

UNIT 3

Basic Computer Architecture

(4 Hours) — Computer Organization & Architecture

This unit introduces the fundamental building blocks of a digital computer system. We trace how computer architecture evolved, look at how a computer is organized into functional units, study the memory hierarchy and cache, examine hard disk organization, and then move into the internal machine-level view: how instructions are represented as instruction codes, how registers and a common bus connect the CPU's internal components, what an instruction set is, and how the timing and control mechanism drives the instruction cycle.

3.1 Introduction

3.1.1 History of Computer Architecture

Computer architecture describes the functional behaviour of a computer system as seen by a programmer — the instruction set, addressing modes, registers, and data types it supports. It is distinct from computer organization, which deals with how the hardware actually implements that architecture (control signals, buses, memory interfacing, and so on). The evolution of computer architecture is generally traced through generations based on the underlying technology used to build the hardware.

Generation	Period	Technology	Key Characteristics
1st	1940s–1950s	Vacuum tubes	Very large, high power consumption, machine language programming
2nd	1950s–1960s	Transistors	Smaller, faster, more reliable; assembly language introduced
3rd	1960s–1970s	Integrated Circuits (IC)	Multiple transistors on a chip; high-level languages, OS development
4th	1970s–present	Microprocessors (VLSI)	Entire CPU on a single chip; personal computers emerge
5th	Present – future	AI, parallel & multicore processing	Massive parallelism, machine learning accelerators, quantum research

Historically, architecture also branched along design philosophies. Early machines used the von Neumann architecture, in which a single memory space stores both program instructions and data, and both are fetched over a shared bus. An alternative, the Harvard architecture, keeps instruction memory and data memory physically separate so both can be accessed

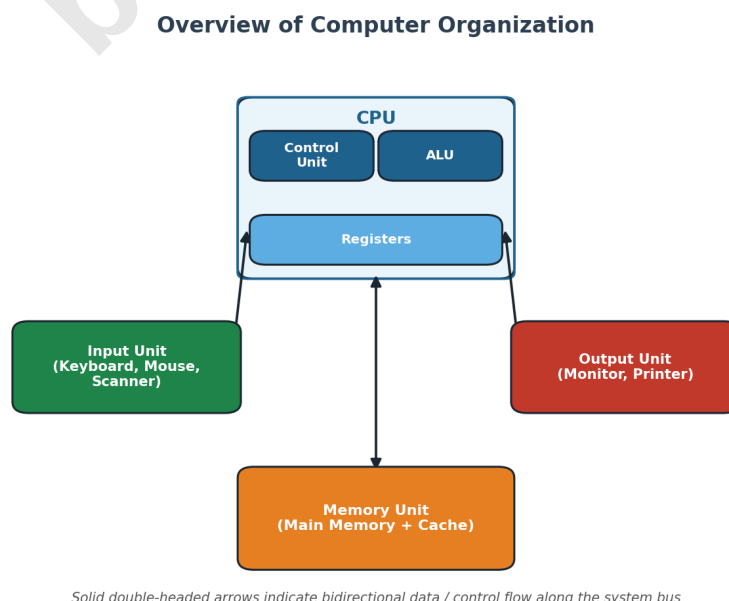
simultaneously — a design still common in embedded processors and digital signal processors today. As chip fabrication improved, architecture also diverged into CISC (Complex Instruction Set Computer, e.g. Intel x86) which favours rich, multi-step instructions, and RISC (Reduced Instruction Set Computer, e.g. ARM), which favours a small set of simple, uniform-length instructions executed very quickly.

Note: Von Neumann's 1945 report on the EDVAC is widely regarded as the origin of the stored-program concept that underlies almost all modern computers.

3.1.2 Overview of Computer Organization

At the organizational level, a computer system is built from four functional units that cooperate to accept input, process it, and produce output.

- **Input Unit** — accepts data and instructions from the outside world (keyboard, mouse, scanner) and converts them into a form the machine can process.
- **Central Processing Unit (CPU)** — the “brain” of the computer, containing the Control Unit (CU), which fetches and decodes instructions and generates control signals, and the Arithmetic Logic Unit (ALU), which performs arithmetic and logical operations. A set of registers inside the CPU holds data and addresses temporarily during execution.
- **Memory Unit** — stores programs and data, organized as a hierarchy from very fast, small registers/cache down to large, slower secondary storage.
- **Output Unit** — converts processed results into a human-readable or usable form (monitor, printer, speaker).



Solid double-headed arrows indicate bidirectional data / control flow along the system bus

Figure 3.1 — Functional units of a computer and their interconnection

All of these units communicate over a common set of wires called the system bus, which is conceptually split into an address bus (carries memory/IO addresses), a data bus (carries the actual data being transferred), and a control bus (carries signals such as read/write, clock, and interrupt lines).

3.1.3 Memory Hierarchy and Cache

No single memory technology can be simultaneously very fast, very large, and very cheap. The memory hierarchy resolves this conflict by arranging several kinds of storage in levels: the fastest and most expensive (but smallest) memory sits closest to the CPU, while progressively slower, cheaper, and larger memory sits farther away.

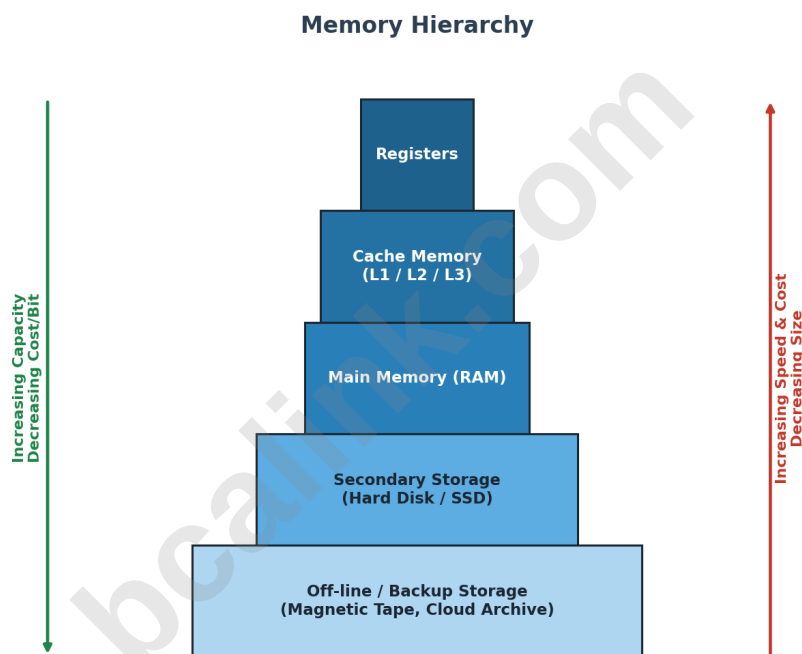


Figure 3.2 — The memory hierarchy, from fast/small registers to slow/large offline storage

Level	Typical Size	Typical Access Time	Managed By
Registers	Few bytes (32–128 registers)	< 1 ns	Compiler / CPU
Cache (L1/L2/L3)	KB to a few MB	1–10 ns	Hardware (cache controller)
Main Memory (RAM)	GBs	50–100 ns	Operating System
Secondary Storage	TBs	Milliseconds	Operating System / user
Offline / Backup	TBs and above	Seconds or more	User / archival software

Cache memory is a small, very fast memory placed between the CPU and main memory. It works on the principle of locality of reference: programs tend to access the same memory locations repeatedly over a short period (temporal locality), and tend to access locations near recently used ones (spatial locality). Because of this, copying recently or nearby used data into cache means most memory requests can be served quickly without going all the way to RAM. When the CPU finds the data it needs already in cache, this is a cache hit; when it does not, it is a cache miss, and the data must be fetched from a slower level and placed into the cache for future use.

Example 3.1 — Average Memory Access Time

A system has a cache hit ratio of 0.90. Cache access time is 5 ns and main memory access time is 80 ns. Find the average memory access time (AMAT).

Solution

$AMAT = (\text{Hit ratio} \times \text{Cache time}) + (\text{Miss ratio} \times \text{Memory time})$

$AMAT = (0.90 \times 5) + (0.10 \times 80) = 4.5 + 8 = 12.5 \text{ ns}$

This is far closer to the cache's 5 ns than to memory's 80 ns — illustrating why even a modest hit ratio dramatically improves effective speed.

3.1.4 Organization of Hard Disk

A hard disk drive (HDD) is a secondary storage device that stores data magnetically on one or more rotating circular platters. Its physical organization directly affects how quickly data can be located and transferred.

- **Platters** — rigid, magnetically coated circular disks stacked on a common spindle; data is recorded on both surfaces.
- **Tracks** — each platter surface is divided into concentric circular paths called tracks.
- **Sectors** — each track is further divided into small arcs called sectors, the smallest addressable unit of storage (traditionally 512 bytes, now often 4 KB).
- **Cylinder** — the set of tracks located at the same radial position across all platter surfaces; data in one cylinder can be accessed without moving the read/write heads.
- **Read/Write Heads** — one per platter surface, mounted on an actuator arm that moves them together radially across the platters.
- **Spindle Motor** — rotates the platters at a constant speed (e.g. 5,400 / 7,200 / 15,000 RPM).

The time to access data on a disk has three components: seek time (time to move the head to the correct track), rotational latency (time for the desired sector to rotate under the head), and

transfer time (time to actually read/write the data once positioned). Minimizing these delays — for example through disk scheduling algorithms and file placement strategies — is a major concern in operating system design.

Note: Solid State Drives (SSDs) have no moving parts and therefore no seek time or rotational latency in the mechanical sense, which is why they vastly outperform HDDs on random access workloads.

Key Takeaways — Section 3.1

- ✓ Computer architecture (programmer's view) is distinct from computer organization (hardware implementation view); design philosophies include von Neumann vs Harvard, and CISC vs RISC.
- ✓ A computer is organized into four functional units — Input, CPU (CU + ALU + registers), Memory, and Output — connected via address, data, and control buses.
- ✓ The memory hierarchy trades speed against capacity and cost; cache memory exploits temporal and spatial locality to bridge the CPU–memory speed gap.
- ✓ A hard disk is organized into platters, tracks, sectors, and cylinders; total access time = seek time + rotational latency + transfer time.

3.2 Instruction Codes

An instruction code is a group of bits that instructs the computer to perform a specific operation. It is the machine-level representation of a program statement and is usually divided into an operation code (opcode), which specifies the operation to be performed, and an address part, which specifies the operand(s) or their location(s).

3.2.1 Stored Program Organization

In the stored program concept, both instructions and data are stored together in the same read/write memory, encoded as binary numbers, and instructions are fetched from memory one at a time and executed by the control unit. This lets a computer be reprogrammed simply by loading different instructions into memory, rather than by rewiring hardware.

A simple stored-program computer may use a single-address instruction format such as:

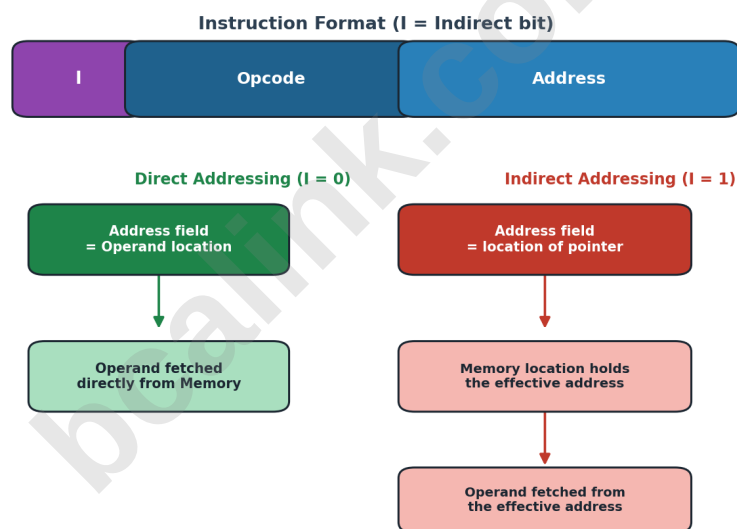


Figure 3.3 — A single-address instruction format, showing the indirect bit, opcode, and address fields

In such a format, one bit of the instruction (often the leftmost bit of the address part) is reserved as the indirect bit (I), which tells the control unit how to interpret the address field.

3.2.2 Indirect Address

- Direct Addressing ($I = 0$) — the address field contains the actual (effective) address of the operand in memory. The operand is fetched with a single memory reference.
- Indirect Addressing ($I = 1$) — the address field contains the address of a memory location that itself holds the effective address of the operand (a pointer). This requires two memory references: one to read the pointer, and one to read the actual operand.

Example 3.2 — Direct vs Indirect Addressing

Suppose an instruction's address field contains the value 500.

- (a) Under direct addressing, what does the CPU do?
- (b) Under indirect addressing, if memory location 500 contains the value 800, what does the CPU do?

Solution

- (a) Direct: the CPU treats 500 itself as the operand's address and fetches the operand directly from memory location 500.
- (b) Indirect: the CPU first reads memory location 500, obtaining the value 800. It then treats 800 as the effective address and fetches the actual operand from memory location 800.

Note: Indirect addressing is powerful for implementing pointers, arrays, and dynamic data structures, but it costs an extra memory cycle compared with direct addressing.

3.2.3 Computer Registers

Registers are small, extremely fast storage locations built directly into the CPU. They hold data, addresses, or control information that is actively being used during instruction execution. A typical basic computer includes the following registers:

Register	Name	Function
DR	Data Register	Holds the operand read from / to be written to memory
AR	Address Register	Holds the memory address for the next read/write
AC	Accumulator	General-purpose register that holds intermediate arithmetic/logic results
IR	Instruction Register	Holds the instruction currently being decoded/executed
PC	Program Counter	Holds the address of the next instruction to be fetched
TR	Temporary Register	Holds intermediate data during a computation
INPR	Input Register	Holds an input character received from an input device
OUTR	Output Register	Holds an output character to be sent to an output device

During the instruction cycle, the Program Counter is copied into the Address Register so that the instruction word stored at that memory address can be fetched into the Instruction Register; the Program Counter is then incremented so it points to the next instruction in sequence (unless a branch instruction changes its value directly).

3.2.4 Common Bus System

Rather than wiring separate paths between every pair of registers (which would need an impractical number of connections), most basic computers connect all registers and the ALU through a single shared pathway called the common bus. A multiplexer (MUX) selects, under control-unit supervision, which register places its contents onto the bus at any given moment; the value on the bus can then be loaded into one or more destination registers, or fed into the ALU.

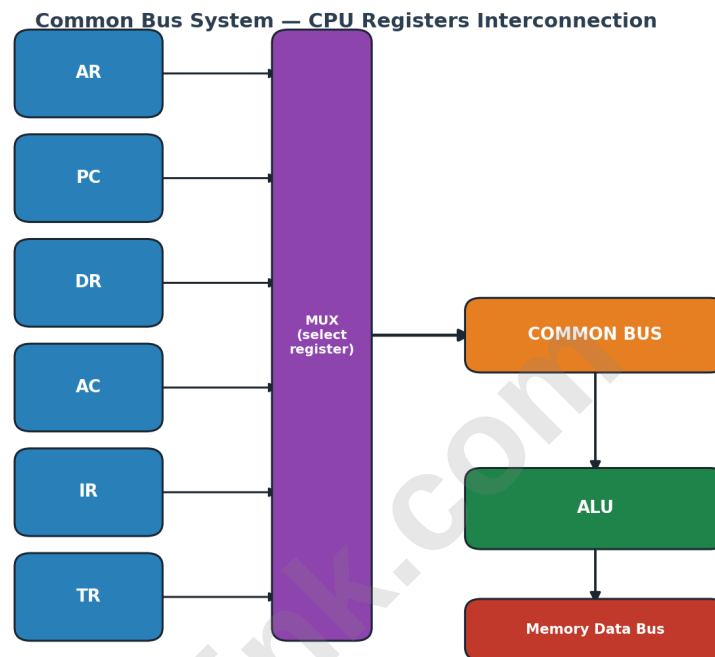


Figure 3.4 — A common bus system connecting CPU registers, the ALU, and memory

This design greatly reduces the number of wires needed inside the CPU. The trade-off is that only one register can drive the bus at a time, so the control unit must schedule bus usage carefully — this is precisely the job of the timing and control mechanism described in Section 3.2.6.

Note: Selection lines (often called S2 S1 S0 for a bus with several sources) are decoded by the MUX to choose which register's output appears on the common bus during a given clock cycle.

3.2.5 Instruction Set

The instruction set of a computer is the complete collection of machine instructions that its CPU is designed to execute. It defines everything a programmer (or a compiler) can ask the hardware to do. Instructions are typically grouped into functional categories:

Category	Purpose	Examples
Data Transfer	Move data between registers, memory, and I-O	LOAD, STORE, MOVE, IN, OUT

Category	Purpose	Examples
Arithmetic	Perform numeric computation	ADD, SUB, MUL, DIV, INC, DEC
Logical	Perform bitwise operations	AND, OR, XOR, NOT, COMPLEMENT
Shift	Shift bit patterns left or right	SHL, SHR, ROL, ROR
Control Transfer	Alter the normal sequence of execution	JUMP, BRANCH, CALL, RETURN, SKIP
I-O / Control	Manage input-output and machine state	HALT, INTERRUPT, NOP

Instructions are also classified by how many operand addresses they explicitly reference — zero-address (stack-based), one-address (uses an implied accumulator, as in the format of Figure 3.3), two-address, and three-address instructions — as well as by their addressing mode (immediate, direct, indirect, register, indexed, and so on), which determines how the effective operand address is calculated.

3.2.6 Timing and Control — Instruction Cycle

The timing and control unit generates the sequence of control signals, synchronized to the master clock, that drives every register transfer and ALU operation inside the CPU. It is commonly implemented using a ring counter or sequence counter that produces a series of evenly spaced timing signals (T_0 , T_1 , T_2 , ...), each of which enables a particular micro-operation.

The repeating sequence of steps the CPU performs to fetch, decode, and execute one complete instruction is called the instruction cycle. It consists of the following phases:

The Instruction Cycle

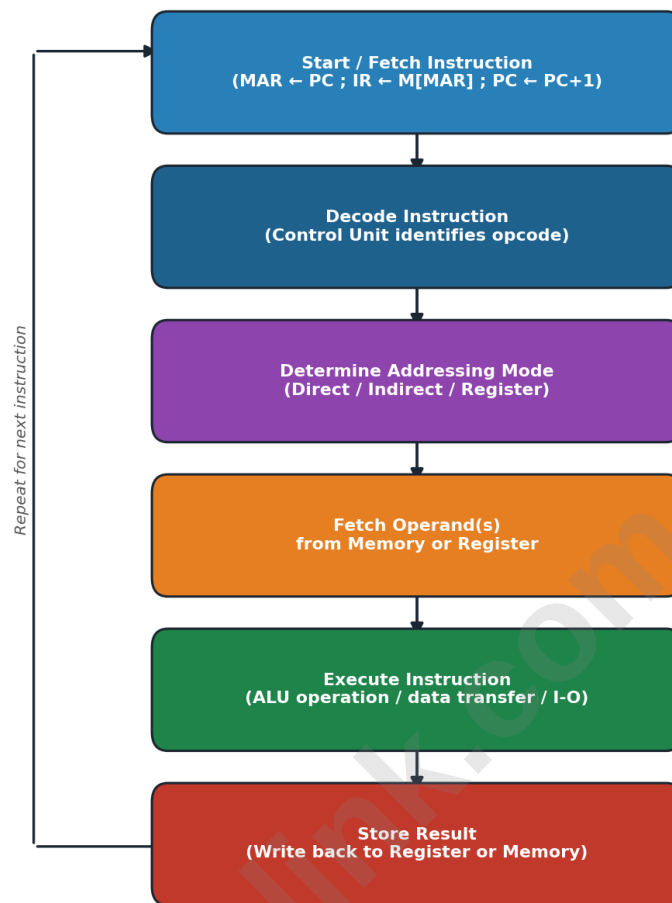


Figure 3.5 — The instruction cycle: fetch → decode → determine addressing mode → fetch operand → execute → store result

- **Fetch** — the instruction addressed by the Program Counter is copied from memory into the Instruction Register, and the Program Counter is incremented.
- **Decode** — the Control Unit examines the opcode field of the Instruction Register to determine which operation must be performed.
- **Determine Addressing Mode** — the indirect bit (and any other mode bits) are examined to work out how to compute the effective address of the operand.
- **Fetch Operand** — the actual data needed for the operation is retrieved from a register or from memory (with an extra memory reference if indirect addressing is used).
- **Execute** — the specified operation (arithmetic, logic, data transfer, or I-O) is carried out, typically inside the ALU.
- **Store Result** — the outcome is written back to a register or memory location, after which the cycle repeats for the next instruction fetched from the updated Program Counter.

If an interrupt request is pending after an instruction completes, the control unit diverts into an interrupt cycle before returning to the fetch phase: it saves the current Program Counter value, loads the address of the interrupt service routine into the Program Counter, and services the interrupt before resuming normal execution.

Example 3.3 — Register Transfers During Fetch

Write the register-transfer statements that describe the fetch phase of the instruction cycle for a basic computer with registers PC, AR, IR, and memory M.

Solution

$AR \leftarrow PC$ (copy address of next instruction into Address Register)

$IR \leftarrow M[AR]$ (read the instruction word from memory into the Instruction Register)

$PC \leftarrow PC + 1$ (advance the Program Counter to point to the following instruction)

Key Takeaways — Section 3.2

- ✓ An instruction code is split into an opcode (what to do) and an address part (where the operand is); one bit of the address part is often reserved as the indirect bit.
- ✓ Direct addressing needs one memory reference to reach the operand; indirect addressing needs two, since the address field points to a pointer rather than to the operand itself.
- ✓ Registers such as AR, PC, DR, AC, IR, and TR temporarily hold addresses and data during execution and are interconnected through a common bus controlled by a multiplexer.
- ✓ The instruction set defines every operation the CPU can execute, grouped into data transfer, arithmetic, logical, shift, and control-transfer categories.
- ✓ The instruction cycle — fetch, decode, determine addressing mode, fetch operand, execute, store result — is driven by the timing and control unit and repeats for every instruction in a program.

— End of Unit 3 —