Processor Fundamentals

A-Level Computer Science

Von Neumann architecture

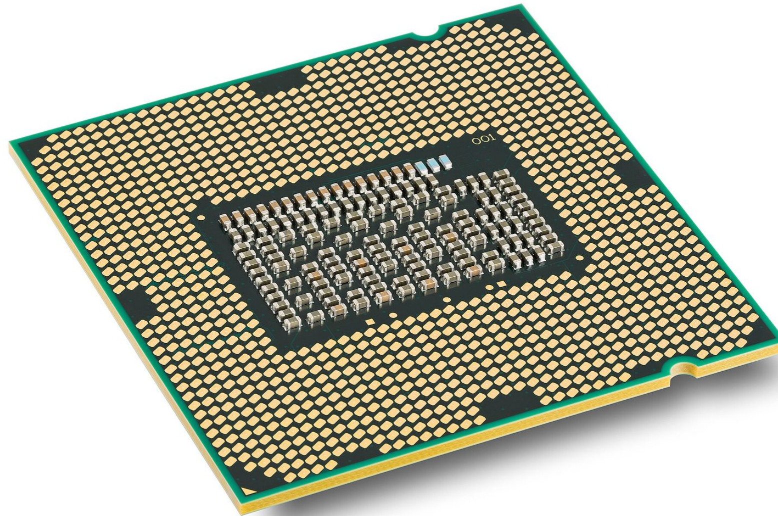
The **Von Neumann architecture** 冯 · 诺依曼体系结构 underlies almost every general-purpose computer:

- a **single memory** holds both **program instructions** and **data** (the **stored program** 存储程序 idea).
- a **processor** 处理器 (CPU) fetches instructions from memory and runs them one at a time.
- instructions run **in order** unless a branch changes the flow.

The stored-program idea is what makes a computer flexible: change the program and you change what it does, with no rewiring.

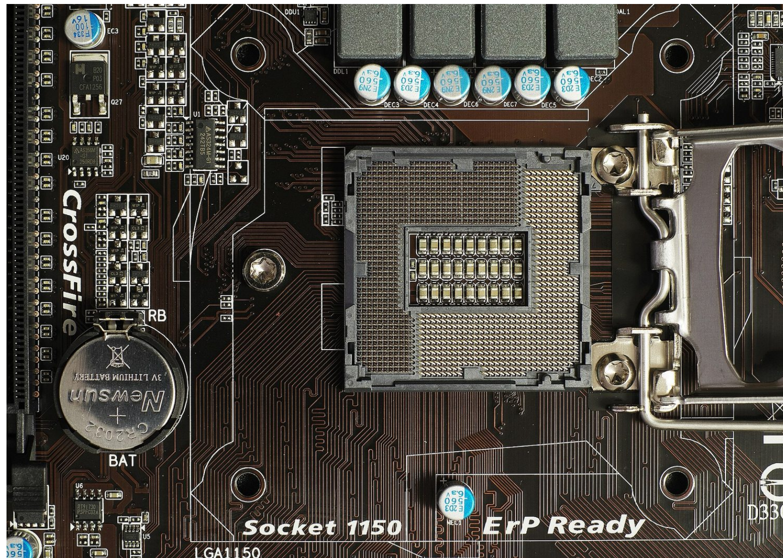
The CPU's main parts

All of these parts sit inside one small chip. The diagram later in this section shows how they connect; the photo below shows the real thing.



A modern CPU: the whole processor is one small chip (here seen from below, showing the contacts)

Image: Eric Gaba (Sting - fr:Sting), CC BY-SA 3.0 (commons.wikimedia.org)



The matching CPU socket on the motherboard: the chip's contacts press onto these pins

Image: Smial (talk), FAL (commons.wikimedia.org)

Arithmetic and Logic Unit (ALU)

The **ALU** 算术逻辑单元 does the arithmetic (add, subtract, ···) and logic (AND, OR, comparisons). It takes operands from **registers** 寄存器 and puts results back in a register.

Control Unit (CU)

The **control unit** 控制单元 **decodes** each instruction and sends the **control signals** to carry it out —opening data paths, telling the ALU what to do, and controlling memory reads and writes.

System clock

The clock sends a steady stream of pulses that keep the CPU in step. Each instruction takes a fixed number of cycles, and the **clock speed** 时钟频率 (e.g. 3.8 GHz) is one factor in performance.

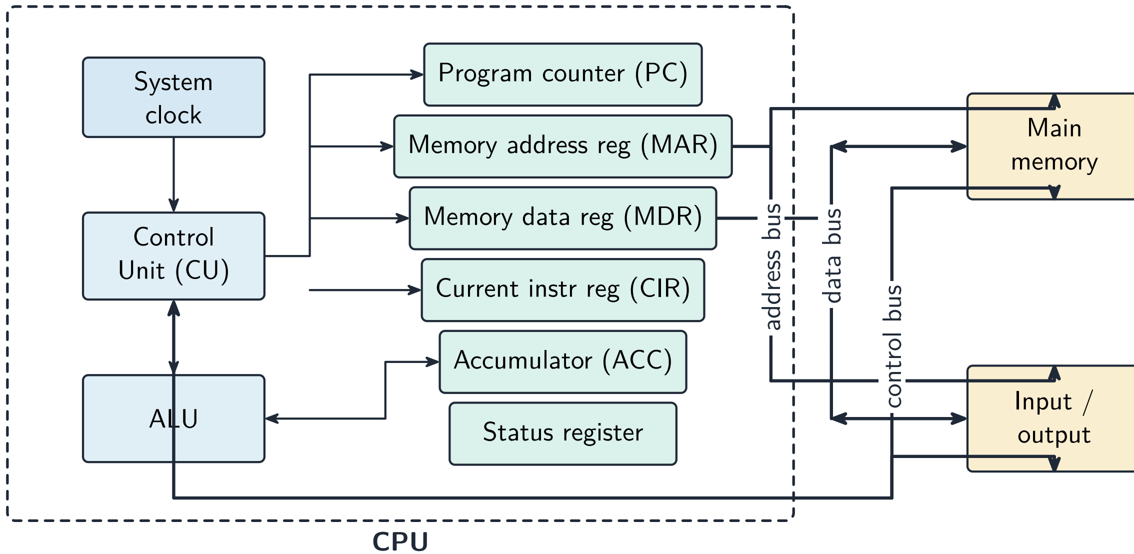
Registers

Registers are tiny, very fast stores inside the CPU. **Special-purpose** ones each have a fixed job in the cycle:

- **Program Counter** 程序计数器 (PC) —the address of the **next** instruction.
- **Memory Address Register** 内存地址寄存器 (MAR) —the address being read or written.
- **Memory Data Register** 内存数据寄存器 (MDR) —the data going to or from memory.
- **Current Instruction Register** 当前指令寄存器 (CIR) —the instruction being decoded.
- **Accumulator** 累加器 (ACC) —the value the ALU is working on.

- **Status Register** 状态寄存器—holds **flags** 标志 (carry, zero, negative, overflow) used by branches.
- **Index Register** 变址寄存器—an offset used in indexed addressing.

General-purpose registers 通用寄存器 are used by the programmer for temporary values during a calculation.



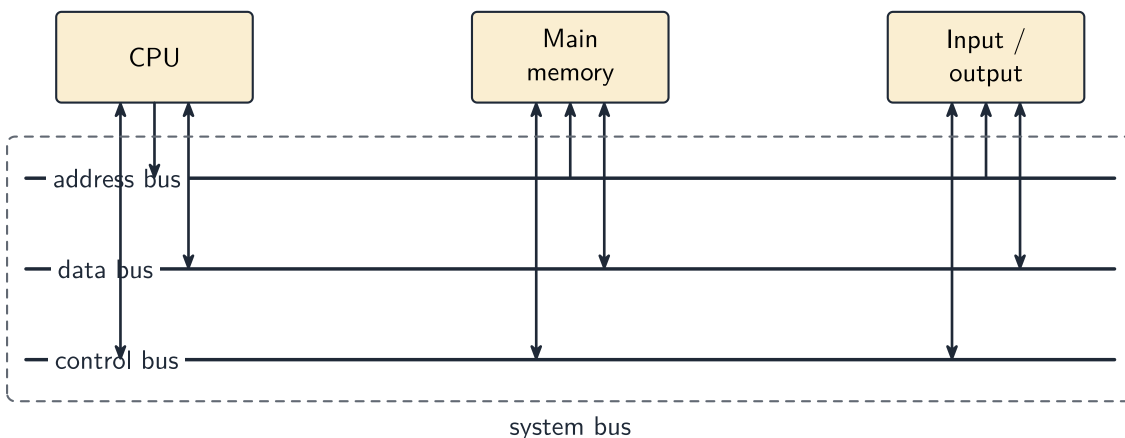
The Von Neumann CPU: registers, control unit and ALU linked by buses

Buses

Three internal **buses** 总线 (sets of parallel wires) connect the parts:

- **address bus** 地址总线—carries the memory address. **One-way** (CPU → memory).
- **data bus** 数据总线—carries the data. **Two-way**.
- **control bus** 控制总线—carries control signals (read, write, interrupt). **Two-way**.

An n-bit address bus can reach 2^n memory locations. The data-bus width sets how many bits move per access (often the word size).



The three system buses connecting the CPU, memory and input/output

What affects performance

- **clock speed** —more cycles per second.
- **number of cores** 核心—a multi-core CPU runs several threads at once.
- **word size** 字长—a 64-bit CPU handles 64-bit chunks per cycle and can address far more memory than a 32-bit one.
- **amount of RAM** 随机存取存储器—more RAM holds more of the working set; too little forces the OS to **page** 分页 to disk.
- **cache** 高速缓存 size —more cache cuts average memory access time.
- **secondary storage** 辅助存储器 type —an SSD loads programs far faster than an HDD.
- **bus width and speed** —wider/faster buses move data more quickly.

Match the specs to the workload: a quad-core beats a dual-core on parallel work, but higher per-core speed wins on single-threaded work.

Ports

A **port** 端口 is a physical socket for connecting a **peripheral** 外围设备:

- **USB** —general-purpose (keyboards, drives, phones).
- **HDMI** —digital video and audio to a screen.
- **Ethernet (RJ-45)** —wired LAN. Audio jacks —headphones/microphone.

Different ports use different signals, so an HDMI cable will not fit a USB socket. USB-C is unusual in carrying video, data and power.

Fetch-Execute cycle

The CPU repeats the **fetch-execute cycle** 取指-执行周期, one run per machine instruction.

Fetch

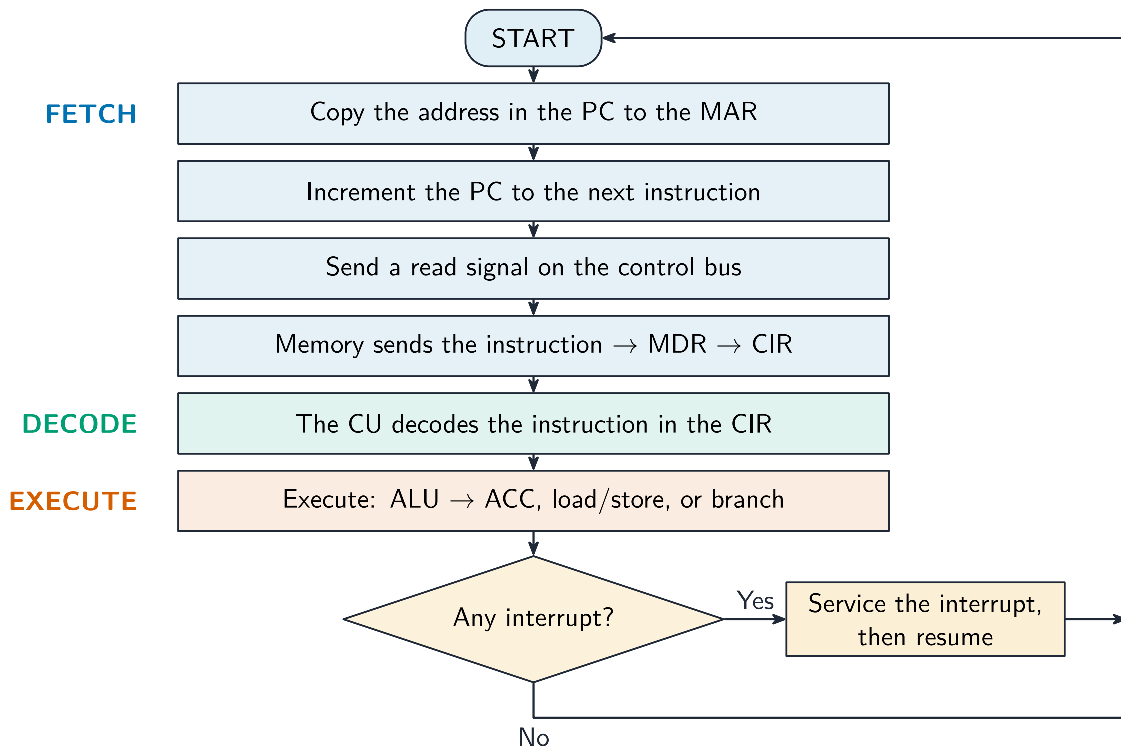
1. the PC's address is copied to the MAR.
2. the PC is **incremented** to point to the next instruction.
3. a **read** signal goes over the control bus.
4. memory puts the instruction on the data bus.
5. it is copied into the MDR, then into the CIR.

Decode

The CU decodes the instruction in the CIR —what operation, and which operands or addresses.

Execute

The CU carries it out: arithmetic/logic goes to the ALU (result to the ACC); a load/store moves data between memory and a register; a branch changes the PC. Then the cycle repeats.



The fetch-execute cycle, with a check for interrupts each time

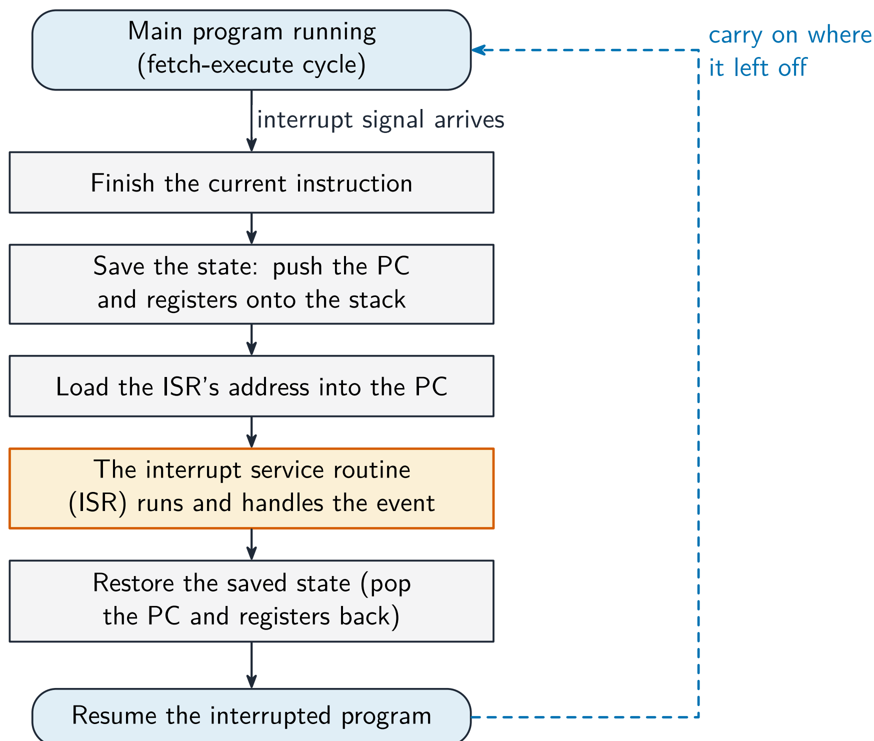
Interrupts

An **interrupt** 中断 is a signal that **pauses** the normal cycle so the CPU can handle an urgent event (a key press, a packet arriving, a hardware fault, division by zero, the OS timer).

Handling one:

1. finish the current instruction.
2. **save the state** (PC and registers).
3. load the address of the **interrupt service routine** 中断服务程序 (ISR) into the PC and run it.
4. the ISR handles the event.
5. **restore** the saved state and carry on.

Interrupts let the system respond promptly without the CPU constantly checking devices, and are how the OS multitasks.



How an interrupt fits into the fetch-execute cycle

Assembly language and machine code

The CPU actually runs **machine code** 机器码—bit patterns, specific to one architecture. **Assembly language** 汇编语言 is a readable form, with one instruction per machine instruction, written using **mnemonics** 助记符 like `LDD`, `ADD`, `JMP`. An **assembler** 汇编器 translates it to machine code.

Two-pass assembler

A two-pass assembler reads the source twice:

- **pass 1** builds a **symbol table** 符号表: each time a **label** 标签 (like `LOOP:`) appears, record its address; no code yet.
- **pass 2** generates code: translate each instruction, and when one refers to a label (like `JMP LOOP`), look up its address in the symbol table.

Two passes handle **forward references** 前向引用 (a jump to a label defined later).

Example instruction set

Cambridge uses a small generic set: data movement (`LDD`, `LDM`, `LDI`, `LDX`, `STO`, `MOV`), arithmetic (`ADD`, `SUB`, `INC`, `DEC`), logic/bit (`AND`, `OR`, `XOR`, `LSL`, `LSR`), compare and branch (`CMP`, `JMP`, `JPE`, `JPN`), I/O (`IN`, `OUT`), and `END`. The exact mnemonics are given in the paper's reference table.

Addressing modes

The **addressing mode** 寻址方式 says how the CPU finds the operand:

- **immediate addressing** 立即寻址—the operand is the value in the instruction. `LDM #10` loads 10.
- **direct addressing** 直接寻址—the instruction holds an address; the operand is the value there. `LDD 200`.
- **indirect addressing** 间接寻址—the instruction holds an address that holds **another** address, which is the data. `LDI 200`.
- **indexed addressing** 变址寻址—effective address is `address + index register`; used for arrays. `LDX 100` with IR = 5 reads address 105.
- **relative addressing** 相对寻址—the address is relative to the PC.

Tracing an assembly program

To **trace** it: make a table with columns for the PC, ACC, index register, each variable and any flags. Step through the instructions, updating the table after each; follow branches when they change the PC; stop at `END`. A common pattern is a loop over an array using indexed addressing.

Binary shifts

A **logical shift** 逻辑移位 moves all the bits left or right by some places, filling new positions with 0.

- **left shift by 1** (`LSL #1`) —bits move left, a 0 enters on the right; for an unsigned number this is **× 2**.
- **right shift by 1** (`LSR #1`) —bits move right, a 0 enters on the left; for an unsigned number this is **integer ÷ 2**.

Shifting by n places multiplies or divides by 2^n. Example: `00001011` (11) `LSL #1` → `00010110` (22).

An **arithmetic right shift** keeps the sign bit so a negative signed number stays negative.

Bit manipulation for monitoring/control

Embedded devices often use one **bit** 位 of a register per signal (e.g. bit n = LED n). Using a **mask** 掩码:

- **set** bit n: `R = R OR` a mask with bit n set.
- **clear** bit n: `R = R AND` a mask with bit n clear and the rest set.
- **toggle** bit n: `R = R XOR` a mask with bit n set.
- **test** bit n: `R AND` the mask, then check if the result is non-zero.

Bit manipulation is fast, uses little memory, and lets one byte hold up to 8 on/off states.