Machine Learning Compilers

Bridging Models and Hardware
for Maximum Performance

A first-principles course: computers, compilers, GPUs,
and the systems that make AI fast

by Laxmen Murali

First edition

Contents

Chapter 1 — What Is a Computer? From Switches to Thinking Machines

> *"What I cannot create, I do not understand."*
>
> Richard Feynman, written on his blackboard at the time of his death

---

1.1 Learning Objectives

By the end of this chapter, you will be able to:

- Explain what a computer fundamentally is, and what it is not, in plain language.
- Describe how a simple on/off switch (a transistor) becomes a logic gate, how logic gates become an adder, and how adders become a machine that computes.
- Explain the **stored program concept**: the single most important idea in all of computing.
- Draw and explain the **von Neumann architecture** and the **fetch–decode–execute cycle**.
- Trace, by hand, the execution of a real program on a simple machine, one step at a time.
- Explain the **ladder of abstraction**, the tower of layers from physics up to AI models, and point to exactly where ML compilers live in that tower.
- Build a complete, working computer simulator (**TOY-100**) in about 40 lines of Python, a machine we will keep extending throughout this book.
- Answer the classic entry-level systems interview questions about how computers work.

This chapter is the foundation stone. Every single topic in this book, GPUs, LLVM, tensor IRs, FlashAttention, all of it, is built from the ideas on these pages. Take your time here.

1.2 A Story: The Room Full of Computers

We'll begin in 1944, in a secret laboratory on a mesa in New Mexico called Los Alamos. The people there were designing the atomic bomb, and they had a problem that had

nothing to do with physics and everything to do with arithmetic.

To predict what would happen inside an imploding bomb core, they needed to solve enormous systems of equations. Not solve them cleverly, solve them *numerically*: millions of small multiplications and additions, each one feeding the next. No human being could do this alone in a lifetime.

So they filled a room with people and mechanical calculators, clattering desktop machines called Marchants that could multiply two numbers if you cranked them. And here is the beautiful part, the part that matters for this entire book: the person who figured out how to make that room *fast* was a young physicist named Richard Feynman.

Feynman's insight was that the problem wasn't the speed of any one person. It was the *organization* of the work. He and his colleagues broke each big calculation into tiny steps and arranged the people like an assembly line: this person only multiplies, that person only adds, this one only squares a number and passes the card to the next desk. A single problem flowed through the room like a car through a factory.

Then they got greedy. Why should the room work on only one problem at a time? They ran several calculations *simultaneously*, using different colored cards for each problem, flowing through the same room, through the same people, interleaved. When someone discovered an error in one calculation, they didn't restart everything, they figured out exactly which cards were contaminated, recomputed just those, and spliced the corrections back into the stream.

Stop and look at what this room invented, decades before the vocabulary existed:

- Breaking a computation into tiny, dumb, repeatable steps → **instructions**
- One specialist per step, work flowing between them → **pipelining**
- Multiple problems in flight at once, sharing the same workers → **parallelism and multi-tenancy**
- Recomputing only what an error touched → **incremental computation and dependence analysis**
- Deciding who does what, when, to keep everyone busy → **scheduling**

Here's the punchline. And I want you to hold onto it for the next thousand pages: **that room *was* a computer.** The people were the processing units. The cards were the memory. Feynman's arrangement of the work was the *program*, and the act of arranging it well was *compilation*.

Everything in this book, every GPU kernel, every compiler pass, every scheduling algorithm, is a descendant of that room. The machines got faster by a factor of a trillion. The ideas barely changed.

One more historical note, because it reframes everything: the word "computer" originally meant *a person*. From the 1600s until roughly the 1950s, a "computer" was a job title, like "baker" or "carpenter", someone employed to do calculations. The women who calculated rocket trajectories at NASA (you may know Katherine Johnson's story from the film *Hidden Figures*) were called computers. The machine was named after the human, not the other way around. A computer is not a magical thinking box. It is a worker that follows instructions. It just happens to be made of silicon instead of flesh, and to work about a trillion times faster.

1.3 What Is a Computer, Really?

Explain It Like I'm Five

A computer is a machine that follows a recipe.

That's it. That's the whole thing.

You give it a list of very simple steps, "take this number, add it to that number, write down the answer, if the answer is zero skip to step 7", and it follows them. One at a time. In order. Without getting bored, without getting distracted, without asking why.

The steps it can follow are *incredibly* dumb. A computer cannot "recognize a cat." It cannot "understand a sentence." It can only do things like: copy a number from here to there, add two numbers, compare two numbers, and jump to a different step in the recipe. That's roughly the whole menu.

So how does a machine that can only add and copy numbers end up recognizing cats and writing poetry? Because it follows *billions* of those dumb steps *per second*, and because very clever people have figured out how to write recipes in which billions of dumb steps add up to something that looks like intelligence.

Hold onto this: **a computer is not smart. It is fast.** All the smartness lives in the recipes. And this book is about the programs that write the recipes for the fastest machines on Earth.

The Core Intuition

Every computer, from the chip in a musical greeting card to the GPU cluster training a frontier AI model, is the same three things:

1. **A place to keep numbers** (memory)
2. **A thing that changes numbers** (a processor)
3. **A recipe saying which changes to make in which order** (a program)

Memorize that triad. When we meet a GPU in Part VI, it will be: a *huge* place to keep numbers, *thousands* of things that change numbers, and recipes called kernels. When we build an ML compiler in Part XI, its job will be: given a recipe written for mathematicians, rewrite it into the best possible recipe for the machine. Different costumes, same skeleton.

The Analogy We'll Reuse All Book: The Kitchen

Picture a professional kitchen.

- The **pantry** is memory: shelves of ingredients (numbers), each shelf with a label (an address).
- The **chef** is the processor: they can chop, mix, and heat (add, multiply, compare), but only what's on the counter in front of them.
- The **counter** is the registers: a tiny workspace holding just the few ingredients being worked on *right now*.
- The **recipe** is the program: numbered steps, followed in order.
- The **finished dish** is the output.

Notice something important that will dominate this book from Chapter 5 onward: the chef is fast, but *walking to the pantry is slow*. A great kitchen isn't one with a faster chef, it's one organized so the chef almost never has to walk to the pantry. Remember this sentence. In modern computing, **arithmetic is nearly free; moving data is expensive.** A huge fraction of ML compiler engineering is pantry logistics.

1.4 The Humble Switch: Where It All Begins

Every concept in this book, we'll build from the ground up. So let's go all the way to the ground. What is a computer *made of*?

ELI5

A computer is made of tiny light switches. Billions of them. Each switch is either ON or OFF. That's all a computer physically is: an unimaginably large box of switches flipping each other on and off very, very fast.

The Real-World Object: The Transistor

The switch is called a **transistor**, invented at Bell Labs in 1947 by John Bardeen, Walter Brattain, and William Shockley (they got the Nobel Prize for it). Before the transistor, computers used vacuum tubes, glowing glass bulbs the size of your thumb that burned out constantly. ENIAC, completed in 1945, used about 17,468 of them, filled a large room, and could do around 5,000 additions per second.

A transistor does the same job as a tube, acts as a switch, but it's a solid speck of silicon with no moving parts and nothing to burn out. And crucially, it can be made *small*. Absurdly small. Modern transistors are a few nanometers across; tens of thousands of them would fit across the width of a human hair. A modern GPU packs on the order of a hundred billion of them onto a chip the size of your fingernail-to-palm.

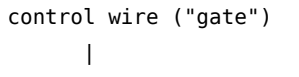
What makes a transistor magical isn't that it's a switch. Your wall has switches. It's *who flips it*:

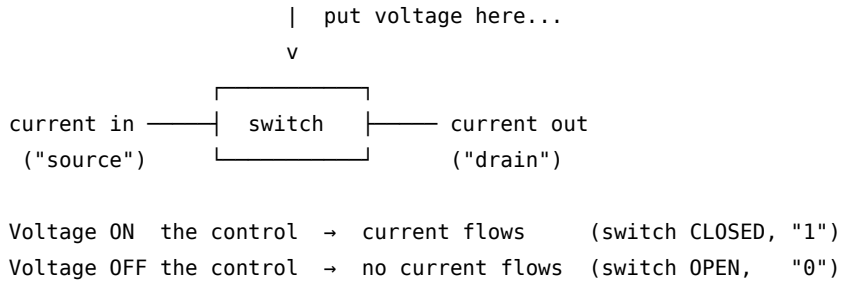
> **A transistor is a switch that is flipped by electricity, which means switches can flip other switches.**

Read that again. A wall switch needs a human finger. A transistor needs only a voltage on its control wire. So the output of one switch can be the control of another switch. Switches controlling switches controlling switches, that's a chain reaction of logic, and it needs no human in the loop. This one property is why computers can exist.

Visually

A transistor has three terminals. Think of it as a gate in a garden fence for electric current:





We give the two states names: a wire carrying voltage is a **1**, a wire without voltage is a **0**. That's the entire origin of binary, which gets its own full treatment in Chapter 2. For now: ON = 1, OFF = 0.

What Problem Does It Solve?

The transistor solves the problem of **automation of logic**. Before it (and its ancestor the relay/tube), any decision, "if this, then that", required a human or a slow mechanical contraption. The transistor made decisions: electrically fast (a modern transistor can switch billions of times per second), microscopically small, essentially free to operate, and, critically, *composable*: outputs can drive inputs.

Composability is the deep idea. One switch is a toy. Switches that control switches are a *language*.

1.5 From Switches to Logic Gates

ELI5

If you wire a few switches together in clever patterns, you get little machines that make tiny decisions. These are called **logic gates**. A gate looks at one or two input wires (each carrying a 1 or a 0) and produces one output wire (a 1 or a 0), according to a fixed rule.

There are three rules to start with, and they have friendly names:

- **AND**: "I output 1 only if *both* my inputs are 1."
- **OR**: "I output 1 if *at least one* of my inputs is 1."
- **NOT**: "I output the opposite of my input."

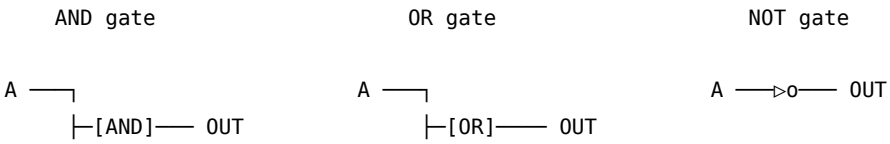
Analogy

Think of a nightclub with two bouncers:

- An **AND** gate is a door with two bouncers where *both* must nod to let you in. Strict.
- An **OR** gate is a door where *either* bouncer nodding gets you in. Lenient.
- A **NOT** gate is a contrarian who says the opposite of whatever they're told. Told "yes," says "no."

Visually

We draw gates like this, and we describe their complete behavior with a **truth table**, a table listing every possible input combination and the resulting output. Truth tables matter because they are *complete*: two circuits with the same truth table are interchangeable, no matter how differently they're built. (File that thought away, "same behavior, different implementation" is the legal foundation of every compiler optimization you will ever write.)

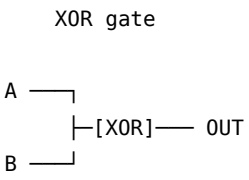


A	B	OUT
0	0	0
0	1	0
1	0	0
1	1	1

A	B	OUT
0	0	0
0	1	1
1	0	1
1	1	1

A	OUT
0	1
1	0

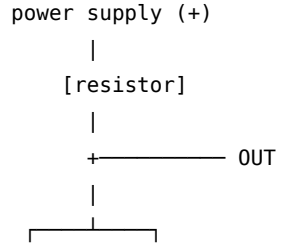
One more gate earns a seat at the table, because arithmetic will need it. **XOR** ("exclusive or") outputs 1 if its inputs are *different*:

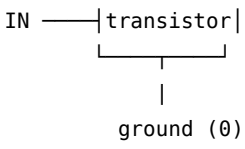


A	B	OUT	
0	0	0	"1 if A and B disagree"
0	1	1	
1	0	1	
1	1	0	

How Are Gates Built From Transistors?

Here is a NOT gate, the simplest, so you can see there's no magic. In real chips we use a technology called CMOS, which pairs two complementary kinds of transistor, but the essence is one transistor and a resistor:






```
IN = 1  →  transistor conducts → OUT is drained to ground → OUT = 0
IN = 0  →  transistor blocks   → OUT stays at supply       → OUT = 1
```

An AND gate takes about 6 transistors in CMOS; XOR takes about 8–12 depending on the design. The details are electrical engineering and we won't need them again. What we need is the punchline:

> **Transistors → gates. Gates are the atoms of computation.** From here up, we never think about electrons again. We think in 1s, 0s, and gates.

You've just climbed the first rung of the most important ladder in computing, the ladder of abstraction. We'll name it properly in Section 1.10.

A Delicious Mathematical Fact

How many possible two-input gates could exist? A two-input gate is just a truth table with 4 rows, and each row's output can independently be 0 or 1. So there are 2^4 = **16 possible two-input gates**. That's it. That is the entire universe of two-input logic. AND, OR, XOR are three of the sixteen; the others include NAND (NOT-AND), NOR, "always 0," "ignore B and output A," and so on.

And here's a fact that delights everyone who meets it: **one single gate type, NAND, can build all the others**, and therefore can build an entire computer. Every CPU and GPU on Earth could, in principle, be built from one gate design repeated billions of times. (You'll prove a piece of this yourself in Exercise 4.)

1.6 Making Gates Do Math: The Adder

Now the first real payoff. We'll wire gates together into a circuit that performs *addition*, the workhorse operation of all computing, and (as you'll see in Parts V–IX) essentially the only thing AI hardware does all day, at a rate of quadrillions per second.

ELI5

We want a circuit where you put in two numbers and out comes their sum. We'll start laughably small: adding two *one-digit binary numbers*. Each input is a single bit, 0 or 1.

There are only four possible problems this circuit will ever face:

```
0 + 0 = 0
0 + 1 = 1
1 + 0 = 1
1 + 1 = 2   ...uh oh.
```

“2” doesn’t fit in a single binary digit, binary digits only go 0, 1. Just like 7+5 in grade school, where you write 2 and *carry the 1*, binary 1+1 is written as `10`: sum digit 0, carry 1. (Chapter 2 makes binary counting second nature; for today, trust the table.)

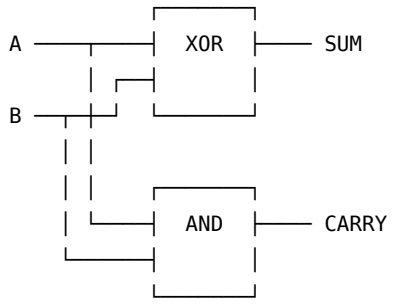
So our circuit needs **two outputs**: a SUM bit and a CARRY bit.

The Discovery

Write out what we need as a truth table, then stare at it:

```
A | B | CARRY | SUM
--+---+-------+----
0 | 0 |   0   |  0
0 | 1 |   0   |  1
1 | 0 |   0   |  1
1 | 1 |   1   |  0
```

Look at the CARRY column: it’s 1 only when both inputs are 1. *That’s exactly the AND gate.* Now the SUM column: it’s 1 when the inputs differ. *That’s exactly XOR.* We don’t need to invent anything. Addition was hiding inside the gates all along:

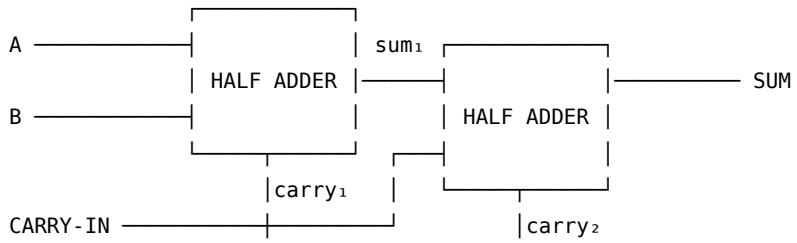


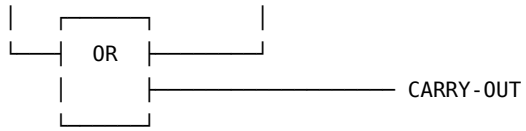
THE HALF ADDER: two gates that do arithmetic.

This circuit is called a **half adder**. Take a moment. Two bouncer-logic gates, wired side by side, and *arithmetic emerges*. This is the first miracle of computer science: **math is logic in disguise.** (George Boole suspected this in 1854; Claude Shannon nailed it to circuits in his 1937 master’s thesis, arguably the most consequential master’s thesis ever written.)

From Half to Full: Step-by-Step

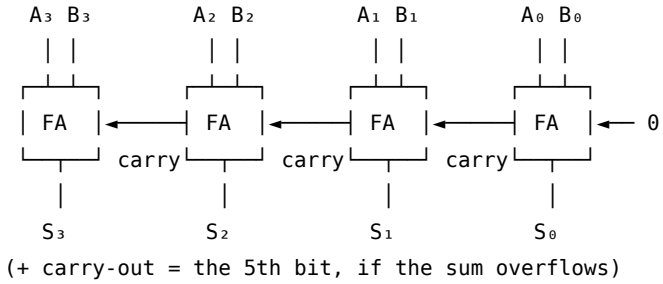
Why “half”? Because when we add multi-digit numbers, middle digits must accept a *carry from the previous column*, grade-school addition again. A circuit that adds three bits (A, B, and carry-in) is a **full adder**, and it’s just two half adders plus an OR:





In words: add A and B (first half adder), then add the incoming carry to that result (second half adder). A carry-out happens if *either* stage produced a carry, hence the OR. (Both can't carry at once, Exercise 5 asks you to prove it.)

Now the grand construction. Chain full adders, each column's carry-out feeding the next column's carry-in, exactly like grade school, and you can add numbers of any size. Here is a 4-bit **ripple-carry adder** (the carry "ripples" left, like a wave):



A 64-bit adder is this picture, 64 columns wide, around 1,000 transistors. Your laptop's CPU contains many dozens of adders. An H100 GPU contains, in effect, *hundreds of thousands* of adder-like circuits running simultaneously. When you hear "the GPU does a quadrillion operations per second," picture fields of these gate-chains, all rippling at once.

The Performance Seed (Plant It Now, Harvest It All Book)

Notice something subtle about the ripple-carry adder: column 3 *cannot finish* until column 2's carry arrives, which waits on column 1, which waits on column 0. The gates are all fast, but they're stuck in a *dependency chain*, a bucket brigade where each bucket waits for the previous one. A 64-bit ripple adder is ~64 gate-delays long even though it's only using 2–3 gate-delays of "real work" per column.

Hardware designers fixed this with cleverer circuits (carry-lookahead adders) that compute carries in parallel. But savor the shape of the problem, because you will meet it at every single layer of this book: **the enemy of speed is rarely the amount of work, it's the chain of waiting.** Dependency chains will reappear as pipeline stalls (Ch. 4), memory latency (Ch. 5), sequential bottlenecks in parallel algorithms (Ch. 33), and the reason attention kernels are hard to make fast (Ch. 52). Feynman's room knew it too: the assembly line is only as fast as its slowest, most-waited-on desk.

1.7 The Mathematics: Boolean Algebra

Time to put on slightly more formal glasses. Everything above has an elegant algebra underneath, invented by George Boole in 1854, almost a century before electronic computers, as an attempt to write the "laws of thought" as equations.

The Setup

Boolean algebra is arithmetic in a universe with only two numbers: 0 and 1. We define three operations, matching our gates. Notation you'll see in the literature:

- AND: written `A · B` or `A ∧ B` (behaves like multiplication: `1·1=1`, anything·0 = 0)
- OR: written `A + B` or `A ∨ B` (like addition, except `1+1=1`, the output can't exceed 1)
- NOT: written `Ā` or `¬A`

A **Boolean function** takes n bits in, produces bits out: mathematically, a function f: $\{0,1\}^n \to \{0,1\}^m$. Our half adder is a function from $\{0,1\}^2 \to \{0,1\}^2$. Truth tables are simply the complete listing of such a function, possible only because the input space is finite (2^n rows).

The Laws

These are the rewrite rules of the two-valued universe. Each is easy to verify by checking all input combinations:

```
Identity:        A · 1 = A            A + 0 = A
Annihilation:    A · 0 = 0            A + 1 = 1
Idempotence:     A · A = A            A + A = A
Complement:      A · Ā = 0            A + Ā = 1
Commutativity:   A · B = B · A        A + B = B + A
Associativity:   (A·B)·C = A·(B·C)    (A+B)+C = A+(B+C)
Distributivity:  A·(B+C) = A·B + A·C
De Morgan's:     ¬(A·B) = Ā + B̄       ¬(A+B) = Ā · B̄
```

De Morgan's laws are the celebrities: "NOT (A AND B)" is the same as "(NOT A) OR (NOT B)." In bouncer terms: failing a two-bouncer strict door means at least one bouncer rejected you. Obvious when said aloud; priceless when rewiring circuits.

Why a Compiler Book Cares About 170-Year-Old Algebra

Because these laws are **equalities**, and an equality is a license to *substitute*: anywhere the left side appears, you may install the right side instead, and *no observer can tell the difference*. If the right side is cheaper, fewer gates, fewer instructions, fewer nanoseconds, you just optimized for free.

That is a compiler's whole soul: **find provably-equal-but-cheaper rewrites, apply them, repeat.** Circuit designers use these laws to shrink transistor counts. LLVM uses richer algebraic rules to shrink instruction counts (`x*2` → `x<<1`; `x+0` → `x`, Chapter 21). ML compilers use tensor-level rules to shrink whole GPU workloads (fusing `add` into `matmul`, folding transposes, Chapter 39). Same move, three altitudes. Boolean algebra is where you learn the move with training wheels on.

Two little counting results worth keeping in your pocket:

1. **n wires can represent 2^n distinct states.** Ten switches: 1,024 states. 64 wires: 18.4 quintillion. This exponential is why fixed-size chunks of wires (Chapter 2's "words") can name astronomically many things.

2. **There are 2^(2ⁿ) distinct Boolean functions of n inputs.** For n=2: 16 gates, as we counted. For n=6, it's $2^{64} \approx 1.8 \times 10^{19}$ possible functions. The design space explodes instantly, a first hint of why "search for the best circuit/kernel" (auto-tuning, Chapter 42) is genuinely hard.

1.8 Remembering Things: A One-Paragraph Preview

Gates as described are *forgetful*: change the inputs and the output changes instantly; nothing is stored. A computer also needs to *remember*. The trick, one of the loveliest in engineering, is **feedback**: wire two gates so each one's output feeds the other's input, and the pair locks into a stable state, holding a 1 or a 0 until deliberately changed. Such a loop is called a **latch** or **flip-flop**, and it is one bit of memory built from the same gates as everything else. Chain millions of them (or use denser cousins like DRAM cells, a capacitor holding charge like a leaky bucket) and you have the pantry. Chapter 3 is devoted entirely to memory; here we only needed to establish that it exists and is made of the same stuff. From now on, assume we have a big array of numbered shelves that hold numbers.

1.9 The Big Idea: The Stored Program

Everything so far, gates, adders, memory, existed in some form in the mechanical and electromechanical machines of the early 1940s. Now comes the idea that created *software*, and with it the entire modern world. It is the single most important idea in this book.

The Problem It Solved

ENIAC, the great room-sized calculator of 1945, had a dirty secret: to change what it computed, technicians **rewired it**. Programming ENIAC meant days of re-plugging cables and setting switches by hand. The machine could add 5,000 numbers a second, but changing its *mind* took a week. The recipe was built into the machine's body, like a music box that plays one tune, forever, because the tune *is* the drum's pins.

The Idea

In 1945, John von Neumann circulated a report (building on work by Eckert, Mauchly, and, conceptually, Alan Turing a decade earlier) proposing something that sounds almost too simple:

> **Store the instructions in the same memory as the data. Instructions are just numbers.**

Sit with this. A recipe step like "add shelf 11's contents to the counter" gets encoded as a number, say, 211, and placed on a shelf *in the pantry itself*, right alongside the ingredients. The machine reads shelf 0, interprets the number found there as a command, obeys it, reads shelf 1, obeys, and so on.

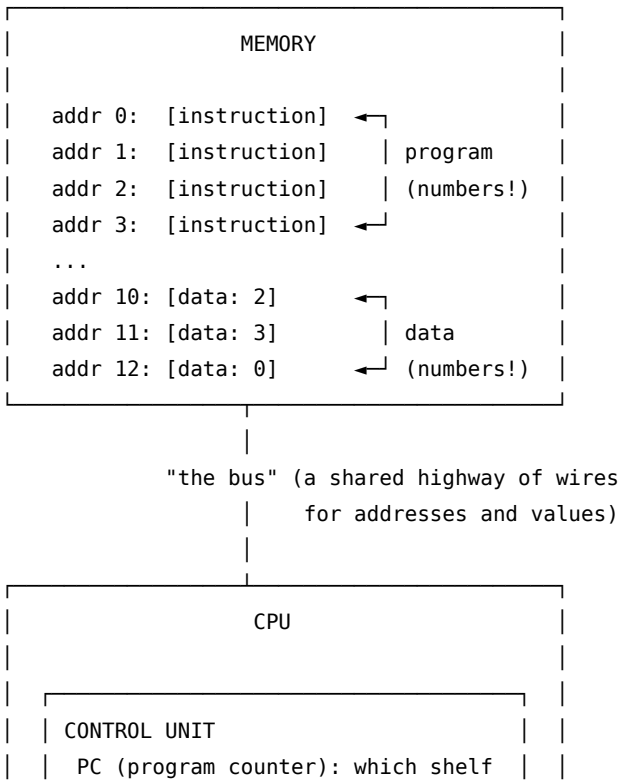
The consequences cascade fast, and each one is enormous:

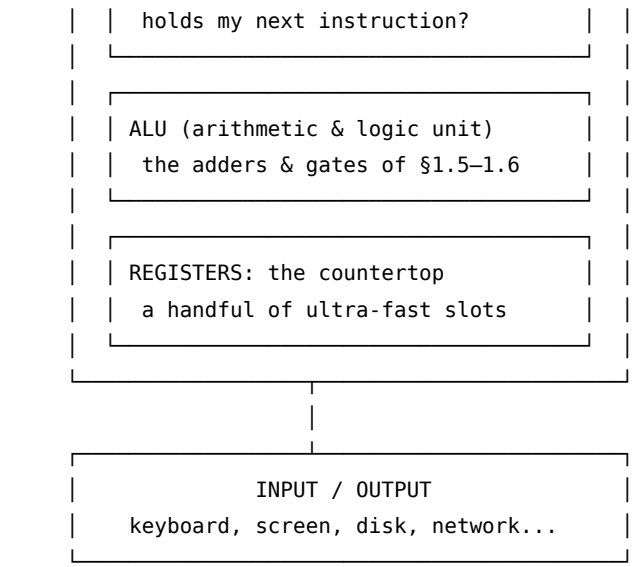
1. **Changing the program = changing memory contents.** No rewiring. Loading a new program takes milliseconds, not days. Software is born as something *soft*.
2. **Programs can be shipped, copied, and sold**, they're just numbers. Every app store, every `pip install`, every model checkpoint you'll ever download descends from this.
3. **Programs can read and write programs.** A program whose input is a program and whose output is a *better* program is called… a **compiler**. The tool this book teaches you to build is only possible because instructions are data.
4. **Programs can even modify themselves** while running, mostly a party trick today, but the principle powers JIT (just-in-time) compilers, which generate fresh machine code mid-execution. PyTorch 2's `torch.compile` (Chapter 45) does exactly this: it manufactures new instructions and places them in memory, and the machine gleefully runs numbers that didn't exist a second ago.

I want to be dramatic about this because it deserves drama: **the stored program concept is the reason "computer science" exists as a field.** Machines that calculate are engineering. Machines whose *behavior is data*, data that other machines can generate, analyze, transform, and optimize, that's a new kind of thing in the universe. When you write your first compiler pass in Part III, you'll be exercising the fourth consequence directly. You'll be the program that rewrites programs.

The Architecture, Drawn

The scheme von Neumann described is still, three-quarters of a century later, the blueprint of essentially every processor on Earth, your laptop's CPU, your phone's chip, and (with a twist we'll meet in Part VI: multiply everything by thousands) every GPU:





Map it back to the kitchen: memory is the pantry (one big one, holding both recipe cards and ingredients, that's the von Neumann twist), the ALU is the chef's hands, the registers are the countertop, and the control unit is the chef's eyes reading the recipe, with the **program counter (PC)** as the finger keeping their place on the card.

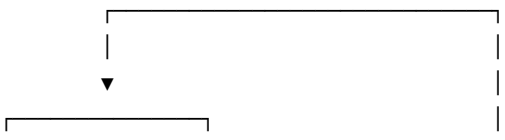
The Famous Flaw: The von Neumann Bottleneck

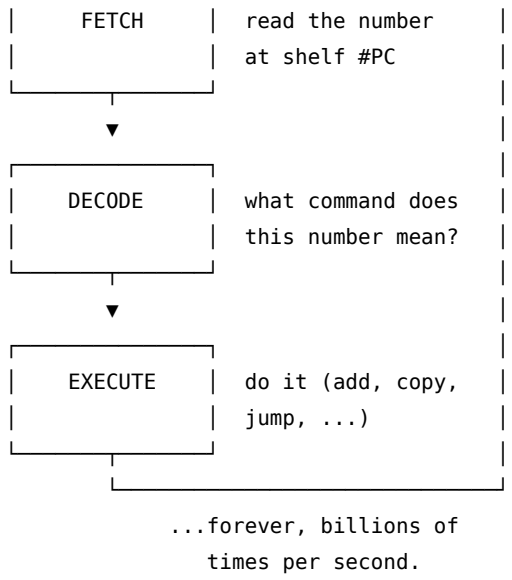
One highway (the bus) connects CPU and memory, and *everything*, instructions and data, travels on it. The CPU can compute far faster than the highway can deliver, so the processor spends much of its life waiting, like a chef drumming their fingers while an intern jogs to the pantry. John Backus (inventor of Fortran) named this the **von Neumann bottleneck** in his 1977 Turing Award lecture, and grumbled that we'd built our whole industry around "pushing vast numbers of words back and forth" through a straw.

He was right, and it got worse: since then, compute speed has improved orders of magnitude faster than memory speed. The entire memory hierarchy (Chapter 5), GPU shared memory (Chapter 35), operator fusion (Chapter 39), and FlashAttention (Chapter 52) are all, at heart, schemes for smuggling work around this one bottleneck. It is not an exaggeration to say **a large fraction of ML compiler engineering is a 75-year argument with this diagram.**

1.10 The Heartbeat: Fetch, Decode, Execute

So memory holds numbers, some of which are instructions. How does the machine actually *run*? With one loop. The simplest, most important loop in all of computing, the machine's heartbeat:





Fetch: read memory at the address in the PC. **Decode**: split the fetched number into "which operation?" and "operating on what?". **Execute**: fire the right circuit, the adder, the memory port, or (for jumps) overwrite the PC itself, which is how loops and decisions happen. Then advance the PC and go again. A 3 GHz processor sings this song roughly three billion times per second. It is the only song it knows.

Let's Build a Machine: TOY-100

Abstract explanations breed fuzzy understanding, and this book doesn't deal in fuzz. So we now define a complete, real (if tiny) computer, and by the end of the chapter you'll have built and programmed it yourself. We'll call it **TOY-100**, and we will keep it as a pet for the whole book, feeding it new abilities in Parts II and III until, in Part XI, we build a compiler that targets it.

The design (savor how little a "computer" requires):

- **Memory**: 100 shelves, addresses 0–99, each holding one number.
- **One register**: the accumulator, **ACC**, a countertop with room for exactly one number.
- **The PC**: which shelf to read the next instruction from. Starts at 0.
- **Instruction format**: each instruction is a 3-digit number. First digit = opcode (which command), last two digits = operand (which address). So `211` decodes as opcode 2, operand 11.

The eight commands (the **instruction set architecture**, or ISA, the machine's complete vocabulary):

```
opcode  name    meaning
------  ------  ------------------------------------------------
  0     HALT    stop the machine
  1     LOAD  a  ACC ← memory[a]          (pantry → countertop)
  2     ADD   a  ACC ← ACC + memory[a]
  3     SUB   a  ACC ← ACC − memory[a]
  4     STORE a  memory[a] ← ACC          (countertop → pantry)
```

```
5     JUMP  a  PC ← a                 (jump to step a)
6     JZ    a  if ACC == 0: PC ← a    (the machine's only "if"!)
7     PRINT a  print memory[a]
```

Eight commands. That's the whole vocabulary, and it's genuinely enough to compute anything computable, given time and memory. (This shocking claim is called Turing completeness; we'll touch it in Chapter 12.) Real ISAs like x86 have thousands of instructions, but they're conveniences, not necessities.

A Program, and a Hand Trace

Here's a complete TOY-100 program that adds 2 + 3 and prints the result. The left column is the shelf address; the middle is the number actually stored there; the right is what it means:

```
addr  contents   meaning
────  ────────   ──────────────────────────
  0     110      LOAD 10    (ACC ← mem[10])
  1     211      ADD 11     (ACC ← ACC + mem[11])
  2     412      STORE 12   (mem[12] ← ACC)
  3     712      PRINT 12
  4     000      HALT
 ...
 10       2     ← data: first number
 11       3     ← data: second number
 12       0     ← data: room for the answer
```

Notice: shelf 0 holds `110` and shelf 10 holds `2`, and *there is nothing marking one as "instruction" and the other as "data."* They're both just numbers on shelves. Only the PC's path through memory makes some numbers instructions. That's the stored-program idea made concrete.

Now trace it, slowly, by hand, the way you'd never let yourself do again but must do once:

```
step │ PC │ fetched │ decoded    │ effect                      │ ACC │ mem[12]
─────┼────┼─────────┼────────────┼─────────────────────────────┼─────┼────────
 1   │ 0  │  110    │ LOAD 10    │ ACC ← mem[10] = 2           │  2  │   0
 2   │ 1  │  211    │ ADD 11     │ ACC ← 2 + mem[11] = 5       │  5  │   0
 3   │ 2  │  412    │ STORE 12   │ mem[12] ← 5                 │  5  │   5
 4   │ 3  │  712    │ PRINT 12   │ prints "5"                  │  5  │   5
 5   │ 4  │  000    │ HALT       │ machine stops               │  5  │   5
```

Congratulations, you just executed a program by hand, doing exactly what the control unit's gates do. Every program you have ever run, your browser, a game, GPT generating text, is this table, with more rows. A large language model producing one word of output is this table with a few hundred billion rows, mostly `MULTIPLY` and `ADD`. There is no other magic. **This is the whole trick, done quickly.**

Where's the "If"? Where's the Loop?

Two of the eight instructions deserve special ceremony. `JUMP` and `JZ` write into the *PC itself*, they steer the machine's attention. `JZ` ("jump if zero") is the machine's entire capacity for

decision: every `if` statement, every `while` loop, every branching choice in every program ultimately compiles down to some flavor of "compare, then conditionally jump." Here's a countdown loop, 5-4-3-2-1, to show a program that revisits its own steps:

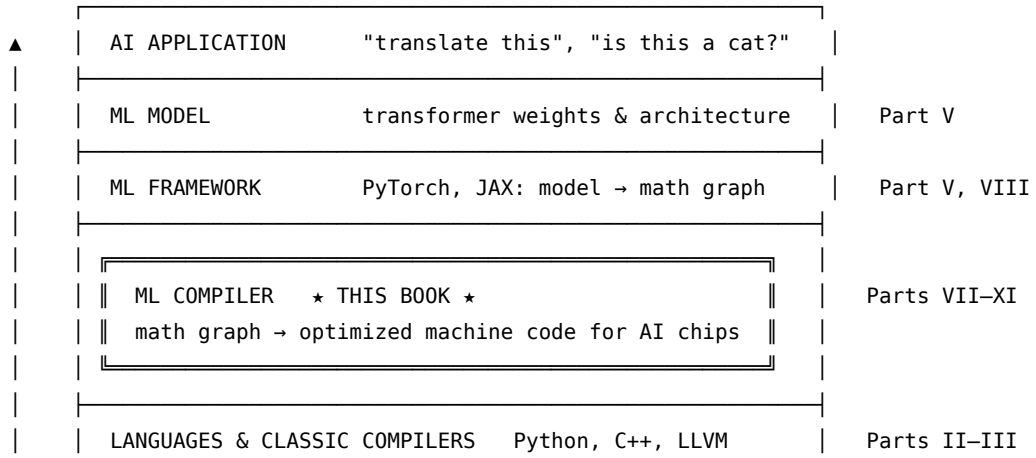
```
addr  contents   meaning
────  ────────   ──────────────────────────────────────
  0     110      LOAD 10      │ ACC ← counter
  1     608      JZ 8         │ if counter hit 0 → addr 8 (done)
  2     410      STORE 10     │ save counter
  3     710      PRINT 10     │ print it
  4     110      LOAD 10      │
  5     311      SUB 11       │ counter minus 1
  6     410      STORE 10     │ save it back
  7     500      JUMP 0       │ ...and around again
  8     000      HALT
 10       5      ← the counter
 11       1      ← the constant 1
```

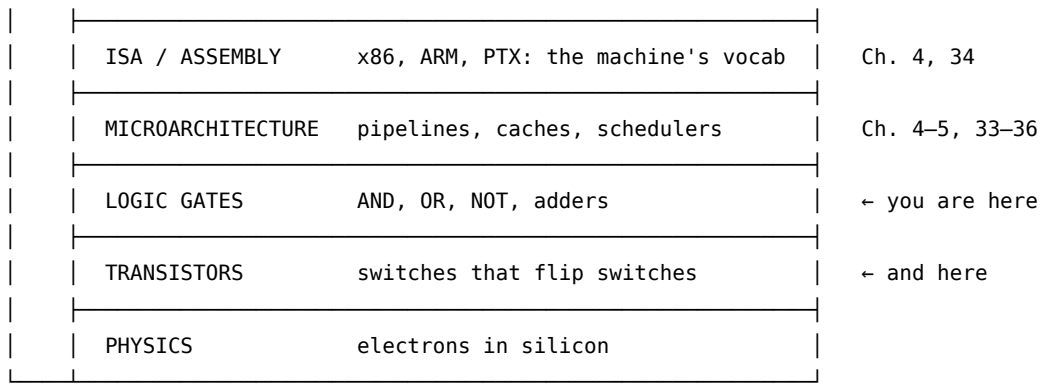
Trace three iterations of this yourself with a pencil (Exercise 2 makes it official). Feel how `JUMP 0` at address 7 throws the PC backward, and how `JZ 8` is the escape hatch that keeps it from looping forever. When you can predict this program's every step, you understand program execution better than most working programmers, not because they couldn't, but because they've never sat with it.

And a small aside that will pay off in Part III: look how much ceremony the loop needs, load, store, load, store, juggling one lonely accumulator. High-level languages exist to hide this ceremony; *compilers* exist to translate the pleasant version back into this, ideally with less ceremony than a human would write. The gap between "what you meant" and "what the machine needs" is the space every compiler lives in.

1.11 The Ladder of Abstraction: The Map of This Entire Book

We've climbed from electrons to running programs in ten sections. We'll pause on the landing, turn around, and look at the staircase, because the staircase itself is the most important mental model in computing, and it is literally the map of this book.





Three rules govern life on this ladder. They deserve frames on the wall:

Rule 1, Each layer only needs the layer below. The gate designer thinks in transistors, never electrons. The CPU designer thinks in gates, never transistors. You just programmed TOY-100 thinking in instructions, never gates, notice that you *could*, and that it felt natural. This is the only known way humans manage systems with 10^{11} parts: nobody understands the whole thing; everybody understands one floor and trusts the floor below. Abstraction is civilization's best trick, and computing is its purest expression.

Rule 2, Each layer is a *language*, and the borders between layers are *translators*. A PyTorch model is a sentence in the "tensor math" language. The machine speaks only the "instructions" language. Something must translate, and every translator on the ladder is called a compiler (or its sibling, an interpreter, Chapter 13 draws that line). ML compilers are the translators for the strangest border: from "mathematics on billion-element arrays" down to "keep 100,000 tiny adders fed without tripping over the von Neumann bottleneck."

Rule 3, Layers leak, and performance work is the study of the leaks. Correctness respects the floors; *speed does not.* Why is one PyTorch program 10× faster than another that computes the same thing? The answer is never on the PyTorch floor, it's three floors down, in cache behavior or warp scheduling, whispering up through the ceiling. A framework user can ignore the leaks. A performance engineer cannot. **An ML compiler engineer is a person who is comfortable on every floor of this building**, who can ride the elevator from "attention layer" to "memory coalescing" and back without getting dizzy. That's why this book insists on starting from transistors when its title says ML Compilers: we are not taking a detour; we are surveying the building you'll be working in.

Keep this diagram bookmarked. Every subsequent chapter is me pointing at one floor and saying: this one, now, in detail.

1.12 Engineering Perspective: The Real Thing

TOY-100 is honest but tiny. We'll calibrate against the machines this book ultimately targets, so you know how much the picture scales without changing shape.

The chips. A high-end CPU (say, an AMD EPYC or Apple M-series) packs tens of billions

of transistors; NVIDIA's H100 GPU, the workhorse of the current AI boom, has ~80 billion; Apple's M2 Ultra, ~134 billion. At ~5 nanometer feature sizes, transistors are so small that ~20,000 of them fit across a human hair's width. They are manufactured, mostly by one company, TSMC in Taiwan, using lithography machines built by one company, ASML in the Netherlands, each machine costing more than a Boeing 787, by projecting patterns of light onto silicon wafers, layer by layer, like silk-screening a t-shirt 60 times with atomic precision. A modern fab costs $20+ billion. This is, by most reckonings, the most precise manufacturing humanity does.

The clock. All those gates march to a metronome, the **clock**, ticking ~3–5 billion times per second (3–5 GHz) on CPUs, ~1–2 GHz on GPUs. Each tick, every circuit settles into its next state: fetch, decode, execute at industrial scale. Chapter 4 reveals the tricks (pipelining! Feynman's assembly line, in silicon) that let one tick retire *multiple* instructions.

The scale of the vocabulary. TOY-100 has 8 instructions; x86 has ~1,500+ variants; ARM, hundreds; NVIDIA GPUs speak PTX and SASS (Chapter 34). But every one of them is still a number in memory, fetched-decoded-executed. Trace table's the same; it just has more columns.

And the same trinity everywhere. Memory, processor, program. An H100 is: 80 GB of memory (pantry), ~16,000 arithmetic lanes organized into 132 "streaming multiprocessors" (a vast brigade of chefs), running kernels (recipes), with a von Neumann bottleneck so severe that feeding the chefs, not cooking, is the central engineering problem. You already have the concepts; Part VI just adds zeros.

1.13 Performance Analysis: Learning to Think in Nanoseconds

This is a performance book wearing a compiler book's clothing, so let's start training the reflex now: **always know the numbers.**

How Fast Is Fast?

A 3 GHz clock means one tick every **0.33 nanoseconds**. A nanosecond is a billionth of a second, for humans, an unimaginable sliver, so use this yardstick instead: **in one nanosecond, light travels about 30 centimeters.** One foot. A ruler on your desk.

Now the fact that makes chip design a branch of geography: in one 0.33 ns clock tick, light travels ~10 cm, and electrical signals in silicon are slower, managing only a few centimeters. **A signal cannot cross a large chip in one tick.** Distance on the chip is time. This is why chips can't just be made bigger, why "make the clock faster" hit a wall in the mid-2000s (plus heat, power grows roughly with the cube of clock speed), and why the industry's answer to "how do we go faster?" pivoted from *faster* to *more*: more cores, more parallel lanes, which handed the difficulty to software, which is why your career (this book) exists. When someone asks why we can't just build a 100 GHz CPU instead of wrestling with parallelism: physics said no, around 2005, permanently.

The Trillion-Fold Ascent

computer	additions/second	vs. previous
Human with pencil	~0.1	—
Los Alamos room (1944)	~10s, pipelined	~100×
ENIAC (1945)	~5,000	~500×
Cray-1 supercomputer (1976)	~160 million	~30,000×
Laptop CPU (2020s)	~100 billion	~1,000×
NVIDIA H100 GPU (2020s)	$\sim 10^{15}$ (a quadrillion)	~10,000×

From Feynman's room to an H100 is a factor of about $\mathbf{10^{14}}$, a hundred trillion. If walking speed had improved that much, you could stroll from Earth to the Sun in about a minute. This is why "dumb steps, but fast" wins: no amount of cleverness in a slow medium competes with adequate cleverness executed 10^{14} times faster.

The Two Numbers That Rule Everything: Latency and Throughput

Feynman's room teaches the distinction. Ask: *how long does one calculation take, start to finish?* That's **latency**, maybe hours, even in the optimized room, because hundreds of small steps happen in sequence. Now ask: *how many calculations finish per day?* That's **throughput**, and the assembly line made it enormous even though each individual problem still crawled.

- **Latency** = time for one thing. (One customer's meal: 20 minutes.)
- **Throughput** = things per unit time. (Kitchen output: 200 meals/hour.)

They are *not* reciprocals of each other, they respond to different medicine (parallelism boosts throughput but does nothing for a single item's latency), and confusing them is the most common performance-reasoning error in industry. The distinction will chase us everywhere: a GPU is a low-latency-be-damned throughput monster; interactive AI chat wants low latency per token; batch inference wants throughput; serving systems (Chapter 55) walk the tightrope between them daily.

Amdahl's Whisper

One more seed. In the room, suppose card-sorting between desks takes 10% of total time, and you parallelize the *arithmetic* infinitely, infinite clerks, free multiplication. Your maximum speedup is 10×, forever, because the sorting remains. The unimprovable part sets the ceiling: that's **Amdahl's Law** (formally in Chapter 7: speedup ≤ 1/(serial fraction)). It's why profiling (measuring where time actually goes) precedes optimizing in every competent performance effort, and why "we made the matmuls 2× faster but the model isn't faster" is a sentence you will someday say, understand, and fix.

1.14 Optimization: The Three Big Ideas (Already!)

It may seem premature to discuss optimization in Chapter 1. It isn't, because at this altitude you can see something that's invisible from inside the weeds: **there are only**

about three big ideas in all of performance engineering, and Feynman's room already used every one. The rest of this book is these three ideas wearing forty different costumes.

Idea 1, Parallelism: more workers. The room ran multiple problems as interleaved card colors through many human calculators at once. Costumes it will wear later: multicore CPUs (Ch. 6), SIMD vector units (Ch. 22), the GPU's tens of thousands of lanes (Ch. 33–34), tensor/pipeline/expert parallelism across whole datacenters (Ch. 56). Limit: Amdahl's ceiling, plus coordination costs, even in 1944, someone had to sort the cards.

Idea 2, Pipelining: never let a worker idle. The multiplication desk shouldn't wait for a problem to *finish*; the moment card #1 moves on, card #2 arrives. Latency per problem: unchanged. Throughput: multiplied by the number of stages. Costumes: CPU instruction pipelines (Ch. 4), GPU memory-latency hiding via warp swapping (Ch. 34, 36), software pipelining in kernels (Ch. 54), pipeline-parallel training (Ch. 56).

Idea 3, Specialization: shape the worker to the work. The person who *only multiplies* gets scarily good at multiplying, and a circuit that only multiplies 16×16 matrices beats a general processor at that job by 10–100× in speed and energy. Costumes: the GPU itself (specialized for throughput), tensor cores (Ch. 36, matrix-multiply organs), TPUs, and every "AI accelerator" startup. The price of specialization is rigidity, and **that price is paid in software, by compilers.** The more specialized the hardware, the harder the translation from flexible human intent, and the more valuable the translator. Specialization is, quite directly, why ML compiler engineers are scarce and expensive.

And braided through all three, the meta-idea we planted in the kitchen: **feed the workers, data movement, not arithmetic, is usually the real cost.** Fusion, tiling, memory planning, FlashAttention: pantry logistics, all of it.

When you meet any performance technique from now on, in this book or in the wild, run the checklist: *which of the three is this? And is it really about compute, or about feeding?* You'll be startled how completely those two questions dissect the field.

1.15 Build It: Gates and Adders in Python

Feynman's blackboard rule is this book's rule: what we cannot create, we do not understand. So we now create everything from this chapter, in Python. (If you've never written Python, follow along anyway, every line is explained in English, and Chapter 8 formally teaches programming. You can return and re-run this later; it will feel like meeting an old friend.)

Simulating Gates

```
# gates.py — the atoms of computation, simulated

def NOT(a):
    """The contrarian: output the opposite of the input."""
    if a == 1:
        return 0
```

```
    else:
        return 1

def AND(a, b):
    """The strict door: 1 only if BOTH inputs are 1."""
    if a == 1 and b == 1:
        return 1
    else:
        return 0

def OR(a, b):
    """The lenient door: 1 if AT LEAST ONE input is 1."""
    if a == 1 or b == 1:
        return 1
    else:
        return 0

def XOR(a, b):
    """The difference detector: 1 if the inputs DISAGREE.
    Built from the other gates to prove we can:
    XOR(a,b) = (a OR b) AND NOT(a AND b)  — 'at least one, but not both'."""
    return AND(OR(a, b), NOT(AND(a, b)))
```

Line-by-line, for complete beginners:

- `def NOT(a):`, `def` *defines* a reusable recipe (a **function**) named `NOT` that takes one input, which we'll call `a` inside the recipe. The colon and indentation mean "the following indented lines are the recipe's body."
- `if a == 1:`, a decision. `==` asks "are these equal?" (one `=` would *assign*, a classic beginner trap). Note the pleasing echo: Python's `if` will ultimately execute as some machine's conditional jump, a `JZ`, just like TOY-100's.
- `return 0`, "the answer is 0; recipe over." `return` hands a value back to whoever called the function.
- In `XOR`, we don't use `if` at all, we *compose* gates, feeding outputs into inputs, exactly as wires would: "(a OR b) AND NOT (a AND b)" is a direct transliteration of a real XOR circuit. Switches flipping switches, in software.

A dirty secret worth smiling at: Python's `and`, `or`, `not` mean we're using a computer's gates to pretend to be gates. Every simulation in this book has this Escher quality, we always build the thing out of the thing. That's not cheating; that's Rule 1 of the ladder (each layer builds on one below), plus the stored-program idea (behavior is data, so any machine can imitate any other). Sit with the vertigo until it becomes cozy; it's the field's defining sensation.

Verify Exhaustively

For 2-input gates, the input space is 4 rows, so we don't spot-check, we check *everything*. Remember this luxury and mourn it later: exhaustive verification dies of exponential explosion the moment inputs grow (64-bit adder: 2^{128} cases), which is why testing strategy (Chapter 61) is a real discipline.

```
print(" A  B | AND OR XOR")
for a in (0, 1):              # try both values of a...
    for b in (0, 1):          # ...and for each, both values of b
        print(f" {a}  {b} |  {AND(a,b)}   {OR(a,b)}   {XOR(a,b)}")
```

The nested `for` loops enumerate all four combinations, (0,0), (0,1), (1,0), (1,1). The `f"..."` is an f-string: the curly-brace holes get filled with each value. Output:

```
 A  B | AND OR XOR
 0  0 |  0   0   0
 0  1 |  0   1   1
 1  0 |  0   1   1
 1  1 |  1   1   0
```

Compare with §1.5's truth tables: identical. Our simulated universe agrees with the paper one.

The Adders

Now transliterate §1.6's circuits into code, notice the code *is* the circuit diagram, read as prose:

```
def half_adder(a, b):
    """Add two bits. Returns (sum_bit, carry_bit)."""
    s = XOR(a, b)        # the SUM wire   (1 if inputs differ)
    c = AND(a, b)        # the CARRY wire (1 only for 1+1)
    return s, c

def full_adder(a, b, carry_in):
    """Add three bits (two addends + incoming carry).
    Exactly the two-half-adders-plus-OR circuit of §1.6."""
    s1, c1 = half_adder(a, b)          # first half adder: a+b
    s2, c2 = half_adder(s1, carry_in)  # second: (a+b) + carry_in
    return s2, OR(c1, c2)              # carry out if EITHER stage carried

def add_bits(a_bits, b_bits):
    """Ripple-carry adder for equal-length bit lists (leftmost = biggest digit).
    Returns (sum_bits, overflow_bit)."""
    result = []
    carry = 0
    # Grade school: start from the RIGHTMOST column, hence reversed()
    for a, b in zip(reversed(a_bits), reversed(b_bits)):
        s, carry = full_adder(a, b, carry)   # this column's sum; carry ripples on
        result.append(s)
    result.reverse()                          # we built it right-to-left; flip it
    return result, carry
```

New Python in `add_bits`, explained: `[]` is an empty **list** (an ordered collection); `.append(x)` adds to its end. `reversed(...)` walks a list backwards. `zip(xs, ys)` pairs lists up element-by-element, so the loop sees one *column* of the addition at a time, rightmost column first, exactly like pencil-and-paper. The variable `carry` is the wave rippling leftward: each

column's carry-out becomes, on the next loop spin, the next column's carry-in.

Test, 3 + 5 (binary `0011` and `0101`, again, Chapter 2 drills this; today, trust):

```
s, overflow = add_bits([0,0,1,1], [0,1,0,1])
print(s, overflow)        # → [1, 0, 0, 0] 0
```

`[1,0,0,0]` is binary 1000 = eight. **3 + 5 = 8, computed by six simulated gates in a bucket brigade.** You have personally built arithmetic out of decisions. On a real chip this happens in a fraction of a nanosecond, sixty-four columns wide, millions of instances at once, but it is *this*, exactly this.

1.16 Mini Project: Build TOY-100 Itself

Now the chapter's summit: implement the whole computer, memory, PC, ACC, and the fetch-decode-execute heartbeat, in about 40 lines. This program is a **virtual machine** (VM): a machine made of software. (That phrase should now provoke a knowing nod instead of confusion: programs are data, so a program can *be* a machine. CPython, the JVM, and CUDA's PTX layer are this same idea at industrial strength, Chapters 13 and 34.)

```
# toy100.py — a complete computer in ~40 lines

# The vocabulary (ISA): opcode numbers from §1.10, given names for readability.
HALT, LOAD, ADD, SUB, STORE, JUMP, JZ, PRINT = 0, 1, 2, 3, 4, 5, 6, 7

def decode(instruction):
    """Split a 3-digit number into (opcode, operand): 211 → (2, 11)."""
    opcode  = instruction // 100    # integer division: 211 // 100 = 2
    operand = instruction % 100     # remainder:        211 % 100  = 11
    return opcode, operand

def run(memory, trace=False):
    """The fetch-decode-execute heartbeat. `memory` is a list of 100 ints."""
    pc  = 0                         # program counter: start at shelf 0
    acc = 0                         # the accumulator (countertop): starts empty

    while True:                     # the heartbeat loops until HALT breaks it
        instruction = memory[pc]                # ── FETCH
        opcode, operand = decode(instruction)   # ── DECODE
        if trace:
            print(f"pc={pc:2d}  instr={instruction:03d}  acc={acc}")
        pc += 1                     # advance the bookmark BEFORE executing
                                    # (so jumps can overwrite it cleanly)
        # ── EXECUTE: exactly one of these fires
        if   opcode == HALT:  break
        elif opcode == LOAD:  acc = memory[operand]
        elif opcode == ADD:   acc = acc + memory[operand]
        elif opcode == SUB:   acc = acc - memory[operand]
```

```
        elif opcode == STORE: memory[operand] = acc
        elif opcode == JUMP:  pc = operand
        elif opcode == JZ:
            if acc == 0:
                pc = operand
        elif opcode == PRINT: print(memory[operand])
        else:
            raise RuntimeError(f"Unknown opcode {opcode} at address {pc-1}")

    return memory                    # hand back memory so tests can inspect it
```

The lines that repay staring:

- `HALT, LOAD, ... = 0, 1, ...`, names for the opcode numbers, nothing more. The machine only ever sees numbers; the names are for *us*. (Foreshadowing: an assembler, Chapter 15's warm-up, is a program that converts names to numbers so humans never write `211` again. You'll build one in Project B.)
- `decode`, pure arithmetic surgery: `//` chops off the last two digits, `%` keeps them. Real CPUs decode with gates that slice *binary* fields rather than decimal digits, but it's the same knife.
- `pc += 1` **before** execute, the little design decision with real consequences: since the PC already points *past* the current instruction, a `JUMP` simply overwrites it and nothing fights. Real ISAs do the same (when x86 code reads its own instruction pointer mid-instruction, it sees the *next* address). You've hit your first piece of *semantics pedantry*, precise rules about tiny orderings, which is the daily bread of compiler correctness (Chapter 18 onward).
- `while True:` + `break`, the heartbeat has no natural end; only `HALT` stops it. Feed it a program without a reachable HALT and it runs forever, you'll engineer this disaster deliberately in Exercise 7, because a bug you've built on purpose is a bug you'll recognize forever after.
- The `if/elif` chain, this is the control unit: look at the opcode, fire one circuit. A real control unit is this chain *as hardware*, all comparisons in parallel.

Run the Programs

```
# --- 2 + 3, from §1.10 ---
memory = [0] * 100        # 100 shelves, all holding 0
memory[0] = 110           # LOAD 10
memory[1] = 211           # ADD 11
memory[2] = 412           # STORE 12
memory[3] = 712           # PRINT 12
memory[4] = 0             # HALT
memory[10] = 2            # data
memory[11] = 3            # data
run(memory)               # prints: 5

# --- countdown 5..1, from §1.10 ---
memory = [0] * 100
program = [110, 608, 410, 710, 110, 311, 410, 500, 0]
```

```
memory[:len(program)] = program      # copy program to shelves 0..8
memory[10] = 5                       # the counter
memory[11] = 1                       # the constant 1
run(memory)                          # prints: 5 4 3 2 1 (one per line)
```

(`[0] * 100` builds a 100-zero list; `memory[:len(program)] = program` pastes the program over shelves 0–8. Loading a program *is* writing numbers into memory, the stored-program concept as a single line of code.)

Run the first one with `trace=True` and watch §1.10's hand-trace table print itself, live, machine-generated. There is something quietly moving about that: the pencil trace and the running program agree, because they are the same computation at two altitudes.

Why This 40-Line Toy Is Load-Bearing for a 1,000-Page Book

Because it is the **target**. When Part III teaches compilation, TOY-100 is what we'll compile *to*, you'll write a compiler that turns `x = 2 + 3` into `[110, 211, 412]` automatically. When Part XI builds an ML compiler, a grown-up TOY-100 with tensor instructions will be its backend. And when you meet real IRs, LLVM (Ch. 23), PTX (Ch. 34), they'll feel familiar, because you'll have owned a machine since Chapter 1. We didn't build a toy; we built the *anchor*.

1.17 Debugging Guide: When the Machine Does the Wrong Thing

Debugging is the half of engineering that tutorials skip and professionals live in. We start the habit in Chapter 1 and never drop it. Three lessons, sized to what we've built:

Lesson 1, The machine is never wrong. TOY-100 (real hardware, too, to a first approximation) does *exactly* what the numbers in memory say. "The computer is misbehaving" means, with probability indistinguishable from 1, "my numbers don't say what I believe they say." This mindset shift, from *why is it doing that?!* to *what did I actually tell it?*, is 60% of debugging skill, acquirable today, on a 40-line machine.

Lesson 2, Make the invisible visible, then read what it says. Execution is invisible; that's why bugs survive. Our `trace=True` is the antidote, and it is not a toy feature. It is this book's first **observability tool**, ancestor of a noble line: `gdb` single-stepping (Ch. 14), LLVM's `-print-after-all` showing IR between passes (Ch. 23), `TORCH_LOGS` dumping Inductor's decisions (Ch. 45), Nsight tracing GPU kernels (Ch. 36). Same instinct every time: *don't theorize about what the machine did, make it tell you.* A worked example: suppose you reorganize the countdown program and it starts printing `5 4 3 2 1 0`, one extra line. You could re-read the code and squint. Instead, run the trace: it will show you the exact step where `PRINT` fired with `mem[10] = 0`, and the PC value tells you which instruction did it, you'll see immediately that your rearrangement put the `PRINT` *before* the `JZ` check instead of after it. The trace converts "somewhere in my program" into "address 3, step 31." Shrinking the *where* is the whole game of debugging.

Lesson 3, Binary-search the timeline. When a long run goes wrong, don't read 10,000

trace lines. Check the middle: state healthy? Bug's in the second half. Sick? First half. Repeat. Feynman's room did precisely this to salvage contaminated calculations, find the first bad card, recompute forward from there. $\log_2(10{,}000) \approx 14$ checks beats 10,000. This is the same divide-and-conquer that will find miscompiled functions via bisection in Chapter 61 (`llvm-reduce`, pass bisection), a professional technique, fully learnable on a toy.

And one professional reflex to install immediately: **when you find a bug, first write the smallest program that triggers it** (ours: three instructions?), *then* fix it, then keep the small program as a permanent test. Chapter 61 will name this regression testing; you can practice it in tonight's exercises.

1.18 Interview Questions

These are real questions asked in systems/new-grad interviews (and, in sharper forms, in ML systems interviews). Attempt each before reading the answer, the struggle is the workout; the answer is just the stretch after.

Q1. "Explain what a computer does, to someone non-technical, in under a minute." *(Tests: whether your understanding survives without jargon to hide behind.)* Strong answer: A computer is a machine that follows a recipe of extremely simple steps, copy a number, add two numbers, and if some number is zero, skip to a different step. What makes it useful isn't cleverness, it's speed: billions of steps per second, without fatigue or boredom. Everything impressive, video, games, AI, is billions of dumb steps arranged cleverly by humans. The intelligence is in the recipes.

Q2. "What is the stored-program concept and why did it matter?" Programs are stored in the same memory as data, an instruction is just a number on a shelf. Consequences: reprogramming = rewriting memory (seconds, not days of rewiring); programs become shippable, copyable data; and programs can process other programs, which is what makes compilers, interpreters, operating systems, and JITs possible at all. Strong finisher: it also created computing's biggest security headache (data smuggled into being treated as code, buffer overflows), and its biggest performance headache, the von Neumann bottleneck, since instructions and data share one memory highway.

Q3. "Walk me through fetch-decode-execute." Loop forever: **Fetch** the number at the address in the program counter. **Decode** it, split into operation and operands. **Execute**, fire the matching circuit; ordinary instructions then advance the PC by one, while jumps write a new value into the PC, which is how ifs and loops exist. Bonus points: mention that modern CPUs run many iterations of this loop *overlapped* (pipelining) and even out of order, while carefully preserving the illusion of one-at-a-time execution, the "illusion" framing signals real understanding (and sets up Chapter 4).

Q4. "Why can't we just keep increasing clock speed instead of going parallel?" Two walls. **Heat**: power dissipation grows superlinearly with frequency (roughly $\propto V^2 f$, and higher f demands higher V), so past ~4–5 GHz air-cooled silicon literally can't shed the watts. **Distance**: at 3 GHz a signal travels only a few centimeters per tick in silicon, near chip scale, so faster clocks would mean signals can't cross the chip in a cycle. Both walls arrived ~2005; the industry's pivot was "more workers, not faster workers," which moved the burden onto parallel software, and is the direct economic reason GPU

programming and ML compilers are careers.

Q5. "What's the difference between latency and throughput? Give an example where improving one doesn't improve the other." Latency: time for one item, start to finish. Throughput: items per second. A pipeline (assembly line) multiplies throughput while leaving each item's latency untouched, or slightly worse. Adding lanes to a highway raises throughput; it doesn't get *one* car there faster. Sharp example for ML-flavored interviews: batching inference requests raises tokens/second served (throughput) while *increasing* each user's time-to-response (latency), the central tension of Chapter 55.

Q6. "What is the von Neumann bottleneck?" CPU and memory share one channel; instructions and data compete for it; the processor computes far faster than the channel delivers, so it starves. Modern mitigations: caches (Ch. 5), prefetching, wide memory buses, and, at the ML-compiler level, *restructuring computation itself* to touch memory less: fusion, tiling, recomputation (Ch. 39, 52). If you can name FlashAttention as "a scheme to defeat the bottleneck for attention," you've turned a definition question into a hire signal.

Q7. "One gate type can build any computer. Which, and roughly why?" NAND (NOR also works). Why: NOT is NAND with both inputs tied together; AND is NAND then NOT; OR falls out via De Morgan's law from NOTted inputs. With AND/OR/NOT you can realize any truth table (write it as a sum-of-products), and with gates + feedback you get memory too, so NAND suffices for logic *and* storage. (You'll verify the first steps in Exercise 4.)

Q8. (Systems-design flavored) "You have 1,000 clerks with calculators and a month of arithmetic. Organize them." They're asking you to reinvent Feynman's room; do it knowingly. Decompose into small steps; specialize clerks (multipliers, adders, checkers); pipeline the flow so nobody idles; run independent problems in parallel (colored cards); add error-checking rows and recompute only contaminated work; watch for the bottleneck desk (profile!) and re-balance staff toward it; batch similar operations to amortize setup. Every clause maps onto a chapter of this book, which is rather the point of this book.

1.19 Common Mistakes and Misconceptions

"The computer understands English / Python / code." It doesn't, any more than a music box understands music. Understanding-shaped behavior is produced by layers of translation (the ladder!) ending in gate-flips. Practical consequence: when the machine "misunderstands" you, some translator on the ladder turned your intent into instructions you didn't expect, and *that's* where to look. Blame flows down the ladder, never sideways into mysticism.

"Computers are smart." Inverted: they are stupendously dumb and stupendously fast, and the trade is worth it because $10^{14}\times$ speed beats cleverness. Engineers who forget this start expecting the machine to "figure out what I meant", a category error that costs real debugging hours. (Yes, LLMs complicate the *feeling* of this statement. They don't complicate the fact: an LLM is dumb steps, arranged by training into smart-shaped behavior, Part V.)

"Hardware and software are fundamentally different things." They're interchangeable costumes for computation: any circuit can be simulated in software (you did it today, TOY-100), any program can be baked into a circuit. What differs is the *trade-off*: hardware is fast and frozen; software is slow and fluid. Where to draw that line is a live billion-dollar question, it is literally the "should we build a custom chip for transformers?" debate (Part XII), and engineers who see the line as movable think better than those who see it as natural law.

"Memory and storage are the same thing." In this book, **memory** (RAM) = the fast, volatile pantry the CPU works from; **storage** (disk/SSD) = the slow, permanent warehouse. Muddling them muddles every performance conversation you'll ever have; Chapter 3 formalizes the difference (spoiler: ~100,000× latency gap).

"More GHz = faster computer" / "GPU beats CPU." Neither, unconditionally. Clock speed is one factor among many (work per tick, core count, memory speed, Ch. 4–7). GPUs devastate CPUs on throughput-parallel work and *lose* on serial, branchy, latency-bound work (Ch. 33). "Faster at *what*?" is the only defensible phrasing. Interviewers deduct for the unconditioned claim.

"The trace/hand-simulation stuff is beneath me; I get the concept." The concept isn't the skill. Predicting *exactly* what a machine does, step by step, is the skill, the same muscle that later reads IR dumps, disassembly, and kernel traces without flinching. Every strong systems engineer I can point to does low-level traces *fluently*, and fluency comes only from reps. Do the pencil traces. Tonight. Nobody's watching.

"1 + 1 = 2, so my half adder should output 2." A single wire can't say "2", it's ON or OFF. Two-bit answers need two wires (carry & sum: `10`). Sounds trivial; it is the seed of the deepest recurring theme in numerical computing: **representations have finite capacity, and arithmetic that overflows its container corrupts silently.** In Chapter 27 this exact issue, sums outgrowing FP16's container, will be why loss-scaling exists and why your training run NaN'd at 3 a.m.

1.20 Exercises

Pencil-and-paper first, resist the urge to skip to code. (Estimated effort in parentheses.)

Exercise 1, Truth-table drills. (10 min) Write full truth tables for: (a) NAND (NOT of AND); (b) NOR; (c) the three-input majority function MAJ(A,B,C) = 1 when two or more inputs are 1. Then express MAJ using AND/OR, and check your expression against all 8 rows. (Majority gates are the carry logic of full adders, compare your table's pattern to §1.6.)

Exercise 2, The pencil trace. (15 min) Hand-trace the countdown program of §1.10 in a table with columns: step, PC, fetched, decoded, ACC, mem[10], output. Run it to completion (≈ 40 steps). Check yourself against `run(memory, trace=True)`. If any row disagrees, find the *first* disagreeing row, Lesson 3's timeline-bisection, in miniature.

Exercise 3, Reverse engineering. (15 min) Without running it, determine what this program does, then verify on your VM: shelves 0–6 = `[110, 611, 211... ]`, wait, no hints from formatting; here it is plainly: `mem[0..6] = [110, 605, 312, 410, 500, 710, 0]`, `mem[10] = 12`,

`mem[11] = 0`, `mem[12] = 3`. (Sneaky detail: what's at shelf 11, and does JZ ever fire? What happens on shelf 10 over time, and does this program *terminate*?)

Exercise 4, NAND builds the world. (20 min) Using *only* NAND: build NOT (hint: feed the same signal to both inputs), then AND, then OR (hint: De Morgan, OR(a,b) = NAND(NOT a, NOT b)). Verify each with truth tables. You have now shown every circuit in this chapter, hence every computer, could be one gate design, repeated.

Exercise 5, A tiny proof. (10 min) In the full adder, prove the two half adders can never *both* produce carry = 1 (hence the OR never receives 1,1, some texts use XOR there instead; explain why both work). Hint: the first half adder's carry is 1 only when A=B=1; what is its *sum* output in that case, and what does the second half adder then see?

Exercise 6, De Morgan on the street. (10 min) "The alarm fires unless the door is closed and the badge is valid." Write the alarm's Boolean expression two ways, as a NOT-of-AND and, via De Morgan, as an OR, and argue in plain English that they describe the same policy. (This tiny rewrite is precisely the kind a compiler makes when one form is cheaper on the target machine.)

Exercise 7, Build the disaster. (10 min) Write the shortest TOY-100 program that never halts. Then write one that never halts *for a subtler reason*, a JZ that can never fire because of arithmetic upstream. Run both, feel the hang, Ctrl-C, and reflect: from the *outside*, can you tell "still working" from "looping forever"? (You've just met the halting problem's shadow; Chapter 12 introduces it properly, and it will constrain what optimizing compilers can ever promise.)

Coding Exercises

Coding 1, Multiply on a machine with no multiply. (30 min) TOY-100 has no MUL. Write a TOY-100 program (numbers in memory, not Python!) that computes `mem[10]` × `mem[11]` into `mem[12]` by repeated addition with a loop counter. Test with 6 × 7 = 42, and the edge cases 0 × 5 and 5 × 0. (Every ISA omits something; synthesizing missing operations from present ones is a compiler's daily chore, Chapter 22 calls it *instruction selection* and *legalization*.)

Coding 2, Extend the ISA. (20 min) Add opcode 8, `INC a` (mem[a] ← mem[a] + 1) to `toy100.py`: pick the encoding, extend `decode`'s reach if needed, add the `elif`, write a test. Then the interesting question, in one written paragraph: rewrite Coding 1's multiply using INC, how many *fewer* instructions execute for 6 × 7? You've just done a hardware/software co-design trade study, the game Chapter 36's tensor cores play at billion-dollar stakes.

Coding 3, A 16-bit adder and its limits. (20 min) Using `add_bits`, compute 40,000 + 30,000 with 16-bit inputs. The true answer (70,000) exceeds 16 bits' capacity (65,535), inspect the overflow bit. Now *ignore* the overflow bit and decode the 16 sum bits you got: what wrong-but-tidy number appears? (Preview of Chapter 2's modular arithmetic, the machine doesn't crash on overflow; it *wraps*, silently, which is worse.)

Coding 4, Self-modifying code (advanced, optional). (30 min) Write a TOY-100 program that STOREs a computed value *into an address the program itself will later execute*, e.g., manufacture the instruction `710 + k` in ACC and store it ahead of the PC. Predict the behavior, then trace it. One paragraph: why do modern systems mark memory pages

"no-execute" to forbid exactly this in most software (hint: Q2's security note), and why do JIT compilers (Ch. 13, 45) get a special pass?

Mini Projects

Project A, TOY-100 Deluxe. (2–4 hrs) Grow your VM: (1) a `step()` method + interactive mode, press Enter to advance one instruction, printing state (you're building a debugger; Chapter 14's gdb will feel like family); (2) breakpoints: `run(memory, break_at={3})` pauses when PC hits address 3; (3) a cycle counter reporting instructions executed, your first *profiler*, plus a per-opcode histogram (which instruction dominates the countdown program? multiply?); (4) guardrails: catch reads of nonsense instructions and out-of-range addresses with *helpful* messages naming the address and PC. (Error-message craftsmanship is compiler UX, Chapter 16 onward.)

Project B, The Assembler: your first translator. (3–5 hrs) Write `assemble(text) -> memory` turning this pleasant text into TOY-100 numbers:

```
       LOAD  counter
top:   JZ    done
       STORE counter
       PRINT counter
       LOAD  counter
       SUB   one
       STORE counter
       JUMP  top
done:  HALT
counter: 5
one:     1
```

Requirements: two passes (first pass records what address each label like `top:` lands on; second pass emits numbers, filling label references in, try writing it single-pass and discover *why* forward references force two passes); comments with `#`; errors that name the offending line. **Understand what you will have built: a translator from human-friendly notation to machine numbers.** A small compiler. The 1,000-page journey from here to "ML compiler" is this project, deepened, better source languages (Parts II–III), better targets (Part VI), and above all *optimization* between the two (everything else). You've started.

1.21 Chapter Summary

We began in a room in 1944 where "computer" still meant a person, and Feynman was inventing pipelining with colored index cards. We ended with you building a working computer in Python. The route: a transistor is a switch that other switches can flip, which makes logic *composable*; a handful of switch-patterns, AND, OR, NOT, XOR, are the atoms of decision; arithmetic was hiding inside logic all along (half adder = one XOR + one AND), and Boolean algebra's rewrite laws are the original optimization license: replace anything with anything provably equal but cheaper. Feedback loops give gates memory; one shared memory holding *both* data and instructions-as-numbers, the

stored-program concept, turns behavior into data and makes compilers possible; and a three-beat heartbeat, fetch-decode-execute, animates it all, with conditional jumps as the machine's entire capacity for choice. Above this rises the ladder of abstraction, with ML compilers occupying one specific, strange floor: translating mathematics-on-tensors down to instruction streams for the most parallel machines ever built. And already visible from Chapter 1: the three big ideas of performance (parallelism, pipelining, specialization), the tyranny of data movement over arithmetic, latency versus throughput, Amdahl's ceiling, and the debugging creed, the machine is never wrong; make it show you what you actually said.

1.22 Key Takeaways

- **A computer is not smart; it is fast.** All intelligence lives in the recipes, and in the people (and compilers) that write them.
- **Composability is the miracle.** Switches flip switches → gates → adders → machines. Each layer speaks only to the next: the ladder of abstraction.
- **Math is logic in disguise.** SUM = XOR, CARRY = AND. Shannon's observation started everything.
- **Equality is a license to optimize.** Boolean laws, and every compiler rewrite descended from them, swap equals for cheaper equals. Correct-but-cheaper is the entire compiler business model.
- **Instructions are numbers.** Therefore programs are data; therefore programs can write, analyze, and improve programs; therefore compilers. The stored-program concept is the field's founding act.
- **The heartbeat is fetch–decode–execute**, and jumps that overwrite the PC are all the "choice" a machine has.
- **The bottleneck is the bus.** Arithmetic is cheap; moving data is dear. Most of ML-compiler engineering is pantry logistics against the von Neumann bottleneck.
- **Three ideas rule performance:** parallelism, pipelining, specialization, all invented by a room full of humans in 1944, all wearing silicon costumes now.
- **Latency ≠ throughput,** and Amdahl's Law caps every parallel dream at 1/(serial fraction).
- **Debug by making the machine confess:** trace it, bisect the timeline, shrink the reproducer. The machine is never wrong.
- **You own a computer now.** TOY-100, 8 instructions, 100 shelves, 40 lines, is real, it's yours, and this book will grow it into an ML compiler's target.

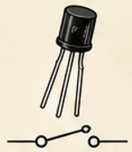
What's Next

TOY-100's shelves held friendly decimal numbers, a small lie of convenience. Real machines store *everything*, numbers, text, instructions, images, and the 70 billion weights of a language model, as patterns of 1s and 0s, and the *choice of pattern* turns out to be a genuine engineering decision with billion-dollar consequences (ask anyone who's debugged an FP16 overflow). Chapter 2, **Binary: The Language of Machines**, teaches you to read, write, and *feel* binary: counting, negative numbers via a beautiful trick called two's complement, why $0.1 + 0.2 \neq 0.3$ on every computer on Earth, and the first honest look at the floating-point formats, FP32, FP16, BF16, whose trade-offs quietly govern all of modern AI.

End of Chapter 1. Next: Chapter 2, “Binary: The Language of Machines.”

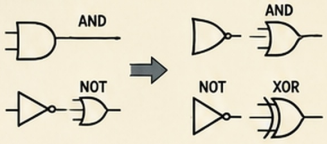
FROM SWITCHES TO SYSTEMS. FROM IDEAS TO IMPLEMENTATION. NOW YOU CAN BUILD A WORKING COMPUTER IN PYTHON.

TRANSISTOR (c. 1947)



A switch that other switches can flip.

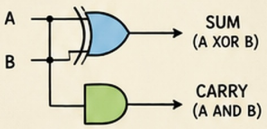
TRANSISTOR 1947) → LOGIC GATES (1950s)



The atoms of decision.

ARITHMETIC FROM LOGIC

HALF ADDER



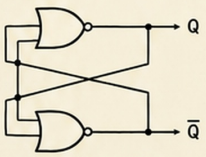
Arithmetic was hiding inside logic all along.

BOOLEAN ALGEBRA

$$A + 0 = A$$
$$A \cdot 1 = A$$
$$A + A = A$$
$$A \cdot \bar{A} = A$$
$$A + \bar{A} = 1$$
$$A \cdot \bar{A} = 0$$
$$A + BC = (A + B)(A + C)$$

Rewrite laws = Original optimization license.

FEEDBACK GIVES MEMORY

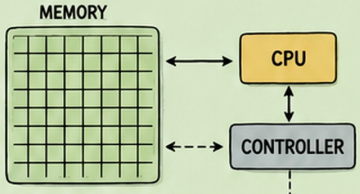


State = the past that influences the future.

ONE SHARED MEMORY: DATA + INSTRUCTIONS-AS-NUMBERS

ADDRESS	CONTENT (NUMBERS)	
0	0001 0101	(LOAD 5)
1	0011 0000	(ADD 0)
2	0101 0001	(STORE 1)
3	1110 0000	(JMP 0)
4	0000 0101	(DATA 5)
5	0000 0000	(DATA 0)
...	...	

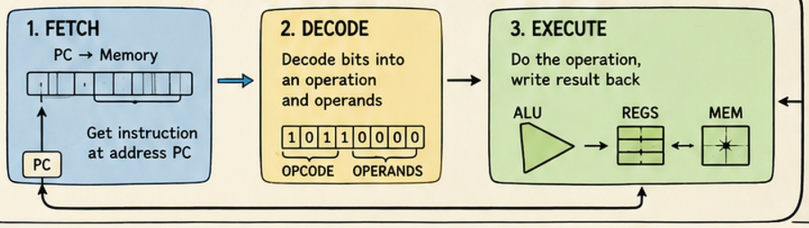
Same memory holds code and data.



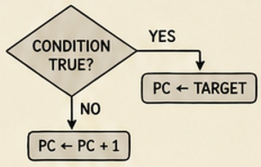
STORED-PROGRAM CONCEPT

Behavior becomes data. Programs can be written, changed, loaded, and compiled.

THE MACHINE HEARTBEAT: FETCH – DECODE – EXECUTE (REPEAT)

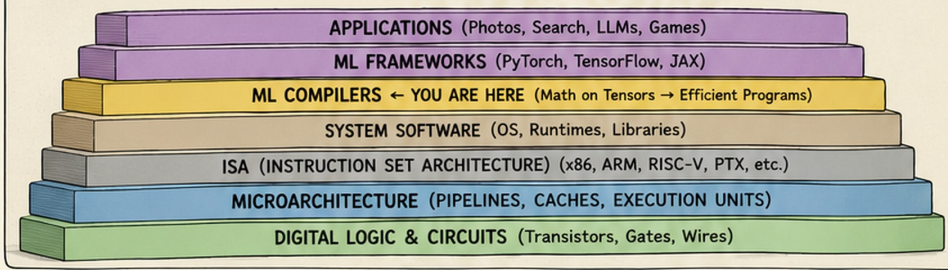


CHOICE = CONDITIONAL JUMP



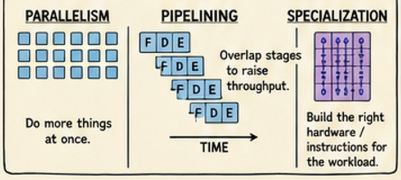
The machine's entire capacity for choice.

THE LADDER OF ABSTRACTION



Each floor speaks the language of the one below it.

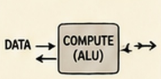
THREE BIG IDEAS OF PERFORMANCE (ALREADY VISIBLE)



Do more things at once.

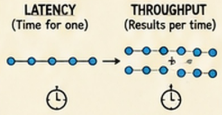
Build the right hardware / instructions for the workload.

THE TYRANNY OF DATA MOVEMENT



Moving data is often more expensive than doing math.

LATENCY vs THROUGHPUT



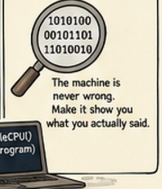
Don't confuse the two.

AMDAHL'S LAW

$$\text{Speedup} \le \frac{1}{(1-P)+P/S}$$

The serial part sets the ceiling.

DEBUGGING CREED



The machine is never wrong. Make it show you what you actually said.