Computer Function and Interconnection

By : Bilal Ahmed

Von Neumann Architecture

Unified Memory

- **Data and instructions** are stored **together** in the same read/write memory.

Memory Addressing

- Memory is **addressed by location only**, without distinguishing between data and instructions.

Sequential Execution

- Instructions are executed **one after the other**, in the order they are stored, unless the flow is changed by special instructions (like jumps or branches).

The Motivation Behind the Architecture

Before: Hardwired Programming (Fixed Functionality).

Computation was performed using **custom-built hardware**.

Each task had a **specific physical configuration** of logic components.

Changing the program meant **rewiring** the hardware (**hardwired program**).

Problem: Rewiring for every new task was **inflexible, time-consuming, and expensive**.

The Breakthrough: General-Purpose Hardware + Control Signals

General-Purpose Hardware

- A single, fixed set of logic circuits can perform **many different operations** (add, subtract, AND, etc.).
- The operation performed depends on the **control signals** sent to the hardware.

How It Works

- Instead of rewiring, we just send **different control signals** to get different operations.
- This allows **reprogrammability** without changing the physical hardware.

From Control Signals to Instructions

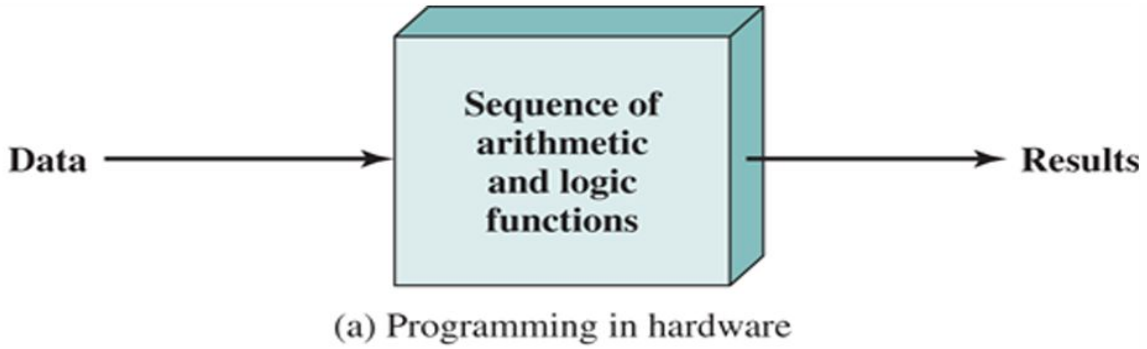
📌 Problem:

- How do we easily supply the right control signals for each step in a program?

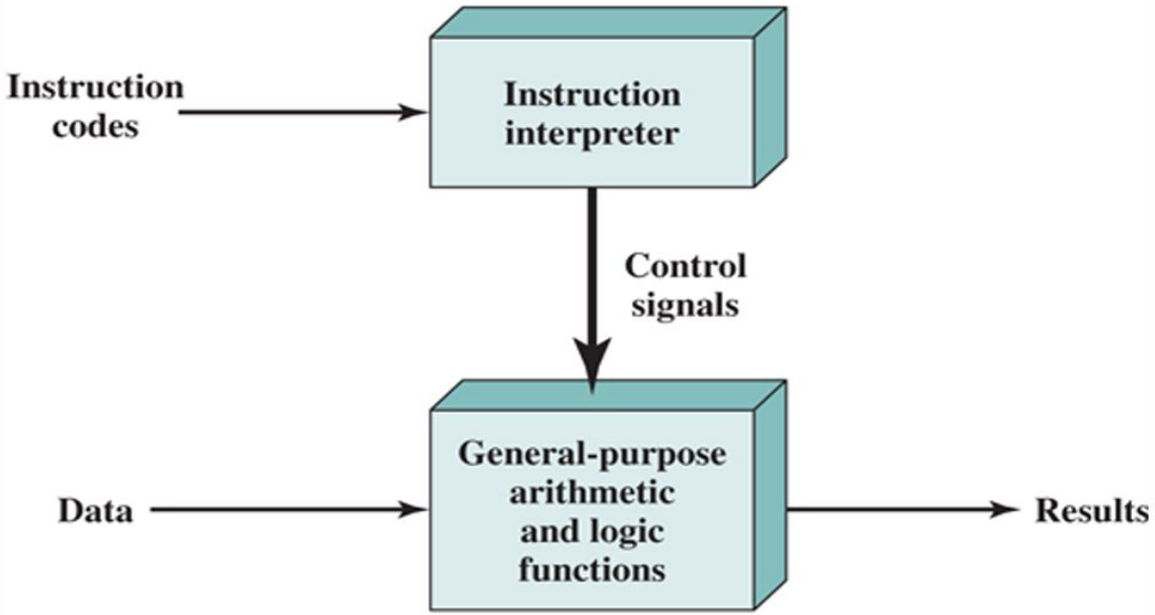
✅ Solution:

- **Define a unique code** for each possible set of control signals.
- Store these codes in memory alongside data.
- Add a **decoder** in hardware that reads the code and produces the corresponding control signals.

Diagram



(a) Programming in hardware



(b) Programming in software

Figure a:

- In **custom hardware**, data goes in → results come out.
- Control is "wired in" and fixed.

Figure b:

- In **programmable hardware**, both data and control codes go in.
- Hardware decodes instructions → generates control signals → produces results.

I/O and Memory

- A functioning computer requires additional components:
  - **Input module** – accepts data/instructions and converts them into internal signals.
  - **Output module** – reports results.
  - Together, these form the **I/O components**.
- Programs are not always executed sequentially (e.g., jump instructions in the IAS machine).
- Operations may require non-sequential access to multiple data elements.
- A storage area is therefore needed:
  - **Main memory** – temporarily stores both instructions and data.
  - Same memory can be used for instructions and data (Von Neumann concept).

Memory

Main memory assigns an address to each instruction and data element.

The CPU can fetch any item directly using its address.

This allows execution to **jump** to different instructions (branching, loops).

Data can be accessed in a **non-sequential** order as required by operations.

Thus, memory supports flexible program flow and complex computations.

CPU Registers for Memory and I/O Operations

- The **CPU exchanges data with memory** using two internal registers:
  - **Memory Address Register (MAR):** holds the address in memory for the next read or write.
  - **Memory Buffer Register (MBR):** holds data to be written to memory or receives data read from memory.
- For **I/O operations**, two similar registers are used:
  - **I/O Address Register (I/OAR):** specifies the I/O device involved.
  - **I/O Buffer Register (I/OBR):** holds data exchanged between the CPU and the I/O module.

Computer Components: Top-Level View

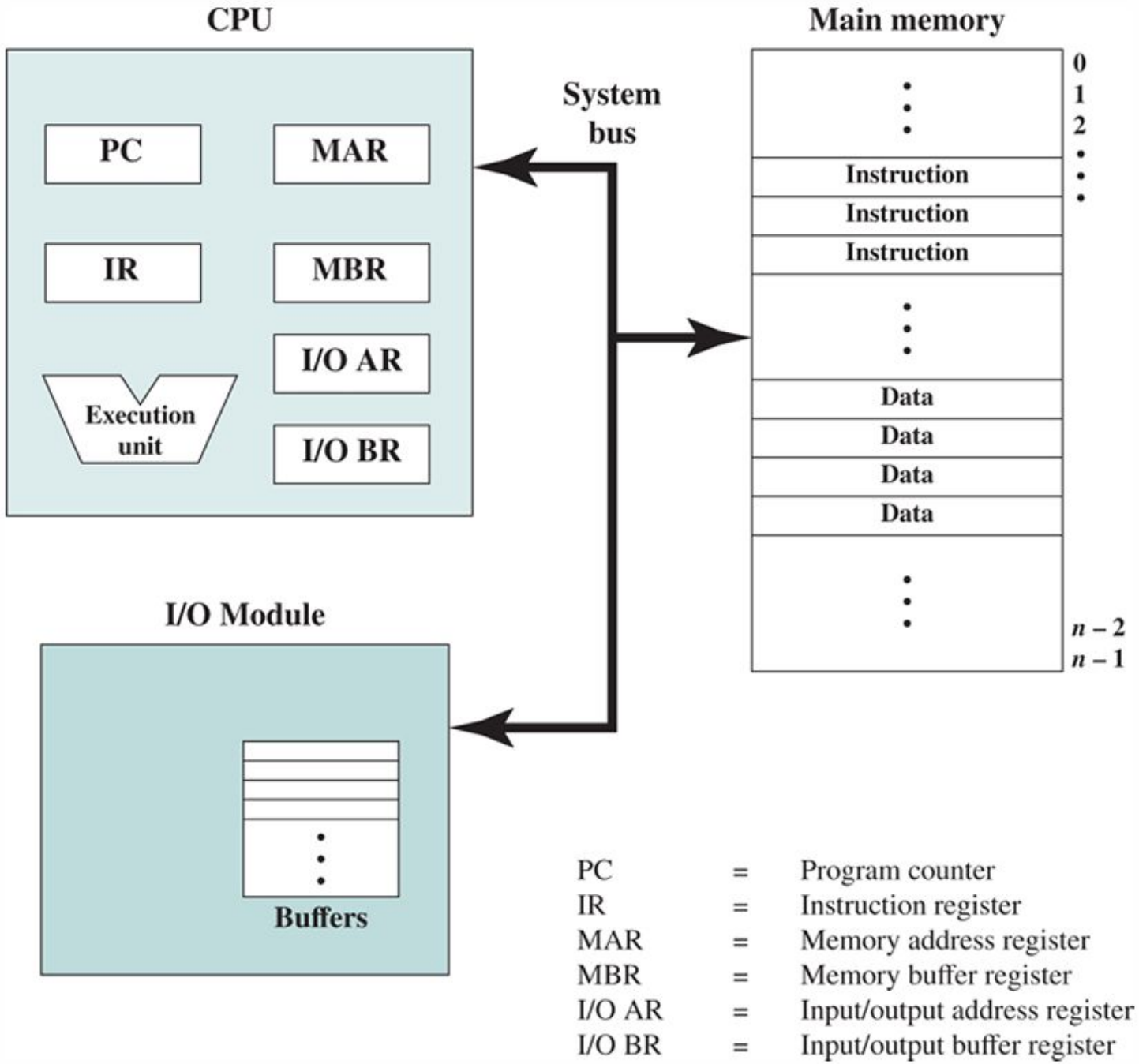


Figure 3.2 Computer Components: Top-Level View

Memory and I/O Modules

Memory module

- Consists of sequentially numbered addresses (locations).
- Each location stores a binary number (interpreted as instruction or data).

I/O module

- Transfers data between external devices, CPU, and memory.
- Uses internal buffers to temporarily hold data before transfer.

Program Execution Overview

A computer’s basic function is to **execute programs** stored in memory.

A program consists of a **set of instructions**.

The **processor** performs the actual work by executing these instructions.

Instruction processing (simplest form):**Fetch**: Read the next instruction from memory and **Execute**: Carry out the instruction.

Program execution = continuous cycle of **fetch** → **execute**.

Instruction Cycle

Processing of a single instruction = **Instruction Cycle**.

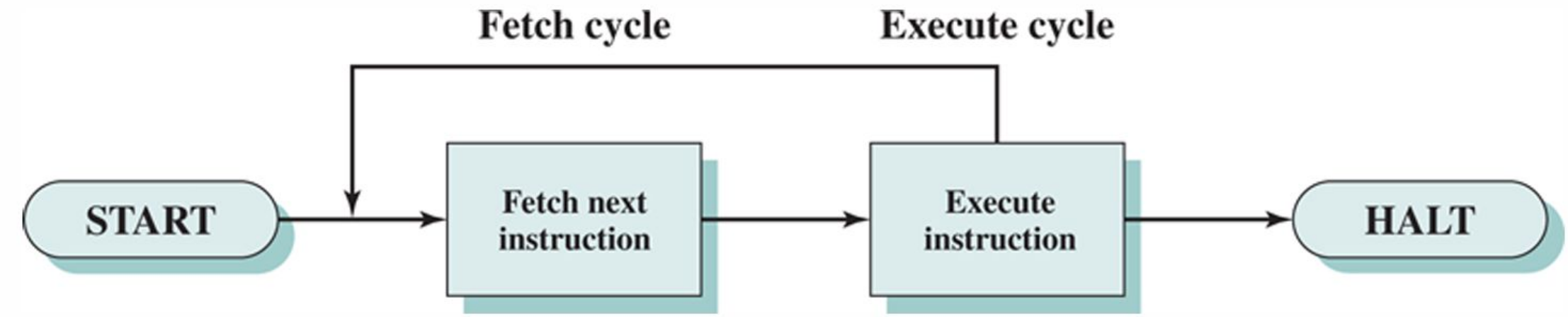
Simplified form consists of two steps (see **Figure in next slide**):

- **Fetch cycle** – retrieves the instruction from memory.
- **Execute cycle** – carries out the instruction.

Program execution halts only if:

- The machine is turned off.
- An unrecoverable error occurs.
- A program instruction explicitly halts execution.

Instruction Cycle



Instruction Fetch

At the start of each instruction cycle, the processor **fetches an instruction** from memory.

Program Counter (PC): holds the address of the next instruction.

PC is usually incremented after each fetch → fetches instructions sequentially.

Example: If PC = 300, instructions will be fetched from 300, 301, 302 … unless altered.

Instruction Register (IR): stores the fetched instruction.

Instruction Execution

The instruction specifies the action to be performed.

Processor **interprets and executes** the instruction.

Execution categories:

- **Processor–Memory:** transfer data between CPU and memory.
- **Processor–I/O:** transfer data between CPU and I/O devices.

Instruction Execution

Data Processing: arithmetic or logic operations on data.

Control: alters the sequence of execution.

- Example: instruction at location 149 specifies next instruction at 182.
- PC is updated accordingly → next fetch occurs from 182 instead of 150.

A single instruction may involve a **combination** of these actions.

Execution Categories Overview

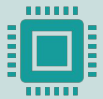
Processor–Memory: Transfer data between CPU and memory.

Processor–I/O: Transfer data between CPU and I/O devices.

Data Processing: Perform arithmetic or logic operations.

Control: Change the sequence of instruction execution.

Interconnection Structure



A computer is built from three basic types of modules: **processor, memory, and I/O**.

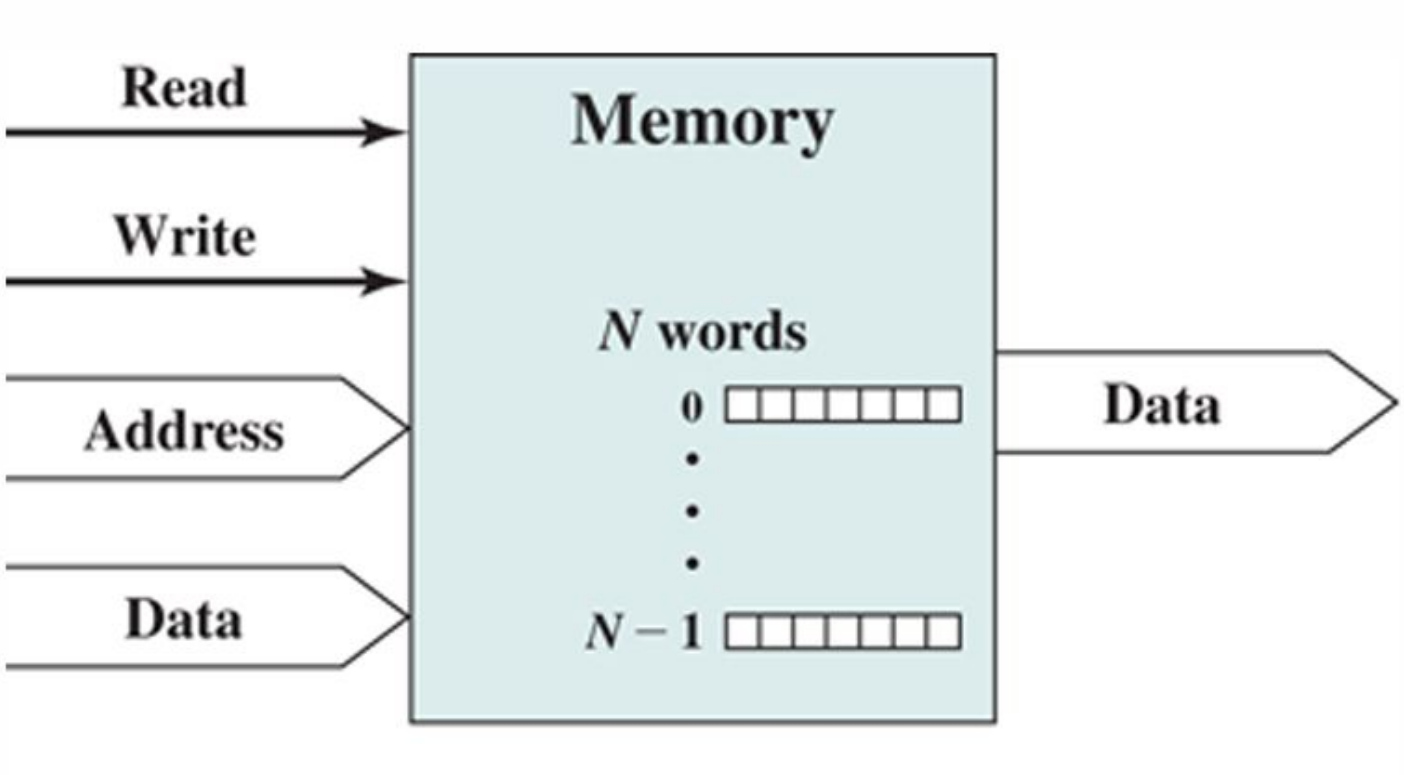
These modules must communicate with each other.

The communication paths between modules form the **interconnection structure**.

The design of this structure depends on the types of exchanges required among modules.

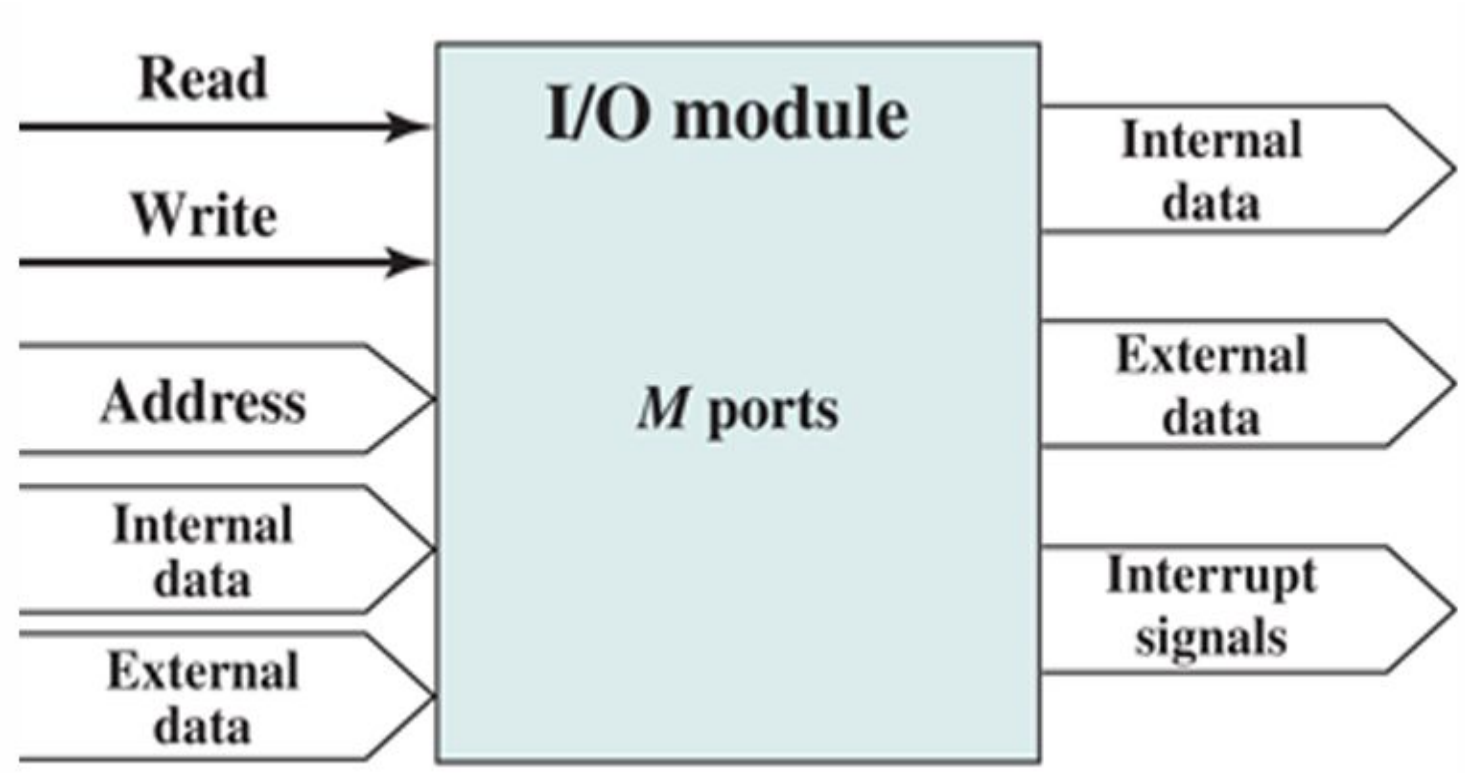
Memory

- Consists of *N words*, each with a unique address (0 … N–1).
- Supports **read** and **write** operations.
- Address specifies the location; control signals indicate operation type.

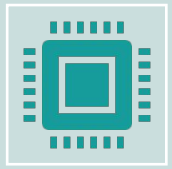


I/O Module

- Functionally similar to memory (read/write).
- Can control multiple external devices via **ports** (each with a unique address 0 … M–1).
- Provides external data paths for input/output.
- Can send **interrupt signals** to the processor.



Internal and External Data in I/O

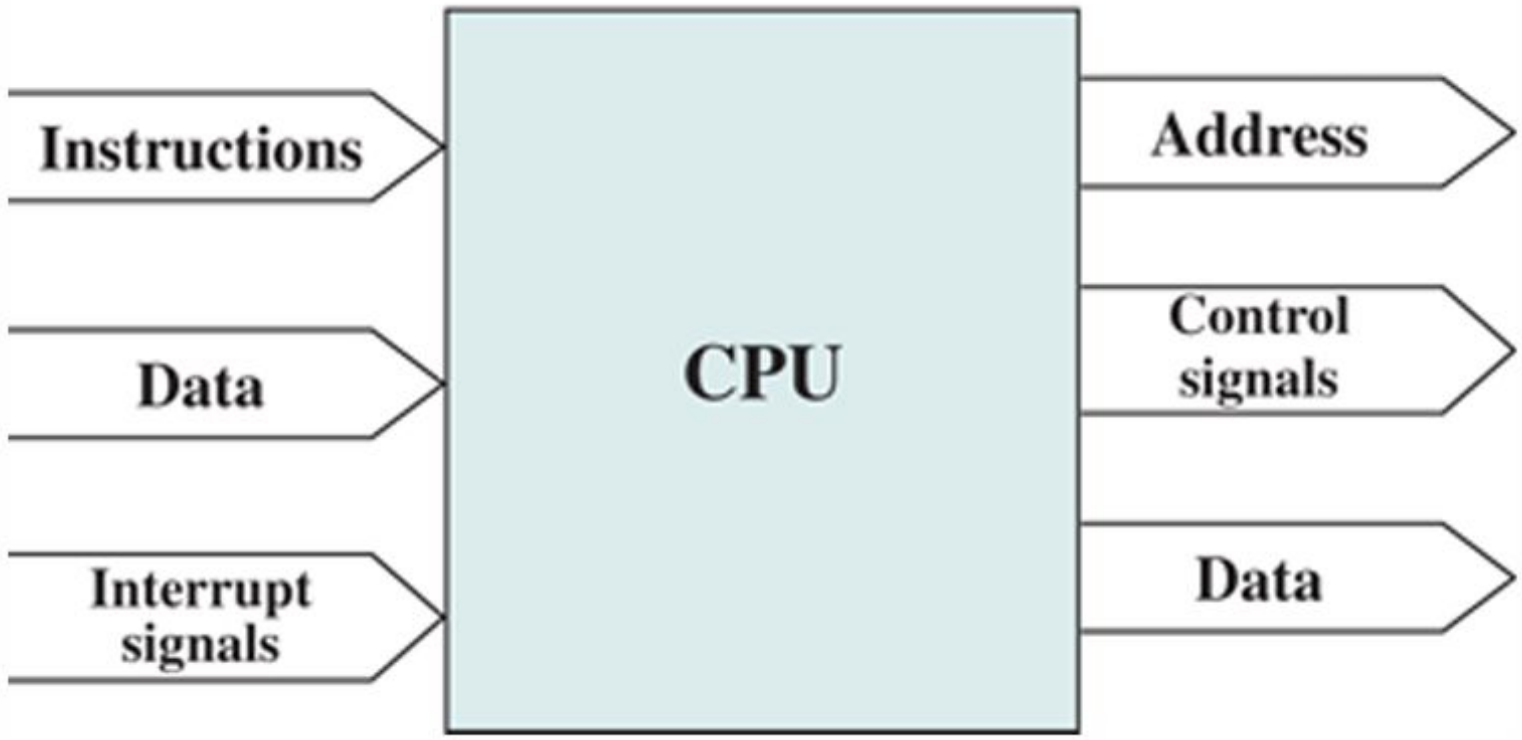


Internal Data: Data exchanged between **CPU/memory and the I/O module**.

External Data: Data exchanged between the **I/O module and external devices**.

Processor

- Reads instructions and data, processes them, and writes results.
- Uses control signals to manage system operation.
- Handles **interrupt signals** from I/O modules.



Types of Data Transfers

Memory → Processor: processor reads instruction or data from memory.

Processor → Memory: processor writes data to memory.

I/O → Processor: processor reads data from an I/O device.

Processor → I/O: processor sends data to an I/O device.

I/O ↔ Memory: I/O module exchanges data directly with memory (DMA).

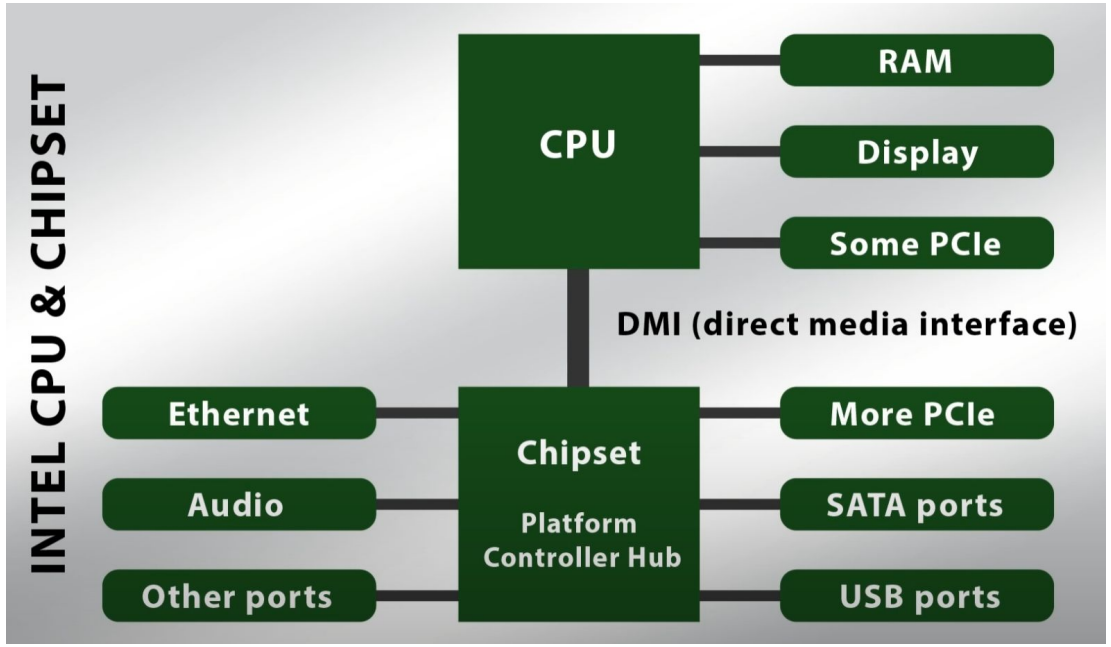
Bus and Interconnection Structures

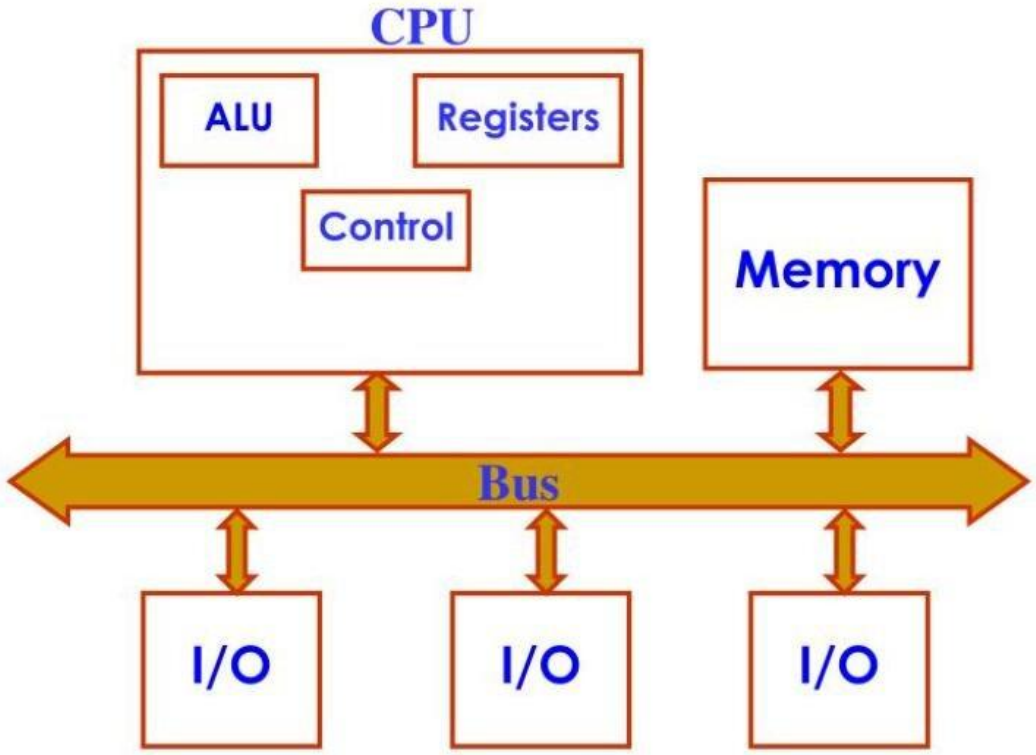
Bus was the dominant method of connecting computer components for decades.

In general-purpose computers, **point-to-point interconnections** have largely replaced buses.

Bus structures are still widely used in **embedded systems** (e.g., microcontrollers).

Bus vs Point-to-Point Interconnection





Bus vs Point-to-Point Interconnection

Bus Interconnection (Right diagram)

- Single shared communication pathway (bus).
- CPU, memory, and I/O all compete for access.
- Simple and low cost, but limited speed and scalability.

Point-to-Point Interconnection (Left diagram)

- Dedicated links between CPU, memory, and chipset.
- Multiple parallel data transfers possible.
- Higher performance, supports modern multi-core CPUs.

BUS vs Point-to-Point Topology

Feature	Bus Interconnection	Point-to-Point Interconnection
Topology	Single shared communication path for all devices	Dedicated links between pairs of devices
Number of Simultaneous Transfers	Only one device can transmit at a time	Multiple transfers can occur simultaneously
Scalability	Limited; performance drops as more devices are added	Highly scalable; adding devices doesn't significantly reduce performance
Cost	Low; simple design and fewer lines	Higher; more wiring and complex control required
Performance	Lower bandwidth due to contention	Higher bandwidth and lower latency
Use Cases	Embedded systems, microcontrollers	Modern CPUs, multi-core processors, high-performance SoCs
Complexity	Simple to implement	More complex; requires routing and arbitration mechanisms
Reliability	Single point of failure if bus fails	More reliable; failure in one link does not affect others

Timeline of Computer Interconnections

Bus-Based Interconnection (1950s–1990s)

- Shared communication path connecting all devices.
- Simple, low-cost, used in early PCs and microcontrollers.

Point-to-Point Interconnection (1990s–present)

- Dedicated links between devices for simultaneous transfers.
- Higher bandwidth, lower latency, scalable → modern CPUs, multicore SoCs.

Emerging/Advanced Interconnects

- **Crossbar switches**: multiple simultaneous transfers without contention.
- **Networks-on-Chip (NoC)**: for manycore processors and high-performance SoCs.

Microcontrollers (Embedded Systems)

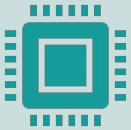
Most microcontrollers still use **bus-based interconnections** internally (e.g., AMBA AHB, APB in ARM Cortex-M).

Reason: Simple, low-cost, low-power, sufficient for single-core or low-speed operations.

Practical example: **I²C, SPI, UART** → peripheral buses allowing multiple sensors/devices to communicate with the MCU.

Modern CPUs / SoCs

Use **point-to-point interconnects** for high-performance communication.

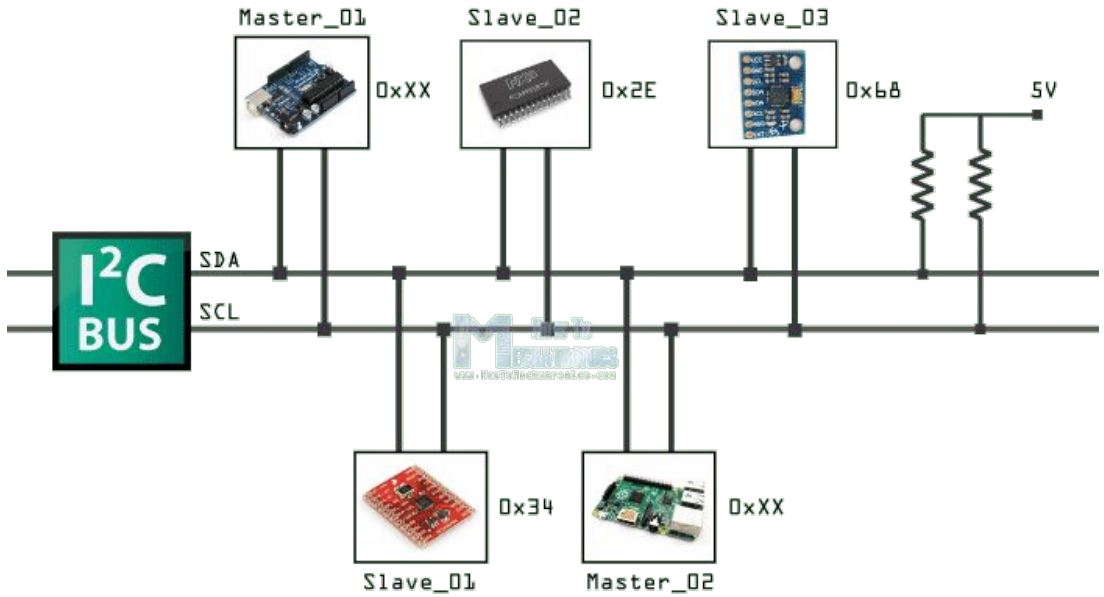


Example: Intel QPI, Intel DMI, AMD Infinity Fabric, ARM AMBA AXI (on multicore SoCs).

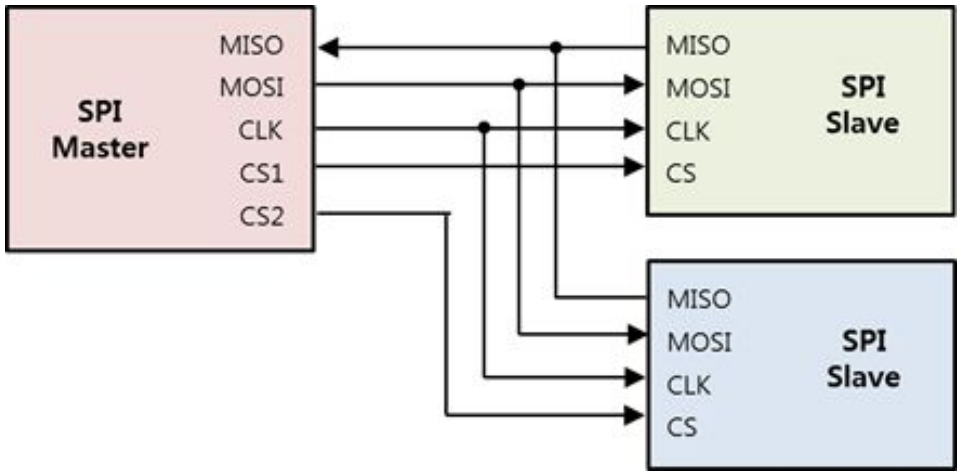
Enables **simultaneous data transfers**, reducing bottlenecks in multicore or multiprocessor setups.

Bus in microcontrollers

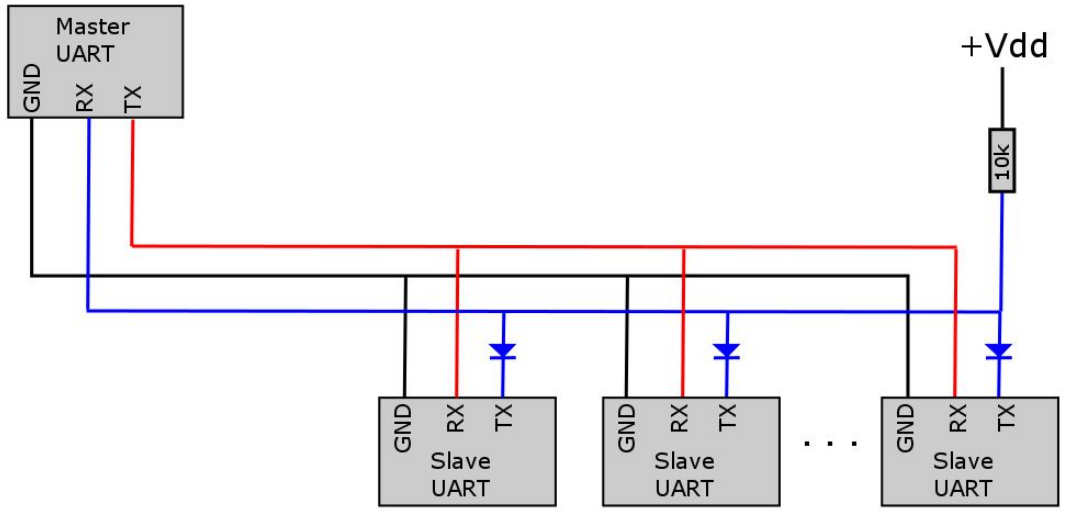
I2c



SPI



UART



Bus Characteristics

A **bus** is a communication pathway connecting two or more devices.

It is a **shared medium** – signals from one device are available to all others.

Only **one device** can transmit at a time; simultaneous transmissions cause errors.

A bus has multiple **lines**, each carrying binary signals (0 or 1).

Lines can transmit data **sequentially** (over time) or **in parallel** (e.g., 8-bit data over 8 lines).

System Bus and Its Functional Groups

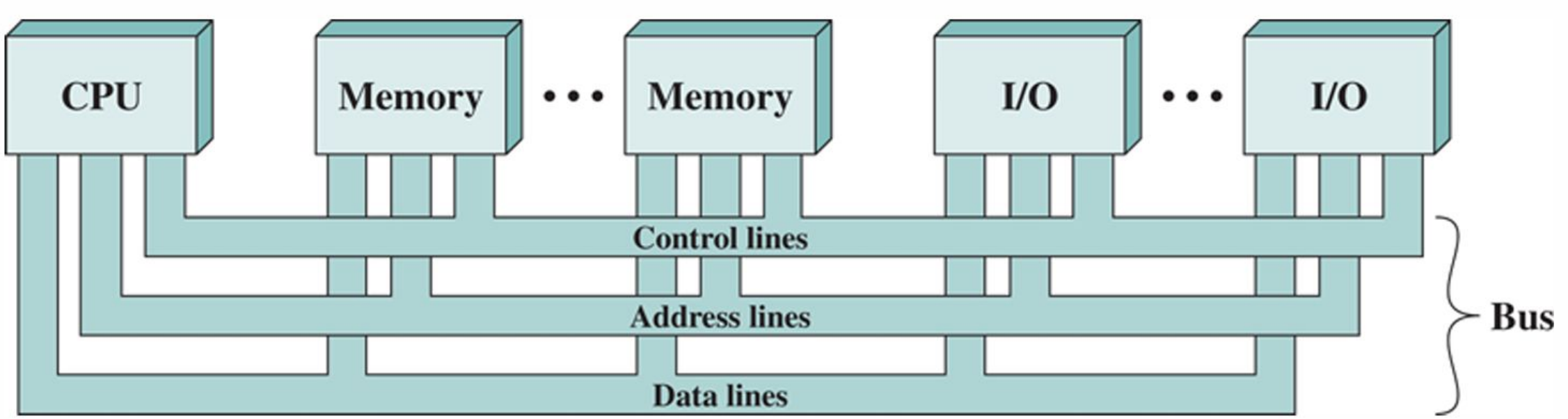
A **system bus** connects major components: processor, memory, and I/O.

Common interconnection structures are based on one or more system buses.

A system bus typically has **50–100+ lines**, each with a specific function.

Bus lines are grouped into three functional categories: data, address and control lines.

May also include **power lines** for modules.



- **Data lines** – transfer data.
- **Address lines** – specify source/destination.
- **Control lines** – manage operations.

Data Bus

- Provides path for transferring data among modules.
- Width = number of lines (e.g., 32, 64, 128 bits).
- Determines how many bits can be transferred at once.
- Wider bus → better performance.
- *Example:* 32-bit bus with 64-bit instructions requires two memory accesses.

Address Bus

- Specifies the **source or destination** of data.
- Width defines the **maximum memory capacity** of the system.
- Also used for addressing **I/O ports**.
- High-order bits → select module.
- Low-order bits → select location/port within module.

Control Lines

Control access to shared data and address lines.

Transmit two types of signals:

Timing signals: indicate validity of data/address.

Command signals: specify operations to perform.

Ensure proper coordination among system modules.