Computer Organization

IB SL Study Guide

Contents

CPU Architecture

The Central Processing Unit (CPU) is the component that executes program instructions. It consists of three main sub-units, connected internally and to the rest of the computer via a set of electrical pathways called **buses**.

ALU, Control Unit, and Registers

Component	Full Name	Function
ALU	Arithmetic Logic Unit	Performs arithmetic (add, subtract, multiply) and logical (AND, OR, NOT, XOR, compare) operations
CU	Control Unit	Fetches instructions from memory, decodes them, and coordinates execution across the CPU
Registers	—	Ultra-fast temporary storage locations inside the CPU; each has a specific role

Key Registers

Register	Name	Role
PC	Program Counter	Holds the memory address of the **next** instruction to be fetched
IR	Instruction Register	Holds the **current** instruction being decoded and executed
MAR	Memory Address Register	Holds the memory address the CPU wants to read from or write to
MDR	Memory Data Register	Holds the data just read from memory or about to be written to memory
ACC	Accumulator	General-purpose register storing the result of ALU operations

MEMORISE THIS

Register roles — PC points ahead, IR holds now: PC = next instruction's address; IR = current instruction; MAR = address on the bus; MDR = data on the bus; ACC = ALU result.

System Buses

The CPU communicates with memory and I/O devices through three buses:

Bus	Direction	Carries
Address bus	CPU → memory / I/O (one-way)	The memory address to be accessed
Data bus	Bidirectional	The actual data or instruction being transferred
Control bus	Bidirectional	Control signals (read/write, clock, interrupt, reset)

The **clock** generates a regular pulse that synchronises all CPU operations. Clock speed is measured in GHz (gigahertz); more pulses per second means more instructions can be executed per second. However, more cores, cache size, and instruction set efficiency also affect overall performance.

> ⚠ **EXAM ALERT**
>
> The IB exam sometimes asks about factors that affect CPU performance. The standard three are: **clock speed** (GHz), **number of cores**, and **cache size**. Do not confuse clock speed with bus speed — they are separate specifications.

Memory Hierarchy

Memory is organised in a hierarchy from fastest/smallest/most expensive to slowest/largest/cheapest.

The memory hierarchy from fastest to slowest is: **Registers → Cache → RAM → Secondary Storage**

Level	Speed	Capacity	Cost per GB	Volatile?
Registers	Fastest	Bytes	Very high	Yes
Cache (L1/L2/L3)	Very fast	KB – MB	High	Yes
RAM	Fast	GB	Moderate	Yes
SSD	Moderate	GB – TB	Low-moderate	No
HDD	Slow	GB – TB	Low	No
Optical / USB	Slow	GB	Very low	No

Why the hierarchy exists: Faster memory is physically more expensive to manufacture. The CPU uses a small amount of very fast memory (registers and cache) for immediate work, backed by larger but slower RAM for active programs, and even larger secondary storage for persistent data.

> 💡 **IB TIP**
>
> In Paper 1 questions, if asked to "compare two levels of the memory hierarchy", structure your answer as: speed (faster/slower), capacity (more/less), cost (higher/lower), and volatility (volatile/non-volatile). Four attributes — four marks.

Cache Memory

Cache is a small, very fast volatile memory that sits between the CPU and RAM. When the CPU needs data, it checks the cache first (**cache hit**). If not found (**cache miss**), it retrieves from RAM and stores a copy in cache for future use.

- **L1 cache** — smallest and fastest; inside the CPU core
- **L2 cache** — larger, slightly slower; often still inside the CPU
- **L3 cache** — largest, shared between cores

Secondary Storage

Secondary storage is non-volatile and retains data when power is removed. It is used for long-term storage of files, programs, and the operating system.

Type	Technology	Typical Use	Advantage	Disadvantage
HDD	Magnetic spinning platters	Bulk data storage, OS	Low cost per GB, large capacity	Mechanical parts = slower, fragile if dropped
SSD	NAND flash (no moving parts)	OS drive, fast applications	Fast access, silent, durable	More expensive per GB than HDD
Optical (CD/DVD/Blu-ray)	Laser reads/writes pits on disc	Distribution, archiving	Cheap per disc, long shelf life	Slow, low capacity, easily scratched
USB Flash	NAND flash	Portable transfer	Compact and portable	Small capacity vs. HDD, wears out over many writes

⚠ EXAM ALERT

"State one advantage of an SSD over an HDD" is a common short-answer question. Accept: faster read/write speeds; no moving parts so more durable/reliable; quieter; lower power consumption. Do NOT simply write "better" — the mark scheme requires a specific attribute.

The Fetch-Execute Cycle

The CPU continuously repeats a three-stage cycle to process instructions stored in RAM. Each iteration processes one instruction.

The Three Stages

1. Fetch

- The address stored in the PC is copied to the MAR
- The instruction at that memory address is retrieved and placed in the MDR
- The instruction is copied from MDR to the IR
- The PC is incremented to point to the next instruction

2. Decode

- The Control Unit interprets the binary instruction held in the IR
- It identifies the operation type (load, store, add, jump, etc.) and any operands

3. Execute

- The CU activates the appropriate hardware:
    - If arithmetic/logic: data is sent to the ALU; result stored in ACC

- If memory read: MAR gets the data address; data arrives in MDR, then moves to ACC or a register
- If memory write: data from ACC is placed in MDR; MAR holds the destination address; write signal sent
- If jump: PC is updated to the jump target address instead of incrementing

WORKED EXAMPLE

Worked Example — Full cycle trace:

Memory contents:

Address	Contents
100	LOAD 200 (load the value at address 200 into ACC)
101	ADD 201 (add the value at address 201 to ACC)
200	15
201	7

PC starts at 100.

Cycle 1 — LOAD 200:

- Fetch: MAR ← 100; MDR ← LOAD 200; IR ← LOAD 200; PC ← 101
- Decode: CU identifies a memory-read instruction, operand = 200
- Execute: MAR ← 200; MDR ← 15; ACC ← 15

Cycle 2 — ADD 201:

- Fetch: MAR ← 101; MDR ← ADD 201; IR ← ADD 201; PC ← 102
- Decode: CU identifies an addition, operand = 201
- Execute: MAR ← 201; MDR ← 7; ALU computes 15 + 7 = 22; ACC ← 22

⚠ EXAM ALERT

A common Paper 1 error is forgetting that the PC is incremented **during the fetch stage**, not after the execute stage. Also remember: the MAR always receives an address; the MDR always receives data. Swapping these in an answer loses marks.

Binary and Hexadecimal Representation

All data and instructions inside a computer are stored and processed in binary — patterns of 0s and 1s (bits). Hexadecimal (base 16) is used as a compact human-readable shorthand for binary.

Unsigned Binary

Each bit position represents a power of 2, from 2^0 (rightmost) upward.

8-bit place values:

128 64 32 16 8 4 2 1

WORKED EXAMPLE

Binary to decimal — convert `10110011`:

```
128 64 32 16 8 4 2 1
1   0  1  1  0 0 1 1
```

Value = $128 + 32 + 16 + 2 + 1 = 179$

Decimal to binary — convert 45:

$45 \div 2 = 22$ remainder **1** (LSB) $22 \div 2 = 11$ remainder **0** $11 \div 2 = 5$ remainder **1** $5 \div 2 = 2$ remainder **1** $2 \div 2 = 1$ remainder **0** $1 \div 2 = 0$ remainder **1** (MSB)

Read remainders from bottom to top: **101101** = 45

Hexadecimal

Hexadecimal uses base 16, with digits 0–9 and A–F (where A = 10, B = 11, C = 12, D = 13, E = 14, F = 15). One hex digit represents exactly 4 binary bits (a **nibble**), making hex a compact notation for binary.

MEMORISE THIS

Hex–binary nibble table (memorise these):

Hex	Binary	Decimal
0	0000	0
1	0001	1
…	…	…
9	1001	9
A	1010	10
B	1011	11
C	1100	12
D	1101	13
E	1110	14
F	1111	15

WORKED EXAMPLE

Binary to hex — convert `11001010`:

Split into nibbles: `1100` `1010`

`1100` = C, `1010` = A

Result: **CA** (or 0xCA)

Hex to binary — convert `3F`:

`3` = 0011, `F` = 1111

Result: **00111111**

Hex to decimal — convert `2B`:

$2 \times 16^1 + 11 \times 16^0 = 32 + 11 = 43$

Two's Complement

Two's complement is the standard method computers use to represent **negative integers** in binary.

Representing Negative Numbers

For an n-bit two's complement system:

- Positive numbers: represented normally in binary (leading bit = 0)
- Negative numbers: the most significant bit (MSB) has a **negative** place value of -2^{n-1}
- Range for n bits: -2^{n-1} to $+2^{n-1} - 1$

For 8-bit two's complement: range is -128 to $+127$.

Converting a Positive Integer to Its Negative Two's Complement

Method: **invert all bits, then add 1**

WORKED EXAMPLE

Find the 8-bit two's complement of -35:

Step 1 — Represent +35 in binary: `00100011`

Step 2 — Invert all bits: `11011100`

Step 3 — Add 1: `11011101`

Verification: $-128 + 64 + 16 + 8 + 4 + 1 = -128 + 93 = -35$ ✓

Reading a two's complement number `11110000`:

MSB place value = -128

$-128 + 64 + 32 + 16 = -128 + 112 = -16$

EXAM ALERT

The range of an 8-bit two's complement system is -128 to $+127$ — NOT -127 to $+127$. The negative range extends one further than the positive because zero occupies one of the positive slots. Students frequently get the boundary wrong; learn it explicitly.

Logical Operations

The ALU performs logical (Boolean) operations on binary values. These operations work **bit by bit** across corresponding bit positions.

Truth Tables

A	B	AND	OR	XOR
0	0	0	0	0
0	1	0	1	1
1	0	0	1	1
1	1	1	1	0

A	NOT A
0	1
1	0

MEMORISE THIS

Quick rules:

- AND: both must be 1 to get 1
- OR: at least one must be 1 to get 1
- XOR: exactly one must be 1 to get 1 (differs)
- NOT: flips the bit

Bit Manipulation with Logical Operations

Logical operations are used to manipulate specific bits within a byte:

- **AND with a mask** — used to **clear** (force to 0) specific bits. Apply mask with 0 where you want to clear.
- **OR with a mask** — used to **set** (force to 1) specific bits. Apply mask with 1 where you want to set.
- **XOR with a mask** — used to **toggle** specific bits. Apply mask with 1 where you want to flip.

WORKED EXAMPLE

Clear bits 3 and 4 of `10111110` using AND:

Mask: `11100111` (0s in positions 3 and 4)

```
10111110 AND 11100111 = 10100110
```

Machine Instruction Concepts

A **machine instruction** is a binary-encoded command that the CPU can execute directly. Students do not need to learn a full assembly language, but should understand the concept.

Every machine instruction consists of:

- **Opcode** — specifies the operation (e.g., LOAD, STORE, ADD, JUMP)
- **Operand** — specifies the data or memory address involved

Common instruction types:

- **Data transfer:** LOAD (memory → register), STORE (register → memory)
- **Arithmetic:** ADD, SUBTRACT
- **Logical:** AND, OR, NOT
- **Branch/Jump:** unconditional jump (always jump to address), conditional jump (jump if result = 0, etc.)
- **Halt:** stop execution

Programs stored in memory as binary machine code are fetched and executed by the CPU one instruction at a time via the fetch-decode-execute cycle.

Practice Questions

▶ Q1 — State the function of the MAR and MDR during a memory read operation. [4 marks]

▶ Q2 — Convert the binary number `01101001` to decimal and to hexadecimal. [4 marks]

▶ Q3 — Find the 8-bit two's complement representation of −23. Show all working. [3 marks]

▶ Q4 — State the range of values that can be stored in a 4-bit two's complement system. [2 marks]

▶ Q5 — Trace the fetch-decode-execute cycle for the instruction STORE 300, where the accumulator currently holds the value 42 and the PC contains 105. [4 marks]

▶ Q6 — A byte has the value `10101100`. Apply an OR operation with mask `00001111`. State the result and explain what this operation achieves. [3 marks]