Hardware Aspect of Computer Science & Engineering

Unit 1: History, Computer Basics, Data Representation & Evolution of Programming Languages

Dr. Gopal Chandra Jana

Dept. of CSE, Sharda University
Fundamentals of Computing and AI (CSAI1101) — Unit 1

September 2, 2026

Course Page:
https://www.gcjana.in/courses/shardauniversity/2601/fundamentals-of-computing-and-artificial-intelligence/

Outline

1. History of Computing Systems
2. Computer Basics & Organisation
3. Data Representation: Number Systems
4. Binary Codes
5. Evolution of Programming Languages
6. Summary & Practice

Learning Outcomes of Unit 1

After this unit, a student will be able to...

1. **Explain** how computing systems evolved — from the abacus to modern computers — and the five generations of computers.
2. **Describe** the basic structure of a computer and the von Neumann organisation (CPU, memory, I/O, buses).
3. **Convert** numbers between the decimal, binary, octal and hexadecimal number systems.
4. **Represent** data using binary codes — BCD, ASCII, Unicode, Gray code — and signed numbers.
5. **Classify** programming languages into machine, assembly and high-level languages, and explain how each is translated.

Mapping to the Syllabus

A. History of Computing Systems, Computer Basics & Organisation. **B.** Data Representation: Number Systems, Conversions, Binary Codes. **C.** Evolution of Programming Languages.

Why This Unit Matters

The foundation of everything you will study

Before we can *program* a computer or teach it to be *intelligent*, we must understand **what a computer is** and **how it stores information**.

- Every photo, message and AI model is ultimately **0s and 1s**.
- The CPU only understands **machine language** — pure binary.
- Understanding hardware makes you a **better** programmer and AI engineer.

Our journey in this unit

From *ancient counting tools* → *modern chips* → *how data is coded* → *how we talk to the machine*.

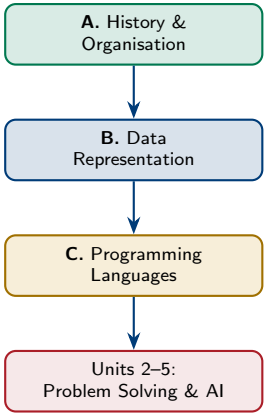


Figure: How this unit builds towards the rest of the course.

What is a Computer?

Definition

A **computer** is an electronic device that accepts **data** as input, processes it according to a set of **instructions** (a program), and produces **information** as output — and can **store** it for later use.

The word "computer"

Originally, a *"computer"* was a **person** who performed calculations by hand! The word was used this way as far back as 1613. Today it means the *machine*.

- Data = raw facts (e.g. marks: 78, 92, 65).
- Information = processed, meaningful data (e.g. average = 78.3).

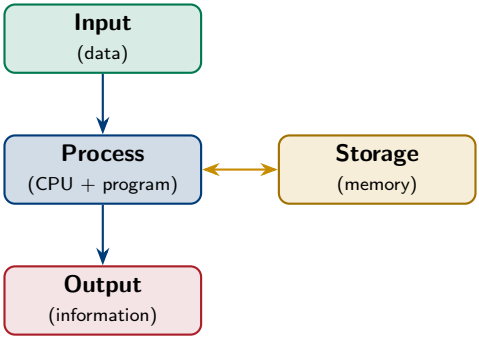


Figure: The Input–Process–Output (IPO) cycle.

Characteristics of a Computer

Why computers are so useful

- **Speed**: performs billions of operations per second.
- **Accuracy**: results are correct if input and logic are correct.
- **Diligence**: never gets tired or bored — no fall in performance.
- **Versatility**: same machine writes an email, plays music, trains an AI model.
- **Storage**: remembers huge amounts of data for years.
- **Automation**: runs a task on its own once started.

Limitations (be honest!)

- **No IQ**: a computer has no intelligence of its own — it only follows instructions.
- **GIGO**: *Garbage In, Garbage Out* — wrong input gives wrong output.
- **No feelings / common sense**: cannot judge like a human.
- Cannot correct its own logic errors.

Remember

A computer is **fast and obedient**, not **wise**. The wisdom must come from *you*, the programmer.

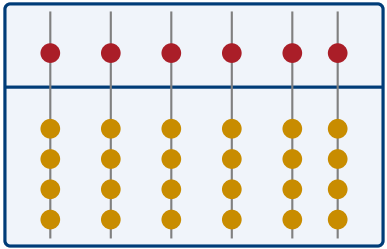
Early Counting & Calculating Tools

Before electronics — humans needed help to count

- **Abacus** (∼3000 BC, China): beads on rods; the first calculating tool. Still taught today for mental maths!
- **Napier's Bones** (1617): John Napier's rods for multiplication. He also invented *logarithms*.
- **Pascaline** (1642): Blaise Pascal's gear-based adding machine.
- **Leibniz Calculator** (1673): could add, subtract, multiply and divide.

Key idea

These were *mechanical* — gears and beads. No memory, no program. Yet they planted the seed: **machines can calculate**.



Abacus: beads = digits

Figure: The abacus — the ancestor of the calculator.

Charles Babbage — "Father of the Computer"

The visionary of the 1800s

- **Difference Engine** (1822): a mechanical machine to compute mathematical tables automatically.
- **Analytical Engine** (1837): the first design of a **general-purpose** computer. It already had the parts of a modern computer:

→ a *Mill* (like today's **CPU/ALU**),

→ a *Store* (like today's **memory**),

→ *punched cards* for input (from the Jacquard loom).

Ada Lovelace — the first programmer

She wrote the first algorithm intended for the Analytical Engine and saw that machines could handle *symbols*, not just numbers.

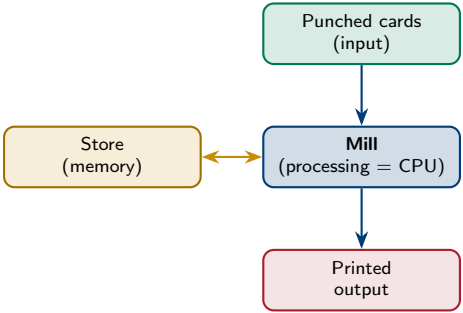


Figure: The Analytical Engine — a modern computer, 100+ years early!

The First Electronic Computers

From gears to electronics (1930s–1940s)

World War II drove the need for fast calculations (e.g. artillery tables, code-breaking). Machines moved from *mechanical* to *electronic*.

Landmark machines

- **Mark I** (1944): electro-mechanical, built at Harvard by Howard Aiken. Grace Hopper programmed it.
- **ENIAC** (1945): the first **general-purpose electronic** computer, by Eckert & Mauchly.
- **UNIVAC** (1951): the first **commercial** computer sold for business use.

ENIAC by the numbers

- $\sim$ **18,000** vacuum tubes
- Weighed $\sim$ **30 tonnes**, filled a large room
- $\sim$ 5,000 additions per second — about 1,000$\times$ faster than a mechanical calculator
- Programmed by *re-plugging cables* by hand!

Contrast

Your smartphone is *millions* of times more powerful and fits in your pocket.

The Stored-Program Idea (von Neumann, 1945)

The big breakthrough

Early machines were re-wired for each new task. **John von Neumann** proposed a decisive idea:

Store the program in memory, alongside the data.

- Programs become *data* that can be loaded and changed easily.
- No re-wiring — just load a new program.
- This is the **stored-program concept**, the basis of *every* computer today.

EDVAC / EDSAC

The first machines built on this idea (late 1940s). We study the full *von Neumann architecture* later in this unit.

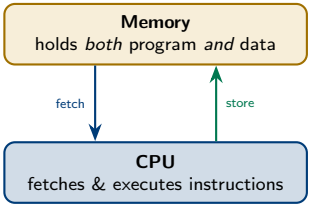
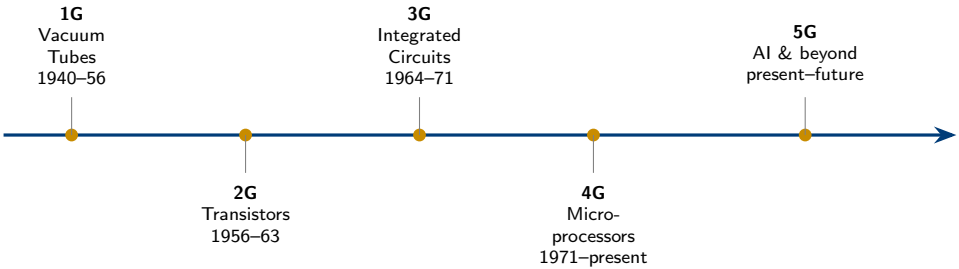


Figure: Program and data live together in memory.

Generations of Computers — The Big Picture



The pattern of progress

Each generation is defined by its **main electronic component**. Over time computers became **smaller**, **cheaper**, **faster**, **more reliable** and used **less power**. This trend continues today.

First & Second Generation

1st Generation (1940–1956): Vacuum Tubes

- Component: **vacuum tubes**.
- Huge, generated a lot of heat, frequent failures.
- Used *machine language* only; punched cards for input.
- Examples: *ENIAC, UNIVAC, IBM 650*.

Drawbacks

Very large, costly, unreliable, high power use.

2nd Generation (1956–1963): Transistors

- Component: **transistors** (invented 1947, Bell Labs).
- Smaller, faster, cheaper, more reliable, less heat.
- *Assembly language* and early high-level languages (FORTRAN, COBOL) appear.
- Examples: *IBM 1401, IBM 7090*.

Key upgrade

One transistor replaced many bulky tubes — the first big step in **miniaturisation**.

Third & Fourth Generation

3rd Generation (1964–1971): ICs

- Component: **Integrated Circuits (ICs)** — many transistors on one silicon chip (Kilby & Noyce, 1958–59).
- Even smaller, faster, cheaper; keyboards & monitors appear.
- *Operating systems* enable multiprogramming / time-sharing.
- Example: *IBM System/360*.

4th Generation (1971–present): Microprocessors

- Component: **microprocessor** — the *whole CPU* on a single chip (Intel 4004, 1971).
- Made possible by **VLSI** (thousands→millions of transistors per chip).
- Gave birth to the **Personal Computer (PC)** and the GUI.
- Examples: *Intel 8086, Apple, IBM PC*.

Moore's Law (1965): the number of transistors on a chip roughly *doubles* about every two years.

Fifth Generation & Summary Table

5th Generation (present & future): Artificial Intelligence

Based on **ULSI**, parallel processing and AI. Goal: machines that can **reason, learn and understand** natural language — exactly the subject of Units 4 & 5 of this course!

Gen	Component	Feature	Example
1st	Vacuum tubes	Huge, hot, machine code	ENIAC, UNIVAC
2nd	Transistors	Smaller, assembly lang.	IBM 1401
3rd	Integrated circuits	OS, time-sharing	IBM System/360
4th	Microprocessors	PCs, GUI, VLSI	Intel 8086, IBM PC
5th	AI / ULSI	Learning, reasoning	Modern AI systems

Trend: smaller · faster · cheaper · more reliable · less power — generation after generation.

Classification of Computers by Size

By size & power

- **Supercomputer**: fastest, for weather forecasting, research, AI training. (India: *PARAM, AIRAWAT*.)
- **Mainframe**: large organisations — banks, railways — serving thousands of users.
- **Minicomputer**: mid-sized, for departments / small firms.
- **Microcomputer**: personal computers, laptops, tablets, smartphones.

By purpose & by data type

Purpose:

- *General-purpose* — your laptop (many tasks).
- *Special-purpose* — ATM, washing-machine chip (one task).

Data handled:

- *Digital* — works on discrete 0/1 (most computers).
- *Analog* — works on continuous signals (e.g. speedometer).
- *Hybrid* — both (e.g. hospital ICU monitors).

Quick Check — Round 1 (History)

[Q1] Who is called the "Father of the Computer", and what was his general-purpose machine called?

[Q2] Name the main electronic component of each generation: 1st, 2nd, 3rd, 4th.

[Q3] What was the key idea of the "stored-program concept"?

[Q4] True/False: A computer has intelligence of its own.

Think for 60 seconds — answers on the next slide.

Answers — Round 1

[A1]

Charles Babbage; his general-purpose machine was the **Analytical Engine** (1837).

[A2]

1st = **vacuum tubes**, 2nd = **transistors**, 3rd = **integrated circuits**, 4th = **microprocessors**.

[A3]

Store the *program in memory along with the data*, so the machine can be re-used for a new task just by loading a new program — no re-wiring.

[A4]

False. A computer only follows instructions (*GIGO* — garbage in, garbage out). Intelligence must be programmed.

The Functional Units of a Computer

Five basic units

1. **Input unit**: takes data in (keyboard, mouse, scanner).
2. **Memory unit**: stores data & instructions.
3. **ALU**: does arithmetic & logic.
4. **Control unit**: directs all operations.
5. **Output unit**: gives results out (monitor, printer).

ALU + Control Unit + Registers = the **CPU** (the "brain").

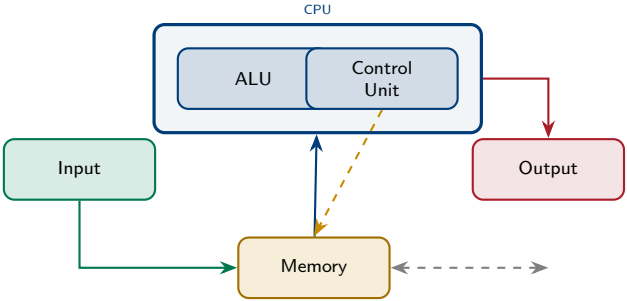


Figure: Solid = data flow, dashed = control signals.

The von Neumann Architecture

The model behind (almost) all computers

A single memory holds **both** instructions and data, connected to the CPU by a set of **buses**.

- **CPU** = Control Unit + ALU + Registers.
- **Memory** = main memory (RAM).
- **I/O** = input & output devices.
- **Buses** = wires that carry data, addresses & control signals.

The "von Neumann bottleneck"

Since data *and* instructions share one path to memory, that path can become a traffic jam — a limit on speed.

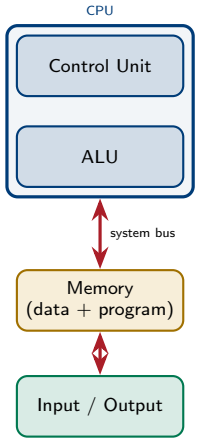


Figure: CPU ↔ Memory ↔ I/O over a shared bus.

Inside the CPU

The three parts of the CPU

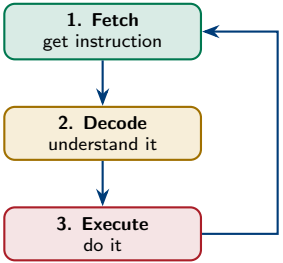
- **Control Unit (CU)**: the "manager". Fetches and decodes instructions, tells other units what to do.
- **Arithmetic Logic Unit (ALU)**: the "calculator". Does *arithmetic* ($+, -, \times, \div$) and *logic* ($>, <, =$, AND, OR, NOT).
- **Registers**: tiny, super-fast storage *inside* the CPU for the values being worked on right now.

Clock speed

The CPU is driven by a **clock**. A speed of 3 GHz means 3 *billion* ticks per second — each tick can trigger work.

The Fetch–Decode–Execute Cycle

The CPU repeats this loop billions of times per second:



The Memory Hierarchy

Fast-but-small vs. slow-but-large

As we go **up** the pyramid: faster, costlier, smaller.
As we go **down**: slower, cheaper, larger.

- **Registers** — inside CPU, fastest.
- **Cache** — small, very fast buffer near CPU.
- **Main memory (RAM)** — holds running programs; *volatile*.
- **Secondary storage** — SSD/HDD; *non-volatile*, permanent.

RAM vs. ROM

RAM = temporary working memory (lost on power off). *ROM* = permanent start-up instructions (kept on power off).

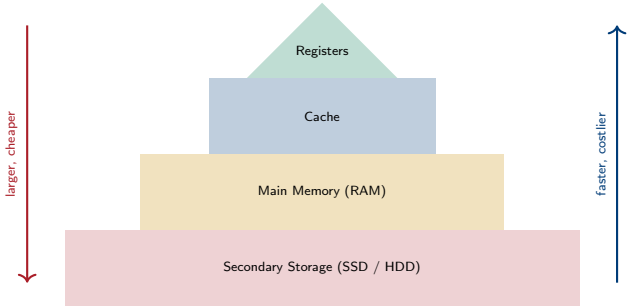


Figure: The memory hierarchy pyramid.

Units of Memory

Counting in bits and bytes

- **Bit** = a single *b*inary dig*it*: 0 or 1.
- **Nibble** = 4 bits.
- **Byte** = 8 bits — stores one character.
- **Word** = the natural chunk a CPU handles (e.g. 32 or 64 bits).

A handy rule

1 byte $\approx$ one letter. So the word "`INDIA`" takes **5 bytes**.

Unit	Size
1 Byte	8 bits
1 Kilobyte (KB)	1024 Bytes
1 Megabyte (MB)	1024 KB
1 Gigabyte (GB)	1024 MB
1 Terabyte (TB)	1024 GB
1 Petabyte (PB)	1024 TB

Each step up is $\times 1024$ $(= 2^{10})$, because computers count in powers of 2.

Input, Output & Storage Devices

Input devices

Feed data *in*:

- Keyboard, mouse
- Scanner, webcam
- Microphone
- Touchscreen

Output devices

Give results *out*:

- Monitor / screen
- Printer
- Speaker
- Projector

Storage devices

Keep data for later:

- SSD, Hard disk
- USB pen drive
- Memory card
- DVD / CD

Some devices do both!

A **touchscreen** is input *and* output. A **hard disk** both reads (in) and writes (out) — so it is an I/O device.

Hardware vs. Software

Hardware

The **physical** parts you can *touch*.

- CPU, RAM, hard disk
- Keyboard, monitor, motherboard

"The body of the computer."

Software

The **programs** — instructions you *cannot* touch.

- **System software**: OS (Windows, Linux, Android), compilers, device drivers.
- **Application software**: browser, MS Word, games, WhatsApp.

"The soul of the computer."

The Operating System (OS)

The most important system software — it manages the hardware and acts as a **bridge** between you and the machine.

- Manages memory, files, processes, devices.
- Examples: Windows, Linux, macOS, Android.

Analogy

Hardware is a *stage*; software is the *play*. Without a play, the stage does nothing. Without a stage, the play cannot happen.

Quick Check — Round 2 (Basics & Organisation)

[Q1] Name the three parts of the CPU.

[Q2] How many bits are there in 1 byte, and how many bytes in 1 KB?

[Q3] State the three steps of the machine cycle (the CPU's basic loop).

[Q4] Which is volatile: RAM or ROM? What does "volatile" mean?

Answers — Round 2

[A1]

Control Unit (CU), **Arithmetic Logic Unit (ALU)**, and **Registers**.

[A2]

1 byte = **8 bits**; 1 KB = **1024 bytes** ($= 2^{10}$ bytes).

[A3]

Fetch → **Decode** → **Execute** (then repeat).

[A4]

RAM is volatile — "volatile" means its contents are *lost when power is switched off*. ROM keeps its contents.

Why Computers Use Binary

Two states are easy to build

A computer is made of electronic switches (transistors). A switch is naturally in one of **two** states:

- ON = high voltage = **1**
- OFF = low voltage = **0**

So computers store *everything* — numbers, text, images, sound, AI models — as patterns of **0s and 1s**: the **binary** system.

Reliability

Two clearly-separated voltage levels are hard to confuse, so binary is very *robust* to electrical noise. Ten levels (decimal) would be far harder to tell apart.

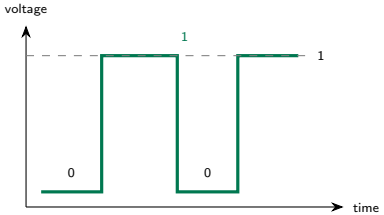


Figure: A digital signal is just HIGH (1) or LOW (0).

The Four Number Systems

System	Base	Digits used	Example
Decimal	10	0–9	$(759)_{10}$
Binary	2	0, 1	$(1011)_2$
Octal	8	0–7	$(327)_8$
Hexadecimal	16	0–9, A–F	$(2F)_{16}$

The base tells you two things

- how many different digits are allowed, and
- the "weight" of each position.

Hexadecimal letters

In hex, we run out of digits after 9, so we use letters: A=10, B=11, C=12, D=13, E=14, F=15.

Positional Value — The Key Idea

Every digit has a place value = (base)$^{\text{position}}$

Positions are counted from **0 on the right**, increasing to the left.

Decimal: $(759)_{10}$

$$\begin{array}{ccc} 7 & 5 & 9 \\ 10^2 & 10^1 & 10^0 \end{array}$$

$$= 7\times100 + 5\times10 + 9\times1$$
$$= 700 + 50 + 9 = 759.$$

Binary: $(1011)_2$

$$\begin{array}{cccc} 1 & 0 & 1 & 1 \\ 2^3 & 2^2 & 2^1 & 2^0 \end{array}$$

$$= 1\times8 + 0\times4 + 1\times2 + 1\times1$$
$$= 8 + 0 + 2 + 1 = (11)_{10}.$$

The same rule works for every base — only the base number changes.

Conversion 1 — Any Base → Decimal

Method: multiply each digit by its place value, then add

This works for binary, octal and hex — just use the right base.

Binary → Decimal: $(1101)_2$

$= 1\times2^3 + 1\times2^2 + 0\times2^1 + 1\times2^0$
$= 8 + 4 + 0 + 1 = \mathbf{(13)}_{10}$

Octal → Decimal: $(327)_8$

$= 3\times8^2 + 2\times8^1 + 7\times8^0$
$= 192 + 16 + 7 = \mathbf{(215)}_{10}$

Hex → Decimal: $(2F)_{16}$

Remember F $= 15$.
$= 2\times16^1 + 15\times16^0$
$= 32 + 15 = \mathbf{(47)}_{10}$

Watch out

A digit must be *valid* for its base. $(1012)_2$ is **illegal** — binary has no digit "2".

Conversion 2 — Decimal → Any Base

Method: repeated division; read remainders *bottom to top*

Divide by the base again and again; collect remainders; the last remainder is the **most significant** (left-most) digit.

Decimal → Binary: $(25)_{10}$

$25 \div 2 = 12$ rem **1**
$12 \div 2 = 6$ rem **0**
$6 \div 2 = 3$ rem **0**
$3 \div 2 = 1$ rem **1**
$1 \div 2 = 0$ rem **1**

Read up: $\mathbf{(11001)_2}$.

Decimal → Hex: $(255)_{10}$

$255 \div 16 = 15$ rem **15 = F**
$15 \div 16 = 0$ rem **15 = F**

Read up: $\mathbf{(FF)_{16}}$.

Check: $F \times 16 + F = 240 + 15 = 255$. ✓

Conversion 3 — Binary ↔ Octal / Hex (Shortcut)

Group the bits — no division needed!

Because $8 = 2^3$ and $16 = 2^4$: group binary digits into **3s for octal** and **4s for hex**, from the right.

Binary → Octal

$(11010111)_2$, group in 3s from right; pad the left group with 0s:

$$\underbrace{011}_{3}\ \underbrace{010}_{2}\ \underbrace{111}_{7}$$

$\Rightarrow$ **$(327)_8$**.

Binary → Hex

$(11010111)_2$, group in 4s from right:

$$\underbrace{1101}_{D}\ \underbrace{0111}_{7}$$

$\Rightarrow$ **$(D7)_{16}$**.

Reverse is just as easy

Hex $(2F)_{16}$: write each digit as 4 bits — $2 = 0010$, $F = 1111 \Rightarrow (00101111)_2$.

Binary–Octal–Decimal–Hex Reference Table

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

Memorising 0–15 in all four systems makes every conversion fast. Notice hex counts 0–F, then "carries" to 10.

Binary Addition

The four basic rules

$$0 + 0 = 0$$
$$0 + 1 = 1$$
$$1 + 0 = 1$$
$$1 + 1 = 10 \quad (0, \text{ carry } 1)$$

Just like decimal, a **carry** moves to the next column on the left.

Check in decimal

The example on the right: $13 + 6 = 19$. And $(10011)_2 = 16 + 2 + 1 = 19$. ✓

Worked example: $1101 + 0110$

$$\begin{array}{r} {}^{1}1\,{}^{1}1\,0\,1 \\ +\ \ 0\,1\,1\,0 \\ \hline 1\,0\,0\,1\,1 \end{array}$$

Column by column (right to left):

- $1 + 0 = 1$
- $0 + 1 = 1$
- $1 + 1 = 0$, carry 1
- $1 + 0 +$ carry $1 = 0$, carry 1
- write the final carry 1

Result: $\mathbf{(10011)_2}$.

Representing Negative Numbers

The computer has no "−" key — only 0s and 1s

We reserve the **left-most bit** as a **sign bit**: **0 = positive**, **1 = negative**. Three schemes exist:

The three schemes (for -5, 8-bit)

- **Sign-Magnitude**: sign bit + value.
  $+5 = 00000101$, $-5 = 10000101$.
- **1's Complement**: flip every bit of $+5$.
  $-5 = 11111010$.
- **2's Complement**: 1's complement $+1$.
  $-5 = 11111011$.

Why 2's complement wins

Modern computers use **2's complement** because:

- subtraction becomes plain *addition*,
- there is only *one* zero (no "-0"),
- the hardware is simpler.

2's complement in 2 steps

(1) Invert all bits. (2) Add 1. That's the whole trick.

2's Complement Subtraction — Worked Example

Compute $7 - 5$ using 4-bit 2's complement

The idea: $7 - 5 = 7 + (-5)$. So we *add* the 2's complement of 5.

Step 1–2: find -5

$+5 = 0101$

Invert: 1010

Add 1: $1010 + 1 = \mathbf{1011} \quad (= -5)$

Step 3: add to $+7$

$$\begin{array}{rr} & 0\,1\,1\,1 \quad (+7) \\ + & 1\,0\,1\,1 \quad (-5) \\ \hline & 1\,0\,0\,1\,0 \end{array}$$

Discard the carry out of the 4th bit $\Rightarrow 0010 = \mathbf{2}$.

Result $= (0010)_2 = 2$**, which is exactly** $7 - 5$**.** Subtraction done with only an adder!

What is a Binary Code?

Coding = agreeing on a pattern

A **binary code** is an agreed way to represent numbers, letters or symbols using groups of **0s and 1s**.

- To store the letter `A`, everyone must agree on *which* bit pattern means `A`.
- Codes let different computers exchange data correctly.

Two families of codes

- **Numeric codes** — represent numbers (e.g. BCD, Gray).
- **Alphanumeric codes** — represent text: letters, digits, symbols (e.g. ASCII, Unicode, EBCDIC).

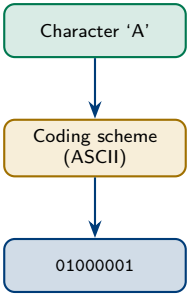


Figure: A code turns a character into a fixed bit pattern.

BCD — Binary Coded Decimal

Each decimal digit → its own 4 bits

Instead of converting the whole number, code *each digit separately* using 4 bits.

$$(59)_{10} \Rightarrow \underbrace{0101}_{5}\ \underbrace{1001}_{9} = (0101\,1001)_{\text{BCD}}$$

Useful in **calculators, digital clocks and meters**, where digits are shown one at a time.

BCD is not pure binary

Pure binary of $(59)_{10}$ is 111011. BCD of 59 is 01011001 — *different!* BCD wastes some patterns (1010–1111 are unused).

Digit	BCD	Digit	BCD
0	0000	5	0101
1	0001	6	0110
2	0010	7	0111
3	0011	8	1000
4	0100	9	1001

BCD uses only patterns 0000–1001, one per decimal digit.

ASCII — Coding Text

American Standard Code for Information Interchange

- The most common code for English text.
- Original ASCII uses **7 bits** $\Rightarrow 2^7 = 128$ characters. Extended ASCII uses **8 bits** $\Rightarrow 256$.
- Covers A–Z, a–z, 0–9, punctuation, and control codes (Enter, Tab, Space).

Handy landmarks

- `'A'` $= 65$, `'B'` $= 66$, ... (uppercase)
- `'a'` $= 97$, `'b'` $= 98$, ... (lowercase)
- `'0'` $= 48$, `'1'` $= 49$, ... (digit characters)

Notice `'a'` $-$ `'A'` $= 32$ — the "case" gap.

Example: the word "Hi"

Char	Decimal	Binary (8-bit)
H	72	01001000
i	105	01101001

Careful!

The character `'5'` is *not* the number 5. `'5'` has ASCII code 53. They are stored differently.

Unicode & EBCDIC

Unicode — text for every language

ASCII cannot represent Hindi, Tamil, Chinese or emoji. **Unicode** was created to encode **every** writing system in the world.

- Can represent over a million characters.
- **UTF-8** is the popular form; it is *backward compatible* with ASCII.
- Powers the multilingual web, WhatsApp, and . . . emoji.

Example

Hindi (Devanagari) letters, Tamil script and emoji each have their own *Unicode code point* — ASCII simply has no room for them, but Unicode does.

EBCDIC

Extended Binary Coded Decimal Interchange Code:

- An 8-bit code developed by *IBM*.
- Used mainly on IBM **mainframe** computers.
- Largely replaced by ASCII / Unicode on modern machines.

Quick comparison

ASCII $\rightarrow$ 7/8-bit, universal on PCs. EBCDIC $\rightarrow$ 8-bit, IBM mainframes. Unicode $\rightarrow$ all languages, modern standard.

Gray Code — The "One Bit Changes" Code

A code where neighbours differ by only 1 bit

In **Gray code**, moving from one number to the next flips **exactly one bit**. This avoids errors in position sensors and rotary encoders.

Why it matters

In ordinary binary, going $3 \to 4$ is $011 \to 100$ — *three* bits change at once. If the sensor reads them at slightly different instants, it may briefly show a wrong value. Gray code changes only one bit, so this cannot happen.

Dec	Binary	Gray
0	000	000
1	001	001
2	010	011
3	011	010
4	100	110
5	101	111
6	110	101
7	111	100

Read down the Gray column: each row differs from the previous by just one bit.

How a Computer Stores a Photo (Putting It Together)

Everything ends up as bits

A colour image is a grid of tiny dots called **pixels**. Each pixel's colour is stored as three numbers — **Red, Green, Blue** — each 0–255, i.e. **8 bits** each.

- 1 pixel $= 3 \times 8 = 24$ bits $= 3$ bytes.
- Pure red $= (255, 0, 0) = (11111111, 00000000, 00000000)$.
- A 1000×1000 image $= 3$ million bytes ≈ 3 MB.

The unifying idea

Numbers, text, images, sound and AI models — *all* are just different **binary codes**. That is the power of binary.

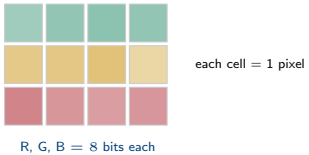


Figure: A picture is a grid of pixels, each a triple of bytes.

Quick Check — Round 3 (Data Representation)

[Q1] Convert $(1010)_2$ to decimal, and $(20)_{10}$ to binary.

[Q2] What is $(2C)_{16}$ in decimal?

[Q3] Write $(37)_{10}$ in BCD.

[Q4] How many characters can 7-bit ASCII represent, and what is the ASCII code of `'A'`?

Try them on paper — answers next slide.

Answers — Round 3

[A1]

$(1010)_2 = 8 + 2 = \mathbf{(10)_{10}}$. $(20)_{10}$: $20 \div 2$ remainders give $\mathbf{(10100)_2}$.

[A2]

$(2C)_{16} = 2 \times 16 + 12 = 32 + 12 = \mathbf{(44)_{10}}$ (C $= 12$).

[A3]

Code each digit in 4 bits: $3 = 0011$, $7 = 0111 \Rightarrow$ **0011 0111**.

[A4]

$2^7 =$ **128** characters; `'A'` = **65** ($= 01000001$).

Talking to the Machine

The communication gap

- Humans think in **words and ideas** (English, maths).
- The CPU understands only **machine language** — pure binary.
- A **programming language** is the bridge between the two.

Over time, languages moved *closer to humans* and *further from raw binary* — making programming easier.

Three levels

- **Machine language** (1st gen) — binary.
- **Assembly language** (2nd gen) — short mnemonics.
- **High-level languages** (3rd gen onward) — near-English.

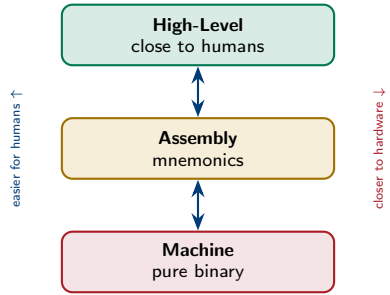


Figure: The ladder of programming languages.

Low-Level: Machine Language (1st Generation)

The only language the CPU truly understands

- Written in **binary** (0s and 1s) — opcodes and addresses.
- **Fastest** to run: no translation needed.
- **Machine-dependent**: code for one CPU won't run on another.

What it looks like

An instruction "add two numbers" might be:

```
10110000 01100001
```

Readable? Not at all — yet this is exactly what the CPU executes.

Very hard for humans

A single mistake — one wrong bit — breaks the program, and it is almost impossible to read or debug. Nobody writes large programs this way today.

Pros & cons

+ fastest execution, direct hardware control.
– error-prone, unreadable, not portable.

Low-Level: Assembly Language (2nd Generation)

Binary replaced by short mnemonics

Instead of 10110000, we write memory-joggers called **mnemonics**: `ADD`, `SUB`, `MOV`, `JMP`.

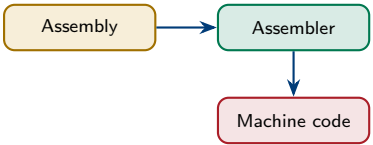
- Much easier to read and write than binary.
- Still **machine-dependent** (tied to one CPU family).
- Needs an **assembler** to translate it into machine code.

Where it is still used

Device drivers, embedded systems, and places needing tight speed or direct hardware access.

Assembly vs. machine code

Assembly	Machine
`MOV AL, 61h`	`10110000 01100001`
`ADD AL, BL`	`00000000 11011000`



High-Level Languages (3rd Generation onward)

Near-English, easy and portable

High-level languages (HLLs) use words and maths close to human language.

- **Easy** to learn, read, write and debug.
- **Portable**: the same code runs on many machines (after re-translation).
- **Machine-independent**: you needn't know the CPU details.
- Need a **compiler** or **interpreter** to translate.

Familiar examples

C, C++, Java, **Python**, JavaScript. In this course you will use Python for AI and ML.

"Add two numbers" in each level

Level	Code
High-level	`c = a + b`
Assembly	`ADD AX, BX`
Machine	`00000011...`

Trade-off

HLLs are easier but slightly *slower* than assembly, because a translator sits in between.

Language Translators

Turning your code into machine code

Source code (what you write) must be translated into machine code (what the CPU runs). Three translators do this:

- **Assembler**: assembly → machine code.
- **Compiler**: whole HLL program → machine code *at once*. (C, C++)
- **Interpreter**: translates and runs *line by line*. (Python)

Compiler vs. Interpreter

Compiler	**Interpreter**
Whole program at once	Line by line
Faster execution	Slower execution
Errors shown at end	Stops at first error
e.g. C, C++	e.g. Python

Analogy

A *compiler* is like translating a whole book before handing it over. An *interpreter* is like a live translator speaking sentence by sentence.

The Generations of Programming Languages

Gen	Type	Nature	Example
1GL	Machine language	Binary; CPU-native	raw 0s and 1s
2GL	Assembly language	Mnemonics	`ADD`, `MOV`
3GL	High-level (procedural)	Near-English	C, C++, Java, Python
4GL	Very high-level	Problem-oriented	SQL, MATLAB
5GL	AI / declarative	State *what*, not *how*	Prolog, LISP

The clear trend

Each generation is **closer to human thinking** and **further from binary**.

Looking ahead

5GL and AI languages connect directly to Units 4 & 5, where you will make machines *reason and learn*.

Quick Check — Round 4 (Languages)

[Q1] Which language does the CPU directly understand?

[Q2] What is the job of an *assembler*?

[Q3] State one key difference between a compiler and an interpreter.

[Q4] Match: (i) `MOV AL, 5` (ii) `c=a+b` (iii) `10110000` with (a) machine (b) assembly (c) high-level.

Answers — Round 4

[A1]

Machine language (pure binary) — the only language the CPU runs without translation.

[A2]

An assembler translates **assembly language** (mnemonics) into **machine code**.

[A3]

A **compiler** translates the *whole* program at once; an **interpreter** translates and runs it *line by line*. (Compiled code runs faster.)

[A4]

(i)–(b) assembly, (ii)–(c) high-level, (iii)–(a) machine.

Summary of Unit 1

What we covered

1. **History** — from the abacus and Babbage's Analytical Engine to ENIAC and the stored-program idea (von Neumann).
2. **Generations** — vacuum tubes → transistors → ICs → microprocessors → AI; smaller, faster, cheaper each time.
3. **Computer basics** — five units, the CPU (CU + ALU + registers), the fetch–decode–execute cycle, and the memory hierarchy.
4. **Number systems** — decimal, binary, octal, hex; conversions in all directions; binary arithmetic and 2's complement.
5. **Binary codes** — BCD, ASCII, Unicode, EBCDIC and Gray code.
6. **Programming languages** — machine, assembly and high-level; assemblers, compilers and interpreters.

Everything a computer does is built on hardware and binary — the foundation for all of AI and ML ahead.

Key Terms to Remember

Hardware & history

- **CPU** = CU + ALU + registers
- **RAM** (volatile) vs. **ROM** (permanent)
- **Bit, byte, KB, MB, GB**
- **von Neumann** = stored-program model
- **Moore's Law** = transistors double $\sim$2 yrs

Data & languages

- **Base**: 2 (binary), 8 (octal), 16 (hex)
- **2's complement** = invert then $+1$
- **ASCII**: `'A'`$= 65$, `'a'`$= 97$
- **BCD**: each digit $\rightarrow$ 4 bits
- **Compiler** (all at once) vs. **Interpreter** (line by line)

Practice Questions (Exam Oriented)

Short answer (2–5 marks)

1. List the four characteristics and two limitations of a computer.
2. Name the components of each generation of computers (1st–4th).
3. Draw and label the von Neumann architecture.
4. Convert $(45)_{10}$ to binary, octal and hexadecimal.
5. Differentiate between RAM and ROM.

Long answer (8–10 marks)

6. Explain the fetch–decode–execute cycle with a diagram.
7. Perform $(13)_{10} - (7)_{10}$ using 5-bit 2's complement, showing all steps.
8. Compare machine, assembly and high-level languages on at least five points.
9. Explain BCD, ASCII, Unicode and Gray code with one example each.
10. Differentiate compiler, interpreter and assembler with a diagram.

Think & Explore (Take-Home)

Mini tasks

- Write your *name* in ASCII (decimal codes for each letter).
- Convert your *roll number* into binary and hexadecimal.
- Find the RAM, storage and CPU speed of your own phone/laptop — identify each unit you studied.
- Add $(1011)_2 + (1101)_2$ and verify in decimal.

Reading & reflection

- Read the chapter on *Data Representation* in Peter Norton's *Introduction to Computers*.
- Watch a short video on how ENIAC worked — note how far we have come.
- Prepare for Unit 2: *visual thinking, flowcharts and pseudocode*.

Reminder

Post any doubts on the course page. Assessment-1 will cover Units 1 & 2.

References

Textbooks (as per the course page)

1. Peter Norton, *Introduction to Computers*, Tata McGraw Hill, 6th Edition. **[Main text]**
2. Reema Thareja, *Fundamentals of Computers*, Oxford University Press.
3. P. K. Sinha & P. Sinha, *Computer Fundamentals*, BPB Publications.
4. V. Rajaraman, *Fundamentals of Computers*, PHI Learning.

Further reading

5. M. Morris Mano, *Digital Logic and Computer Design*, Pearson (number systems & codes).
6. A. S. Tanenbaum, *Structured Computer Organization*, Pearson.

Course page: https://www.gcjana.in/courses/shardauniversity/2601/fundamentals-of-computing-and-artificial-intelligence/

Thank You

Any Questions?

"The computer was born to solve problems that did not exist before."
— Bill Gates

Post your doubts on the course page →
https://www.gcjana.in/courses/shardauniversity/2601/fundamentals-of-computing-and-artificial-intelligence/#Post-doubts