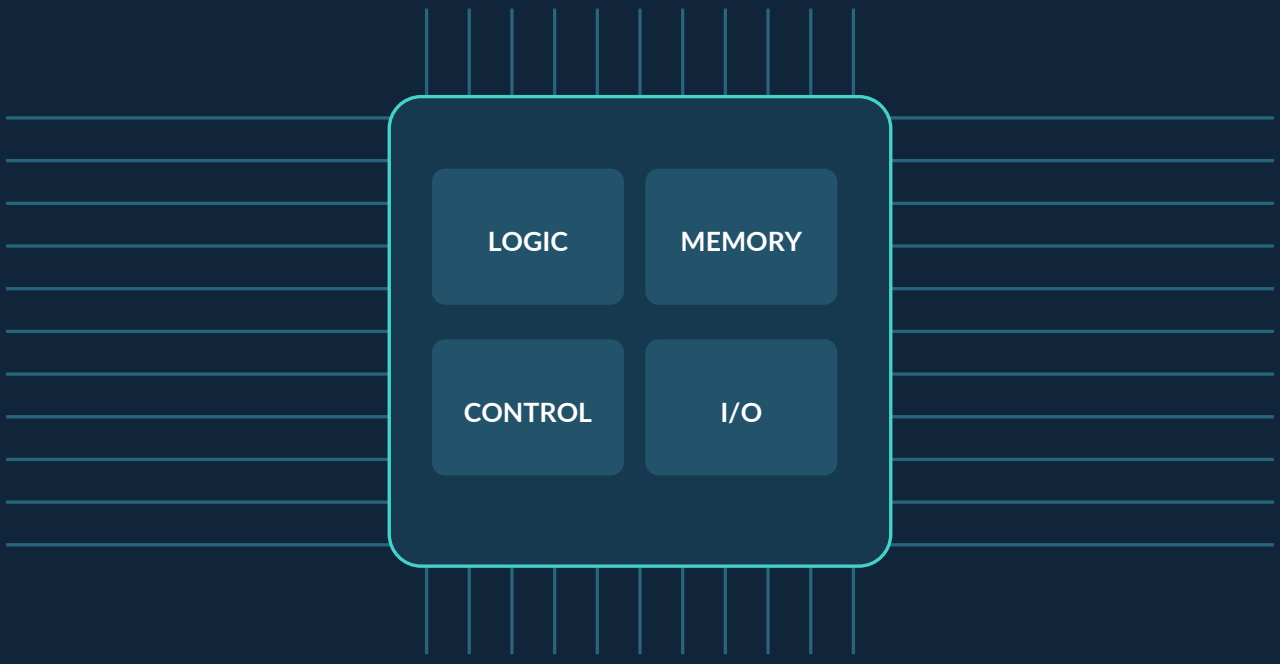
COMPUTER SCIENCE / RESEARCH BOOK

How Computers Really Work

From transistors to artificial intelligence

An expanded technical guide to the physical, architectural and software layers of digital computation.



27 chapters / 108 lessons / 44 references

Expanded edition • November 2026

Contents

1 / 5

01 **What makes a computer a computer?** 8
A machine, a model and a program 9
Hardware and software contracts 10
Computer families and system boundaries 11

02 **A brief history of programmable machines** 12
From mechanical calculators to relays 13
Stored programs and early architecture 14
Microprocessors, personal computers and cloud 15

03 **Electricity, silicon and transistors** 16
Voltage, charge and digital noise margins 17
Semiconductor junctions and MOSFET control 18
CMOS networks and switching energy 19

04 **How a microchip is manufactured** 20
From hardware specification to mask layout 21
Wafer processing and lithography 22
Packaging, testing and yield 23

05 **Binary, hexadecimal and signed integers** 24
Positional notation and bit manipulation 25
Signed arithmetic and overflow 26
Endianness, alignment and serialization 27

06 **Text, images, audio and floating point** 28
Unicode, bytes and text boundaries 29
Sampling, pixels and compression 30
Floating-point layout and numerical stability 31

Contents

2 / 5

07 **Logic gates and Boolean algebra** 32
Boolean identities and universal gates 33
Multiplexers, decoders and encoders 34
Combinational logic and critical paths 35

08 **Arithmetic circuits, clocks and state** 36
Full adders, subtractors and carry networks 37
Latches, flip-flops and register files 38
Finite-state machines and hardware control 39

09 **CPU anatomy and the instruction set** 40
CPU architectural state and register roles 41
Instruction decoding and datapath controls 42
ISA design: x86-64, Arm and RISC-V 43

10 **A real RISC-V instruction, bit by bit** 44
RISC-V load and store instructions 45
Branches, jumps and loops 46
Function calls and the stack convention 47

11 **Pipelining, speculation and parallel execution** 48
The five-stage pipeline model 49
Data hazards, forwarding and stalls 50
Out-of-order execution and branch prediction 51

12 **Registers, caches, DRAM and locality** 52
SRAM, DRAM and the memory controller 53
Cache mapping, tags and replacement 54
Locality, miss cost and prefetching 55

Contents

3 / 5

13 **Virtual memory and address translation** 56
Address spaces, pages and page tables 57
Page faults, demand paging and copy-on-write 58
Memory pressure, swap and working sets 59

14 **SSDs, hard disks and file systems** 60
Magnetic disks, flash and block interfaces 61
SSD translation, garbage collection and endurance 62
File systems, journaling and durability 63

15 **Motherboards, interconnects and input/output** 64
Motherboard power, clocks and platform topology 65
PCI Express, USB and interconnect bandwidth 66
Interrupts, DMA and device queues 67

16 **What happens when power is switched on?** 68
Reset, firmware and hardware initialization 69
UEFI, bootloaders and the kernel handoff 70
Secure boot, measured boot and recovery 71

17 **Operating systems: processes and system calls** 72
Processes, threads and protected execution 73
Scheduling, context switches and CPU time 74
System calls, exceptions and device services 75

Contents

4 / 5

18 **Concurrency, synchronization and virtualization** 76
Race conditions, atomics and mutual exclusion 77
Condition variables, deadlocks and progress 78
Virtual machines, containers and isolation 79

19 **From source code to executable instructions** 80
Lexing, parsing and semantic analysis 81
Intermediate representation and optimization 82
Assembly, linking, loading and runtimes 83

20 **GPU rendering, displays and accelerators** 84
Raster graphics, pixels and the rendering pipeline 85
GPU execution, memory and parallel work 86
Display scanout, input lag and media engines 87

21 **How the internet and a browser work** 88
Network addressing, routing and DNS 89
TCP, UDP, QUIC and reliable delivery 90
HTTP, TLS and the browser rendering path 91

22 **How artificial intelligence runs on computers** 92
Neural networks as numerical computation 93
Transformers, attention and token generation 94
AI hardware, precision and memory constraints 95

Contents

5 / 5

23 **Measuring performance and energy** 96
Latency, throughput, IPC and CPU-time equations 97
Amdahl's law and parallel scaling 98
Profiling, bottlenecks and thermal limits 99

24 **Computer security and reliability** 100
Memory protection, permissions and least privilege 101
Encryption, hashing and digital signatures 102
Reliability, ECC, backups and failure recovery 103

25 **The future: chiplets, specialized systems and quantum** 104
Chiplets, stacked memory and heterogeneous integration 105
Embedded, edge and specialized computing 106
Quantum computing: a different computational model 107

26 **End-to-end: opening a website on a computer** 108
Case study: launching an application 109
Case study: reading and saving a file 110
Case study: typing a letter and seeing it appear 111

27 **Practical study and verification exercises** 112
Lab: build a simple truth table and adder 113
Lab: inspect a program's instructions 114
Lab: measure locality and resource use 115

Glossary and notation 116–117

Reference directory 118–121

How to use this book

Scope and method

This book is a secondary educational synthesis of established computer science, engineering and technical standards. It explains the ordinary digital computer from physical signals to applications. It is not a peer-reviewed experimental paper, a product review, or an exhaustive catalog of every implementation. Numerical examples are illustrative unless otherwise identified.

A progressive reading path

Chapters 1–8 develop electrical, mathematical and digital logic foundations. Chapters 9–16 cover CPUs, memory, storage, I/O and startup. Chapters 17–25 extend into operating systems, languages, graphics, networks, AI, performance and reliability. Chapters 26–27 connect the layers through examples and practical exercises.

Source strategy

University courses support foundational models; instruction-set and protocol specifications establish architectural contracts; engineering documentation illustrates particular systems. Bracketed numbers point to linked references. A reference is a starting point for deeper reading, not a claim that every sentence reproduces that source.

Conventions

Unless stated otherwise, binary numbers use ordinary positional notation, addresses are hexadecimal when prefixed with 0x, and KiB/MiB/GiB denote powers of 1024. KB/MB/GB denote decimal units in calculations. Block diagrams abstract away implementation detail; arrows indicate logical flow rather than a complete wiring schematic.

What to do with the examples

Follow each worked example, then change its input and predict the result. Use the lab chapters to test your reasoning on a small, safe software experiment. Refer back to the relevant chapter when a result is unexpected.

CHAPTER 01 / OVERVIEW

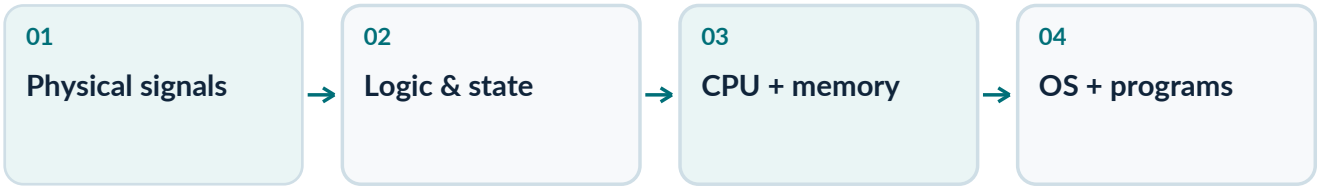
What makes a computer a computer?

A computer is a programmable physical system that represents information, transforms it according to rules, preserves state and communicates results. A calculator and a data-center server share these principles, even though their capabilities, scale and interfaces differ enormously. The important distinction is not size or speed, but the ability to execute a sequence of operations under program control.

The stored-program model lets a processor obtain instructions and operands from memory. Historically, early machines often had to be rewired or configured for a new task; stored programs made general-purpose reprogramming much easier. Modern implementations may have separate instruction and data caches while presenting a shared address space to software. The architecture is therefore not a literal picture of every physical wire.

The machine can be understood as a hierarchy of abstractions. At the bottom, voltage and current change in circuits. Logic gates create Boolean functions. Sequential circuits retain state. A processor interprets instructions, an operating system manages resources, and applications use those resources to solve human problems. Each interface hides details of the layer below it, but performance and correctness still depend on those details.

CONCEPT DIAGRAM



A simple calculation

Entering 7 + 5 triggers input events, software parses the expression, a processor or runtime computes 12, and a display renders the characters. Many more operations occur around the addition than in the addition itself.

Sources [1, 2] / Reference directory

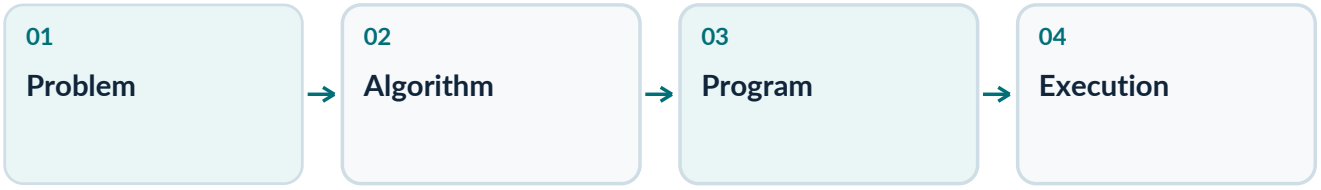
CHAPTER 01 / DEEP DIVE 1

A machine, a model and a program

Computability is about what a procedure can accomplish in principle, not how quickly a particular computer runs it. A finite program can repeatedly read inputs, change state and choose its next operation. A model such as a Turing machine idealizes memory as unbounded; a physical laptop has finite storage, power and time. General-purpose hardware approximates the model within those limits.

An algorithm specifies a method independently of its implementation. A program expresses that method in a language, while a running computation consists of actual state changes. The same sorting algorithm can be implemented on a microcontroller, a multicore CPU or a GPU, but its cost depends on data movement and execution architecture.

CONCEPT DIAGRAM



Trace a tiny algorithm

Start with sum = 0. Read 4, 7 and 2; update sum to 4, 11 and 13. The program is the rule; the successive sums are the changing machine state. A larger memory does not change the rule, but permits larger input.

Sources [1, 2] / Reference directory

CHAPTER 01 / DEEP DIVE 2

Hardware and software contracts

Abstraction is a contract rather than a claim that lower layers do not exist. A processor promises instruction behavior, a system-call interface promises operations on OS-managed resources, and a library promises a programming interface. A compiler can change the emitted machine code while preserving the language's required behavior.

Different contracts have different compatibility consequences. A source program may be portable if it avoids platform-specific assumptions. A compiled executable depends on an instruction set, operating-system ABI and required libraries. Even a compatible binary can be slower on another machine because the physical resources differ.

TECHNICAL REFERENCE

Contract	Example	What it hides
ISA	RV64I	Datapath design
ABI	Calling convention	Register allocation
API	File read	Device protocol

Same program, different CPU

A C implementation of integer addition may compile to AArch64 on one computer and RISC-V on another. The source can remain unchanged, but those native instruction bytes generally cannot execute directly on the other ISA.

Sources [1, 9, 10] / Reference directory

CHAPTER 01 / DEEP DIVE 3

Computer families and system boundaries

A microcontroller typically combines processing, memory and peripheral controllers in one chip and often performs a dedicated control task. A desktop prioritizes interactive applications and flexible peripherals. Servers prioritize capacity, reliability and concurrent workloads. Supercomputers coordinate many processors and accelerators through high-speed networks.

A system-on-chip integrates multiple subsystems, often including CPU cores, graphics, memory controllers and media engines. Integration reduces some communication overhead, while a modular system can offer replaceable components. Neither approach makes all workloads faster; the required interfaces and resource limits determine suitability.

TECHNICAL REFERENCE

System	Typical emphasis
Microcontroller	Predictable control
Laptop	Battery and responsiveness
Server	Concurrent services
HPC cluster	Parallel throughput

Choose a system for a task

A temperature logger may need only a small controller, while a video renderer benefits from larger memory and parallel hardware. A database serving many clients may prioritize storage latency and network reliability instead of graphics performance.

Sources [1, 10, 19] / Reference directory

CHAPTER 02 / OVERVIEW

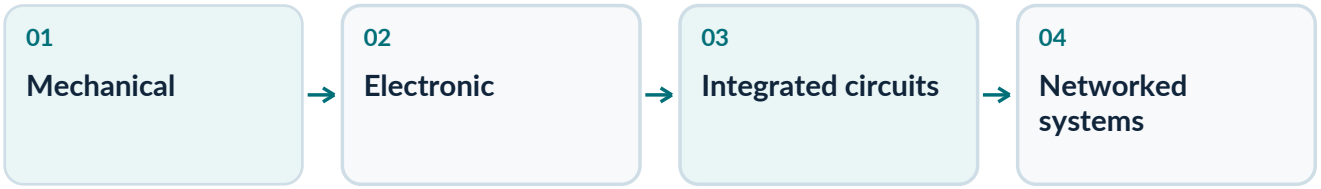
A brief history of programmable machines

Mechanical calculation preceded electronic computing by centuries. Nineteenth-century designs such as Babbage's Analytical Engine anticipated programmable operations, although the complete machine was not constructed in his lifetime. Electromechanical relay machines and electronic vacuum-tube machines later demonstrated large-scale computation. Early electronic computers were expensive, bulky and often specialized.

The Manchester Baby executed a stored program in 1948; EDSAC became an early practical stored-program service in 1949. Transistors displaced many vacuum tubes because smaller solid-state devices offered advantages in size, energy use and reliability. Integrated circuits placed many components on a single chip, and the microprocessor integrated the processing function into a chip in the early 1970s. Affordable personal computers, graphical interfaces, networking and mobile systems subsequently broadened access to computing.

Today's computing landscape spans microcontrollers, personal computers, servers, supercomputers and specialized accelerators. Progress has depended on manufacturing, architecture, software tools and communications together. The history is not a sequence of one inventor replacing another; it is a long interaction among mathematical ideas, engineering and practical needs.

CONCEPT DIAGRAM



Why the stored program matters

Changing an algorithm can mean loading new bytes into memory rather than rebuilding circuitry. Hardware remains general-purpose within the capabilities defined by its instruction set.

Sources [2, 3] / Reference directory

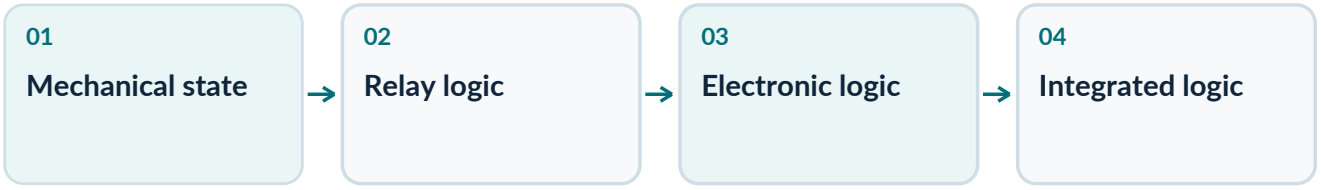
CHAPTER 02 / DEEP DIVE 1

From mechanical calculators to relays

Mechanical devices encoded numbers through physical positions of wheels, gears or other components. Addition required moving components in a controlled sequence; carry mechanisms transferred overflow to the next digit. Such machines demonstrate that arithmetic need not be implemented with electricity.

Electromechanical relays make switching decisions using electrically controlled mechanical contacts. Networks of switches can implement conditional logic, but the movement introduces delay and wear. Electronic switching removed much of this mechanical bottleneck. The underlying ideas of representation, arithmetic and control continued across these technological changes.

CONCEPT DIAGRAM



A carry is a physical event

When a decimal wheel advances from 9 to 0, the next wheel must increase. In a binary electronic adder, a carry plays a similar mathematical role, although its implementation is a voltage signal rather than a mechanical linkage.

Sources [2, 3] / Reference directory

CHAPTER 02 / DEEP DIVE 2

Stored programs and early architecture

Early programmable systems used various ways to specify a sequence of operations, including external media and physical configuration. With stored programs, instructions can reside in addressable memory. This lets software select different routines, branch according to computed values and load new programs with comparatively little physical alteration.

The conceptual architecture often described as von Neumann organizes a processor, memory and input/output around a stored program. Real systems diverge from the simple block diagram: instruction and data paths can be separate, memory is hierarchical, and devices can transfer data independently. The model remains useful as a way to reason about program state.

CONCEPT DIAGRAM



Program changes without rebuilding

Imagine memory containing a sequence that adds two numbers. Replacing that sequence with one that compares them changes the work performed by the same processor. No new arithmetic wiring is required if the ISA supports both operations.

Sources [2, 3] / Reference directory

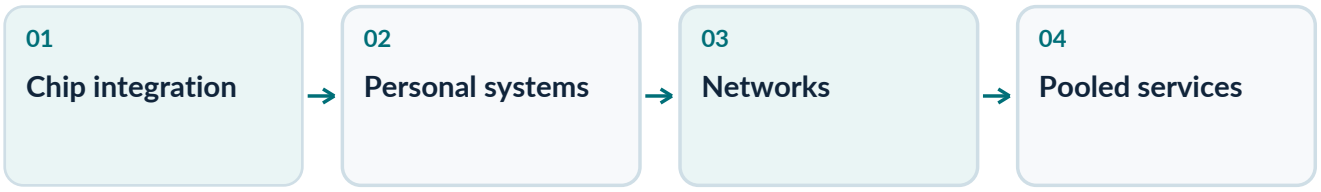
CHAPTER 02 / DEEP DIVE 3

Microprocessors, personal computers and cloud

Integrated circuits made it practical to connect many electronic components without individually wiring every transistor. A microprocessor places instruction-processing circuitry on a chip; supporting memory and I/O can be on other chips or integrated nearby. Increased integration lowered barriers to building small, affordable computing systems.

Personal computers made interactive applications widely accessible, while networks let machines cooperate. Cloud computing uses pooled infrastructure, virtualization and automated management to provide computing resources as services. A cloud server is still a physical computer: a request consumes processor time, memory bandwidth, storage and network capacity somewhere in a data center.

CONCEPT DIAGRAM



Local versus remote execution

Editing a file locally can use your own CPU and SSD. Editing through a web service may also send changes to a remote machine for persistence. In both cases, electrical switching and instruction execution remain fundamental.

Sources [2, 5, 10] / Reference directory

CHAPTER 03 / OVERVIEW

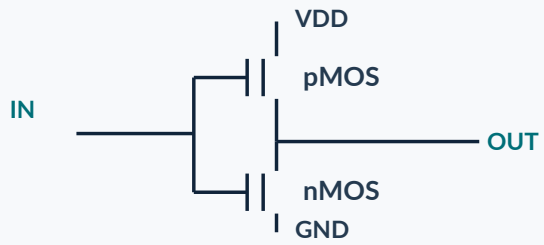
Electricity, silicon and transistors

Voltage is electrical potential difference; current is the rate of charge flow. A digital circuit uses ranges of voltage to encode logical values. The exact levels depend on the technology, supply voltage and design. A logical zero is not always exactly zero volts, and a logical one is not universally five volts. Noise margins allow imperfect physical signals to be treated reliably as discrete symbols.

Silicon is a semiconductor. Introducing selected impurity atoms changes its charge-carrier population: donor doping produces n-type material and acceptor doping produces p-type material. In a MOS field-effect transistor, an insulated gate controls the conductivity of a channel between source and drain. An n-channel device normally conducts strongly when its gate is sufficiently positive relative to its source; a p-channel device responds to the opposite polarity. Real devices have leakage, capacitance and finite switching times.

Complementary MOS (CMOS) combines n-channel and p-channel networks. In a simple inverter, the p-channel device connects the output toward the supply when the input is low, while the n-channel device connects it toward ground when the input is high. Ideally there is no direct static conduction path between rails in either stable state, although real circuits consume leakage power and switching energy.

CONCEPT DIAGRAM



Input LOW: pMOS ON, nMOS OFF.
Input HIGH: pMOS OFF, nMOS ON.
The output is the logical inverse.

Dynamic power

A common approximation is P = alpha C V^2 f, with switching activity alpha, effective capacitance C, supply V and clock frequency f. Voltage reduction can save energy, but may reduce switching speed and operating margin.

Sources [1, 4, 5, 32] / Reference directory

CHAPTER 03 / DEEP DIVE 1

Voltage, charge and digital noise margins

Electric charge is a physical property, voltage describes potential difference, and current measures charge flow per unit time. Capacitors store charge; resistive paths dissipate energy. On-chip wires have electrical properties that affect both speed and power. Digital circuits still obey analog physical laws.

To communicate a logical value reliably, a transmitting circuit must produce a voltage range that a receiving circuit recognizes. Designers specify acceptable input and output levels. The separation between these ranges provides tolerance to noise. Signals near a threshold can be ambiguous; digital abstractions are valid only when electrical and timing requirements are met.

TECHNICAL REFERENCE

Quantity	Unit	Meaning
Voltage	V	Potential difference
Current	A	Charge per second
Resistance	Ohm	Opposition to current
Capacitance	F	Charge per volt

Illustrative noise margin

If a receiver accepts high inputs at 0.7 V or above and the transmitter guarantees at least 0.9 V, the high-level margin is 0.2 V. These are invented teaching values, not a universal logic standard.

Sources [1, 4, 32] / Reference directory

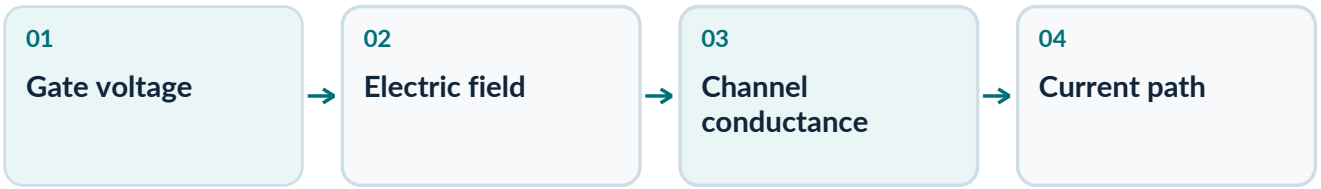
CHAPTER 03 / DEEP DIVE 2

Semiconductor junctions and MOSFET control

Pure silicon has a crystal structure whose electrical behavior can be modified by adding carefully controlled impurities. Donor and acceptor doping change the availability of mobile charge carriers. Junctions between differently doped regions exhibit directional electrical behavior, which is central to many semiconductor devices.

A metal-oxide-semiconductor field-effect transistor uses an electric field from its gate to influence conduction between source and drain. The gate is insulated from the channel in the idealized structure. An n-channel device generally conducts strongly with a sufficiently positive gate-to-source voltage; p-channel behavior has opposite polarity. Actual devices have leakage, parasitic capacitances and continuous current-voltage characteristics.

CONCEPT DIAGRAM



Switch abstraction and its limits

An ideal switch has zero resistance when closed and infinite resistance when open. A real MOSFET has finite on-resistance and off-state leakage. The simplified model helps explain logic, but detailed circuit simulation is needed for power and timing design.

Sources [1, 4, 32] / Reference directory

CHAPTER 03 / DEEP DIVE 3

CMOS networks and switching energy

Complementary MOS logic uses pull-up and pull-down networks to drive an output toward its supply rails. In a simple inverter, the pMOS pull-up conducts for a low input and the nMOS pull-down conducts for a high input. NAND and NOR gates use different series and parallel transistor arrangements to implement their truth tables.

Charging and discharging capacitance consumes energy. A common first-order model for dynamic power is P = alpha C V^2 f, where alpha represents switching activity under the chosen convention, C is effective capacitance, V is supply voltage and f is clock frequency. Static leakage and short-circuit current also contribute, so the formula is not a complete chip-power model.

TECHNICAL REFERENCE

Input	Pull-up	Pull-down	Output
Low	On	Off	High
High	Off	On	Low

Voltage and power

With all other terms fixed, reducing supply voltage from 1.0 V to 0.8 V changes the modeled dynamic power to 0.8 squared = 0.64 of the original. Real systems may also change frequency, leakage and timing margin.

Sources [1, 32] / Reference directory

CHAPTER 04 / OVERVIEW

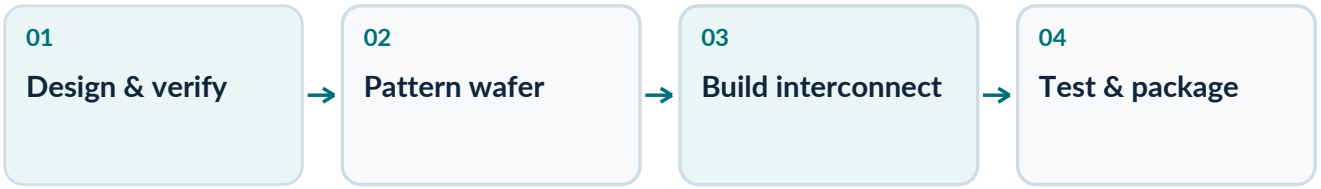
How a microchip is manufactured

Chip development begins with architecture, circuit design and verification. Engineers describe digital behavior using hardware description languages, synthesize logic and perform placement and routing. Timing analysis checks whether signals can reach their destinations within design constraints. Physical verification checks layout rules and connectivity before the design is sent for fabrication.

Manufacturing starts with highly purified silicon wafers. Repeated processes deposit films, coat surfaces with photoresist, expose patterns using lithography, develop the resist, etch selected regions and modify material properties through processes such as implantation. The sequence varies with the device and layer. Metal interconnect layers then connect the transistors into functional circuits. Precision alignment of many patterned layers is essential.

Finished wafers undergo electrical testing and are separated into dies. Packaging provides electrical connections, physical protection and paths for heat removal. Some packages contain multiple dies or chiplets connected by advanced interconnects. Devices are tested again and may be classified according to measured characteristics. Yield, manufacturing cost and power consumption influence commercial design decisions. A process-node name is a technology-generation label, not a simple measurement of every transistor feature.

CONCEPT DIAGRAM



Design versus fabrication

A chip architecture specifies what a design should do. A fabrication process determines how its devices and wiring are physically realized. Identical architectural instructions can be implemented by chips made with different technologies.

Sources [5, 6] / Reference directory

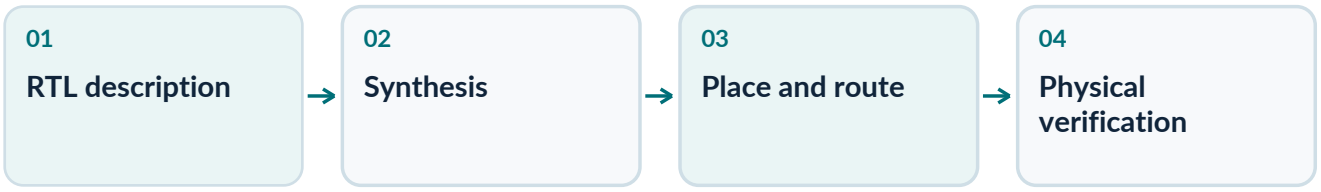
CHAPTER 04 / DEEP DIVE 1

From hardware specification to mask layout

Chip development starts with a specification: instruction behavior, interfaces, power budget, performance targets and reliability constraints. Hardware description languages describe the behavior and structure of circuits. Engineers verify modules with simulation, assertions and other techniques before manufacturing.

Logic synthesis maps register-transfer descriptions onto cells supported by the manufacturing process. Physical design then places those cells, routes wires and constructs power and clock networks. Timing, voltage drop, signal integrity and design rules are checked repeatedly. A design that simulates logically may still be physically impractical if routing or timing constraints cannot be satisfied.

CONCEPT DIAGRAM



A timing constraint

A circuit targeting 1 GHz has a nominal 1 ns clock period. Register delays, combinational paths, clock uncertainty and setup requirements must fit into that budget. Adding more logic between registers may force a redesign or lower clock rate.

Sources [1, 5, 6] / Reference directory

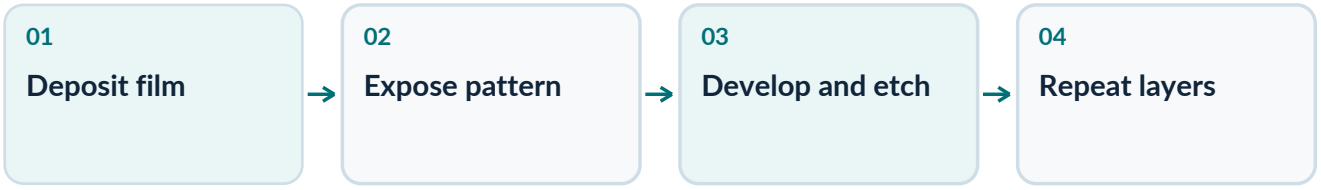
CHAPTER 04 / DEEP DIVE 2

Wafer processing and lithography

Manufacturing builds patterned material layers on a semiconductor wafer. Deposition adds thin films; lithography creates patterns in photoresist using controlled light; etching removes selected material; implantation or related techniques modify material properties. Repeated processing forms active devices and multiple levels of interconnect.

The dimensions involved depend on the process and on the particular feature being manufactured. Process-node names are not a simple measurement of every transistor dimension. Alignment between layers, defect density and process variation matter as much as nominal feature size. Measurement and process control help manufacturers detect departures from the intended geometry.

CONCEPT DIAGRAM



Why alignment matters

A contact must overlap the appropriate underlying structure. If patterns shift too far relative to one another, a connection can become resistive, short to another feature or fail entirely. Overlay control is therefore essential to reliable chips.

Sources [5, 6] / Reference directory

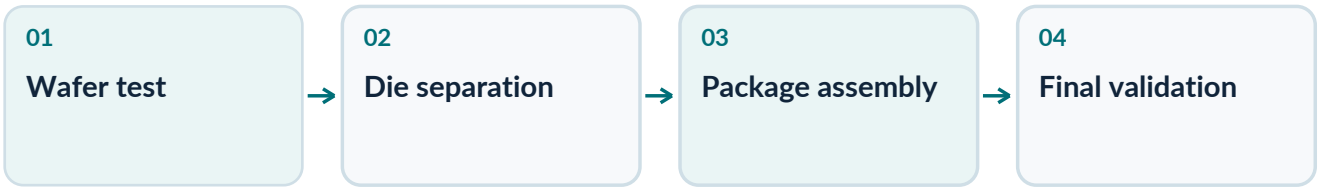
CHAPTER 04 / DEEP DIVE 3

Packaging, testing and yield

After fabrication, dies are tested and separated from the wafer. Packaging provides electrical connections, mechanical support and a path for heat removal. Modern packages may integrate multiple dies, stacked memory or interposers. The final product includes the package and its connections, not simply a piece of patterned silicon.

Manufacturing defects, variation and testing limits affect yield, the proportion of usable products. Some dies can be classified into different performance or power grades, depending on their measured properties. Tests may include functional patterns, memory checks and stress tests, but no finite test guarantees that every possible defect is absent.

CONCEPT DIAGRAM



Illustrative production yield

A hypothetical batch has 500 candidate dies and 430 pass its test criteria. Yield is 430/500 = 86%. This simple ratio does not include packaging losses, different product grades or the cost of testing.

Sources [5, 6] / Reference directory

CHAPTER 05 / OVERVIEW

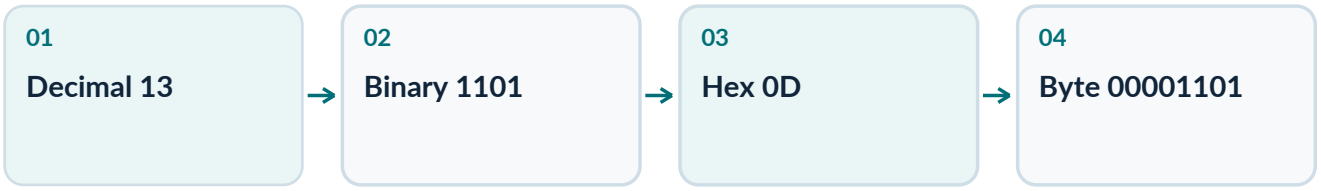
Binary, hexadecimal and signed integers

A bit represents one of two logical states. Eight bits form a byte; an unsigned n-bit field has 2^n possible patterns and represents integers from 0 through 2^n - 1 under ordinary binary encoding. The place values in binary are powers of two. Hexadecimal uses sixteen symbols and provides a compact notation because one hexadecimal digit corresponds to four bits.

Two's complement encodes signed integers by assigning the most significant bit a negative weight. An eight-bit signed value ranges from -128 through 127. The pattern 11111111 is 255 when unsigned and -1 when interpreted as signed. Hardware often uses the same binary addition circuit for both signed and unsigned values; the meaning of overflow differs. Unsigned addition wraps modulo 2^n in fixed-width arithmetic, while signed overflow rules depend on the instruction or programming language.

Byte order describes how a multi-byte number is arranged at consecutive memory addresses. Little-endian stores the least significant byte first; big-endian stores the most significant byte first. It does not reverse the bits within every byte. A data type supplies interpretation, constraints and operations; the same bits can encode many different kinds of information.

CONCEPT DIAGRAM



Worked conversion

13 = 8 + 4 + 1. In eight-bit two's complement, -13 is 11110011: invert 00001101 and add one. A 32-bit value 0x12345678 has bytes 78 56 34 12 in little-endian order.

Sources [1, 7] / Reference directory

CHAPTER 05 / DEEP DIVE 1

Positional notation and bit manipulation

A base-b number assigns each digit a weight that is a power of b. Binary has only two digit values, but the same positional principle as decimal. Hexadecimal is especially convenient for debugging because groups of four binary bits map exactly to hexadecimal digits.

Bitwise operators act on corresponding bits independently. AND can isolate selected fields, OR can set bits, and XOR can toggle them. Shifts move bit positions; right shifts can be logical or arithmetic depending on the operation and language. Fixed-width shifts must respect the rules for shift counts and discarded bits.

TECHNICAL REFERENCE

Operation	Example	Result
AND	1100 AND 1010	1000
OR	1100 OR 1010	1110
XOR	1100 XOR 1010	0110
Left shift	0011 by 1	0110

Extract a field

For byte 0xB6, mask with 0x0F to retain the low nibble: 0x06. Shifting the original byte logically right by four gives 0x0B. These operations let software interpret packed status fields.

Sources [1, 9] / Reference directory

CHAPTER 05 / DEEP DIVE 2

Signed arithmetic and overflow

A fixed-width integer has a finite set of bit patterns. In unsigned interpretation, all patterns represent nonnegative values. In two's complement interpretation, the highest bit contributes a negative place value. The arithmetic circuit can often use the same addition mechanism for both interpretations.

Overflow is a question about whether the mathematical result fits in the chosen interpretation. An unsigned carry out indicates an out-of-range unsigned sum. For signed addition, overflow occurs when two operands with the same sign produce a result with the opposite sign. Programming languages differ in how they handle these cases; checking a result requires knowing both the representation and the language contract.

TECHNICAL REFERENCE

4-bit operands	Bits after add	Interpretation
0111 + 0001	1000	Signed overflow
1111 + 0001	0000	Unsigned overflow
0011 + 0010	0101	Fits both

Design with wider intermediates

Adding two 8-bit unsigned numbers can produce at most 510. A 9-bit unsigned intermediate is sufficient. A compiler or programmer must actually use the wider representation before discarding bits; widening an already wrapped result cannot recover the lost carry.

Sources [1, 9] / Reference directory

CHAPTER 05 / DEEP DIVE 3

Endianness, alignment and serialization

A memory address identifies a location in an architectural address space. Multi-byte values occupy multiple addresses; byte order determines their ordering. Alignment constrains where some data objects may begin and can affect whether an operation is supported or efficient. The exact requirements are architecture and software dependent.

Serialization converts structured data into a specified sequence of bytes so it can be stored or transmitted. A portable format must define integer widths, byte order, lengths and other field rules. Dumping a language structure directly may include padding, native endianness or implementation-specific layout and therefore need not be portable.

TECHNICAL REFERENCE

Address	Little endian	Big endian
0x1000	78	12
0x1001	56	34
0x1002	34	56
0x1003	12	78

> **Interoperability**
>
> The value 0x12345678 must be written according to the file or network protocol. A reader on a machine with different native byte order can reconstruct the same mathematical value by following that protocol rather than assuming its own memory layout.

Sources [9, 10] / Reference directory

CHAPTER 06 / OVERVIEW

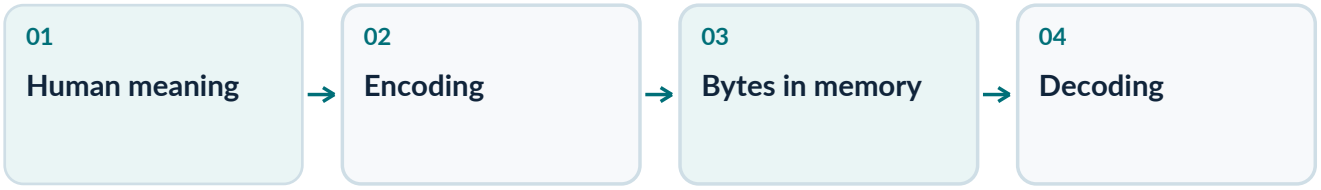
Text, images, audio and floating point

Text is represented by character encodings. Unicode assigns code points to characters and other textual elements; UTF-8 maps these code points to sequences of one to four bytes. ASCII letters occupy one byte in UTF-8, but many other scripts require multiple bytes. A displayed character may involve several code points, so counting bytes, code points and user-perceived characters gives different results.

A raster image is an array of samples such as RGB values. An uncompressed 1920 by 1080 image at 24 bits per pixel occupies 6,220,800 bytes before metadata or row padding. Audio stores sampled signals: sampling rate specifies samples per second, while bit depth describes quantization precision. Lossless compression preserves the original information; lossy compression discards selected detail to reduce size. Encoded video often exploits similarity across frames as well as spatial redundancy.

Floating-point values represent a finite set of numbers using sign, exponent and significand fields. IEEE 754 binary32 has one sign bit, eight exponent bits and twenty-three stored fraction bits. Its ordinary normalized values have an implicit leading one. Some decimal fractions have no finite binary expansion, so calculations may round. Special patterns encode infinities, NaNs, zeros and subnormal values. Use integer units or an appropriate decimal type for exact decimal accounting where required.

CONCEPT DIAGRAM



Why 0.1 is tricky

The decimal fraction 1/10 repeats in binary, much as 1/3 repeats in decimal. A finite binary floating-point format approximates it, so arithmetic comparisons require attention to rounding.

Sources [7, 8] / Reference directory

CHAPTER 06 / DEEP DIVE 1

Unicode, bytes and text boundaries

Unicode separates abstract text elements from their encoded byte representation. UTF-8 uses variable-length byte sequences; ASCII code points have the same single-byte encoding as ASCII. Continuation-byte patterns distinguish later bytes of a multi-byte sequence. Decoders must reject or otherwise handle invalid sequences according to their API contract.

Text operations are not always byte operations. Combining marks and sequences of code points can form one user-perceived character. Case conversion, normalization and line breaking require rules beyond indexing a byte array. A user interface that truncates text after an arbitrary byte count may cut a UTF-8 sequence or split a visible character.

TECHNICAL REFERENCE

Character	Code point	UTF-8 hex
A	U+0041	41
é	U+00E9	C3 A9
€	U+20AC	E2 82 AC
U+1F600 glyph	U+1F600	F0 9F 98 80

Measure correctly

The string A followed by U+20AC contains two code points and four UTF-8 bytes. A storage limit in bytes and a text editor cursor position are therefore different quantities, even before considering combining characters.

Sources [7] / Reference directory

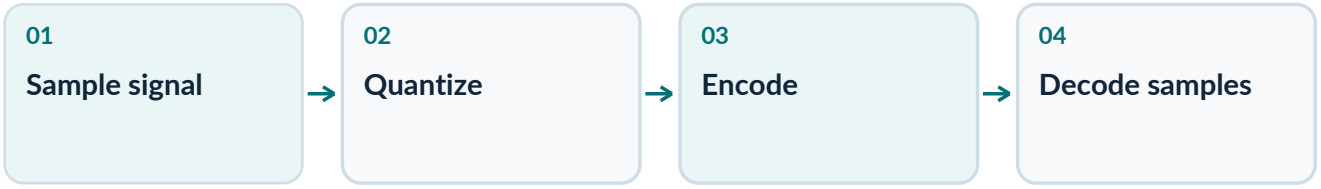
CHAPTER 06 / DEEP DIVE 2

Sampling, pixels and compression

Digital media represents signals with finite samples. Image resolution determines how many sample positions are available, while color format determines the components and precision at each position. Audio sampling converts changes over time into a discrete sequence; reconstruction of a suitably band-limited signal requires sufficient sampling frequency and filtering.

Compression reduces representation size. Lossless methods exploit redundancy without changing the reconstructed data; lossy methods permit controlled approximation. Image and video encoders can transform blocks, predict from neighboring data and encode residual differences. Decoding must follow the format's rules, and quality depends on the signal and encoding decisions.

CONCEPT DIAGRAM



Raw audio storage

Stereo audio at 48,000 samples/s, 16 bits/sample/channel needs 48,000 x 2 x 2 = 192,000 bytes/s. One minute takes 11,520,000 bytes before headers or compression. Compression ratios are content-dependent.

Sources [7, 8] / Reference directory

CHAPTER 06 / DEEP DIVE 3

Floating-point layout and numerical stability

Binary floating point distributes representable numbers unevenly. Near large magnitudes, adjacent representable values may be farther apart than near zero. Normalized numbers combine an exponent with a significand; subnormal numbers allow a gradual approach to zero. Rounding is part of the operation semantics.

Numerical algorithms must account for cancellation, overflow, underflow and accumulated error. Subtracting nearly equal approximate values can remove their shared leading digits, leaving a result with relatively poor accuracy. Reordering operations can change results because finite-precision addition is not generally associative. A mathematically equivalent expression may have a different numerical error profile.

TECHNICAL REFERENCE

Binary32 field	Width
Sign	1 bit
Biased exponent	8 bits
Stored fraction	23 bits

Exact and approximate

The fraction 0.5 equals 1/2 and has a finite binary representation. The fraction 0.1 equals 1/10 and does not. A decimal accounting program can instead represent money in integer smallest units if its range and rounding requirements permit.

Sources [8] / Reference directory

CHAPTER 07 / OVERVIEW

Logic gates and Boolean algebra

A gate realizes a Boolean function. NOT reverses its input; AND is true when both inputs are true; OR is true when at least one input is true; XOR is true when the inputs differ. NAND and NOR negate AND and OR respectively. Either NAND alone or NOR alone can express all Boolean functions through combinations of gates.

Digital designers combine gates into combinational circuits, whose outputs are determined by current inputs after propagation delay. A multiplexer selects one of multiple inputs using control bits. A decoder selects an output based on a binary code. Comparators test ordering or equality. Such circuits can be expressed using truth tables, Boolean expressions or structural connections. Logic synthesis translates descriptions into technology-specific networks and optimizes area, delay and power.

Computation is not instantaneous. Each gate has delay, and wiring introduces additional capacitance and resistance. The longest relevant path through combinational logic constrains the minimum clock period in a synchronous design. A gate-level implementation can be logically correct yet fail at a target operating frequency if timing constraints are violated.

TECHNICAL REFERENCE

A / B	AND	OR	XOR	NAND
0 / 0	0	0	0	1
0 / 1	0	1	1	1
1 / 0	0	1	1	1
1 / 1	1	1	0	0

Half adder

Sum = A XOR B; carry = A AND B. For inputs 1 and 1, the sum bit is 0 and carry is 1, which represents binary 10 (decimal 2).

Sources [1] / Reference directory

CHAPTER 07 / DEEP DIVE 1

Boolean identities and universal gates

Boolean algebra describes functions whose arguments and results belong to a two-element domain. The identities A AND 1 = A and A OR 0 = A express neutral elements. De Morgan's identities connect complement, conjunction and disjunction, allowing equivalent logic circuits to be expressed using different gates.

Any Boolean function of finitely many inputs can be expressed as a disjunction of minterms, one for each input pattern that produces true. This canonical form is not necessarily efficient. Designers simplify expressions or use synthesis tools to find implementations with lower delay, area or power. NAND gates alone suffice because they can construct NOT, AND and OR.

TECHNICAL REFERENCE

Expression	Equivalent
NOT (A AND B)	NOT A OR NOT B
NOT (A OR B)	NOT A AND NOT B
A XOR B	(A AND NOT B) OR (NOT A AND B)

Universal gate construction

Tie both inputs of a NAND gate to A to obtain NOT A. Feed NAND(A,B) into another NAND with tied inputs to obtain A AND B. These constructions prove functional completeness, not optimality.

Sources [1, 32] / Reference directory

CHAPTER 07 / DEEP DIVE 2

Multiplexers, decoders and encoders

A multiplexer connects one selected input to an output. For two inputs A and B with select S, output Y = (NOT S AND A) OR (S AND B). Wider multiplexers can choose among more sources and are commonly used when an arithmetic unit must select operands or a register must choose its next value.

A decoder converts an encoded value into selected outputs, for example a two-bit address into four one-hot selection lines. An encoder performs a related reverse operation; priority encoders define behavior when more than one input is active. Combinational circuits can exhibit transient changes during propagation, so timing must be considered before outputs are sampled.

TECHNICAL REFERENCE

S	Y
0	A
1	B

Decode address 10

A two-to-four decoder receiving binary 10 activates output 2, assuming ordinary binary indexing and active-high outputs. A memory array may use decoded selections to enable a row, although real arrays include additional circuitry.

Sources [1] / Reference directory

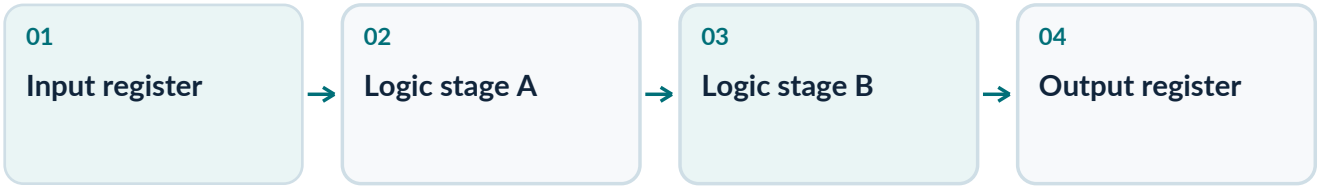
CHAPTER 07 / DEEP DIVE 3

Combinational logic and critical paths

Combinational circuit outputs depend on current inputs, once signals have had enough time to propagate. A path is a chain of gates and wires from an input or register to an output or register. Propagation delay varies with loading, process, voltage, temperature and the transition being measured.

A critical path is one that limits the timing target under the analysis assumptions. Parallel hardware can shorten a sequential dependency chain, at the possible cost of additional area and power. Gate count alone is therefore a poor predictor of speed. Designers also check short paths, because data that changes too early can violate a receiving register's hold requirement.

CONCEPT DIAGRAM



Timing calculation

Assume clock-to-Q = 80 ps, two logic delays of 160 and 210 ps, setup = 50 ps and uncertainty = 30 ps. The minimum modeled period is 530 ps, giving about 1.89 GHz before other constraints. The values are illustrative.

Sources [1, 32] / Reference directory

CHAPTER 08 / OVERVIEW

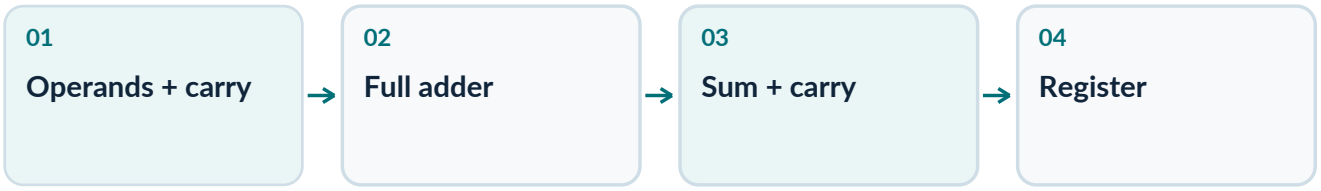
Arithmetic circuits, clocks and state

A full adder receives two operand bits and a carry-in bit. It produces S = A XOR B XOR Cin and Cout = (A AND B) OR (Cin AND (A XOR B)). Chaining full adders creates a multi-bit ripple-carry adder; faster alternatives use parallel carry computation at the cost of extra hardware. Subtraction can reuse addition by complementing the second operand and adding one.

Combinational logic alone cannot remember previous inputs. Sequential circuits add storage, for example latches and edge-triggered flip-flops. A register groups several storage elements to hold a multi-bit word. A finite-state machine uses stored state plus current inputs to compute its next state and outputs. Control units, communication protocols and device controllers all rely on this idea.

In a synchronous circuit a clock coordinates when state is sampled. Data must be stable for specified setup and hold intervals around the sampling edge. Signals crossing unrelated clock domains require deliberate synchronization; otherwise metastability can cause unpredictable behavior. Designers must also account for clock skew and distribution delay. Many modern systems contain several independently clocked regions rather than a single universal clock.

CONCEPT DIAGRAM



A four-bit addition

0101 (5) + 0011 (3) = 1000 (8). Carries propagate from lower to higher positions. In an unsigned four-bit field this is valid; in signed four-bit two's complement it overflows the range -8 to 7.

Sources [1] / Reference directory

CHAPTER 08 / DEEP DIVE 1

Full adders, subtractors and carry networks

A full adder combines three one-bit inputs into a sum bit and carry bit. The carry records whether at least two inputs are one. Repeating this structure across bit positions implements addition with a ripple carry. The dependency on lower bits can make the longest carry path expensive as width grows.

Carry-lookahead and parallel-prefix techniques compute groups of propagate and generate conditions so carry information can travel through a shallower logical network. Subtraction can be represented as A + NOT B + 1 for fixed-width two's complement arithmetic. Arithmetic hardware may additionally compute flags or comparison conditions, depending on the ISA.

TECHNICAL REFERENCE

A B Cin	Sum	Cout
0 0 0	0	0
0 1 1	0	1
1 0 1	0	1
1 1 1	1	1

Subtract with a four-bit adder

6 - 3 becomes 0110 + 1100 + 1 = 1 0011. Retain the low four bits, 0011, representing 3. The carry beyond the word is discarded under modulo-16 arithmetic.

Sources [1] / Reference directory

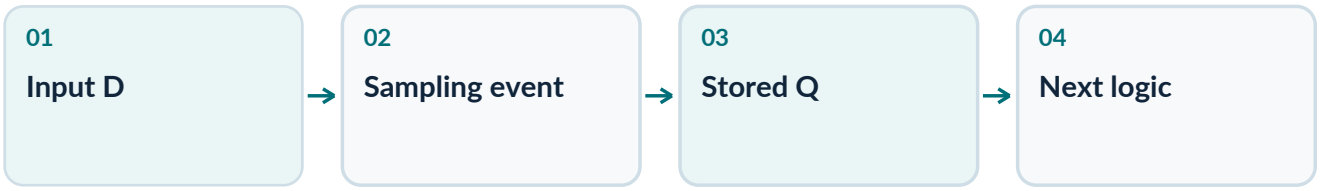
CHAPTER 08 / DEEP DIVE 2

Latches, flip-flops and register files

A latch can follow its input while an enable condition holds, then preserve a value when the enable changes. An edge-triggered flip-flop samples its input around a clock edge and preserves the result between sampling events. Both require electrical timing conditions for reliable operation.

A register groups storage elements to hold a word. A register file adds selection and access ports, allowing an instruction to identify operand registers and destinations. Additional simultaneous reads or writes can make the storage circuitry more expensive. Reset and enable signals control initialization and selective updates where the design requires them.

CONCEPT DIAGRAM



A three-cycle trace

Suppose a rising-edge register starts at 0 and D is 1, then 0, then 1 before three edges. If setup and hold conditions are met, Q becomes 1, 0, 1 at those edges. Between edges the ideal register holds its last value.

Sources [1, 32] / Reference directory

CHAPTER 08 / DEEP DIVE 3

Finite-state machines and hardware control

A finite-state machine maintains one of a finite set of states. A transition function combines current state and input to choose the next state. Output logic can depend on state alone or on state and input. State encoding determines how those abstract states are represented by storage bits.

Controllers use these machines to sequence operations. A memory controller, for example, can track whether it is idle, waiting for a response or completing a request. Real systems may have many interacting state machines. Verification checks reachable states, legal transitions and behavior when requests arrive in unexpected orders.

TECHNICAL REFERENCE

State	Input condition	Next state
Idle	Request	Busy
Busy	Not done	Busy
Busy	Done	Complete
Complete	Acknowledged	Idle

A request handshake

A requester raises valid and holds its data until the receiver raises ready on a sampling edge. Only when both are asserted is a transfer counted. This avoids losing requests when the receiver is temporarily busy.

Sources [1] / Reference directory

CHAPTER 09 / OVERVIEW

CPU anatomy and the instruction set

An instruction set architecture (ISA) defines the contract visible to machine-code programs: instruction encodings, registers, supported operations, address behavior and related rules. Examples include x86-64, AArch64 and RISC-V. Microarchitecture describes how a particular implementation realizes that contract. Two processors with the same ISA can have very different pipelines, caches, power budgets and execution throughput.

A simplified CPU contains a program counter (PC), instruction-fetch logic, decoder, register file, arithmetic and logic units (ALUs), load/store logic and control circuitry. The PC identifies the next instruction in the architectural execution model. A register file holds operands and results. Execution units perform operations such as addition, comparison or shifts, while memory instructions transfer values between registers and addressed memory.

Modern cores may contain separate execution units for integers, floating point and vectors. Their memory hierarchy can include translation caches and multiple data or instruction cache levels. Not every instruction uses every component. Interrupts and exceptions may redirect control flow; privileged mechanisms let the operating system handle events while restricting access from ordinary programs. The diagram is a conceptual path, not a universal physical layout.

CONCEPT DIAGRAM



Register operation

In RISC-V, add x7, x5, x6 reads registers x5 and x6, sums the values modulo the register width, and writes x7. No data-memory load is implied by this instruction.

Sources [1, 9] / Reference directory

CHAPTER 09 / DEEP DIVE 1

CPU architectural state and register roles

Architectural state is the information needed to describe the program-visible progress of a processor. It includes general registers, an instruction address and any other state defined by the ISA. Privileged execution adds control registers and status fields used for exceptions, interrupts and memory protection.

A register can have a conventional software role even if the hardware treats it as a general-purpose register. Calling conventions assign registers for arguments, return values and temporary storage. In RISC-V, x0 is architecturally zero, whereas other roles such as stack pointer x2 arise from the ABI convention. The distinction matters when reading assembly.

TECHNICAL REFERENCE

RISC-V name	Role
x0	Constant zero
x1 / ra	Return address (ABI)
x2 / sp	Stack pointer (ABI)
x10 / a0	Argument or return (ABI)

State after arithmetic

If x6 contains 7 and x7 contains 5, ADD x5,x6,x7 writes 12 to x5 in RV32I. The source registers remain unchanged. A real core may use internal physical registers, but the committed architectural result follows the ISA.

Sources [9, 33] / Reference directory

CHAPTER 09 / DEEP DIVE 2

Instruction decoding and datapath controls

An instruction word encodes an operation and its operands. A decoder examines fields such as opcode and function bits and determines which functional units and register addresses are needed. The decoder may generate direct control signals or translate the instruction into internal operations, depending on the implementation.

A datapath contains the registers, arithmetic units, multiplexers and wires through which operand values travel. Control logic selects inputs, determines an operation and decides whether a result is written. Separating control from data makes the design easier to reason about, even though both are ultimately circuits on the chip.

CONCEPT DIAGRAM



Choose an ALU input

For ADDI x5,x6,12, one ALU input receives x6 and the other receives a sign-extended immediate. For ADD x5,x6,x7, the second input instead receives x7. A multiplexer can implement that selection in a simple datapath.

Sources [1, 9] / Reference directory

CHAPTER 09 / DEEP DIVE 3

ISA design: x86-64, Arm and RISC-V

An ISA defines the software-visible machine, not a particular core design. x86-64 has variable-length instruction encodings and a long compatibility history. AArch64 uses fixed 32-bit instruction encodings. The RISC-V base integer ISA uses 32-bit instructions, with optional compressed 16-bit encodings and other extensions.

A load-store architecture restricts ordinary arithmetic to registers and uses separate memory-access instructions. Other ISAs can express memory operands in some arithmetic instructions. Modern implementations of different ISAs may nevertheless use similar internal techniques such as multiple execution units and caches. Neither instruction count nor instruction length alone determines performance.

TECHNICAL REFERENCE

ISA	Example design trait
x86-64	Variable-length encodings
AArch64	32-bit instructions
RV32I	32-bit base instructions

Compare compiler targets

Compile a simple function for two supported targets and inspect the generated assembly. Compare the number of loads, arithmetic instructions and branches. A fair speed comparison still requires actual hardware, matched inputs and optimization settings.

Sources [9, 33, 34, 35] / Reference directory

CHAPTER 10 / OVERVIEW

A real RISC-V instruction, bit by bit

RV32I is a base integer instruction set with 32 architectural integer registers, x0 through x31. Register x0 always reads as zero; writes to it do not change its value. Most base instructions are 32 bits wide. Optional extensions can introduce other operations and, in the compressed extension, shorter instruction encodings.

For an R-type ADD instruction, bits 31:25 specify funct7, 24:20 rs2, 19:15 rs1, 14:12 funct3, 11:7 rd and 6:0 opcode. For add x7,x5,x6, the fields are 0000000, 00110, 00101, 000, 00111 and 0110011. Combining them yields 0x006283B3. This hexadecimal word is an instruction, not the same thing as its byte sequence in a little-endian memory representation.

Suppose x5 = 7 and x6 = 5. The fetch mechanism obtains the instruction, decode recognizes ADD and the register indices, operands enter an execution unit, and the result 12 becomes the new architectural value of x7. For this ordinary instruction, execution proceeds to the next sequential address. A real superscalar processor may overlap these actions with other instructions, but must preserve the specified architectural behavior.

TECHNICAL REFERENCE

funct7	rs2	rs1	funct3	rd	opcode
0000000	00110	00101	000	00111	0110011
31:25	24:20	19:15	14:12	11:7	6:0

Instruction bytes

0x006283B3 becomes B3 83 62 00 in little-endian byte order. With x5 = 7, x6 = 5, the architectural result is x7 = 12.

Sources [9] / Reference directory

CHAPTER 10 / DEEP DIVE 1

RISC-V load and store instructions

In the RV32I load-store model, memory is accessed by dedicated instructions. A load computes an effective address from a base register plus a sign-extended immediate, reads the specified width, and writes a register result. Byte and halfword loads can sign-extend or zero-extend depending on their opcode variant.

A store also computes an effective address but writes data from a source register instead of producing a destination-register result. Access permissions, alignment and the execution environment can influence whether an operation completes or raises an exception. The meaning of the memory bytes depends on the selected access width and byte order.

TECHNICAL REFERENCE

Instruction	Meaning
lw x5, 4(x6)	Load 32-bit word
lb x5, 0(x6)	Load signed byte
lbu x5, 0(x6)	Load unsigned byte
sw x5, 4(x6)	Store 32-bit word

Sign extension

If the addressed byte is 0xFE, LB in RV32I yields 0xFFFFFFFE, representing -2 as signed. LBU yields 0x000000FE, representing 254. The byte in memory is identical.

Sources [9] / Reference directory

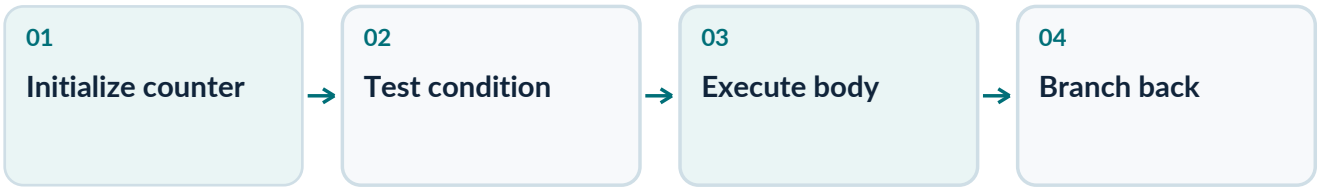
CHAPTER 10 / DEEP DIVE 2

Branches, jumps and loops

A conditional branch tests a relationship between register values and chooses whether the next instruction address is a target or the normal sequential address. RISC-V includes equality and signed or unsigned ordering comparisons. Branch immediates encode PC-relative offsets with an alignment constraint.

Unconditional jumps transfer control without a condition. JAL also writes a link address to a destination register, which can support function calls. JALR calculates a target using a register and immediate, enabling returns and indirect calls. A loop is not a special physical structure: it is a sequence of instructions whose control flow returns to an earlier address.

CONCEPT DIAGRAM



Loop invariant

Set x5 = 3 and x6 = 0. At each iteration add x5 to x6, decrease x5 by 1 and branch while x5 is nonzero. The accumulated values are 3, 5 and 6; the result is the sum from 1 through 3.

Sources [9] / Reference directory

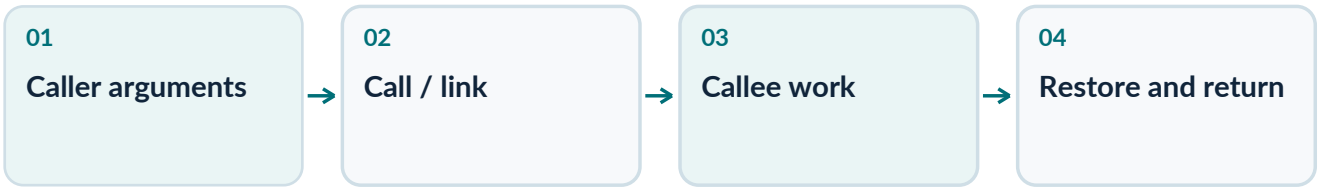
CHAPTER 10 / DEEP DIVE 3

Function calls and the stack convention

A calling convention defines how separately compiled functions exchange arguments and results and which registers must be preserved. A caller can use call-clobbered registers freely but must save any needed values across a call. A callee must restore callee-preserved registers it changes. Leaf functions may need no stack frame.

A stack is a memory region used for per-call storage, such as saved registers and local values that cannot remain in registers. Return addresses may stay in a link register until a nested call makes saving necessary. Stack alignment and layout are ABI requirements, not universal properties of all CPUs or languages.

CONCEPT DIAGRAM



Nested calls

Function f calls g, and g calls h. If g overwrites its link register when calling h, it must preserve its original return address somewhere first, commonly in its stack frame. Otherwise it cannot reliably return to f.

Sources [9, 36] / Reference directory

CHAPTER 11 / OVERVIEW

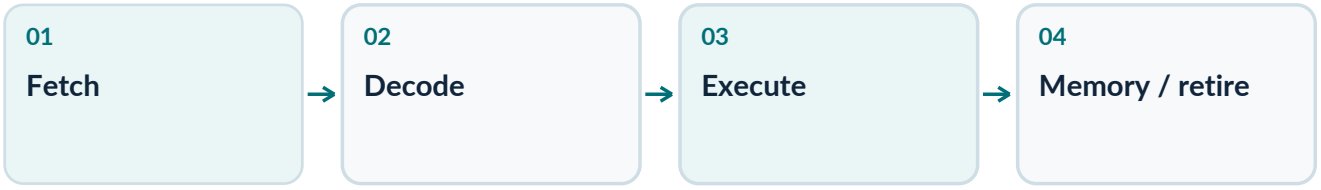
Pipelining, speculation and parallel execution

A common educational pipeline divides instruction handling into fetch, decode, execute, memory access and writeback. Each stage can work on a different instruction. Once filled, an ideal single-issue pipeline may complete one instruction per cycle, even though an individual instruction takes several stages. Real pipelines experience hazards and stalls.

Data hazards arise when an operation depends on an earlier result. Forwarding can route a result directly to a consumer; some dependencies require waiting. Structural hazards occur when operations compete for a resource. Control hazards occur when the next instruction address depends on a branch. Branch prediction guesses the path early; a wrong prediction requires discarding or correcting speculative work. Modern processors may execute operations out of program order when dependencies permit and retire results in a way consistent with architectural rules.

Multiple cores execute independent instruction streams, and simultaneous multithreading shares core resources between hardware threads. SIMD instructions apply one operation across several data elements. None of these mechanisms guarantees a fixed speedup: parallelism, dependency chains, memory traffic, branch behavior and workload balance determine what can be achieved. Speculation also has security implications because microarchitectural traces may persist even when architectural results are discarded.

CONCEPT DIAGRAM



Ideal pipeline example

For five stages and ten independent instructions, an ideal one-instruction-per-cycle pipeline needs 5 + 10 - 1 = 14 cycles, compared with 50 cycles if every instruction occupies all five stages serially. This ignores hazards and overhead.

Sources [1, 9] / Reference directory

CHAPTER 11 / DEEP DIVE 1

The five-stage pipeline model

A teaching pipeline divides instructions into fetch, decode, execute, memory and writeback stages. Registers between stages hold intermediate values. Overlapping instructions increases the number completed per unit time, even if an individual instruction still passes through every stage.

Real CPUs may have more stages, multiple instructions in each stage or specialized paths. The five-stage model illustrates the difference between latency and throughput, not a universal CPU layout. Hazards arise when instructions need the same resources, depend on previous results or change the expected control flow.

TECHNICAL REFERENCE

Cycle	Stage of I1	Stage of I2	Stage of I3
1	Fetch	-	-
2	Decode	Fetch	-
3	Execute	Decode	Fetch
4	Memory	Execute	Decode
5	Writeback	Memory	Execute

Ideal throughput

For five one-cycle stages, a sequence of 100 independent instructions takes 104 cycles in the ideal model. Without overlap it would take 500 cycles under the same stage-duration assumption. Stalls, unequal stage delays and overhead reduce the ideal gain.

Sources [1, 32] / Reference directory

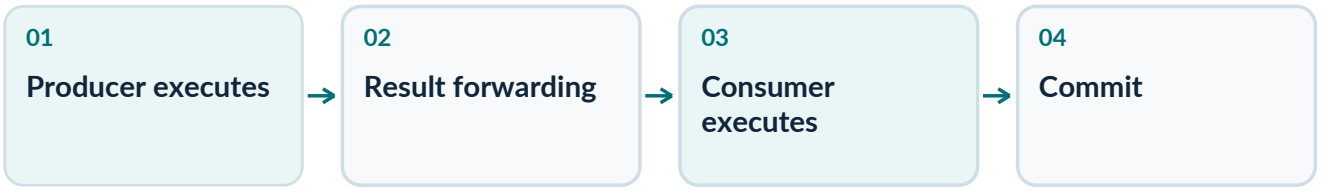
CHAPTER 11 / DEEP DIVE 2

Data hazards, forwarding and stalls

A read-after-write dependency occurs when an instruction needs a value produced by an earlier instruction. A simple pipeline can forward an available result directly to a dependent execution stage rather than waiting for the register file write. Not every result is available early enough to avoid a stall.

Structural hazards arise when two operations compete for a hardware resource. Control hazards arise when a branch changes the instruction stream. Hardware interlocks can delay instructions until conditions are safe. More advanced processors dynamically schedule ready instructions, while preserving architecturally required results and ordering.

CONCEPT DIAGRAM



Load-use hazard

Suppose LW x5,0(x6) is followed by ADD x7,x5,x8. In a classic five-stage teaching design, the load's data is available after memory access, too late for the following instruction's normal execute stage. One stall may be needed even with forwarding; actual cores differ.

Sources [1, 32] / Reference directory

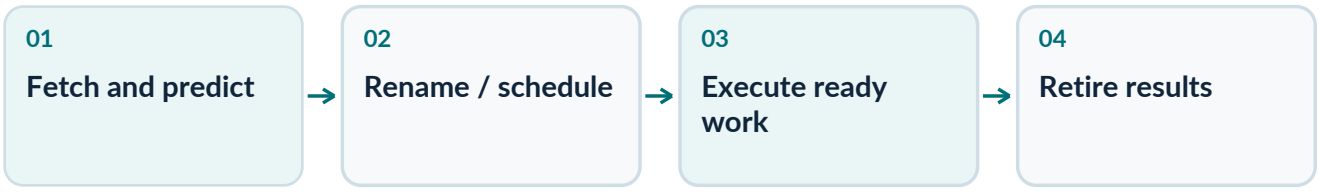
CHAPTER 11 / DEEP DIVE 3

Out-of-order execution and branch prediction

An out-of-order processor can execute independent instructions while an older one is waiting for a resource. Register renaming maps architectural names to physical storage to avoid certain false dependencies. Scheduling logic issues ready operations, and retirement machinery makes results architecturally visible in the required order.

Branch prediction estimates control flow so the front end can continue obtaining instructions. Speculative operations may be discarded if a prediction proves incorrect. Such speculation can improve throughput but consumes energy and creates security considerations. The prediction mechanism does not change the program's specified branch condition.

CONCEPT DIAGRAM



Illustrative branch penalty

With 20% branch instructions, 5% mispredictions among branches and a 12-cycle penalty, the average added cost is 0.20 x 0.05 x 12 = 0.12 cycles/instruction in a simplified additive model. Real penalties can overlap other delays.

Sources [1, 34] / Reference directory

CHAPTER 12 / OVERVIEW

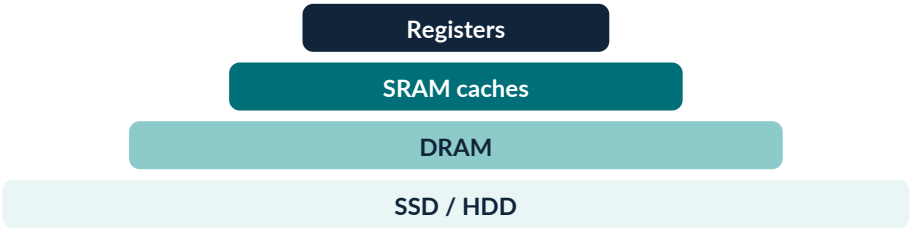
Registers, caches, DRAM and locality

A memory hierarchy balances capacity, cost, bandwidth and access latency. Registers are small and directly named by instructions. SRAM caches hold copies of blocks from lower memory levels. DRAM provides larger main memory, using charge-based storage cells that require periodic refresh. Persistent storage offers much greater capacity but is normally far slower than CPU-local memory for individual accesses.

Caches exploit temporal locality (recently used data may be reused) and spatial locality (nearby data may be needed soon). An address is interpreted using a block offset, an index or set selection, and a tag. A cache hit finds the requested block; a miss triggers a lookup or transfer from another level. Associativity and replacement choices influence which blocks remain. Write-through and write-back policies govern when stores reach the next level.

Shared-memory multiprocessors use coherence mechanisms so cores can coordinate cached copies of a memory location. Coherence is distinct from the memory consistency model, which constrains observable ordering of operations across addresses. Software synchronization is still required for safely sharing mutable data. Cache sizes and access times are implementation-specific; no single latency table describes all computers.

CONCEPT DIAGRAM



Typically larger capacity and higher access latency toward the bottom. Not to scale.

Average memory access time

With a 1 ns cache lookup, 5% miss rate and 80 ns additional miss penalty, AMAT = 1 + 0.05 x 80 = 5 ns. These are illustrative assumptions, not measurements of a specific CPU.

Sources [1, 10] / Reference directory

CHAPTER 12 / DEEP DIVE 1

SRAM, DRAM and the memory controller

SRAM commonly retains a bit using a feedback circuit as long as power remains available. DRAM represents bits with stored charge that gradually leaks, requiring periodic refresh. SRAM cells generally occupy more area per bit, so SRAM is useful for relatively small fast memories while DRAM is economical for larger main memory. Both are volatile in their ordinary forms.

A DRAM controller schedules commands across channels, ranks and banks. Activating a row makes its contents available in a row buffer; subsequent column accesses may reuse that open row. A different row in the same bank can require additional operations. Timing constraints, refresh and competing requests determine observed latency.

CONCEPT DIAGRAM



Row locality

Reading consecutive locations that reside in the same open DRAM row may avoid repeated row activations. The exact mapping of addresses to banks and rows is implementation-dependent; sequential access alone does not guarantee a particular timing.

Sources [1, 10, 11] / Reference directory

CHAPTER 12 / DEEP DIVE 2

Cache mapping, tags and replacement

A cache stores copies of selected memory blocks. A line typically contains multiple adjacent bytes, a tag and status bits. In a set-associative cache, address bits identify a set, while tags distinguish possible blocks in that set. The number of ways limits how many different blocks can coexist there.

A hit supplies data from the cache; a miss requires obtaining the block from another level. Replacement policies choose a line to evict when space is needed. Write-back caches can retain modified data until a writeback, while write-through designs propagate writes sooner. Policies differ across implementations.

TECHNICAL REFERENCE

Example cache	Value
Capacity	32 KiB
Line size	64 bytes
Associativity	8 ways
Number of sets	64

Address breakdown

For the example cache, a byte offset uses 6 bits because 64 = 2^6. The set index also uses 6 bits. In a simple physically indexed model, the remaining high physical address bits form the tag. Actual indexing may use other mappings.

Sources [1, 32] / Reference directory

CHAPTER 12 / DEEP DIVE 3

Locality, miss cost and prefetching

Temporal locality means recently accessed data is likely to be accessed again. Spatial locality means nearby addresses are likely to be accessed. Caches exploit both by retaining recent lines and fetching multiple adjacent bytes on misses. A data structure that scatters nodes can be less cache-friendly than contiguous arrays.

Hardware and software prefetchers try to bring data closer before demand arrives. A useful prefetch hides part of access latency; an inaccurate one wastes bandwidth or evicts useful data. Cache performance is influenced by access patterns, working-set size, competition and the number of outstanding requests.

TECHNICAL REFERENCE

Access pattern	Likely locality
Scan contiguous array	High spatial
Repeated small loop	High temporal
Random linked nodes	Often poor spatial

Average access time

With a 1 ns cache hit time, a 5% miss rate and 40 ns additional miss penalty, AMAT = 1 + 0.05 x 40 = 3 ns. This simplified serial average does not capture overlapping misses or multi-level caches.

Sources [1, 32, 34] / Reference directory

CHAPTER 13 / OVERVIEW

Virtual memory and address translation

Programs generally operate on virtual addresses rather than directly naming physical memory. Each process has a protected address space. A memory management unit (MMU) translates virtual addresses using operating-system-managed mapping structures and enforces access permissions such as read, write, execute and user accessibility.

Paging divides virtual and physical memory into fixed-size units. With 4 KiB pages, the low twelve address bits form the offset. The remaining bits select a virtual page; page-table entries identify the physical frame and relevant attributes. A translation lookaside buffer (TLB) caches translations, reducing the need to traverse page tables for each reference. Page-table structure and supported page sizes depend on the architecture.

When a mapping is missing or an access violates permissions, the processor can raise a page fault. The operating system may allocate a page, load file-backed contents, restore a swapped page, or reject the access. A page fault does not necessarily mean disk I/O. Virtual memory enables isolation, sharing, sparse mappings and demand allocation. It does not give a computer unlimited physical memory; heavy swapping can sharply reduce responsiveness.

CONCEPT DIAGRAM



4 KiB page example

Virtual address 0x12345 has page number 0x12 and offset 0x345. If page 0x12 maps to frame 0xAB, the translated physical address is 0xAB345 (assuming this simplified mapping and page size).

Sources [1, 10, 11] / Reference directory

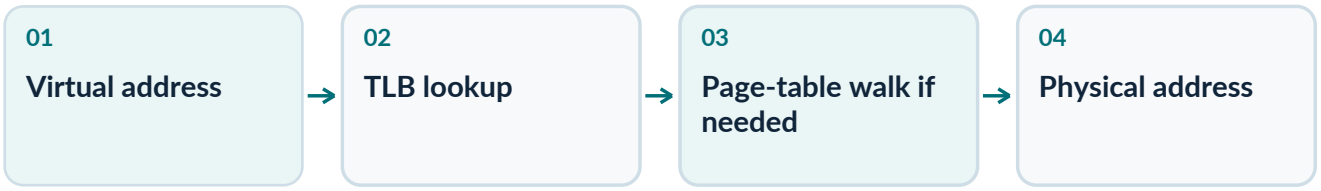
CHAPTER 13 / DEEP DIVE 1

Address spaces, pages and page tables

Virtual memory gives a process an address space whose addresses need not directly identify physical memory. A memory management unit translates virtual page numbers using mappings controlled by privileged software. The offset within a normal page remains the same after translation. Permissions can restrict reading, writing or executing.

Page tables organize mappings in memory, often as multiple levels so unused address ranges do not need full tables. A translation lookaside buffer caches recent translations. A TLB miss can trigger a hardware or software page-table walk depending on the architecture. It is different from a missing or disallowed mapping, which can cause a fault.

CONCEPT DIAGRAM



Page offset

With 4 KiB pages, 0x12345 has virtual page number 0x12 and offset 0x345. If that page maps to physical frame 0xAB, the physical address is 0xAB345, assuming the mapping allows the operation.

Sources [10, 11, 33] / Reference directory

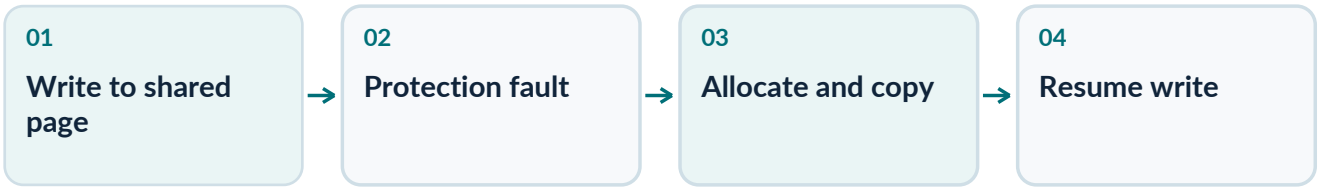
CHAPTER 13 / DEEP DIVE 2

Page faults, demand paging and copy-on-write

An access to an unmapped or protected virtual page can cause a processor exception. The kernel examines the cause and its own memory-management metadata. Some faults are expected, such as first access to a lazily allocated page; others indicate an invalid operation. The kernel may arrange a mapping, load data or terminate the offending operation.

Copy-on-write lets mappings initially share underlying pages while preventing direct modification. When a permitted logical writer attempts a write, the kernel can allocate a private copy and update the mapping. This can avoid copying memory that will never be changed. The optimization still requires careful accounting and synchronization.

CONCEPT DIAGRAM



Cost of lazy allocation

A program reserves 100 MiB of virtual address space but touches one 4 KiB page. Depending on OS policy, most of the reserved range may not yet need physical data pages. Reservation, commitment and residency are distinct concepts.

Sources [10, 11] / Reference directory

CHAPTER 13 / DEEP DIVE 3

Memory pressure, swap and working sets

A working set is the collection of memory pages actively needed over some interval. If active working sets exceed available RAM, the OS must reclaim pages. Clean file-backed pages can often be discarded and reread later; modified pages may require writeback. Anonymous memory may be moved to swap or handled by another policy.

Excessive page movement can cause thrashing: much time is spent retrieving and replacing pages rather than performing useful work. More virtual address space does not create more physical RAM. Memory compression, reclaim heuristics and process limits can help, but no mechanism removes the underlying capacity constraint.

TECHNICAL REFERENCE

Metric	What it indicates
Virtual size	Address range reserved
Resident size	Pages currently in RAM
Major faults	Faults requiring I/O on Linux
Swap use	Anonymous data in swap

Working-set mismatch

A 12 GiB actively accessed data set on a machine with only 8 GiB available RAM may repeatedly evict needed pages. Changing the algorithm to process smaller chunks may improve responsiveness more than adding processor cores.

Sources [10, 11] / Reference directory

CHAPTER 14 / OVERVIEW

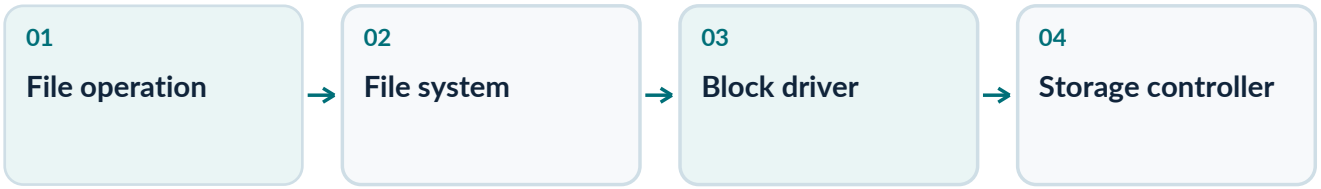
SSDs, hard disks and file systems

A hard disk uses rotating magnetic media and movable heads. Its access time depends partly on seeking and rotation, so sequential access and random access have different costs. NAND flash stores information through charge-related cell states and has no moving mechanical head. It is read and programmed in pages and erased in larger blocks; details vary by flash design.

An SSD controller maps logical block addresses to physical flash locations using a flash translation layer. Wear leveling distributes writes; garbage collection reclaims blocks; error-correction codes help recover data. The operating system can inform the device that certain logical blocks are no longer in use through discard mechanisms. NVMe is a storage command protocol commonly used over PCI Express; it is not itself the flash medium. Queues permit multiple outstanding commands.

A file system builds names, directories, metadata, access rules and allocation structures on top of storage. Journaling or copy-on-write techniques can help maintain consistency after interruptions, but their exact guarantees vary. A successful write call may mean the operating system accepted bytes into a cache, not that they survived a power failure. Applications that need durability must use appropriate synchronization and understand the guarantees of the entire storage stack.

CONCEPT DIAGRAM



Why backup matters

A file system may recover its metadata after a crash yet still lose recently modified application data. Backups protect against a different class of failures and should be independently restorable.

Sources [10, 12, 29] / Reference directory

CHAPTER 14 / DEEP DIVE 1

Magnetic disks, flash and block interfaces

A hard disk stores information magnetically on rotating platters. Access requires positioning heads and waiting for the desired sector to rotate beneath them. Sequential transfers can amortize these positioning costs. Solid-state storage has no rotating media but still has controller, flash-access and interface delays.

NAND flash is generally read and programmed at page granularity and erased in larger blocks. Updates often require writing elsewhere and reclaiming old locations later. The SSD controller exposes a logical block interface so software need not manage every physical flash page directly. NVMe defines a command interface designed for nonvolatile storage over suitable transports.

TECHNICAL REFERENCE

Characteristic	HDD	SSD
Physical medium	Magnetic	Flash typically
Moving parts	Yes	No
Random access cost	Mechanical delay	Controller and flash delay

Sequential versus random

Reading 1,000 nearby blocks can be much faster on a hard disk than reading 1,000 widely separated blocks. SSDs reduce mechanical penalties but random access still interacts with queue depth, flash parallelism and controller policies.

Sources [10, 12] / Reference directory

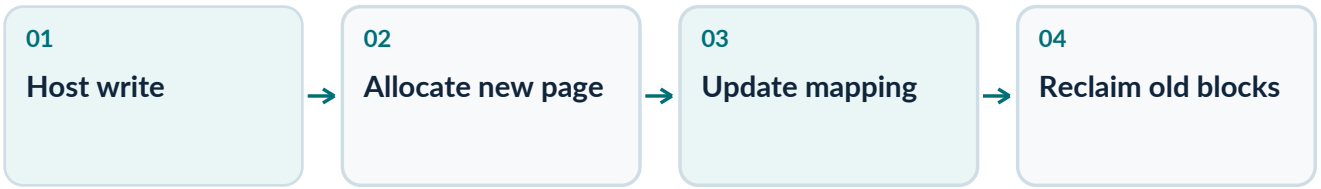
CHAPTER 14 / DEEP DIVE 2

SSD translation, garbage collection and endurance

A flash translation layer maintains a mapping from host-visible block addresses to physical flash locations. New writes may replace a mapping rather than overwrite the original physical page. Old pages become invalid, and garbage collection copies still-valid pages before erasing a block for reuse.

Wear leveling spreads erase cycles across the available flash. Overprovisioning provides extra physical capacity to assist management. Write amplification describes how much physical writing occurs relative to host writing. A discard or TRIM command can tell a device that logical blocks no longer contain data the host needs, subject to device and OS behavior.

CONCEPT DIAGRAM



Write amplification

If a host writes 10 GiB and the SSD internally programs 15 GiB while doing relocation, the illustrative write amplification is 1.5. It depends on workload, free space, controller design and measurement interval.

Sources [10, 12] / Reference directory

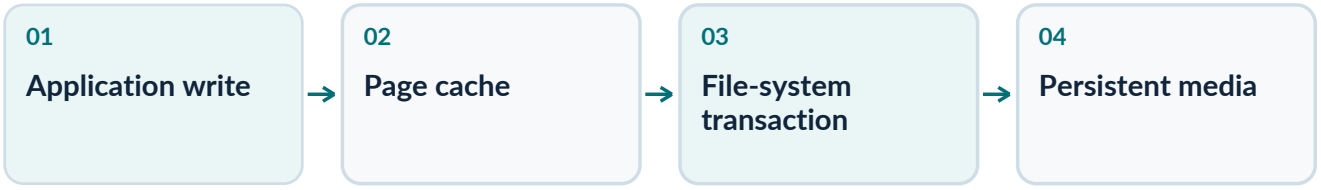
CHAPTER 14 / DEEP DIVE 3

File systems, journaling and durability

A file system organizes persistent blocks into named files, directories and metadata. Many designs use structures such as inodes or allocation tables to record ownership, permissions and block locations. The OS can cache file contents in memory and defer storage writes to improve throughput.

Crash consistency concerns what remains structurally valid after unexpected interruption. Journaling records selected changes in a log so recovery can replay or discard transactions according to the format. Copy-on-write file systems use other approaches. Neither technique automatically guarantees that the most recent application data has reached durable storage; application synchronization and device guarantees matter.

CONCEPT DIAGRAM



Save a document safely

An editor can write a temporary file, request appropriate synchronization, then rename it over the destination. Exact crash-durability guarantees depend on the file system and whether relevant file and directory metadata are synchronized. A successful write call alone is not a universal durability guarantee.

Sources [10, 29] / Reference directory

CHAPTER 15 / OVERVIEW

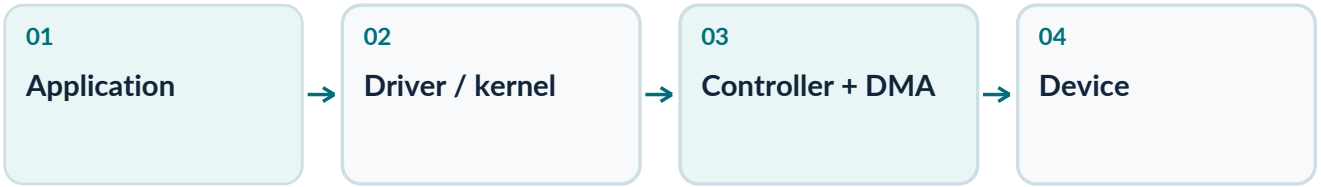
Motherboards, interconnects and input/output

The motherboard or system-on-chip connects computation, memory, storage and peripherals. Modern CPUs often integrate memory controllers and some PCI Express lanes. A platform controller or equivalent fabric may connect additional devices. Clock generation, voltage regulation and thermal management keep components within operating requirements. A system-on-chip combines many of these functions on a die or package.

Interconnects transfer information according to electrical and protocol rules. DDR memory channels link controllers to DRAM; PCI Express provides packet-based point-to-point links; USB connects many external peripherals. Bus widths, lane counts, encoding overhead, protocol revisions and device behavior affect usable bandwidth. Faster advertised link speeds do not automatically imply faster applications.

Device drivers program controllers using mechanisms such as memory-mapped I/O. Interrupts notify a CPU of events; polling checks status repeatedly. Direct memory access (DMA) lets a device move data to or from memory without the CPU copying every byte. The CPU and driver still configure transfers, handle completion and maintain appropriate synchronization. An IOMMU can translate and restrict device-visible addresses, supporting isolation and virtualization.

CONCEPT DIAGRAM



Receiving network data

A network adapter can place received packets into configured memory buffers using DMA. Completion is reported by an interrupt or polling mechanism, after which the OS network stack processes the data.

Sources [1, 13] / Reference directory

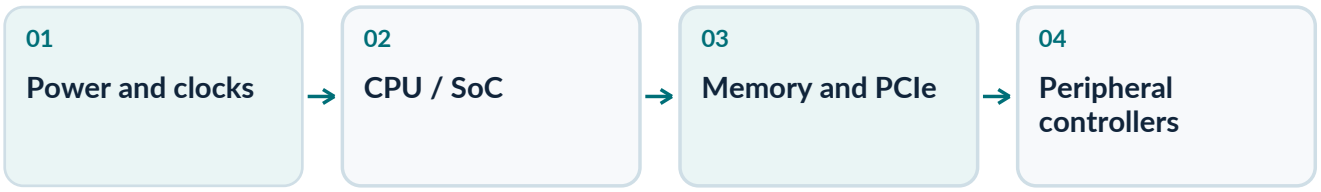
CHAPTER 15 / DEEP DIVE 1

Motherboard power, clocks and platform topology

A motherboard connects processors, memory, storage, external interfaces and power-delivery components. Voltage regulators supply suitable low-voltage power rails to chips. Clock sources and clock-distribution circuits support communication and internal timing. On many modern platforms, functions once assigned to separate chipset components are integrated into the processor package.

The topology determines which devices connect directly to the CPU or through other controllers. Some communication paths share bandwidth; others have dedicated links. A physical connector does not by itself reveal all electrical capabilities. Board layout, firmware configuration, processor support and device capabilities jointly determine the usable system.

CONCEPT DIAGRAM



Shared bandwidth

Two devices behind a shared upstream link can contend for that link when both transfer at once. Their individual advertised speeds do not prove that the system can sustain both at maximum speed simultaneously.

Sources [1, 13] / Reference directory

CHAPTER 15 / DEEP DIVE 2

PCI Express, USB and interconnect bandwidth

PCI Express is a packet-based interconnect with point-to-point links. A link may contain multiple lanes, and supported signaling generations determine raw transfer rates. Negotiated link width and speed depend on both endpoints and the physical path. Protocol overhead reduces useful application throughput below raw signaling capacity.

USB defines a family of host-device communication mechanisms. Connector shapes, supported signaling modes, power delivery and device functionality are separate characteristics. A particular connector may support multiple modes or only a subset. Device discovery and drivers allow the OS to communicate through these protocols.

TECHNICAL REFERENCE

Layer	Example
Physical	Cable or board traces
Link	Lanes and framing
Protocol	Requests and responses
Software	Device driver

Raw lane calculation

For a hypothetical link delivering 8 Gbit/s per lane before overhead, four lanes provide 32 Gbit/s raw, equivalent to 4 GB/s before encoding and protocol overhead. This is a teaching calculation, not a guarantee for any particular PCIe product.

Sources [13, 37, 38] / Reference directory

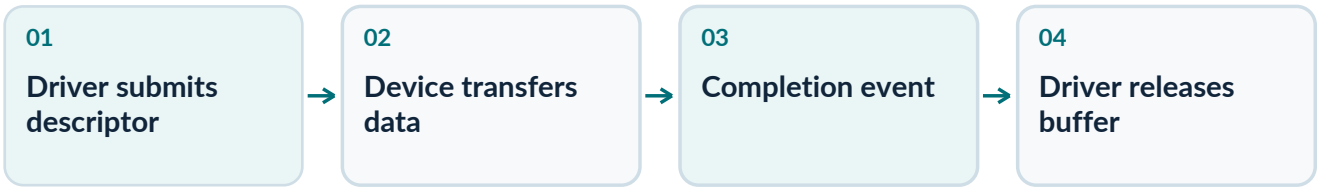
CHAPTER 15 / DEEP DIVE 3

Interrupts, DMA and device queues

Interrupts notify processors of events such as completed I/O or timers. Polling is another strategy: software checks a status source repeatedly. Interrupts can reduce idle checking, while polling can sometimes be useful for high event rates or predictable latency. A device driver chooses a strategy appropriate to the interface.

Direct memory access lets capable devices transfer data without a CPU executing each individual byte copy. Drivers arrange buffers and mappings, submit descriptors and handle completion. An IOMMU can constrain device-visible addresses, helping isolate DMA. Cache synchronization and ownership rules matter on systems that are not fully coherent.

CONCEPT DIAGRAM



Receive a network frame

A NIC may place received bytes into a driver-provided buffer using DMA and signal completion. The CPU then processes protocol headers and hands relevant data to software. It need not execute a separate copy instruction for each byte arriving on the wire.

Sources [13] / Reference directory

CHAPTER 16 / OVERVIEW

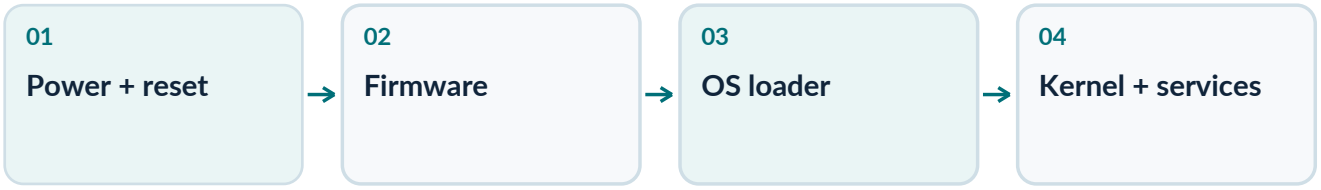
What happens when power is switched on?

Power supplies and voltage regulators establish the rails needed by hardware. Reset logic holds parts of the system in known initial conditions until requirements are met. On reset, a processor begins execution according to its architecture and platform design, typically from firmware stored in nonvolatile memory or mapped into its address space.

Firmware initializes essential components, including enough memory and device support to locate and execute the next stage. On many modern PCs, UEFI supplies standardized interfaces for boot applications and drivers. Its boot manager chooses a configured boot target, often an operating-system loader stored in an EFI system partition. Secure Boot, when enabled and correctly configured, verifies authorized images according to its trust policy. Other platforms may use very different firmware chains.

The loader prepares the operating-system kernel and any required startup data, then transfers control. The kernel initializes memory management, scheduling, interrupt handling and device drivers. It mounts or locates the root file system and launches the initial user-space process. Services start and a login screen or graphical desktop may appear. Embedded devices sometimes boot directly into an application or a real-time operating system, without a desktop.

CONCEPT DIAGRAM



Firmware versus OS

Firmware prepares and hands off the platform; the operating system manages ongoing resource use. Some firmware interfaces remain available after boot, but they do not replace the OS kernel.

Sources [14, 15] / Reference directory

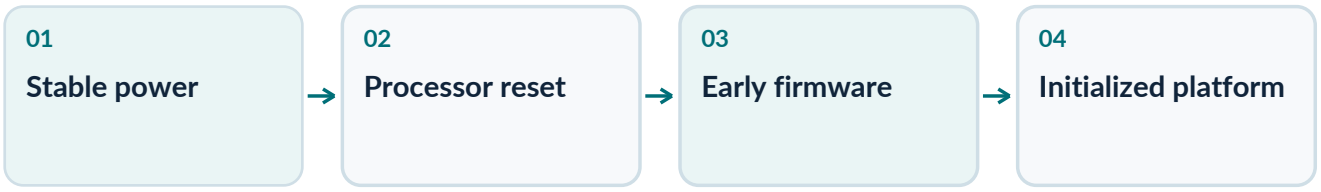
CHAPTER 16 / DEEP DIVE 1

Reset, firmware and hardware initialization

After power and reset conditions are established, a processor begins from architecture-defined initial state and an implementation-specific startup environment. Firmware performs early setup, which can include memory training, device discovery and configuration. Some platforms contain several independent controllers with their own firmware.

Firmware bridges the hardware's initial conditions and the environment expected by later software. It may initialize memory before ordinary application code can use it. Recovery mechanisms and authenticated updates reduce the risk of firmware corruption. Exact startup stages vary among desktop computers, embedded systems and phones.

CONCEPT DIAGRAM



Why RAM may need training

A memory interface must sample high-speed signals at suitable times. Startup training can adjust parameters to account for physical properties of the platform. A failed initialization can prevent the machine from reaching its normal boot menu.

Sources [14, 15] / Reference directory

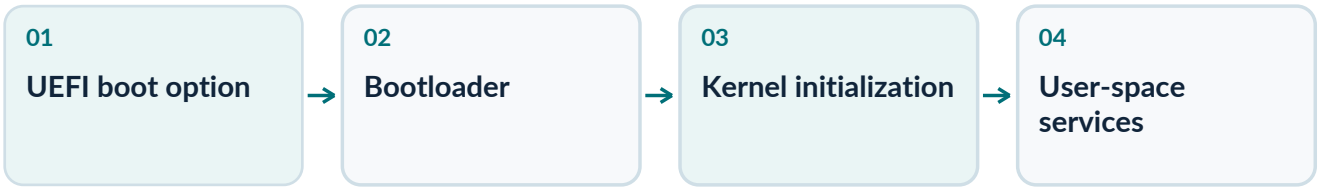
CHAPTER 16 / DEEP DIVE 2

UEFI, bootloaders and the kernel handoff

UEFI specifies interfaces between platform firmware and software that runs in its environment. On a typical UEFI PC, a boot manager chooses a configured boot option and loads an appropriate executable. The bootloader can locate the operating-system kernel and associated boot data, then prepare the environment for the kernel's own initialization.

A kernel establishes memory-management structures, interrupt handling and drivers as needed. It can then initialize its file systems and start the first user-space services. The details differ by operating system. A bootloader is not generally the operating system itself, and firmware services are not identical to ordinary system calls.

CONCEPT DIAGRAM



Boot option versus partition

A firmware boot entry may identify a particular executable on an EFI System Partition. The executable then decides what kernel or other target to load. The firmware entry, bootloader file and kernel image are separate objects.

Sources [14] / Reference directory

CHAPTER 16 / DEEP DIVE 3

Secure boot, measured boot and recovery

Secure boot uses a trust policy to decide whether selected boot components are acceptable, commonly through signature verification. It aims to prevent unauthorized startup code from executing in that chain. The details depend on which components are covered and how keys and revocation lists are managed.

Measured boot records cryptographic measurements of components, potentially using a hardware trust module. Measurements can support later attestation but do not necessarily block execution. Recovery mechanisms help restore an acceptable firmware or boot image after failure. Security relies on protecting update mechanisms, trust anchors and configuration as well as verifying binaries.

TECHNICAL REFERENCE

Mechanism	Primary purpose
Secure boot	Enforce trust policy
Measured boot	Record evidence
Recovery	Restore acceptable state

> **Distinguish outcomes**
>
> A signed component can pass a signature check while still containing a software bug. A recorded measurement can show which component was loaded but does not by itself prove it was safe. Trust mechanisms complement testing and updates rather than replacing them.

Sources [14, 15] / Reference directory

CHAPTER 17 / OVERVIEW

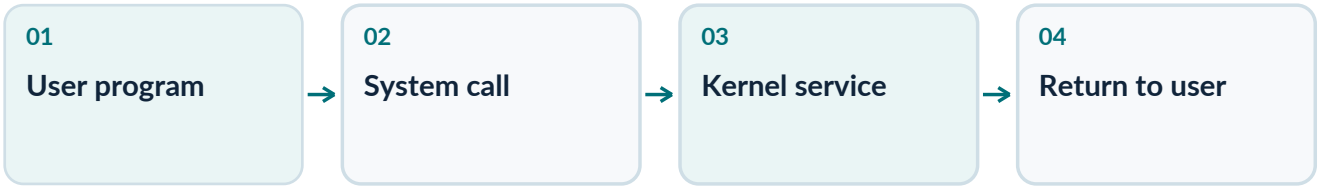
Operating systems: processes and system calls

An operating system provides abstractions over hardware, manages shared resources and enforces protection boundaries. A process is a running program context with an address space, resources and execution state. Threads are execution streams within a process; they usually share the process's address space while having separate register contexts and stacks.

A scheduler chooses runnable threads for available CPU cores. A timer interrupt or another event can cause the kernel to reconsider which thread should run. A context switch saves and restores relevant execution state. The operating system balances responsiveness, throughput, fairness, deadlines or other goals according to its policies. A thread blocked on input does not need to consume CPU time while it waits.

Ordinary applications operate with restricted privileges. A system call requests a kernel service, for example reading a file, allocating an address mapping or creating a process. Special instructions or architecture-defined mechanisms enter the kernel, which validates the request and returns results or errors. Exceptions report synchronous events such as invalid memory references; interrupts are typically asynchronous notifications. The kernel also manages drivers, networking, memory and security checks.

CONCEPT DIAGRAM



Reading a file

A program requests bytes from an open file. The kernel checks permissions and offsets, may use cached pages or initiate device I/O, then returns available data or an error. The calling thread may block while waiting.

Sources [10, 11] / Reference directory

CHAPTER 17 / DEEP DIVE 1

Processes, threads and protected execution

A process is an OS abstraction that associates executing code with resources such as virtual memory and open files. A thread represents an execution context with registers, instruction position and scheduling state. Multiple threads in one process normally share an address space but have their own stacks and execution contexts.

User mode restricts privileged operations; kernel mode permits the OS to configure memory translation, interrupts and hardware resources. A system call provides a controlled entry into kernel code. The kernel validates relevant arguments and permissions before operating on shared resources. A process boundary is useful for isolation but depends on correct enforcement.

TECHNICAL REFERENCE

Resource	Typically shared by threads?
Address space	Yes
Open file table	Often yes
CPU registers	No
Thread stack	No

Two editor windows

An application can use one process with multiple threads or separate processes for different windows. The OS can schedule these contexts independently. A failure in a shared process may affect all its threads, unlike stronger process isolation.

Sources [10, 11] / Reference directory

CHAPTER 17 / DEEP DIVE 2

Scheduling, context switches and CPU time

A scheduler selects runnable execution contexts for available processors. Workloads can be interactive, batch-oriented or subject to real-time constraints. Scheduling policies trade off responsiveness, throughput, fairness and deadlines. Waiting for disk or network I/O usually removes a thread from the runnable set until it can progress.

A context switch saves the execution state of one thread and restores another. Some memory translation state may also change. Switching has direct bookkeeping costs and indirect cache or TLB effects. On a multicore system, each core can run its own context concurrently, subject to shared-resource contention.

TECHNICAL REFERENCE

State	Meaning
Running	Executing on a CPU
Runnable	Ready for a CPU
Blocked	Waiting for an event
Terminated	Finished execution

Round-robin teaching model

Three CPU-bound tasks A, B and C receive 4 ms time slices on one core. Ignoring overhead, the order is A, B, C, A, B, C. Each receives about one-third of CPU time, though actual schedulers use more detailed policies.

Sources [10, 11] / Reference directory

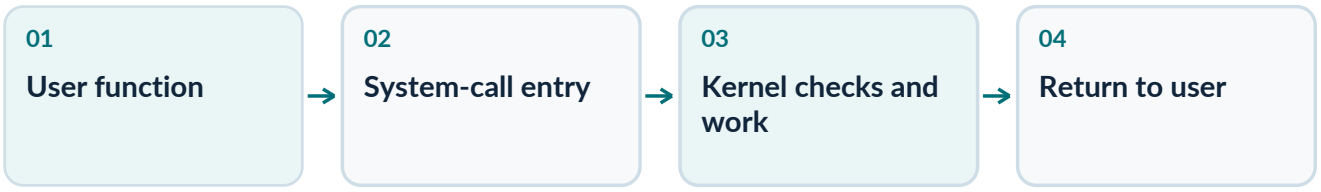
CHAPTER 17 / DEEP DIVE 3

System calls, exceptions and device services

System calls let applications request operations that require OS mediation. A calling convention identifies the operation and arguments; a dedicated instruction or equivalent mechanism transfers control to an appropriate handler. The kernel validates user pointers, looks up resource handles and applies access rules before returning a result or error.

Exceptions arise synchronously from an instruction or execution condition, while hardware interrupts are generally asynchronous relative to program instructions. Both use architecture-defined control transfers. The OS can resolve some conditions and resume execution, deliver an error to the application or terminate it. Not every library call requires a system call.

CONCEPT DIAGRAM



Read a file

A library read wrapper can invoke a system call with a file descriptor, buffer address and length. Cached data may satisfy the request without physical disk access. If storage I/O is needed, the calling thread may wait while the OS runs other work.

Sources [9, 10, 11, 33] / Reference directory

CHAPTER 18 / OVERVIEW

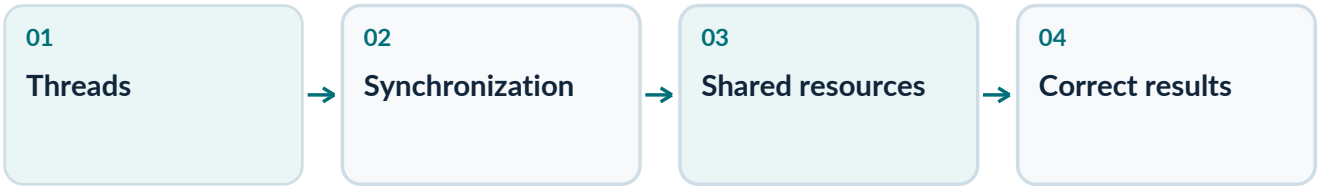
Concurrency, synchronization and virtualization

Concurrency means tasks make progress during overlapping time periods. Parallelism means operations execute simultaneously on multiple resources. A single core can interleave concurrent threads; several cores may execute them at the same time. Shared mutable data introduces races if access is not coordinated. A read-modify-write expression is not necessarily one indivisible operation.

Locks protect critical sections, atomic operations provide specific indivisible behaviors, and condition variables or message queues help threads coordinate. Deadlock can occur when tasks wait cyclically for resources. Memory ordering matters because compilers and processors can reorder operations within allowed rules. Correct synchronization gives the guarantees needed for communication; simply adding a delay is not reliable.

Virtual machines run guest operating systems with virtual hardware under a hypervisor. Hardware virtualization extensions help trap or manage privileged operations and translate guest memory. Containers instead typically isolate applications through operating-system mechanisms while sharing the host kernel. They are not identical security boundaries, and neither model automatically makes a workload secure. Emulation may translate or interpret instructions for another architecture, often with additional cost.

CONCEPT DIAGRAM



Lost update

Two threads read counter = 10, each computes 11, and both store 11. The intended result of two increments is 12. A suitable atomic increment or lock prevents this particular race.

Sources [10, 11, 13] / Reference directory

CHAPTER 18 / DEEP DIVE 1

Race conditions, atomics and mutual exclusion

A race condition occurs when a result depends on the relative timing of concurrent operations. A data race has a more specific meaning in language memory models, often involving conflicting unsynchronized accesses. Even a simple increment may involve multiple machine operations: load, add and store.

A mutex provides mutual exclusion for a critical section. Atomic operations provide indivisible behavior with specified ordering guarantees, but an atomic variable does not automatically make an entire multi-step algorithm correct. The synchronization design must protect invariants across every relevant access path. Memory ordering rules vary by language and architecture.

TECHNICAL REFERENCE

Step	Thread A	Thread B
1	Read 0	Read 0
2	Compute 1	Compute 1
3	Write 1	Write 1

Lost update

Two increments of a shared counter should produce 2. In the shown interleaving both threads compute from the old value and the result is 1. An appropriate atomic increment or mutex-protected operation prevents this particular lost update.

Sources [10, 11] / Reference directory

CHAPTER 18 / DEEP DIVE 2

Condition variables, deadlocks and progress

A condition variable lets a thread wait for a predicate on shared state while releasing its associated lock. The predicate must be rechecked after waking because another thread can change it before the waiter proceeds, and some interfaces allow spurious wakeups. A semaphore maintains permits for controlling access or counting events.

Deadlock occurs when participants cannot proceed because they wait for one another. A common case involves two locks acquired in inconsistent order. Starvation differs: a thread could theoretically proceed but continually loses access to resources. Livelock involves repeated changes without useful progress. Correct concurrent algorithms reason about both safety and progress.

TECHNICAL REFERENCE

Problem	Description
Deadlock	Circular waiting
Starvation	Repeated denial of progress
Livelock	Activity without useful progress

> **Lock ordering**
>
> Thread A needs locks X and Y; thread B needs the same locks. If both always acquire X before Y, this eliminates the simple two-lock circular-wait pattern. It does not prove every possible deadlock in a larger program is impossible.

Sources [10] / Reference directory

CHAPTER 18 / DEEP DIVE 3

Virtual machines, containers and isolation

A virtual machine presents a guest environment with virtual processors, memory and devices. A hypervisor manages the relationship between guest-visible resources and physical resources, often using processor virtualization support. Guest operating systems can run independently within that environment, although their performance depends on host scheduling and resource allocation.

Containers typically share the host kernel while isolating selected resource views and imposing limits. They package applications and dependencies but do not provide a separate guest kernel by default. Sandboxes further restrict operations using permissions and policy. These mechanisms have different trust boundaries and must be configured appropriately.

TECHNICAL REFERENCE

Technique	Kernel arrangement
Process isolation	Host kernel
Container	Shared host kernel
Virtual machine	Guest kernel on virtual hardware

> **Choose an isolation model**
>
> Running a service in a container can simplify deployment and limit visible resources. Running a different OS kernel generally requires a VM or other compatible execution environment. Neither is a substitute for software updates and correct access policy.

Sources [10, 11, 33] / Reference directory

CHAPTER 19 / OVERVIEW

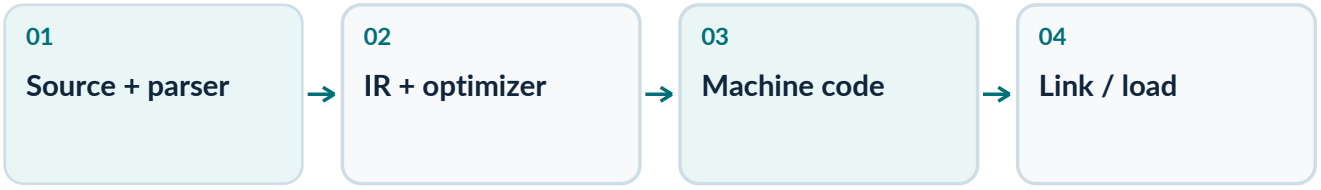
From source code to executable instructions

A compiler translates a program from a source representation into another form. In a typical native compiler, lexical analysis identifies tokens, parsing constructs syntax, and semantic analysis checks meaning and types. Intermediate representations make program transformations easier. Optimization can simplify computations, remove dead code or improve instruction selection while preserving the language's required behavior.

A backend selects target instructions, allocates registers and emits assembly or object code. An assembler encodes instructions and creates object-file sections and relocation information. A linker combines objects and libraries, resolves symbols and constructs an executable or shared library. At launch, an OS loader maps executable segments, sets up the execution environment and may involve a dynamic linker. A calling convention defines how functions exchange arguments, results and preserved registers.

Interpreters and virtual machines offer other execution models. CPython normally compiles source to bytecode that its interpreter executes; exact bytecode depends on the implementation and release. Just-in-time compilers generate machine code during execution, sometimes specializing code using runtime observations. Many environments combine interpretation, native code and runtime services. Ultimately, all software execution relies on physical hardware performing operations.

CONCEPT DIAGRAM



Inspect a tiny C function

For a file containing int add(int a,int b){return a+b;}, clang -O2 -S add.c produces target assembly. The exact output depends on target architecture, compiler version and options.

Sources [9, 16, 17, 18] / Reference directory

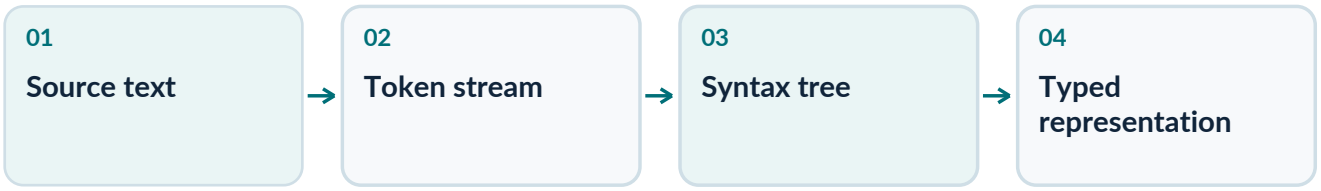
CHAPTER 19 / DEEP DIVE 1

Lexing, parsing and semantic analysis

A compiler front end reads source text and turns characters into tokens such as identifiers, keywords and operators. Parsing checks grammatical structure and builds a representation such as an abstract syntax tree. Semantic analysis resolves names, checks types and validates language-specific rules that grammar alone cannot express.

The same token sequence can be invalid because a variable is undeclared or an operation has incompatible types. Diagnostics should explain these errors at meaningful source locations. The front end establishes language behavior before later optimization or code generation decisions are made. Interpreters often use similar front-end techniques.

CONCEPT DIAGRAM



Parse precedence

The expression 2 + 3 * 4 normally parses as addition whose right operand is multiplication. Evaluating this tree yields 14, not 20. Parentheses can change the tree to make addition happen first.

Sources [16, 17, 39] / Reference directory

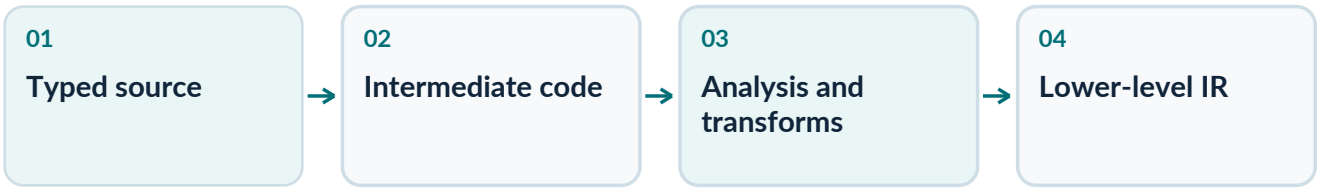
CHAPTER 19 / DEEP DIVE 2

Intermediate representation and optimization

An intermediate representation (IR) expresses operations in a form suited to analysis and transformation. A compiler may use multiple IRs to bridge language-level constructs and machine-specific operations. Control-flow graphs represent possible execution paths; data-flow information tracks where values originate and where they are used.

Optimization aims to improve execution while preserving behavior required by the language. Constant folding can replace a known calculation; dead-code elimination can remove operations that cannot affect observable behavior. Aliasing, exceptions, concurrency and numeric semantics can restrict transformations. A faster transformation that changes required results is not correct.

CONCEPT DIAGRAM



Constant folding

A compiler can replace 6 * 7 with 42 when language rules permit. It cannot generally remove an arbitrary function call merely because its return value is unused: the call may perform I/O, mutate state or raise an observable error.

Sources [16, 17, 39] / Reference directory

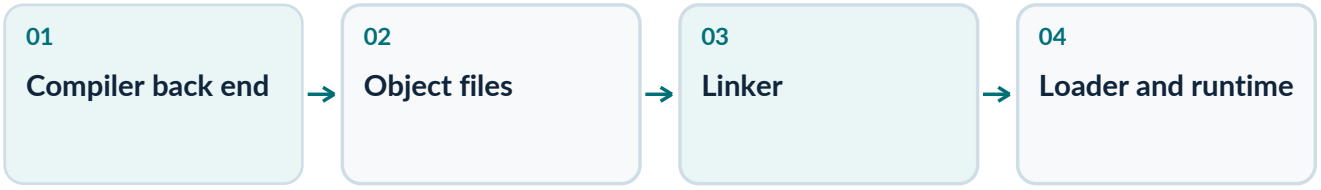
CHAPTER 19 / DEEP DIVE 3

Assembly, linking, loading and runtimes

A compiler back end selects target instructions, assigns registers and schedules operations. An assembler converts assembly into machine bytes and object-file metadata. A linker combines object files, resolves symbol references and arranges sections. Some references remain for a dynamic loader to resolve when the program starts or later.

The OS loader maps executable code and data into an address space and prepares initial execution state. A language runtime may initialize allocators, garbage collection, exceptions or other services before calling the user's main function. Bytecode interpreters and just-in-time compilers provide alternative execution strategies; in every case, physical CPUs ultimately execute machine instructions.

CONCEPT DIAGRAM



Unresolved symbol

One source file calls add(), which is defined in another object file. The linker can resolve the reference when both are included. If the definition is absent and no suitable library supplies it, linking generally reports an unresolved symbol.

Sources [9, 16, 17, 18, 36] / Reference directory

CHAPTER 20 / OVERVIEW

GPU rendering, displays and accelerators

A GPU is designed to deliver high throughput across many similar operations. Graphics workloads transform geometry, rasterize primitives into fragments and compute pixel contributions using shader programs and fixed-function units. A display engine ultimately reads image buffers and transmits pixel information to a display, where electronics drive visible output. Modern graphical desktops compose multiple window surfaces into a final image.

General-purpose GPU computing uses many execution lanes and a hierarchy of registers, shared or local storage and device memory. NVIDIA CUDA organizes work into threads and blocks; groups of threads execute using a SIMT model. Branch divergence, memory access patterns and resource occupancy affect performance. A GPU is not automatically faster than a CPU for sequential, small or transfer-heavy tasks.

Specialized AI accelerators may use matrix multiply units or other dataflow engines. A systolic array passes operands and partial results between arithmetic cells, reducing repeated transfers to distant memory. The CPU often orchestrates these accelerators, preparing data and launching work. Actual system performance depends on compute throughput, memory bandwidth, interconnects, numerical precision and how efficiently software maps the workload onto the hardware.

CONCEPT DIAGRAM

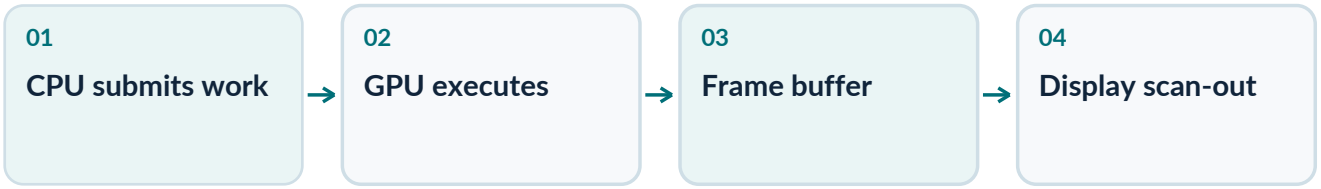


Image parallelism

A brightness adjustment can process millions of pixels independently, making it a good candidate for vector or GPU execution. A short dependency-heavy control task may be better handled on a CPU.

Sources [19, 20, 21] / Reference directory

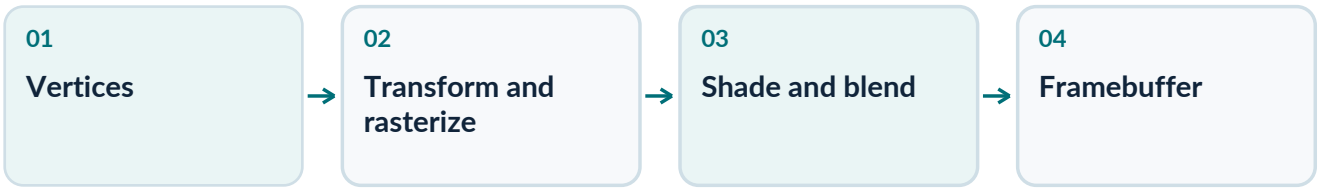
CHAPTER 20 / DEEP DIVE 1

Raster graphics, pixels and the rendering pipeline

A three-dimensional scene can be represented using vertices, geometry, materials, lights and a camera. A graphics pipeline transforms vertices from object coordinates toward screen coordinates, determines which sample locations are covered by primitives, and calculates colors. Depth testing and blending determine how visible fragments combine. Modern graphics APIs expose programmable stages as well as fixed-function operations.

The resulting image can be stored in a framebuffer, which is a memory resource containing pixel data. A compositor may combine several windows or surfaces before display. GPU rendering and display scanout are distinct processes: a display controller can read a completed buffer while rendering proceeds into another one.

CONCEPT DIAGRAM



Frame budget

A 60 Hz target gives approximately 16.67 ms per frame. A 120 Hz target gives about 8.33 ms. Rendering, synchronization, CPU work and display scheduling all contribute to whether a frame is delivered on time.

Sources [19, 27] / Reference directory

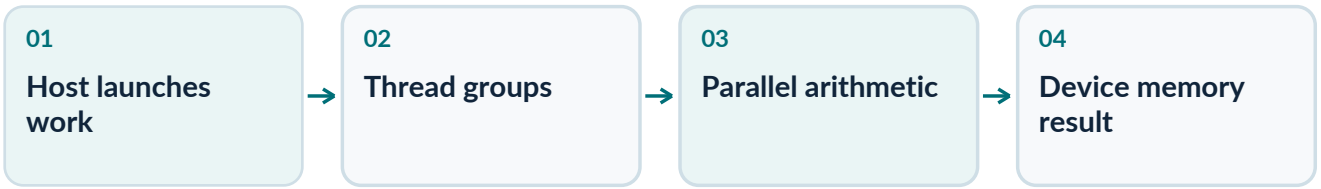
CHAPTER 20 / DEEP DIVE 2

GPU execution, memory and parallel work

GPUs use many execution resources to process large numbers of similar operations. In a SIMT programming model, many logical threads execute together in groups, with hardware coordinating their instructions. If threads in a group choose different branches, the hardware may need to execute different paths separately, reducing utilization.

A GPU memory hierarchy can include per-thread registers, group-accessible scratchpad memory, caches and device memory. Performance depends on arranging data accesses so useful bytes can be transferred efficiently. Synchronization within groups and between kernels has costs and rules. GPUs excel at highly parallel workloads, but a serial dependency chain can limit the advantage.

CONCEPT DIAGRAM



Vector addition

To compute C[i] = A[i] + B[i] for one million elements, assign each element to one logical thread or distribute several elements per thread. The additions are independent, but every element still requires data movement, so memory bandwidth can be the bottleneck.

Sources [19, 21] / Reference directory

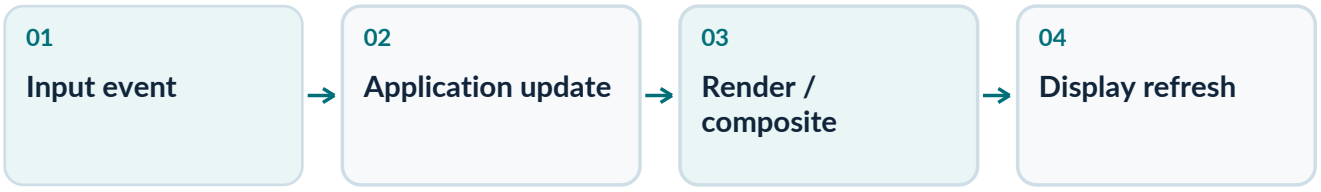
CHAPTER 20 / DEEP DIVE 3

Display scanout, input lag and media engines

A display refresh presents pixel data at a defined cadence or according to a variable refresh protocol. A display controller typically reads a framebuffer and sends a timed signal to the display. Pixel response time, refresh timing and buffering are separate factors in the overall experience.

A computer may also include dedicated video encode and decode units. These handle supported compression formats more efficiently than a general CPU or programmable GPU for many workloads. Support depends on codec, profile, resolution and hardware generation. The decoded frames still need memory and composition before presentation.

CONCEPT DIAGRAM



Framebuffer traffic

An uncompressed 3840 x 2160 buffer with four bytes per pixel contains 33,177,600 bytes. Reading that buffer 60 times per second represents roughly 1.99 GB/s of raw pixel data. Real display paths may use compression, multiple planes or different formats.

Sources [19, 27] / Reference directory

CHAPTER 21 / OVERVIEW

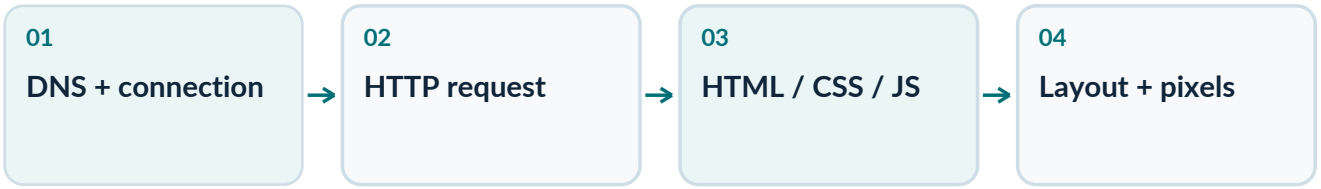
How the internet and a browser work

The internet connects networks through packet forwarding. A browser identifies a resource by its URL. It may resolve the domain through DNS, establish or reuse a connection, authenticate an HTTPS endpoint through TLS and issue an HTTP request. TCP supplies a reliable ordered byte stream; QUIC provides secure multiplexed transport over UDP and is used by HTTP/3. Actual behavior depends on cached results, protocol negotiation and network configuration.

Routers forward IP packets toward their destinations. Link-layer frames carry packets across individual links such as Ethernet or Wi-Fi. Servers process requests and send responses; content delivery networks and caches may serve resources nearer to the client. Different packets can experience different delays, and bandwidth and round-trip time are distinct performance properties.

After receiving HTML, the browser builds a document representation; CSS rules contribute styling, and JavaScript can change the document or request further resources. The rendering engine computes layout, paints and composites visual content. The browser may use several processes and threads for isolation and performance. GPU acceleration can help with some rendering stages. The resulting screen depends on an entire chain of software, network and hardware operations.

CONCEPT DIAGRAM



Latency versus bandwidth

A 1 MB response on a 10 Mbit/s link needs at least about 0.8 seconds of ideal payload transmission, excluding protocol overhead and latency. A fast link can still feel slow if many round trips are required.

Sources [22, 23, 24, 25, 26, 27] / Reference directory

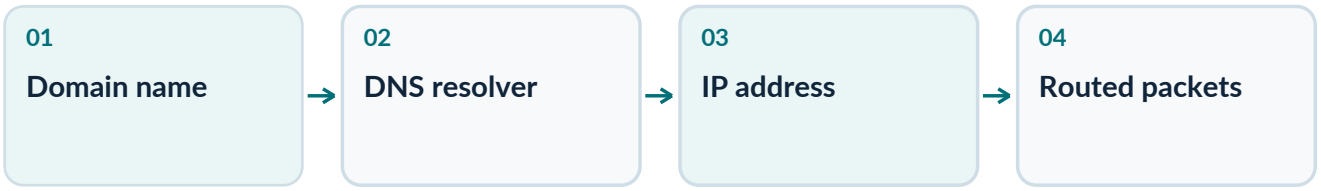
CHAPTER 21 / DEEP DIVE 1

Network addressing, routing and DNS

Networks send data across local links and interconnected routers. IP addresses identify network-layer endpoints or interfaces, and routing decisions select next hops toward destinations. Local link addresses serve a different scope. A packet can traverse several networks whose owners and physical media differ.

DNS maps domain names to records such as IP addresses and aliases. A resolver can use cached answers or query other DNS infrastructure. Records have time-to-live values that influence caching. A domain name can resolve to different addresses over time or by location; an address is not permanently tied to one human-readable name.

CONCEPT DIAGRAM



Cache expiration

A DNS record with a 300-second TTL can typically be reused within its remaining validity interval, subject to resolver policy. After that interval a resolver may need to refresh the answer. This is separate from a web browser's HTTP response cache.

Sources [22, 40, 41] / Reference directory

CHAPTER 21 / DEEP DIVE 2

TCP, UDP, QUIC and reliable delivery

TCP provides an ordered byte stream with mechanisms for acknowledging data, retransmitting losses and controlling sending behavior. Applications frame their own messages within that stream. UDP provides datagrams without TCP's built-in reliable ordered stream semantics; an application or higher-level protocol can add the behavior it needs.

QUIC operates over UDP and provides secure multiplexed streams with transport-level loss recovery and congestion control. Independent streams can reduce some cross-stream blocking compared with multiplexing over one TCP byte stream. Network loss, path latency, congestion and endpoint processing influence actual transfer speed regardless of protocol choice.

TECHNICAL REFERENCE

Transport	Application-facing abstraction
TCP	Ordered byte stream
UDP	Datagrams
QUIC	Secure multiplexed streams

Bandwidth-delay product

A 100 Mbit/s path with a 40 ms round-trip time has a bandwidth-delay product of 4 Mbit, or 500,000 bytes. A sender needs enough outstanding data to use the path fully under ideal conditions; congestion and receiver limits may reduce throughput.

Sources [23, 26, 42] / Reference directory

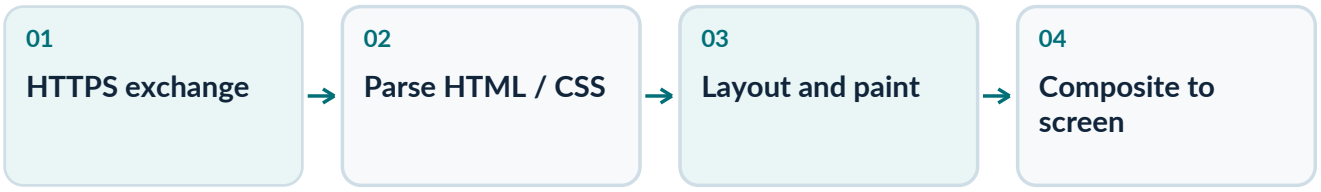
CHAPTER 21 / DEEP DIVE 3

HTTP, TLS and the browser rendering path

HTTP defines request and response semantics, including methods, status codes and metadata. HTTPS adds a protected transport using TLS, with certificate-based server authentication in typical browser connections. Encryption protects data in transit between TLS endpoints; it does not prove that a web application is trustworthy or bug-free.

After receiving resources, a browser parses HTML into a document tree, processes style rules and executes scripts according to their loading rules. Layout calculates geometry, painting describes visible content, and compositing combines layers for display. Caching and parallel resource loading can improve responsiveness, while slow scripts or expensive layout can delay interaction.

CONCEPT DIAGRAM



Cache versus validation

A cached response can be reused directly when allowed by its freshness policy. If validation is required, the browser can send a conditional request using a validator such as an ETag. A 304 response can avoid retransmitting an unchanged representation.

Sources [24, 25, 26, 27] / Reference directory

CHAPTER 22 / OVERVIEW

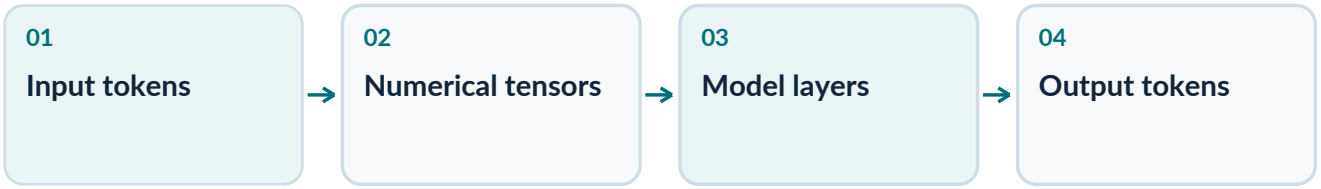
How artificial intelligence runs on computers

A neural network represents a parameterized mathematical function. Training adjusts its parameters using data and an optimization objective, commonly through gradients computed by backpropagation. Inference evaluates a trained model on new inputs. Matrix multiplications, reductions, nonlinear functions and memory operations dominate many architectures, although the exact workload varies.

A language model commonly converts text into tokens, maps tokens to numerical embeddings and applies repeated layers of attention and feed-forward transformations. During autoregressive generation, it predicts a distribution over possible next tokens, selects one according to a decoding procedure and repeats. The system retains intermediate state such as attention key/value caches to avoid repeating some computation. Training is different from generating a response; ordinary inference does not automatically update all model parameters.

GPUs, TPUs and NPUs can accelerate parts of this workload. Lower-precision representations can reduce bandwidth and computation cost, but may change accuracy or numerical behavior. Large models can be constrained by memory capacity, communication and bandwidth as much as by arithmetic. Machine learning does not change the physical foundations of computers: it is software and numerical computation mapped onto electronic hardware.

CONCEPT DIAGRAM



Matrix multiply

A 2 x 3 matrix multiplied by a 3 x 4 matrix produces a 2 x 4 result. Each output requires three products and their sum. Large neural networks perform vast numbers of related operations.

Sources [19, 20, 30, 31] / Reference directory

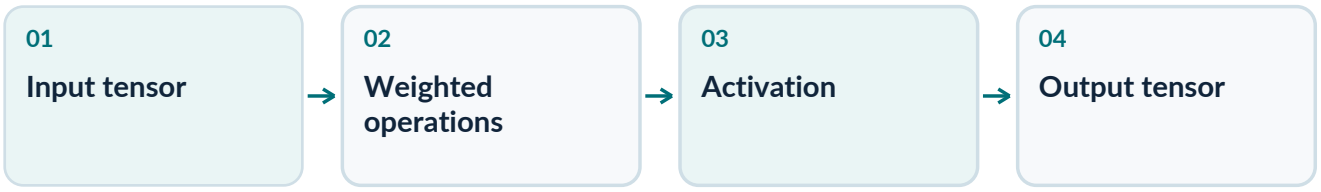
CHAPTER 22 / DEEP DIVE 1

Neural networks as numerical computation

A neural network is a parameterized mathematical function constructed from operations on numerical arrays. A dense layer commonly computes a weighted sum plus a bias, followed by an activation function. The input and learned weights determine outputs; arithmetic operations run on ordinary digital circuits or specialized matrix units.

Training adjusts parameters using a learning objective and an optimization procedure. In gradient-based training, backpropagation applies the chain rule to compute derivatives efficiently through the computational graph. Inference uses trained parameters to evaluate the network for new inputs, though some systems may perform additional adaptation or search.

CONCEPT DIAGRAM



Single artificial neuron

For input [2, 3], weights [0.5, -1] and bias 0.2, the weighted sum is 1 - 3 + 0.2 = -1.8. A ReLU activation maps this to 0. Different parameters and activation functions produce different outputs.

Sources [19, 20, 30] / Reference directory

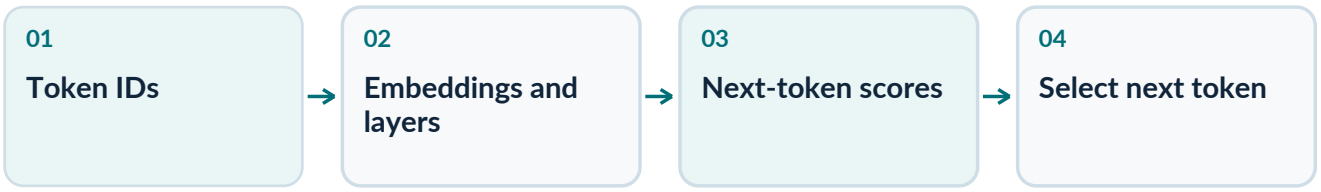
CHAPTER 22 / DEEP DIVE 2

Transformers, attention and token generation

A transformer processes sequences using learned representations and attention mechanisms. In scaled dot-product attention, projected query and key vectors determine scores that are normalized and used to combine value vectors. Multihead attention provides several learned projections, while feed-forward layers and normalization contribute additional transformations.

Many language models represent text using tokens and predict a distribution for the next token. Generation repeatedly evaluates this distribution and selects a token using a decoding rule. Cached key and value tensors can reduce repeated computation for prior tokens. The model's outputs are generated from learned statistical patterns and context; they are not guaranteed to be factually correct.

CONCEPT DIAGRAM



A tiny score vector

Suppose three candidate tokens have probabilities 0.6, 0.3 and 0.1. Greedy decoding selects the first; sampling can choose any according to its probability. This example omits the much larger vocabulary and detailed decoding rules of a real model.

Sources [30, 31] / Reference directory

CHAPTER 22 / DEEP DIVE 3

AI hardware, precision and memory constraints

Matrix multiplication is a major workload in many neural-network architectures. Specialized accelerators provide parallel multiply-accumulate resources and organize data reuse to reduce expensive memory traffic. GPUs and TPUs use different designs and programming ecosystems, but both benefit from parallelizable tensor operations.

Numeric formats trade precision, range, storage and arithmetic throughput. Reduced-precision formats or quantization can reduce model memory and improve speed, with potential accuracy effects depending on the model and method. Training also needs intermediate activations, gradients and optimizer state; inference may need caches and temporary buffers in addition to model weights.

TECHNICAL REFERENCE

Weight format	Ideal bytes for 1B weights
FP32	4 GB
FP16 / BF16	2 GB
8-bit	1 GB
4-bit packed	0.5 GB

Capacity estimate

The table counts only raw weight values using decimal GB. Real model files and runtime allocations can require extra storage for scales, metadata, activations, buffers and caches. A device with exactly the raw weight capacity may therefore be unable to run the model.

Sources [19, 20, 31] / Reference directory

CHAPTER 23 / OVERVIEW

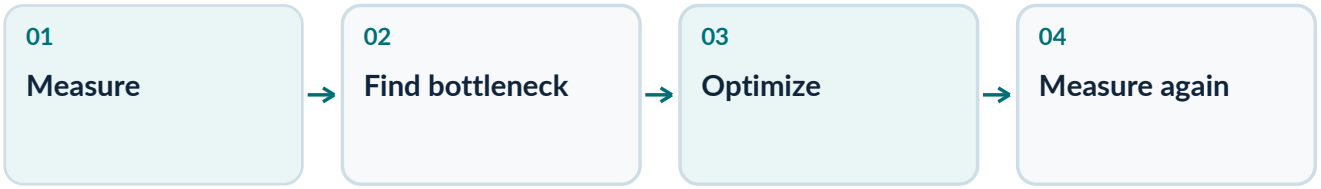
Measuring performance and energy

Performance depends on the workload and the measurement. Latency is the time for an individual operation; throughput is work completed per unit time. Clock frequency is cycles per second, not instructions per second. A simplified CPU time equation is instruction count x average cycles per instruction / clock frequency. Different architectures and compilers change instruction count and average cycles.

Memory-bound programs spend substantial time waiting for or transferring data. Compute-bound programs are primarily limited by execution capacity. Profilers and hardware counters can help identify where time is spent. Good benchmarking controls input size, compiler options, warm-up, system load and power mode, repeats measurements and reports uncertainty. Synthetic tests are useful but do not necessarily predict real applications.

Amdahl's law shows the limit of accelerating only part of a workload: speedup = 1 / ((1-p) + p/s), where p is the original time fraction improved and s is its speedup. Increasing clock frequency, core count or accelerator capability also changes power and heat. Cooling and power limits may constrain sustained performance. Energy per task can be more useful than peak speed for battery-powered or large-scale systems.

CONCEPT DIAGRAM



Amdahl's law

If 80% of a program becomes four times faster, overall speedup = 1 / (0.2 + 0.8/4) = 2.5 times, not four times. Optimizing the remaining part may now offer more benefit.

Sources [1, 10, 19] / Reference directory

CHAPTER 23 / DEEP DIVE 1

Latency, throughput, IPC and CPU-time equations

Latency measures time for one operation or request, while throughput measures completed work per unit time. A CPU performance model can express elapsed processor time as instruction count multiplied by average cycles per instruction and divided by clock frequency. IPC is the reciprocal of CPI over a matching measured interval.

These metrics have limited meaning without a workload and measurement boundary. A higher clock frequency can be offset by more cycles per instruction or longer memory stalls. A system with high throughput can still deliver poor individual request latency if queues grow. Benchmark reports should state configuration, units, input size and whether I/O time is included.

TECHNICAL REFERENCE

Metric	Definition
Latency	Time per request
Throughput	Requests per time
IPC	Instructions per cycle
CPI	Cycles per instruction

CPU time calculation

For 2 billion instructions, average CPI of 1.5 and a constant 3 GHz clock, modeled CPU time is 2e9 x 1.5 / 3e9 = 1 second. This ignores I/O wait and frequency changes; actual elapsed time may differ.

Sources [1, 34] / Reference directory

CHAPTER 23 / DEEP DIVE 2

Amdahl's law and parallel scaling

Parallel execution can accelerate the fraction of a workload that can run concurrently. Amdahl's law gives an ideal speedup of 1 / ((1 - p) + p / N), where p is the parallelizable fraction and N is the number of equally capable workers. The serial portion limits speedup even if the parallel portion becomes extremely fast.

Real scaling includes scheduling, synchronization, communication and shared-resource costs. Some workloads have more useful parallelism as their problem size increases; others saturate memory bandwidth before reaching all available cores. Measuring both fixed-size and growing-size workloads helps distinguish these cases.

TECHNICAL REFERENCE

Workers (p = 0.9)	Ideal speedup
1	1.00x
2	1.82x
4	3.08x
8	4.71x
Unlimited	10.00x

Find the limiting fraction

If a program spends 60% of its time in a component that can be made twice as fast, overall speedup is 1/(0.4 + 0.6/2) = about 1.43x. Optimizing the other components may be more valuable after that improvement.

Sources [1, 19] / Reference directory

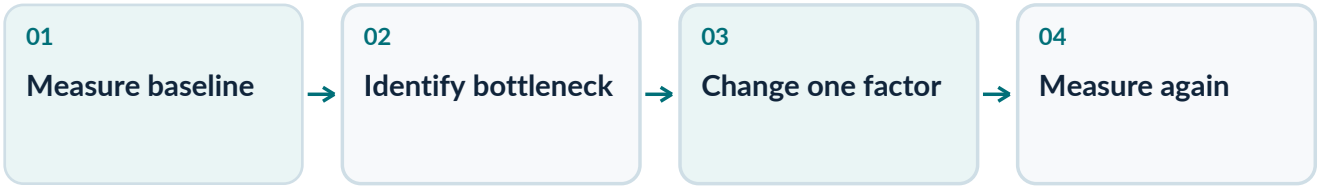
CHAPTER 23 / DEEP DIVE 3

Profiling, bottlenecks and thermal limits

Profiling observes how a program uses time and resources. Sampling profilers estimate where execution spends time; instrumentation records selected events and durations. Hardware performance counters may expose cache misses, branch events or other architecture-specific behavior. Each measurement tool introduces limitations and sometimes overhead.

A CPU-bound task is limited primarily by execution resources, while a memory-bound task spends substantial time waiting for or moving data. An I/O-bound task depends on external storage or network service. Thermal and power limits can change operating frequency during long workloads. Short benchmarks may not represent sustained performance.

CONCEPT DIAGRAM



A fair experiment

Compare two algorithms on identical input, hardware, compiler settings and thread counts. Run repeated trials, report a median and variability, and record whether caches were warm or cold. Do not infer universal speed from a single unusually fast trial.

Sources [21, 34] / Reference directory

CHAPTER 24 / OVERVIEW

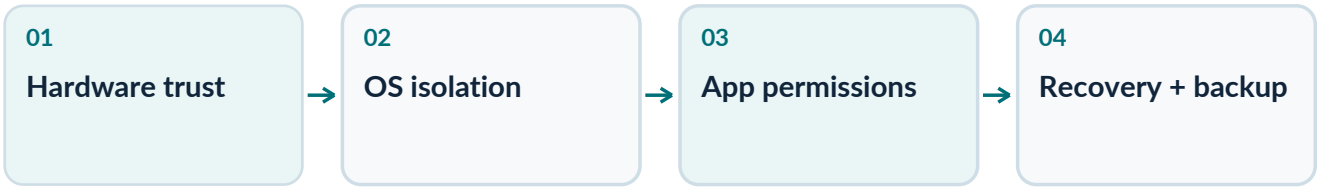
Computer security and reliability

Security protects confidentiality, integrity and availability. Hardware privilege levels and memory permissions separate application code from sensitive system resources. Operating-system access controls restrict files and devices. Cryptographic techniques protect data against unauthorized disclosure or modification, but their effectiveness depends on key handling, implementation and correct trust decisions.

A verified boot chain checks authorized code before execution; measured boot can record component measurements for later evaluation. Firmware resilience also requires detecting and recovering from corruption. Sandboxing reduces application privileges, and timely updates address known defects. Network encryption protects traffic between endpoints but cannot by itself make an endpoint trustworthy. Users and administrators still need appropriate authentication and backups.

Reliability is related but distinct. ECC memory can detect and, within its design limits, correct certain errors. Storage devices use error detection and correction, and redundant systems may tolerate component failures. Checksums detect many forms of accidental corruption but are not necessarily cryptographic authentication. Defense in depth combines controls across hardware, firmware, OS, applications and operations, since no single control covers every failure mode.

CONCEPT DIAGRAM



Least privilege

A photo viewer should not need administrator access merely to display a photo. Restricting privileges reduces the impact of a software defect or compromised application.

Sources [11, 15, 23, 25] / Reference directory

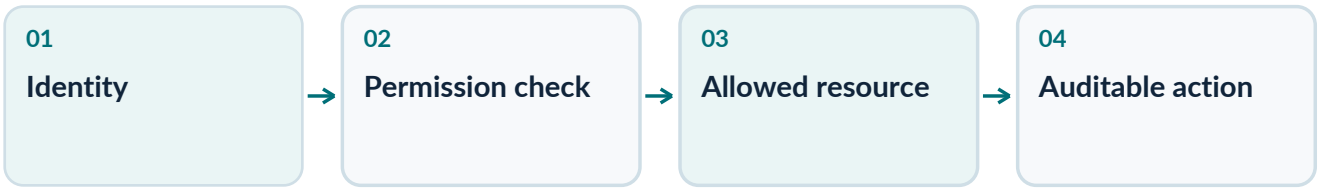
CHAPTER 24 / DEEP DIVE 1

Memory protection, permissions and least privilege

Operating systems use hardware protection features and software policy to restrict access to resources. Page permissions can separate executable code, writable data and inaccessible regions. Process isolation prevents ordinary programs from directly reading another process's memory unless an authorized interface permits it.

Least privilege means granting only the access necessary for a task. File permissions, user identities, service accounts and sandbox policies can reduce the impact of errors. Defense in depth combines independent protections because a single mechanism can fail. Updates, backups and careful configuration remain essential even when hardware isolation is present.

CONCEPT DIAGRAM



A read-only service

A service that only needs to read a public document directory should not automatically receive write access to system configuration or unrelated user files. Limiting those permissions reduces the possible consequences of a software fault.

Sources [10, 11, 15] / Reference directory

CHAPTER 24 / DEEP DIVE 2

Encryption, hashing and digital signatures

Encryption transforms data under a key so authorized parties can recover it. Authenticated encryption additionally protects integrity under its security assumptions. A cryptographic hash produces a fixed-length digest; it is not a reversible encryption function. Password storage uses specially designed password-hashing methods rather than a simple fast hash alone.

Digital signatures allow a verifier to check that a message corresponds to a signing key and has not been altered under the scheme's assumptions. Public-key certificates associate keys with identities through a trust framework. Security depends on key protection, randomness, correct protocol use and implementation, not just selecting a famous algorithm name.

TECHNICAL REFERENCE

Primitive	Purpose
Encryption	Confidentiality
Authenticated encryption	Confidentiality and integrity
Hash	Digest of data
Signature	Verify origin and integrity

Verify a download

A trusted publisher can distribute a signed release manifest containing file digests. Verification checks both the signature using a trusted public key and the downloaded file's digest. A digest obtained from an untrusted location alone cannot establish publisher identity.

Sources [25, 43] / Reference directory

CHAPTER 24 / DEEP DIVE 3

Reliability, ECC, backups and failure recovery

Digital systems can fail due to hardware defects, transient disturbances, firmware issues or software errors. Error-detecting codes add redundant information so certain corruptions can be detected. Error-correcting codes can also recover from defined error patterns, with capability depending on the code and system design.

Reliability requires multiple layers: hardware monitoring, filesystem consistency, service redundancy, backups and tested recovery procedures. RAID can improve availability against some disk failures but is not a backup because it may replicate accidental deletion or corruption. Backups must be independently recoverable and tested against realistic failures.

TECHNICAL REFERENCE

Mechanism	Protects against some
Parity	Bit corruption detection
ECC	Defined correctable errors
Redundancy	Component failure
Backup	Data loss and deletion

Simple parity

For data bits 1011, there are three ones. Add parity bit 1 to obtain even parity across the four data bits and parity bit. One flipped bit changes the parity check; two flipped bits can escape this simple check, so parity is not complete protection.

Sources [10, 15] / Reference directory

CHAPTER 25 / OVERVIEW

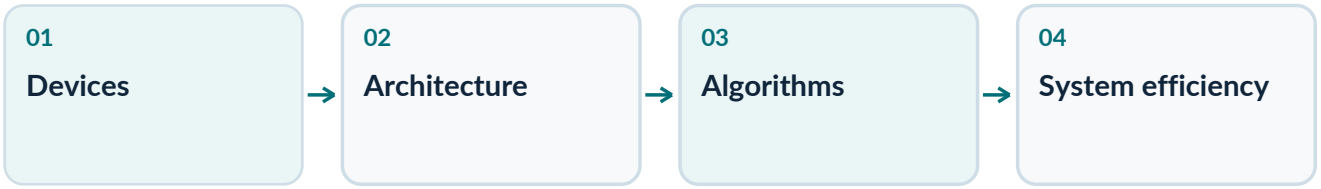
The future: chiplets, specialized systems and quantum

Future computing improvements are not solely about making transistors smaller. Engineers increasingly optimize communication, packaging and memory placement. Chiplets can combine dies optimized for different functions, while advanced packaging brings components closer together. High-bandwidth memory and near-memory processing address data-movement bottlenecks. Specialized accelerators trade some generality for efficiency on particular operations.

Classical computing uses bits and digital logic. Quantum computing instead manipulates quantum states using qubits and quantum gates, with measurement producing classical outcomes. Quantum algorithms may offer advantages for particular problem classes, but qubits are vulnerable to noise. Error correction encodes logical information across physical resources and introduces substantial overhead. Quantum hardware is not a general replacement for ordinary CPUs, memory or operating systems.

Research also explores new device materials, photonic communication, neuromorphic circuits and more energy-efficient software. These approaches differ in maturity and suitability. Claims about future capabilities should distinguish laboratory demonstrations, commercially available systems and theoretical results. The durable lesson is that computing progress is a whole-system problem, involving devices, circuits, architecture, software, algorithms and economics.

CONCEPT DIAGRAM



Specialization tradeoff

A matrix accelerator can be highly efficient for dense numerical operations yet cannot replace all the control logic, operating-system tasks and irregular workloads a general-purpose CPU handles.

Sources [5, 6, 20, 28] / Reference directory

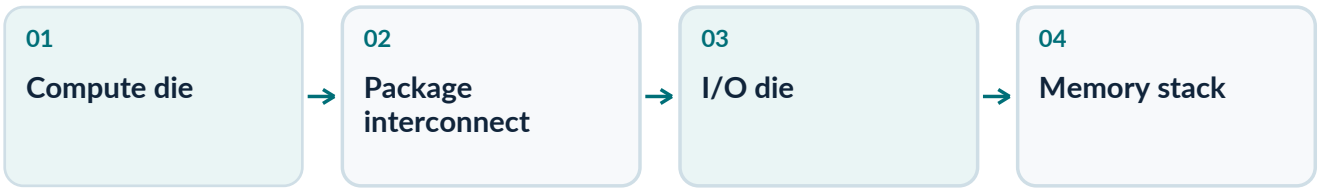
CHAPTER 25 / DEEP DIVE 1

Chiplets, stacked memory and heterogeneous integration

A chiplet architecture combines multiple dies in one package rather than placing every subsystem on a single large die. Different dies can be optimized for different manufacturing processes or functions. High-density package connections can provide substantial bandwidth, although communication between dies introduces physical constraints and design complexity.

Stacked memory places multiple memory dies in a closely connected structure. High-bandwidth memory interfaces use many parallel connections to support data-intensive workloads. Packaging must manage heat, power delivery, signal integrity and testability. The best partition of functions depends on yield, cost and performance, not integration alone.

CONCEPT DIAGRAM



A partitioning trade-off

An arithmetic subsystem may benefit from a dense leading-edge process, while some I/O circuits may be suitable for a different process. Combining these dies could reduce certain costs, but added package complexity and inter-die latency must be considered.

Sources [5, 6, 20] / Reference directory

CHAPTER 25 / DEEP DIVE 2

Embedded, edge and specialized computing

Embedded systems often interact directly with sensors and actuators and may require predictable response times. A real-time system is one whose correctness depends partly on when results arrive. A fast average response does not guarantee a deadline; designers consider worst-case behavior and resource contention.

Specialized accelerators implement selected operations using structures adapted to a workload. Examples include digital signal processors, video codecs and neural-network engines. Edge computing performs computation near the source of data, potentially reducing network latency or data transfer. Trade-offs include local power, storage, maintenance and hardware capability.

TECHNICAL REFERENCE

System	Typical design priority
Microcontroller	Timing and low power
DSP	Signal arithmetic
Video engine	Codec throughput
Edge accelerator	Local inference efficiency

Sensor-processing deadline

A sensor produces a new sample every 10 ms. A control system must process each relevant sample and deliver its response before the applicable deadline. An average computation time of 2 ms is insufficient evidence if occasional delays exceed the deadline.

Sources [1, 19, 20] / Reference directory

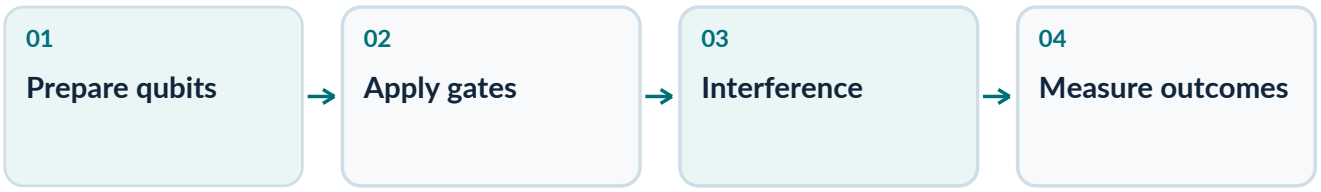
CHAPTER 25 / DEEP DIVE 3

Quantum computing: a different computational model

Quantum computation uses quantum states rather than ordinary deterministic bit states as its fundamental model. A qubit can be described by amplitudes associated with basis states, and multiple qubits can have entangled states. Quantum gates transform amplitudes; measurement yields classical outcomes with probabilities determined by the state and measurement basis.

Quantum algorithms use interference and other quantum properties to solve certain problems in ways that may offer advantages under specified assumptions. They do not make all computations faster. Noise, control, error correction and resource overhead are major engineering challenges. Classical computers are still required for control, measurement processing and ordinary computing tasks.

CONCEPT DIAGRAM



One ideal qubit

A qubit prepared in an equal superposition of the two computational basis states yields either 0 or 1 with probability one-half when measured in that basis. It does not reveal both answers in a single measurement. Repeated preparations can estimate outcome probabilities.

Sources [28, 44] / Reference directory

CHAPTER 26 / OVERVIEW

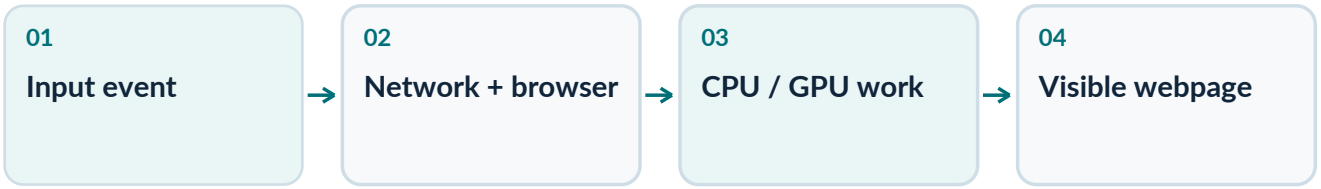
End-to-end: opening a website on a computer

A keyboard or touchscreen reports input to a controller, and device software turns it into an event. The browser's event loop processes that event and interprets the requested URL. The operating system provides sockets, memory mappings and scheduling; network resolution and connection setup may happen or be reused from previous activity.

A network interface moves data through device buffers. The kernel and browser process protocols and authenticate the connection when required. Received bytes enter memory, perhaps passing through several cache levels as CPU cores execute parsing, decoding and script instructions. The browser allocates objects and constructs internal representations of the document and styles. Some assets may be read from persistent caches rather than downloaded.

The rendering engine computes sizes and positions, creates drawing commands and combines layers. GPU work may be scheduled through a graphics driver. The display engine presents the completed image, and the user sees the webpage. Beneath this sequence, billions of microscopic switching events implement logical operations, state transitions and data transfers. Above it, applications and standards provide interfaces that make the entire process programmable.

CONCEPT DIAGRAM



The central insight

The computer never has to understand a webpage the way a human does. It follows precisely defined representations and operations, whose combined behavior produces the experience we recognize.

Sources [9, 11, 13, 22, 23, 24, 25, 26, 27] / Reference directory

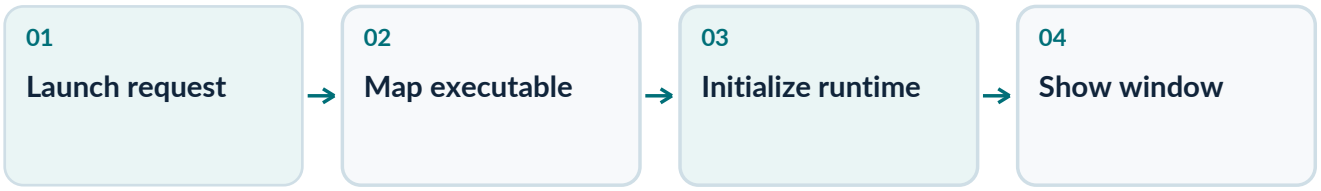
CHAPTER 26 / DEEP DIVE 1

Case study: launching an application

When a user selects an application, the desktop environment requests that the OS start the appropriate executable or invoke an existing process. For a new process, the kernel establishes an address space and initial thread. The executable loader maps code and data, and a dynamic loader may resolve required libraries.

The application's runtime initializes its environment and then enters its main logic. On-demand page faults may bring code or data into memory from storage. The scheduler gives the thread CPU time, while the application creates interface surfaces and registers event handlers. A compositor eventually makes its window visible.

CONCEPT DIAGRAM



Cold versus warm launch

A first launch after startup may need to fetch more code and data from storage. A later launch can benefit from file-system caches, even if the application itself has been closed. That difference illustrates why benchmark cache state matters.

Sources [10, 11, 16, 17] / Reference directory

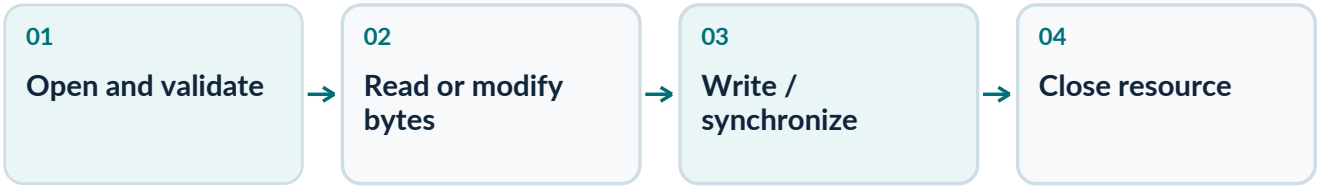
CHAPTER 26 / DEEP DIVE 2

Case study: reading and saving a file

To open a file, a program usually requests an OS-managed handle. The kernel resolves the path, checks permissions and accesses file-system metadata. Subsequent reads may be served from cached pages; misses lead to block-device requests and eventual completion notifications. The program interprets returned bytes according to the file format.

Saving reverses part of the data flow but also changes metadata and requires attention to consistency. An application may transform its in-memory state into a serialized representation, issue writes and request durability when necessary. Storage hardware maintains logical-to-physical mappings and completes requests according to its interface guarantees.

CONCEPT DIAGRAM



Why formats matter

A file containing the text 123 stores characters, whereas a binary file may store the number 123 in a fixed-width integer field. Both can be correct, but a reader needs the format specification to interpret the bytes correctly.

Sources [10, 11, 12, 29] / Reference directory

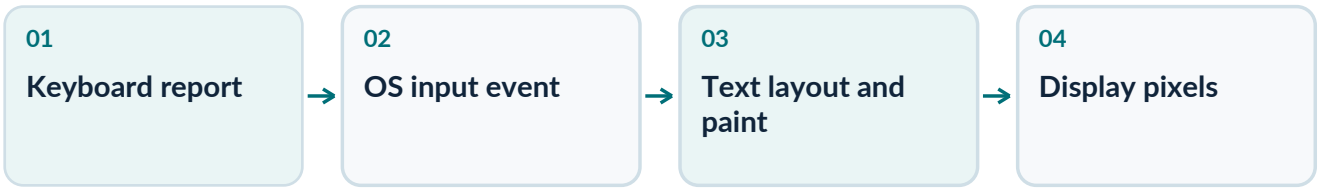
CHAPTER 26 / DEEP DIVE 3

Case study: typing a letter and seeing it appear

A keyboard controller detects changes in key state and sends input reports through its interface. The operating system's input subsystem translates reports into events and routes them according to focus and policy. A keyboard layout and input method determine how key events become text. Pressing a key is not necessarily equivalent to producing a single character.

The application updates its internal text state and requests a visual update. Text shaping selects and positions glyphs from fonts; layout determines placement; painting and compositing create or update a visible surface. The display controller presents the resulting pixels. The whole sequence crosses hardware, drivers, OS services and application code.

CONCEPT DIAGRAM



The letter A

With a typical English layout, pressing a letter key while Shift is active can produce uppercase A. A different layout or input method may interpret the same key differently. The physical key identifier, Unicode character and rendered glyph are separate representations.

Sources [10, 13, 27] / Reference directory

CHAPTER 27 / OVERVIEW

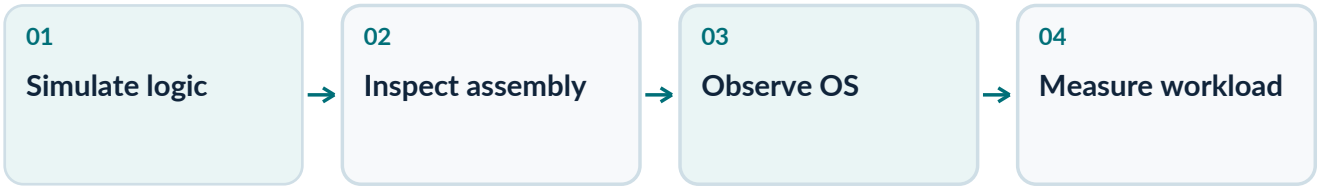
Practical study and verification exercises

Start at digital logic. Build a truth table for a half adder and a full adder using a logic simulator or a short program. Check all eight full-adder input combinations. Then implement a fixed-width binary adder in software and inspect both the sum and carry-out. Explain the difference between signed and unsigned overflow using four-bit operands.

Move to architecture. Use a RISC-V emulator or an educational simulator to step through ADD, load, store and branch instructions. Record PC and register changes. Compile a simple function at two optimization levels and compare assembly; do not assume every source variable becomes a physical register. For Python, use the standard dis module to inspect interpreter bytecode and contrast it with native machine instructions.

Finally explore systems. Use a task manager or system monitor to observe CPU, memory, storage and network activity while opening an application. Compare sequential and random array traversal using repeatable local benchmarks, keeping dataset size and runtime conditions explicit. Record actual observations and distinguish them from expectations. Use browser developer tools to study a page's network requests and rendering timing. These experiments link the explanations in this guide to measurable behavior without needing to open or modify physical hardware.

CONCEPT DIAGRAM



Self-test

Why is GHz not instructions per second? Why can a page fault occur without disk access? Why does a file write not always imply durability? Why can GPU acceleration lose to a CPU for a small task? Explain each using the relevant chapter.

Sources [1, 9, 10, 16, 18] / Reference directory

CHAPTER 27 / DEEP DIVE 1

Lab: build a simple truth table and adder

A truth table is a complete specification of a finite Boolean function. Start by listing every combination of two input bits A and B, then calculate XOR for the sum and AND for the carry. For three inputs A, B and Cin there are eight combinations. Check each against ordinary integer addition of those bits.

Extend the design into a four-bit addition by processing the least significant position first and passing the carry forward. Keep a record of each carry to verify the intermediate states. The test cases should include zero, a chain of carries and a result that exceeds the chosen width. This can be done entirely on paper or in software.

TECHNICAL REFERENCE

Test inputs	Expected 4-bit result	Carry
0000 + 0000	0000	0
0111 + 0001	1000	0
1111 + 0001	0000	1
1010 + 0101	1111	0

Check your implementation

If your four-bit adder returns 0000 with carry 1 for 1111 + 0001, it is correct for the unsigned arithmetic model. For signed arithmetic, record overflow separately. Carry and signed overflow are not the same flag.

Sources [1, 32] / Reference directory

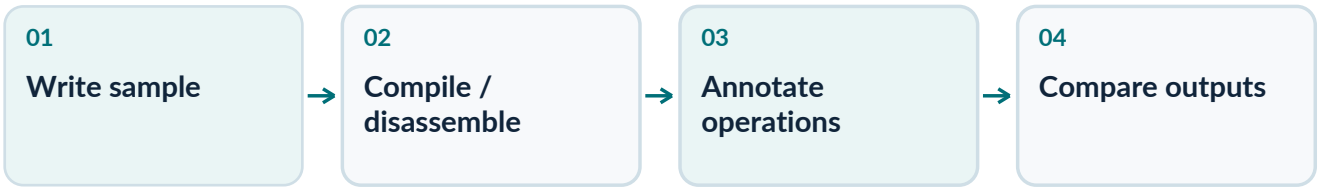
CHAPTER 27 / DEEP DIVE 2

Lab: inspect a program's instructions

Choose a small program containing a function that returns a sum of two integers. Use a compiler's documented assembly-output option, such as clang -S, to inspect the target assembly. Compare an unoptimized build with an optimized build. Make sure you record the compiler version, architecture and flags, since output is not universal.

Identify argument handling, arithmetic and return behavior. In a Python program, the documented dis module can show the bytecode of a simple function; this bytecode is not the same as native processor instructions. The runtime implements its execution semantics using machine code. Keep the experiment limited to your own sample code.

CONCEPT DIAGRAM



Explain the difference

If an optimizing compiler reduces a function to an add and return, while an unoptimized version stores values on the stack first, explain why both may produce the same result. Assembly reflects implementation strategy, not only source-level syntax.

Sources [9, 16, 17, 18] / Reference directory

CHAPTER 27 / DEEP DIVE 3

Lab: measure locality and resource use

Create a modest array of integers and measure the time needed to sum it in sequential order. Compare with a traversal that visits the same elements in a reproducibly shuffled order. Use a high-resolution monotonic timer, repeat measurements and confirm that both versions compute the same sum. Avoid allocating enough memory to disrupt your system.

The sequential traversal can benefit from spatial locality and predictable access, while randomized indexing may incur more cache misses. However, language overhead, data size, compiler optimization and cache capacity can obscure the effect. Use results to form a hypothesis rather than treating one run as proof of a universal hardware property.

TECHNICAL REFERENCE

Record	Example unit
Data size	Elements and bytes
Traversal	Sequential or shuffled
Time	Milliseconds
Correctness	Matching sum

> **Interpret cautiously**
>
> If sequential traversal takes 20 ms and shuffled traversal takes 60 ms, the observed ratio is 3x for that experiment. Repeat it at several safe array sizes. The result may change when the array fits entirely within cache or when the runtime overhead dominates.

Sources [10, 21, 34] / Reference directory

Glossary

ABI	Application binary interface; machine-level conventions for compiled software.
ALU	Arithmetic logic unit; executes integer arithmetic and Boolean operations.
API	Application programming interface; a software-level contract.
ASIC	Application-specific integrated circuit.
Bandwidth	Quantity of data transferred per unit time.
Bit / byte	A binary digit; a byte ordinarily consists of eight bits.
Cache	Storage that keeps copies of selected data for faster reuse.
Clock	A timing signal used to coordinate synchronous circuits.
CMOS	Complementary metal-oxide-semiconductor logic technology.
CPI / IPC	Cycles per instruction / instructions per cycle.
CPU	Central processing unit; executes architectural instructions.
DMA	Direct memory access; device-managed transfers under system control.
DRAM	Dynamic random-access memory; requires refresh.
ECC	Error-correcting code; detects or corrects specified corruption patterns.
Firmware	Software used for low-level device or platform operation.
GPU	Graphics processing unit; also used for parallel computation.
ISA	Instruction set architecture; the machine-code-visible contract.
JIT	Just-in-time compilation; generates executable code at runtime.
Kernel	Privileged core of an operating system.
Latency	Time required to complete an operation.

Glossary / continued

MMU	Memory management unit; handles address translation and protection.
MOSFET	Field-effect transistor controlled by an insulated gate.
NVMe	Interface specification for nonvolatile memory devices.
OS	Operating system; manages resources and provides services.
Page	Fixed-size unit in a particular virtual memory mapping scheme.
Pipeline	Overlapped processing of work through multiple stages.
RAM	Random-access memory; often used to refer to volatile main memory.
Register	Small architectural or internal processor storage location.
RTL	Register-transfer level description of a digital circuit.
SIMD / SIMT	Data-parallel instruction / thread execution models.
SoC	System on a chip; integrates multiple system functions.
SRAM	Static random-access memory; retains state while powered.
SSD	Solid-state drive; typically stores data in nonvolatile flash.
System call	Controlled request from software to the OS kernel.
TLB	Translation lookaside buffer; caches address translations.
TLS	Transport Layer Security; cryptographically protected communication.
UEFI	Unified Extensible Firmware Interface.
Virtual memory	Address-space abstraction with translation and protection.
Wafer	Semiconductor substrate on which multiple dies are fabricated.
Working set	Data or pages actively needed during a given interval.

Reference directory

Linked academic material, standards and engineering documentation. Numbered citations throughout the book correspond to the entries below.

[1] **MIT OpenCourseWare - Computation Structures (2017)**
https://ocw.mit.edu/courses/6-004-computation-structures-spring-2017/

[2] **Computer History Museum - Timeline of Computers**
https://www.computerhistory.org/timeline/computers/

[3] **Computer History Museum - Stored-program computer**
https://www.computerhistory.org/revolution/birth-of-the-computer/4/87

[4] **OpenStax - University Physics: Semiconductors and Doping**
https://openstax.org/books/university-physics-volume-3/pages/9-6-semiconductors-and-doping

[5] **ASML - How microchips are made**
https://www.asml.com/en/technology/all-about-microchips/how-microchips-are-made

[6] **ASML - Lithography principles**
https://www.asml.com/en/technology/lithography-principles

[7] **Unicode Consortium - UTF-8, UTF-16, UTF-32 and BOM**
https://www.unicode.org/faq/utf_bom.html

[8] **NIST Digital Library of Mathematical Functions - Arithmetics and Error Measures**
https://dlmf.nist.gov/3.1

[9] **RISC-V International - RV32I Base Integer Instruction Set**
https://docs.riscv.org/reference/isa/unpriv/rv32.html

[10] **Arpaci-Dusseau and Arpaci-Dusseau - Operating Systems: Three Easy Pieces**
https://pages.cs.wisc.edu/~remzi/OSTEP/

[11] **Linux Kernel Documentation - Memory Management**
https://docs.kernel.org/6.14/admin-guide/mm/index.html

Reference directory

Linked academic material, standards and engineering documentation. Numbered citations throughout the book correspond to the entries below.

[12] **NVM Express - NVMe Specification FAQ**
https://nvmexpress.org/faq_category/specifications/

[13] **Linux Kernel Documentation - Dynamic DMA mapping guide**
https://docs.kernel.org/core-api/dma-api-howto.html

[14] **UEFI Forum - UEFI Boot Manager**
https://uefi.org/specs/UEFI/2.10/03_Boot_Manager.html

[15] **NIST SP 800-193 - Platform Firmware Resiliency Guidelines**
https://www.nist.gov/publications/platform-firmware-resiliency-guidelines

[16] **Clang - Compiler command guide**
https://clang.llvm.org/docs/CommandGuide/clang.html

[17] **LLVM - Target-Independent Code Generator**
https://www.llvm.org/docs/CodeGenerator.html

[18] **Python Documentation - dis: Disassembler for Python bytecode**
https://docs.python.org/3/library/dis.html

[19] **NVIDIA - CUDA Programming Guide: Programming Model**
https://docs.nvidia.com/cuda/cuda-programming-guide/01-introduction/programming-model.html

[20] **Google Cloud - TPU architecture**
https://docs.cloud.google.com/tpu/docs/system-architecture-tpu-vm

[21] **NVIDIA - Nsight Systems Analysis Guide**
https://docs.nvidia.com/nsight-systems/AnalysisGuide/

[22] **IETF RFC 1034 - Domain Names: Concepts and Facilities**
https://www.rfc-editor.org/rfc/rfc1034.html

Reference directory

Linked academic material, standards and engineering documentation. Numbered citations throughout the book correspond to the entries below.

[23] **IETF RFC 9293 - Transmission Control Protocol**
https://www.rfc-editor.org/rfc/rfc9293.html

[24] **IETF RFC 9110 - HTTP Semantics**
https://www.rfc-editor.org/rfc/rfc9110.html

[25] **RFC 9846 - Transport Layer Security 1.3**
https://www.rfc-editor.org/info/rfc9846/

[26] **IETF RFC 9000 - QUIC Transport**
https://www.rfc-editor.org/rfc/rfc9000.html

[27] **MDN - How browsers work**
https://developer.mozilla.org/en-US/docs/Web/Performance/Guides/How_browsers_work

[28] **IBM Quantum - From error mitigation to fault-tolerant quantum computing**
https://www.ibm.com/quantum/blog/qec-continuum

[29] **Linux Kernel Documentation - ext4 journal**
https://docs.kernel.org/filesystems/ext4/journal.html

[30] **Google for Developers - Neural networks: training using backpropagation**
https://developers.google.com/machine-learning/crash-course/neural-networks/backpropagation

[31] **Vaswani et al. - Attention Is All You Need (2017)**
https://arxiv.org/abs/1706.03762

[32] **MIT OpenCourseWare - Computation Structures lecture notes (2009)**
https://ocw.mit.edu/courses/6-004-computation-structures-spring-2009/pages/lecture-notes/

[33] **RISC-V International - Supervisor-Level ISA**
https://docs.riscv.org/reference/isa/priv/supervisor.html

Reference directory

Linked academic material, standards and engineering documentation. Numbered citations throughout the book correspond to the entries below.

[34] **Intel - Intel 64 and IA-32 manuals and optimization references**
https://www.intel.com/content/www/us/en/developer/articles/technical/intel-sdm.html

[35] **Arm - The A64 ISA and compilers**
https://developer.arm.com/community/arm-community-blogs/b/architectures-and-processors-blog/posts/the-a64-isa-and-compilers

[36] **RISC-V International - RISC-V ELF psABI specification**
https://riscv-non-isa.github.io/riscv-elf-psabi-doc/

[37] **PCI-SIG - PCI Express architecture**
https://pcisig.com/what-pci-express-pcie-architecture

[38] **USB Implementers Forum - Document library**
https://www.usb.org/documents

[39] **LLVM - Language reference manual**
https://llvm.org/docs/LangRef.html

[40] **RFC 791 - Internet Protocol (IPv4)**
https://www.rfc-editor.org/info/rfc791/

[41] **RFC 8200 - Internet Protocol Version 6 (IPv6)**
https://www.rfc-editor.org/info/rfc8200/

[42] **RFC 768 - User Datagram Protocol**
https://www.rfc-editor.org/info/rfc768/

[43] **NIST FIPS 197 - Advanced Encryption Standard**
https://csrc.nist.gov/pubs/fips/197/final

[44] **IBM Quantum Learning - Quantum information**
https://quantum.cloud.ibm.com/learning/en/courses/basics-of-quantum-information/single-systems/quantum-information