

Hardware and virtual machines

A-Level Computer Science 9618 · Topic 15 revision notes

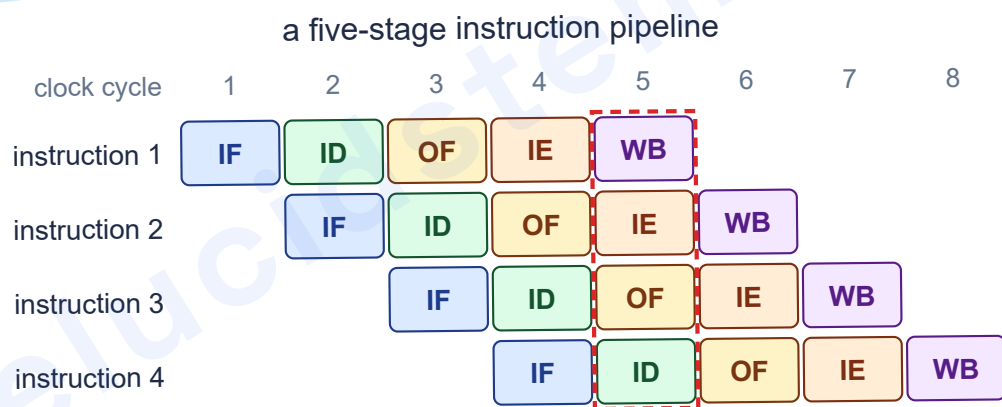
Everything from the 6 lessons of this topic on a few pages: the key idea of each lesson, the diagram to remember, the points to learn and the traps examiners set. Read it, cover it, test yourself.

- | | |
|--|---|
| 1. <u>RISC and CISC processors and pipelining.</u> | 4. <u>Boolean algebra</u> |
| 2. <u>Parallel processing architectures</u> | 5. <u>Karnaugh maps</u> |
| 3. <u>Virtual machines</u> | 6. <u>Half adders, full adders and flip-flops</u> |

1 RISC and CISC processors and pipelining

Syllabus 15.1

A **RISC** (Reduced Instruction Set Computer) processor has a **small set of simple, fixed-length instructions**, most of which execute in **one clock cycle**; only LOAD and STORE instructions access memory, and all other operations work on **registers**, so a RISC processor has **many general-purpose registers**. A **CISC** (Complex Instruction Set Computer) processor has a **large set of complex, variable-length instructions**, many of which take **several clock cycles** and can work directly on memory. Because RISC instructions are uniform, they suit **pipelining**: the fetch-execute cycle is split into stages, and while one instruction is being executed the next ones are already being decoded and fetched, so once the pipeline is full **one instruction completes every clock cycle**. An **interrupt** in a pipelined processor means the partly processed instructions must either be allowed to finish or be **flushed** before the interrupt service routine runs.



cycle 5: four instructions at different stages at once

4 instructions take $5 + 4 - 1 = 8$ cycles, not $4 \times 5 = 20$

Fig. 1: a five-stage pipeline (instruction fetch, instruction decode, operand fetch, instruction execute, write back). Each instruction still takes five cycles, but a new one starts every cycle.

Worked example: One task, two designs

Two values held in memory must be multiplied and the result stored back in memory. Compare how a CISC and a RISC processor do this.

1. CISC: a single complex instruction fetches both operands from memory, multiplies them and stores the result. It is one instruction, but it takes several clock cycles.

2. RISC: LOAD the first value into a register, LOAD the second value into another register, MULTIPLY the two registers, STORE the result. Four simple instructions.
3. Each RISC instruction takes about one clock cycle and has the same format, so the four can be pipelined and overlap.
4. The RISC program is longer (more memory for the code) and the compiler does more work; the CISC hardware is more complex.

Answer: CISC: fewer instructions, each taking several cycles. RISC: more instructions, each taking one cycle and suited to pipelining

Compare, then explain why.

- ✓ **Differences:** give each one as a matched pair, saying what RISC does and what CISC does.
- ✓ **Pipelining:** name the stages and say that different instructions are at different stages at the same time.
- ✓ **Cycle counts:** $k \times n$ without a pipeline, $k + n - 1$ with one; show both before subtracting.
- ✓ **Interrupts:** flush or complete the pipeline, save the PC and registers, run the ISR, restore and refill.

Watch out!

Do not say RISC processors "cannot do" complex operations. They do them with a **sequence of simple instructions**. The difference is where the complexity lives: in the hardware (CISC) or in the compiler and program (RISC).

Pipelining does **not** make a single instruction faster: each one still passes through all five stages. It increases **throughput**, the number of instructions completed per second. A jump or a branch can also force the instructions already fetched behind it to be **flushed**, wasting cycles.

When a pipeline is flushed, the discarded instructions are **not lost**: the saved program counter points to the first one that did not complete, so they are fetched again after the ISR.

2 Parallel processing architectures

Syllabus 15.1

Flynn's classification describes a computer by its **instruction streams** and **data streams**. **SISD** (single instruction, single data): one processor executes one instruction on one data item at a time, as in the Von Neumann model. **SIMD** (single instruction, multiple data): one instruction is applied to **many data items at the same time**, as in array processors and graphics processors. **MISD** (multiple instruction, single data): several processors apply different instructions to the **same data**; it is rare. **MIMD** (multiple instruction, multiple data): several processors each execute **their own instructions on their own data**, as in multicore processors. A **massively parallel computer** links a **very large number of processors**, each usually with its own memory, by a high-speed network so that they work **simultaneously on different parts of the same problem**.

Flynn's four basic computer architectures

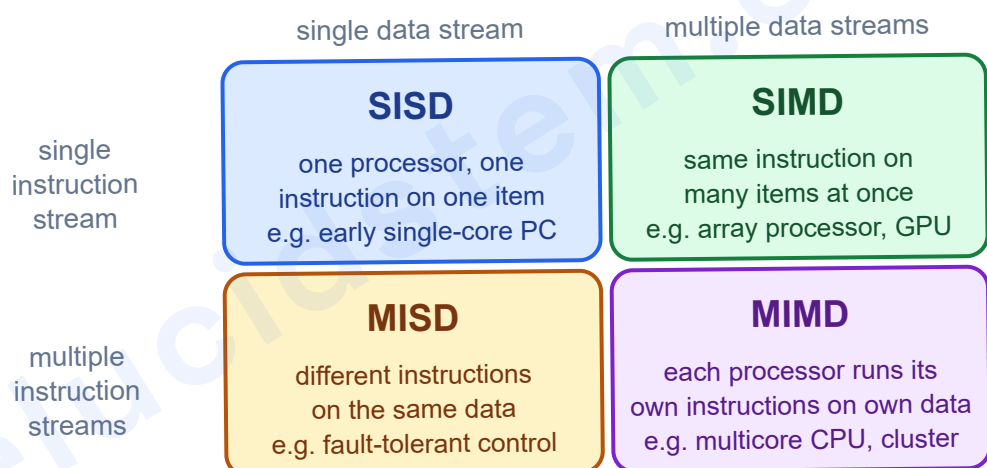


Fig. 1: Flynn's classification, by number of instruction streams (rows) and data streams (columns)

Worked example: Matching the architecture to the task

A program must add 10 to each of 1000 values in an array, all at the same time. Which architecture suits this best, and why?

1. The same instruction (add 10) is needed for every item.
2. There are many data items, so there are multiple data streams.
3. One instruction stream, many data streams: SIMD.
4. Each processing unit applies the add to its own element in the same clock cycle.

Answer: SIMD: one instruction applied to many data items at once

Name it, then explain it.

- ✓ **Expand the acronym:** single or multiple instruction, single or multiple data.
- ✓ **Say what happens:** how many processors, and whether they run the same or different instructions on the same or different data.
- ✓ **Massively parallel:** very many processors, simultaneously, different parts of the same program, connected by a network.
- ✓ **Examples:** GPU or array processor for SIMD; multicore processor for MIMD.

Watch out!

Do not confuse SIMD with MIMD. In SIMD every unit executes the **same** instruction in lock-step; in MIMD each processor can be running a **completely different** program.

A massively parallel computer is not the same as a multicore PC. The mark points are a **very large number** of processors, working **at the same time** on **different parts of the same program**, communicating through a **network**.

Doubling the number of processors does not halve the time unless the **whole** task can be split and the processors never need to communicate.

3 Virtual machines

Syllabus 15.1

A **virtual machine** (VM) is an **emulation of a computer system in software**. The **virtual machine software** (a **hypervisor**) runs on the real hardware, usually on top of a **host operating system**, and presents emulated hardware to one or more **guest operating systems**, each of which behaves as if it had a whole computer to itself. Guest instructions are passed to, or translated for, the real hardware by the virtual machine software. VMs are used to run **several operating systems on one computer**, to run **legacy software**, to **test** software safely and to **share servers** between many customers. The main **benefits** are **isolation** (a crash or a virus in one guest does not affect the host or the others) and **saving hardware**; the main **limitations** are **reduced performance** and the need for a **powerful host** with plenty of memory.

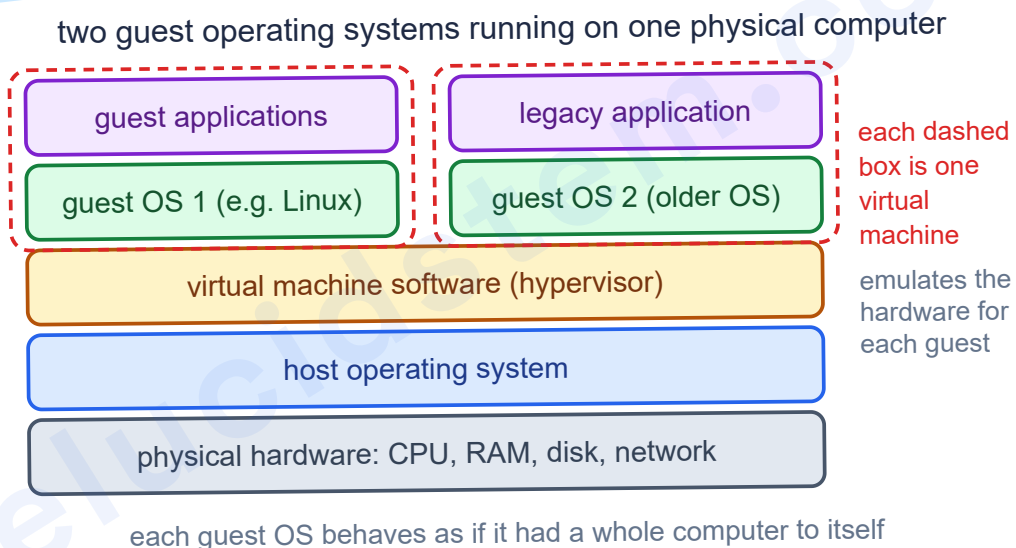


Fig. 1: the virtual machine software sits between the guests and the host, giving each guest operating system its own emulated hardware

Worked example: Explaining a role in context

A company's payroll program only runs on an operating system that is no longer supported, and the old computer is failing. Explain how a virtual machine could help.

1. Virtual machine software is installed on a new computer with a modern host operating system.
2. A virtual machine is created and the old operating system is installed as a guest.
3. The payroll program is installed and run in the guest, which behaves like the old computer.
4. The company can keep using the program while the old hardware is retired.

Answer: Run the old OS as a guest on new hardware, and run the legacy program inside it

Concept, role, consequence.

- ✓ **Concept:** software that emulates the hardware of a computer; guest OS runs on the host.
- ✓ **Roles:** give concrete examples: several OSs, legacy software, testing, malware analysis, server consolidation.
- ✓ **Benefits:** isolation, less hardware, easy backup and snapshots.
- ✓ **Limitations:** slower because of the overhead, needs a powerful host, some hardware cannot be emulated.

Watch out!

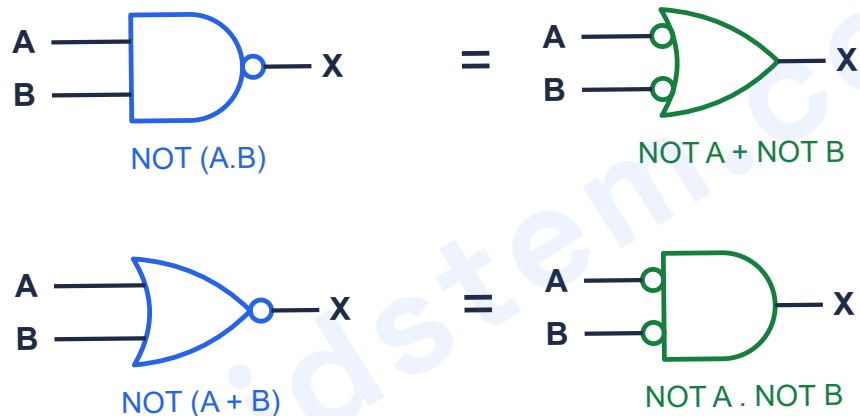
A virtual machine is **not** extra hardware and it is not the same as virtual memory. It is **software** that emulates a whole computer. "It is slower" earns little on its own. Say **why**: the virtual machine software has to intercept and translate the guest's instructions, which adds an overhead.

4 Boolean algebra

Syllabus 15.2

In Boolean algebra $A.B$ means A AND B, $A + B$ means A OR B and a **bar** over a term means NOT, so $\bar{A}$ is NOT A. The **laws of Boolean algebra** (identity, null, idempotent, inverse, commutative, associative, distributive, absorption) let an expression be rewritten without changing its truth table. **De Morgan's laws** say $\overline{A.B} = \bar{A} + \bar{B}$ and $\overline{A + B} = \bar{A}.\bar{B}$: **break the bar and change the operator**. Simplifying an expression means fewer gates, so a circuit that is **cheaper, smaller, faster and uses less power**. A simplification can always be checked by comparing the **truth tables** of the original and the result.

De Morgan's laws as gate equivalences



break the bar, change the operator: a bubble on each input swaps AND and OR

Fig. 1: a NAND gate does the same job as an OR gate with both inputs inverted, and a NOR gate the same as an AND gate with both inputs inverted

Worked example: De Morgan, then absorb

Simplify $X = \overline{A.B + A + C}$.

$$1. X = \overline{A.B + A + C}$$

$$2. = (\overline{A + B}) + \overline{A.C} \text{ De Morgan on each bracket}$$

$$3. = A + \overline{B} + \overline{A.C} \text{ double complement: the double bar on A cancels}$$

$$4. = A + \overline{A.C} + \overline{B} \text{ commutative: reorder}$$

$$5. = A + \overline{C} + \overline{B} \text{ redundancy: } A + \text{NOT } A.Y = A + Y$$

$$6. = A + \overline{B} + \overline{C} \text{ reorder the answer}$$

Answer: $X = A + \overline{B} + \overline{C}$

Show every law.

- ✓ **Notation:** $A.B$ is AND, $A + B$ is OR, a bar is NOT; keep bars clearly over the right letters.
- ✓ **De Morgan:** break the bar and change the operator, one bracket at a time.
- ✓ **Look for patterns:** $X + X.Y = X$, $X + \text{NOT } X.Y = X + Y$, $X + \text{NOT } X = 1$, $X.\text{NOT } X = 0$.
- ✓ **Check:** compare a few rows of the truth table of your answer with the original.

Watch out!

In Boolean algebra $A + A = A$, not $2A$, and $A.A = A$, not A squared. There are only two values, 0 and 1.

The commonest error is to break the bar but forget to change the operator. $A + B$ is **not** $A + \bar{B}$: try $A = 1$, $B = 0$, which gives 0 on the left and 1 on the right.

An answer with no working rarely earns full marks. Show each law you use, and do not combine several laws in one line.

5 Karnaugh maps

Syllabus 15.2

A **Karnaugh map** (K-map) is a grid with one cell for each row of the truth table. The rows and columns are labelled in **Gray code order** (00, 01, 11, 10), so cells that are next to each other, including across the edges, differ in **exactly one input**. Each 1 is circled in a **group of 1, 2, 4 or 8** neighbouring cells forming a rectangle; groups must be **as large as possible** and **as few as possible**, may **overlap** and may **wrap round** the edges. Each group gives one product term containing only the inputs that **do not change** inside it, and the terms are ORed together to give a **sum of products**. The benefit: a **systematic, visual** method that gives the **simplest** expression with less risk of error than Boolean algebra.

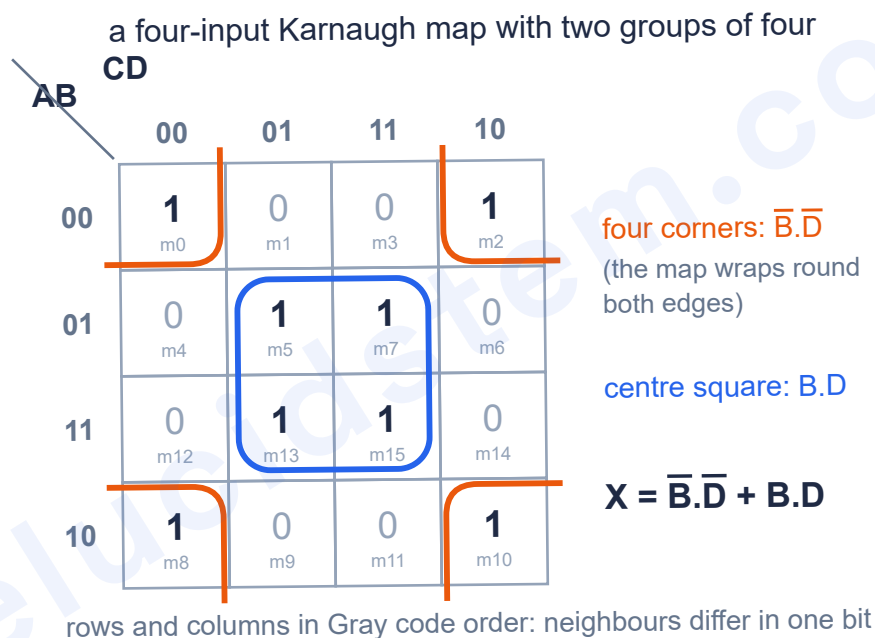


Fig. 1: a four-input K-map. The four corner cells are neighbours because the map wraps round, giving $\bar{B}\bar{D}$; the centre square gives $B.D$

Worked example: A three-input map

Find the simplest expression for the K-map below.

A \ BC 00 01 11 10

0	0	1	1	0
1	1	1	1	0

1. The whole column $BC = 01$ and the whole column $BC = 11$ are 1s; together they make a square of four. In these cells $C = 1$ throughout, while A and B both change, so the group gives C .

2. The 1 in row $A = 1$, column 00 is not yet covered. Group it with its neighbour in column 01 (overlapping the first group), making a pair.
3. In that pair $A = 1$ and $B = 0$ throughout, while C changes: the term is $A \cdot \overline{B}$.
4. OR the terms together.

Answer: $X = C + A \cdot \overline{B}$

Map, group, read off.

- ✓ **Layout:** label rows and columns 00, 01, 11, 10 and fill the 1s from the truth table.
- ✓ **Groups:** 1, 2, 4 or 8 cells, rectangles only, as large and as few as possible, wrapping and overlapping allowed.
- ✓ **Terms:** keep only the inputs that stay the same within each group.
- ✓ **Answer:** OR the terms; one mark is often for each correct group or term.

Watch out!

Labelling the columns 00, 01, 10, 11 (binary order) is the most common mistake: then neighbouring cells 01 and 10 differ in two inputs, and every group in that part of the map is wrong.

Do not stop at the first groups you see. A pair inside a possible group of four gives a term with an extra letter, which loses the simplification mark.

Every group must be checked for the inputs that stay the same. If an input is 0 in some cells of a group and 1 in others, it must not appear in that term.

6 Half adders, full adders and flip-flops

Syllabus 15.2

A **half adder** adds two bits A and B , giving a **sum** $S = A \text{ XOR } B$ and a **carry** $C = A \text{ AND } B$. A **full adder** also adds a **carry in**, so it can be chained: $S = A \text{ XOR } B \text{ XOR } C_{in}$ and $C_{out} = A.B + C_{in}.(A \text{ XOR } B)$; it can be built from **two half adders and an OR gate**. A **flip-flop** is a circuit whose outputs are **fed back** into its inputs, so it has two stable states and **stores one bit**. An **SR flip-flop** is set by $S = 1$, reset by $R = 1$, holds its value when $S = R = 0$, and must not have $S = R = 1$. A **JK flip-flop** changes only on a **clock pulse** and removes the invalid state: $J = K = 1$ makes it **toggle**. Flip-flops are the **data storage elements** of registers and static RAM.

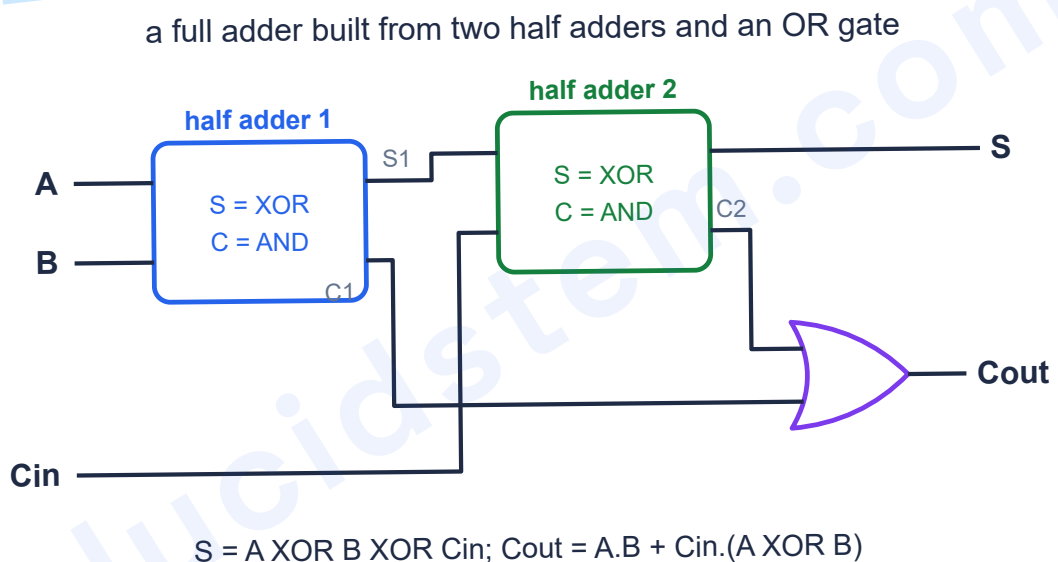


Fig. 1: a full adder. The first half adder adds A and B ; the second adds the carry in to their sum; either half adder can produce the carry out

Worked example: A chain of four full adders

A 4-bit ripple-carry adder adds 0110 and 0111. Give the sum and carry out of each full adder, starting from the least significant bit.

1. Bit 0: $0 + 1 + \text{carry in } 0 = 1$, so $S = 1$, $C_{out} = 0$.
2. Bit 1: $1 + 1 + 0 = 10$, so $S = 0$, $C_{out} = 1$.
3. Bit 2: $1 + 1 + 1 = 11$, so $S = 1$, $C_{out} = 1$.
4. Bit 3: $0 + 0 + 1 = 1$, so $S = 1$, $C_{out} = 0$: no overflow.

Answer: 1101 ($6 + 7 = 13$)

Table first, then explain.

- ✓ **Adders:** S is XOR of the inputs; the carry is 1 when at least two inputs are 1.
- ✓ **Circuits:** a full adder is two half adders plus an OR gate on the two carries.
- ✓ **SR:** set, reset, hold (no change), and say why $S = R = 1$ is invalid.
- ✓ **JK:** clocked; $J = K = 1$ toggles; stores one bit, used in registers and SRAM.

Watch out!

Do not write the full adder carry as $A.B$ only: a carry in of 1 plus one other 1 also produces a carry. $C_{out} = 1$ for rows 011, 101, 110 and 111.

Saying $S = R = 1$ "gives $Q = 1$ " loses the mark. For NOR gates both outputs go to 0, which breaks the rule that Q and $\bar{Q}$ are opposite, so the state is invalid.

"A flip-flop stores data" is too vague. Say it stores **one bit**, and name a use: registers, static RAM or cache.