

Hardware and Virtual Machines

A-Level Computer Science

RISC vs CISC processors

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

CISC
many, complex instructions
variable length
does more per instruction

RISC
few, simple instructions
fixed length
each instruction is fast

CISC has many complex instructions; RISC has few simple ones



A motherboard links the CPU, memory and other parts together

Image: ASUS, Product image (www.newegg.com)

CISC

A **CISC** 复杂指令集 (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

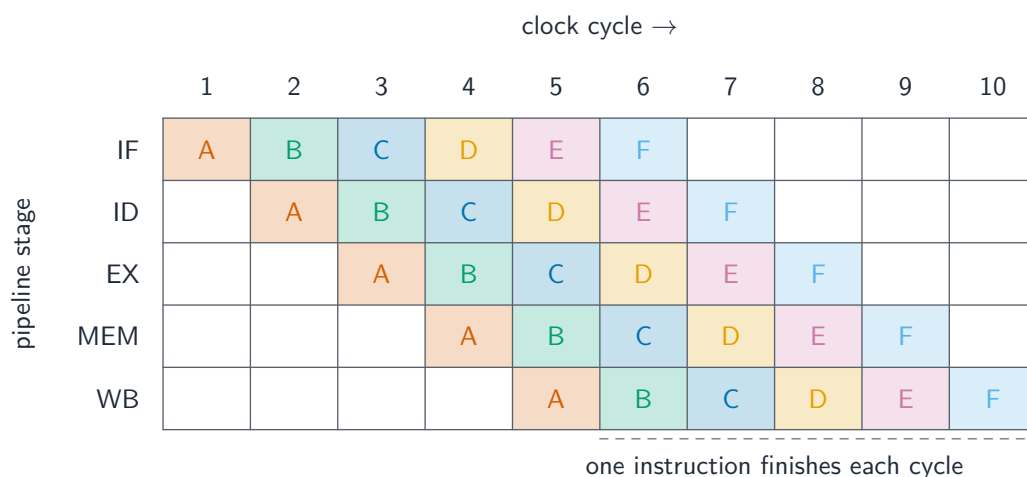
A **RISC** 精简指令集 (Reduced Instruction Set Computers) has a **small set of simple** instructions, each doing one basic operation, all of **fixed length** (fast to decode). Only **load** and **store** touch memory; everything else is **register** 寄存器-to-register. Programs are longer but each instruction 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.

Pipelining

A **pipeline** 流水线 processes instructions in **overlapping stages**, like an assembly line: Fetch → Decode → Execute (in the **ALU** 算术逻辑单元) → 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** 冒险—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.

A processor running this fast gives off a lot of heat, so a **heat-sink** 散热器 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.



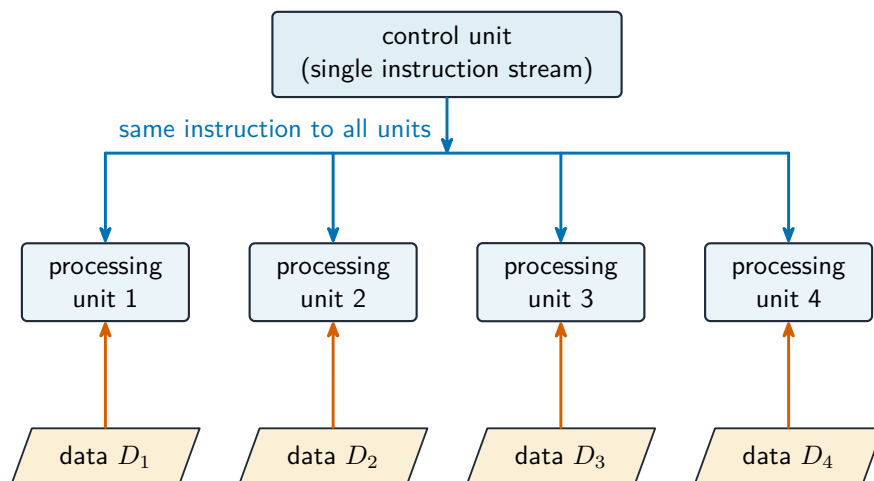
A CPU heat-sink and fan carry heat away from the processor

Image: osman gucel, CC BY 2.0 (commons.wikimedia.org)

Flynn's taxonomy

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

- **SISD** —one instruction, one data stream (a traditional single core).
- **SIMD** 单指令多数据—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** 多指令多数据—many processors run **different instructions on different data** (multi-core CPUs, clusters). The most general.



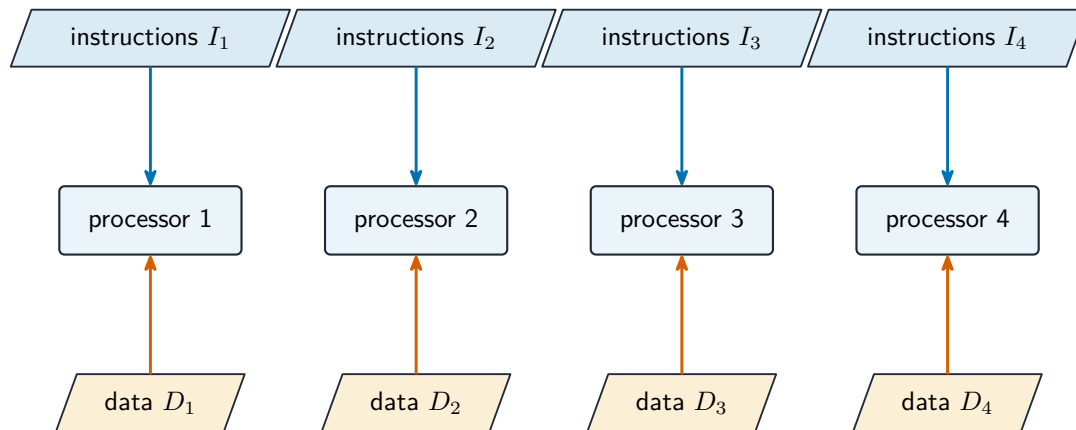
SIMD: many processors run the same instruction on different data

A **graphics card** 显卡 (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)

Image: NVIDIA, Product image (www.newegg.com)



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

Massively parallel computers

A **massively parallel** 大规模并行 system uses **thousands of processors** on a fast network, each with its own memory (**distributed memory** 分布式内存), exchanging data by messages. It is MIMD, needs specially-written software (MPI, CUDA), and suits climate simulation, large **machine learning** 机器学习 training, and astrophysics. The largest **supercomputers** 超级计算机 are massively parallel.

The processors live in tall **server** 服务器 racks, often filling a whole room (a **data centre** 数据中心), 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

Image: Carl Lender from Sunrise, USA, CC BY 2.0 (commons.wikimedia.org)

Virtual machines

A **virtual machine** 虚拟机 (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** 虚拟机监控器 creates and manages VMs, each booting its own guest OS. Uses: run different OSes on one machine; server consolidation; **sandboxing** 沙箱 (risky software runs isolated); snapshots.
- a **process (language) VM** runs one program in portable **bytecode** 字节码—the JVM (Java), the CLR (.NET), CPython. Benefits: portability (“write once, run anywhere”), runtime safety checks, and **just-in-time compilation** 即时编译 for near-native speed. The cost is an extra layer and needing the VM installed.

Boolean algebra

Boolean algebra 布尔代数 simplifies **Boolean** 布尔 expressions, which can equally be described by **truth tables** 真值表. Symbols: $+$ for OR, $\cdot$ 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 $A + \bar{A} = 1$, $A \cdot \bar{A} = 0$.
- **De Morgan’s laws** 德摩根定律: $(A + B)' = A' \cdot B'$; $(A \cdot B)' = A' + B'$ —negate the whole, swap AND/OR, negate each operand.
- **absorption** 吸收律: $A + AB = A$.

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

Karnaugh maps

A **Karnaugh map** 卡诺图 (K-map) simplifies a Boolean expression by **grouping adjacent 1s** from a truth table. Columns and rows use **Gray code** 格雷码 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 $\bar{A}B$ 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

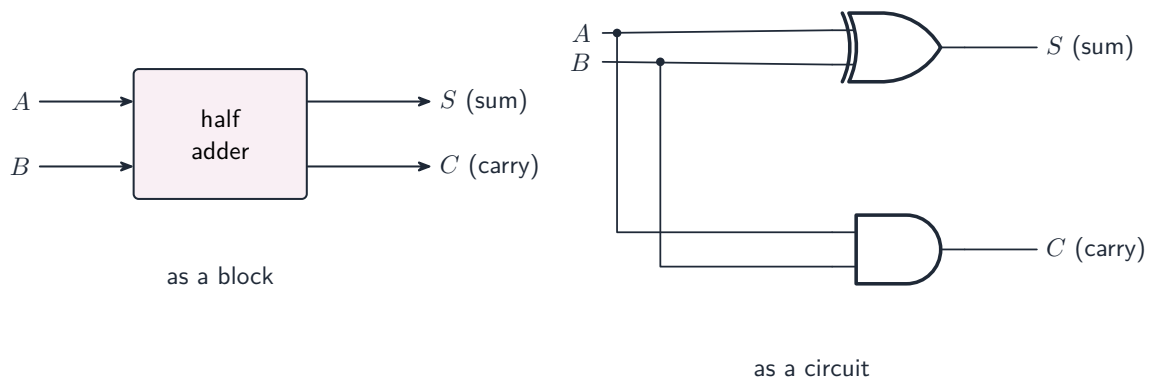
$X = B$. Compare that with the sum of products read straight off the table, $\overline{A}B + AB$: the same circuit, two gates fewer. Two rules do most of the work - make each group **as large as possible** (a group of 2 drops one variable, 4 drops two, 8 drops three), and remember the map **wraps around** its edges, so the leftmost and rightmost columns are adjacent. That wrap is the grouping most candidates miss.

Half adder and full adder

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

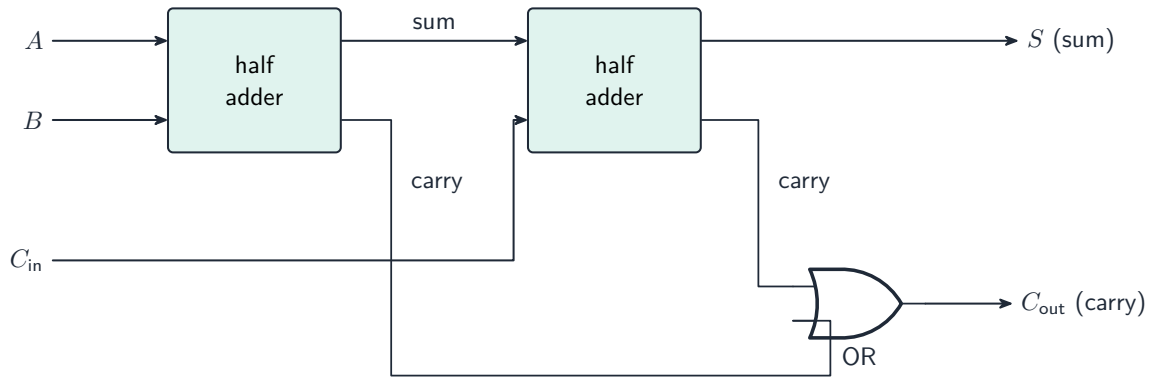
A	B	S	C
0	0	0	0
0	1	1	0
1	0	1	0
1	1	0	1

So $S = A \text{ XOR } B$ and $C = A \text{ AND } B$. 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** 全加器 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

Flip-flops

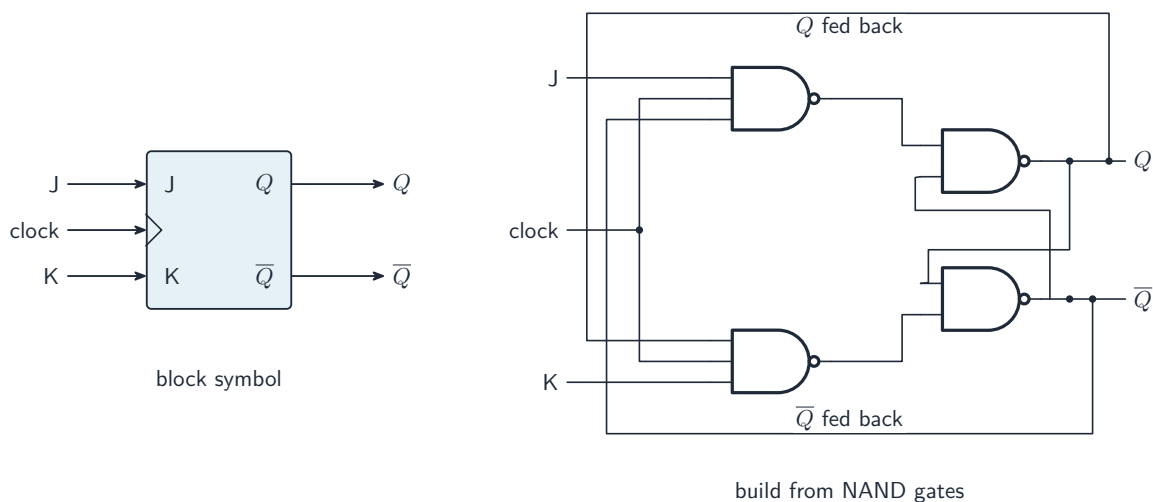
A **flip-flop** 触发器 is a **bistable** 双稳态 circuit —two stable states (0 and 1) —that **remembers** its state. It stores one bit and is the basic element of registers and SRAM.

SR flip-flop

An **SR flip-flop** SR 触发器 has inputs **S** (set) and **R** (reset) and outputs Q and $\bar{Q}$. 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 NAND gates.

JK flip-flop

A **JK flip-flop** JK 触发器 improves on it by using the previously-invalid 1,1 input as a **toggle** 翻转 (the output flips). This makes it ideal for building **counters** 计数器 (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** 静态 RAM cells.

Exam tips

- Compare **RISC vs CISC** (simple, fast, uniform instructions vs complex ones) and why RISC suits pipelining.
- Simplify logic with **Boolean algebra / Karnaugh maps** —group the 1s in powers of two.
- Explain a **half adder vs full adder** (the full adder handles a carry-in) and what a flip-flop stores.
- Place a machine in **Flynn's taxonomy** (SISD, SIMD, MISD, MIMD).