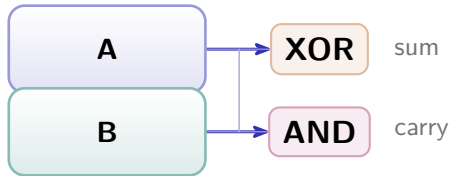


Hardware and Virtual Machines

A-Level Computer Science ■ Topic 15

A-Level Computer Science



What we will cover

- 1 RISC vs CISC & pipelining
- 2 Flynn's taxonomy & parallelism
- 3 Virtual machines
- 4 Boolean algebra & K-maps
- 5 Adders & flip-flops
- 6 Recap



Two styles of CPU design

CISC

many, often complex instructions
variable length – decoding is intricate
many instructions touch memory
more work per instruction
e.g. Intel x86

RISC

few, simple instructions
fixed length – fast to decode
only *load* and *store* touch memory
each instruction is quick and predictable
e.g. ARM, RISC-V

The comparison that the exam wants

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

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

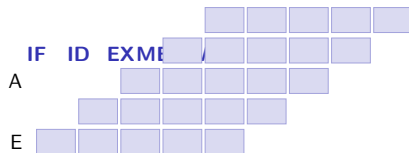
Pipelining – an assembly line for instructions

A **pipeline** processes instructions in **overlapping stages**:

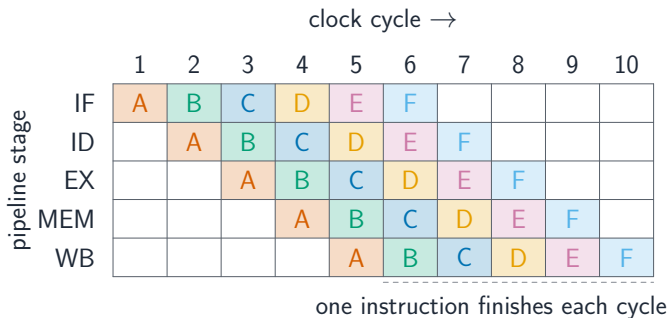
Fetch → Decode → Execute → Memory →
Write back

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 **hazard** stalls it: a *data* hazard (result not ready) or a *control* hazard (a branch makes the next address unknown).



Pipelining overlaps the stages of six instructions



Each instruction still takes the same time; what changes is that one **finishes** every cycle instead of every four. A branch empties the pipeline and the gain is lost until it refills.



Flynn's taxonomy

Flynn's taxonomy sorts computers by instruction streams $\times$ data streams.

SISD

one instruction, one data stream –
a traditional single core

SIMD

one instruction on *many* data
items – GPUs, vector units

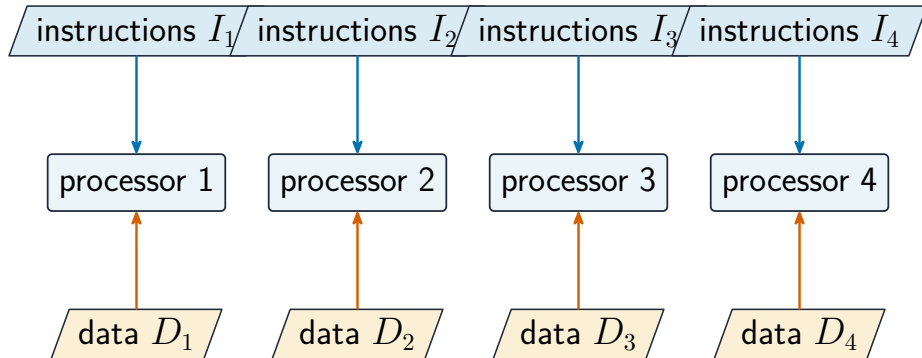
MISD

several operations on the same
data – rare, mostly theoretical

MIMD

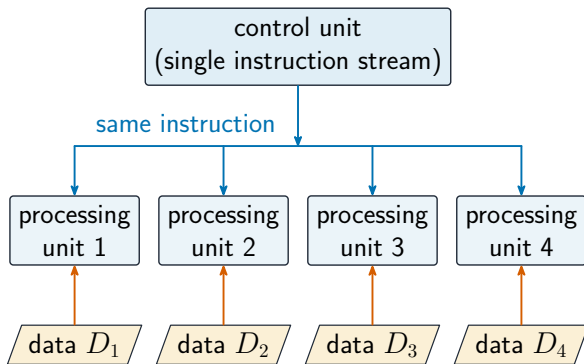
many processors, different instruc-
tions on different data – multi-
core, clusters

MIMD: each processor runs its own instructio – one



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

MIMD: each processor runs its own instructio – two

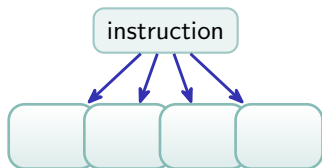


SIMD: many processors run the same instruction on different data

SIMD vs MIMD, in pictures

SIMD

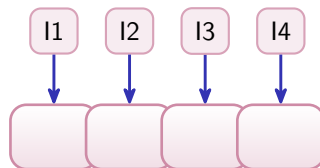
one instruction, many data



a graphics card: thousands of cores, same op
on many pixels

MIMD

many instructions, many data



multi-core CPUs and clusters

Massively parallel computers

A **massively parallel** system uses **thousands of processors** on a fast network.

- each has its own memory – **distributed memory**
- they exchange data by messages
- it is MIMD
- needs specially-written software (MPI, CUDA)

Suits climate simulation, large **machine learning** training, astrophysics. The largest **supercomputers** are massively parallel.



Two kinds of virtual machine

A **virtual machine** is a **software emulation of a whole computer** – the software inside sees a CPU, memory and disks that look real.

system VM

runs a complete OS
a **hypervisor** creates and manages VMs
uses: different OSes on one machine; server consolidation; **sandboxing** ; snapshots

process VM

runs one program in portable **bytecode**
JVM, CLR, CPython
write once, run anywhere;
runtime safety; **JIT**
cost: an extra layer



Laws you must be able to use

Boolean algebra simplifies expressions. + is OR, · is AND, an overbar is NOT.

Law	Form
identity	$A + 0 = A, \quad A \cdot 1 = A$
null	$A + 1 = 1, \quad A \cdot 0 = 0$
idempotent	$A + A = A$
inverse	$A + \overline{A} = 1, \quad A \cdot \overline{A} = 0$
De Morgan	$(A + B)' = A' \cdot B', \quad (A \cdot B)' = A' + B'$
absorption	$A + AB = A$

Example: $Z = AB + A\overline{B} = A(B + \overline{B}) = A$. Fewer terms $\Rightarrow$ fewer gates.

Boolean algebra, and where it runs

Truth tables

Truth tables list every input combination with its output – the definitive way to prove two expressions equivalent.

De Morgan's laws

De Morgan's laws : $\overline{A \cdot B} = \overline{A} + \overline{B}$ and $\overline{A + B} = \overline{A} \cdot \overline{B}$ – break the bar and swap the operator.

The ALU

The **ALU** is where these gates end up: it performs the arithmetic and logic the instruction set exposes.

Just-in-time compilation

Just-in-time compilation compiles bytecode to native code *while* the program runs, so hot paths get compiled and the rest is interpreted.

Simplify with the laws, then check with a truth table. The check costs four rows and catches every algebra slip.

Karnaugh maps – group the 1s

A **Karnaugh map** simplifies by **grouping adjacent 1s**.

- rows and columns in **Gray code** (00, 01, 11, 10)
- groups are rectangles whose sides are 1, 2, 4, 8
- the map **wraps around** its edges
- a group of 2 drops one variable; of 4 drops two

1s in $\bar{A}B$ and AB

The two 1s share the $B = 1$ column. A changes, so it disappears:

$$X = B$$

Two gates fewer than $\bar{A}B + AB$.



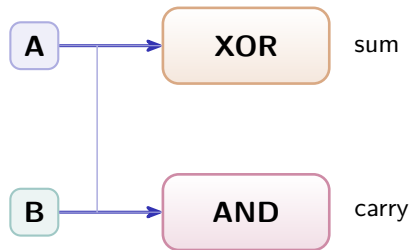
Half adder – two bits, no carry-in

A **half adder** adds two bits A and B :

$$S = A \text{ XOR } B, \quad C = A \text{ AND } B$$

It ignores any carry-in – hence “half”.

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

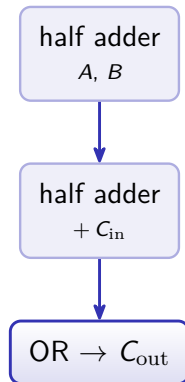


Full adder – three bits, including carry-in

A **full adder** adds A , B and a carry-in:

$$S = A \text{ XOR } B \text{ XOR } C_{\text{in}}$$

Built from **two half adders plus an OR gate**.
Chain them (each carry-out feeds the next carry-in) to make a multi-bit *ripple-carry* adder.



Flip-flops – one bit of memory

A **flip-flop** is a **bistable** circuit: two stable states, so it **remembers** one bit. Building block of registers and SRAM.

SR

SR

S sets $Q = 1$; R resets $Q = 0$

$S = 0, R = 0$ holds

$S = 1, R = 1$ is **invalid**

two cross-coupled NOR gates

JK

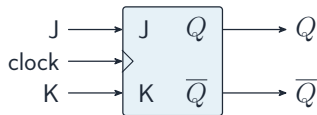
JK

$J = 1, K = 1$ is a **toggle**

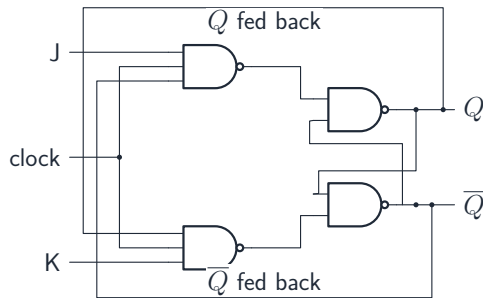
ideal for **counters**

usually *clocked* – inputs act on a clock edge

Flip-flops – one bit of memory, in detail



block symbol



build from NAND gates

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

Pipelining



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

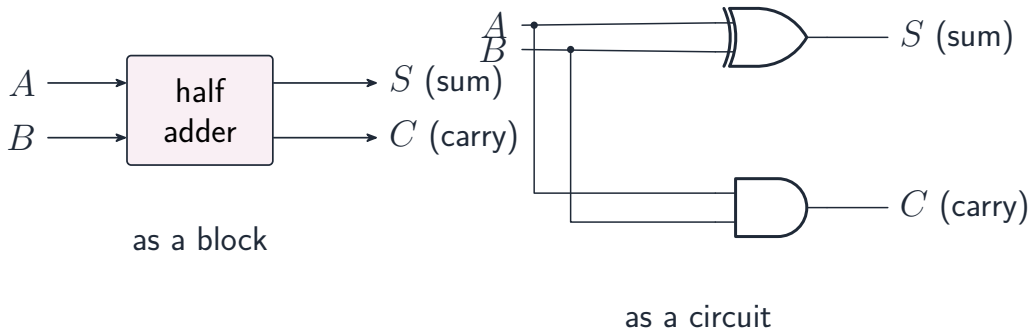
Flynn's taxonomy, continued



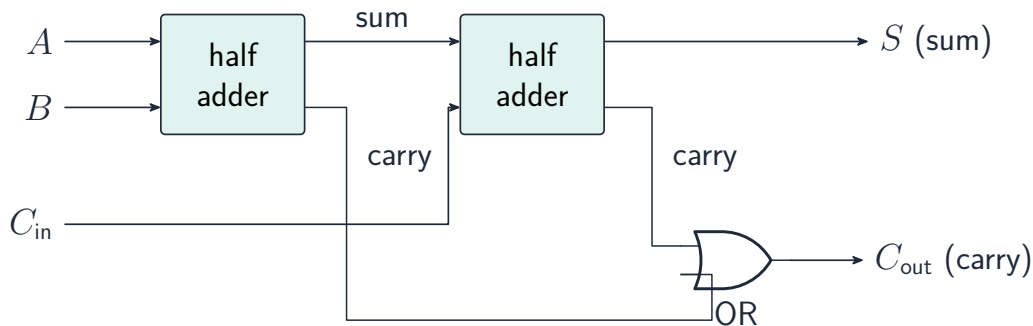
A graphics card: its GPU runs the same instruction on many data items at once (SIMD)

Half adder and full adder

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



A full adder is built from two



A half adder cannot take a carry in, which is why one alone can only add the least significant bit. Chain full adders and the carry ripples along the word.

RISC vs CISC processors

CISC has many complex instructions; RISC has few simple ones

CISC

many, complex instructions
variable length
does more per instruction

RISC

few, simple instructions
fixed length
each instruction is fast

RISC vs CISC processors, continued



A motherboard links the CPU, memory and other parts together

Massively parallel computers, continued



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



Topic 15 – key takeaways

1 RISC

simple, fixed-length, pipeline-friendly

2 Flynn

SISD, SIMD, MISD, MIMD

3 Logic

De Morgan, K-maps, half vs full adder

4 Memory

flip-flop stores one bit; VMs emulate a computer

Check yourself

- 1 Why does RISC suit pipelining better than CISC?
- 2 A GPU running one instruction on many pixels – SISD, SIMD or MIMD?
- 3 What extra input does a full adder have that a half adder does not?
- 4 What does $J = 1, K = 1$ do on a JK flip-flop?

Answers 1. fixed-length, simple stages of equal time 2. SIMD 3. carry-in 4. toggle