

Topic 03

University Computer Science

What this course is, and how to use it

This is the third course in University Computer Science: **computer systems fundamentals** – from a bit to a running program: how numbers live in wires, how gates decide, how a processor steps, and how the OS and the toolchain deliver your program to silicon.

Scope limit, stated once: a study course, not a degree; every program fragment is original and self-contained.

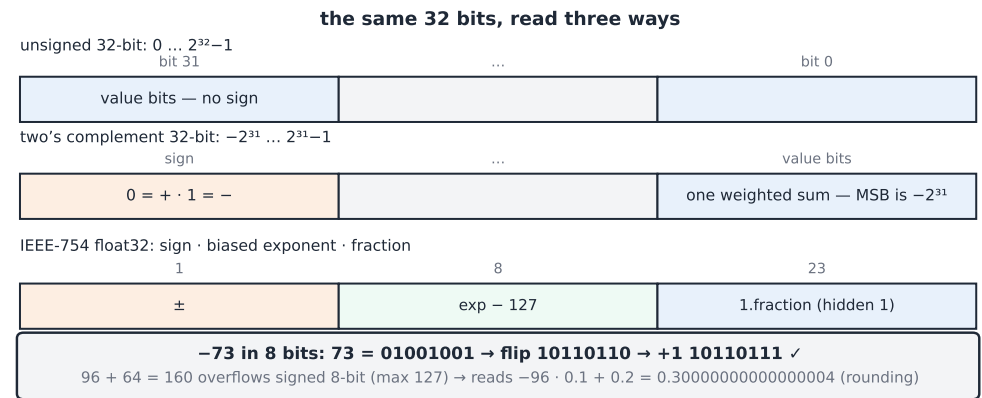
- Units 3.1–3.2 are the machine’s alphabet: bits, encoding, gates.
- Units 3.3–3.4 are the machine itself: processor, memory, instructions.
- Units 3.5–3.6 are the machinery around it: OS services, the compilation pipeline.

The systems habit: **every abstraction is a contract with the layer below** – each unit names the contract, and every conversion is something you can verify by script.

Binary and data encoding

Why it matters. Everything below the source code is bits: knowing how numbers, fractions and text are encoded explains every surprise from overflowing counters to $0.1 + 0.2 \neq 0.3$.

The idea, with a picture. Base-2 positional notation; n -bit **unsigned** range 0 to $2^n - 1$; **two’s complement** for signed: flip all bits, add 1 – range -2^{n-1} to $2^{n-1} - 1$ ✓. The carry out of the top bit is always **discarded**; **overflow** happens only when two operands of the same sign give a result of the other sign. **IEEE-754** float: sign, exponent (biased), fraction ✓; text: **ASCII** (7 bits, 128 codes) and **Unicode** (code points, UTF-8 variable width).



The same 32 bits read three ways: unsigned, two’s complement, and an IEEE float – layout is the meaning

The derivation (why two’s complement wins). Two’s complement makes subtraction an addition: $a - b = a + (-b)$ with one adder. The proof is modular: $-b \equiv 2^n - b \pmod{2^n}$, and flip-plus-one computes exactly $2^n - b$ ✓. One circuit for + and –; that economy is why it became universal.

Worked example 1 – conversions and overflow. 8-bit two’s complement: $-73?$ $73 = 01001001_2$; flip $\rightarrow 10110110$; add 1 $\rightarrow 10110111 = -73$ ✓. Add $96 + 64 = 160$: unsigned fine (10100000_2), but signed 8-bit max is 127 – overflow, and the stored pattern reads -96 ✓.

Worked example 2 – float surprises. 0.1 in binary is infinite ($0.000110011\dots$); IEEE rounds – so $0.1 + 0.2$ yields 0.30000000000000004 : a **rounding** error, not a language bug ✓. Also: because the exponent scales the gap, adjacent floats are NOT evenly spaced – dense near zero, sparse far away.

Common mistakes. Assuming a bit pattern has one value: 10110111 is 183 unsigned but -73 signed (the bits are the bits; the type reads them). Expecting float equality after arithmetic. Confusing character code with number (the character 1 stores the code 49, not 1).

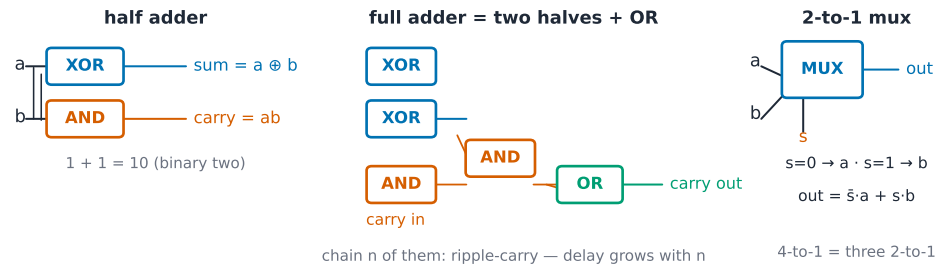
Summary. Unsigned $0..2^n - 1$; two’s complement flip-and-add-1, one adder for + and –; discard the sign carry; IEEE: sign-exponent-fraction with rounding; ASCII 7-bit, Unicode code points, UTF-8 variable width; $0.1 + 0.2 \neq 0.3$ by rounding.

Logic gates and digital building blocks

Why it matters. Gates are the physics-to-logic boundary: every program is ANDs, ORs and NOTs in the end, and boolean algebra is the optimiser.

The idea, with a picture. AND, OR, NOT; combine into **NAND** (universal –

every gate can be built from it ✓). Building blocks: the **half adder** (XOR = sum, AND = carry), **full adder** (two half adders + OR, carries chained into a ripple-carry adder), the **multiplexer** (select line picks one input) ✓.



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

The derivation (why boolean algebra optimises circuits). Identities ($x+x=x$, $x \cdot x=x$, $x+\bar{x}=1$) let two expressions with one truth table collapse into one circuit – e.g. the mux expression $x = \bar{s}a + sb$ is already minimal, while a naive OR of the four rows where the output is 1 is not ✓. Fewer gates, less power, less delay.

Worked example 1 – one-bit addition. Sum $s = a \oplus b$, carry $c = ab$: rows $1+0 \rightarrow s=1, c=0$; $1+1 \rightarrow s=0, c=1$ (binary 10) ✓ – two gates, correct arithmetic. Chain four for $0101 + 0011$.

Worked example 2 – a 2-to-1 mux in gates. Output $= \bar{s}a + sb$: when $s=0$, a passes; when $s=1$, b passes ✓. Then $4\text{-to-1} = \text{three } 2\text{-to-1}$ stages. A 32-to-1 is the same idea, five layers deep.

Common mistakes. Writing XOR as OR (the $1+1$ row). Forgetting the carry chain's propagation delay (ripple adds gate-delays per bit). Believing gates are instantaneous (delay is what limits clock speed).

Summary. NAND universal; half adder XOR+AND; full adder chains; ripple-carry delay grows with width; mux $\bar{s}a + sb$; boolean identities minimise circuits.

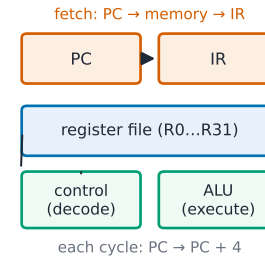
Processor and memory overview

Why it matters. The CPU is a loop and the memory is a pyramid: both pictures explain speed, cost and why the cache decides performance more than the clock.

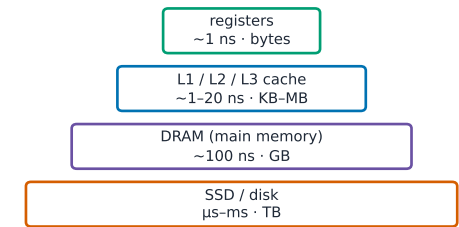
The idea, with a picture. Fetch-decode-execute cycle – – : fetch (PC → memory → IR), decode (instruction → control signals), execute (ALU / memory / PC update), PC → PC + 4 ✓. The memory hierarchy: registers (ns) →

L1/L2/L3 cache → DRAM (~100 ns) → SSD → disk – each layer bigger and slower ✓.

the fetch-decode-execute loop



the memory hierarchy



smaller, faster, costlier ↑
bigger, slower, cheaper ↓

locality makes the pyramid work — one cache line carries 64 adjacent bytes

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

The derivation (why the hierarchy exists). Speed vs cost vs size is a physical trade-off: fast memory is expensive per byte, big memory is slow. The hierarchy exploits **locality** – programs touch a small working set repeatedly (temporal) and nearby addresses (spatial) – so a small fast cache holds most references ✓.

Worked example 1 – tracing three instructions. LOAD R1, 0(R2) / ADD R3, R1, R1 / STORE R3, 0(R2): each step registers change (R1, then R3, then memory) ✓; the PC advances by 4 after each instruction; a branch writes a new address into the PC.

Worked example 2 – why data layout matters. An array sum is limited by memory, not arithmetic: reading one cache line brings 64 adjacent bytes, so row-major access is fast; a transposed loop is many times slower at the same operation count ✓. The hierarchy, not the clock, decides.

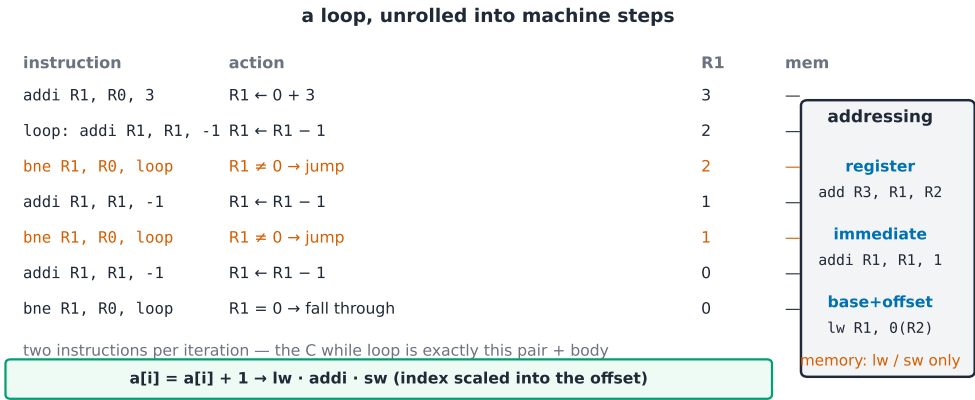
Common mistakes. Confusing IR (instruction register) and PC. Reading DRAM latency in cycles as if constant (cache hits and misses differ by orders of magnitude). Assuming the CPU's view of memory is the physical one (3.5's virtual memory interposes).

Summary. Fetch → decode → execute, PC+4; registers-file-IR-ALU around the loop; hierarchy: registers-cache-DRAM-SSD; locality makes it work; data layout can dominate arithmetic.

Machine instructions

Why it matters. Assembly is the contract every compiler targets and every performance question bottoms out in: registers, immediates and memory moves.

The idea, with a picture. RISC instructions: `add rd, rs1, rs2` (registers only), `addi` (immediate), `lw/sw` (memory, addressing like `0(R2)`), `beq/bne` (compare and jump) ✓. **Addressing modes** : register, immediate, base+offset – memory only via load/store.



Each line: *fetch, execute, the register delta* – a loop unrolled into its machine steps

The derivation (why RISC is load/store only). One instruction shape decodes fast and pipelines well: every memory touch is an explicit `lw/sw`; arithmetic is register-to-register. CISC packs memory access into arithmetic – more compact, harder to pipeline ✓. The trade is instruction count against per-instruction speed.

- Worked example 1 – a loop in machine steps.** `loop: addi R1, R1, -1; bne R1, R0, loop`: subtract, compare-with-zero, jump back – two instructions per iteration; a C `while` loop maps to exactly this pair plus the body ✓.
- Worked example 2 – mapping C to instructions.** `a[i] = a[i] + 1` compiles to `lw` (load), `addi, sw` (store) with the index scaled into the offset ✓. Array indexing is address arithmetic; pointers are base registers; that is the whole secret.

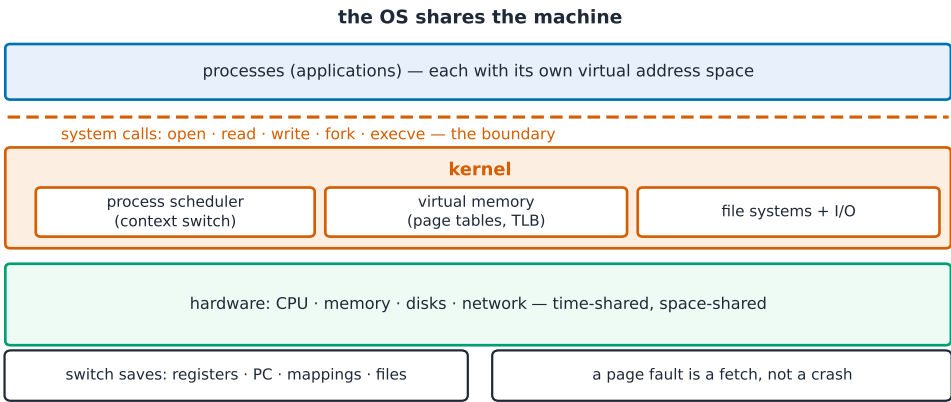
Common mistakes. Reading `0(R2)` as the value 0 (it is the offset 0 on base R2). Forgetting to initialise the loop counter register. Using the index *i* as the byte offset without scaling it by the element width (`4i` for 4-byte words).

Summary. `add/addi/lw/sw/beq`; memory only via load/store base+offset; `while` → decrement-and-branch; array indexing is scaled address arithmetic; `a[i]+1` is `lw-addi-sw`.

The operating-system interface

Why it matters. No program touches hardware: the OS shares the machine safely – processes, virtual memory and files are the services every program consumes.

The idea, with a picture. A **process** = program + execution state (address space, registers, open files); a **context switch** saves and restores that state ✓. **Virtual memory** : each process sees its own addresses; the OS + hardware (TLB, page tables) map virtual → physical. **Files** are named byte streams; system calls (`open, read, write, fork, execve`) are the boundary where your code asks the kernel ✓.



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

The derivation (why virtual memory). Isolation (one process cannot read another's pages), illusion (more memory than physical, via paging to disk), and convenience (programs link to fixed addresses) – three needs one abstraction serves ✓. A **page fault** occurs when an access cannot proceed with the current page mapping or permissions. The OS may load or allocate a valid page; an invalid access may instead terminate the process.

- Worked example 1 – what a context switch preserves.** Registers, PC, memory mappings, open-file table: everything the next instruction of that process needs; what does NOT carry: CPU caches (shared, just cold) ✓. The save area per process is exactly this list.
- Worked example 2 – a system call in C.** `int fd = open(path, O_RDONLY); read(fd, buf, 4096);` – each call traps into the kernel, runs with kernel privileges, returns ✓. The same trio from Python: `open, read` – the interface is the contract, the language wraps it.

Common mistakes. Mixing process (own memory) and thread (shares memory).

Believing malloc touches disk (memory, not files). Reading a page fault as always a crash or always a disk fetch (many valid faults are recoverable, while invalid accesses can fail).

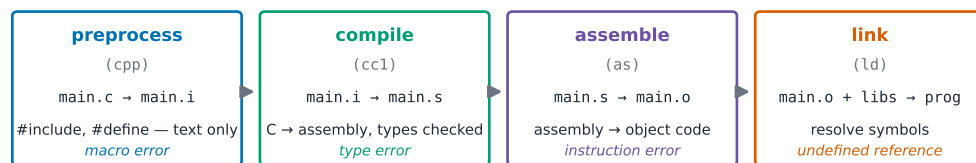
Summary. Process = program + state; switch saves/loads it; virtual → physical via page tables; page faults require OS handling; system calls are the boundary; open/read/write/fork/execve.

Compilation and execution path

Why it matters. Source is not a program: the pipeline from .c to a running process has four stages, each with its own error class – and each stage’s output is inspectable.

The idea, with a picture. Preprocessor (expand #include and #define – text only) → compiler (C → assembly) → assembler (assembly → object code, relocations unresolved) → linker (resolve symbols across objects + libraries → executable) ✓; the loader starts it as a process (3.5).

source to process: four stages, four error classes



the loader starts prog as a process (3.5): map segments, set PC to main
compiled: translate once, run many · interpreted: read and execute via a VM

every intermediate file can be kept and inspected — 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 its own error class

The derivation (why stages separate). Separate compilation: a million-line program recompiles one changed file; libraries distribute as objects/so-files; each error localises to one stage – macro errors, type errors, instruction errors, symbol errors ✓. The pipeline is an engineering economy, not a formality.

Worked example 1 – the four files. main.c → (cpp) main.i → (cc1) main.s → (as) main.o → (ld) prog; the extensions name the stages ✓. Any intermediate can be inspected with the right flag.

Worked example 2 – compiled vs interpreted. C: all four stages before running (fast, machine-specific binary). Python: source is read and executed by an interpreter (CPython compiles to bytecode, the VM runs it) ✓. Compile once-run-many vs read-every-run: the trade in one line.

Common mistakes. Calling every failure a compile error (undefined reference is the LINKER). Forgetting headers are text (macros have no type). Assuming optimisation cannot change float results (reassociation can).

Summary. cpp (text) → cc (types) → as (instructions) → ld (symbols); undefined reference = linker; compiled vs interpreted is when translation happens.