Hardware and Virtual Machines

A-Level Computer Science

RISC vs CISC processors

Two styles of CPU design. The CPU itself plugs into the **motherboard**, the main board that links the processor, the memory and every other part of the computer together.

CISC	RISC
many, complex instructions	few, simple instructions
variable length	fixed length
does more per instruction	each instruction is fast

CISC has many complex instructions; RISC has few simple ones

A motherboard links the CPU, memory and other parts together

■ CISC

A **CISC** (Complex Instruction Set Computers) has **many, often complex** instructions (one may do several memory accesses and operations), of **variable length**, so decoding is intricate. It does more per instruction in hardware. Examples: Intel x86.

■ RISC

A **RISC** (Reduced Instruction Set Computers) has a **small set of simple** instructions, each doing one basic operation, all of **fixed length** (fast to decode). Only **load** and **store** touch memory; everything else is **register** to register. Programs are longer but each instruction is quick and predictable, which suits pipelining. Examples: ARM, RISC-V.

Feature	CISC	RISC
Instruction set	many	few
Instruction length	variable	fixed
Memory access	many instructions	only load/store
Pipeline-friendly	harder	naturally
Per-instruction cycles	varies	usually 1

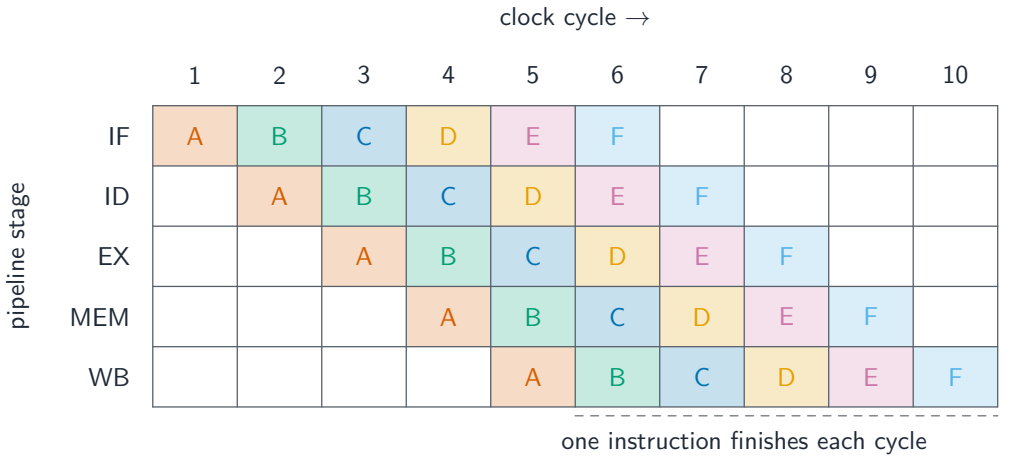
The trade-off is doing **more per instruction** (CISC) vs doing each instruction **faster and more predictably** (RISC). Modern Intel chips translate CISC instructions into simpler RISC-like micro-ops internally.

"Identify four features of a RISC processor." Any four of: a **small** set of **simple** instructions; instructions of **fixed length** (one word); most instructions complete in **one clock cycle**; **many** general-purpose registers; only **load and store** instructions access memory (all arithmetic is register to register); **hard-wired** control (no microcode); designed for **pipelining**; the **compiler** does more of the work, so programs contain more instructions and need more memory. **"Identify four features of a CISC processor."** Any four of: a **large** set of instructions, many of them **complex** (one instruction may do several operations); instructions of **variable length**; instructions that take **several clock cycles**; **fewer** registers; instructions that can **access memory directly**; **microprogrammed** control; less suited to pipelining; **shorter** programs, so a simpler compiler and less memory. **"Describe what is meant by RISC and CISC" (two marks each):** name the expansion and give the defining idea (few simple single-cycle instructions; many complex multi-cycle instructions).

Interrupt handling on the two designs. On a **CISC** processor the current instruction, however complex, is **completed** before the interrupt is serviced; the processor then saves the contents of its registers (including the program counter) on the stack, jumps to the interrupt service routine, and restores the registers afterwards. On a **RISC** processor with a **pipeline**, several instructions are part-way through at the moment the **interrupt** arrives, so the processor must either let every instruction in the pipeline finish, or **discard** (flush) the partly executed instructions and restart them after the interrupt; either way the pipeline is emptied, the registers are saved, and the service routine runs. The exam phrasing: "pipelining makes interrupt handling more complex, because the contents of the pipeline must be dealt with before the interrupt can be serviced".

Pipelining

A **pipeline** processes instructions in **overlapping stages**, like an assembly line: Fetch → Decode → Execute (in the **ALU**) → Memory access → Write back. Each stage works on a different instruction at once, so once the pipeline is full, **one instruction completes per cycle**. RISC's fixed-length, simple instructions make every stage take the same time. A pipeline can stall on a **hazard** – a data hazard (an instruction needs a result not ready yet) or a control hazard (a branch makes the next address unknown).



Pipelining overlaps the stages of six instructions, so one finishes each cycle

RISC chips keep data in **many registers** because memory is slow and registers are fast; the compiler allocates values to registers wisely.

"Describe the use of pipelining in RISC processors" (three marks). (1) The **fetch–execute cycle is divided into stages** (fetch, decode, execute, memory access, write back); (2) **several instructions** are in the pipeline at once, each at a **different stage**, so while one is being executed the next is being decoded and the one after fetched; (3) a new instruction is started, and one completed, in **every clock cycle** once the pipeline is full, which increases **throughput** (the number of instructions completed per second), although each instruction still takes the same time on its own. Fixed-length single-cycle RISC instructions are what make the stages equal and the pipeline possible.

> **Worked example.** A processor uses five pipeline stages (IF, ID, OF, EX, WB). Four instructions enter the pipeline one after another. In which cycle does the last instruction complete, and how many cycles would the four take without pipelining?

Instruction 1 occupies IF in cycle 1, ID in 2, OF in 3, EX in 4 and WB in 5; instruction 2 starts one cycle later and finishes in cycle 6; instruction 3 in cycle 7; instruction 4 in cycle **8**. In general n instructions through k stages take $n + k - 1$ cycles, here

$4 + 5 - 1 = 8$. Without pipelining each instruction takes all five cycles before the next starts: $4 \times 5 = 20$ cycles. The exam's table is filled by writing each instruction's stages diagonally, one column to the right of the previous instruction.

A processor running this fast gives off a lot of heat, so a **heat-sink** and fan sit on top of it. The metal fins spread the heat and the fan blows it away, keeping the CPU cool enough to work.

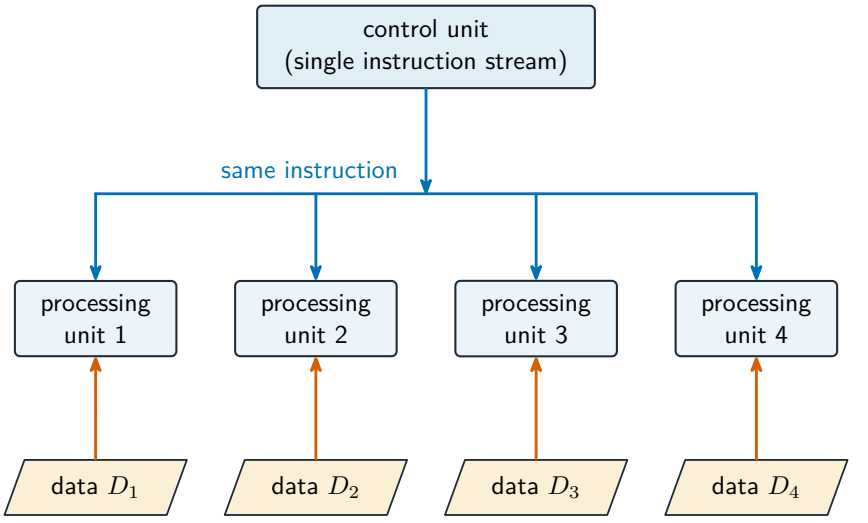
A CPU heat-sink and fan carry heat away from the processor

Flynn's taxonomy

Flynn's taxonomy sorts computers by the number of instruction and data streams:

- **SISD** – one instruction, one data stream (a traditional single core).
- **SIMD** – one instruction works on **many data items at once** (GPUs, CPU vector extensions). Great for images, video, scientific arrays.
- **MISD** – several operations on the same data; rare, mostly theoretical.
- **MIMD** – many processors run **different instructions on different data** (multi-core CPUs, clusters). The most general.

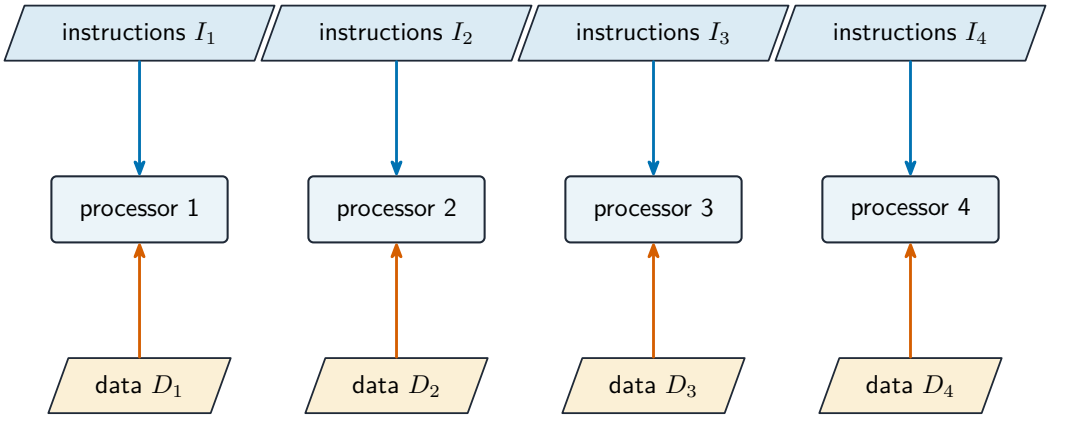
Describing the four architectures (two marks each). *SISD:* a **single processor** executes **one instruction at a time** on **one item of data**; no parallelism, the traditional von Neumann machine. *SIMD:* **one instruction** is applied simultaneously to **many data items**, by many processing elements acting in step; used for array and graphics processing. *MISD:* **several processors** apply **different instructions** to the **same data**; rarely used, for example a fault-tolerant system where several processors check one stream. *MIMD:* **many processors**, each executing **its own instructions** on **its own data**, independently; the multi-core computer and the cluster.



SIMD: many processors run the same instruction on different data

A **graphics card** (with its GPU) is a real example of SIMD hardware: it has thousands of small cores that run the same instruction on many pixels or numbers at once, which is why GPUs are so fast for images, video and machine learning.

A graphics card: its GPU runs the same instruction on many data items at once (SIMD)



MIMD: each processor runs its own instructions on its own data

Massively parallel computers

A **massively parallel** system uses **thousands of processors** on a fast network, each with its own memory (**distributed memory**), exchanging data by messages. It is MIMD, needs specially-written software (MPI, CUDA), and suits climate simulation, large **machine learning** training, and astrophysics. The largest **supercomputers** are massively parallel.

"Outline the characteristics of massively parallel computers" (three marks). A **very large number of processors** (thousands), **each with its own memory**, connected by a **network** (a high-speed interconnect or bus) so that they can pass **messages** to one another; they work **simultaneously** on parts of the **same problem**, so the problem must be written as a program that can be split into parts that run in parallel and combine their results. It is an MIMD arrangement.

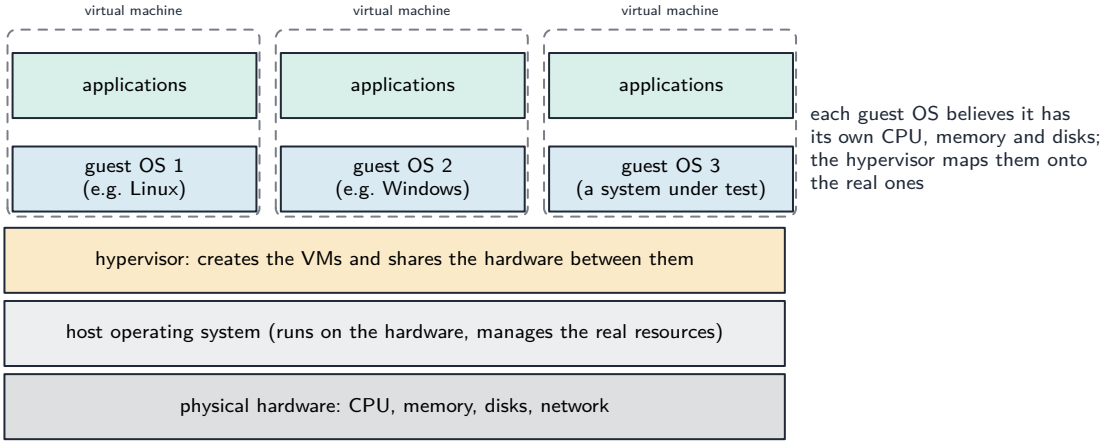
The processors live in tall **server** racks, often filling a whole room (a **data centre**), wired together so they can work on one big problem at the same time.

Rows of servers in a data centre, like those used for massively parallel computing

Virtual machines

A **virtual machine** (VM) is a **software emulation of a whole computer** – the software inside sees a CPU, memory and disks that look real but are managed by host software.

- a **system VM** runs a complete OS. A **hypervisor** creates and manages VMs, each booting its own guest OS. Uses: run different OSes on one machine; server consolidation; **sandboxing** (risky software runs isolated); snapshots.
- a **process (language) VM** runs one program in portable **bytecode** – the JVM (Java), the CLR (.NET), CPython. Benefits: portability ("write once, run anywhere"), runtime safety checks, and **just-in-time compilation** for near-native speed. The cost is an extra layer and needing the VM installed.



One real computer, several apparent ones: the host operating system and hypervisor share the hardware, and each guest operating system runs as if it had a machine of its own

"Describe what is meant by a virtual machine" (two marks). *A software emulation (implementation) of a computer system that runs on a host computer and behaves, to the programs running inside it, like a separate physical computer with its own processor, memory and storage.* The **host operating system** runs on the actual hardware, manages the real resources and (through the hypervisor) creates and controls the virtual machines; each **guest operating system** runs inside a virtual machine, manages the applications in it, and is unaware that its hardware is virtual.

Benefits (give two). Several **different operating systems** can run on one machine at the same time; software can be **tested** on many systems without buying the hardware; a **new** computer system can be emulated and tried before it is built; each VM is **isolated**, so a crash or malware in one does not affect the host or the others; VMs can be **copied, moved and backed up** as files, and a server can be shared between many users, **reducing** hardware cost. **Limitations (give two).** A VM runs **more slowly** than the real hardware because every instruction passes through the emulation layer; it consumes the host's **memory and processing power**, so the host must be powerful; some hardware features or devices are **not emulated** exactly, so the tested software may behave differently on the real machine; **licences** are needed for each guest OS, and setting the system up needs expertise.

Boolean algebra

Boolean algebra simplifies **Boolean** expressions, which can equally be described by **truth tables** . Symbols: + for OR, · for AND (often omitted), an overbar for NOT.

Key laws include commutative, associative and distributive (as in ordinary algebra),

plus:

- identity $A + 0 = A$, $A \cdot 1 = A$; null $A + 1 = 1$, $A \cdot 0 = 0$.
- idempotent $A + A = A$; inverse $A + \overline{A} = 1$, $A \cdot \overline{A} = 0$.
- **De Morgan's laws** : $(A + B)' = A' \cdot B'$; $(A \cdot B)' = A' + B'$ – negate the whole, swap AND/OR, negate each operand.
- **absorption** : $A + AB = A$.

Simplifying reduces the number of terms, so the resulting logic circuit has fewer gates. Example: $Z = AB + A\overline{B} = A(B + \overline{B}) = A$.

The laws with their names (quote the name at each step when "show all working" is asked).

Law	OR form	AND form
identity	$A + 0 = A$	$A \cdot 1 = A$
null (annulment)	$A + 1 = 1$	$A \cdot 0 = 0$
idempotent	$A + A = A$	$A \cdot A = A$
complement (inverse)	$A + \overline{A} = 1$	$A \cdot \overline{A} = 0$
commutative	$A + B = B + A$	$A \cdot B = B \cdot A$
associative	$A + (B + C) = (A + B) + C$	$A(BC) = (AB)C$
distributive	$A + BC = (A + B)(A + C)$	$A(B + C) = AB + AC$
absorption	$A + AB = A$	$A(A + B) = A$
De Morgan	$\overline{A + B} = \overline{A} \cdot \overline{B}$	$\overline{A \cdot B} = \overline{A} + \overline{B}$
double negation	$\overline{\overline{A}} = A$	

> **Worked example.** Simplify $X = \overline{\overline{(A \cdot B)} \cdot \overline{(A + B)}}$, showing all working.

$X = \overline{\overline{(A \cdot B)}} + \overline{\overline{(A + B)}}$ (De Morgan on the outer bar) $= A \cdot B + A + B$ (double negation) $= A + B$ (absorption, $A + AB = A$, applied with $A + B$ absorbing AB).

> **Worked example.** Simplify $(\overline{A + B}) \cdot (\overline{A} + B)$.

$= \overline{A} \cdot \overline{B} \cdot (\overline{A} + B)$ (De Morgan) $= \overline{A}\,\overline{B}\,\overline{A} + \overline{A}\,\overline{B}\,B$ (distributive) $= \overline{A}\,\overline{B} + 0$ (idempotent, complement) $= \overline{A}\,\overline{B}$.

> **Worked example.** Simplify $Y = \overline{A}\,\overline{B}\,\overline{C} + \overline{A}\,\overline{B}\,C + A\,\overline{B}\,C$.

$= \overline{A}\,\overline{B}(\overline{C} + C) + A\,\overline{B}\,C$ (distributive) $= \overline{A}\,\overline{B} + A\,\overline{B}\,C$ (complement, identity) $= \overline{B}(\overline{A} + AC)$ (distributive) $= \overline{B}(\overline{A} + C)$, using $\overline{A} + AC = (\overline{A} + A)(\overline{A} + C) = \overline{A} + C$. Applying De Morgan to a three-input term works the same way: $\overline{A + B + C} = \overline{A} \cdot \overline{B} \cdot \overline{C}$.

Sum-of-products from a truth table. Take every row whose output is 1, write the AND of its inputs (a variable barred where it is 0), and OR the terms: a row with $A = 1, B = 0, C = 1$ gives $A\,\overline{B}\,C$. This is the **sum-of-products** form the exam asks for, and it is the starting point for both algebraic simplification and the

Karnaugh map.

Karnaugh maps

A **Karnaugh map** (K-map) simplifies a Boolean expression by **grouping adjacent 1s** from a truth table. Columns and rows use **Gray code** order (`00`, `01`, `11`, `10`) so adjacent cells differ in one variable.

Place a 1 in each cell where the output is 1. Find rectangular groups of 1s whose sides are powers of 2 (1, 2, 4, 8), wrapping around edges if it makes a bigger group. **The larger the group, the simpler the term**: a group of 2 drops one variable, a group of 4 drops two, and so on – variables that change within the group disappear. OR the group terms together for the simplified expression. Cover every 1 using as few, as large, groups as possible.

> **Worked example.** A Karnaugh map for A and B has 1s in the cells $\overline{A}B$ and AB. Simplify. The two 1s are **adjacent** - they share the $B = 1$ column - so group them as a rectangle of 2. Inside that group B stays 1 throughout while A **changes** from 0 to 1, and any variable that changes within a group **disappears**. So the group leaves simply $X = B$. Compare that with the sum of products read straight off the table, $\overline{A}B + AB$: the same circuit, two gates fewer. Two rules do most of the work - make each group **as large as possible** (a group of 2 drops one variable, 4 drops two, 8 drops three), and remember the map **wraps around** its edges, so the leftmost and rightmost columns are adjacent. That wrap is the grouping most candidates miss.

three variables: the worked example

$C \backslash AB$	00	01	11	10
0	1	1	0	1
1	1	1	0	1

red loop of 4 (columns 00, 01): $A = 0$ throughout, so $\overline{A}$
blue loop of 4 wrapping round (columns 00, 10): $B = 0$, so $\overline{B}$
$Z = \overline{A} + \overline{B}$

four variables: a wrap-around loop

$CD \backslash AB$	00	01	11	10
00	1	0	0	1
01	0	0	0	0
11	0	0	0	0
10	1	0	0	1

the four corners are one loop of 4: $B = 0$ and $D = 0$
in every corner, so the term is $\overline{B}\,\overline{D}$

Loops of 1, 2, 4 or 8 ones; the term for a loop keeps only the variables that do not change inside it. Edges join, so a loop may wrap round, and the four corners count as adjacent

Building and reading a K-map. Label the columns AB and the rows C (or CD) in Gray-code order `00 01 11 10`, so that neighbouring cells differ in one variable only. Put a 1 in every cell whose minterm appears in the expression (or whose truth-table row outputs 1). Then draw the **fewest, largest** loops that cover every 1: each loop

must be a rectangle of $1, 2, 4$ or 8 cells, loops may **overlap**, may **wrap** across the left–right and top–bottom edges, and the four corners together make a loop. For each loop write the variables that are **constant** inside it (barred if 0), and OR the loop terms: that is the optimal sum-of-products. **Why use one?** It gives the simplest expression **without algebra**, in a few steps, with less chance of error, and the same map suits three or four variables.

> **Worked example.** $Z = \overline{A}\,\overline{B}\,\overline{C} + \overline{A}\,\overline{B}\,C + \overline{A}\,B\,\overline{C} + \overline{A}\,B\,C + A\,\overline{B}\,\overline{C} + A\,\overline{B}\,C.$

On the three-variable map the 1s fill columns 00, 01 and 10 in both rows. The loop of four over columns 00 and 01 has $A = 0$ throughout and B, C both varying: term $\overline{A}$. The loop of four over columns 00 and 10 (wrapping round) has $B = 0$ throughout: term $\overline{B}$. So $Z = \overline{A} + \overline{B}$, which Boolean algebra confirms: $\overline{A}(\overline{B} + B) + \ldots = \overline{A} + \overline{B}$. Two loops of two would also be correct but not **optimal**; a loop is as large as the 1s allow.

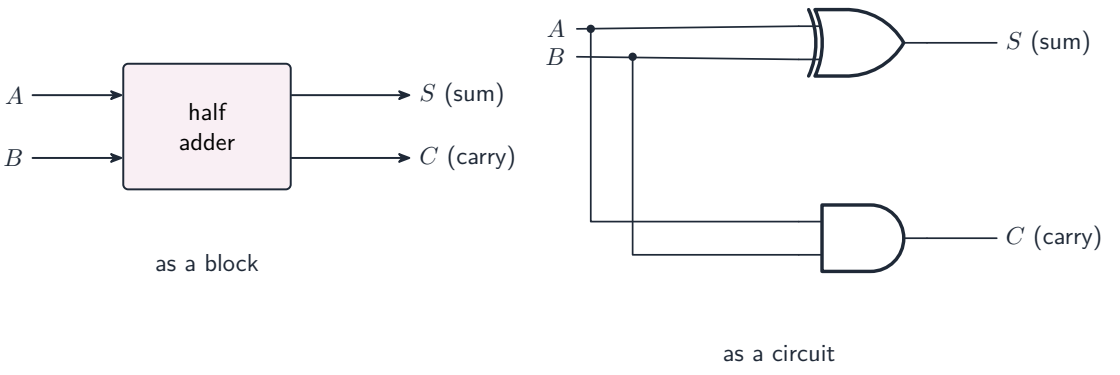
Worked example (four variables). A map has 1s only in its four corners: $\overline{A}\,\overline{B}\,\overline{C}\,\overline{D}$, $A\,\overline{B}\,\overline{C}\,\overline{D}$, $\overline{A}\,\overline{B}\,C\,\overline{D}$ and $A\,\overline{B}\,C\,\overline{D}$. Because the top and bottom rows are adjacent and so are the outer columns, the corners are one loop of four; $B = 0$ and $D = 0$ in all of them while A and C vary, so $Z = \overline{B}\,\overline{D}$.

Half adder and full adder

A **half adder** adds two single bits A and B, giving a sum S and a **carry** C:

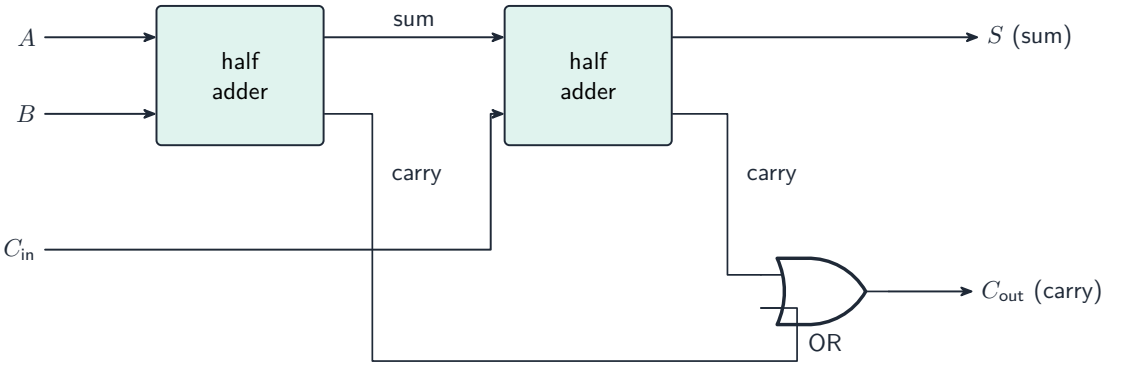
A	B	S	C
0	0	0	0
0	1	1	0
1	0	1	0
1	1	0	1

So $S = A$ XOR B and $C = A$ AND B. It ignores any carry-in – hence "half".



A half adder, as a block and as a circuit of an XOR and an AND gate

A **full adder** adds three bits (A, B, carry-in), giving a sum and a carry-out: $S = A \text{ XOR } B \text{ XOR } C_{\text{in}}$. It can be built from two half adders plus an OR gate. Chaining full adders (each carry-out feeding the next carry-in) makes a multi-bit "ripple-carry" adder.



A full adder is built from two half adders and an OR gate

The full-adder truth table. With inputs A, B and the carry-in C_{in}: the sum S is 1 when an **odd** number of inputs is 1, and the carry-out is 1 when **two or more** inputs are 1.

A	B	C_{in}	S	C_{out}
0	0	0	0	0
0	0	1	1	0
0	1	0	1	0
0	1	1	0	1
1	0	0	1	0
1	0	1	0	1
1	1	0	0	1
1	1	1	1	1

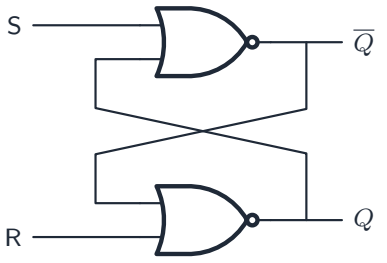
The circuit questions the exam sets. Given a circuit of an XOR and an AND gate sharing two inputs, or two half adders and an OR gate, "complete the truth table (show your working)" means adding a **column for every intermediate gate output** and filling the rows in order; "state the name of the circuit" is **half adder** or **full adder**; "state the purpose of each output" is the **sum** of the bits and the **carry** to the next column. Sum-of-products for the half adder: $S = \overline{A}B + A\overline{B}$, $C = AB$. A chain of full adders, each passing its carry-out to the next carry-in, adds two multi-bit numbers.

Flip-flops

A **flip-flop** is a **bistable** circuit – two stable states (0 and 1) – that **remembers** its state. It stores one bit and is the basic element of registers and SRAM.

■ SR flip-flop

An **SR flip-flop** SR has inputs **S** (set) and **R** (reset) and outputs Q and $\overline{Q}$. `S=1,R=0` sets Q to 1; `S=0,R=1` resets it to 0; `S=0,R=0` holds; `S=1,R=1` is **invalid**. Built from two cross-coupled NOR gates.



two NOR gates, each output fed back to the other's input

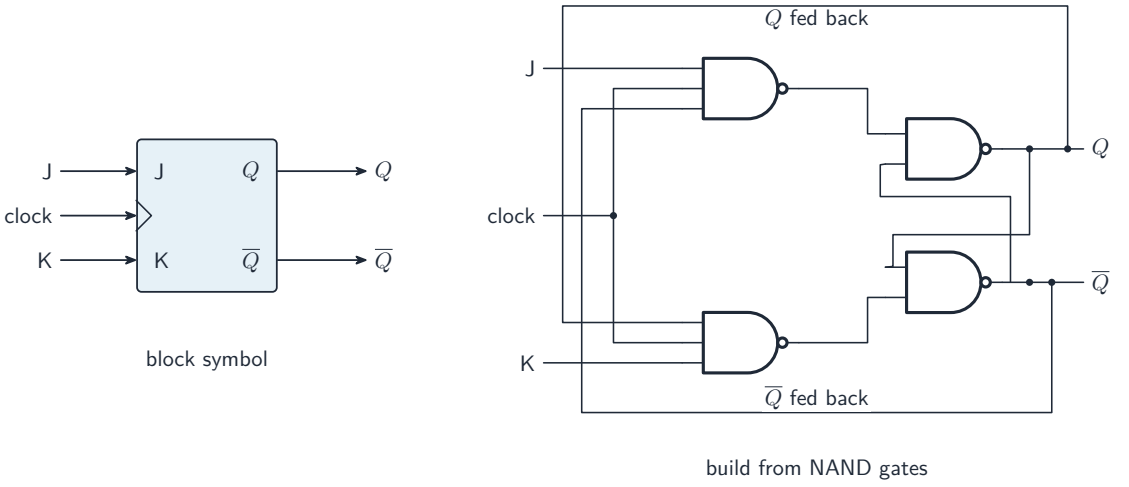
S	R	Q	$\overline{Q}$	effect
0	0	Q	$\overline{Q}$	hold: no change (memory)
1	0	1	0	set Q to 1
0	1	0	1	reset Q to 0
1	1	0	0	invalid: both outputs 0

The SR flip-flop: two NOR gates feeding each other. With both inputs 0 the outputs hold whatever they were, which is the memory; S sets Q to 1, R resets it, and S = R = 1 is not allowed

"Draw a logic circuit for an SR flip-flop and label the inputs." Two **NOR** gates (or two NAND gates), the output of each connected back to one input of the other; the free input of one gate is **S**, of the other **R**; the outputs are Q and $\overline{Q}$. The **feedback** is what the marks are for: without it there is no memory. **"State the purpose of a flip-flop."** *To store one bit of data*; it is the basic memory element from which registers and static RAM are built, and it holds its value until it is deliberately changed. The invalid input $S = R = 1$ makes both outputs 0, so that $\overline{Q}$ is no longer the complement of Q, and the state after both inputs return to 0 is unpredictable, which is the SR flip-flop's weakness.

■ JK flip-flop

A **JK flip-flop** JK improves on it by using the previously-invalid `1,1` input as a **toggle** (the output flips). This makes it ideal for building **counters** (a chain of toggling flip-flops). It is usually **clocked** – inputs act only on a clock edge, keeping flip-flops synchronised.



A JK flip-flop: its symbol and a build from NAND gates

Flip-flops are the building blocks of registers (n bits = n flip-flops), counters, and **SRAM** RAM cells.

JK flip-flop truth table. The **clock** input decides when the J and K inputs are read, so the output changes only on a clock pulse: with $J = K = 0$ the output is **held**; $J = 1, K = 0$ **sets** Q to 1; $J = 0, K = 1$ **resets** it to 0; $J = K = 1$ **toggles** it (Q becomes $\overline{Q}$). The last row is exactly the SR flip-flop's forbidden input turned into a useful one, which is why the JK is preferred: every input combination is valid, and the clocked operation makes it the building block of counters and shift registers.

Definitions the examiner accepts

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

Term	Definition
RISC	a processor with a small set of simple, fixed-length instructions, most executed in one clock cycle, using many registers and pipelining
CISC	a processor with a large set of complex, variable-length instructions, many taking several clock cycles and accessing memory directly
pipelining	dividing the fetch–execute cycle into stages so that several instructions are processed at once, each at a different stage
SISD / SIMD / MISD / MIMD	one instruction on one data item; one instruction on many data items; many instructions on one data item; many instructions on many data items

Term	Definition
massively parallel computer	thousands of processors, each with its own memory, connected by a network and working simultaneously on one problem
virtual machine	a software emulation of a computer system running on a host computer and behaving like a separate physical computer
hypervisor	the software that creates virtual machines and shares the host's hardware between them
truth table	a table listing every combination of inputs to a logic circuit with the resulting output(s)
sum-of-products	a Boolean expression written as the OR of AND terms, one term for each input combination giving 1
Karnaugh map	a grid of the truth-table outputs, arranged in Gray-code order, in which loops of adjacent 1s give the simplified expression
half adder	a circuit that adds two bits, producing a sum and a carry
full adder	a circuit that adds two bits and a carry-in, producing a sum and a carry-out
flip-flop	a bistable circuit that stores one bit, holding its output until its inputs change it

Exam tips

- RISC and CISC are answered as lists of features: simple, fixed, one cycle, many registers, load/store, pipelined against complex, variable, multi-cycle, fewer registers, direct memory access, microcode. Four of each.
- Pipelining: stages, several instructions at once, one completed per cycle, higher throughput; $n + k - 1$ cycles for n instructions through k stages; interrupts must empty the pipeline.
- Flynn's four categories are "how many instruction streams" by "how many data streams"; say what runs on what. Massively parallel: many processors, own memory, network, same problem.
- Virtual machine: emulation of a computer on a host; host OS on the hardware, hypervisor sharing it, guest OS inside. Two benefits and two limitations, each a full sentence.
- Boolean algebra: name each law as you use it; De Morgan swaps the operator and negates each term; check with a truth table if in doubt.

- K-map: Gray-code order, largest loops of 1/2/4/8, wrapping allowed, one term per loop with the unchanging variables. State why: simplest expression with no algebra.
- Half adder gives sum and carry; full adder also takes a carry-in; SR flip-flop is two cross-coupled NOR/NAND gates and stores one bit; JK's 1,1 input toggles.

■ Common mistakes

- Swapping the RISC and CISC feature lists, or offering "faster" as a feature; give the design features, not a verdict.
- Describing pipelining as "running instructions in parallel on several cores"; it is stages of one processor overlapping.
- Confusing SIMD (one instruction, many data) with MIMD (many of both), or describing MISD as the common case.
- Defining a virtual machine as "a copy of a computer" without the word emulation or the host and guest.
- Applying De Morgan to only part of an expression under a long bar, or dropping the bar without swapping AND for OR.
- Looping a group of three, or a non-rectangular group, in a K-map; ordering the columns 00, 01, 10, 11 instead of Gray code.
- Writing the carry of a half adder as XOR and the sum as AND.
- Drawing an SR flip-flop as two gates with no feedback, or leaving out the invalid state from its truth table.