load, Add, Store, etc.)
  - **Operands** (where the data comes from or goes to → registers, memory locations).
- **Mnemonics**
- Real processors don't actually use English words like Load or Add.
- Instead, they use short mnemonics, e.g.:
  - Load → LD
  - Store → ST or STR
  - Add → ADD
- Different processors (Intel, ARM, MIPS) may use different mnemonics for the same idea.

Assembly Language Notation

Assembly language is just a **human-readable shorthand for machine instructions**. Instead of writing 1010 0001 0100 ... in binary, you write Load R2, LOC. Mnemonics are abbreviations that keep it concise but still readable.

RISC and CISC Instruction Sets

- When we talk about **RISC vs. CISC**, we're basically talking about two philosophies for designing a computer's instruction set—the "language" that the CPU speaks.

RISC (Reduced Instruction Set Computer)

- **Core idea**: Keep instructions *simple* and *uniform*.
- **One instruction = one word in memory.**
- This makes decoding easier and faster.
- **Operands must be in registers** before you do arithmetic/logic.
- Example: If you want to add two numbers from memory, you must first Load them into registers, then Add.
- **Pipeline-friendly**: Since instructions are all the same size and simple, the CPU can line them up and execute them in an assembly-line (pipelined) fashion. This boosts speed.
- **Trade-off**: More instructions are needed to complete a complex task, but each runs very fast.
- **Examples**: ARM, MIPS, SPARC, RISC-V.

CISC (Complex Instruction Set Computer)

- **Core idea**: Make each instruction powerful, even if it's complex.
- Instructions may be **multi-word** and can directly work on memory.
- Example: You can ADD a value in memory directly to a register—no separate Load step needed.
- This means you can often do more work with a single instruction.
- **Trade-off**: Instructions are harder to decode, can take variable time, and don't pipeline as cleanly.
- **Examples**: Intel x86, VAX, older IBM System/360.

Feature	RISC	CISC
Instruction size	Fixed (usually 1 word)	Variable (1–15 bytes for x86)
Operands	Registers only (load/store model)	Memory or registers
Complexity	Few simple instructions	Many complex instructions
Performance	Optimized for speed & pipelining	Optimized for doing more in fewer instructions
Examples	ARM, MIPS, RISC-V	Intel x86, AMD64

Basic Instruction Types

- 1. **Three-Address Instruction**
- ADD R1, R2, R3 ; R3 ← R1 + R2
- Uses **3 operands**: two sources (R1, R2) and one destination (R3).
- Flexible, since the result can go into a different register.
- Needs more bits to represent all 3 addresses → instruction size is large.

2. **Two-Address Instruction**

- ADD R1, R2 ; R2 ← R1 + R2
- Uses **2 operands**: one source (R1) and one that is both source and destination (R2).
- More compact than 3-address but less flexible (result overwrites one operand).
- Still needs multiple words to fit.

3. **One-Address Instruction**

- ADD A ;AC ← AC + M[A]
- Uses **1 operand**.
- Here, one operand is implied → the **Accumulator (AC)**.
- Explicit operand (A) is typically a memory location.
- Smaller instructions, but limited because everything must go through AC.

4. Zero-Address Instruction

- ADD ;Pop two values from stack, add them, push result back
- No operands are explicitly mentioned.
- Uses a **stack-based architecture**.
- Operands are implicitly taken from the **top of the stack**.
- Very compact instructions.

- More addresses = more flexible, but larger instructions.
- Fewer addresses = compact, but less flexible (need accumulator or stack).

Basic Instruction Types

- Example: Evaluate E=(A+B) * (C+D)
- Three-Address

- **Step 1** ADD A, B, R1

R1 ← M[A] + M[B]

Take values of A and B from memory.

Store the result in register R1.

- **Step 2** ADD C, D, R2

R2 ← M[C] + M[D]

Take values of C and D from memory.

Store the result in register R2.

- **Step 3** MUL R1, R2, E

M[E] ← R1 × R2

Multiply R1 and R2.

Store the result into memory location E.

Basic Instruction Types

3 addresses

Opcode Operand 1, Operand 2, Result

a = b + c;

- **Advantage**
- Very **flexible** because you can keep results in different registers or memory locations without overwriting inputs.
- Easy to understand — looks like normal high-level code.
- **Disadvantage**
- **Instruction length is large**:
  - You need space for **three addresses + opcode**.
  - That means longer instruction words, harder to fit in limited instruction size.
- **Not common in real CPUs**

Basic Instruction Types

- Example: Evaluate E=(A+B) * (C+D)
- Two-Address

Step 1 MOV A, R1

R1 ← M[A]

Load A into register R1.

Step 2 ADD B, R1

R1 ← R1 + M[B]

Add B to R1 (now R1 = A + B).

Step 3 MOV C, R2

R2 ← M[C]

Load C into register R2

Step 4 ADD D, R2

R2 ← R2 + M[D]

Add D to R2 (now R2 = C + D).

Step 5 MUL R1, R2

R2 ← R1 × R2

Multiply R1 and R2. Result stored in R2.

Step 6 MOV R2, E

M[E] ← R2

Store the result into memory location E.

Basic Instruction Types

- **Advantages:**
- Shorter instruction word than 3-address.
- More widely used in real architectures (balance between size and flexibility).
- **Disadvantages:**
- One operand gets overwritten (less flexible).
- Often need extra **MOV** instructions, so more instructions overall.

- **Advantages:**
- Shorter instruction word than 3-address.
- More widely used in real architectures (balance between size and flexibility).
- **Disadvantages:**
- One operand gets overwritten (less flexible).
- Often need extra **MOV** instructions, so more instructions overall.