N9LYA TECHNICAL GUIDES

Computer Science A to Z

For People Who Didn't Take the Course

Jerome M. Kutche
N9LYA

Second Edition · September 2026

About This Book

Most people who work with computers professionally never studied computer science.

They came up through the trades, through IT support, through a bootcamp, through automating something at work that grew into a career. They can write code that runs, configure systems that work, and solve problems that matter. And there's a persistent sense of a foundation missing underneath — a body of knowledge everyone else supposedly got in a lecture hall.

This book is that foundation, written for exactly that person.

What It's Actually For

Computer science is not programming. Programming is a skill; computer science is the body of theory about computation, information, and the machines that process it. You can be an excellent programmer with no computer science, and you'll hit a ceiling — problems you can't recognize, solutions you can't reach for, and inefficiencies you can't see because you don't know they're avoidable.

The value is not in memorizing algorithms. You will rarely implement a sorting algorithm. The value is in:

- **Recognizing the shape of a problem** — "this is a graph traversal" or "this is a caching problem" — because naming it gives you a hundred years of accumulated solutions.
- **Knowing what's expensive** — why your loop is slow, why that query brought the database down, why memory usage grows without bound.
- **Knowing what's impossible** — some problems have no efficient solution and some have no solution at all, and recognizing them saves you from a doomed project.
- **Reasoning about systems you can't see inside** — what the operating system is doing, what the cache is doing, what happens between your code and the silicon.

The Approach

Hardware up. Part I starts at binary and works up through logic gates to a working processor, because understanding what's actually happening makes everything above it concrete rather than magical. If you've read an electronics book, this will feel like coming at the same territory from the other direction.

Practical bias throughout. Where theory matters, it's here. Where theory is taught mainly because it's teachable, it's noted and abbreviated.

Systems get real coverage. Operating systems, concurrency, storage, and databases — Part V — because that's where working programmers actually get hurt, and it's the part self-taught practitioners most often skip.

Concurrency is treated seriously, including real-time and deterministic execution, which most CS material ignores entirely. Anyone who has worked with industrial controllers already understands scan cycles and deterministic timing; that intuition transfers directly and is worth naming.

Debugging gets its own treatment, because it's the actual job and it's taught almost nowhere.

Who This Is For

Self-taught programmers filling in the foundation. Skim Part II, read Parts III through V carefully.

Complete beginners. Part I assumes nothing. It's the longest on-ramp in the book and it's worth the time.

IT and operations people who want to understand what's happening underneath. Parts I and V are yours.

Technicians and controls people moving toward software. Your systems intuition is better than you think — Parts I and V will feel familiar in structure even where the vocabulary is new.

Students wanting the practical companion to a formal course.

How to Use It

Each chapter ends with a **Trace Check** — exercises to work through, verify, or implement. Do them. Reading about a hash table produces the illusion of understanding; implementing one badly and then fixing it produces the real thing.

Code examples use Python where a language is needed, because it reads closest to pseudocode. The concepts are language-independent, and where a language's specific behavior matters, I say so.

About the Author

Jerome M. Kutche holds the amateur radio callsign N9LYA, first licensed in 1991. He spent over three decades in industrial electrical and controls work — Allen-Bradley and Siemens PLC systems, motor control, plant electrical maintenance and supervision — holding NFPA 70E and thirty-nine other professional certifications. From 1990 to 1996 he built RF test and measurement equipment for the telecommunications and cable television industries.

He serves as Indiana's 44Net/AMPRNet coordinator and is trustee for two amateur radio clubs. He runs a segmented homelab with virtualized infrastructure, local AI inference, and rather more automation than is strictly necessary — most of it written, broken, and rewritten by hand.

He learned computer science the long way, in the order the problems demanded rather than the order a curriculum would prescribe. This book is the version he wishes he'd had.

Part I — Foundations

Chapter 1: Representation

Everything a computer does is manipulation of numbers. Understanding how everything else becomes numbers is the first foundation.

Binary

Computers use base 2 because the underlying hardware has two reliable states — a voltage present or absent, a transistor conducting or not. Anything with more states requires distinguishing voltage levels precisely, which is fragile.

Positional notation works the same in any base. In decimal, each position is a power of 10. In binary, a power of 2.

```
Binary  1 0 1 1 0 1 0 1
Weight 128 64 32 16 8 4 2 1
Value  128 + 32 + 16 + 4 + 1 = 181
```

Converting decimal to binary: repeatedly divide by 2, collecting remainders, then read them backward.

Bits and bytes. A bit is one binary digit. A byte is 8 bits — 256 possible values, 0 to 255. This grouping is a historical convention that became universal.

Word size is how many bits the processor handles natively — 32 or 64 on modern general-purpose machines, 8 or 16 on many microcontrollers. It determines the largest number handled in one operation and the amount of memory that can be addressed.

Hexadecimal

Base 16, using 0–9 and A–F. **It exists because one hex digit is exactly four bits**, so a byte is exactly two hex digits, making binary readable without being unwieldy.

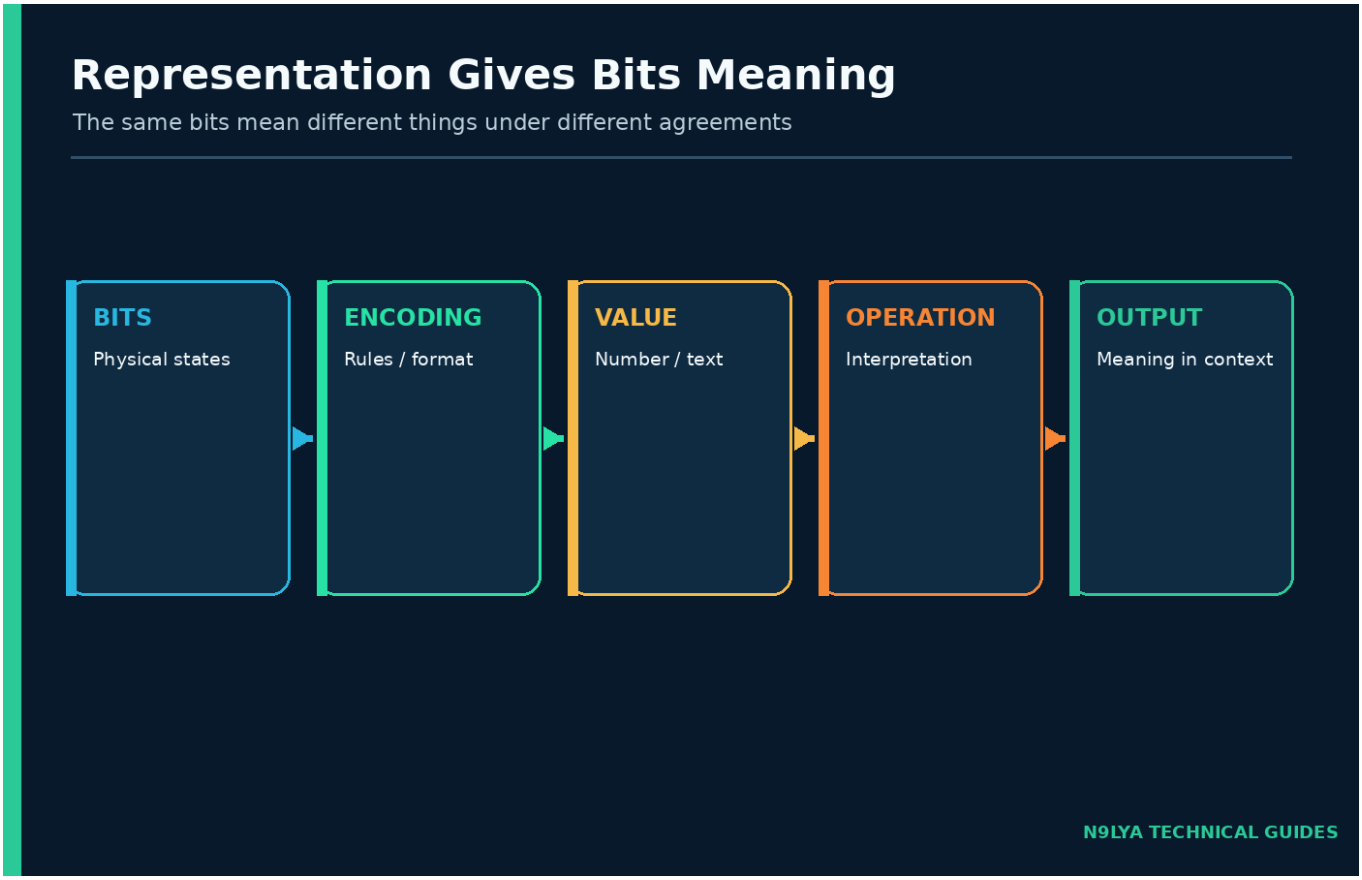

Figure 1: Representation gives bits meaning

```
Binary 1011 0101
Hex       B    5    →  0xB5  =  181
```

Where you'll meet it: memory addresses, color values, MAC addresses, byte dumps, error codes, and anywhere raw data is displayed. Learn to convert single hex digits to four bits on sight; it pays back constantly.

Octal (base 8, three bits per digit) survives mainly in Unix file permissions, where `755` means three groups of three permission bits.

Negative Numbers

Two's complement is how essentially all modern hardware represents signed integers, and it's worth understanding why rather than just accepting it.

To negate: invert every bit, then add 1.

```
 5 = 0000 0101
invert 1111 1010
add 1  1111 1011  = −5
```

Why this representation won: addition and subtraction use the same circuitry regardless of sign. Adding 5 and −5 in two's complement produces zero with a carry out that's discarded. No special-case hardware needed.

The asymmetry: an 8-bit signed value ranges −128 to +127. There's one more negative than positive, because zero occupies a slot on the positive side. **This causes real bugs** — negating the most negative value overflows back to itself.

Overflow happens when a result exceeds the representable range, and it wraps silently in most languages. Adding 1 to the maximum signed 8-bit value gives −128. This has caused genuine failures in production systems, including some spectacular ones.

Floating Point

Representing fractional values. The IEEE 754 standard splits a number into a sign bit, an exponent, and a mantissa — effectively scientific notation in binary.

The essential fact: most decimal fractions cannot be represented exactly in binary, exactly as 1/3 cannot be represented exactly in decimal.

```
0.1 + 0.2 == 0.3        # False
0.1 + 0.2               # 0.30000000000000004
```

This is not a bug. It's a consequence of finite binary representation, and it appears in every language using hardware floating point.

The practical rules:

Never compare floats for equality. Compare against a tolerance: `abs(a - b) < 1e-9`.

Never use floating point for money. Use integer cents, or a decimal type. Financial calculations accumulating floating-point error produce discrepancies that auditors notice.

Watch accumulation. Summing many small floats accumulates error. Summing in sorted order, or using compensated summation, reduces it.

Special values exist: positive and negative infinity, and NaN (not a number). **NaN is not equal to itself**, which is a legitimate and confusing property that breaks naive comparison and sorting code.

Precision: 32-bit floats give roughly 7 decimal digits, 64-bit doubles roughly 15–16. Use doubles unless memory or bandwidth genuinely constrains you.

Text

ASCII — 7 bits, 128 characters, covering English letters, digits, punctuation, and control codes. Adequate for English in 1963 and inadequate immediately afterward.

The mess that followed: dozens of incompatible 8-bit extensions, each covering a different set of languages, with no way to tell which was in use.

Unicode assigns a unique number — a code point — to every character in every writing system, plus emoji, plus a great deal more. Over a million possible code points.

UTF-8 is the dominant encoding. Variable width, 1 to 4 bytes per character, and it is **backward compatible with ASCII** — any ASCII file is a valid UTF-8 file. That compatibility is why it won.

UTF-16 uses 2 or 4 bytes and appears in Windows and Java internals.

What causes real bugs:

Character count versus byte count are different. A 10-character string may be 25 bytes. Truncating by bytes can split a character in half.

A code point is not a user-perceived character. An accented letter may be one code point or two (base plus combining mark), and emoji are frequently sequences of several. String length is genuinely ambiguous, and different languages answer it differently.

Encoding mismatches produce mojibake — the characteristic garbage of text decoded with the wrong encoding.

The rule: decode at the boundary, work in the language's string type, encode at the boundary. Know what encoding every input uses. Default to UTF-8 everywhere you have a choice.

Everything Else

Images — arrays of pixel values. Compression removes redundancy, either losslessly or by discarding detail.

Audio — amplitude sampled at intervals, quantized to integers.

Instructions — the processor's own operations, encoded as numbers. Chapter 4.

The unifying point: bits have no inherent meaning. The same 32 bits are an integer, a float, four characters, or a pixel, depending entirely on how they're interpreted. **Type information is a property of your code, not of the data**, which is why type errors produce nonsense rather than errors at the hardware level.

Trace Check

1. Convert 214 to binary and hex by hand. Verify.
2. Compute −37 in 8-bit two's complement.
3. In your language, evaluate `0.1 + 0.2 == 0.3` and explain the result.
4. Take a string with a non-ASCII character. Compare its character length and byte length.
5. Find the maximum value of a signed 32-bit integer and add one. Observe.

Chapter 2: Logic and Digital Foundations

How arithmetic emerges from switches. This is the bridge between electronics and computation.

Boolean Algebra

Two values, true and false, and a small set of operations.

Operation	Symbol	Result
NOT	¬, !	Inverts
AND	∧, &&	True if both
OR	∨,	
XOR	⊕, ^	True if different

Identities worth knowing:

```
A AND 1 = A          A OR 0 = A
A AND 0 = 0          A OR 1 = 1
A AND A = A          A OR A = A
A AND ¬A = 0         A OR ¬A = 1

NOT(A AND B) = NOT A OR NOT B      (De Morgan)
NOT(A OR B)  = NOT A AND NOT B     (De Morgan)
```

De Morgan's laws are the practically useful ones. They let you transform conditions into equivalent forms, which matters when simplifying tangled conditionals and when a language or hardware only gives you certain operations.

```
if not (a and b):        # equivalent to
if (not a) or (not b):
```

Gates

A logic gate implements a Boolean operation in hardware — transistors arranged so output voltage depends on input voltages.

NAND and NOR are functionally complete: any Boolean function can be built from either alone. This is why they dominate in silicon — one gate type, replicated.

Combinational logic produces output that depends only on current inputs. Adders, multiplexers, decoders, comparators.

Sequential logic has memory. Its output depends on inputs and stored state. Flip-flops, registers, counters.

Building an Adder

Worth walking through, because it shows arithmetic emerging from logic.

A half adder adds two bits, producing a sum and a carry:

```
Sum   = A XOR B
Carry = A AND B
```

A full adder also accepts a carry in, so adders can chain:

```
Sum      = A XOR B XOR Cin
Carry out = (A AND B) OR (Cin AND (A XOR B))
```

Chain 64 full adders and you have a 64-bit adder. That's genuinely how the arithmetic unit in your processor adds numbers, modulo optimizations for carry propagation speed.

And subtraction is free — negate the second operand in two's complement and add. That's why two's complement won.

Multiplication is repeated shifted addition. Division is harder and slower, which is why division is measurably more expensive than multiplication in real code.

Sequential Logic and State

A **flip-flop** stores one bit, updating on a clock edge. Group them and you have a **register**. Registers are the processor's fastest storage.

The clock synchronizes everything. Each tick, values propagate through combinational logic and get captured by flip-flops. **The clock period must exceed the longest propagation path**, which is what limits clock speed.

This is why clock speed alone doesn't determine performance — a processor doing more work per cycle at a lower clock can outperform a faster one doing less.

Bitwise Operations in Code

Boolean logic applied to whole values, bit by bit.

```
a & b       # AND
a | b       # OR
a ^ b       # XOR
~a          # NOT
a << n      # shift left  (multiply by 2^n)
a >> n      # shift right (divide by 2^n)
```

Practical uses:

Flags packed into one integer:

```
READ, WRITE, EXECUTE = 1, 2, 4
perms = READ | WRITE          # 3
has_write = perms & WRITE     # nonzero if set
perms &= ~WRITE               # clear it
```

Masking — extract specific bits: `(value >> 8) & 0xFF` gets the second byte.

Subnet math — network addressing is entirely bitwise. Masking an address against a subnet mask is a single AND operation.

XOR properties: `a ^ a == 0` and `a ^ 0 == a`. Used in checksums, parity, and simple encryption. XORing twice with the same key returns the original.

Shifts as arithmetic — compilers do this automatically now, and understanding it explains why powers of two appear everywhere in performance-sensitive code.

Note the shift caveat: right-shifting a negative number is implementation-dependent in some languages — arithmetic shift preserves the sign bit, logical shift doesn't.

Trace Check

1. Build a truth table for `(A OR B) AND NOT C`.
2. Apply De Morgan's law to simplify `not (x > 5 and y < 3)`.
3. Trace a full adder adding 1 + 1 + carry.
4. Implement a permission flag system using bitwise operations.
5. Verify that `a ^ b ^ b == a` for several values.

Chapter 3: How a Computer Works

The machine underneath. Understanding this makes performance behavior comprehensible rather than mysterious.

The Core Components

The CPU executes instructions.

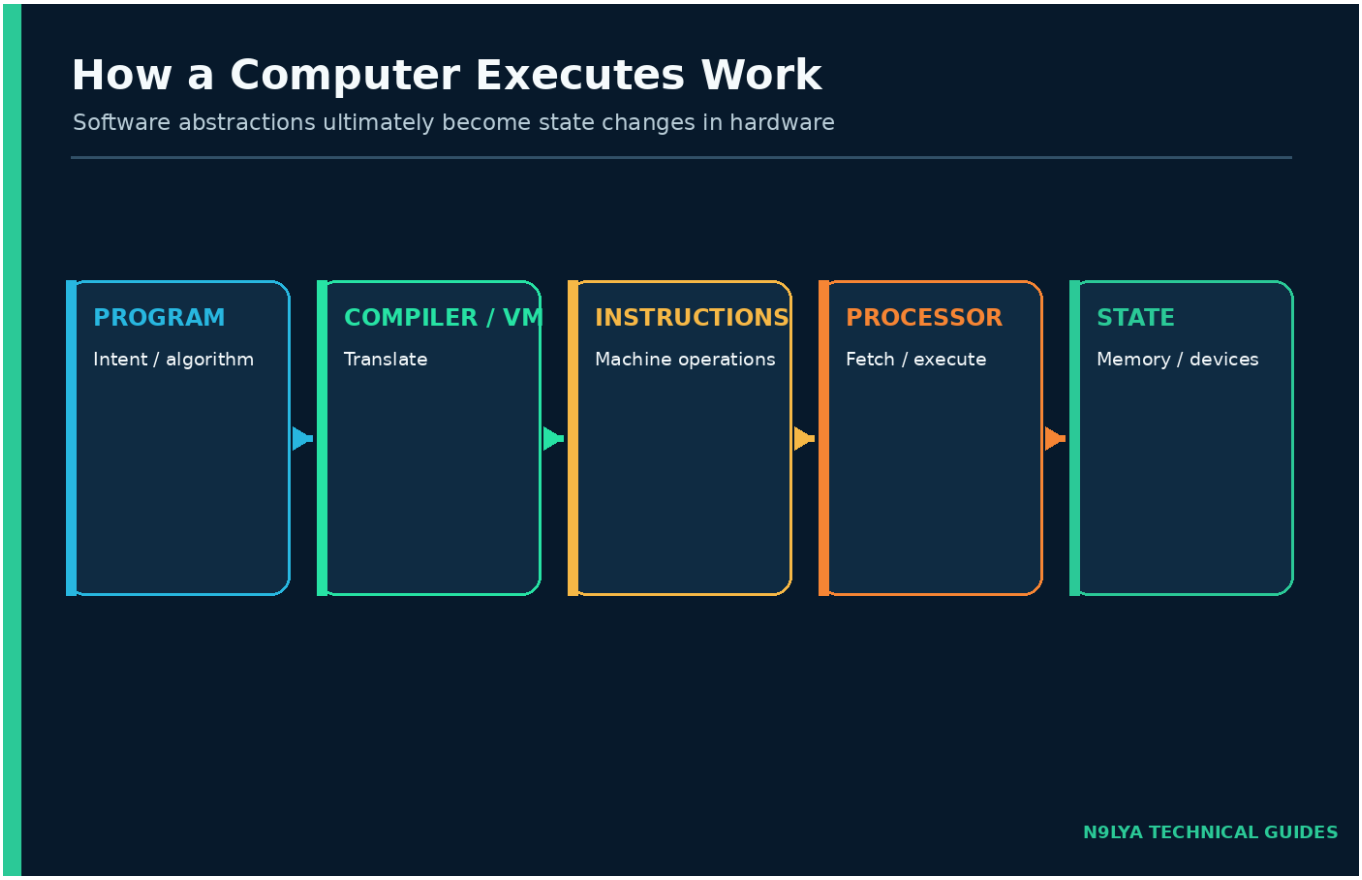

Figure 2: How a computer executes work

Memory (RAM) holds instructions and data currently in use. Volatile.

Storage holds data persistently. Slower by orders of magnitude.

I/O — everything else: network, display, keyboard, sensors.

Buses carry data between them.

Inside the CPU

Registers — a small number of very fast storage locations, directly addressable by instructions. Typically 16 to 32 general-purpose registers, each one word wide. **Everything the CPU computes on must be in a register**, which is why data movement between memory and registers dominates instruction counts.

ALU (Arithmetic Logic Unit) — performs arithmetic and logic. Chapter 2 built one.

Control unit — decodes instructions and orchestrates everything else.

Program counter — holds the address of the next instruction.

Status flags — zero, carry, overflow, negative. Set by operations and tested by conditional branches. **This is how `if` statements work at the machine level**: compare sets flags, branch tests them.

The Instruction Cycle

Fetch the instruction at the program counter. **Decode** it. **Execute** it. **Write back** the result. Increment the program counter. Repeat.

Billions of times per second. That's the whole machine.

Instruction Sets

The set of operations a processor understands — the boundary between hardware and software.

Typical categories: data movement (load, store, move), arithmetic, logic, comparison, branching, and function call/return.

x86-64 — the dominant desktop and server architecture. CISC heritage, large and irregular instruction set, enormous backward compatibility.

ARM — dominant in mobile and embedded, and increasingly in laptops and servers. RISC design, more regular, historically better performance per watt.

RISC-V — an open standard instruction set, growing quickly, particularly in embedded and academic contexts.

The CISC/RISC distinction has blurred. Modern x86 processors decode complex instructions into simpler internal operations, which is essentially a RISC core behind a CISC interface.

Assembly language is a human-readable form of machine instructions, one line per instruction. You will probably never write it and reading it is occasionally invaluable — when debugging optimized code, when the compiler did something surprising, or when working close to hardware.

Making It Fast

Modern processors do far more than the simple cycle suggests, and this is where performance intuition comes from.

Pipelining — overlapping stages of consecutive instructions, like an assembly line. While one instruction executes, the next decodes and a third fetches. Throughput approaches one instruction per cycle despite each taking several.

Branch prediction — pipelines break when the next instruction depends on a branch outcome not yet computed. The processor predicts, speculatively executes, and discards the work if wrong.

Prediction accuracy is typically above 95%, and a misprediction costs the whole pipeline — often 15–20 cycles.

This has a direct code consequence: predictable branches are far cheaper than unpredictable ones. A loop condition is predicted perfectly; a branch on random data is predicted at chance. Sorting data before processing it can make branchy code dramatically faster, which surprises people until they know why.

Superscalar execution — multiple execution units running several instructions per cycle in parallel.

Out-of-order execution — reordering instructions to avoid stalls, while preserving observable behavior.

SIMD — single instruction, multiple data. One instruction operating on a vector of values. This is how numeric and media code achieves its speed, and it's why array operations in numeric libraries vastly outperform equivalent loops.

Multiple cores — genuinely independent processors. Chapter 24.

The consequence of all this: the processor is not executing your code in the order you wrote it. It's preserving the appearance of doing so, single-threaded. Once multiple threads are involved, that appearance breaks, which is the root of a whole class of concurrency bugs.

Interrupts

Hardware needs attention asynchronously. An **interrupt** suspends execution, saves state, runs a handler, and resumes.

This is how I/O works without polling, how preemptive multitasking works (a timer interrupt returns control to the operating system), and how real-time systems respond to events.

Interrupt handlers must be fast and careful. They run with normal execution suspended, and they cannot safely use anything that isn't reentrant. Anyone who has written a PLC interrupt routine or a microcontroller ISR already knows this discipline; it's the same discipline.

Trace Check

1. Read the disassembly of a small function you wrote. Identify the loads, stores, and branches.
2. Time a loop over sorted versus randomly ordered data with a data-dependent branch. Explain the difference.
3. Find your processor's core count, cache sizes, and clock speed.
4. Identify which instruction set architecture your machine uses.

Chapter 4: Memory

The single most important thing to understand for performance, and the least visible.

The Hierarchy

Speed and capacity trade against each other, so systems use layers.

Level	Typical Size	Latency (approx.)
Registers	Hundreds of bytes	1 cycle
L1 cache	32–128 KB	~4 cycles
L2 cache	256 KB–2 MB	~12 cycles
L3 cache	8–64 MB	~40 cycles
RAM	8–512 GB	~200+ cycles
SSD	0.5–8 TB	~100,000 cycles
Hard disk	Terabytes	~10,000,000 cycles

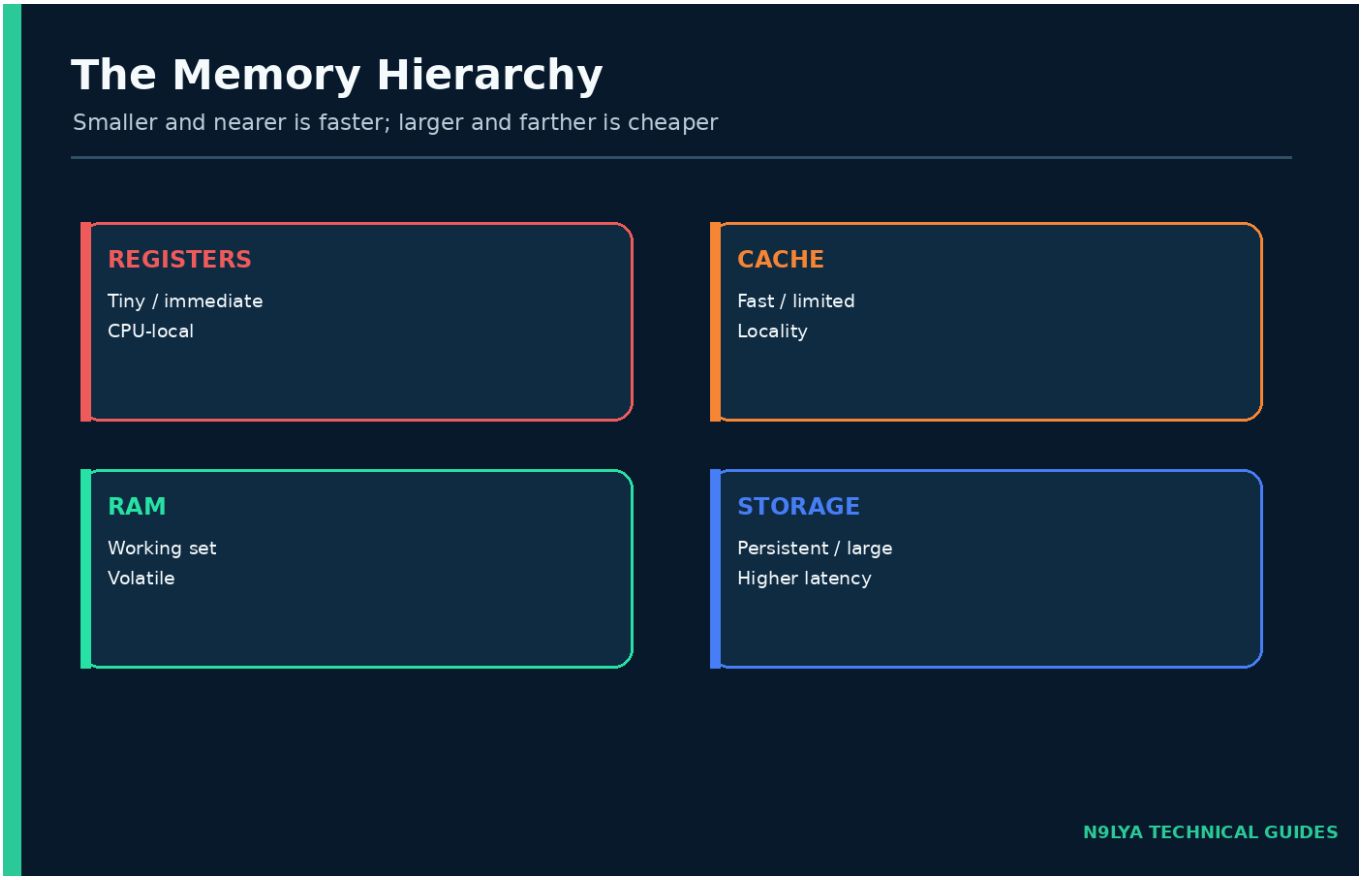

Figure 3: The memory hierarchy

End of the free sample

Computer Science A to Z — Second Edition · by Jerome M. Kutche

That was the first 12 of 69 pages. The full book continues with the practical detail, worked examples and recovery steps that don't fit in a sample.

Get the full book ($8.99, PDF + EPUB):

https://booksrus.biz/product/computer-science-a-to-z/

Also on Gumroad:
https://n9lya4.gumroad.com/l/xtptmk

All 17 Second Edition guides in one bundle for $89.99:
booksrus.biz/product/n9lya-technical-guides-complete-bundle

This sample may be shared freely in unmodified form. © Jerome M. Kutche. Second Edition.