Processor Fundamentals

A-Level Computer Science

Von Neumann architecture

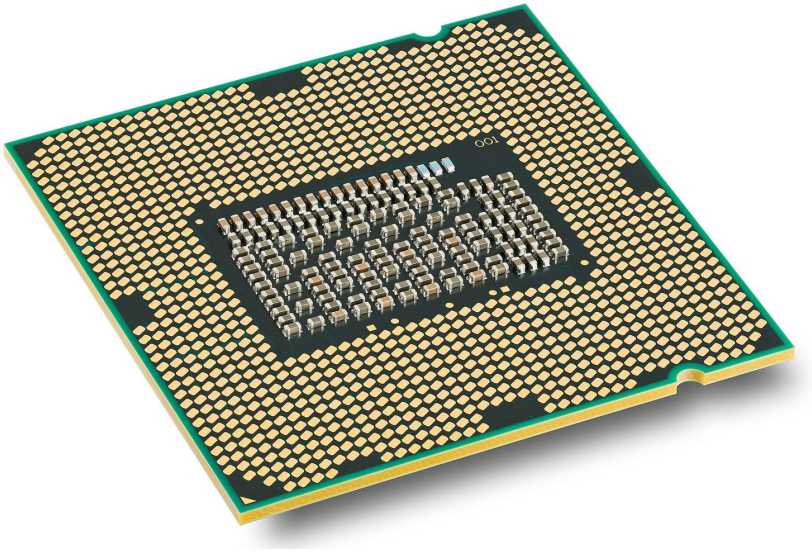
The **Von Neumann architecture** · underlies almost every general-purpose computer:

- a **single memory** – the **Immediate Access Store** (IAS) – holds both **program instructions** and **data** (the **stored program** concept).
- a **processor** (CPU) fetches instructions from memory and runs them one at a time.
- instructions run **in order** unless a branch changes the flow.

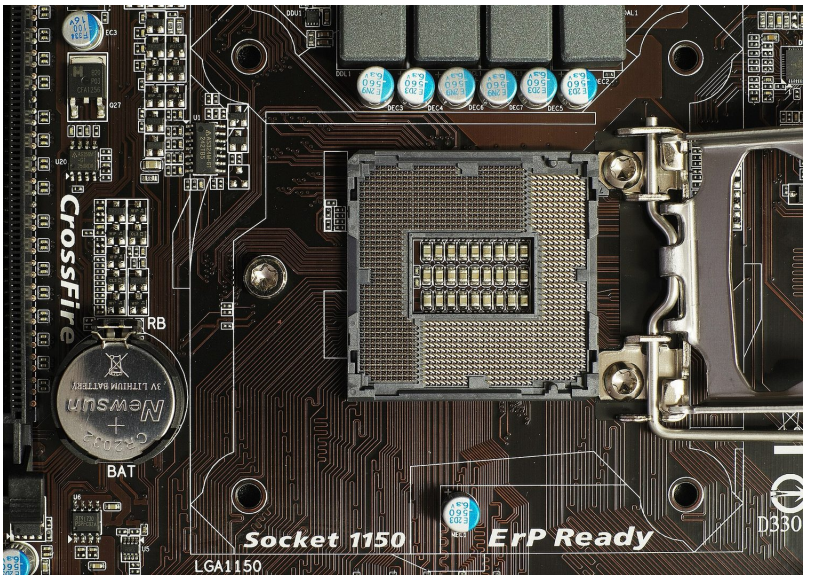
The stored-program idea is what makes a computer flexible: change the program and you change what it does, with no rewiring.

The CPU's main parts

All of these parts sit inside one small chip. The diagram later in this section shows how they connect; the photo below shows the real thing.



A modern CPU: the whole processor is one small chip (here seen from below, showing the contacts)



The matching CPU socket on the motherboard: the chip's contacts press onto these pins

■ Arithmetic and Logic Unit (ALU)

The **ALU** does the arithmetic (add, subtract, …) and logic (AND, OR, comparisons). It takes operands from **registers** and puts results back in a register.

■ Control Unit (CU)

The **control unit** **decodes** each instruction and sends the **control signals** to carry it out – opening data paths, telling the ALU what to do, and controlling memory reads and writes.

■ System clock

The clock sends a steady stream of pulses that keep the CPU in step. Each instruction takes a fixed number of cycles, and the **clock speed** (e.g. 3.8 GHz) is one factor in performance.

"Explain how the CU and the system clock work together": the clock emits pulses at a fixed frequency; the control unit uses each pulse to move the fetch-execute cycle on by one step, sending its control signals in time with the pulses, so every part of the processor changes state together. A faster clock means more steps per second, up to the point where the circuits cannot settle between pulses.

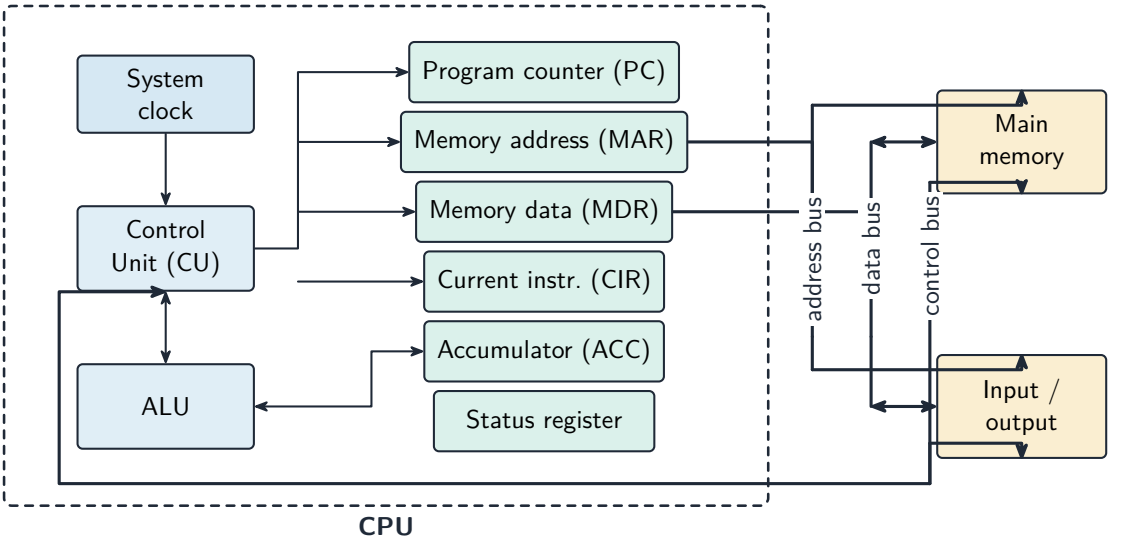
Registers

Registers are tiny, very fast stores inside the CPU. The **special purpose registers** each have a fixed job in the cycle:

- **Program Counter** (PC) – the address of the **next** instruction.
- **Memory Address Register** (MAR) – the address being read or written.
- **Memory Data Register** (MDR) – the data going to or from memory.
- **Current Instruction Register** (CIR) – the instruction being decoded.
- **Accumulator** (ACC) – the value the ALU is working on.
- **Status Register** – holds **flags** (carry, zero, negative, overflow) used by branches. Each flag is one bit, set or cleared by the ALU after an operation: the zero flag after a comparison that matched, the carry flag when an addition overflowed the register, the negative flag when a result is negative. A conditional jump reads the flags to decide whether to branch, and an overflow flag can raise an interrupt.
- **Index Register** – an offset added to an address in indexed addressing; incrementing it steps through an array one element at a time.

The "complete the table describing the role of each register" question wants one precise sentence per register in these terms: the PC holds the address of the next instruction to be fetched; the MAR holds the address of the location being read from or written to; the MDR holds the data or instruction just read from, or about to be written to, that location; the CIR holds the instruction currently being decoded and executed; the ACC holds the result of the last arithmetic or logic operation.

General-purpose registers are used by the programmer for temporary values during a calculation. Movements of data between registers and memory are written in **register transfer** notation – e.g. `MAR ← [PC]` ("copy the contents of PC into MAR").



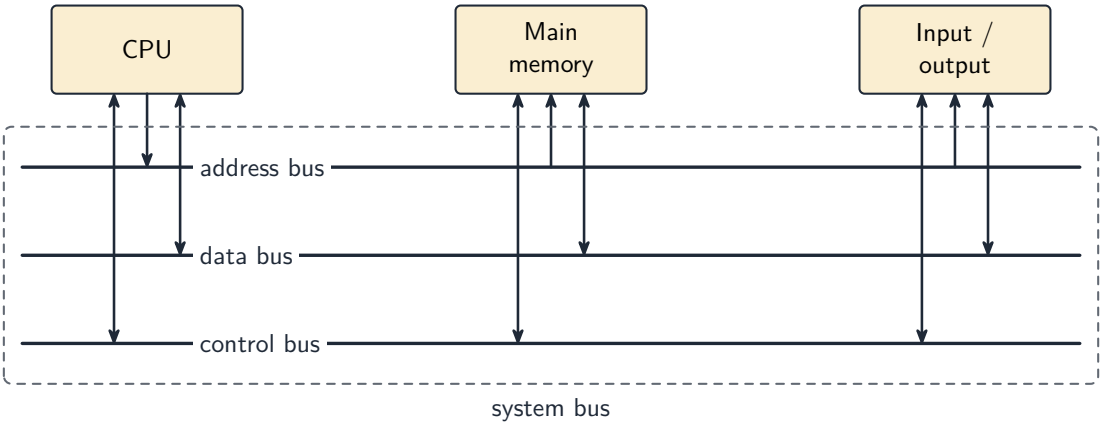
The Von Neumann CPU: registers, control unit and ALU linked by buses

Buses

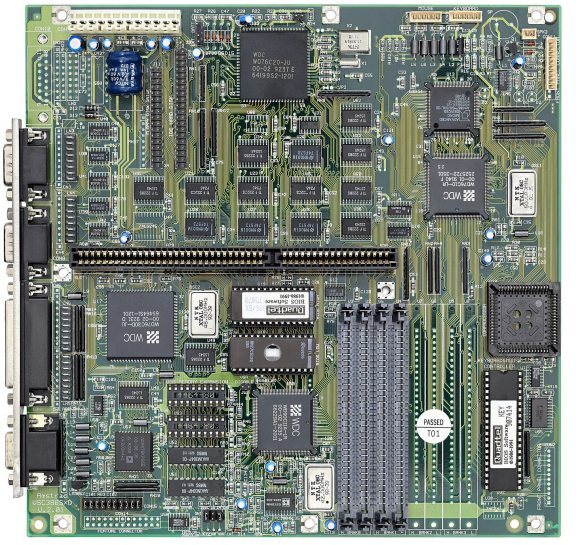
Three internal **buses** (sets of parallel wires) connect the parts:

- **address bus** – carries the memory address. **One-way** (CPU → memory).
- **data bus** – carries the data. **Two-way**.
- **control bus** – carries control signals (read, write, interrupt). **Two-way**.

An n-bit address bus can reach 2^n memory locations. The data-bus width sets how many bits move per access (often the word size).



The three system buses connecting the CPU, memory and input/output



A motherboard: the CPU, memory and I/O all sit on one set of buses – the printed tracks running between them

What affects performance

- **clock speed** – more cycles per second.
- **number of cores** – a multi-core CPU runs several threads at once.
- **word size** – a 64-bit CPU handles 64-bit chunks per cycle and can address far more memory than a 32-bit one.
- **amount of RAM** – more RAM holds more of the working set; too little forces the OS to **page** to disk.
- **cache memory** size – more cache cuts average memory access time.
- **secondary storage** type – an SSD loads programs far faster than an HDD.
- **bus width and speed** – wider/faster buses move data more quickly.

Match the specs to the workload: a quad-core beats a dual-core on parallel work, but higher per-core speed wins on single-threaded work.

Each factor is a two-mark answer with a reason attached:

- **More cores**: each core can fetch and execute its own instruction at the same time, so several programs, or the threads of one program, run in parallel. But a program must be written to use more than one core, so doubling the cores does not double the speed.
- **Higher clock speed**: more fetch-execute cycles per second, so more instructions per second; the limit is the heat produced.

- **Wider bus**: a wider data bus moves more bits in each transfer, so fewer transfers are needed for the same data; a wider address bus can address more memory locations.
- **Cache memory**: a small, fast memory inside or next to the processor that keeps the instructions and data used most recently or most often. Reading them from cache is much faster than from RAM, so the processor spends less time waiting.

"Explain why the new computer performs better" is answered by comparing the two specifications line by line: a higher clock speed executes more instructions per second, more cores run more tasks at once, more cache means fewer slow accesses to RAM, and more RAM means fewer transfers to disk.

Ports

A **port** is a physical socket for connecting a **peripheral** :

- **USB** (Universal Serial Bus) – general-purpose (keyboards, drives, phones).
- **HDMI** (High Definition Multimedia Interface) – digital video and audio to a screen.
- **VGA** (Video Graphics Array) – older analogue video output to a monitor.
- **Ethernet (RJ-45)** – wired LAN. Audio jacks – headphones/microphone.

Different ports use different signals, so an HDMI cable will not fit a USB socket. USB-C is unusual in carrying video, data and power.

"Explain how the computer connects to the monitor through HDMI": the HDMI port sends the video and the audio as one digital signal down a single cable, so no conversion to analogue is needed and the picture is not degraded; the cable carries high-definition resolutions and the monitor's own port decodes the signal. A USB device is **plug-and-play**: when it is connected the computer detects it, identifies it, loads or installs the driver it needs, and can supply it with power, all without a restart.

Fetch-Execute cycle

The CPU repeats the **fetch-execute cycle** - , one run per machine instruction.

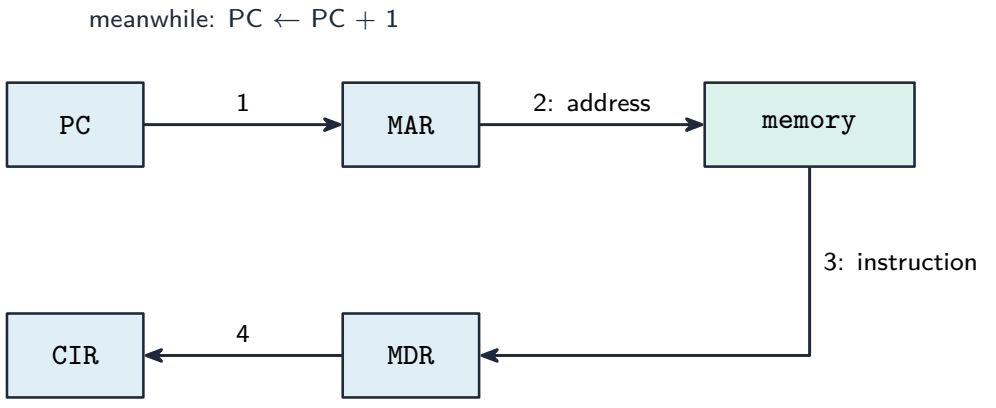
■ Fetch

1. the PC's address is copied to the MAR.
2. the PC is **incremented** to point to the next instruction.
3. a **read** signal goes over the control bus.
4. memory puts the instruction on the data bus.
5. it is copied into the MDR, then into the CIR.

The exam asks for these steps in **register transfer notation** , where `[X]` means the contents of register X and `[[MAR]]` means the contents of the memory location whose address is in the MAR:

```
MAR ← [PC]          the address of the next instruction goes to the MAR
PC  ← [PC] + 1      the PC now points to the following instruction
MDR ← [[MAR]]       the instruction at that address is read into the MDR
CIR ← [MDR]         the instruction is copied into the CIR for decoding
```

The order matters: the PC is incremented straight after its address has been copied, so that a jump executed later can still overwrite it. During execution the same notation describes each instruction; for `LDD 200`, for example, `MAR ← 200`, `MDR ← [[MAR]]`, `ACC ← [MDR]`.



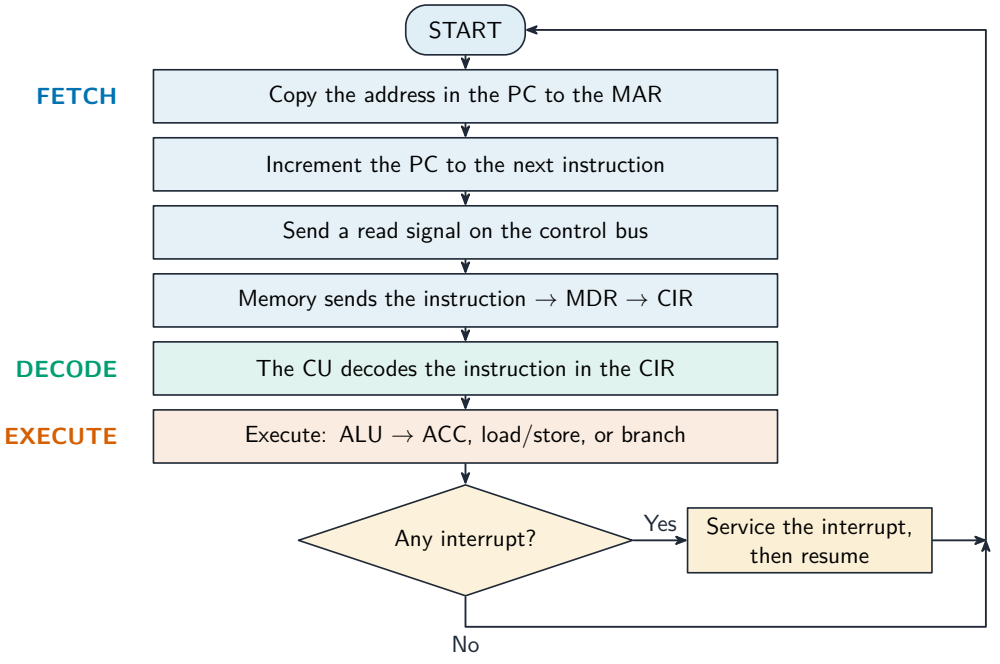
The register transfers in a fetch: PC → MAR → memory → MDR → CIR, with the PC incremented

Decode

The CU decodes the instruction in the CIR – what operation, and which operands or addresses.

Execute

The CU carries it out: arithmetic/logic goes to the ALU (result to the ACC); a load/store moves data between memory and a register; a branch changes the PC. Then the cycle repeats.



The fetch-execute cycle, with a check for interrupts each time

Interrupts

An **interrupt** is a signal that **pauses** the normal cycle so the CPU can handle an urgent event (a key press, a packet arriving, a hardware fault, division by zero, the OS timer).

Handling one:

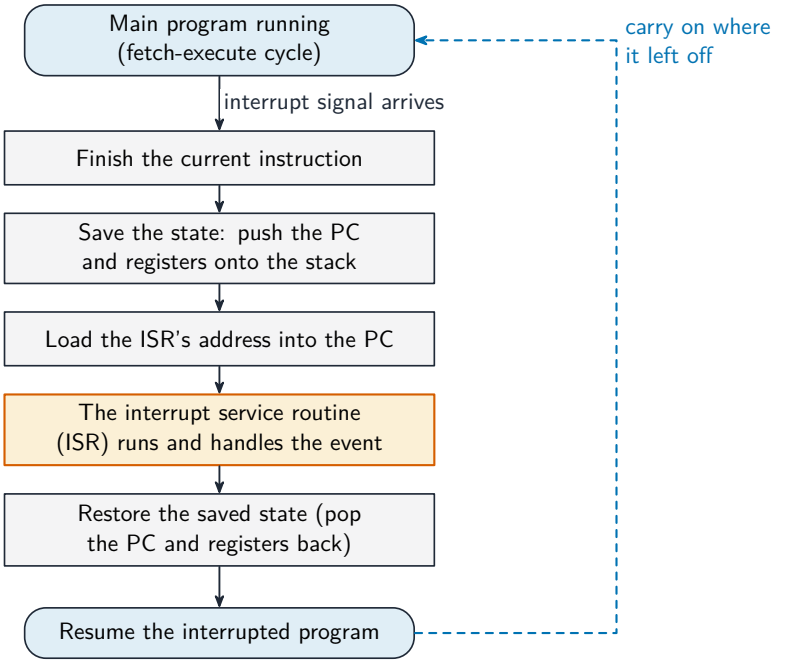
1. finish the current instruction.
2. **save the state** (PC and registers).
3. load the address of the **interrupt service routine** (ISR) into the PC and run it.
4. the ISR handles the event.
5. **restore** the saved state and carry on.

Interrupts let the system respond promptly without the CPU constantly checking devices, and are how the OS multitasks.

"Explain how an interrupt from an input device is detected and handled in the F-E cycle" is a four-mark answer with these points: the device sends an interrupt signal that sets the interrupt flag in the **interrupt register** ; the processor checks that register at the **end of every fetch-execute cycle**, after the current instruction has finished executing; if a flag is set and the interrupt has a higher priority than the current task, the contents of the PC and the other registers are saved onto the **stack** ; the address of the interrupt service routine is loaded into the PC and the routine

runs; when it finishes, the saved values are restored from the stack and the interrupted program continues from where it stopped.

Causes worth naming: a hardware interrupt from a device (a key pressed, a printer buffer empty, a network packet arriving), a software interrupt from a fault (division by zero, an illegal instruction, arithmetic overflow), a timer interrupt from the operating system marking the end of a time slice, and a power failure warning.



How an interrupt fits into the fetch-execute cycle

Assembly language and machine code

The CPU actually runs **machine code** – bit patterns, specific to one architecture. **Assembly language** is a readable form, with one instruction per machine instruction, written using **mnemonics** like `LDD`, `ADD`, `JMP`. An **assembler** translates it to machine code.

An assembler turns mnemonics into machine-code bit patterns

Two-pass assembler

A two-pass assembler reads the source twice:

- **pass 1** builds a **symbol table**: each time a **label** (like `LOOP:`) appears, record its address; no code yet.
- **pass 2** generates code: translate each instruction, and when one refers to a label (like `JMP LOOP`), look up its address in the symbol table.

Two passes handle **forward references** (a jump to a label defined later).

Worked example. Apply the two-pass process to this program, whose first instruction is stored at address 100.

```
        LDD  COUNT
LOOP:   DEC  ACC
        CMP  #0
        JPN  LOOP
        END
COUNT:  5
```

Pass 1 reads each line, counts the address it will occupy, and records every label in the symbol table: `LOOP` = 101 (the `DEC` line) and `COUNT` = 105 (the data line). No code is produced. Pass 2 reads the program again and translates each line into machine code, replacing each mnemonic by its **opcode** and each symbolic address by the number from the symbol table: `LDD COUNT` becomes the opcode for `LDD` with **operand** 105, and `JPN LOOP` becomes the opcode for `JPN` with operand 101. The jump back to `LOOP` could have been resolved in one pass, but a jump *forward* to a label not yet seen could not, which is why the assembler makes two.

Example instruction set

Cambridge uses a small generic set, printed in the paper's reference table, with one general-purpose register, the accumulator (ACC), and an index register (IX). An operand written `#n` is a denary number, `Bn` a binary number and `&n` a hexadecimal number; `<address>` is a location number or a label.

Group	Instruction	What it does
Data movement	`LDM #n`	load the number n into ACC (immediate)
	`LDD <address>`	load the contents of the address into ACC (direct)
	`LDI <address>`	the address holds another address; load the contents of that one into ACC (indirect)

Group	Instruction	What it does
	`LDX <address>`	add IX to the address and load the contents of the result into ACC (indexed)
	`LDR #n`	load the number n into IX
	`MOV <register>`	copy ACC into the named register (IX)
	`STO <address>`	store the contents of ACC at the address
Input and output	`IN`	read a key press and put its ASCII code in ACC
	`OUT`	output the character whose ASCII code is in ACC
Arithmetic	`ADD <address>` / `ADD #n`	add the contents of the address, or the number, to ACC
	`SUB <address>` / `SUB #n`	subtract from ACC
	`INC <register>` / `DEC <register>`	add 1 to, or subtract 1 from, ACC or IX
Compare	`CMP <address>` / `CMP #n`	compare ACC with the contents of the address, or with n, and set the flag
	`CMI <address>`	compare ACC with the contents of the address held at the address (indirect)
Jump	`JMP <address>`	jump to the address unconditionally
	`JPE <address>` / `JPN <address>`	jump if the last compare was equal / not equal
Bit manipulation	`AND`, `OR`, `XOR` with `#n`, `Bn`, `&n` or `<address>`	bitwise operation on ACC
	`LSL #n` / `LSR #n`	shift ACC logically n places left or right
	`END`	end the program

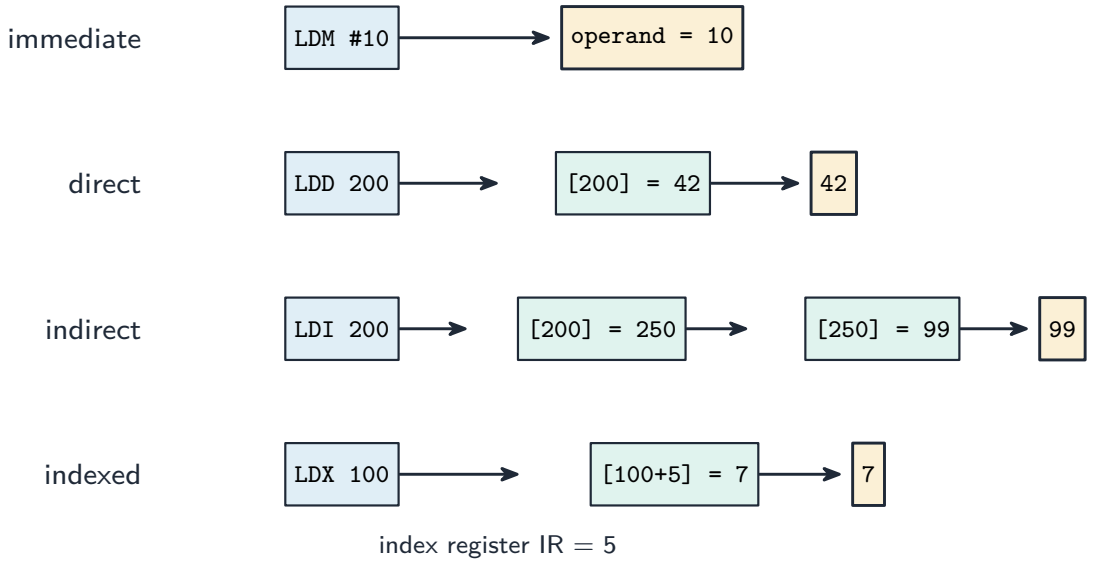
The "assembly language instructions are grouped" question wants the group names, and an instruction from each: data movement, input and output, arithmetic, unconditional and conditional jumps, compare, and bit manipulation.

Addressing modes

The **addressing mode** (the **modes of addressing**) says how the CPU finds the operand:

- **immediate addressing** – the operand is the value in the instruction. `LDM #10` loads 10.
- **direct addressing** – the instruction holds an address; the operand is the value there. `LDD 200`.
- **indirect addressing** – the instruction holds an address that holds **another** address, which is the data. `LDI 200`.
- **indexed addressing** – effective address is `address + index register`; used for arrays. `LDX 100` with IR = 5 reads address 105.

(**Relative addressing** gives the address as an offset from the PC – used for jumps.)



How each addressing mode reaches its operand – immediate, direct, indirect and indexed

Worked example. Memory holds: location `200` = `250`, location `250` = `99`, location `105` = `7`. The index register holds `5`. What is in the accumulator after each of `LDM #200`, `LDD 200`, `LDI 200` and `LDX 100`? Follow how far each mode has to look. `LDM #200` is **immediate** - the operand *is* the number written in the instruction, so the accumulator holds **200**. `LDD 200` is **direct** - go to location 200 and take what is there: **250**. `LDI 200` is **indirect** - location 200 holds 250, which is *another address*, so go on to location 250: **99**. `LDX 100` is **indexed** - add the index register to the address, 100 + 5 = 105, and read location 105: **7**. Count the hops to keep them apart: immediate 0, direct 1, indirect 2, indexed 1 (once the index has been added).

Tracing an assembly program

To **trace** it: make a table with columns for the PC, ACC, index register, each variable and any flags. Step through the instructions, updating the table after each; follow branches when they change the PC; stop at `END`. A common pattern is a loop over an array using indexed addressing.

Worked example. Trace this program. Address 200 holds 5 and address 201 holds 0.

```
100   LDD  200
101   CMP  #0
102   JPE  108
103   OUT
104   DEC  ACC
105   STO  200
106   LDD  201
107   JMP  100
108   END
```

Write one row for each instruction executed, filling in only the columns that change:

Instruction	ACC	200	201	Output
start		5	0	
`LDD 200`	5			
`CMP #0`				
`JPE 108`				not taken
`OUT`				character with code 5
`DEC ACC`	4			
`STO 200`		4		
`LDD 201`	0			
`JMP 100`				
`LDD 200`	4			

and so on, until `LDD 200` loads 0, the compare sets the equal flag, `JPE 108` is taken and the program ends. Three things the examiner checks: a `CMP` changes no register, only a flag; a jump not taken still counts as executed; and `OUT` outputs a character, so it goes in the output column, not the ACC column. "State the effect of changing `LDD 10` to `LDM #10`": the ACC would hold the number 10 instead of the contents of address 10.

Binary shifts

A **logical shift** moves all the bits left or right by some places, filling new positions with 0.

- **left shift by 1** (`LSL #1`) – bits move left, a 0 enters on the right; for an unsigned number this is $\times$ **2**.
- **right shift by 1** (`LSR #1`) – bits move right, a 0 enters on the left; for an unsigned number this is **integer $\div$ 2**.

Shifting by n places multiplies or divides by 2^n. Example: `00001011` (11) `LSL #1` $\rightarrow$ `00010110` (22).

Bits shifted off the end are lost, so the multiplication is only correct while they were zeros. `LSL #2` on the two's-complement integer `11001010` gives `00101000`: the two 1s that fell off the left are gone, the sign bit has changed, and the result is no longer four times the original.

An **arithmetic right shift** keeps the sign bit so a negative signed number stays negative. A **cyclic shift** (rotate) feeds the bit that drops off one end back in at the other end, so no bits are lost.

"Show the result of an arithmetic right shift of 3 places on `10011110`": copy the sign bit into each vacated place, `11110011`. The same shift on `01011100` gives `00001011`. A cyclic left shift of 1 on `10000110` gives `00001101`: the leading 1 reappears on the right.

LSL #1	00001011	$\longrightarrow$	00010110	$\times 2$	(0 in on the right)
LSR #1	00001011	$\longrightarrow$	00000101	$\div 2$	(0 in on the left)
ASR #1	10110100	$\longrightarrow$	11011010	sign bit kept (stays negative)	

Logical left ($\times 2$), logical right ($\div 2$) and arithmetic right (keeps the sign bit)

The difference between the two right shifts is a single bit. Take `11110000`, which is 240 read as unsigned and -16 read as signed. `LSR #1` brings in a 0 and gives `01111000` = 120, which is the correct half of 240. `ASR #1` copies the sign bit instead and gives

`11111000` $= -8$, which is the correct half of -16. Neither is wrong – each halves the value under one reading.

`start`	1	1	1	1	0	0	0	0	$= 240$ unsigned, -16 signed	
`LSR #1`	0	1	1	1	1	0	0	0	$= 120$	halves the **unsigned** value
`ASR #1`	1	1	1	1	1	0	0	0	$= -8$	halves the **signed** value

The two results differ in **one bit** — the one that entered on the left. **LSR** always brings in a 0; **ASR** copies the sign bit, so a negative number stays negative and $-16 \div 2$ comes out right.

Logical and arithmetic right shift on the same byte: only the bit that enters on the left differs

Bit manipulation for monitoring/control

Embedded devices often use one **bit** of a register per signal (e.g. bit n = LED n). Using a **mask** – **bit masking** – you can:

- **set** bit n: `R = R OR` a mask with bit n set.
- **clear** bit n: `R = R AND` a mask with bit n clear and the rest set.
- **toggle** bit n: `R = R XOR` a mask with bit n set.
- **test** bit n: `R AND` the mask, then check if the result is non-zero.

set bit 2	**clear bit 6**	**toggle bit 3**
`01001000`	`01001000`	`01001000`
`OR 00000100`	`AND 10111111`	`XOR 00001000`
`= 01001100`	`= 00001000`	`= 01000000`

Set a bit with OR, clear it with AND, toggle it with XOR – each using a mask

Bit manipulation is fast, uses little memory, and lets one byte hold up to 8 on/off states.

In the exam's instruction set these are `AND`, `OR` and `XOR` with a mask written as a denary, binary or hexadecimal operand. With the ACC holding `10101100`:

Instruction	Mask	Result in ACC	Effect
`AND B00001111`	`00001111`	`00001100`	keeps only the low four bits (clears the others)
`OR #1`	`00000001`	`10101101`	sets the least significant bit, leaving the rest unchanged
`XOR &FF`	`11111111`	`01010011`	inverts every bit
`AND B00001000` then `CMP #0`	`00001000`	`00001000`	tests bit 3: the compare is not equal, so bit 3 was set
`LSL #2`		`10110000`	shifts left two places, losing the top two bits
`LSR #3`		`00010101`	shifts right three places, zeros entering on the left

"Write the instruction that sets the least significant bit to 1 and leaves the others unchanged": `OR #1`, or `OR B00000001`. To clear a bit use `AND` with a mask that has a 0 in that place and 1s elsewhere; to test a bit, `AND` with a mask that has a 1 only in that place, then compare the result with zero. In a monitoring device, one bit of a register per sensor lets a single `AND` check whether a particular sensor is on, and one `OR` switches an actuator's control bit on without disturbing the others.

Definitions the examiner accepts

A definition question is marked against fixed wording. Learn these exactly, and give one answer only.

Term	Definition
stored program concept	the program instructions and the data are both held in main memory, and instructions are fetched and executed one at a time
register	a small, very fast storage location inside the processor with a specific purpose
Program Counter	the register holding the address of the next instruction to be fetched
Memory Address Register	the register holding the address of the memory location being read from or written to
Memory Data Register	the register holding the data or instruction just read from, or about to be written to, memory

Term	Definition
Current Instruction Register	the register holding the instruction currently being decoded and executed
Accumulator	the general-purpose register holding the result of the last arithmetic or logic operation
cache memory	small, fast memory close to the processor holding frequently used instructions and data
interrupt	a signal from a device or program that causes the processor to pause the current task and run an interrupt service routine
assembly language	a low-level language in which each mnemonic instruction corresponds to one machine-code instruction
immediate addressing	the operand is the value written in the instruction
direct addressing	the operand is the contents of the address written in the instruction
indirect addressing	the address in the instruction holds the address of the operand
indexed addressing	the operand's address is the address in the instruction plus the contents of the index register
relative addressing	the operand's address is given as an offset from the address of the current instruction
logical shift	every bit moves the given number of places and zeros fill the vacated places

Exam tips

- Learn the **fetch-execute cycle** in register-transfer terms (PC, MAR, MDR, CIR, ACC) and what increments the PC.
- Name each register's job; the **address bus is one-way**, the data bus is two-way.
- Distinguish the **addressing modes** (immediate, direct, indirect, indexed) – a frequent question.
- Explain how clock speed, number of cores, cache size and word length affect **performance**.
- For a **binary shift**, state whether it is logical or arithmetic; a left shift multiplies by 2, a right shift divides by 2.

Common mistakes

- Saying the PC holds the current instruction, or the MDR holds an address. The PC holds the address of the *next* instruction; the MDR holds data or an instruction, never an address.
- Leaving the increment of the PC out of the fetch, or putting it after the execute. It happens as soon as the address has been copied to the MAR.
- Reading `LDD 10` as "load 10". `LDD 10` loads the contents of address 10; `LDM #10` loads the number 10.
- Putting a value in the ACC column for `CMP` or `OUT`. A compare sets a flag only; an output goes to the output column.
- Saying an interrupt is handled "immediately". The processor finishes the current instruction and checks for interrupts at the end of the cycle.
- Using a logical right shift on a negative two's-complement number. Only an arithmetic shift keeps the sign bit.