



Foundations of Computer Organization

From switches and gates to instruction sets: logic design, RTL, number systems, IEEE 754 arithmetic, ISA design, RISC-V and the limits of computation

What this handbook covers

- | | |
|--|---|
| 1 Organization, architecture & the abstraction stack | 4 IEEE 754 floating-point arithmetic |
| 2 Gate-level and sequential design | 5 Instruction set architecture & RISC-V |
| 3 RTL design, PLDs & processor performance | 6 Computability and complexity |

Dr. Bikram Paul

School of Computing and Electrical Engineering (SCEE)
Indian Institute of Technology Mandi
bikram@iitmandi.ac.in

Jan–June 2026

Contents

1	Organization, Architecture and the Abstraction Stack	1
1.1	Computer organization versus computer architecture	1
1.1.1	Why study organization and architecture?	2
1.2	A generic modern computing system	2
1.3	Von Neumann and Harvard architectures	2
1.3.1	The stored-program (von Neumann) machine	2
1.3.2	The Harvard machine	3
1.4	The evolution of computers	4
1.5	The role of VLSI	5
1.6	The hierarchy of digital systems	6
1.6.1	Computer architecture in levels: from problem to electrons	7
1.7	The processor in an SoC and the hardware–software interface	7
1.8	Cost, yield and why integration wins	8
	Exercises	9
2	Gate-Level and Sequential Logic Design	10
2.1	The system equation	10
2.1.1	From equation to gates	10
2.1.2	Cost and delay	10
2.1.3	An all-NAND and all-NOR design exercise	11
2.2	The design hierarchy and the design process	12
2.3	Combinational design: the full adder	12
2.3.1	Specification and equations	12
2.3.2	Three implementations	13
2.3.3	The ripple-carry adder	13
2.4	Sequential circuits	14
2.4.1	The D flip-flop and its timing	14
2.4.2	Finite-state machines: Moore and Mealy	15
2.4.3	A Mealy example: the serial adder	15
2.4.4	A second Mealy example: serial multiply-by-three	16
2.5	Synchronous sequential design procedure	17

2.5.1	Flip-flop excitation tables	17
2.5.2	Complete example: synchronous mod-5 counter with JK flip-flops	18
	Exercises	20
3	Register-Transfer Design, Programmable Logic and Processor Performance	21
3.1	Hardware design abstraction	21
3.2	The register-transfer level	22
3.2.1	The multifunction unit	22
3.2.2	Multiplexers as universal function generators	22
3.2.3	A 4-bit magnitude comparator from adders	23
3.2.4	Registers and the sequential data loader	24
3.3	Programmable logic devices	24
3.3.1	Field-programmable gate arrays	25
3.3.2	FPGA-based SoCs: Zynq and the ZedBoard	26
3.4	Processor-level design	27
3.4.1	The processor as a datapath plus a controller	27
3.4.2	Memory addressing	27
3.4.3	A single-bus processor	27
3.4.4	Coprocessors	28
3.5	Quantifying processor performance	28
3.5.1	The performance equation	29
	Exercises	30
4	Floating-Point Representation and Arithmetic	32
4.1	Why floating point?	32
4.1.1	The limits of fixed point	32
4.1.2	Scientific notation	32
4.2	The IEEE 754 standard	33
4.2.1	General format	33
4.2.2	Precision formats	33
4.3	Encoding and decoding	34
4.3.1	Encoding procedure	34
4.4	Special values	35
4.4.1	Subnormal (denormalized) numbers	35
4.4.2	A complete picture: an 8-bit toy format	36
4.5	Floating-point addition and subtraction	36

4.5.1	Algorithm	36
4.6	Floating-point multiplication	38
4.6.1	Algorithm	38
4.7	Floating-point division	39
4.8	Signs, rounding and precision	40
4.8.1	Sign handling	40
4.8.2	Rounding modes	40
4.8.3	Loss of precision	41
4.9	Conversion between integers and floating point	42
4.9.1	Integer to floating point	42
4.9.2	Floating point to integer	42
4.10	Hardware floating-point units	43
4.11	Modern reduced-precision formats	43
	Exercises	44
5	Instruction Set Architecture and RISC-V	46
5.1	What an ISA specifies	46
5.1.1	Elements of an ISA	46
5.2	Where are the operands? 0-, 1-, 2- and 3-address machines	47
5.3	Sequential versus dataflow execution	48
5.4	The stack (PUSH/POP) processor	49
5.4.1	Stack organization	49
5.5	RISC versus CISC	51
5.5.1	Complex and simple instructions	51
5.5.2	The semantic gap	51
5.6	A tour of real ISAs	52
5.6.1	Alpha	52
5.6.2	MIPS	52
5.6.3	ARM (A32)	53
5.6.4	LC-3b and bit steering	53
5.7	RISC-V	53
5.7.1	What is RISC-V?	54
5.7.2	RV32I: the 32-bit integer base	54
5.7.3	Instruction formats	55
5.7.4	Arithmetic and logical instructions	56

5.7.5	Encoding and decoding by hand	56
5.7.6	Execution in a pipeline: a preview	57
5.7.7	Instruction mix and implementations	57
	Exercises	59
6	Computability and Complexity: The Limits of Any Processor	60
6.1	Problems and Turing machines	60
6.2	Decidable (Turing-solvable) problems	61
6.3	Computational complexity	62
6.4	Undecidable (Turing-unsolvable) problems	63
6.4.1	The Halting Problem	63
6.4.2	Other unsolvable problems and reductions	63
6.5	Why this matters to a computer architect	64
	Exercises	65

Organization, Architecture and the Abstraction Stack

A modern smartphone contains a system-on-chip (SoC) with more than fifteen billion transistors, yet a programmer writing $x = a + b$; never thinks about a single one of them. This chapter explains how that is possible. We separate the two viewpoints from which a computer can be studied — *architecture* and *organization* — trace how the computer evolved from mechanical calculators to three-dimensional chip stacks, and build the layered abstraction stack that the rest of the course climbs, one level at a time.

1.1 Computer organization versus computer architecture

The two terms are often used interchangeably in casual speech, but they answer different questions.

Definition 1.1: Computer architecture

The **functional behaviour** of a computer as seen by the machine-language programmer or compiler: the instruction set, the data types and formats, the instruction formats, the addressing modes, the programmer-visible registers, the memory model, and the way the system interacts with external software (exceptions, privilege). The architecture is a *specification*; the **instruction set architecture (ISA)** is its core.

Definition 1.2: Computer organization

The **physical implementation** of an architecture: how the functional units (ALU, register file, control unit, memories) are interconnected, which control signals and data paths carry each instruction, how each component is designed, and how performance is improved by techniques such as pipelining and parallelism. Organization is also called **microarchitecture** when we speak about the processor core.

A useful test is to ask whether a property is *visible to a correctly written program*. Whether the machine has a multiply instruction is architectural: a program either may or may not use `mul`. Whether that multiply is computed by a 4-cycle Booth multiplier or a 1-cycle Wallace tree is organizational: the program produces the same answer either way, only faster or slower.

Key idea

Architecture is the contract; *organization* is one way to honour it. A single architecture (for example RV64GC) is implemented by many organizations — from a tiny in-order core in a microcontroller to a wide out-of-order core in a server.

Table 1.1: Architecture versus organization: classifying design decisions.

Architectural (visible to software)	Organizational (invisible to software)
Number of general-purpose registers (32 in RV32I)	Number of physical registers used for renaming (e.g. 128)
Instruction encoding and length	Number of pipeline stages
Byte ordering (little-endian)	Cache size, associativity, replacement policy
Existence of mul/div (M extension)	Multiplier algorithm (array, Booth, Wallace)
Floating-point format (IEEE 754 binary32)	Width of the FPU datapath, latency of fdiv
Virtual-address width (Sv39 = 39 bits)	TLB size and organization
Memory consistency model (RVWMO)	Store buffer depth, write policy

1.1.1 Why study organization and architecture?

- **To know existing machines.** Every embedded, mobile and server platform you will work on is built from these ideas; reading a datasheet requires the vocabulary.
- **To design efficient systems.** Speed, cost and energy pull in different directions. Quantitative reasoning about trade-offs (Section 3.5) is the core professional skill of a computer architect.
- **To write faster programs.** An algorithm that is cache-friendly or branch-predictor-friendly can run an order of magnitude faster than an equivalent one that is not, on the same hardware.
- **To build new machines.** Embedded controllers, AI accelerators, post-quantum cryptographic co-processors and HPC nodes are all designed by people who understand both sides of the hardware–software interface.

The topics covered in this course are: processor design (ALU, control unit, pipelining), the memory hierarchy (cache, main memory, secondary storage), input/output and communication, and instruction sets (RISC versus CISC).

1.2 A generic modern computing system

Figure 1.1 shows the major blocks found in almost every computer, from a microcontroller to a server. A **processor** (one or more cores) fetches instructions and data through a hierarchy of **caches** from **main memory**. An **interconnect** — historically a shared bus, today usually a network-on-chip or a crossbar — links the processor to memory controllers, to accelerators (GPU, NPU, crypto engine), and to **I/O controllers** that talk to storage, networks and humans.

1.3 Von Neumann and Harvard architectures

1.3.1 The stored-program (von Neumann) machine

In his 1945 *First Draft of a Report on the EDVAC*, John von Neumann described a machine in which **instructions and data share one memory** and are moved over **one path** between memory and the processor. Instructions are executed **sequentially**: a program counter (PC) names the next instruction, which is fetched, decoded and executed before the PC advances. Because instructions are just data, a program can load other programs (a loader, a compiler, an operating system) —

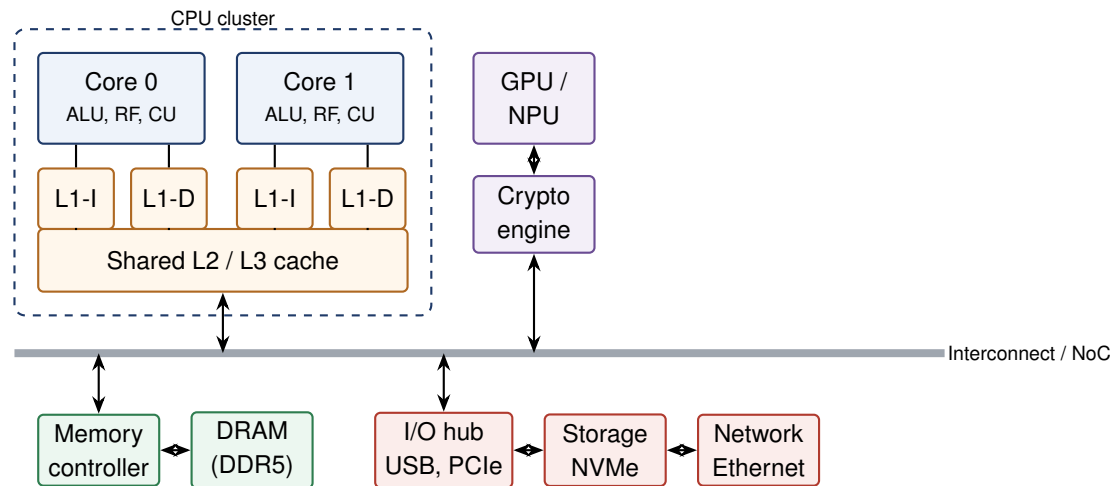


Figure 1.1: A generic modern computing system. On an SoC everything above the DRAM and storage devices sits on a single die.

the single most consequential idea in computing.

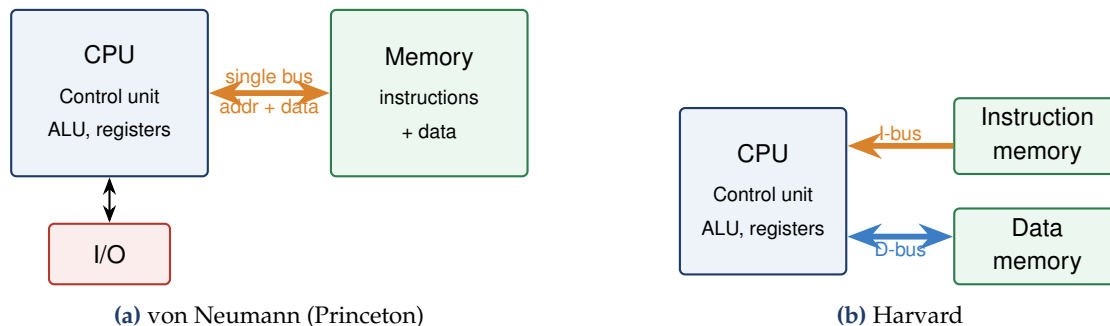


Figure 1.2: The two classical memory organizations.

The simplicity of a single memory and a single bus has a price: an instruction fetch and a data access cannot occur at the same time. The processor can compute no faster than the bus can deliver the words it needs — the **von Neumann bottleneck** (a phrase coined by John Backus in his 1977 Turing Award lecture).

1.3.2 The Harvard machine

The Harvard Mark I (designed by Howard Aiken, built by IBM, 1944) read its instructions from punched paper tape and kept data in electromechanical counters — two physically separate stores. A **Harvard architecture** therefore has **separate buses (and memories) for instructions and data**, so both can be accessed in the same cycle. The cost is duplicated wiring and a more complex, resource-intensive control design; and a pure Harvard machine cannot easily treat code as data (to load a program you need a special path).

Advanced Insight: The modified Harvard architecture in every modern core

Real processors are *von Neumann at the memory level and Harvard at the L1 level*. There is one unified physical address space in DRAM (so the OS can load programs), but the core has a split L1 instruction cache and L1 data cache with independent ports. A pipeline can then fetch an instruction and execute a load in the same cycle — exactly the structural-hazard

avoidance we will need in Unit 3. Digital-signal processors and many microcontrollers (e.g. AVR, PIC) go further and keep code in flash and data in SRAM with genuinely separate buses.

Worked Example 1.1: Quantifying the von Neumann bottleneck

A single-issue processor runs at $f = 2 \text{ GHz}$ and completes one instruction per cycle. Each instruction is 4 bytes. On average 35% of instructions are loads or stores, each moving 4 bytes. What memory bandwidth is demanded, and what happens if one shared bus delivers 8 GB/s?

Solution. Instruction bandwidth = $2 \times 10^9 \times 4 \text{ B} = 8 \text{ GB/s}$. Data bandwidth = $2 \times 10^9 \times 0.35 \times 4 \text{ B} = 2.8 \text{ GB/s}$. Total demand = 10.8 GB/s.

With a single 8 GB/s bus the processor is starved: it can execute at most

$$\frac{8 \times 10^9}{4 + 0.35 \times 4} \text{ instructions/s} = \frac{8 \times 10^9}{5.4} \approx 1.48 \times 10^9 \text{ instr/s,}$$

i.e. only 74% of its potential. With split instruction and data paths of 8 GB/s each, instruction fetch exactly saturates its bus and data uses 35% of the other — the processor runs at full speed. (Caches, studied in Unit 4, reduce the *off-chip* demand by servicing most of these accesses on-chip.)

1.4 The evolution of computers

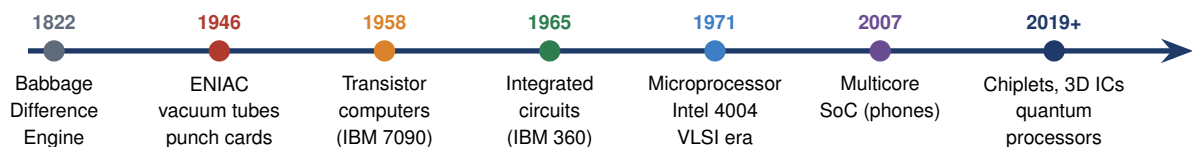


Figure 1.3: Milestones in the evolution of computing technology.

1. **Mechanical era** (1600s–1930s). Pascal’s and Leibniz’s calculators; Charles Babbage’s Difference Engine (1822) evaluated polynomials by finite differences, and his never-completed Analytical Engine already had a “mill” (ALU), a “store” (memory) and punched-card programs.
2. **Vacuum tubes** (1940s–1950s). ENIAC (1946) had 17 468 tubes, consumed about 150 kW and was programmed by rewiring; EDVAC and the Manchester Baby introduced the stored program. Punch cards and paper tape provided input/output.
3. **Transistors** (late 1950s). Discrete transistors replaced tubes; machines became smaller, cooler and far more reliable. High-level languages (FORTRAN, COBOL) and operating systems appeared.
4. **Integrated circuits** (1960s). Many transistors on one silicon die. The IBM System/360 (1964) introduced the idea of a *family* of machines sharing one architecture but with different organizations — the architecture/organization split of Section 1.1 made commercial.
5. **VLSI and microprocessors** (1971–). The Intel 4004 placed an entire 4-bit CPU on one chip with 2 300 transistors. Very-large-scale integration enabled pipelined, superscalar and out-of-order processors, and eventually complete systems on a chip.

6. **Beyond CMOS scaling.** Chiplets and 2.5D/3D integration, domain-specific accelerators, and quantum processors that use superposition and entanglement for specific problem classes.

1.5 The role of VLSI

Very-large-scale integration changed not just how computers are built but *what kinds* of computers are worth building:

- Feature size shrank from about $10\ \mu\text{m}$ (early 1970s) to the “3 nm” generation (2022–) — a linear factor above 3000 and an areal factor of millions.
- Transistor budgets made advanced organizations possible: pipelining, superscalar issue, out-of-order execution, large on-chip caches.
- Logic speed improved much faster than DRAM latency, so **CPU speed** $\gg$ **main-memory speed**: the von Neumann bottleneck became the *memory wall*, motivating Unit 4.
- Cheap logic made micro-controllers and networked embedded computers ubiquitous.
- Entire systems — CPU, GPU, memory controllers, I/O — integrate on a single chip (**SoC**), and 3D ICs stack dies for even higher density and bandwidth.

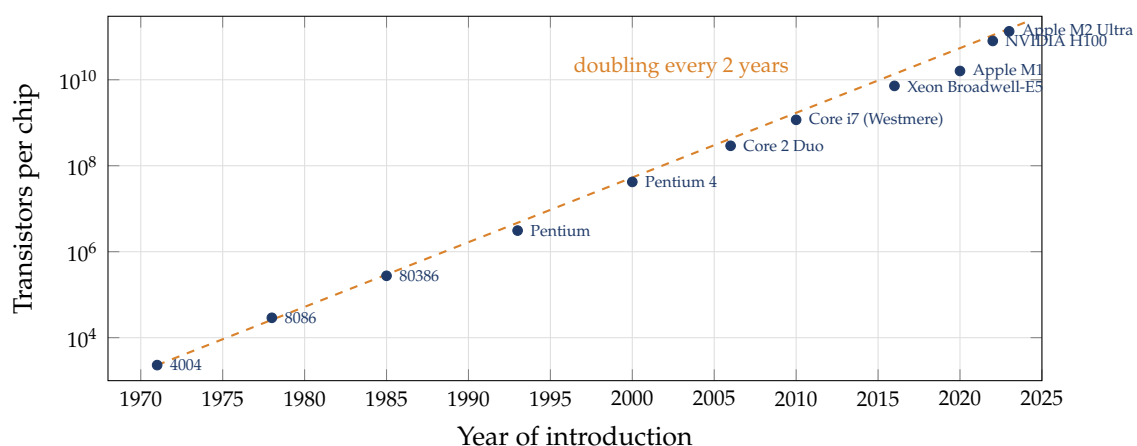


Figure 1.4: Moore’s law: transistor counts of representative processors (log scale). The dashed line doubles every two years starting from the 4004.

Worked Example 1.2: Moore’s law and Dennard scaling with numbers

(a) Starting from 2 300 transistors in 1971 and doubling every two years, predict the transistor count in 2021.

$N(2021) = 2300 \times 2^{(2021-1971)/2} = 2300 \times 2^{25} = 2300 \times 33\,554\,432 \approx 7.7 \times 10^{10}$. Real 2021–2022 flagship chips held $5\text{--}8 \times 10^{10}$ transistors, so the 50-year extrapolation is accurate within a factor of about 1.5.

(b) Under classical *Dennard scaling*, each generation shrinks linear dimensions and supply voltage by $\kappa = \sqrt{2} \approx 1.4$. Dynamic power of one gate is $P = \alpha CV^2 f$. Capacitance scales as $1/\kappa$, voltage as $1/\kappa$, frequency as κ ; so per-gate power scales as $\frac{1}{\kappa} \cdot \frac{1}{\kappa} \cdot \kappa = \frac{1}{\kappa} = \frac{1}{2}$. Area per gate also scales as $1/\kappa^2 = \frac{1}{2}$. Hence **power density stays constant** while there are twice as many gates running $1.4\times$ faster.

(c) After about 2005, V could no longer be reduced (threshold voltage and leakage limits).

With V fixed, per-gate power scales as $\frac{1}{\kappa} \cdot 1 \cdot \kappa = 1$, while area halves: **power density doubles every generation**. This “end of Dennard scaling” is why clock frequencies plateaued near 3–5 GHz and industry turned to multicore and specialised accelerators (“dark silicon”).

1.6 The hierarchy of digital systems

Complexity is managed by **abstraction**: each level hides the details of the level below and offers a simpler model to the level above. Figure 1.5 shows the levels used in this course.

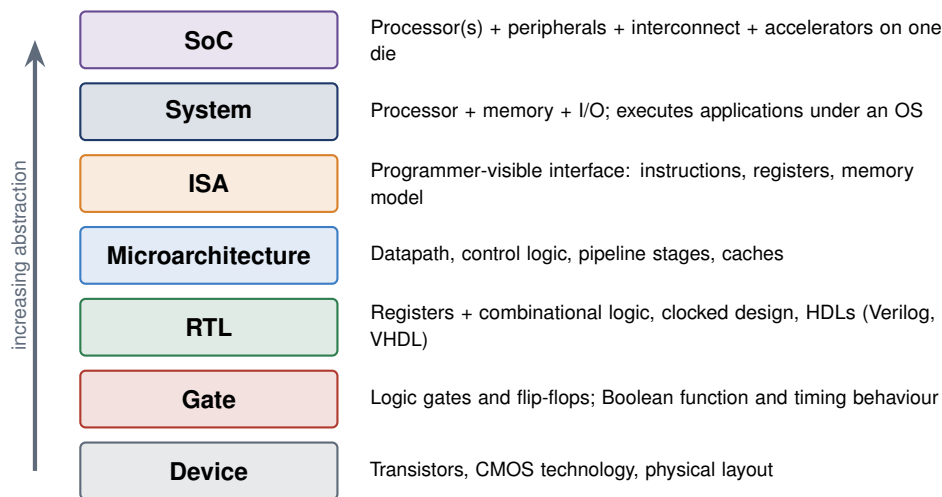


Figure 1.5: The digital-system hierarchy, from devices up to the system-on-chip.

Device level. Transistors, the CMOS technology that builds them, and their physical implementation (layout, lithography). The relevant quantities are currents, voltages, capacitances and delays.

Gate level. Logic gates (NAND, NOR, XOR, ...) and storage elements (latches, flip-flops). We reason with Boolean algebra plus timing (propagation delay, setup and hold).

Register-transfer level (RTL). Words of bits held in registers and transformed by combinational blocks between clock edges. Designs are written in hardware description languages (HDLs) such as Verilog or VHDL and synthesised into gates.

Microarchitecture level. The organization of a processor: datapath, control logic, pipeline stages, caches, predictors.

ISA level. The programmer-visible interface: instructions, registers, memory model.

System level. Processor + memory + I/O executing applications, usually under an operating system.

SoC level. One or more processors integrated with peripherals, interconnects and accelerators on one chip.

1.6.1 Computer architecture in levels: from problem to electrons

Computer architects often draw the stack from the other direction, as a chain of *transformations* from a human problem down to the motion of electrons (Figure 1.6). The ISA sits in the middle: everything above it is software, everything below it is hardware.

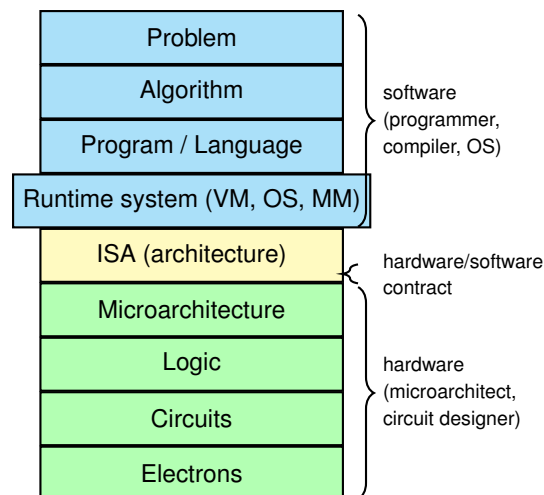


Figure 1.6: Computer architecture in levels: a problem is transformed until electrons can solve it.

Definition 1.3: Microarchitecture

Everything done in hardware *without exposure to software*: pipelining; in-order versus out-of-order execution; memory-access scheduling; speculative execution; superscalar (multiple-issue) processing; clock gating; caching (levels, size, associativity, replacement); prefetching; voltage/frequency scaling; error correction.

Design strategy. The best designs are produced by people who can move freely across levels. An algorithm designer who knows the cache hierarchy lays out data contiguously; a microarchitect who knows the compiler’s code-generation patterns optimises the common case. As the slide on design strategy emphasises, the user drives requirements through the problem/algorithm/program levels, and the runtime system (virtual machine, OS, memory manager) mediates between programs and the ISA.

1.7 The processor in an SoC and the hardware–software interface

Inside an SoC the processor is the **central control unit**: it executes software and coordinates peripherals by reading and writing their memory-mapped registers, handling their interrupts, and programming their DMA engines.

The ISA as a contract

The ISA is a *contract* between hardware and software. Software (compilers, OS, applications) may rely on exactly the behaviour the ISA specifies — no more, no less. Hardware may

implement that behaviour in any way it likes. Because both sides honour the same contract, a binary compiled in 2015 still runs on a 2026 processor, and a new processor can be introduced without recompiling the world.

The contract runs in both directions. If the specification says that the result of reading register `x0` is always zero, the hardware *must* guarantee it and software *may* use `x0` as a free constant. If the specification says the order in which two stores from different cores become visible is not guaranteed without a fence, software that omits the fence is wrong, even if it happens to work on today's chip.

Worked Example 1.3: Semantic gap: a digital clock

The lecture shows two clocks side by side: an LCD clock showing “12:15:45” and a “binary clock” of blue LEDs. Both implement the same *function* (time of day), but at very different *levels of abstraction*. The LCD clock does the work of translating to the human representation; the LED clock leaves it to the user, who must read each column as a BCD digit.

Reading the LED clock: columns are hours-tens, hours-units, minutes-tens, minutes-units, seconds-tens, seconds-units; rows are weights 8, 4, 2, 1 from top. A column with lit LEDs at weights 4 and 1 represents $4 + 1 = 5$. If the six columns show $\{1\}, \{2\}, \{1\}, \{4, 1\}, \{4\}, \{4, 1\}$ the time is 12:15:45.

This is the **semantic gap**: the distance between how a problem is expressed by people and how the machine represents it. A CISC ISA narrows the gap with complex instructions; a RISC ISA leaves the gap wide and relies on the compiler to bridge it (Chapter 5).

1.8 Cost, yield and why integration wins

Advanced Insight: Die cost and the economics of SoCs and chiplets

The cost of a die is

$$C_{\text{die}} = \frac{C_{\text{wafer}}}{N_{\text{dies/wafer}} \times Y_{\text{die}}}, \quad N_{\text{dies/wafer}} \approx \frac{\pi(d/2)^2}{A} - \frac{\pi d}{\sqrt{2A}}, \quad Y_{\text{die}} = \left(1 + \frac{D_0 A}{\alpha}\right)^{-\alpha},$$

where d is the wafer diameter, A the die area, D_0 the defect density and α a clustering parameter (≈ 3 –4).

For a 300 mm wafer costing \$17 000, $D_0 = 0.1 \text{ cm}^{-2}$, $\alpha = 4$:

$A \text{ (mm}^2\text{)}$	dies/wafer	yield	good dies	cost/good die
100	640	90.6%	580	\$29
400	143	68.3%	98	\$174
800	65	48.2%	31	\$544

(Example row for $A = 100 \text{ mm}^2 = 1 \text{ cm}^2$: $N = \pi \cdot 150^2/100 - \pi \cdot 300/\sqrt{200} = 706.9 - 66.6 \approx 640$; $Y = (1 + 0.1/4)^{-4} = 0.906$.)

Doubling die area more than triples cost. This is why very large designs are now split into **chiplets** — several small, high-yield dies joined by an advanced package — while small and medium systems integrate everything into one SoC to save board area, power and interconnect latency.

Chapter summary

- Architecture = what the programmer sees (ISA, formats, addressing, memory model); organization = how it is built (datapath, control, pipelines, caches).
- von Neumann machines share one memory/bus for instructions and data and suffer the bottleneck; Harvard machines split them. Modern cores are modified Harvard: split L1 caches over a unified memory.
- VLSI scaling (Moore's law) supplied the transistors for advanced organizations; the end of Dennard scaling pushed the industry to multicore, SoCs, accelerators and chiplets.
- Abstraction levels: device, gate, RTL, microarchitecture, ISA, system, SoC. The ISA is the contract that separates software from hardware.

Exercises

E1.1 Classify each as architectural or organizational: (a) the presence of a branch delay slot; (b) the number of entries in a branch target buffer; (c) the width of the virtual address; (d) whether the L1 cache is write-back; (e) the encoding of addi.

Answer: (a) architectural — it changes program semantics; (b) organizational; (c) architectural; (d) organizational; (e) architectural.

E1.2 A 1.5 GHz single-issue core issues one 4-byte instruction per cycle; 30% of instructions are loads/stores of 8 bytes. Compute the bandwidth demand. If a unified bus provides 6 GB/s, what fraction of peak performance is achievable?

Answer: $6 + 3.6 = 9.6 \text{ GB/s}$; $6 / (4 + 2.4) = 0.94 \times 10^9 \text{ instr/s}$, i.e. 62.5%.

E1.3 Using the yield model above with $D_0 = 0.2 \text{ cm}^{-2}$ and $\alpha = 4$, compute the yield of a 2 cm^2 die and of each half if it is split into two 1 cm^2 chiplets. Which is cheaper per good system, ignoring packaging cost?

Answer: $Y_2 = (1.1)^{-4} = 0.683$; $Y_1 = (1.05)^{-4} = 0.823$. Two chiplets need $2/0.823 = 2.43 \text{ cm}^2$ of silicon per good system versus $2/0.683 = 2.93 \text{ cm}^2$: chiplets win.

E1.4 Explain why the end of Dennard scaling makes a 32-core chip with all cores at full frequency thermally infeasible in a fixed power envelope. What is “dark silicon”?

E1.5 Give two examples each of information that crosses the ISA boundary from hardware to software and from software to hardware.

2

Gate-Level and Sequential Logic Design

Every processor, however sophisticated, is ultimately a network of gates and flip-flops. This chapter revisits digital design from the viewpoint of a computer architect: how a specification becomes a *system equation*, how the equation becomes gates with a measurable cost and delay, how adding memory (flip-flops) turns combinational functions into finite-state machines, and how a synchronous circuit is designed step by step. The running examples — the full adder, the serial adder, a serial multiply-by-three circuit and a synchronous mod-5 counter — reappear later as building blocks of datapaths.

2.1 The system equation

Any digital system with n inputs and m outputs computes

$$[out_1, out_2, \dots, out_m] = f(in_1, in_2, \dots, in_n). \quad (2.1)$$

If the outputs depend *only* on the present inputs the system is **combinational**. If they also depend on past inputs — i.e. the system has memory — it is **sequential**, and f must also take the current *state* as an argument and produce the next state.

The simplest interesting example is the two-input function

$$f(0,0) = 0, \quad f(0,1) = 1, \quad f(1,0) = 1, \quad f(1,1) = 0,$$

which is the **exclusive-OR**: $f(x_1, x_2) = x_1 \oplus x_2 = \bar{x}_1 x_2 + x_1 \bar{x}_2$.

2.1.1 From equation to gates

The sum-of-products form maps directly onto two NOT gates, two AND gates and one OR gate (Figure 2.1a). A function can also be viewed as a **graph**: nodes are operations and edges carry values from producers to consumers (Figure 2.1b). This *dataflow graph* view will reappear when we study dataflow machines and instruction scheduling: the length of the longest path determines the minimum delay, and nodes that are not connected by a path can work in parallel.

2.1.2 Cost and delay

Two numbers characterise a gate-level design:

- **Cost** — commonly measured as the number of gates, or the number of gate inputs (a better proxy for transistor count: a k -input CMOS NAND uses $2k$ transistors).
- **Delay** — the number of gate levels on the longest (*critical*) path from an input to an output, multiplied by the gate delay.

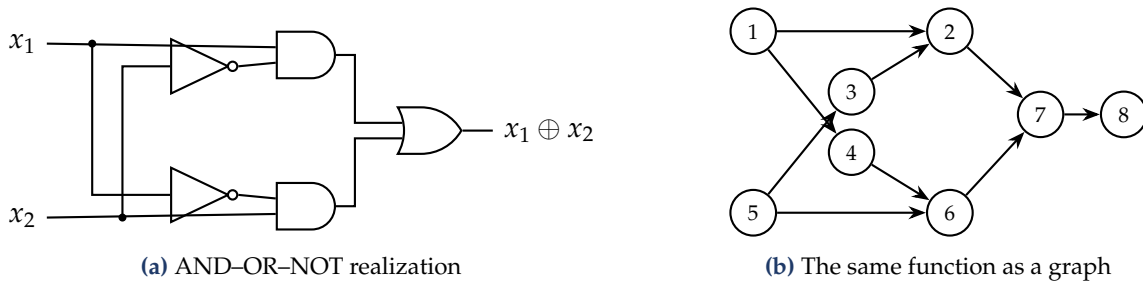


Figure 2.1: Two views of the XOR system equation. In (b) nodes 1, 5 are inputs, 3, 4 inverters, 2, 6 AND gates, 7 the OR gate and 8 the output.

Worked Example 2.1: Cost and delay of three XOR implementations

Assume every gate has delay τ and count transistors for static CMOS (NOT = 2, 2-input NAND/NOR = 4, 2-input AND/OR = 6 because they are NAND/NOR + NOT).

Implementation	Gates	Transistors	Delay
AND-OR-NOT (Fig. 2.1a)	2 NOT + 2 AND + 1 OR	$4 + 12 + 6 = 22$	3τ (NOT $\rightarrow$ AND $\rightarrow$ OR)
Four 2-input NANDs	4 NAND	16	3τ
Transmission-gate XOR	—	8–12	$\approx 1-2\tau$

The NAND-only version is $\overline{\overline{x_1 x_1 x_2} \cdot \overline{x_2 x_1 x_2}}$: the first NAND computes $p = \overline{x_1 x_2}$, two NANDs compute $\overline{x_1 p}$ and $\overline{x_2 p}$, and a final NAND combines them. It saves 27% of the transistors at the same logical depth — the reason NAND is the “universal” gate in standard-cell libraries.

2.1.3 An all-NAND and all-NOR design exercise

The lecture poses the function

$$f_1(a, b, c, d) = a(b + \bar{c})d + \bar{a}(b + d)(b + c)(c + d) + b\bar{c}\bar{d}.$$

Worked Example 2.2: Minimising f_1 and realising it with NAND gates

Step 1: minterms. Evaluating all 16 input combinations ($abcd$ read as a binary number) gives $f_1 = \sum m(3, 4, 5, 6, 7, 9, 12, 13, 15)$.

Step 2: Karnaugh map. (Rows ab , columns cd , both in Gray order.)

$ab \backslash cd$	00	01	11	10
00	0	0	1	0
01	1	1	1	1
11	1	1	1	0
10	0	1	0	0

Prime implicants that are needed: $\bar{a}b$ (row 01), $b\bar{c}$ (columns 00/01 of rows 01/11), bd (columns 01/11 of rows 01/11), $\bar{a}cd$ (covers m_3), $a\bar{c}d$ (covers m_9).

$$f_1 = \bar{a}b + b\bar{c} + bd + \bar{a}cd + a\bar{c}d.$$

Step 3: NAND-NAND. A two-level AND-OR network converts directly to NAND-NAND by De Morgan’s law, $\sum P_i = \overline{\prod \bar{P}_i}$: five first-level NANDs (three 2-input: $\bar{a}\bar{b}$, $\bar{b}\bar{c}$, $\bar{b}\bar{d}$; and two

3-input: $\overline{acd}, \overline{a\bar{c}d}$, one 5-input output NAND, plus two NANDs used as inverters to make $\bar{a}$ and $\bar{c}$. Total: 8 NAND gates, 3 levels including the inverters.

Step 4: NOR–NOR. Use the product-of-sums form $f_1 = (b + d)(a + b + c)(\bar{a} + b + \bar{c})(\bar{a} + \bar{c} + d)$: four first-level NORs, one 4-input output NOR, plus inverters for $\bar{a}, \bar{c}$ — also 7–8 gates. Either form is acceptable; the choice in practice depends on which gate is faster in the target technology (NAND in CMOS, because series nMOS transistors are faster than series pMOS).

2.2 The design hierarchy and the design process

Computer design is carried out at three broad levels, distinguished by the size of the components and the unit of information they handle (Table 2.1).

Table 2.1: System design hierarchy (after J. P. Hayes).

Level	Components	IC density	Information unit
Gate	Logic gates, flip-flops	SSI	Bits
Register	Registers, counters, combinational circuits, small sequential circuits	MSI	Words
Processor	CPUs, memories, I/O devices	VLSI	Blocks of words

At every level the design process is iterative (Figure 2.2): construct an initial design, evaluate its cost and performance, and if the goals are not met, modify the design and repeat.

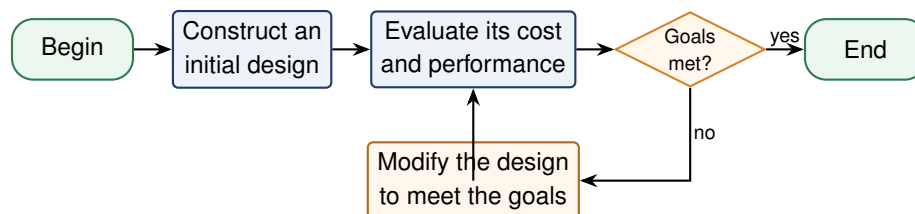


Figure 2.2: The iterative design loop.

2.3 Combinational design: the full adder

2.3.1 Specification and equations

A **full adder** adds two operand bits x_0, y_0 and an incoming carry c_{-1} , producing a sum bit s_0 and an outgoing carry c_0 .

x_0	y_0	c_{-1}	c_0	s_0
0	0	0	0	0
0	0	1	0	1
0	1	0	0	1
0	1	1	1	0
1	0	0	0	1
1	0	1	1	0
1	1	0	1	0
1	1	1	1	1

Reading off the minterms,

$$s_0 = \bar{x}_0\bar{y}_0c_{-1} + \bar{x}_0y_0\bar{c}_{-1} + x_0\bar{y}_0\bar{c}_{-1} + x_0y_0c_{-1} = x_0 \oplus y_0 \oplus c_{-1}, \quad (2.2)$$

$$c_0 = x_0y_0 + x_0c_{-1} + y_0c_{-1} \quad (\text{the majority function}). \quad (2.3)$$

2.3.2 Three implementations

1. **Two-level AND–OR**: four 3-input ANDs and a 4-input OR for s_0 ; three 2-input ANDs and a 3-input OR for c_0 ; three inverters. Delay 3τ (with inverters).
2. **Two half adders and an OR** (Figure 2.3): a half adder produces $x_0 \oplus y_0$ and x_0y_0 ; a second half adder adds c_{-1} . Then $c_0 = x_0y_0 + (x_0 \oplus y_0)c_{-1}$. This is the form used in most cell libraries because the *propagate* signal $p = x_0 \oplus y_0$ and *generate* signal $g = x_0y_0$ are reused by fast adders.
3. **All-NAND**: the half-adder version maps to 9 two-input NAND gates (the classic textbook figure on the slides, with an XOR made from four NANDs and the OR built from a NAND fed by inverted inputs).

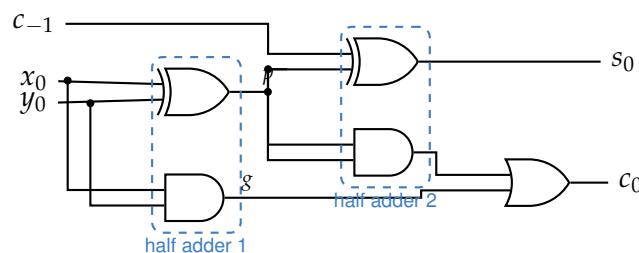


Figure 2.3: Full adder built from two half adders and an OR gate.

2.3.3 The ripple-carry adder

Connecting n full adders in a chain, carry-out to carry-in, gives an n -bit **ripple-carry adder** (Figure 2.4). It is cheap (n full adders) but slow: the carry may have to propagate through all n stages.

Worked Example 2.3: Ripple-carry delay

In a full adder the carry path $c_{-1} \rightarrow c_0$ passes through one AND and one OR (2τ) and the path $x_0, y_0 \rightarrow p$ through one XOR ($\approx 2\tau$ if XOR = 2 gate levels). For an n -bit ripple-carry adder the worst case is: generate p_0 (2τ), then n carry stages (2τ each), then the final XOR

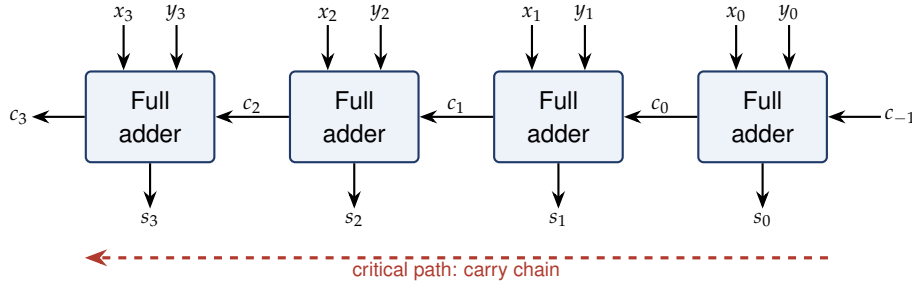


Figure 2.4: A 4-bit ripple-carry adder. At the register level it is drawn as a single block with 4-bit inputs X, Y , carry-in c_{-1} , 4-bit sum S and carry-out c_3 .

for s_{n-1} (2τ):

$$T_{\text{RCA}}(n) \approx 2\tau + 2n\tau + 2\tau = (2n + 4)\tau.$$

With $\tau = 20$ ps: $n = 4$: 240 ps; $n = 32$: 1.36 ns; $n = 64$: 2.64 ns. A 64-bit ripple adder alone would limit the clock below 400 MHz — unacceptable for a modern core.

Advanced Insight: Carry-lookahead and parallel-prefix adders

Define per bit $g_i = x_i y_i$ (generate) and $p_i = x_i \oplus y_i$ (propagate). Then $c_i = g_i + p_i c_{i-1}$, and unrolling,

$$c_3 = g_3 + p_3 g_2 + p_3 p_2 g_1 + p_3 p_2 p_1 g_0 + p_3 p_2 p_1 p_0 c_{-1}.$$

A 4-bit carry-lookahead (CLA) block computes all four carries in two gate levels after g, p . Group signals $G = g_3 + p_3 g_2 + p_3 p_2 g_1 + p_3 p_2 p_1 g_0$ and $P = p_3 p_2 p_1 p_0$ let CLA blocks be nested hierarchically, giving delay $O(\log n)$. Parallel-prefix adders (Kogge–Stone, Brent–Kung, Han–Carlson) treat (g, p) pairs with the associative operator $(g, p) \circ (g', p') = (g + pg', pp')$ and compute all carries in $\lceil \log_2 n \rceil$ prefix levels: for $n = 64$ that is 6 levels, i.e. roughly $2\tau + 6 \cdot 2\tau + 2\tau = 16\tau = 320$ ps with the numbers above — $8\times$ faster than ripple, at the cost of $O(n \log n)$ area. Every high-performance integer ALU uses some prefix adder.

2.4 Sequential circuits

2.4.1 The D flip-flop and its timing

A sequential circuit stores state in **flip-flops**. The edge-triggered D flip-flop samples its input D at the rising clock edge and holds it at output y (also written Q) until the next edge: $y(i+1) = D(i)$, where i counts clock cycles.

Three parameters govern every synchronous design:

- **Setup time** t_{su} : D must be stable this long *before* the edge.
- **Hold time** t_h : D must remain stable this long *after* the edge.
- **Clock-to-Q delay** t_{cq} : time from the edge to a valid output.

A change of D inside the setup–hold window (a “glitch” near the triggering edge on the slide) may drive the flip-flop into a *metastable* state.

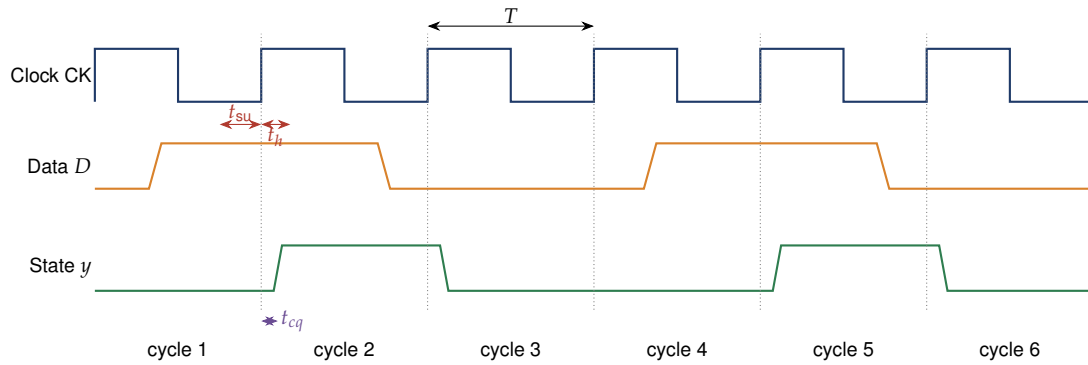


Figure 2.5: Timing of an edge-triggered D flip-flop. The data must be stable for t_{su} before and t_h after each rising edge; the output changes t_{cq} after the edge.

Worked Example 2.4: Maximum clock frequency and hold check

A register-to-register path has $t_{cq} = 60$ ps, combinational logic delay $t_{logic,max} = 740$ ps, $t_{logic,min} = 35$ ps, $t_{su} = 50$ ps, $t_h = 40$ ps and clock skew $t_{skew} = 30$ ps (capture clock arrives late).

Setup constraint: $T \geq t_{cq} + t_{logic,max} + t_{su} + t_{skew} = 60 + 740 + 50 + 30 = 880$ ps, so $f_{max} = 1/880$ ps ≈ 1.14 GHz.

Hold constraint: $t_{cq} + t_{logic,min} \geq t_h + t_{skew}$, i.e. $60 + 35 = 95 \geq 40 + 30 = 70$ ps. Satisfied with 25 ps slack. If the shortest path had only 5 ps of logic, $65 < 70$ would violate hold, and buffers would have to be inserted — a hold violation cannot be fixed by slowing the clock.

2.4.2 Finite-state machines: Moore and Mealy

A synchronous sequential circuit is a **finite-state machine (FSM)**: combinational next-state logic, a state register, and combinational output logic.

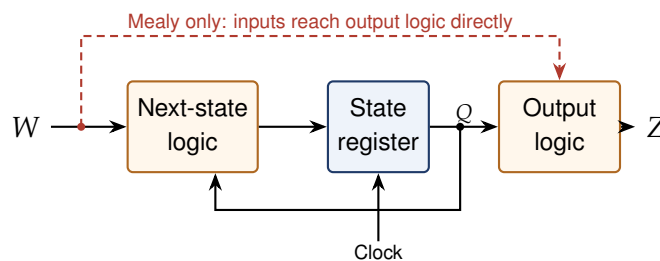


Figure 2.6: Generic FSM. In a **Moore** machine the output depends only on the state Q ; in a **Mealy** machine (red path) it also depends directly on the primary inputs W .

- **Moore:** $Z = \lambda(Q)$. Outputs change only after clock edges, so they are glitch-free and easy to time, but the machine typically needs more states and reacts one cycle later.
- **Mealy:** $Z = \lambda(Q, W)$. Fewer states and a same-cycle response, but an input glitch can propagate to the output, and a combinational path exists from input to output through the machine.

2.4.3 A Mealy example: the serial adder

A **serial adder** adds two numbers presented one bit per clock cycle, least significant bit first. Only one bit of history matters: the carry. So the machine has two states, S_0 (carry = 0) and S_1 (carry

$= 1$), implemented by a single D flip-flop y whose output feeds the carry input of a full adder (Figure 2.7).

Present state	Input x_1x_2 : next state, output z			
	00	01	10	11
S_0 ($y = 0$)	$S_0, 0$	$S_0, 1$	$S_0, 1$	$S_1, 0$
S_1 ($y = 1$)	$S_0, 1$	$S_1, 0$	$S_1, 0$	$S_1, 1$

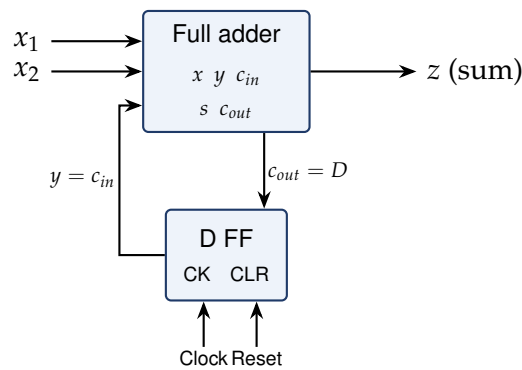


Figure 2.7: Serial adder: one full adder plus one D flip-flop holding the carry.

Worked Example 2.5: Running the serial adder

Add $X = 0110_2 = 6$ and $Y = 0111_2 = 7$. Bits enter LSB first; the flip-flop is reset to 0.

Cycle	x_1	x_2	y (carry in)	z	next y	state
1	0	1	0	1	0	$S_0 \rightarrow S_0$
2	1	1	0	0	1	$S_0 \rightarrow S_1$
3	1	1	1	1	1	$S_1 \rightarrow S_1$
4	0	0	1	1	0	$S_1 \rightarrow S_0$

Output (LSB first) $1, 0, 1, 1 \Rightarrow Z = 1101_2 = 13 = 6 + 7$. ✓ The hardware cost is one full adder and one flip-flop regardless of word length — trading time (n cycles) for area, the oldest trade-off in computer design.

2.4.4 A second Mealy example: serial multiply-by-three

The lecture asks for a sequential circuit that multiplies an unsigned number N of arbitrary length by 3, with N entered serially (LSB first) on input x and $3N$ emerging serially on output z , using D flip-flops and NAND gates only.

Worked Example 2.6: Designing the $\times 3$ serial multiplier

Idea. $3N = 2N + N$. Bit i of $2N$ is bit $i - 1$ of N . So at cycle i we must add x_i (bit of N), x_{i-1} (bit of $2N$) and a carry c . Because $x_i + x_{i-1} + c \leq 3$, the carry is a single bit. The machine must remember two bits: $p = x_{i-1}$ and c .

Equations (full adder on x, p, c):

$$z = x \oplus p \oplus c, \quad c^+ = xp + xc + pc, \quad p^+ = x.$$

State table (state = pc):

State pc	$x = 0$: next, z	$x = 1$: next, z
00 (A)	00, 0	10, 1
01 (B)	00, 1	11, 0
10 (C)	00, 1	11, 0
11 (D)	01, 0	11, 1

States B and C have identical rows, so they are equivalent and can be merged: the minimal machine has **three** states, i.e. still two flip-flops but with a simpler next-state logic if encoded carefully. (Equivalently, the state can be interpreted as the pending value $p + c \in \{0, 1, 2\}$.)

Trace for $N = 1011_2 = 11$ followed by two flushing zeros (the product of an n -bit number by 3 needs $n + 2$ bits):

cycle	x	p	c	z	p^+	c^+
0	1	0	0	1	1	0
1	1	1	0	0	1	1
2	0	1	1	0	0	1
3	1	0	1	0	1	1
4	0	1	1	0	0	1
5	0	0	1	1	0	0

z (LSB first) = 1, 0, 0, 0, 0, 1 $\Rightarrow 100001_2 = 33 = 3 \times 11$. ✓

NAND realisation. c^+ is the majority function $\overline{x}p \cdot \overline{x}c \cdot \overline{p}c$ (four NANDs); z is a three-input XOR (two 4-NAND XORs). Two D flip-flops store p and c .

2.5 Synchronous sequential design procedure

The lecture lists six steps for designing a synchronous counter or FSM:

- Step 1:** Determine the number of flip-flops required.
- Step 2:** Draw the state diagram.
- Step 3:** Write the excitation table of the chosen flip-flop type.
- Step 4:** Build the state-excitation table (present state, next state, required flip-flop inputs).
- Step 5:** Obtain the flip-flop input equations using Karnaugh maps.
- Step 6:** Draw the circuit from the K-map equations.

2.5.1 Flip-flop excitation tables

An excitation table answers the *inverse* question of a characteristic table: given a desired transition $Q_N \rightarrow Q_{N+1}$, which inputs cause it? (X = don't care.)

Q_N	Q_{N+1}	S	R	J	K	D	T
0	0	0	X	0	X	0	0
0	1	1	0	1	X	1	1
1	0	0	1	X	1	0	1
1	1	X	0	X	0	1	0

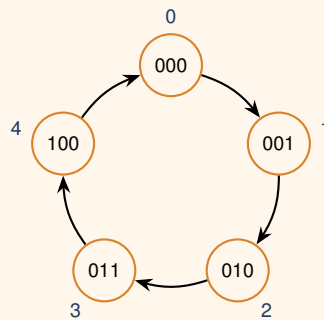
Note how the JK flip-flop has the most don't-cares: it combines the freedom of SR (set or reset) with the toggle of T. More don't-cares mean larger K-map groups and simpler logic.

2.5.2 Complete example: synchronous mod-5 counter with JK flip-flops

Worked Example 2.7: Synchronous mod-5 counter

Step 1. A modulo-5 counter counts 0, 1, 2, 3, 4, 0, ... The number of flip-flops n satisfies $2^{n-1} < N \leq 2^n$; for $N = 5$, $2^2 = 4 < 5 \leq 8$, so $n = 3$ ($Q_2Q_1Q_0$).

Step 2. State diagram.



Steps 3–4. State-excitation table (JK excitation from the table above):

Present			Next			Flip-flop inputs					
Q_2	Q_1	Q_0	Q_2^+	Q_1^+	Q_0^+	J_2	K_2	J_1	K_1	J_0	K_0
0	0	0	0	0	1	0	X	0	X	1	X
0	0	1	0	1	0	0	X	1	X	X	1
0	1	0	0	1	1	0	X	X	0	1	X
0	1	1	1	0	0	1	X	X	1	X	1
1	0	0	0	0	0	X	1	0	X	0	X
1	0	1	-	-	-	X	X	X	X	X	X
1	1	0	-	-	-	X	X	X	X	X	X
1	1	1	-	-	-	X	X	X	X	X	X

Step 5. K-maps. (Rows Q_2 , columns Q_1Q_0 in Gray order 00, 01, 11, 10.)

	00	01	11	10
J_2 : 0	0	0	1	0
1	X	X	X	X

K_2 : all entries 1 or X

	00	01	11	10
J_1 : 0	0	1	X	X
1	0	X	X	X

	00	01	11	10
K_1 : 0	X	X	1	0
1	X	X	X	X

	00	01	11	10
J_0 : 0	1	X	X	1
1	0	X	X	X

K_0 : all entries 1 or X

Grouping with the don't-cares:

$$J_2 = Q_1 Q_0, \quad K_2 = 1; \quad J_1 = Q_0, \quad K_1 = Q_0; \quad J_0 = \overline{Q_2}, \quad K_0 = 1.$$

Step 6. Circuit. Three JK flip-flops clocked together; one 2-input AND gate for J_2 ; J_1 and K_1 wired to Q_0 ; J_0 wired to $\overline{Q_2}$; K_2 and K_0 tied to V_{cc} .

Verification of the sequence: from 011: $J_2 = 1, K_2 = 1$ toggles $Q_2 \rightarrow 1$; $J_1 = K_1 = 1$ toggles $Q_1 \rightarrow 0$; $J_0 = 1, K_0 = 1$ toggles $Q_0 \rightarrow 0$: next = 100. ✓ From 100: $J_2 = 0, K_2 = 1$ resets Q_2 ; $J_1 = K_1 = 0$ holds $Q_1 = 0$; $J_0 = 0, K_0 = 1$ resets Q_0 : next = 000. ✓

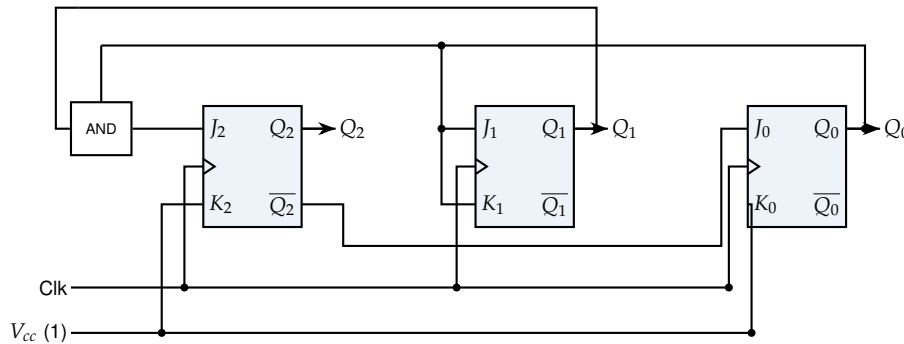


Figure 2.8: Synchronous mod-5 counter: $J_2 = Q_1 Q_0$, $K_2 = 1$, $J_1 = K_1 = Q_0$, $J_0 = \overline{Q_2}$, $K_0 = 1$.

Pitfall: forgetting the unused states

A mod-5 counter built with 3 flip-flops has three unused states (101, 110, 111). After a power-up glitch the counter could start in one of them. Always check that every unused state eventually returns to the main cycle ("self-starting"). For the design above: 101 $\rightarrow$ 010, 110 $\rightarrow$ 010, 111 $\rightarrow$ 000 — the counter is self-starting. If it were not, the don't-cares must be assigned explicitly to force a return, or a reset must be guaranteed.

Worked Example 2.8: The same counter with D flip-flops

With D flip-flops $D_i = Q_i^+$ directly. From the table (unused states as don't-cares): $D_2 = Q_1 Q_0$, $D_1 = Q_1 \oplus Q_0$, $D_0 = \overline{Q_2} \overline{Q_0}$. Gate count: one AND, one XOR, one NOR ($\overline{Q_2} + \overline{Q_0}$) — versus a single AND for the JK design. The JK version needs fewer gates but each JK flip-flop is larger than a D flip-flop; in modern standard-cell design D flip-flops are used almost exclusively, because timing analysis and scan-chain test insertion assume them.

Advanced Insight: State encoding, metastability and synchronizers

Encoding. An FSM with S states needs at least $\lceil \log_2 S \rceil$ flip-flops (binary encoding), but *one-hot* encoding (one flip-flop per state) often produces smaller, faster next-state logic on FPGAs, where flip-flops are plentiful. Gray encoding minimises switching activity (power) for sequential state traversals.

Metastability. When an asynchronous input violates t_{su}/t_h , a flip-flop may hover at an intermediate voltage before resolving. The probability that it is still unresolved after time t_r decays as e^{-t_r/τ_m} , giving

$$\text{MTBF} = \frac{e^{t_r/\tau_m}}{T_0 f_{clk} f_{data}}.$$

With $\tau_m = 20$ ps, $T_0 = 50$ ps, $f_{clk} = 1$ GHz, $f_{data} = 100$ MHz and one full cycle of resolution time ($t_r \approx 900$ ps): $e^{45} \approx 3.5 \times 10^{19}$, so MTBF $\approx 3.5 \times 10^{19} / (50 \times 10^{-12} \cdot 10^9 \cdot 10^8) = 7 \times 10^{12}$ s ≈ 220 000 years. With only half a cycle ($t_r = 400$ ps), $e^{20} \approx 4.9 \times 10^8$ and MTBF ≈ 0.1 s — a certain failure. This is why every clock-domain crossing uses a two-flip-flop synchronizer.

Chapter summary

- A digital system is a system equation **out** = $f(\mathbf{in})$ (combinational) or $(\mathbf{out}, \text{state}^+) = f(\mathbf{in}, \text{state})$ (sequential).
- Gate-level designs are judged by cost (gates or transistors) and delay (levels on the critical path). NAND/NOR are universal; two-level SOP maps to NAND–NAND.
- The full adder ($s = x \oplus y \oplus c$, $c^+ = \text{maj}(x, y, c)$) is the atom of arithmetic; ripple adders are $O(n)$ slow, prefix adders $O(\log n)$.
- Synchronous timing: $T \geq t_{cq} + t_{logic} + t_{su} + t_{skew}$; hold must be met independently of T .
- Moore outputs depend on state only; Mealy outputs also on inputs. Design flow: count flip-flops, state diagram, excitation table, K-maps, circuit; check unused states.

Exercises

E2.1 Implement $f = \bar{a}b + ac$ with (a) NAND gates only and (b) NOR gates only. Count gates and levels.

Answer: (a) $\overline{\bar{a}b \cdot \bar{a}c}$: 3 NAND + 1 inverter-NAND = 4 gates, 3 levels. (b) POS $f = (a + b)(\bar{a} + c)$: $\overline{a + \bar{b} + \bar{a} + c}$, 4 NORs incl. inverter.

E2.2 Compute the delay of a 16-bit ripple-carry adder and of a 16-bit Kogge–Stone adder using the models in this chapter ($\tau = 25$ ps).

Answer: RCA: $(2 \cdot 16 + 4)\tau = 36\tau = 900$ ps. KS: $2\tau + 4 \cdot 2\tau + 2\tau = 12\tau = 300$ ps.

E2.3 Trace the serial adder for $X = 1011$, $Y = 0101$.

Answer: $11 + 5 = 16$: outputs 0,0,0,0 with a final carry of 1 — a fifth cycle with zero inputs is needed to emit it.

E2.4 Design a synchronous mod-6 counter using T flip-flops. Is it self-starting?

Answer: $T_0 = 1$, $T_1 = Q_0\bar{Q}_2$, $T_2 = Q_0Q_1 + Q_0Q_2$; unused states $110 \rightarrow 111 \rightarrow 010$: self-starting.

E2.5 Design a Moore FSM that outputs 1 when the last three input bits were 101 (overlapping allowed). How many states does the equivalent Mealy machine need?

Answer: Moore: 4 states ($S_0, S_1, S_{10}, S_{101}$). Mealy: 3 states.

E2.6 A path has $t_{cq} = 45$ ps, $t_{su} = 30$ ps, skew 20 ps. What is the largest logic delay allowed at 2.5 GHz?

Answer: $400 - 45 - 30 - 20 = 305$ ps.

E2.7 Repeat the $\times 3$ serial multiplier trace for $N = 0111_2$.

Answer: $z = 10101_2 = 21$.

3

Register-Transfer Design, Programmable Logic and Processor Performance

The gate level is too fine-grained to design a processor: a 32-bit adder is already hundreds of gates. Designers therefore work at the **register-transfer level (RTL)**, where the primitives are multi-bit components — multiplexers, adders, comparators, registers, counters — and the behaviour is described as transfers of words between registers on clock edges. This chapter develops the RTL view, shows how programmable logic devices (ROM, PLA, PAL, FPGA) implement arbitrary functions, climbs to the processor level, and closes with the quantitative performance equation that every later unit uses.

3.1 Hardware design abstraction

Figure 3.1 shows the full translation chain from an application to transistors. Each arrow is a tool or a design step; each box is a representation.

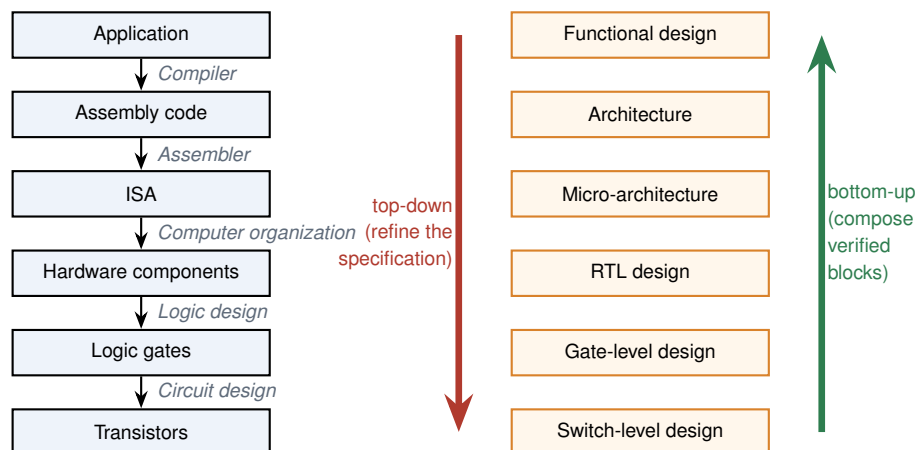


Figure 3.1: Hardware design abstraction. Left: the software-to-silicon translation chain. Right: the hardware design levels, traversed top-down during specification and bottom-up during integration.

- **Top-down design** starts from the function, partitions it into an architecture, then a microarchitecture, then RTL blocks, refining until every block is a library component. It keeps the global goals visible.
- **Bottom-up design** builds and verifies small blocks (cells, adders, register files) and composes them. It exploits re-use and accurate characterisation.

Real projects mix both: architects work top-down while library and IP teams work bottom-up,

meeting at the RTL.

3.2 The register-transfer level

3.2.1 The multifunction unit

The generic RTL component (Figure 3.2) has m -bit **data input lines** $X_1, X_2, X_3, \dots$, a k -bit **function select** F choosing one of up to 2^k operations, an **enable** E , **data output lines** Z_1, Z_2 , and a **termination status** S that tells a controller the operation has finished (needed for multi-cycle operations such as division). Control inputs (F, E) come from the control unit; control outputs (S) go back to it. This split between *data* and *control* is the fundamental organizing principle of every processor: **datapath + controller**.

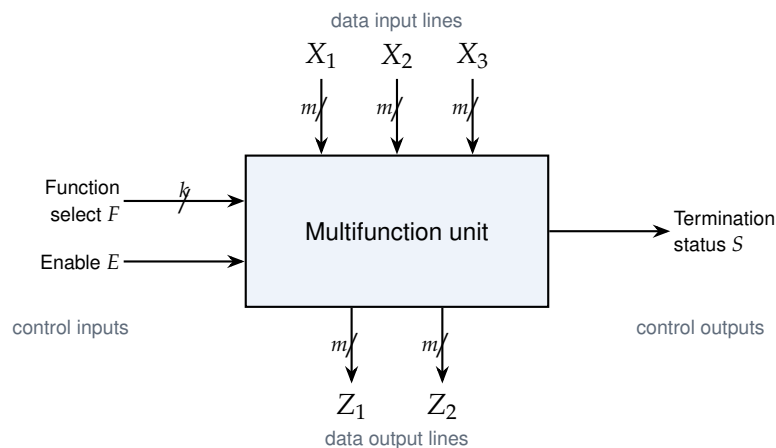


Figure 3.2: A generic register-level multifunction unit.

3.2.2 Multiplexers as universal function generators

A **multiplexer** (MUX) with p select bits routes one of 2^p m -bit data inputs $X_0, \dots, X_{2^p-1}$ to its output Z :

$$Z = \sum_{i=0}^{2^p-1} (S = i) \cdot X_i \cdot e.$$

A 2^p -input, 1-bit multiplexer can implement *any* Boolean function of p variables: connect the variables to the select lines and the truth-table column (constants 0/1) to the data inputs. Each data input is literally one row of the truth table.

Worked Example 3.1: A full adder from multiplexers

Version 1 (8-way, 2-bit MUX). Use x_0, y_0, c_{-1} as the three select lines and wire each of the eight 2-bit data inputs to the constant pair (s_0, c_0) of the corresponding truth-table row: $X_0 = 00, X_1 = 10, X_2 = 10, X_3 = 01, X_4 = 10, X_5 = 01, X_6 = 01, X_7 = 11$ (written s_0c_0). No gates at all — only constants.

Version 2 (optimized: 4-way, 2-bit MUX). Use only x_0, y_0 as selects and feed *functions of the remaining variable* c_{-1} to the data inputs:

x_0y_0	input	s_0	c_0
00	X_0	c_{-1}	0
01	X_1	$\bar{c}_{-1}$	c_{-1}
10	X_2	$\bar{c}_{-1}$	c_{-1}
11	X_3	c_{-1}	1

Only one inverter is needed (for $\bar{c}_{-1}$). The general rule (Shannon expansion): $f(a, b, c) = \bar{a}\bar{b}f(0, 0, c) + \bar{a}bf(0, 1, c) + a\bar{b}f(1, 0, c) + abf(1, 1, c)$; a 2^p -way MUX plus at most one inverter realises any function of $p + 1$ variables.

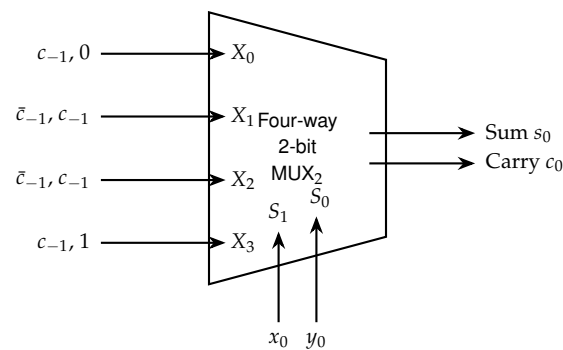


Figure 3.3: Optimized multiplexer implementation of a full adder.

3.2.3 A 4-bit magnitude comparator from adders

Re-using arithmetic blocks for comparison is a recurring theme (the RISC-V ALU computes `slt` and branch conditions with its subtractor). The lecture's comparator (Figure 3.4) uses two 4-bit adders whose *sum* outputs are ignored.

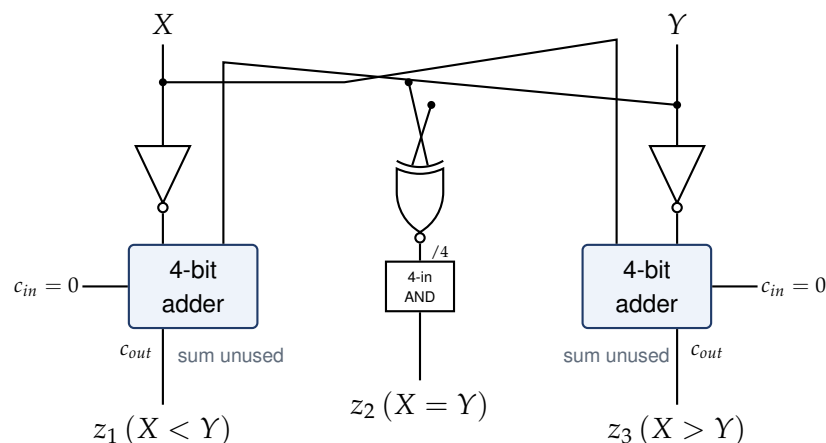


Figure 3.4: 4-bit comparator: two adders produce $X < Y$ and $X > Y$ from their carry-outs; a bitwise XNOR followed by a 4-input AND produces $X = Y$.

Worked Example 3.2: Why the carry-out of $\bar{X} + Y$ means $X < Y$

For 4-bit unsigned numbers $\bar{X} = 15 - X$. The left adder computes $\bar{X} + Y = 15 - X + Y$; its carry-out is 1 iff the sum is ≥ 16 , i.e. iff $Y - X \geq 1$, i.e. $X < Y$. Symmetrically the right adder

sets z_3 iff $X > Y$.

Check with $X = 0101_2 = 5$, $Y = 1001_2 = 9$: $\bar{X} = 1010_2 = 10$; $10 + 9 = 19 \geq 16$ so $z_1 = 1$.

Right adder: $X + \bar{Y} = 5 + 6 = 11 < 16$, $z_3 = 0$. XNOR: $0101 \odot 1001 = 0011$, AND = 0 so $z_2 = 0$. ✓

3.2.4 Registers and the sequential data loader

A **register** is a set of m D flip-flops sharing a clock. A *loadable* register holds its value unless LOAD is asserted; CLEAR resets it. Internally the load function is a 2-way multiplexer in front of the flip-flops that selects either the new data X (LOAD=1) or the register's own output (LOAD=0), as in Figure 3.5. In RTL notation:

if LOAD then $Z \leftarrow X$; if CLEAR then $Z \leftarrow 0$.

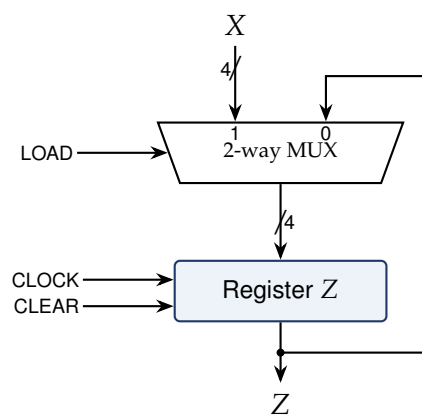


Figure 3.5: Implementation of a 4-bit loadable register (sequential data loader).

Advanced Insight: Clock gating versus load-enable multiplexers

The feedback-MUX register burns dynamic power every cycle because the flip-flops are clocked even when holding. Low-power designs replace the MUX by an *integrated clock-gating cell* (latch + AND) that stops the register's clock when LOAD=0. Synthesis tools do this automatically for banks of ≥ 4 –8 flip-flops sharing an enable; on mobile SoCs clock gating typically saves 20–40% of dynamic power. It is the “clock gating” item in the list of microarchitectural techniques in Chapter 1.

3.3 Programmable logic devices

A **programmable logic device (PLD)** implements user-defined functions in a fixed silicon fabric. All classical PLDs exploit the two-level AND–OR (sum-of-products) form, and differ in which plane is programmable.

- **PLA (programmable logic array):** both the AND plane and the OR plane are programmable. Most flexible; product terms can be shared among outputs. The lecture's PLA has 51 product terms over its inputs.
- **ROM:** the AND plane is a *fixed* full decoder generating all 2^n minterms; only the OR plane is programmed. A ROM with n address bits and m data bits implements any m functions of n variables — the truth table is literally stored.

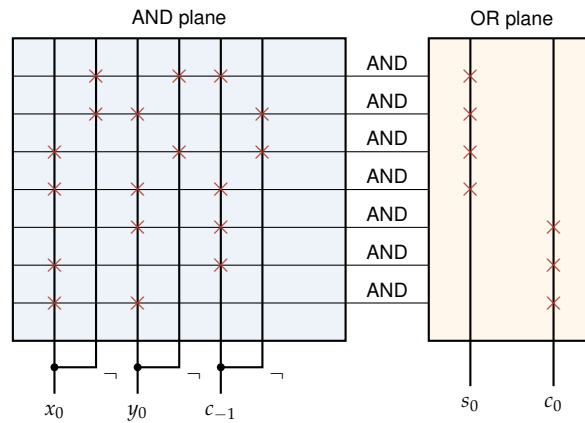


Figure 3.6: A PLA programmed as a full adder. Each horizontal line is a product term; a $\times$ marks a programmed connection. Seven product terms suffice: four minterms for s_0 and the three terms y_0c_{-1} , x_0c_{-1} , x_0y_0 for c_0 .

- **PAL (programmable array logic)** — the question posed on the ROM slide: the AND plane is programmable but the OR plane is *fixed*, each output OR-ing a dedicated group of (typically 7–8) product terms. PALs (and their successors GALs) are faster and cheaper than PLAs because the fixed OR plane needs no programmable switches, at the cost of no product-term sharing.

Table 3.1: Comparison of two-level PLDs.

Device	AND plane	OR plane	Best for
ROM	fixed (full decoder)	programmable	dense, arbitrary truth tables (lookup)
PLA	programmable	programmable	sparse functions with shared terms
PAL	programmable	fixed	fast control logic, glue logic

Worked Example 3.3: Sizing ROM and PLA implementations

(a) Implement a 4-bit $\times$ 4-bit unsigned multiplier as a ROM. Address width = 8 bits, data width = 8 bits: $2^8 \times 8 = 2048$ bits. For 8 $\times$ 8-bit operands: $2^{16} \times 16 = 1\,048\,576$ bits = 128 KiB — the exponential growth of ROM size in the number of inputs is why ROMs implement only small functions (or are used as lookup tables inside larger algorithms, as in table-based division and transcendental functions).

(b) The full adder as a ROM: $2^3 \times 2 = 16$ bits. As a PLA: 3 inputs, 7 product terms, 2 outputs; the AND plane has $2 \cdot 3 \times 7 = 42$ crosspoints and the OR plane $7 \times 2 = 14$: 56 programmable points — more than the ROM's 16 bits, because the full adder is *not* sparse (half of all minterms are used). PLAs win only when few product terms are needed relative to 2^n .

(c) A 7-segment decoder (4 inputs, 7 outputs) as a ROM: $16 \times 7 = 112$ bits.

3.3.1 Field-programmable gate arrays

Multiplexer-based logic cells. Since a 4-input 1-bit MUX implements any function of 3 variables (two on select lines, functions of the third on the data lines), an array of such cells interconnected by programmable wiring can implement any circuit. The lecture's examples show one MUX cell producing $abcd$ (AND), another producing $a + b + c + d$ (OR) by appropriate tie-offs, and another

producing $\bar{a}$ — each is a generic block whose behaviour is set entirely by its input connections. Early Actel FPGAs were built exactly this way.

LUT-based FPGAs. Modern FPGAs (AMD/Xilinx, Intel/Altera, Lattice) use k -input **look-up tables (LUTs)**: a 2^k -bit configuration SRAM addressed by the k inputs — i.e. a tiny ROM — followed by an optional flip-flop. A **configurable logic block (CLB)** groups several LUTs, flip-flops, fast carry chains and wide multiplexers. CLBs are connected through **connection blocks (CB)** and **switch blocks (SB)** arranged as a programmable **crossbar**-like mesh (Figure 3.7). Configuration memory — written at power-up from a bitstream — defines the LUT contents and every switch setting.

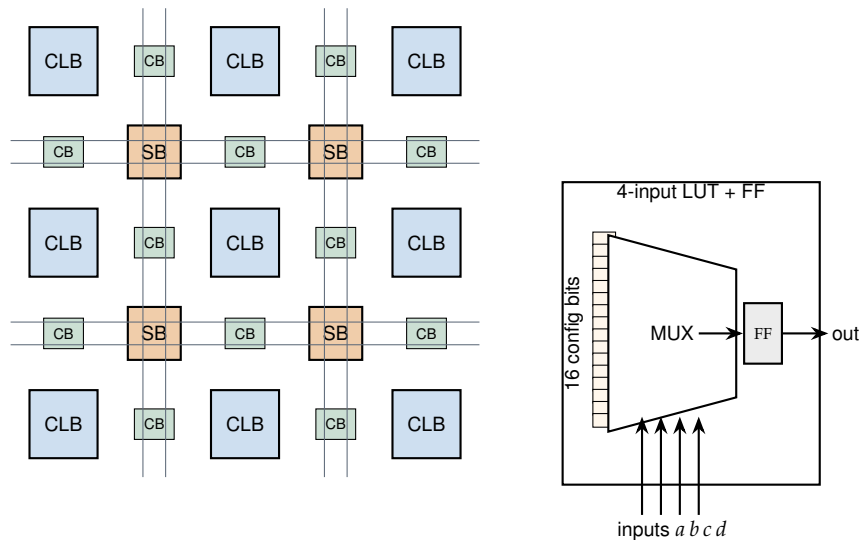


Figure 3.7: Island-style FPGA fabric: configurable logic blocks (CLB), connection blocks (CB) and switch blocks (SB). Right: a 4-input LUT is a 16-to-1 MUX selecting one of 16 configuration bits, optionally registered.

Worked Example 3.4: Mapping functions to LUTs

- (a) How many 4-LUTs does a 16-bit equality comparator need? Each bit pair needs an XNOR; a 4-LUT can XNOR two bit-pairs *and* AND them (4 inputs). That gives 8 LUTs for the first level, then $8 \rightarrow 2$ (each LUT ANDs 4 signals), then $2 \rightarrow 1$: $8 + 2 + 1 = 11$ LUTs, 3 levels of logic.
- (b) With 6-input LUTs (modern AMD/Xilinx devices): first level, 3 bit-pairs per LUT $\Rightarrow \lceil 16/3 \rceil = 6$ LUTs; second level, one LUT ANDs 6 signals: total 7 LUTs, 2 levels.
- (c) Configuration bits for a 6-LUT: $2^6 = 64$. A device with 400 000 6-LUTs stores 25.6 Mbit of LUT configuration alone — and the routing switches typically need several times more.

3.3.2 FPGA-based SoCs: Zynq and the ZedBoard

The AMD (Xilinx) **Zynq-7000** integrates a *processing system* (PS) — a dual-core ARM Cortex-A9 with NEON SIMD and FPUs, 32 KB L1 I/D caches per core, a 512 KB shared L2, 256 KB on-chip memory, DDR controller, and peripherals (USB, Ethernet, SD, UART, SPI, I²C, CAN, GPIO) — with a block of *programmable logic* (PL) on the same die. PS and PL communicate through several AXI interfaces: general-purpose ports, high-performance ports to the DDR controller, and the accelerator coherency port (ACP) into the L2/snoop control unit. The **ZedBoard** is a low-cost development board for the Zynq-7020 with 512 MB DDR3, HDMI and VGA outputs,

audio, OLED, switches, push buttons, Pmod and FMC expansion headers. In this course it is the platform for building a RISC-V core in the PL alongside the ARM cores.

3.4 Processor-level design

3.4.1 The processor as a datapath plus a controller

At the processor level, the components are CPUs, memories, I/O controllers and buses. The CPU itself is decomposed into a **datapath** (registers, ALU, multiplexers, internal buses) and a **control unit** that issues the control signals (F , E , $LOAD$, MUX selects) in the correct sequence.

3.4.2 Memory addressing

Most machines are **byte addressable**: each address names one byte, and a 32-bit word occupies four consecutive addresses. Word addresses are therefore multiples of 4 ($0x0000_0000$, $0x0000_0004$, $\dots$, $0xFFFF_FFFC$).

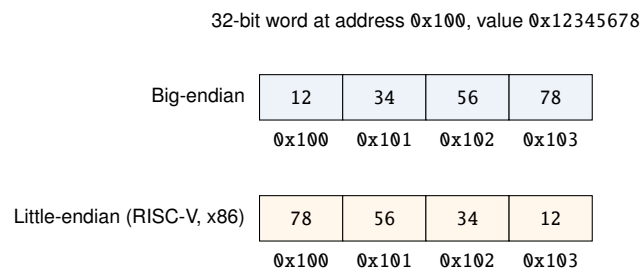


Figure 3.8: Byte ordering of a 32-bit word in a byte-addressable memory.

- **Little-endian:** the least significant byte is at the lowest address (RISC-V, x86, ARM by default).
- **Big-endian:** the most significant byte is at the lowest address (network byte order, IBM z, classic MIPS/SPARC).
- **Alignment:** a k -byte access is aligned if its address is a multiple of k . Misaligned accesses may be slower, trap, or be split into two accesses.

The instruction cycle in register-transfer notation (square brackets = “contents of”):

1. Fetch the instruction pointed to by the PC: $IR \leftarrow [[PC]]$.
2. Increment the PC (byte addressable, 4-byte instructions): $PC \leftarrow [PC] + 4$.
3. Carry out the actions specified by the instruction in IR.

3.4.3 A single-bus processor

Figure 3.9 shows the classic single-bus organization (after Hamacher et al.). All registers — PC, MAR (memory address register), MDR (memory data register), IR, general registers $R0 \dots R(n-1)$, temporaries Y , Z , $TEMP$ — hang on one internal bus. The ALU takes one operand from Y and the other from the bus (or the constant 4 via a MUX, for PC increment), and writes its result to Z .

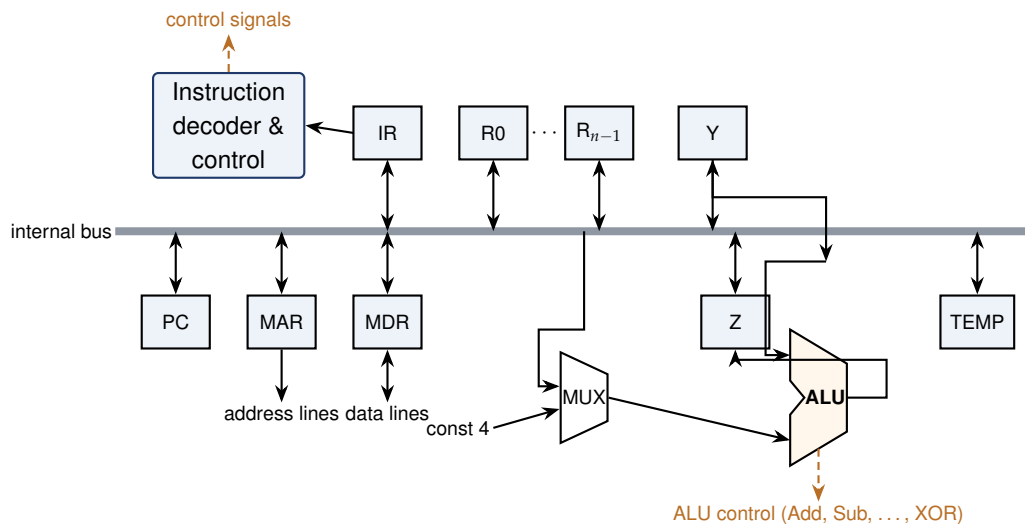


Figure 3.9: Single-bus processor organization.

Worked Example 3.5: Control sequence for ADD R1, (R3) on the single-bus machine

The instruction adds the memory word whose address is in R3 to R1. Each numbered step is one clock cycle; “out” gates a register onto the bus, “in” loads it from the bus.

Step	Action
1	PC _{out} , MAR _{in} , Read, Select4, Add, Z _{in}
2	Z _{out} , PC _{in} , Y _{in} , WMFC (wait for memory function complete)
3	MDR _{out} , IR _{in}
4	R3 _{out} , MAR _{in} , Read
5	R1 _{out} , Y _{in} , WMFC
6	MDR _{out} , SelectY, Add, Z _{in}
7	Z _{out} , R1 _{in} , End

Seven cycles because only one transfer can use the bus per cycle. With a 1 GHz clock and a memory that answers in one cycle, this instruction takes 7 ns. A three-bus organization (two source buses, one destination bus) can merge steps 5–7 into a single cycle; a pipelined Harvard organization (Unit 3) overlaps the fetch of the next instruction with the execution of this one. The single-bus machine is the archetype of *multi-cycle, microprogrammed* control studied in Unit 3.

3.4.4 Coprocessors

A **coprocessor** extends a CPU with specialised functions (floating point, cryptography, graphics) while sharing the system bus. The CPU selects the coprocessor through an address decoder, the coprocessor raises BUSY while working, can raise an *interrupt request* on completion or error, and uses *synchronization signals* to align its operand transfers with the CPU’s instruction stream. The Intel 8087 FPU was the classic example; today coprocessors are mostly integrated as functional units (the RISC-V F/D extensions) or as memory-mapped accelerators on an SoC interconnect.

3.5 Quantifying processor performance

3.5.1 The performance equation

Let a program execute N instructions on a processor with clock frequency f (MHz) and average cycles per instruction CPI. Then

$$T = \frac{N \times \text{CPI}}{f} \mu\text{s}, \quad t_E = \frac{T}{N} = \frac{\text{CPI}}{f} \mu\text{s}, \quad \text{MIPS} = \frac{N}{T \times 10^6} \cdot 10^6 = \frac{f}{\text{CPI}}. \quad (3.1)$$

If instruction type I_i occurs with probability p_i and takes time t_i (or CPI_i cycles), the average instruction time is

$$t_E = \sum_{i=1}^n p_i t_i \mu\text{s}, \quad \text{CPI} = \sum_{i=1}^n p_i \text{CPI}_i. \quad (3.2)$$

The iron law of performance

Time = $\frac{\text{Instructions}}{\text{Program}} \times \frac{\text{Cycles}}{\text{Instruction}} \times \frac{\text{Seconds}}{\text{Cycle}}$. The ISA and compiler set the first factor, the microarchitecture the second, and circuit technology plus microarchitecture the third. Improving one factor often worsens another.

Worked Example 3.6: CPI from the RISC-V instruction mix

The lecture gives a typical RISC-V dynamic mix: arithmetic/logic 40%, load/store 30%, branch/control 20%, immediate 5%, system 3%, other 2%. On a multi-cycle implementation the cycle counts are: ALU 4, load/store 5 (take loads and stores together), branch 3, immediate 4, system 6, other 8.

$$\begin{aligned} \text{CPI} &= 0.40 \cdot 4 + 0.30 \cdot 5 + 0.20 \cdot 3 + 0.05 \cdot 4 + 0.03 \cdot 6 + 0.02 \cdot 8 \\ &= 1.60 + 1.50 + 0.60 + 0.20 + 0.18 + 0.16 = 4.24. \end{aligned}$$

At $f = 500 \text{ MHz}$: $t_E = 4.24/500 \mu\text{s} = 8.48 \text{ ns}$, $\text{MIPS} = 500/4.24 = 117.9$. A program of $N = 2 \times 10^8$ instructions runs in $T = 2 \times 10^8 \times 8.48 \text{ ns} = 1.70 \text{ s}$.

Worked Example 3.7: Comparing two design alternatives

Design A: $f = 1.0 \text{ GHz}$, $\text{CPI} = 1.0$ (single-cycle — see Unit 2) but the long critical path actually limits it to $f = 0.4 \text{ GHz}$. Design B: multi-cycle, $f = 2.0 \text{ GHz}$, $\text{CPI} = 4.24$. Design C: 5-stage pipeline, $f = 1.8 \text{ GHz}$, $\text{CPI} = 1.25$ (stalls).

Times per instruction: A: $1.0/0.4 = 2.50 \text{ ns}$; B: $4.24/2.0 = 2.12 \text{ ns}$; C: $1.25/1.8 = 0.69 \text{ ns}$. Speedup of C over A = $2.50/0.69 = 3.6$. The pipeline wins because it attacks the CPI and the cycle time *simultaneously* — the subject of Unit 3.

Pitfall: using MIPS to compare different ISAs

MIPS ignores how much work an instruction does. If ISA X needs 1.5×10^9 instructions at CPI 1.2 on a 2 GHz clock, and ISA Y needs 1.0×10^9 instructions at CPI 1.6 on the same clock: $\text{MIPS}_X = 2000/1.2 = 1667$, $\text{MIPS}_Y = 2000/1.6 = 1250$, so X “looks” 33% faster. But $T_X = 1.5 \times 10^9 \times 1.2/2 \times 10^9 = 0.90 \text{ s}$ and $T_Y = 1.0 \times 10^9 \times 1.6/2 \times 10^9 = 0.80 \text{ s}$: Y is actually 12.5% faster. Only execution time on real programs is a trustworthy metric.

Advanced Insight: Amdahl's law and the geometric mean

If an enhancement speeds up a fraction F of execution time by a factor S , the overall speedup is

$$\text{Speedup} = \frac{1}{(1 - F) + F/S}.$$

Example: an FP coprocessor makes floating-point code $10\times$ faster, and FP accounts for 40% of runtime: speedup = $1/(0.6 + 0.04) = 1.56$. Even with $S \rightarrow \infty$ the limit is $1/0.6 = 1.67$. Amdahl's law is the quantitative form of "make the common case fast".

When summarising speedups over a benchmark suite (as SPEC CPU does), use the *geometric* mean of the ratios: it is the only mean for which the ranking of two machines does not depend on which machine is chosen as the reference.

Chapter summary

- RTL design manipulates words with multi-bit components (MUX, adder, comparator, register) separated into datapath and control.
- A 2^p -way MUX implements any function of p (or $p + 1$ with an inverter) variables; ROMs store whole truth tables; PLAs program both planes; PALs program only the AND plane; FPGAs use LUTs + programmable routing.
- Processor level: byte-addressed memory (endianness, alignment), the fetch cycle $IR \leftarrow [[PC]]$, $PC \leftarrow [PC] + 4$, and organizations from single-bus to pipelined.
- $T = N \cdot \text{CPI} / f$; CPI is the probability-weighted average over instruction classes; MIPS is misleading across ISAs; Amdahl's law bounds every enhancement.

Exercises

E3.1 Implement $f(a, b, c, d) = \sum m(1, 2, 6, 7, 9, 12, 13, 15)$ using one 8-to-1 MUX with a, b, c on the select lines.

Answer: $X_0 \dots X_7 = d, \bar{d}, 0, 1, d, 0, 1, d$ (one inverter for $\bar{d}$).

E3.2 How many bits does a ROM need to implement an 8-bit binary-to-BCD converter?

Answer: 8 address bits, 10 output bits (3 BCD digits, the hundreds digit needs only 2 bits): $256 \times 10 = 2560$ bits.

E3.3 Write the control sequence for SUB R2, R4 ($R2 \leftarrow R2 - R4$) on the single-bus machine. How many cycles?

Answer: Fetch (3 cycles) + $R2_{out}, Y_{in}$; $R4_{out}, \text{SelectY}, \text{Sub}, Z_{in}$; $Z_{out}, R2_{in}, \text{End}$: 6 cycles.

E3.4 A program runs 40% ALU (CPI 1), 25% loads (CPI 3), 10% stores (CPI 2), 25% branches (CPI 2) at 1.5 GHz with 10^9 instructions. Find CPI, MIPS and T .

Answer: $\text{CPI} = 0.4 + 0.75 + 0.2 + 0.5 = 1.85$; $\text{MIPS} = 811$; $T = 1.23$ s.

E3.5 A cache improvement halves the CPI of loads in the previous exercise. What is the overall speedup? Verify with Amdahl's law.

Answer: New CPI = 1.475; speedup = 1.254. Fraction of time in loads $F = 0.75/1.85 = 0.405$, $S = 2$: $1/(0.595 + 0.203) = 1.254$.

E3.6 Store the 32-bit value `0xCAFEF00D` at address `0x2000` on a little-endian machine. Which byte is at `0x2002`?

Answer: `0xFE`.

4

Floating-Point Representation and Arithmetic

Integers are exact but have a fixed, narrow range; real-world quantities span from Avogadro's number to the charge of an electron. Floating-point numbers trade exactness for range by storing a number in scientific notation. This chapter covers the IEEE 754 standard in depth: the formats and their special values, the algorithms for addition, subtraction, multiplication and division (each with fully worked bit-level examples), rounding and the loss of precision, conversion between integers and floating point, the structure of a hardware floating-point unit, and the reduced-precision formats that power today's AI accelerators.

4.1 Why floating point?

4.1.1 The limits of fixed point

A 32-bit two's-complement integer covers $[-2^{31}, 2^{31} - 1] \approx \pm 2.1 \times 10^9$. It cannot represent

- very large numbers, e.g. 3.14159×10^{23} ;
- very small numbers, e.g. 1.602×10^{-19} (the electron charge in coulombs);
- fractions, e.g. 3.14159.

Fixed-point formats place an implied binary point at a fixed position. In $Qm.n$ format a word has m integer bits and n fraction bits; the value is the integer interpretation times 2^{-n} .

Worked Example 4.1: Fixed-point versus floating-point range

A 16-bit signed Q8.8 number has resolution $2^{-8} = 0.0039$ and range $[-128, 127.996]$. Its *dynamic range* (largest / smallest positive) is $32767/1 \approx 3.3 \times 10^4$, i.e. about 90 dB.

A 16-bit IEEE half-precision number (Section 4.2.2) has largest value 65 504 and smallest positive (subnormal) value $2^{-24} \approx 5.96 \times 10^{-8}$: dynamic range $\approx 1.1 \times 10^{12}$, i.e. 240 dB — with the same 16 bits. The price: the *relative* precision is constant ($\approx 2^{-11}$) rather than the *absolute* precision.

4.1.2 Scientific notation

A floating-point number is written $\pm M \times B^E$ where M is the **mantissa** (significand), B the **base** (2 for binary computers) and E the **exponent**. Examples: $123.45 = 1.2345 \times 10^2$; $0.00012 = 1.2 \times 10^{-4}$; in binary, $101.01_2 = 1.0101_2 \times 2^2$. A number is **normalized** when the mantissa has exactly one non-zero digit before the point. In binary that digit is always 1, so it need not be stored — the **hidden bit**.

4.2 The IEEE 754 standard

IEEE 754 was published in 1985 (revised 2008 and 2019) to standardise floating-point arithmetic across platforms. It defines:

- consistent bit-level representations (so files and network data are portable);
- correctly rounded basic operations ($+$, $-$, $\times$, $\div$, $\sqrt{}$, FMA) under defined rounding modes;
- special values (signed zero, infinities, NaN, subnormals);
- five exception flags: invalid, division by zero, overflow, underflow, inexact.

4.2.1 General format



Figure 4.1: The IEEE 754 single-precision (binary32) format.

- **Sign bit S :** 0 = positive, 1 = negative (sign-magnitude representation).
- **Exponent E :** stored in *biased* form; the true exponent is $E - \text{bias}$.
- **Mantissa M :** the fraction bits after the hidden 1.

$$\text{value} = (-1)^S \times (1.M) \times 2^{E-\text{bias}} \quad (\text{normalized numbers}). \quad (4.1)$$

Why a biased exponent?

With a bias, the exponent field is an *unsigned* integer that grows monotonically with the number's magnitude. Because the exponent sits above the mantissa, two non-negative floats can be compared by comparing their bit patterns *as unsigned integers*: a cheap integer comparator sorts floats. Two's-complement exponents would break this property.

4.2.2 Precision formats

Table 4.1: IEEE 754 binary formats.

Format	Bits	Sign	Exponent	Mantissa	Bias = $2^{k-1} - 1$	Decimal digits
Half (binary16)	16	1	5	10	15	≈ 3.3
Single (binary32)	32	1	8	23	127	≈ 7.2
Double (binary64)	64	1	11	52	1023	≈ 15.9
Quadruple (binary128)	128	1	15	112	16383	≈ 34.0

The number of decimal digits is $(p) \log_{10} 2$ where $p = \text{mantissa bits} + 1$ (hidden bit): for single precision $24 \times 0.301 = 7.22$.

Single precision. Exponent range -126 to $+127$ (stored 1 to 254), range $\pm 1.18 \times 10^{-38}$ to $\pm 3.40 \times 10^{38}$, about 7 decimal digits.

Double precision. 1/11/52 bits, bias 1023, exponent range -1022 to $+1023$, range $\pm 2.23 \times 10^{-308}$ to $\pm 1.80 \times 10^{308}$, 15–17 significant digits; used in scientific computing.

Half precision. 1/5/10 bits, bias 15, exponent range -14 to $+15$, about 3–4 decimal digits; used in graphics, mobile and AI/ML.

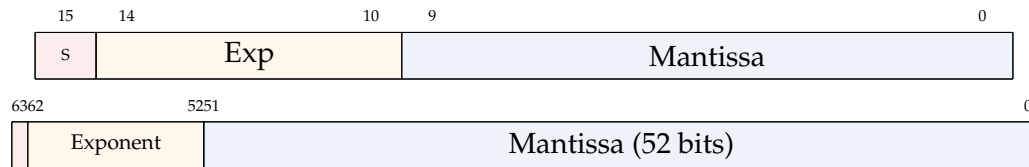


Figure 4.2: Half-precision (top) and double-precision (bottom) layouts.

4.3 Encoding and decoding

4.3.1 Encoding procedure

1. Convert the magnitude to binary (integer part by repeated division by 2, fraction part by repeated multiplication by 2).
2. Normalize to $1.f \times 2^e$.
3. Sign bit from the sign; exponent field = $e + \text{bias}$; mantissa field = f padded (and rounded) to the field width.

Worked Example 4.2: 12.75 in single precision

Step 1. $12.75 = 12 + 0.75 = 1100_2 + 0.11_2 = 1100.11_2$.

Step 2. $1100.11_2 = 1.10011_2 \times 2^3$.

Step 3. $S = 0$; $E = 3 + 127 = 130 = 10000010_2$; $M = 10011\ 000 \dots 0$ (23 bits).

0 10000010 10011000000000000000000

Grouping in nibbles: 0100 0001 0100 1100 0000 $\dots = 0x414C0000$.

Note: the hexadecimal value on the lecture slide (0x41480000) contains a slip in the third nibble; the bit pattern above is the correct one, as any C program printing `*(uint32_t*)&f` confirms.

Worked Example 4.3: -0.1875 in single precision

$0.1875 = 3/16 = 0.0011_2 = 1.1_2 \times 2^{-3}$. $S = 1$; $E = -3 + 127 = 124 = 01111100_2$; $M = 1000 \dots 0$.

1 01111100 10000000000000000000000 = 0xBE400000.

Worked Example 4.4: 5.75 in half precision

$5.75 = 101.11_2 = 1.0111_2 \times 2^2$. $S = 0$; $E = 2 + 15 = 17 = 10001_2$; $M = 0111000000$.

0 10001 0111000000 = 0x45C0.

Worked Example 4.5: Decoding 0xC0B40000

Binary: 1100 0000 1011 0100 0000... $S = 1$; $E = 10000001_2 = 129$, true exponent 2; $M = 01101000...$, so the significand is $1.01101_2 = 1 + 0.25 + 0.125 + 0.03125 = 1.40625$.
Value $= -1.40625 \times 2^2 = -5.625$.

Worked Example 4.6: 0.1 is not exactly representable

$0.1_{10} = 0.0001\overline{1}_2$ (repeating). Normalized: $1.\overline{001}_2 \times 2^{-4}$. $E = -4 + 127 = 123 = 01111011_2$. The 23-bit mantissa is 10011001100110011001100 followed by the remainder 1100...; since the remainder is more than half an ULP, round up: 10011001100110011001101.

$$0.1 \mapsto 0x3DCCCCD = 0.100000001490116...$$

The representation error is 1.49×10^{-9} (relative $1.5 \times 10^{-8} < 2^{-24}$). This is why `0.1f + 0.2f == 0.3f` may be false and why money must never be stored in binary floating point.

4.4 Special values

Table 4.2: IEEE 754 special cases (single precision).

Case	Exponent	Mantissa	Value	Example bit pattern (hex)
Zero	all 0s	all 0s	± 0	0x00000000, 0x80000000
Subnormal	all 0s	non-zero	$\pm(0.M) \times 2^{-126}$	0x00000001 = 2^{-149}
Normalized	1 to 254	any	$\pm(1.M) \times 2^{E-127}$	0x3F800000 = 1.0
Infinity	all 1s	all 0s	$\pm\infty$	0x7F800000, 0xFF800000
NaN	all 1s	non-zero	not a number	0x7FC00000 (quiet NaN)

- **Signed zero.** $+0$ and -0 have different bit patterns but compare equal. The sign matters in limits: $1/+0 = +\infty$, $1/-0 = -\infty$.
- **Infinity** results from overflow or division of a non-zero number by zero; arithmetic on infinities follows the rules of limits ($\infty + 1 = \infty$, $-5/+0 = -\infty$).
- **NaN** results from invalid operations: $0/0$, $\infty - \infty$, $\infty \times 0$, $\sqrt{-1}$. NaN compares *unequal to everything, including itself* — `x != x` is the portable NaN test. A *quiet* NaN (MSB of mantissa = 1) propagates silently; a *signalling* NaN raises the invalid flag when used.

4.4.1 Subnormal (denormalized) numbers

When the exponent field is 0, the hidden bit is 0 and the exponent is fixed at 1 – bias (-126 for single):

$$\text{value} = (-1)^S \times (0.M) \times 2^{-126}.$$

The smallest positive subnormal is $0.00\dots01_2 \times 2^{-126} = 2^{-23} \times 2^{-126} = 2^{-149} \approx 1.4 \times 10^{-45}$, while the smallest positive normal is $2^{-126} \approx 1.18 \times 10^{-38}$.

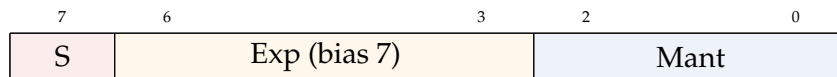
Subnormals provide **gradual underflow**: they fill the gap between zero and the smallest normal number with evenly spaced values, which preserves the important property

$$x - y = 0 \iff x = y.$$

Without subnormals, two different tiny numbers could subtract to zero (“flush to zero”), and code such as `if (x != y) z = 1/(x-y);` could divide by zero.

4.4.2 A complete picture: an 8-bit toy format

To see the whole number line, define an 8-bit format: 1 sign bit, 4 exponent bits (bias = 7), 3 mantissa bits.



Worked Example 4.7: Properties of the 8-bit format

Encoding 5.5: $5.5 = 101.1_2 = 1.011_2 \times 2^2$; $E = 2 + 7 = 9 = 1001_2$; $M = 011$: $\mathbf{0\ 1001\ 011 = 0x4B}$.

Range. Normal exponents $E = 1 \dots 14$ give true exponents $-6 \dots 7$.

- Largest normal: $1.111_2 \times 2^7 = 1.875 \times 128 = 240$ ($\mathbf{0\ 1110\ 111}$).
- Smallest normal: $1.000_2 \times 2^{-6} = 0.015625$.
- Largest subnormal: $0.111_2 \times 2^{-6} = 0.013671875$.
- Smallest subnormal: $0.001_2 \times 2^{-6} = 2^{-9} = 0.001953125$.

Spacing. Between 2^e and 2^{e+1} there are $2^3 = 8$ evenly spaced values with spacing 2^{e-3} : in $[1, 2)$ the values are $1, 1.125, \dots, 1.875$; in $[128, 240]$ the spacing is 16. The *relative* spacing is always between $1/16$ and $1/8$.

Precision trap. $5.5 + 0.1$: 0.1 rounds to the nearest representable value $0.1015625 = 1.101_2 \times 2^{-4}$; aligning to 2^2 shifts it right by 6 bits, beyond the 3 mantissa bits, so $5.5 + 0.1 = 5.5$ in this format. Range and precision are severely limited in such a small format — yet 8-bit formats (FP8) are used today in AI inference (Section 4.11).

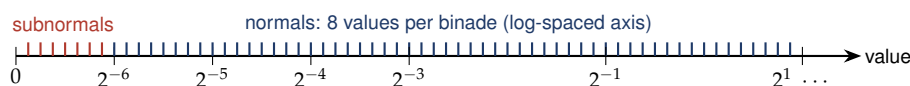


Figure 4.3: Schematic number line of the 8-bit format near zero: subnormals (red) are evenly spaced in $[0, 2^{-6})$ with the same spacing as the first binade of normals, giving gradual underflow. Each binade $[2^e, 2^{e+1})$ holds 8 values.

4.5 Floating-point addition and subtraction

4.5.1 Algorithm

To compute $A \pm B$:

1. **Extract** sign, exponent and mantissa of both operands (restore hidden bits).
2. **Handle special cases:** zero, infinity, NaN.
3. **Align exponents:** shift right the mantissa of the operand with the smaller exponent by the exponent difference $d = |E_A - E_B|$.
4. **Add or subtract** the mantissas according to the effective operation (signs and opcode).

5. **Normalize:** if the sum is ≥ 2 , shift right 1 and increment the exponent; if it is < 1 , shift left by the number of leading zeros and decrement the exponent.
6. **Round** according to the rounding mode (may require renormalization).
7. **Check overflow/underflow.**
8. **Assemble** the result.

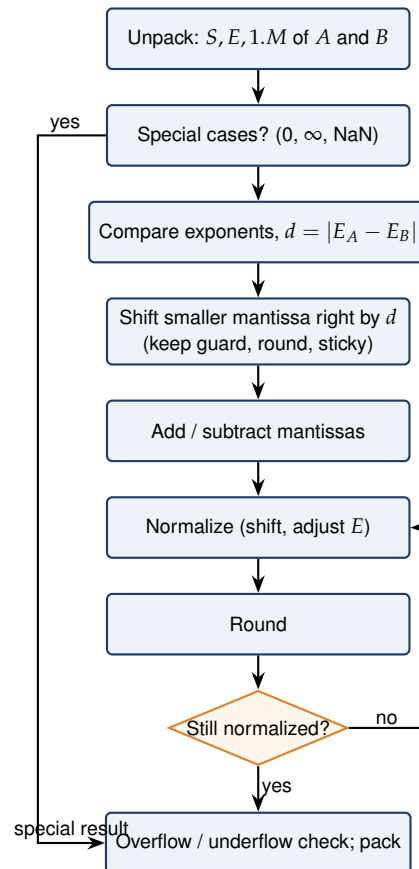


Figure 4.4: Floating-point addition flowchart.

Worked Example 4.8: $7.75 + 2.125$ (single precision)

Step 1: convert. $7.75 = 111.11_2 = 1.1111_2 \times 2^2$; $S = 0$, $E = 129$, $M = 1111000\dots \Rightarrow 0\ 10000001\ 111100000000000000000000\ (0x40F80000)$.

$2.125 = 10.001_2 = 1.0001_2 \times 2^1$; $S = 0$, $E = 128$, $M = 0001000\dots \Rightarrow 0\ 10000000\ 000100000000000000000000\ (0x40080000)$.

Step 2: align to the larger exponent ($E = 129$, i.e. 2^2): $d = 1$. $A = 1.11110_2 \times 2^2$, $B = 0.10001_2 \times 2^2$.

Step 3: add. $1.11110 + 0.10001 = 10.01111_2$.

Step 4: normalize. $10.01111_2 \times 2^2 = 1.001111_2 \times 2^3$; $E = 3 + 127 = 130$.

Result: $0\ 10000010\ 001111000000000000000000 = 0x41E00000$. Check: $1.001111_2 \times 8 = 1001.111_2 = 9.875 = 7.75 + 2.125$. ✓ Exact — no rounding was needed because all bits fitted.

Worked Example 4.9: $5.5 - 3.25$

$5.5 = 101.1_2 = 1.011_2 \times 2^2$ ($E = 129$); $3.25 = 11.01_2 = 1.101_2 \times 2^1$ ($E = 128$).

Align to 2^2 : $B = 0.1101_2 \times 2^2$. **Subtract:** $1.0110_2 - 0.1101_2 = 0.1001_2$. **Normalize:**

$0.1001_2 \times 2^2 = 1.001_2 \times 2^1$, so $E = 1 + 127 = 128$.

Result: $0\ 10000000\ 001000000000000000000000 = 0x40100000 = 1.001_2 \times 2 = 10.01_2 = 2.25$. ✓ Note that subtraction can require a *left* shift by several positions (here 1); in the worst case (“massive cancellation”) the normalizer must shift by up to 24 positions, which is why FP adders contain a **leading-zero counter/anticipator**.

Worked Example 4.10: $12.5 + (-7.25)$ in half precision

Convert. $12.5 = 1100.1_2 = 1.1001_2 \times 2^3$: $S = 0$, $E = 3 + 15 = 18 = 10010_2$, $M = 1001000000$; $A = 0\ 10010\ 1001000000\ (0x4A40)$.

$-7.25 = -111.01_2 = -1.1101_2 \times 2^2$: $S = 1$, $E = 17 = 10001_2$, $M = 1101000000$; $B = 1\ 10001\ 1101000000\ (0xC740)$.

Align to 2^3 : $B = -0.11101_2 \times 2^3$.

Subtract magnitudes (signs differ):

$$\begin{array}{r} 1.10010 \\ -0.11101 \\ \hline 0.10101 \end{array}$$

Normalize: $0.10101_2 \times 2^3 = 1.0101_2 \times 2^2$, $E = 18 - 1 = 17 = 10001_2$.

Result: sign of the larger magnitude (+): $0\ 10001\ 0101000000 = 0x4540 = 1.0101_2 \times 4 = 101.01_2 = 5.25$. ✓

Instructor's note: the corresponding slide shows the intermediate difference as 0.10111_2 and the final pattern $0\ 10001\ 0111000000$ (which decodes to 5.75); the borrow in the third fraction position was dropped. The careful subtraction above gives the correct 5.25.

4.6 Floating-point multiplication

4.6.1 Algorithm

For $A \times B$:

1. Handle special cases (zero, infinity, NaN; $0 \times \infty = \text{NaN}$).
2. Result sign $S = S_A \oplus S_B$.
3. Result exponent $E = E_A + E_B - \text{bias}$ (each stored exponent contains one bias; the sum contains two, so subtract one).
4. Multiply significands $(1.M_A) \times (1.M_B)$: a $24 \times 24 \rightarrow 48$ -bit product for single precision, in $[1, 4)$.
5. Normalize: if the product is ≥ 2 , shift right 1 and increment E .
6. Round, check overflow/underflow, assemble.

Worked Example 4.11: 3.5×2.25

$3.5 = 11.1_2 = 1.11_2 \times 2^1$ ($E = 128$, $M = 1100\dots$); $2.25 = 10.01_2 = 1.001_2 \times 2^1$ ($E = 128$, $M = 0010\dots$).

Sign: $0 \oplus 0 = 0$. Exponent: $128 + 128 - 127 = 129$. Significands:

$$1.11_2 \times 1.001_2 = 1.11_2 \times (1 + 0.001_2) = 1.11_2 + 0.00111_2 = 1.11111_2.$$

$1.11111_2 < 2$, no normalization. Result: $0\ 10000001\ 111110000000000000000000 = 0x40FC0000 = 1.11111_2 \times 4 = 111.111_2 = 7.875 = 3.5 \times 2.25$. ✓

Worked Example 4.12: 1.5×1.75 (needs normalization)

$1.5 = 1.1_2 \times 2^0$, $1.75 = 1.11_2 \times 2^0$, both $E = 127$. Sign 0; exponent $127 + 127 - 127 = 127$; significands $1.1_2 \times 1.11_2 = 10.101_2$. Since $10.101_2 \geq 2$: $10.101_2 = 1.0101_2 \times 2^1$, exponent $\rightarrow 128$. Result $0\ 10000000\ 010100000000000000000000 = 0x40280000 = 2.625$. ✓

Worked Example 4.13: $\pi \times e$ with rounding

Rounded to single precision, $\pi \approx 3.14159 \mapsto 0x40490FD0$ ($1.10010010000111111010000_2 \times 2^1$) and $e \approx 2.71828 \mapsto 0x402DF84D$ ($1.01011011111100001001101_2 \times 2^1$). Sign 0; exponent $128 + 128 - 127 = 129$.

The 24×24 -bit significand product is a 48-bit number $\approx 2.13493_{10}$ (≥ 2), so it is normalized by one right shift and the exponent becomes 130. The 48-bit product must now be rounded to 24 bits: the 24 discarded bits are not all zero, so the result is *inexact*. Rounding to nearest gives

$$\pi \times e \mapsto 0x4108A2B3 = 8.5397215 \dots \quad (\text{exact } 8.5397212652).$$

The relative error, 3×10^{-8} , is below half an ULP ($2^{-24} \approx 6 \times 10^{-8}$) — the guarantee of correct rounding.

4.7 Floating-point division

For $A \div B$: sign $S_A \oplus S_B$; exponent $E_A - E_B + \text{bias}$ (the subtraction cancels the bias, so it must be added back); significand quotient $(1.M_A)/(1.M_B) \in (0.5, 2)$; if the quotient is < 1 , shift left 1 and decrement the exponent. Division by zero produces $\pm\infty$ (and raises the divide-by-zero flag); ∞/∞ and $0/0$ produce NaN.

Worked Example 4.14: $7.5 \div 2.5$

$7.5 = 111.1_2 = 1.111_2 \times 2^2$ ($E = 129$); $2.5 = 10.1_2 = 1.01_2 \times 2^1$ ($E = 128$). Sign 0; exponent $129 - 128 + 127 = 128$. Significands: $\frac{1.111_2}{1.01_2} = \frac{1111_2}{1010_2} = \frac{15}{10} = 1.5 = 1.1_2$. (Equivalently by long division $1111_2 \div 101_2 = 11_2$ remainder 0, then scale.) $1.1_2 \geq 1$: no normalization. Result $0\ 10000000\ 100000000000000000000000 = 0x40400000 = 3.0$. ✓

Worked Example 4.15: $3.0 \div 8.0$

$3.0 = 1.1_2 \times 2^1$ ($E = 128$); $8.0 = 1.0_2 \times 2^3$ ($E = 130$). Exponent $128 - 130 + 127 = 125$; significand $1.1_2/1.0_2 = 1.1_2$. Result $0\ 01111101\ 100000000000000000000000 = 0x3EC00000 = 1.1_2 \times 2^{-2} = 0.375$. ✓

Advanced Insight: How hardware actually divides

Hardware rarely performs long division bit by bit. Two families dominate: **digit recurrence** (SRT division, radix 4 or 16, producing 2–4 quotient bits per cycle using a small lookup table — the table whose missing entries caused the 1994 Pentium FDIV bug) and **functional iteration** (Newton–Raphson or Goldschmidt), which uses the multiplier: to compute $1/b$, iterate $x_{i+1} = x_i(2 - bx_i)$; the number of correct bits *doubles* each iteration. Starting from an 8-bit table estimate, 2 iterations give 32 bits and 3 give 64. This is why division latency (10–20 cycles) is several times that of multiplication (3–5 cycles), and why compilers replace division by a constant with multiplication by its reciprocal.

4.8 Signs, rounding and precision

4.8.1 Sign handling

IEEE 754 is inherently signed: there is no unsigned floating-point format, and the range is symmetric about zero (unlike two's-complement integers, whose negative range is one larger). For addition, if the signs are equal the magnitudes are added and the sign kept; if they differ, the smaller magnitude is subtracted from the larger and the result takes the sign of the larger: $(-3.5) + 2.1 = -(3.5 - 2.1) = -1.4$. For multiplication and division the sign is $S_A \oplus S_B$.

4.8.2 Rounding modes

IEEE 754 defines five rounding-direction attributes:

1. **Round to nearest, ties to even** (default, RNE): round to the closest representable value; if exactly halfway, choose the one with an even (0) LSB. Examples with integers: $2.5 \rightarrow 2$, $3.5 \rightarrow 4$.
2. **Round toward zero** (RTZ, truncation): $2.7 \rightarrow 2$, $-2.7 \rightarrow -2$.
3. **Round toward $+\infty$** (RUP, ceiling): $2.1 \rightarrow 3$, $-2.7 \rightarrow -2$.
4. **Round toward $-\infty$** (RDN, floor): $2.7 \rightarrow 2$, $-2.1 \rightarrow -3$.
5. **Round to nearest, ties away from zero** (RMM): like RNE but $2.5 \rightarrow 3$.

Ties-to-even is the default because it is *unbiased*: over many operations, ties round up and down equally often, so errors do not accumulate in one direction. RISC-V encodes these five modes in the 3-bit `rm` field of FP instructions and the `frm` CSR.

Advanced Insight: Guard, round and sticky bits

To round correctly, the hardware does *not* need the full infinitely precise result — three extra bits suffice. During alignment and normalization the adder keeps:

- the **guard** bit G (first bit beyond the LSB),
- the **round** bit R (second bit beyond),
- the **sticky** bit S (OR of all remaining bits shifted out).

Round-to-nearest-even then increments the significand iff $G \cdot (R + S + \text{LSB}) = 1$.

Example (4-bit significand $1.xxx$). Significand after alignment and addition: $1.011 \mid 101\dots$ ($G = 1, R = 0, S = 1$): more than half an ULP $\Rightarrow$ round up to 1.100 . If instead $G = 1, R =$

$0, S = 0$ (exactly half) and $LSB = 1$: tie, round to even $\Rightarrow 1.100$; with $LSB = 0$ the tie rounds down and the value stays. Without the sticky bit, the hardware could not distinguish “exactly half” from “slightly more than half”.

4.8.3 Loss of precision

Worked Example 4.16: Absorption: $2^{24} + 1$

$A = 1.0 \times 2^{24} = 16\,777\,216$ and $B = 1.0$ are both exact in single precision. Aligning B to A 's exponent shifts it right 24 places — exactly to the guard position. The exact sum $16\,777\,217$ lies halfway between $16\,777\,216$ and $16\,777\,218$; ties-to-even selects $16\,777\,216$ (even mantissa). So $A + B = A$: B is *absorbed*. Consequence: a single-precision loop counter incremented by 1 stops counting at 2^{24} .

Worked Example 4.17: Catastrophic cancellation

Let $A = 1.0000001$ and $B = 1.0000000$; we want $A - B = 10^{-7}$. In single precision A rounds to the nearest float, $1 + 2^{-23} = 1.00000011920929$ (its representation error is only 1.9×10^{-8} , a relative error of 2×10^{-8}). The subtraction itself is *exact*: $A - B = 2^{-23} = 1.1920929 \times 10^{-7}$. Yet compared with the intended 10^{-7} the result is 19% wrong. Nothing went wrong in the subtraction — it simply removed the leading, agreeing digits and promoted the tiny representation error of A to the most significant position. Whenever nearly equal, previously rounded quantities are subtracted, the relative error of the result can be enormous; a classic cure is algebraic rearrangement, e.g. computing $\sqrt{x+1} - \sqrt{x}$ as $1/(\sqrt{x+1} + \sqrt{x})$.

Worked Example 4.18: Rounding error growth in multiplication

Two 24-bit significands produce a 48-bit product that must be rounded back to 24 bits; each rounding contributes a relative error up to $u = 2^{-24} \approx 5.96 \times 10^{-8}$. A product of n factors accumulates a worst-case relative error of about nu : 10^6 multiplications $\Rightarrow$ 6% worst case, though random-walk behaviour typically gives $\sqrt{n}u \approx 6 \times 10^{-5}$.

Practical pitfalls

- Never test floats with $==$. Use $|a - b| < \epsilon \max(|a|, |b|)$ with a *relative* tolerance.
- Do not use floating-point loop counters (absorption, accumulation of 0.1 errors).
- Use decimal or scaled-integer arithmetic for money.
- Choose float vs double deliberately; test extreme values (very large, very small, subnormal, NaN, ± 0).
- Remember that $(a + b) + c \neq a + (b + c)$ in general: compilers do not reorder FP sums unless told (`-ffast-math`), which also makes parallel reductions non-deterministic.

Advanced Insight: Kahan compensated summation

Summing n numbers naively has error growth $O(nu)$. Kahan's algorithm carries a compensation term:

```

float sum = 0.0f, c = 0.0f;
for (int i = 0; i < n; i++) {
    float y = x[i] - c;      // corrected next term
    float t = sum + y;      // low bits of y are lost here...
    c = (t - sum) - y;      // ...and recovered here
    sum = t;
}

```

The error becomes $O(u)$, independent of n . Adding $0.1f$ ten million times naively gives 1 087 937 in single precision (an 8.8% error); with Kahan summation the result is 1 000 000 to within one ULP.

4.9 Conversion between integers and floating point

4.9.1 Integer to floating point

Converting a 32-bit signed integer to single precision (RISC-V `fcvt.s.w`):

1. Judge special numbers (zero converts to $+0$).
2. Record the sign; if negative, convert to magnitude (two's complement).
3. **Detect the leading one** (leading-zero count z); the exponent is $31 - z$, i.e. $E = 31 - z + 127$.
4. **Normalize**: shift the magnitude left by $z + 1$ to remove the leading one; the next 23 bits are the mantissa.
5. **Round** the remaining bits (an integer with more than 24 significant bits cannot be represented exactly).
6. Pack sign, exponent, mantissa.

The lecture's pipeline diagram implements exactly these stages (C1: special-number judgment and complement; C2: leading-zero detection; C3: normalization; C4: rounding, extension and result generation), sharing hardware between single and double precision.

Worked Example 4.19: Converting 123 456 789 to single precision

$123\,456\,789 = 0x075BCD15 = 111\,0101\,1011\,1100\,1101\,0001\,0101_2$ (27 significant bits). Leading one at bit 26: true exponent 26, $E = 26 + 127 = 153 = 10011001_2$. The 26 bits after the leading one are $110101101111100110100010101$; keep 23: 11010110111100110100010 with remainder 101 ($G = 1, R = 0, S = 1$) $\Rightarrow$ round up: 11010110111100110100011 .

$$0\ 10011001\ 11010110111100110100011 = 0x4CEB79A3 = 123\,456\,792.$$

The conversion error is 3: integers above 2^{24} are spaced 2, 4, 8, ... apart in single precision (here spacing $2^{26-23} = 8$).

4.9.2 Floating point to integer

The reverse (`fcvt.w.s`) extracts the true exponent $e = E - 127$; if $e < 0$ the magnitude is < 1 and rounds to 0 or ± 1 ; if $e > 30$ the value overflows a 32-bit integer and the result saturates (RISC-V returns $2^{31} - 1$ or -2^{31} and sets the invalid flag; NaN converts to $2^{31} - 1$). Otherwise

the significand $1.M$ is shifted by $e - 23$ positions, rounded according to the mode (C's cast uses round-toward-zero), and negated if $S = 1$.

Worked Example 4.20: $-5.625 \rightarrow \text{integer}$

$0xC0B40000$: $e = 2$, significand 1.01101_2 ; shift: 101.101_2 ; RTZ gives 5, negate: -5 . Round-to-nearest gives -6 (since $.101_2 = 0.625 > 0.5$). RISC-V lets the instruction choose: `fcvt.w.s a0, fa0, rtz` versus `rne`.

4.10 Hardware floating-point units

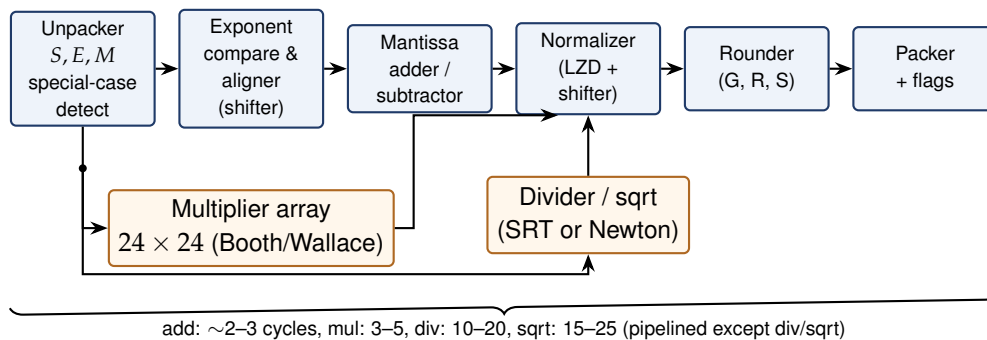


Figure 4.5: Structure of a floating-point unit. The add path (top) and the multiply/divide paths share unpacking, normalization, rounding and packing logic.

The FPU components are the **unpacker** (extracts fields, detects special cases), **aligner** (barrel shifter for addition), **adder/subtractor**, **multiplier** (dedicated significand multiplier), **divider** (iterative), **normalizer** (leading-zero detection and shift), **rounder** and **packer**. Typical latencies: addition/subtraction 2–3 cycles, multiplication 3–5, division 10–20 (iterative, usually not pipelined), square root 15–25. Modern optimizations: full pipelining (a new add or multiply every cycle), multiple parallel execution units, and **fused multiply-add (FMA)**.

Advanced Insight: Fused multiply-add

FMA computes $a \times b + c$ with a *single* rounding at the end. Benefits: higher throughput (two flops per instruction, the basis of every quoted FLOPS figure), and higher accuracy, because the full 48-bit product participates in the addition. Example: $a = b = 1 + 2^{-12}$, $c = -(1 + 2^{-11})$ in single precision. Exactly, $a^2 + c = 1 + 2^{-11} + 2^{-24} - 1 - 2^{-11} = 2^{-24}$. Without FMA, a^2 rounds to $1 + 2^{-11}$ (the 2^{-24} term is a tie and rounds to even), so the result is 0. With FMA the result is exactly $2^{-24} \approx 5.96 \times 10^{-8}$. RISC-V provides `fmadd/fmsub/fnmadd/fnmsub` (the R4-type format with three source registers).

4.11 Modern reduced-precision formats

Machine learning tolerates low precision but needs wide range, which has produced a family of new formats.

bfloat16 is simply the top 16 bits of an FP32 number: conversion is truncation (or rounding) of the low 16 mantissa bits, and the range is unchanged, so overflow problems of FP16 training disappear. **Posits** (Gustafson, 2017) use a variable-length “regime” field so that numbers near 1 get more precision and extreme numbers get more range (tapered precision); a 16-bit posit often

Table 4.3: Floating-point formats used in modern accelerators.

Format	Bits (S/E/M)	Max value	Machine ϵ	Typical use
FP32 (binary32)	1/8/23	3.4×10^{38}	2^{-23}	training master weights
TF32 (NVIDIA)	1/8/10	3.4×10^{38}	2^{-10}	tensor-core matmul inputs
FP16 (binary16)	1/5/10	65 504	2^{-10}	inference, graphics
bfloat16 (Google)	1/8/7	3.4×10^{38}	2^{-7}	training (same range as FP32)
FP8 E4M3	1/4/3	448	2^{-3}	inference weights/activations
FP8 E5M2	1/5/2	57 344	2^{-2}	training gradients

matches FP32 accuracy on well-scaled workloads, but posits are not yet in mainstream hardware.

Worked Example 4.21: FP32 to bfloat16

π in FP32 is `0x40490FDB`. Truncating to the upper half gives `0x4049` = $1.1001001_2 \times 2^1 = 3.140625$. Round-to-nearest-even looks at the discarded half `0x0FDB` < `0x8000`, so it also gives `0x4049`. Relative error 3.1×10^{-4} — adequate for neural-network weights, inadequate for physics.

Chapter summary

- IEEE 754: value = $(-1)^S(1.M)2^{E-\text{bias}}$; bias = $2^{k-1} - 1$ (15, 127, 1023, 16383). Exponent all-0 = zero/subnormal; all-1 = ∞ /NaN.
- Add/sub: align (shift smaller right), add/subtract, normalize, round. Mul: XOR signs, add exponents minus bias, multiply significands. Div: subtract exponents plus bias, divide significands.
- Round-to-nearest-even with guard, round and sticky bits is the default; errors appear as absorption, cancellation and accumulation.
- Integer $\leftrightarrow$ float conversion needs leading-zero detection, shifting and rounding; integers above 2^{24} are not exact in single precision.
- FPUs pipeline add/mul (2–5 cycles), iterate div/sqrt (10–25 cycles); FMA and reduced-precision formats (bfloat16, FP8) dominate modern accelerators.

Exercises

E4.1 Encode -13.625 in single and half precision.

Answer: $-1101.101_2 = -1.101101_2 \times 2^3$: single `0xC15A0000`; half `0xCAD0`.

E4.2 Decode `0x3F400000` and `0x7F7FFFFF`.

Answer: 0.75; the largest finite float $(2 - 2^{-23}) \times 2^{127} \approx 3.4028 \times 10^{38}$.

E4.3 Compute $6.5 + 0.375$ step by step in single precision.

Answer: $6.5 = 1.101_2 \times 2^2$, $0.375 = 1.1_2 \times 2^{-2}$ aligned to $0.00011_2 \times 2^2$; sum $1.10111_2 \times 2^2 = 6.875$, `0x40DC0000`.

E4.4 Compute $(-2.5) \times 0.75$ and $10 \div 0.625$.

Answer: -1.875 (`0xBFF00000`); 16 (`0x41800000`).

E4.5 In the 8-bit format of this chapter, list all representable values in $[4, 8)$ and compute $3.75 + 0.5$.

Answer: $4, 4.5, \dots, 7.5$; $3.75 + 0.5 = 4.25$ is a tie between 4 and 4.5: RNE gives 4 (even mantissa 000).

E4.6 Why does $x - y = 0 \iff x = y$ fail without subnormals? Give an 8-bit-format example.

Answer: $x = 1.001_2 \times 2^{-6}$, $y = 1.000_2 \times 2^{-6}$: $x - y = 2^{-9}$, which is subnormal; flushed to zero it would give 0 although $x \neq y$.

E4.7 What is the largest integer n such that all integers in $[0, n]$ are exactly representable in double precision?

Answer: 2^{53} .

E4.8 Convert the integer $-1\,000\,000$ to single precision.

Answer: $1\,000\,000 = 1.11101000010010000000_2 \times 2^{19}$: `0xC9742400`.

5

Instruction Set Architecture and RISC-V

The instruction set architecture is the most important interface in computing: it is the vocabulary in which every program is ultimately written and the specification every processor must implement. This chapter studies ISA design systematically — what an instruction contains, where operands live, which data types and addressing modes to provide, the RISC versus CISC debate, and the structure of several historic and commercial ISAs (stack machines, Alpha, MIPS, ARM, LC-3b). It then gives a complete, practical treatment of RV32I, the base of the RISC-V ISA used for the rest of the course.

5.1 What an ISA specifies

Definition 5.1: Instruction set architecture

The ISA defines the **architectural state** visible to software (registers, PC, memory, control/status registers) and, for every instruction, how that state is transformed. It specifies instruction formats and encodings, operand locations and addressing modes, data types, the memory model, privilege modes, and the handling of exceptions and interrupts.

5.1.1 Elements of an ISA

Instructions. Each instruction has an **opcode** (what to do) and **operand specifiers** (on what). The addressing modes answer “how is the operand obtained?” — from a register, from the instruction itself (immediate), or from memory at an address computed in some way.

Data types. A data type is a representation of information for which there are instructions that operate on the representation: integers, floating point, characters, binary-coded decimal, and in some ISAs much richer types. The VAX had *INSQUE* and *REMQUE* instructions that insert into and remove from a *doubly linked list*, and *FINDFIRST* for bit vectors; x86 has the *SCAN* string instruction and stack *PUSH/POP*. The Intel 432 (1981) supported *objects* and capabilities in hardware; early RISC machines supported only integers.

Benefit and cost of high-level data types. Richer types narrow the semantic gap: the compiler emits fewer instructions and code is smaller. But they make the hardware more complex, they may not match the exact semantics a language needs (so the compiler cannot use them), and they freeze one implementation choice into the ISA forever. This is the tension between the *compiler/programmer* viewpoint and the *microarchitect* viewpoint.

Addressing modes. More addressing modes let high-level constructs map better to the machine: some accesses are better expressed with a dedicated mode, reducing instruction count and code size — think of array accesses (auto-increment mode), indirection (pointer chasing), or sparse-matrix accesses. The disadvantages: more work for the compiler (which mode to use?) and more work for the microarchitect (more paths to implement and verify).

Table 5.1: Common addressing modes. $M[x]$ denotes the memory word at address x .

Mode	Example	Meaning
Register	<code>add r1, r2, r3</code>	$r1 \leftarrow r2 + r3$
Immediate	<code>addi r1, r2, 5</code>	$r1 \leftarrow r2 + 5$
Displacement (base + offset)	<code>lw r1, 8(r2)</code>	$r1 \leftarrow M[r2 + 8]$
Register indirect	<code>lw r1, (r2)</code>	$r1 \leftarrow M[r2]$
Indexed	<code>lw r1, (r2+r3)</code>	$r1 \leftarrow M[r2 + r3]$
Absolute (direct)	<code>lw r1, (1000)</code>	$r1 \leftarrow M[1000]$
Memory indirect	<code>lw r1, @(r2)</code>	$r1 \leftarrow M[M[r2]]$
Auto-increment	<code>lw r1, (r2)+</code>	$r1 \leftarrow M[r2]; r2 \leftarrow r2 + 4$
Auto-decrement	<code>lw r1, -(r2)</code>	$r2 \leftarrow r2 - 4; r1 \leftarrow M[r2]$
PC-relative	<code>beq r1, r2, L</code>	if equal, $PC \leftarrow PC + \text{offset}$

Types of instructions.

- **Operate instructions** process data: arithmetic and logical operations. They fetch operands, compute a result and store it, with implicit sequential control flow.
- **Data movement instructions** move data between memory, registers and I/O devices, also with implicit sequential control flow.
- **Control flow instructions** change the sequence of instructions executed: branches, jumps, calls, returns, traps.

Other ISA concerns.

- **Privilege modes:** user versus supervisor (and machine, hypervisor); which instructions may each execute?
- **Exception and interrupt handling:** what happens when something goes wrong in an instruction (exception), or an external device requests the processor (interrupt)? Vectored versus non-vectored interrupts.
- **Virtual memory:** each program has the illusion of the entire memory space, which may exceed physical memory (Unit 4).
- **Access protection:** which program may read, write or execute which memory.

5.2 Where are the operands? 0-, 1-, 2- and 3-address machines

The most fundamental classification of ISAs is by how many operands an ALU instruction names explicitly.

- **0-address: stack machine** (`op, push A, pop A`). Operands are implicitly on top of a stack.

- **1-address: accumulator machine** (op ACC, ld A, st A). One operand is the implicit accumulator register.
- **2-address machine** (op S, D): one operand is both source and destination (x86: add eax, ebx).
- **3-address machine** (op S1, S2, D): sources and destination are separate (RISC-V, MIPS, ARM).

Orthogonally, operands may come from memory (*memory–memory*, *register–memory*) or only from registers with separate load/store instructions (*load–store*, all RISC).

Worked Example 5.1: $Z = (A + B) \times (C - D)$ on four machine types

Stack (0-addr)	Accumulator (1-addr)	Register–memory (2-addr)	Load–store (3-addr)
push A	ld C	mov r1, A	lw x1, A
push B	sub D	add r1, B	lw x2, B
add	st T	mov r2, C	lw x3, C
push C	ld A	sub r2, D	lw x4, D
push D	add B	mul r1, r2	add x1,x1,x2
sub	mul T	mov Z, r1	sub x3,x3,x4
mul	st Z		mul x1,x1,x3
pop Z			sw x1, Z
8 instr., 5 data mem- ory refs	7 instr., 7 data refs (incl. temp T)	6 instr., 5 data refs	8 instr., 5 data refs

Code size with assumed encodings (stack: 1-byte ops, 3-byte push/pop with 16-bit address; accumulator: 3 bytes each; reg–mem: 4 bytes each; load–store: 4 bytes each): stack $5 \times 3 + 3 \times 1 = 18$ B; accumulator 21 B; reg–mem 24 B; load–store 32 B.

The load–store program is the *largest*, but every instruction is simple, has a fixed length, and touches memory at most once, which makes it easy to pipeline; and once $A \dots D$ are in registers they can be reused without further memory traffic. This is the RISC bet: more, simpler instructions that each run faster.

5.3 Sequential versus dataflow execution

In the von Neumann (*sequential*) model, instructions execute in program order, one after another, directed by the PC. In the **dataflow** model there is no PC: an instruction (a node) fires as soon as all its input operands are available, and sends its result as *tokens* along arcs to its consumers.

Worked Example 5.2: Parallelism exposed by dataflow

Assume each operation takes one time unit and unlimited functional units. Sequentially the five operations take 5 units. In the dataflow graph, $\{+, \times\}$ fire at $t = 1$, $\{-, +\}$ at $t = 2$ and $\times$ at $t = 3$: 3 units, a speedup of $5/3 = 1.67$. The critical path length (3) bounds the execution time; the total work (5) divided by the critical path gives the *average parallelism* $5/3$.

A dataflow node is described by its operation, the destinations of its result and slots for its arguments (* | R | ARG1 | R | ARG2 | Dest. of result, where R are ready bits). Dataflow

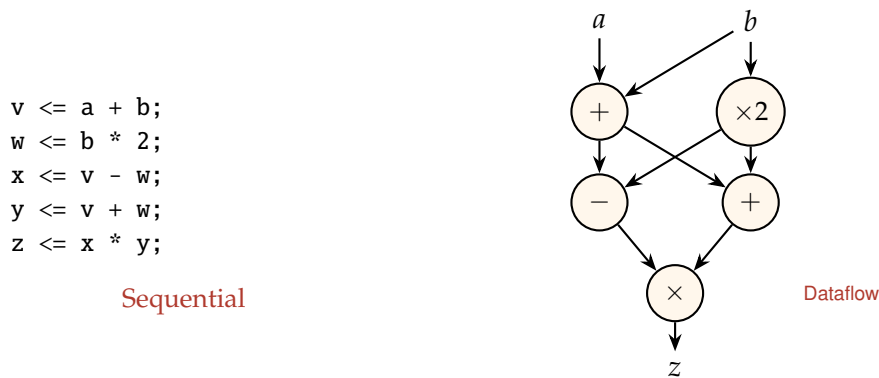


Figure 5.1: The same computation as a sequential program and as a dataflow graph.

also needs *conditional* nodes (a branch node BR routes its input token to the T or F output depending on a Boolean token), *relational* nodes ($>$ producing a Boolean token) and *barrier synchronization* nodes (all outputs fire only when all inputs have arrived). The lecture’s question slide asks what a dataflow graph with Copy, $>$, BR, DEC and $\times$ nodes computes: it is a loop that repeatedly multiplies an accumulator by a decrementing counter while the counter is positive — the **factorial** $n!$.

Advanced Insight: Dataflow lives on inside out-of-order processors

Pure dataflow machines (MIT Tagged-Token, Manchester Dataflow, 1980s) struggled with the overhead of token matching and with poor locality. But the idea won anyway: every modern out-of-order core is a *restricted dataflow machine* hidden behind a sequential ISA. Instructions are fetched in order, renamed, and placed in reservation stations, where they wait for their operands (“ready bits”) and fire when ready — exactly a dataflow node. A *reorder buffer* then retires them in program order so that the sequential illusion is preserved for exceptions. Unit 3 develops the in-order pipeline that is the first step on this road.

5.4 The stack (PUSH/POP) processor

A **stack processor** is a stack-based architecture in which instructions operate implicitly on the top of the stack (TOS). Most instructions have no explicit operand address; examples are PUSH X, POP Y, ADD, SUB.

5.4.1 Stack organization

The **stack pointer (SP)** points to the top element; the stack grows and shrinks with PUSH and POP. With a stack that grows downward in memory:

- **PUSH X:** decrement SP; store the operand at the new location ($SP \leftarrow SP - 1$; $M[SP] \leftarrow X$).
- **POP Y:** read the value at SP; increment SP ($Y \leftarrow M[SP]$; $SP \leftarrow SP + 1$).
- **ADD:** pop two operands, push their sum.

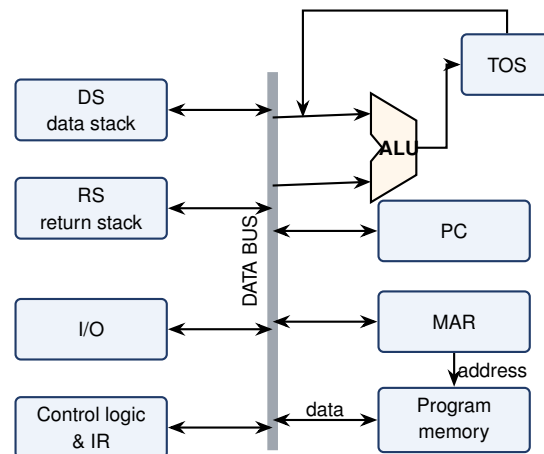


Figure 5.2: A stack processor: data stack (DS), return stack (RS), top-of-stack register (TOS), ALU, PC, MAR and program memory around one data bus.

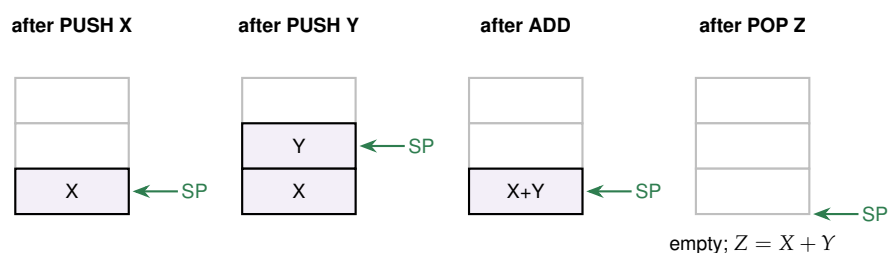


Figure 5.3: Stack evolution while executing PUSH X; PUSH Y; ADD; POP Z for $Z = X + Y$.

Worked Example 5.3: Evaluating an expression on a stack machine

Evaluate $Z = (3 + 4) \times (10 - 2) / 2$. Convert to postfix (reverse Polish): $3\ 4\ +\ 10\ 2\ -\ *\ 2\ /\$. Each operand becomes a PUSH; each operator pops two and pushes one.

Instruction	Stack (top at right)	Max depth
PUSH 3	3	1
PUSH 4	3 4	2
ADD	7	
PUSH 10	7 10	
PUSH 2	7 10 2	3
SUB	7 8	
MUL	56	
PUSH 2	56 2	
DIV	28	
POP Z	—	

10 instructions, maximum depth 3. If each PUSH/POP of a memory variable costs a memory access and the stack itself lives in memory, every ALU operation also reads two stack words and writes one: $4 \times 3 = 12$ extra stack accesses — the “excessive memory traffic” limitation. Real stack machines keep the top few stack entries in registers (TOS, NOS) to avoid it.

Instruction format. Most instructions are *zero-address*: operands are implied (always from the stack), e.g. ADD, SUB, MUL. Some are *one-address*: PUSH A, POP B.

Advantages. Simple, short instruction format (fewer bits); no need to specify operand addresses; natural for expression evaluation in postfix notation; trivial compilers.

Limitations. Excessive memory traffic (frequent PUSH/POP); the single TOS is a bottleneck that makes instruction-level parallelism hard to exploit (every instruction depends on the previous one through the stack); less efficient than register machines in practice.

Where stack machines live today. The Java Virtual Machine, the .NET CLR, WebAssembly and Python bytecode are all stack ISAs — *virtual* ones, chosen for compact code and easy compilation; JIT compilers translate them into register-machine code for execution.

5.5 RISC versus CISC

5.5.1 Complex and simple instructions

A **complex instruction** does a lot of work, e.g. many operations: insert into a doubly linked list, compute an FFT, copy a string. A **simple instruction** does a small amount of work; it is a primitive from which complex operations are built: add, XOR, multiply.

Advantages of complex instructions.

- + Denser encoding $\Rightarrow$ smaller code $\Rightarrow$ better memory utilization, less off-chip bandwidth, better instruction-cache hit rate (better packing of instructions).
- + Simpler compiler: no need to optimize small instructions as much.

Disadvantages of complex instructions.

- Larger chunks of work $\Rightarrow$ the compiler has less opportunity to optimize (limited in the fine-grained optimizations it can do).
- More complex hardware: translation from a high level to control signals, and optimization, must be done by hardware.

5.5.2 The semantic gap

Where should the ISA be placed? Closer to the high-level language (HLL) gives a **small semantic gap** and complex instructions; closer to the hardware control signals gives a **large semantic gap** and simple instructions.

- **RISC:** reduced instruction set computer — fixed-length instructions, load-store, many registers, few addressing modes, simple decoding, designed for pipelining (MIPS, SPARC, Alpha, ARM, RISC-V).
- **CISC:** complex instruction set computer — variable-length instructions, memory operands, many addressing modes, complex instructions (FFT, QUICKSORT, POLY, FP instructions; the VAX INDEX instruction did array access with bounds checking).

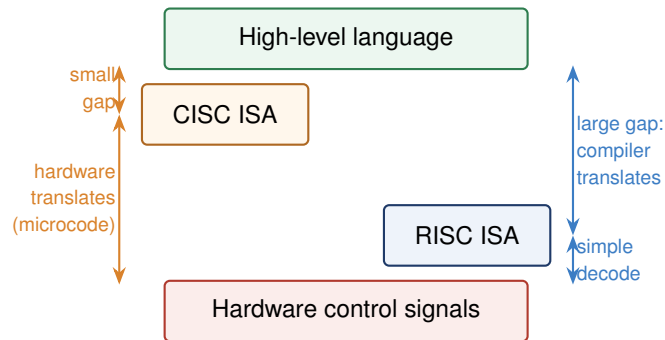


Figure 5.4: The semantic gap and the placement of the ISA.

Worked Example 5.4: Iron law: does RISC or CISC win?

A benchmark compiles to $N_C = 1.0 \times 10^9$ instructions on a CISC machine with $\text{CPI}_C = 2.4$ and to $N_R = 1.35 \times 10^9$ instructions on a RISC machine with $\text{CPI}_R = 1.1$. The CISC machine runs at 3.0 GHz, the RISC at 2.6 GHz (it is a smaller, lower-power design).

$$T_C = \frac{1.0 \times 10^9 \times 2.4}{3.0 \times 10^9} = 0.80 \text{ s}, \quad T_R = \frac{1.35 \times 10^9 \times 1.1}{2.6 \times 10^9} = 0.571 \text{ s}.$$

RISC is $1.40\times$ faster despite executing 35% more instructions. But if the CISC front end translates each instruction into ≈ 1.3 RISC-like micro-ops and runs them on an equally efficient out-of-order core (CPI per micro-op 0.85), its effective CPI becomes $1.3 \times 0.85 = 1.1$ and $T_C = 0.367 \text{ s}$. This is exactly what modern x86 processors do: *CISC outside, RISC inside*. The ISA's influence on performance has shrunk; its influence on decoder complexity, power and verification effort has not.

5.6 A tour of real ISAs

5.6.1 Alpha

DEC Alpha (1992) was a clean 64-bit RISC with four 32-bit formats, all with a 6-bit opcode in bits 31–26:

31	26	25	21	20	16	15	13	12	5	4	0
Opcode	Number										
Opcode	RA	Disp									
Opcode	RA	RB	Disp								
Opcode	RA	RB	SBZ	0	Function			RC			

(PALcode, branch, memory and operate formats respectively.) The operate format has a variant where bit 12 = 1 replaces RB with an 8-bit literal (LIT) — an example of **bit steering**, discussed below.

5.6.2 MIPS

MIPS has three formats; the 6-bit opcode and three 5-bit register fields are always in the same place, which allows register reads to begin before decoding completes.

31	26	25	21	20	16	15	11	10	6	5	0
0		rs		rt		rd		shamt		funct	
opcode		rs		rt		immediate					
opcode		immediate (jump target)									

Worked Example 5.5: Encoding add \$t0, \$s1, \$s2 in MIPS

\$t0 = 8, \$s1 = 17, \$s2 = 18; R-type with funct = 0x20 (add):

$$\underbrace{000000}_{op} \underbrace{10001}_{rs=17} \underbrace{10010}_{rt=18} \underbrace{01000}_{rd=8} \underbrace{00000}_{shamt} \underbrace{100000}_{funct} = 0x02324020.$$

5.6.3 ARM (A32)

Every classic 32-bit ARM instruction begins with a 4-bit **condition field** (bits 31–28) — every instruction is *predicated*: it executes only if the condition (EQ, NE, GT, ...) holds on the flags. The next bits select among about a dozen formats: data processing (with a flexible “operand 2” that can be a rotated immediate or a shifted register), multiply, single data transfer with pre/post-indexing and write-back (P, U, W bits), block data transfer with a 16-bit register list (LDM/STM), branch with link, coprocessor operations and software interrupt. ARM is RISC in being load–store but CISC-like in its rich addressing modes and multiple-register transfers; the 64-bit A64 ISA dropped universal predication and LDM/STM.

5.6.4 LC-3b and bit steering

LC-3b is a 16-bit teaching ISA (Patt and Patel) with a 4-bit opcode in bits 15–12, 8 registers (3-bit fields), and byte addressability. Twelve opcodes cover ADD, AND, XOR, BR, JMP, JSR/JSRR, LDB, LDW, LEA, RTI, SHF, STB, STW and TRAP; two opcode values are unused.

Bit steering means that one bit of the instruction selects how the remaining bits are interpreted. In LC-3b ADD:

15	12	11	9	8	6	5	4	3	2	0
0001		DR		SR1		0	00		SR2	
0001		DR		SR1		1	imm5			

Operation: if (bit[5]==0) DR = SR1 + SR2; else DR = SR1 + SEXT(imm5); setcc();

Worked Example 5.6: Encoding LC-3b instructions

(a) ADD R1, R2, R3: 0001 001 010 0 00 011 = 0x1283.

(b) ADD R2, R3, #-3: imm5 of -3 is 11101: 0001 010 011 1 11101 = 0x14FD. The 5-bit immediate covers [−16, 15].

(c) JSR uses bit 11 as the steering bit: if 1, $PC \leftarrow PC' + \text{LSHF}(\text{SEXT}(\text{PCoffset11}), 1)$; if 0 (JSRR), $PC \leftarrow \text{BaseR}$. In both cases $R7 \leftarrow PC'$ (the incremented PC) first, which is the linkage back to the caller.

5.7 RISC-V

5.7.1 What is RISC-V?

RISC-V is an **open-source instruction set architecture** developed at UC Berkeley from 2010. It is a free and open standard (maintained by RISC-V International), with a **modular** design — a small stable base plus optional extensions — and a growing ecosystem of implementations suitable for every class of computing device. Its principles: a simple, clean-slate design; a modular ISA; a frozen base plus optional standard extensions; explicit support for custom extensions.

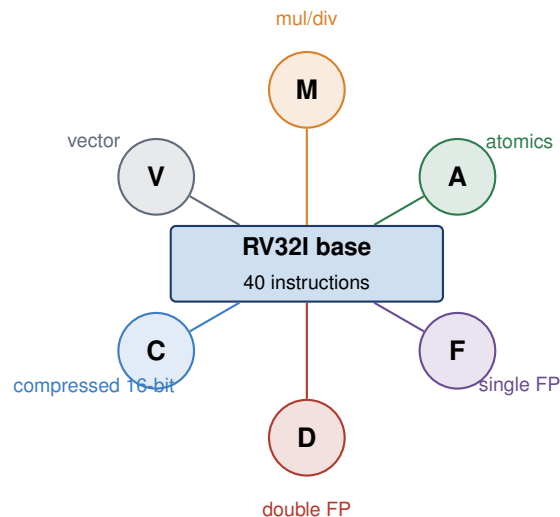


Figure 5.5: RISC-V: a base integer ISA plus standard extensions. “G” = IMAFD (+Zicsr, Zifencei); RV64GC is the common application-processor profile.

Base ISAs. RV32I (32-bit integer base), RV64I (64-bit), RV128I (128-bit, reserved), and RV32E (embedded, 16 registers).

Standard extensions. M (integer multiply/divide), A (atomic memory operations), F (single-precision FP), D (double-precision FP), C (compressed 16-bit instructions), and newer ones: V (vectors), B (bit manipulation), K/Zk (scalar cryptography), H (hypervisor).

5.7.2 RV32I: the 32-bit integer base

- Fixed 32-bit instruction length (16-bit with C).
- 32-bit, byte-addressable address space (4 GB), little-endian.
- 32 integer registers: `x0` is hard-wired to zero; `x1`–`x31` are general purpose. Plus the program counter.
- Load–store architecture: only loads and stores access memory.
- Clean, orthogonal instruction set, minimal hardware requirements, easy to implement and verify, suitable from embedded to high-performance systems.

Memory model. Addresses run from `0x00000000` to `0xFFFFFFFF`; memory is byte addressable and little-endian; aligned accesses are preferred. Loads move memory to a register, stores move a register to memory, at byte, halfword or word granularity; sub-word loads are sign-extended (1b, 1h) or zero-extended (1bu, 1hu).

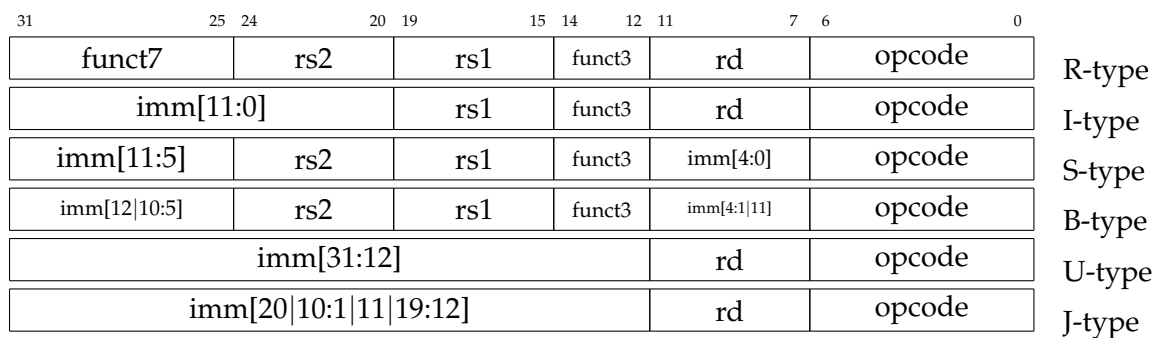
Table 5.2: RV32I integer registers and the standard calling convention (ABI).

Register	ABI name	Description	Saver
x0	zero	hard-wired zero	—
x1	ra	return address	caller
x2	sp	stack pointer	callee
x3	gp	global pointer	—
x4	tp	thread pointer	—
x5–x7	t0–t2	temporaries	caller
x8	s0/fp	saved register / frame pointer	callee
x9	s1	saved register	callee
x10–x11	a0–a1	function arguments / return values	caller
x12–x17	a2–a7	function arguments	caller
x18–x27	s2–s11	saved registers	callee
x28–x31	t3–t6	temporaries	caller

Special registers and state. The **PC** points to the current instruction; it is normally incremented by 4 and modified by branches and jumps. **Control and status registers (CSRs)** are defined mostly in the privileged specification: machine-level (M-mode), supervisor-level (S-mode) and user-level (U-mode) CSRs, performance counters (cycle, instret), and exception/interrupt state (mstatus, mepc, mcause, mtvec).

5.7.3 Instruction formats

All RV32I instructions are 32 bits wide with six basic formats (Figure 5.6).

**Figure 5.6:** The six RV32I instruction formats.

- **R-type:** register–register operations (add, sub, and, ...).
- **I-type:** register–immediate operations and loads (addi, lw, jalr).
- **S-type:** stores (sw); the immediate is split so that rs1/rs2 stay in place.
- **B-type:** conditional branches; a 13-bit, even, PC-relative offset.
- **U-type:** upper-immediate operations (lui, auipc).
- **J-type:** jal; a 21-bit, even, PC-relative offset.

Why the immediates look scrambled

RISC-V keeps *rs1*, *rs2* and *rd* in the *same bit positions* in every format, so register-file reads can start in parallel with decoding. It also places the immediate's *sign bit always in bit 31*, so sign extension can begin before the format is known, and arranges the remaining immediate bits so that each bit of the 32-bit immediate comes from at most two possible instruction-bit positions — minimising the multiplexers in the immediate generator. The apparent scrambling of B and J immediates is a hardware optimization.

5.7.4 Arithmetic and logical instructions

Register–register (R-type)		Register–immediate (I-type)	
<code>add rd,rs1,rs2</code>	$rd = rs1 + rs2$	<code>addi rd,rs1,imm</code>	$rd = rs1 + imm$
<code>sub rd,rs1,rs2</code>	$rd = rs1 - rs2$	<code>slti rd,rs1,imm</code>	$rd = (rs1 < imm)$
<code>sll rd,rs1,rs2</code>	$rd = rs1 \ll rs2[4:0]$	<code>sltiu rd,rs1,imm</code>	unsigned compare
<code>slt rd,rs1,rs2</code>	$rd = (rs1 < rs2)$ signed	<code>xori rd,rs1,imm</code>	$rd = rs1 \oplus imm$
<code>sltu rd,rs1,rs2</code>	unsigned compare	<code>ori rd,rs1,imm</code>	$rd = rs1 imm$
<code>xor rd,rs1,rs2</code>	$rd = rs1 \oplus rs2$	<code>andi rd,rs1,imm</code>	$rd = rs1 \& imm$
<code>srl rd,rs1,rs2</code>	logical right shift	<code>slli rd,rs1,shamt</code>	$rd = rs1 \ll shamt$
<code>sra rd,rs1,rs2</code>	arithmetic right shift	<code>srli rd,rs1,shamt</code>	logical right
<code>or rd,rs1,rs2</code>	$rd = rs1 rs2$	<code>srai rd,rs1,shamt</code>	arithmetic right
<code>and rd,rs1,rs2</code>	$rd = rs1 \& rs2$		

There is no multiplication or division in base RV32I: those require the M extension. There is also no `subi` (use `addi` with a negative immediate), no `mov` (use `addi rd, rs, 0`) and no `nop` (it is `addi x0, x0, 0`) — these are *pseudo-instructions* provided by the assembler.

The ALU supports arithmetic (add, sub, compare: `slt`/`sltu`), logical (AND, OR, XOR) and shift (SLL, SRL, SRA) operations; it receives operands *A*, *B* and an operation code, and produces a result and flags (e.g. zero, used for branches).

5.7.5 Encoding and decoding by hand

Worked Example 5.7: Encoding RV32I instructions

(a) `add x5, x6, x7` (R-type, opcode 0110011, funct3 000, funct7 0000000):

00000000 00111 00110 000 00101 0110011 = 0x007302B3.

`sub x5,x6,x7` differs only in funct7 = 0100000: 0x407302B3.

(b) `addi x10, x10, -1` (I-type, opcode 0010011): imm = -1 = 111111111111:

111111111111 01010 000 01010 0010011 = 0xFFFF50513.

(c) `lw x9, 8(x2)` (I-type, opcode 0000011, funct3 010): 0x00812483.

(d) `sw x9, 12(x2)` (S-type, opcode 0100011): imm = 12 = 00000000 01100; imm[11:5] = 00000000, imm[4:0] = 01100:

00000000 01001 00010 010 01100 0100011 = 0x00912623.

(e) **beq x5, x6, +16** (B-type, opcode 1100011): offset 16 = 0 0 000000 1000 0₂ (bits 12, 11, 10:5, 4:1, 0). imm[12]=0, imm[10:5]=000000, imm[4:1]=1000, imm[11]=0:

0 000000 00110 00101 000 1000 0 1100011 = 0x00628863.

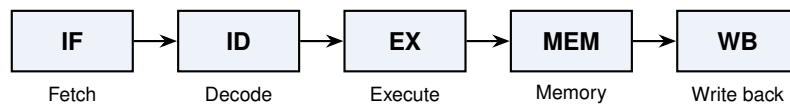
(f) **lui x5, 0x12345** then **addi x5, x5, 0x678** loads the 32-bit constant 0x12345678: 0x123452B7 and 0x67828293. (If the low 12 bits had bit 11 set, addi would sign-extend them as negative, so the assembler adds 1 to the lui value to compensate — the %hi/%lo adjustment.)

Worked Example 5.8: Decoding 0xFE029CE3

Binary: 1111111 000000 00101 001 11001 1100011. Opcode 1100011 $\Rightarrow$ B-type; funct3 001 $\Rightarrow$ bne; rs1 = 00101 = x5; rs2 = 00000 = x0. Immediate: imm[12]=1, imm[10:5]=111111, imm[4:1]=1100, imm[11]=1, so imm = 1 1 111111 1100 0₂ (13 bits) = -8. The instruction is **bne x5, x0, -8**: “branch back two instructions if x5 $\neq$ 0” — the closing branch of a small loop.

5.7.6 Execution in a pipeline: a preview

A RISC-V instruction naturally decomposes into five steps: **IF** (fetch the instruction from memory using the PC), **ID** (decode and read registers), **EX** (execute the operation in the ALU), **MEM** (access memory for loads/stores) and **WB** (write the result back to the register file). Unit 2 builds a single-cycle datapath that performs all five steps in one long clock cycle; Unit 3 pipelines them.



5.7.7 Instruction mix and implementations

Table 5.3: Typical RISC-V dynamic instruction mix (from the lecture).

Type	Description	Probability
Arithmetic/logic	ALU operations (add, sub, ...)	40%
Load/store	memory access	30%
Branch/control	conditional and unconditional jumps	20%
Immediate	ALU operations with immediate values	5%
System	special instructions (e.g. CSR)	3%
Other	rare instructions (e.g. atomics)	2%

Worked Example 5.9: Is adding a “load with post-increment” worth it?

Suppose 40% of the loads/stores in the mix above are in loops and are followed by an addi that increments the base pointer. A post-increment addressing mode would remove those addis: $0.30 \times 0.40 = 12\%$ of instructions disappear. But the new mode needs *two* register writes per instruction, a second register-file write port, and it lengthens the cycle by an estimated 4%.

$N' = 0.88N$, CPI unchanged, $T'_{\text{cycle}} = 1.04T$: $T'/T = 0.88 \times 1.04 = 0.915$ — an 8.5% gain. If the extra write port forced a 15% longer cycle instead, $0.88 \times 1.15 = 1.012$: a loss. This is exactly the quantitative reasoning that led RISC-V to omit auto-increment modes: their benefit is real but small, and their cost falls on *every* instruction of *every* implementation.

Implementations. Academic: Rocket Chip (Berkeley, in-order), BOOM (Berkeley Out-of-Order Machine), PicoRV32 (minimal), CVA6 (ETH/OpenHW, 6-stage application class). Commercial: SiFive cores (E31, E34, U54, U74, P-series), Western Digital SweRV, Andes, T-Head, and many embedded processor vendors. Applications: embedded and IoT, AI/ML accelerators, HPC, education and custom silicon.

Table 5.4: RISC-V compared with other architectures.

Feature	RISC-V	ARM	x86	MIPS
Open source	yes	no	no	no*
Instruction size	32-bit (+16-bit C)	32-bit (+T32)	variable 1–15 B	fixed 32-bit
Integer registers	31 + x0	15 (A32) / 31 (A64)	8–16	31 + \$0
Load–store	yes	yes	no	yes
Extensions	modular	some	many	few
Licensing	free	licensed	licensed	licensed

*MIPS was briefly offered under an open licence in 2019.

RISC-V advantages: open standard with no licensing fees; clean, modern design; modular and extensible; growing ecosystem. **Challenges:** a newer ecosystem than established architectures, software-toolchain maturity in some domains, and market-adoption timelines.

Advanced Insight: Custom extensions and post-quantum cryptography

RISC-V reserves opcode space (custom-0 ... custom-3) for domain-specific instructions. Post-quantum cryptographic schemes such as ML-KEM (Kyber) and ML-DSA (Dilithium) spend most of their time in the number-theoretic transform (NTT) over $\mathbb{Z}_q$ with $q = 3329$ or $q = 8380417$. A software butterfly needs a multiply, a modular (Montgomery or Barrett) reduction and an add/sub — roughly 10–15 RV32IM instructions. A custom `bfly rd, rs1, rs2` instruction performing one Cooley–Tukey butterfly with modular reduction in a 2–3 cycle functional unit can reduce the NTT’s instruction count by 5–8 $\times$. The ratified scalar-crypto extension Zk (AES, SHA-2, SM3/SM4 instructions) shows the same philosophy for classical cryptography; vector-crypto (Zvk*) extends it to the V extension.

Chapter summary

- An ISA defines state, instructions (opcode + operand specifiers), data types, addressing modes, the memory model, privilege, exceptions and protection.
- Operand location classifies ISAs: stack (0-address), accumulator (1), register–memory (2), load–store (3). Dataflow execution fires instructions when operands are ready.
- CISC narrows the semantic gap with complex instructions; RISC widens it and relies on compilers and pipelining. Modern x86 translates CISC into RISC-like micro-ops.
- RV32I: 32 registers (x0 = 0), byte-addressed little-endian memory, load–store, six

formats (R, I, S, B, U, J) with fixed register-field positions and sign bit at bit 31.

Exercises

E5.1 Write $Z = A \times B + C \times D$ for stack, accumulator and load-store machines; count instructions and data memory references.

Answer: Stack: 8 instr. (push A, push B, mul, push C, push D, mul, add, pop Z), 5 refs; accumulator: 7 (ld C, mul D, st T, ld A, mul B, add T, st Z), 7 refs; load-store: 8 (4 loads, 2 mul, add, store), 5 refs.

E5.2 Encode `or x8, x9, x10, andi x8, x8, 0xFF, sb x8, -4(x2)`.

Answer: `0x00A4E433; 0xFF47413; imm -4 = 11111111100: 0xFE810E23`.

E5.3 Decode `0x00A00093` and `0xFFC4A303`.

Answer: `addi x1, x0, 10 (li ra, 10); lw x6, -4(x9)`.

E5.4 What range of byte offsets can a single `beq` reach? A single `jal`?

Answer: `[-4096, +4094]; [-1 MiB, +1 MiB - 2]`.

E5.5 Evaluate the postfix expression `5 1 2 + 4 * + 3 -` on a stack machine, showing the stack after each step.

Answer: 14; maximum depth 3.

E5.6 An ISA change reduces instruction count by 10% but raises CPI from 1.20 to 1.28 at the same clock. Is it worthwhile?

Answer: $0.9 \times 1.28 / 1.20 = 0.96$: 4% faster — yes, marginally.

E5.7 Explain why `x0` being hard-wired to zero lets RISC-V omit `mov`, `nop`, `neg` and `beqz` as real instructions.

6

Computability and Complexity: The Limits of Any Processor

Every processor in this course — from the stack machine to an out-of-order RISC-V core — is equivalent in *what* it can compute: they differ only in *how fast*. This chapter makes that statement precise. We introduce the Turing machine, define which problems are solvable (decidable) and which are not, prove that the Halting Problem is undecidable, and separate solvability from efficiency with complexity classes. The chapter ends by connecting these ideas back to hardware: Turing completeness of ISAs and the undecidability of general hardware/software verification.

6.1 Problems and Turing machines

In computability theory a **problem** is a question with a yes/no answer for each input, for example “is n prime?”. Formally, a problem is a *language* $L \subseteq \Sigma^*$: the set of input strings (over alphabet Σ) for which the answer is yes.

Definition 6.1: Turing machine

A Turing machine (TM) $M = (Q, \Sigma, \Gamma, \delta, q_0, q_{\text{acc}}, q_{\text{rej}})$ consists of a finite set of states Q , an input alphabet Σ , a tape alphabet $\Gamma \supset \Sigma$ containing a blank symbol $\sqcup$, a transition function $\delta : Q \times \Gamma \rightarrow Q \times \Gamma \times \{L, R\}$, a start state and accepting/rejecting halt states. The machine has an infinite tape and a read/write head; at each step it reads the symbol under the head, and according to δ writes a symbol, moves left or right, and changes state.

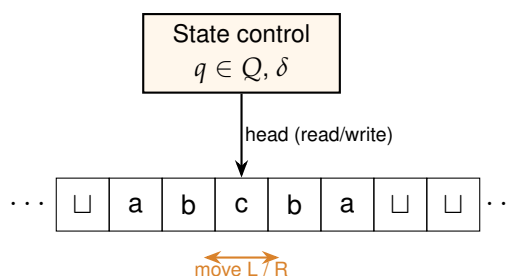


Figure 6.1: Turing machine schematic: finite control plus an unbounded tape.

A TM is a processor with unbounded memory

The finite control is the processor’s control unit and registers (finite state); the tape is memory. A real computer has finite memory and is technically a (huge) finite-state machine, but with $2^{8 \times 10^{10}}$ states for 10 GB of RAM, the TM is the right model for reasoning about what algorithms can do. The **Church–Turing thesis** asserts that every effectively computable

function is computable by a TM.

Worked Example 6.1: A TM that increments a binary number

Input: a binary number on the tape, head on its least significant (rightmost) bit. Transitions:

State	Read	Write, move	Next state
q_0 (carry)	1	0, L	q_0
q_0	0	1, L	q_{acc}
q_0	$\sqcup$	1, L	q_{acc}

Trace on 1011 (= 11): read 1 $\rightarrow$ write 0 (1010), move left; read 1 $\rightarrow$ 0 (1000); read 0 $\rightarrow$ 1 (1100), halt. Result 1100 = 12. $\checkmark$ Three steps; in general the number of steps equals the number of trailing ones plus one — the same carry-propagation behaviour as the ripple-carry adder of Chapter 2.

6.2 Decidable (Turing-solvable) problems

Definition 6.2: Recursive (decidable) language

A language $L \subseteq \Sigma^*$ is **recursive** if there exists a TM M that *halts on every input* and

- halts and accepts w if $w \in L$;
- halts and rejects w if $w \notin L$.

Such problems are always solvable by an algorithm. The class of recursive languages is denoted **R**.

Examples of Turing-solvable problems.

- **Halting in k steps:** given M, w, k , does M halt on w within k steps? (Simulate for k steps.)
- **Palindrome checking:** does a string read the same forwards and backwards?
- **Primality testing:** is n prime? Naive trial division takes $O(\sqrt{n}) = O(2^{k/2})$ divisions for a k -bit number — exponential in the input *length*. The AKS test (2002) runs in polynomial time in k , so **PRIMES** $\in$ **P**.

Worked Example 6.2: Why trial division is exponential

For a k -bit number $n < 2^k$, trial division up to $\sqrt{n}$ needs up to $2^{k/2}$ divisions. A 64-bit n : $2^{32} \approx 4.3 \times 10^9$ divisions — a few seconds at 10^9 divisions/s. A 2048-bit RSA-sized n : $2^{1024} \approx 1.8 \times 10^{308}$ divisions — more than the number of atoms in the observable universe ($\sim 10^{80}$) for every nanosecond since the Big Bang ($\sim 4 \times 10^{26}$ ns). Yet a probabilistic Miller–Rabin test on the same number needs only $O(k^3)$ bit operations: well under a millisecond. The problem is decidable either way; the *algorithm* determines whether it is feasible.

6.3 Computational complexity

Definition 6.3: Time complexity

The time complexity of a TM M is $T_M(n)$ = the maximum number of steps M takes to halt on any input of length n .

Complexity classes organize the solvable problems:

- **P**: problems decidable in polynomial time, $T(n) = O(n^c)$.
- **NP**: problems whose yes-answers can be *verified* in polynomial time given a certificate (e.g. Boolean satisfiability: check a proposed assignment).
- **EXP**: problems decidable in exponential time $O(2^{n^c})$.

$\mathbf{P} \subseteq \mathbf{NP} \subseteq \mathbf{EXP}$ and, by the time-hierarchy theorem, $\mathbf{P} \subsetneq \mathbf{EXP}$. Whether $\mathbf{P} = \mathbf{NP}$ is the most famous open problem in computer science. Note that **decidability** $\neq$ **efficiency**: some solvable problems are hopelessly expensive.

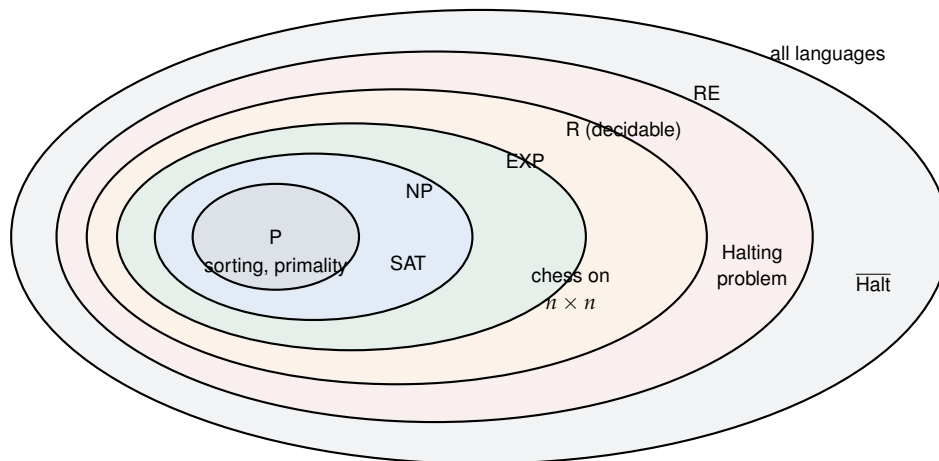


Figure 6.2: Nested complexity and computability classes (assuming $\mathbf{P} \neq \mathbf{NP}$). The Halting Problem is recursively enumerable but not decidable; its complement is not even RE.

Worked Example 6.3: Polynomial versus exponential, in seconds

A machine executes 10^9 basic steps per second. Running times for input size n :

$T(n)$	$n = 20$	$n = 50$	$n = 100$
n^2	$0.4 \mu\text{s}$	$2.5 \mu\text{s}$	$10 \mu\text{s}$
n^3	$8 \mu\text{s}$	$125 \mu\text{s}$	1 ms
2^n	1.0 ms	13 days	$4 \times 10^{13} \text{ years}$
$n!$	77 years	10^{48} years	—

A $1000\times$ faster processor (a decade or two of progress) lets an n^3 algorithm handle inputs $10\times$ larger, but a 2^n algorithm only $\log_2 1000 \approx 10$ elements larger. Faster hardware cannot rescue a bad algorithm.

6.4 Undecidable (Turing-unsolvable) problems

Definition 6.4: Recursively enumerable language

A language $L \subseteq \Sigma^*$ is **recursively enumerable (RE)** if there exists a TM M such that if $w \in L$, M halts and accepts; if $w \notin L$, M may either halt and reject or *run forever*.

$\mathbf{R} \subsetneq \mathbf{RE}$, and problems outside $\mathbf{R}$ are **undecidable**: no algorithm answers them correctly on all inputs.

6.4.1 The Halting Problem

Definition 6.5: Halting problem

$L_H = \{\langle M, w \rangle \mid M \text{ is a TM that halts on input } w\}$.

Theorem (Turing, 1936). $L_H \notin \mathbf{R}$: the Halting Problem is undecidable. (It is, however, in RE: simulate M on w and accept if it halts.)

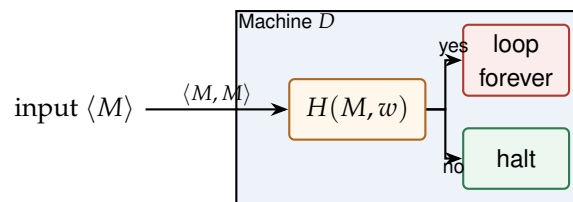


Figure 6.3: The diagonal construction: D runs the hypothetical decider H on $\langle M, M \rangle$ and does the opposite of its prediction.

Proof sketch.

1. Assume a decider H exists for L_H : $H(\langle M, w \rangle)$ always halts, accepting iff M halts on w .
2. Construct a new machine D that, on input $\langle M \rangle$, runs H on $\langle M, M \rangle$ (does M halt when given its own description?). If H accepts, D loops forever; if H rejects, D halts.
3. Analyse $D(\langle D \rangle)$. If D halts on $\langle D \rangle$, then H accepted $\langle D, D \rangle$, so D loops — contradiction. If D loops, then H rejected, so D halts — contradiction.
4. Therefore H cannot exist. ■

The argument is the same diagonalization Cantor used to prove that the reals are uncountable.

6.4.2 Other unsolvable problems and reductions

- **Emptiness:** given a TM M , is $L(M) = \emptyset$?
- **Equivalence:** given two TMs M_1, M_2 , do they accept the same language?
- **Rice's theorem:** *every* non-trivial property of the language accepted by a TM is undecidable.

These are proved by **reduction**: if a known unsolvable problem A reduces to a new problem B ($A \leq_T B$: a decider for B would yield a decider for A), then B is unsolvable.



Worked Example 6.4: Halting $\leq_T$ Emptiness

Given $\langle M, w \rangle$, build M_w : “on any input x , ignore x , run M on w ; if it halts, accept”. Then $L(M_w) = \Sigma^*$ if M halts on w , and $\emptyset$ otherwise. A decider for emptiness applied to M_w would decide halting. Since halting is undecidable, so is emptiness.

How to determine solvability mathematically. To prove a problem *solvable*, construct an explicit TM (algorithm) that halts on all inputs. To prove it *unsolvable*, reduce a known undecidable problem to it (e.g. Halting $\leq_T$ Emptiness).

6.5 Why this matters to a computer architect

Advanced Insight: Turing completeness of ISAs — even with one instruction

An ISA is **Turing complete** if, given unbounded memory, it can simulate any TM. All practical ISAs are. Remarkably, a single instruction suffices: `SUBLEQ  $a, b, c$` means “ $M[b] \leftarrow M[b] - M[a]$; if $M[b] \leq 0$ goto c ”. Addition, for instance, is

`subleq A,Z; subleq Z,B; subleq Z,Z` (with $Z = 0$) $\Rightarrow B \leftarrow B + A$.

Three instructions for an add: the semantic gap is enormous, but nothing is lost in *computability*. Everything that separates a good ISA from SUBLEQ is about *efficiency* — the iron law of Chapter 3, not the Church–Turing thesis.

Advanced Insight: Undecidability and hardware verification

Rice’s theorem says no tool can decide arbitrary semantic properties of arbitrary programs — e.g. “does this firmware ever write outside its buffer?”. Hardware is finite-state, so properties of an RTL design *are* decidable in principle (by exhaustive state exploration), but the state space of a 64-bit core is astronomically large ($2^{\text{\#flip-flops}}$). Practical formal verification therefore uses *bounded* model checking (check all behaviours up to k cycles — the decidable “halting in k steps” problem of this chapter) with SAT/SMT solvers whose worst case is exponential (NP-complete) but which succeed on many real designs. Simulation-based verification, the subject of VL-401, samples behaviours rather than proving them.

Chapter summary

- Turing-solvable = decidable = recursive languages (**R**): a TM halts on every input with the right answer.
- Turing-unsolvable = undecidable (outside **R**). RE languages may loop on non-members; $\mathbf{R} \subsetneq \mathbf{RE}$.
- The Halting Problem is the canonical undecidable problem (proof by diagonalization); others follow by reduction.
- Complexity theory further classifies solvable problems by efficiency: $\mathbf{P} \subseteq \mathbf{NP} \subseteq \mathbf{EXP}$. Hardware speed changes constants, not complexity classes.

Exercises

E6.1 Design a TM that decides whether a binary string has an even number of 1s. How many states are needed?

Answer: 2 states (even/odd) plus accept/reject; one left-to-right pass.

E6.2 Trace the increment TM on 0111. How many steps?

Answer: 1000; 4 steps.

E6.3 Show that “does program P ever print hello?” is undecidable by reduction from the Halting Problem.

Answer: Construct P' : run M on w (printing nothing), then print hello. P' prints hello iff M halts on w .

E6.4 Is “does M halt on w within 10^6 steps?” decidable? What is the complexity of your algorithm in terms of $k = 10^6$?

Answer: Yes; simulate for k steps: $O(k)$ simulated steps.

E6.5 A problem has an $O(n^2)$ algorithm and an $O(2^{n/4})$ algorithm. For which n is the exponential one faster (ignore constants)?

Answer: $2^{n/4} < n^2$ for $n < \approx 44$ (and $n \leq 1$): small inputs.

E6.6 Using SUBLEQ, write a program fragment that copies $M[A]$ to $M[B]$.

Answer: subleq B,B; subleq A,Z; subleq Z,B; subleq Z,Z (clear B, $Z = -A$, $B = A$, clear Z).