

Cambridge AS & A Level · Computer Science 9618 · Notes

Processor fundamentals AS

Read online at pastopic.com/cambridge/computer-science-9618/notes/processor-fundamentals

Last checked 30 September 2026

Read first: [Information representation](#)

© 2026 Pastopic. Free for personal study. Please share the link, not the file.

What you need to know

By the end of this topic you should be able to:

1. **Describe the Von Neumann model** of a computer and the **stored program concept**.
2. **Explain what registers are for**, the difference between **general purpose** and **special purpose** registers, and the role of each special purpose register: Program Counter (PC), Memory Data Register (MDR), Memory Address Register (MAR), Accumulator (ACC), Index Register (IX), Current Instruction Register (CIR) and Status Register (SR).
3. **Explain the roles** of the Arithmetic and Logic Unit (ALU), the Control Unit (CU), the system clock and the Immediate Access Store (IAS).
4. **Explain how data moves** between the parts of a computer on the **address bus**, **data bus** and **control bus**.
5. **Explain how performance depends on** the processor type and number of cores, the bus width, the clock speed and the cache memory.
6. **Explain how ports connect peripheral devices**: USB, HDMI and VGA.
7. **Describe the stages of the fetch-execute (F-E) cycle**, and write them in **register transfer notation**.
8. **Explain interrupts**: what causes them, what they are used for, the Interrupt Service Routine (ISR), when in the F-E cycle they are detected and how they are handled.
9. **Explain how assembly language relates to machine code**.
10. **Describe the two-pass assembler**, and apply it to a short assembly program.
11. **Trace a simple assembly language program**.
12. **Know the instruction groups**: data movement, input and output, arithmetic, unconditional and conditional jumps, and compare.
13. **Use the addressing modes** immediate, direct, indirect, indexed and relative.
14. **Carry out binary shifts**: logical, arithmetic and cyclic, left and right.
15. **Use bit manipulation to monitor or control a device**: test and set bits with a mask, using AND, OR and XOR.

The exam prints the instructions you need inside each question. Binary, hexadecimal and two's complement come from Information representation.

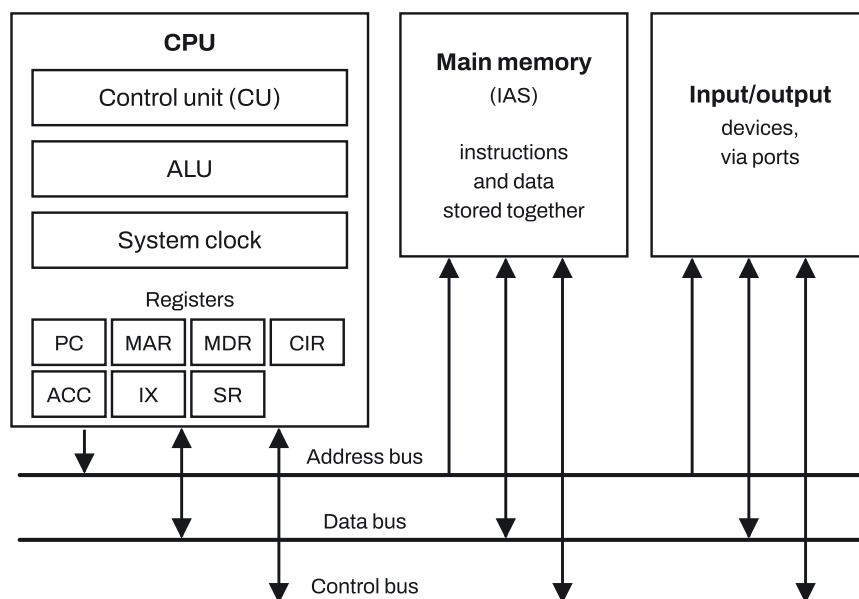
Understand it

The Von Neumann model

Almost every computer follows the design described by John von Neumann in the 1940s:

- **one processor (CPU)** that carries out instructions one after another;
- **one main memory** that holds **both the program's instructions and the data** it works on, in binary, side by side;
- **buses** that carry addresses, data and control signals between the CPU, the memory and the input/output devices.

The **stored program concept** is the key idea: a program's instructions are **stored in main memory**, just like data, and the processor **fetches them one at a time**, decodes them and executes them in order. To run a different program, load different instructions into memory.



Registers

A **register** is a tiny, very fast storage location **inside the CPU**, holding one value the processor is using right now.

Register	What it holds
Program Counter (PC)	The address of the next instruction to be fetched
Memory Address Register (MAR)	The address of the memory location about to be read from or written to
Memory Data Register (MDR)	The data or instruction just read from that location, or about to be written to it

Register	What it holds
Current Instruction Register (CIR)	The instruction that is being decoded and executed now
Accumulator (ACC)	The value being worked on: the results of arithmetic and logic operations
Index Register (IX)	A number that is added to an operand to form an address (indexed addressing)
Status Register (SR)	Individual bits called flags , set or cleared by results (for example carry, negative, zero, overflow) and checked to decide whether to jump; also interrupt flags

General purpose registers hold whatever data the programmer chooses during calculations. **Special purpose registers** have one fixed job and hold the **state of the program's execution**. (The syllabus lists the ACC with the special purpose registers, but the assembly questions call it "the only general purpose register"; use the question's term.)

ALU, CU, system clock and IAS

- **Arithmetic and Logic Unit (ALU)**: does the arithmetic (add, subtract) and the logic (AND, OR, XOR, comparisons, shifts). Results go to the ACC, and the ALU sets flags in the status register.
- **Control Unit (CU)**: fetches instructions, **decodes** them and manages their execution, sending **control signals** along the **control bus** (for example "read memory now") to coordinate the other components.
- **System clock**: produces a steady stream of **timing signals** (pulses). The CU sends its signals **on each clock pulse**, so all the components stay **synchronised** and each step happens in the right order. That is its purpose; its speed is a separate idea (below).
- **Immediate Access Store (IAS)**: the memory the CPU can access **directly** (main memory, RAM), holding the programs and data in current use. Anything on a disk must be copied into it first.

Buses

A **bus** is a set of parallel wires that carries signals between components.

Bus	Carries	Direction
Address bus	The address of the memory location (or device) to use, from the MAR	One way : from the CPU out
Data bus	Data and instructions, to and from the MDR	Both ways
Control bus	Control and timing signals: memory read, memory write, clock pulses, interrupt requests, bus request and bus grant, reset	Both ways

For example, to read from memory the MAR puts the address on the **address bus**, the CU puts a "read" signal on the **control bus**, and memory puts the contents on the **data bus**, which arrive in the MDR.

What makes a computer faster

- **Clock speed.** Each step of the F-E cycle takes a set number of clock pulses, so a faster clock means **more F-E cycles per second** and more instructions processed per second. A 3.2 GHz clock ticks 3,200,000,000 times a second. Faster clocks make more heat, and the processor may still wait for slower memory.
- **Number of cores.** A **core** is a complete processing unit (its own CU, ALU and registers). With more cores, **different instructions can be processed at the same time** (in parallel). Two cores are rarely twice as fast, because:
 - the software must be written to split its work between cores, otherwise extra cores sit idle;
 - the cores must communicate and share memory and buses, which adds delay (latency);
 - some tasks must be done in order, so they cannot be split;
 - cores can end up waiting for each other (in the worst case a deadlock);
 - more cores produce more heat.
- **Processor type.** Different designs do different amounts of work per clock cycle, so two processors at the same clock speed can perform differently.
- **Bus width.** A wider **data bus** moves **more bits in each transfer**, so fewer transfers are needed. A wider **address bus** can carry larger addresses, so **more memory locations can be addressed**: n lines address 2^n locations. A 16-bit address bus reaches $2^{16} = 65,536$ locations; a 32-bit one reaches $2^{32} = 4,294,967,296$.
- **Cache memory.** A small amount of very fast memory (SRAM) on or next to the processor, holding **copies of the most frequently used instructions and data**. The processor looks there first, and cache is much faster than RAM. **More cache** holds more of these items, so the processor goes to slower RAM less often and waits less.

Ports

A **port** is the socket a peripheral plugs into, with rules for how data passes through it.

- **USB (Universal Serial Bus):** the standard port for most peripherals (keyboards, printers, flash drives). It sends data **serially**, one bit after another, and can supply power. When a device is plugged in, the computer **detects the voltage change**, the device sends its **identification**, and the operating system **loads the right device driver**, so it works straight away.
- **HDMI (High Definition Multimedia Interface):** for monitors, TVs and projectors. It is **digital**, carries **audio and video in one cable**, and has a **high bandwidth**, enough for a high-resolution screen, sent as a stream of **uncompressed** digital signals with little interference.
- **VGA (Video Graphics Array):** an older, **analogue, video-only** port (sound needs another cable). The digital image is converted to analogue and back, losing quality at high resolutions.

The fetch-execute cycle

The CPU repeats three stages for every instruction: **fetch, decode, execute**.

Register transfer notation writes each step as "destination $\leftarrow$ source", where square brackets mean "the contents of":

- $[PC]$ is the value in the PC (an address);
- $[[MAR]]$ is the contents of the **memory location whose address is in the MAR**.

The fetch stage is:

1. $MAR \leftarrow [PC]$: the address of the next instruction is copied from the PC into the MAR (and put on the address bus).
2. $PC \leftarrow [PC] + 1$: the PC is incremented, so it already points to the following instruction. (Steps 1 and 2 can happen at the same time.)
3. $MDR \leftarrow [[MAR]]$: the instruction at that address is copied along the data bus into the MDR.
4. $CIR \leftarrow [MDR]$: the instruction is copied into the CIR.

Then the CU **decodes** the instruction in the CIR (opcode and operand), and it is **executed**. A jump simply puts a new address in the PC, so the next fetch comes from there. At the end of each cycle, before the next fetch, the processor **checks for interrupts**.

Interrupts

An **interrupt** is a signal to the processor that a device or program **needs attention**, so the processor doesn't have to keep checking every device.

Causes:

- **Hardware interrupts:** a key pressed or mouse moved, a printer out of paper or jammed, a buffer that is full or empty and needs data, a hardware fault, a power failure.
- **Software interrupts:** a run-time error such as **division by zero**, a **stack overflow**, a program trying to use memory it is not allowed to, or a program asking the operating system for a service.
- **Timer interrupts:** the end of a process's time slice, so the operating system can switch to another process.

Uses: reading keyboard input as it arrives, keeping a printer supplied with data, handling errors, and sharing the processor between programs.

How an interrupt is handled:

1. When the device needs attention, a flag is set in the **interrupt register**. The current instruction is always finished first.
2. **At the end of each F-E cycle** (before the next fetch) the processor checks the interrupt register for waiting interrupts.
3. If there is one, its **priority** is compared with the priority of the current task. If it is **lower**, it waits and the program carries on.

4. If it is **higher**, the processor **saves the contents of the registers** (including the PC) **on the stack**, so the current program can be resumed later.
5. The address of the **interrupt service routine (ISR)** for that type of interrupt is loaded into the PC. There is a **different ISR for each type** of interrupt (keyboard, printer, division by zero and so on), and the ISR is the code that deals with it.
6. The ISR runs. While it runs, an interrupt of even higher priority can interrupt it in the same way.
7. When the ISR finishes, the processor checks for any other waiting interrupts of higher priority than the saved program, and services them first.
8. Finally the saved register contents are **restored from the stack** and the F-E cycle continues with the interrupted program exactly where it stopped.

Assembly language and machine code

Machine code is the binary the processor actually executes. Each instruction is an **opcode** (what to do) and usually an **operand** (what to do it to: a number or an address).

Assembly language writes the same instructions readably: a **mnemonic** for each opcode (LDD, ADD, JMP) and **labels** (symbolic addresses) for addresses. **Each assembly instruction becomes one machine code instruction** (one-to-one). Both are **low-level** and **specific to one type of processor**. An **assembler** translates assembly into machine code.

The two-pass assembler

A label can be used before the line that defines it (a jump **forward**), so the assembler reads the program twice.

First pass:

- reads the source code one line at a time;
- removes comments and white space;
- keeps count of the address of every line, and adds each **label** and its address to the **symbol table**;
- checks the opcodes against the instruction set (and reports errors).

Second pass:

- reads the code one line at a time again;
- **replaces every label with its address** from the symbol table;
- converts each opcode into its binary machine code;
- **generates the object code** (the machine code program).

For example, this program is loaded from address 0:

Address	Label	Opcode	Operand
0		LDD	COUNT

Address	Label	Opcode	Operand
1	LOOP:	DEC	ACC
2		CMP	#0
3		JPE	DONE
4		JMP	LOOP
5	DONE:	STO	COUNT
6		END	
7	COUNT:	5	

At line 3, DONE has not been seen yet; by the end of the first pass the symbol table is complete:

Symbol	Address
LOOP	1
DONE	5
COUNT	7

The second pass turns JPE DONE into JPE 5, LDD COUNT into LDD 7, and so on. (COUNT: 5 names a memory location and gives it a starting value.)

Instruction groups

Group	What the instructions do	Examples
Data movement	Copy values between memory and registers	LDM, LDD, LDI, LDH, LDR, MOV, STO
Input and output	Read a character from the keyboard, send one to the screen	IN, OUT
Arithmetic operations	Add and subtract	ADD, SUB, INC, DEC
Unconditional and conditional instructions	Jump to another address: always, or only if the last compare was true/false	JMP; JPE, JPN
Compare instructions	Compare the ACC with a value, setting a flag in the status register	CMP, CMI

The bit manipulation instructions (AND, OR, XOR, LSL, LSR) form a further group of logic operations.

Addressing modes

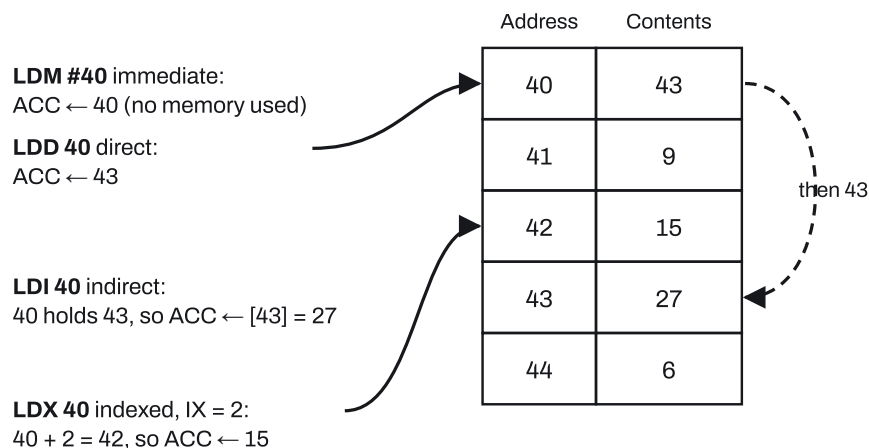
The **addressing mode** says how the operand is used to find the data.

- **Immediate:** the operand **is the value** itself. LDM #40 puts 40 in the ACC. The # means "this number".

- **Direct:** the operand is the **address of the data**. `LDD 40` loads whatever is stored at address 40.
- **Indirect:** the operand is the address of a location that **holds the address of the data**. `LDI 40` goes to address 40, reads an address there, then goes to *that* address and loads its contents.
- **Indexed:** the address of the data is the **operand plus the contents of the index register**. `LDX 40` with `IX = 2` loads from address 42. Changing `IX` walks through a list (an array) with the same instruction.
- **Relative:** the address is an **offset from a base address**, usually the address of the current instruction. An instruction at address 120 with relative operand 6 refers to address 126. Because nothing in the program uses fixed addresses, the program still works wherever it is loaded in memory (it is **relocatable**).

With memory addresses 40 to 44 holding 43, 9, 15, 27 and 6 and `IX = 2`:

- `LDM #40` gives `ACC = 40`;
- `LDD 40` gives `ACC = 43`;
- `LDI 40` gives `ACC = 27` (address 40 holds 43, and address 43 holds 27);
- `LDX 40` gives `ACC = 15` (address $40 + 2 = 42$).



`LDR #n` puts a number in the **IX**, not the **ACC**. `MOV IX` copies the **ACC** into the **IX**. `CMI` is a compare that uses indirect addressing.

Tracing a program

A **trace table** records the values as the program runs:

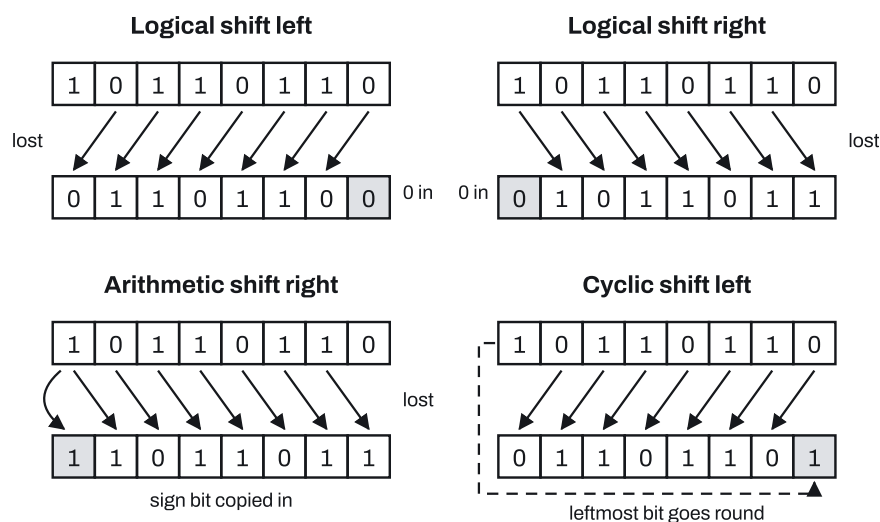
- After each instruction, write its address and **only the values it changes**: the **ACC**, the **IX**, a memory location (only `STO` changes memory) or the output.
- `CMP` changes nothing you record; it only sets a flag. The jump after it goes to its address if the compare was true (`JPE`) or false (`JPN`); otherwise the next instruction follows.
- `OUT` outputs the **character** whose ASCII code is in the **ACC**, so use the ASCII table given and write the character, not the code.
- `IN` puts the ASCII code of the next input character into the **ACC**.

- Keep going until `END`. Worked example 4 traces a full loop.

Binary shifts

A **shift** moves every bit in a register left or right by a number of places. Bits that move off the end are **lost**.

- **Logical shift:** the empty places are filled with **0s**. `LSL #n` and `LSR #n` in the instruction set are logical shifts.
- **Arithmetic shift:** used for **two's complement** (signed) numbers. On a right shift the **sign bit (the leftmost bit)** is **copied** into the empty places, so a negative number stays negative. An arithmetic left shift fills with 0s like a logical one.
- **Cyclic shift (rotation):** the bits that fall off one end **come back in at the other end**, so nothing is lost.



What shifts do to numbers. Each left shift doubles an unsigned number and each right shift halves it, dropping any remainder:

- `0000 1101 = 13`; `LSL #2` gives `0011 0100`, and `52 = 13 × 4` (shifting 2 places multiplies by 2^2);
- `LSR #3` on `0011 0100` gives `0000 0110 = 6`, because $52 ÷ 8 = 6.5$ and the half is lost.

Why the arithmetic shift copies the sign bit. `1110 1000` is -24 in two's complement. An arithmetic right shift of 2 places gives `1111 1010`, which is -6 ($-24 ÷ 4$). A logical right shift of 2 places gives `0011 1010`, which is 58: the sign has been lost.

Bit masks: testing and setting bits

In a device controller each bit of a register can stand for one thing: a sensor on or off, a light lit, a motor running.

Bit manipulation reads or changes single bits without disturbing the others, using a **mask** as the operand.

Line the bits up and work one column at a time:

Operation	Result bit is 1 when	Use it to
AND	both bits are 1	Clear bits (0 in the mask) or test a bit (1 in the mask for just that bit)

Operation	Result bit is 1 when	Use it to
OR	at least one bit is 1	Set bits (1 in the mask)
XOR	the bits are different	Flip (toggle) bits (1 in the mask)

A 0 in an AND mask clears that bit and a 1 keeps it; a 1 in an OR mask sets that bit and a 0 keeps it; a 1 in an XOR mask flips that bit and a 0 keeps it.

With the ACC holding 0101 1001:

- **Test bit 4** (value 16): AND #16 gives 0001 0000. Then CMP #16: equal, so the bit is 1 (had the bit been 0 the result would be 0000 0000). The same idea tests bit 2: AND #4 gives 0000 0000, so bit 2 is 0.
- **Set bit 1** (value 2): OR #2 gives 0101 1011.
- **Clear the upper four bits**: AND &0F gives 0000 1001.
- **Flip every bit**: XOR #255 gives 1010 0110.

#16, B00010000 and &10 are the same mask. An operand without a symbol is an **address**, and the mask is the byte stored there.

Key facts and formulas

There is no formula list in the 9618 exams. The instruction set, with what each instruction does, is printed in every question that needs it, so it is **Given**; everything else here is **Learn it**.

Formula	Status
Instruction set: LDM, LDD, LDI, LDX, LDR, MOV, STO, ADD, SUB, INC, DEC, JMP, CMP, CMI, JPE, JPN, IN, OUT, END, AND, OR, XOR, LSL, LSR and their explanations	Given
#n denary value, Bn binary value, &n hex value; no symbol means an address	Given
Fetch: MAR $\leftarrow$ [PC]; PC $\leftarrow$ [PC] + 1; MDR $\leftarrow$ [[MAR]]; CIR $\leftarrow$ [MDR]; then decode and execute	Learn it
[X] = contents of X; [[MAR]] = contents of the memory location whose address is in the MAR	Learn it
Address bus one way (from CPU); data and control buses both ways	Learn it
n address lines address 2^n memory locations	Learn it
Immediate: operand is the value. Direct: operand is the address. Indirect: operand holds the address. Indexed: operand + IX. Relative: offset from a base address	Learn it
Logical shift fills with 0; arithmetic right shift copies the sign bit; cyclic shift wraps round	Learn it

Formula	Status
Left shift by n : $\times 2^n$; right shift by n : $\div 2^n$, remainder lost (unsigned)	Learn it
AND with 0 clears, AND with 1 tests; OR with 1 sets; XOR with 1 flips	Learn it
Two-pass assembler: pass 1 builds the symbol table; pass 2 replaces labels and generates object code	Learn it

How to do it

We have tagged 51 Paper 1 questions for this topic (2021–2025; 49 different questions, 2 repeated on a second paper). 42 of the 51 mix more than one type, so the counts add up to more than 51.

Question type	Questions
Give the ACC after a bitwise (AND/OR/XOR) or shift (LSL/LSR) instruction	19
State the role of a special purpose register	13
Explain how a factor (clock, cores, cache, bus width) affects performance	11
Trace an assembly program in a trace table	10
Describe the CU, system clock, ALU and buses	9
Perform or describe a logical/arithmetic shift on a binary number	8
Give the ACC after load instructions using different addressing modes	7
Write a bit-manipulation instruction (mask) to set, clear or test bits	7
Name and describe addressing modes	7
Write or correct the F-E cycle in register transfer notation	7
Choose or explain a port (USB, HDMI, VGA)	7
Explain how an interrupt is detected and handled	6
Name instruction groups or give an instruction from each	6
Describe the Von Neumann model, stored program concept, IAS	3
State causes/purpose of interrupts; hardware vs software	3
Say which pass of a two-pass assembler does each task	3
Distinguish general and special purpose registers	2

The ideas are all in "Understand it". This section is about turning them into marks.

1. Give the ACC after a bitwise (AND/OR/XOR) or shift (LSL/LSR) instruction (19 questions)

1. **Write the operand as 8 bits:** # denary (#63 → 0011 1111), & hex, one nibble per digit (&3C → 0011 1100), B binary. No symbol means an **address**: use the byte stored there.
2. Write the ACC above the mask and work column by column ("Bit masks"); for LSL/LSR, move the bits and fill with 0s.
3. Give all 8 bits: 1 mark per byte, none for working.

Variations you will meet:

- **Short programs** (a shift then ADD #5, OR then INC ACC): apply them in order; ADD/INC are ordinary binary addition.
- **Other instructions mixed in (9618/13/O/N/25 Q6)**: a compare changes only the status register, so the ACC is unchanged; LDM #n loads n in binary.
- **"Reloaded before each instruction"**: every row starts from the same value.
- **Match instructions to results (9618/13/M/J/22 Q3)**: work each out, then draw the lines.

2. State the role of a special purpose register (13 questions)

Use the register table: say **what** it holds (address, data, instruction, flags) and **what for**. The PC holds the **address** of the next instruction, not the instruction.

Variations you will meet:

- **MAR and MDR in the F-E cycle** (4 marks): MAR gets the address from the PC; the MDR receives what is read from that address (or holds data to be written there); the instruction passes to the CIR.
- **"Identify a register other than ..."**: pick one not named in the question.
- **Cloze (9618/13/M/J/21 Q3)**: PC, MAR, MDR, CIR in fetch order.
- **Correct wrong statements (9618/12/M/J/25 Q1)**: watch for "instruction" where it should be "address", and "data bus" where it should be control bus.

3. Explain how a factor (clock, cores, cache, bus width) affects performance (11 questions)

1. Name the factor, then give the **mechanism** and its **effect** from "What makes a computer faster", as a linked pair (usually 2 marks per factor), for example "more F-E cycles per second ... so more instructions processed each second".
2. When two specifications are given, say which is higher and why that helps.

Variations you will meet:

- **"Why not twice as fast?" or "drawbacks of more cores" (9618/11/O/N/23 Q5, 9618/11/O/N/25 Q6)**: use the list of reasons under "Number of cores".

- **"Suggest a hardware upgrade"**: name it exactly (more cache, more cores, faster clock, wider bus, more RAM); "upgrade the CPU" scores nothing. More RAM means more running data fits, so less use of slower virtual memory and secondary storage.

4. Trace an assembly program in a trace table (10 questions)

1. Read the program first: find the loop, the jump back and the compare that ends it.
2. Follow "Tracing a program": one row per instruction that changes something, in order; look up the **current** memory value for every operand without #.
3. After CMP, decide true or false before applying JPE/JPN.
4. Stop at END, or exactly where the question says ("when line 90 runs a second time", "when the third input is needed").

Variations you will meet:

- **LDX, LDI, CMI inside a loop**: recalculate the address each time round.
- **"State the effect of changing one instruction"** (9618/12/M/J/25 Q7): re-run it with the change; typically a different value is loaded, the compare is never true and the program **loops forever**.
- **"State the purpose of the code"**: say what it achieves ("adds up the list", "swaps two values using a spare location", 9618/11/O/N/22 Q6).

5. Describe the CU, system clock, ALU and buses (9 questions)

Use "ALU, CU, system clock and IAS" and the bus table. For the clock give its **purpose** (timing signals that synchronise the components), not its speed.

Variations you will meet:

- **"How do the CU and clock work together?"**: the CU sends signals along the control bus on each pulse of the clock, so every component acts in step.
- **"Two signals on the control bus"**: name them (memory read, memory write, interrupt, clock, bus request).
- **Tick the bus**: MAR → address bus; MDR → data bus; clock and CU → control bus.
- **Cloze** (9618/12/M/J/21 Q5).

6. Perform or describe a logical/arithmetic shift on a binary number (8 questions)

1. Move every bit the stated number of places; bits pushed off the end are lost.
2. Fill with 0s for a logical shift; with copies of the leftmost bit for an arithmetic right shift; wrap the lost bits round for a cyclic shift.
3. Check the direction and give all 8 bits.

Variations you will meet:

- **"Describe the difference between a logical and an arithmetic right shift" (9618/13/O/N/24 Q8):** 0s in, against the sign bit copied in.
- **"What mathematical operation is LSR #3?" (9618/13/M/J/21 Q3):** integer division by $2^3 = 8$.

7. Give the ACC after load instructions using different addressing modes (7 questions)

1. Apply each mode as in "Addressing modes": LDM no look-up, LDD one, LDI two, LDX operand + IX then one.
2. Then the arithmetic: ADD 23 adds the **contents** of 23; ADD #23 adds 23.
3. Each program starts from the given ACC and IX unless the question says otherwise.

Variations you will meet: an **IX column** (only LDR, MOV, INC IX, DEC IX change it), and **naming the mode** of each load.

8. Write a bit-manipulation instruction (mask) to set, clear or test bits (7 questions)

1. Choose the operation: set → OR with 1s; clear → AND with 0s; flip → XOR with 1s; test → AND with a single 1.
2. Build the 8-bit mask and write it with a proper symbol: OR B10000000, OR #128 or OR &80 are all accepted.
3. "For **any** 8-bit number": check every other bit is kept whatever it was.

Variations you will meet:

- **Test for an odd number (9618/12/O/N/24 Q8):** odd numbers have a 1 in the least significant bit; AND #1 clears the rest; compare with 1; equal means odd.
- **Clear the register:** AND #0, then compare with 0 to confirm.
- **Turn one given byte into another:** only 0 → 1 changes, OR; only 1 → 0, AND; every bit flipped, XOR with all 1s; bits moved sideways, a shift such as LSL #2.

9. Name and describe addressing modes (7 questions)

Name the mode, then one sentence using the word **operand** (1 mark each). For a mode "not used in the program", check which appear and pick another.

Variations you will meet:

- **Relative addressing and its purpose (9618/13/O/N/22 Q6):** an offset from a base address, so the code can be loaded anywhere in memory.
- **Direct against indexed (9618/11/O/N/23 Q8):** both use the operand to reach an address in memory; indexed adds the IX first.

10. Write or correct the F-E cycle in register transfer notation (7 questions)

Write the four fetch lines in order, one per line, with exact brackets (see "The fetch-execute cycle"). A 4-mark version can give a mark for the order.

Variations you will meet:

- **Describe each given line in words** (9618/13/M/J/22 Q2): "the contents of ... are copied to ...".
- **Find and correct two errors** (9618/13/O/N/21 Q6): look for – instead of +, [MAR] instead of [[MAR]], or swapped registers; give the line number, the error and the corrected line.

11. Choose or explain a port (USB, HDMI, VGA) (7 questions)

1. **Choose:** USB for most peripherals; HDMI for a monitor, especially a high-resolution one with speakers.
2. **Explain** with the features in "Ports", tied to the scenario.

Variations you will meet:

- **HDMI rather than VGA** (4 marks): digital, audio and video in one cable, bandwidth for the resolution, less interference.
- **How USB connects automatically** (9618/11/O/N/24 Q3): voltage change → device ID → driver loaded.

12. Explain how an interrupt is detected and handled (6 questions)

Write the steps from "How an interrupt is handled", one mark each. The points most often missing are **when** (end of each F-E cycle), the **priority** check, and saving the registers on the **stack**. Variants: put lettered stages in order (9618/13/O/N/23 Q9); in context ("a key is pressed"), say what the ISR does.

13. Name instruction groups or give an instruction from each (6 questions)

Use the group table. For "give an instruction with a suitable operand", use only the instructions printed in the question, for example `STO 50` (data movement), `ADD 60` (arithmetic), `JPE 20` (conditional).

14. Describe the Von Neumann model, stored program concept, IAS (3 questions)

Stored program: instructions and data held together in main memory, fetched and executed one at a time. Components other than registers and buses: CU, ALU, system clock, IAS. The IAS is the memory the CPU accesses directly.

15. State causes/purpose of interrupts; hardware vs software (3 questions)

Purpose: a signal asking for the processor's attention. Classify by source: a physical device → hardware; a running program's error or request → software.

16. Say which pass of a two-pass assembler does each task (3 questions)

Use the lists in "The two-pass assembler". Reading line by line belongs to **both** passes, so it may need two lines.

17. Distinguish general and special purpose registers (2 questions)

Two-sided: special purpose registers have a fixed role and hold the state of execution; general purpose ones can be used for anything the programmer chooses and hold data during operations.

Worked examples

The bold codes show where marks are earned, as in Cambridge mark schemes. The number is how many marks.

- **B1** mark for a correct result on its own, with no method needed.

Example 1

The ACC currently contains the positive binary integer 1011 0011. The binary number 1011 0011 is reloaded into the ACC before each instruction is run. Address 50 contains 0101 1100.

(a) Complete the table with the contents of the ACC after each instruction. **[5]**

Instruction	ACC content
AND &0F	
XOR #85	
OR B01001000	
LSR #3	
AND 50	

(b) Explain how bit manipulation can be used to test whether bit 5 (the bit with place value 32) of the ACC is 1, and write the instruction used. **[3]**

(a)

- AND &0F: &0F = 0000 1111, so the top four bits are cleared: **0000 0011. B1**
- XOR #85: 85 = 0101 0101. Columns that differ give 1: 1011 0011 XOR 0101 0101 = **1110 0110. B1**
- OR B01001000: bits 6 and 3 set: **1111 1011. B1**
- LSR #3: move 3 places right, three 0s enter on the left, 011 is lost: **0001 0110. B1**

- AND 50: no #, so the mask is the contents of address 50, 0101 1100: 1011 0011 AND 0101 1100 = **0001 0000. B1**

(b)

- Use a mask with a 1 in bit 5 only and 0s elsewhere, and AND it with the ACC, so every other bit is cleared. **B1**
- Compare the result with 32 (0010 0000): if equal, bit 5 was 1; if the result is 0, bit 5 was 0. **B1**
- Instruction: AND #32 (or AND B00100000, AND &20). **B1**

For 1011 0011 the result is 0010 0000, equal to 32, so bit 5 is 1.

Example 2

The table shows the current contents of memory. The ACC contains 9 and the IX contains 1. Each program starts from these values.

Address	Data
70	74
71	5
72	71
73	12
74	20
75	3

(a) Write the contents of the ACC and the IX after each program has run. [5]

Program	Instructions	ACC	IX
1	LDM #72, ADD 73		
2	LDI 72, INC ACC		
3	LDD 70, SUB #4		
4	LDR #3, LDX 71, DEC ACC		

(b) Name the addressing mode used by LDX 71 and describe it. [2]

(a)

Program	ACC	IX
1	84	1
2	6	1
3	70	1

Program	ACC	IX
4	19	3

B1 for each ACC value, **B1** for the IX column:

- 1: the value 72, plus the **contents** of 73: $72 + 12 = 84$.
- 2: address 72 holds 71, and address 71 holds 5; then add 1: 6.
- 3: the contents of 70 is 74; subtract the value 4: 70.
- 4: `LDR #3` sets the **IX** to 3 (the ACC is untouched); `LDX 71` loads from $71 + 3 = 74$, which holds 20; subtract 1: 19. Only `LDR` changed the IX.

(b) Indexed addressing. **B1** The address of the data is found by adding the contents of the index register to the operand ($71 + 3 = 74$). **B1**

Example 3

(a) Describe the fetch stage of the fetch-execute cycle using register transfer notation. **[4]**

(b) While a program is running, the user presses a key. Explain how the resulting interrupt is detected and handled. **[5]**

(c) State one cause of a software interrupt. **[1]**

(a)

- $MAR \leftarrow [PC]$: the address in the PC is copied into the MAR. **B1**
- $PC \leftarrow [PC] + 1$: the PC is incremented to point to the next instruction. **B1**
- $MDR \leftarrow [[MAR]]$: the instruction at the address held in the MAR is copied into the MDR. **B1**
- $CIR \leftarrow [MDR]$: the instruction is copied into the CIR, ready to be decoded. **B1**

(b) Any five of:

- The keyboard sends an interrupt signal, which sets a flag in the interrupt register. **B1**
- At the end of the current F-E cycle the processor checks the interrupt register. **B1**
- It compares the priority of the keyboard interrupt with the current task; it is serviced only if its priority is higher. **B1**
- The contents of the registers, including the PC, are saved on the stack. **B1**
- The address of the keyboard's interrupt service routine is loaded into the PC and the ISR runs (for example reading the key's code). **B1**
- When it finishes, the processor checks for other interrupts, then restores the registers from the stack and continues the program. **B1**

(c) Division by zero (or a stack overflow). **B1**

Example 4

A processor has two registers, the Accumulator (ACC) and the Index Register (IX), and uses the syllabus instruction set (the exam prints the part you need).

The current contents of main memory are:

Address	Instruction
20	LDR #0
21	LDX 60
22	ADD 70
23	STO 70
24	INC IX
25	LDD 71
26	DEC ACC
27	STO 71
28	CMP #0
29	JPN 21
30	LDD 70
31	ADD #48
32	OUT
33	END

Address	Data
60	2
61	4
62	1
70	0
71	3

ASCII codes: 54 → '6', 55 → '7', 56 → '8'.

(a) Trace the program using a trace table with columns for the instruction address, the ACC, the IX, addresses 70 and 71, and the output. **[6]**

(b) State the purpose of the program. **[2]**

(a) The loop runs from 21 to 29. Address 71 is a counter starting at 3, and the loop repeats while it is not 0 (JPN jumps when the compare is false). Only rows where something changes are shown; the CMP and JPN rows change nothing.

Instruction address	ACC	IX	70	71	Output
			0	3	
20		0			
21	2				
22	2				
23			2		
24		1			
25	3				
26	2				
27				2	
21	4				
22	6				
23			6		
24		2			
25	2				
26	1				
27				1	
21	1				
22	7				
23			7		
24		3			
25	1				
26	0				
27				0	
30	7				
31	55				
32					7

B1 for each block: first value stored; counter 3 → 2; second pass (`LDR 60` with `IX = 1` reads address 61), total 6; counter 2 → 1; third pass, total 7, counter 0; `ACC 7` then 55 and output '7'. Note `ADD #48` adds the number 48, turning 7 into ASCII 55, the character '7'.

(b) It adds up the three numbers stored at addresses 60 to 62 (using the `IX` to step through them) **B1** and outputs the total as a character. **B1**

Common mistakes

- **Getting indexed addressing wrong.** `LDR 40` with `IX = 2` loads from address **42**. Examiners report students adding the `IX` to the *contents* of address 40, or loading 40 itself into the `ACC`. Add the `IX` to the **operand**, then look up that address.
- **Putting `LDR` in the wrong register.** `LDR #0` sets the `IX` to 0; the `ACC` does not change.
- **Ignoring #.** `ADD 12` adds what is stored at address 12; `ADD #12` adds 12. Check every operand.
- **Vague register roles.** "The index register holds the index" or "the status register shows the status" earns nothing. Say what it holds and what it is used for.
- **"The PC holds the next instruction."** It holds the **address** of the next instruction. The `MAR` also holds an address, the `MDR` holds data.
- **Restating the question on performance.** "A faster clock makes the computer faster" gets no marks. Say **how**: more F-E cycles per second, so more instructions processed per second; more cores process instructions **simultaneously**.
- **Interrupt answers that skip steps.** Many answers forget **when** interrupts are checked (end of each F-E cycle), the **priority** check, and that each type of interrupt has **its own ISR**. The current instruction is finished first; the program is not stopped instantly.
- **Describing the clock's speed when asked for its purpose.** The purpose is to produce timing signals that **synchronise** the components.
- **"Control signals" as a control bus signal.** Name them: read, write, interrupt, clock, bus request.
- **Misplaced brackets in register transfer notation.** `MDR ← [ [MAR] ]` needs two pairs; `MAR ← [PC]` one.
- **Shifting the wrong way, or rotating.** A logical left shift of 0000 1101 by 2 is 0011 0100, not a right shift and not a cyclic shift. Re-read "left"/"right" and "logical"/"cyclic".
- **Copying the opcode table** when asked "what mathematical operation does `LSR #2` perform": answer "integer division by 4".
- **Writing an ASCII code in the output column.** `OUT` outputs the **character**.
- **Mixing up the assembler passes.** The symbol table is built in the **first** pass; labels are replaced and object code produced in the **second**.

How to get the marks

- **One mark per point.** Give at least as many separate points as the marks, each on a new line, in technical terms: "control bus", "address", "interrupt service routine", not "wires" or "the program".

- **"State" / "Identify" / "Name"**: a short answer (a register, a mode, a port, a group). **"Describe"**: say what it is or does. **"Explain"**: a point plus **how** or **why** ("so more instructions are processed each second").
- **Byte answers**: always write all 8 bits. There are no part marks within a byte.
- **Trace tables**: one row per changed value, in order; follow the question's stopping point exactly. Marks are for blocks of rows, so a slip early can still leave later blocks correct if you carry on accurately.
- **Instructions you write** must come from the table printed in the question, with an operand in the right form (#, B or & for a value, no symbol for an address).
- **Register transfer notation**: exact register names and brackets, one step per line, in order.
- **Performance and port questions**: tie each point to the scenario (the given specifications, the high-resolution monitor with speakers, the flash drive), and give both halves: the feature and its effect.
- **Differences** must be two-sided: say what each of the two things does.
- **Interrupt handling**: aim for five distinct steps in order: detected at the end of the F-E cycle → priority → registers saved on the stack → ISR runs → check again, restore, continue.

Quick check

1. The ACC contains the two's complement integer 1010 0111. Give its contents after (a) an arithmetic right shift of 2 places, (b) a logical right shift of 2 places, (c) a logical left shift of 1 place, (d) a cyclic right shift of 1 place.
2. Write one bit manipulation instruction, for any 8-bit value in the ACC, that (a) sets bit 7 and bit 0 to 1 and leaves the rest unchanged, (b) clears the lower four bits and leaves the upper four unchanged, (c) flips every bit.
3. Addresses 30 to 34 contain 34, 12, 31, 7 and 9, and the IX contains 2. Give the ACC after each separate instruction: `LDM #32, LDD 32, LDI 32, LDX 31`.
4. (a) For each task, say which pass of a two-pass assembler does it: removes comments; generates object code; adds labels to the symbol table. (b) Name the instruction group of `JPN`, `OUT` and `CMI`.
5. (a) Give two reasons why a quad-core processor does not run programs four times as fast as a single core of the same clock speed. (b) How many memory locations can a 24-bit address bus address?

Answers

1. (a) 1110 1001: the sign bit 1 is copied into the two empty places. (b) 0010 1001: two 0s enter. (c) 0100 1110: a 0 enters on the right, the leftmost 1 is lost. (d) 1101 0011: the rightmost 1 wraps round to the left.
2. (a) OR B10000001 (or OR #129, OR &81): OR with 1 sets, with 0 keeps. (b) AND &F0 (or AND B11110000, AND #240): AND with 0 clears, with 1 keeps. (c) XOR &FF (or XOR #255, XOR B11111111).
3. LDM #32 gives 32; LDD 32 gives 31; LDI 32 gives 12 (address 32 holds 31, and address 31 holds 12); LDX 31 gives 7 (address $31 + 2 = 33$).
4. (a) First pass; second pass; first pass. (b) JPN: conditional instruction; OUT: input and output; CMI: compare.
5. (a) Any two: the software may not be written to use several cores, so some sit idle; the cores must communicate and share memory, which adds delay; some tasks must be done in sequence and cannot be split; cores can wait for each other. (b) $2^{24} = 16,777,216$ locations.

Past questions to try

- 9618/13/M/J/25 Q5: trace a loop with a compare and jumps, name two addressing modes, and set the most significant bit.
- 9618/13/O/N/24 Q7: the ACC after short programs using direct, indirect and immediate loads, then clearing a register with a mask.
- 9618/11/O/N/22 Q6: a longer trace with indexed addressing and ASCII output, the purpose of a code fragment that swaps two values, bitwise results and the assembler passes.
- 9618/11/M/J/23 Q6: how the computer handles an interrupt from a key press.