

## Computer Systems Fundamentals

Curated computer science study track (not a degree, no licensure claim). Original explanations and practice only, following the reviewed course handout.

## From a bit to a running program

- 3.1-3.2 the alphabet (bits, encoding, gates); 3.3-3.4 the machine; 3.5-3.6 the machinery around it.
- Every conversion is verifiable by script; every abstraction is a contract with the layer below.

## Binary and data encoding: three readings of the same bits

- 1 **Ranges Unsigned** 0 to  $2^n - 1$ ; **two's complement**  $-2^{n-1}$  to  $2^{n-1} - 1$  — flip all bits, add 1 (computes  $2^n - b$ ).
- 2 **Worked**  $-73$  in 8 bits:  $73 = 01001001$ , flip  $10110110$ ,  $+1 = 10110111$ ;  $96 + 64 = 160$  overflows signed 8-bit (max 127)  $\rightarrow$  reads  $-96$ .
- 3 **Float/text IEEE-754** sign-exponent-fraction:  $0.1$  has no finite binary form, so  $0.1 + 0.2 = 0.300000000000000004$  (rounding); ASCII 7 bits, Unicode code points, UTF-8 variable width.

# Bit-layout boxes

## the same 32 bits, read three ways

unsigned 32-bit:  $0 \dots 2^{32}-1$

|                                           |
|-------------------------------------------|
| bit 31 ... bit 0: all value bits, no sign |
|-------------------------------------------|

two's complement 32-bit:  $-2^{31} \dots 2^{31}-1$

|                        |                                      |
|------------------------|--------------------------------------|
| sign bit: 0 = +, 1 = - | value bits: the MSB weighs $-2^{31}$ |
|------------------------|--------------------------------------|

IEEE-754 float32: sign · biased exponent · fraction

|   |                    |                                |
|---|--------------------|--------------------------------|
| ± | exp - 127 (8 bits) | 1.fraction, hidden 1 (23 bits) |
|---|--------------------|--------------------------------|

**-73 in 8 bits: 01001001 → flip 10110110 → +1 = 10110111**

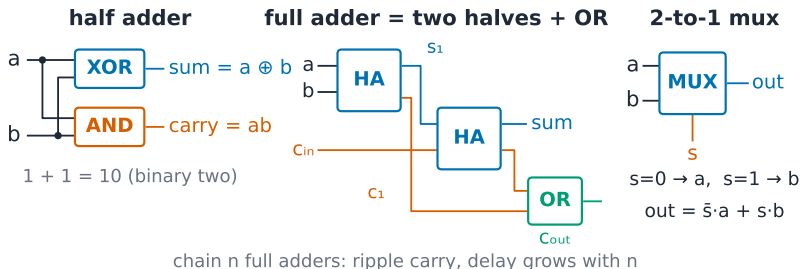
96 + 64 overflows signed 8-bit to  $-96 \cdot 0.1 + 0.2 = 0.300000000000000004$

The same 32 bits read three ways: **unsigned**, **two's complement**, IEEE float.

## Logic gates: from truth tables to adders

- 1 **Half adder**  $\text{Sum} = a \text{ XOR } b$ ,  $\text{carry} = ab$  —  $1+1 = 10$  in two gates; **full adder** = two halves + OR, chained into ripple-carry (delay grows with  $n$ ).
- 2 **Mux**  $\text{Out} = \bar{s} \cdot a + s \cdot b$ :  $s=0$  passes  $a$ ,  $s=1$  passes  $b$ ; a 4-to-1 is three 2-to-1 stages.
- 3 **Algebra**  $x + x = x$ ,  $x + \bar{x} = 1$  — fewer gates, less power, less delay; gate delay limits clock speed.

# Gate schematic: half adder, full adder, mux



**NAND is universal: every gate above can be built from NAND alone**

boolean identities minimise circuits: fewer gates, less power, less delay

XOR-and-AND make a **half adder**; two make a **full adder**; a mux picks a lane.

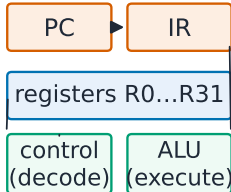
## Processor and memory: the loop and the pyramid

- 1 **Loop** Fetch ( $PC \rightarrow \text{memory} \rightarrow IR$ ), decode (control signals), execute (ALU/memory),  $PC \rightarrow PC + 4$ .
- 2 **Pyramid** Registers  $\sim 1 \text{ ns} \rightarrow L1/L2/L3 \rightarrow \text{DRAM} \sim 100 \text{ ns} \rightarrow \text{SSD}$ : fast memory is expensive per byte, big memory is slow.
- 3 **Layout** One **cache** line carries 64 adjacent bytes — row-major access beats transposed loops at the same op count.

# CPU and memory block diagram

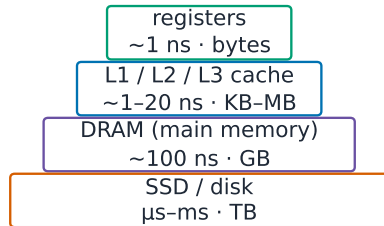
## the fetch-decode-execute loop

fetch: PC  $\rightarrow$  memory  $\rightarrow$  IR



each cycle: PC  $\rightarrow$  PC + 4

## the memory hierarchy



↑ smaller, faster, costlier per byte

**locality makes it work: one cache line brings 64 adjacent bytes**

The fetch-decode-execute loop around the register file, with the memory pyramid alongside.

## Machine instructions: registers, immediates, memory

- 1 **Shapes** add rd, rs1, rs2 (registers); addi (immediate); lw/sw with base+offset; beq/bne compare and branch.
- 2 **Loop** addi R1, R1, -1; bne R1, R0, loop — two instructions per iteration, exactly the C while pair.
- 3 **Indexing**  $a[i] = a[i] + 1 \rightarrow$  lw · addi · sw: the index scales into the offset; a pointer is a base register.

# Instruction and register trace

## a loop, unrolled into machine steps

| instruction                                      | action                                   | R1 |
|--------------------------------------------------|------------------------------------------|----|
| addi R1, R0, 3                                   | $R1 \leftarrow 0 + 3$                    | 3  |
| loop: addi R1, R1, -1                            | $R1 \leftarrow R1 - 1$                   | 2  |
| bne R1, R0, loop                                 | $R1 \neq 0 \rightarrow \text{jump}$      | 2  |
| addi R1, R1, -1                                  | $R1 \leftarrow R1 - 1$                   | 1  |
| bne R1, R0, loop                                 | $R1 \neq 0 \rightarrow \text{jump}$      | 1  |
| addi R1, R1, -1                                  | $R1 \leftarrow R1 - 1$                   | 0  |
| bne R1, R0, loop                                 | $R1 = 0 \rightarrow \text{fall through}$ | 0  |
| each pass runs addi + bne: the loop is this pair |                                          |    |

**a[i] += 1 → lw, addi, sw**

### addressing modes

#### register

add R3, R1, R2

#### immediate

addi R1, R1, 1

#### base+offset

lw R1, 0(R2)

memory: lw / sw only

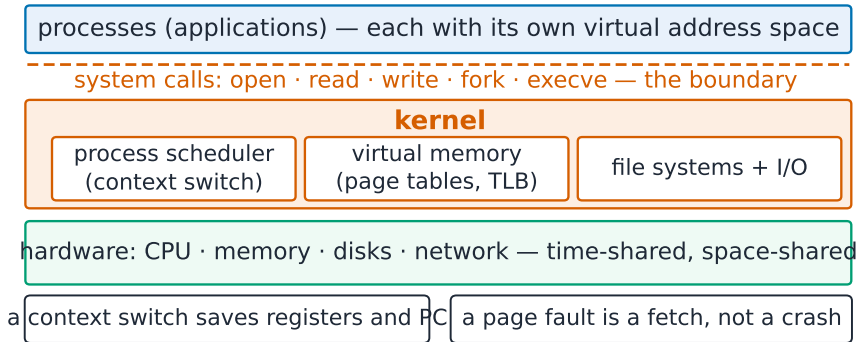
Each line: fetch, execute, the register delta — a loop unrolled.

# The operating-system interface: services, not hardware

- 1 **Process** Program + state; a **context switch** saves registers, PC, memory mappings, open-file table — exactly what the next instruction needs.
- 2 **Virtual memory** Page tables + TLB give each **process** its own addresses: isolation, over-commit, fixed link addresses.
- 3 **Calls** open · read · write · fork · execve cross into the kernel; a **page fault** is demand paging working.

# The OS layer diagram

## the OS shares the machine



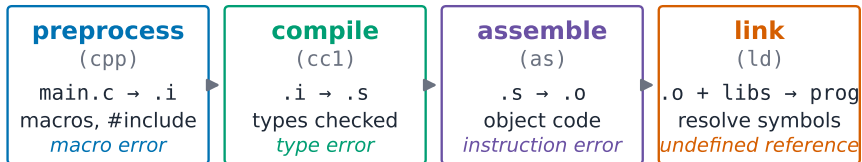
System calls cross the boundary; the kernel shares CPU, memory and devices.

## Compilation and execution: four stages, four errors

- 1 **Pipeline** `main.c`  $\rightarrow$  (cpp) `main.i`  $\rightarrow$  (cc1) `main.s`  $\rightarrow$  (as) `main.o`  $\rightarrow$  (ld) `prog`: every intermediate can be kept and inspected.
- 2 **Errors** Macro errors (cpp), type errors (cc1), instruction errors (as), undefined reference (ld) — each stage its own class.
- 3 **Trade** Compiled: translate once run many; interpreted (Python): `source`  $\rightarrow$  `bytecode`  $\rightarrow$  VM, read and execute every run.

# The toolchain pipeline

**source to process: four stages, four error classes**



**the loader maps prog's segments and sets PC to main**

compiled: translate once, run many · interpreted: read and execute via a VM

keep every intermediate file: the pipeline is the debugger's map

**undefined reference is the LINKER speaking, not the compiler**

Preprocess, compile, assemble, link — each stage its own file format and error class.

## Check yourself

- $-73$  as 8-bit **two's complement**...? (10110111.)
- Sum and carry of  $1+1$  in a **half adder**...? (0 and 1.)
- The PC advances by...? (4.)
- Undefined reference comes from...? (The **linker**.)