THE LOGIC BENCH WORKBOOK

From a Single Gate to a Working CPU

A hands-on course in digital logic — from one switch and one lamp, through gates, arithmetic, memory and sequencing, to a real 8-bit CPU and the MOS 6502, built from gates and openable all the way down.

Logic Bench · every box opens, nothing inside is fake

Contents

Part IV — Memory

Part V — Sequential machines & the clock

Part VI — Routing & codes

Part VII — Buses & larger structure

Part VIII — Stored memory: RAM & ROM

Part IX — The CPU

Part X — The 6502

Appendices

Part I — The Atoms (native symbols)

Lesson 01 — The Switch

> **Part I · The Atoms** — native symbol The first lesson. After this: the Button (Lesson 02).

What you will learn

- What a switch does and why it is where almost every circuit begins.
- The difference between a signal that is **held** on and one that is only momentary.
- How to read the lit/dark state of a wire from its source.

The idea

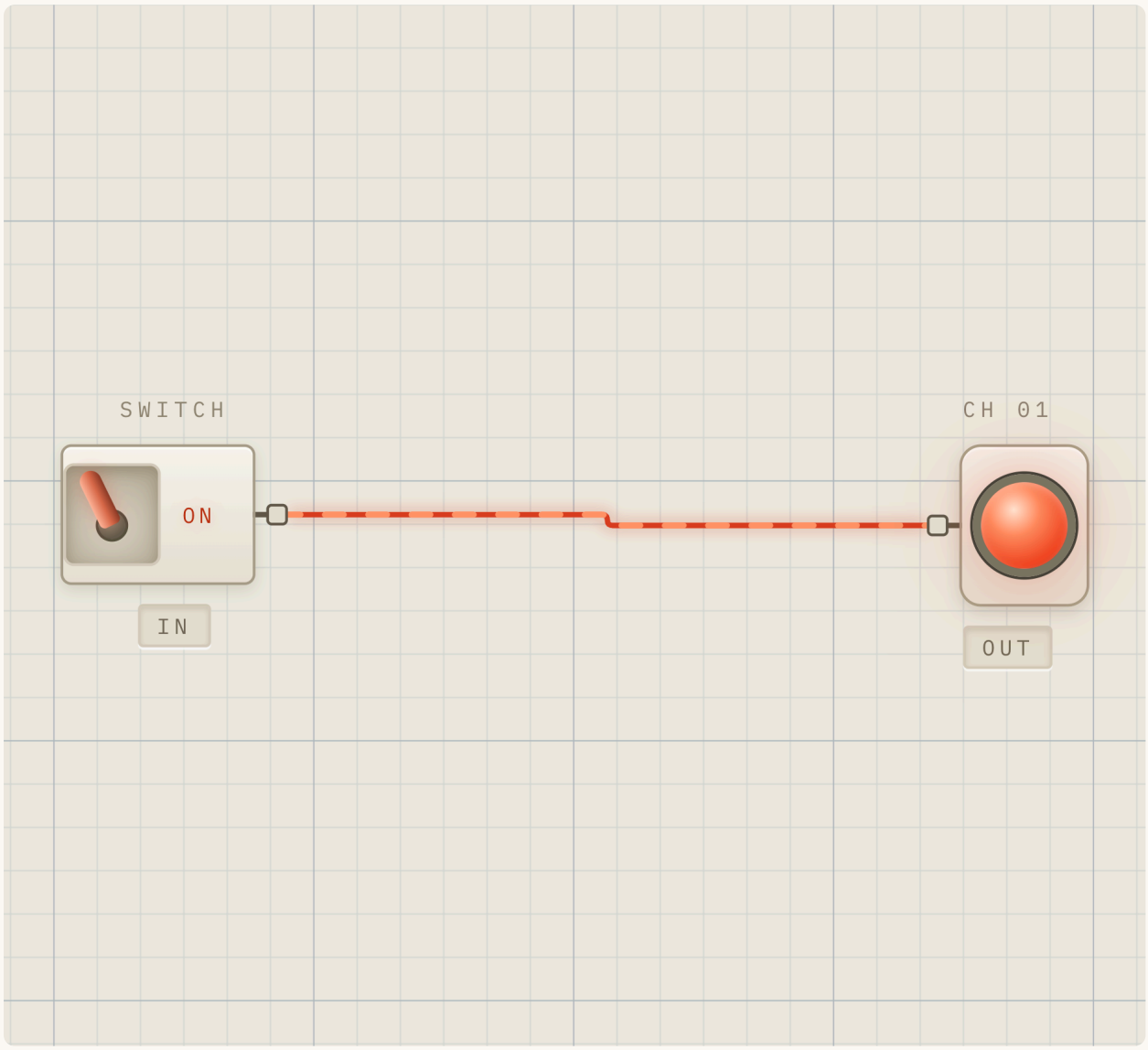
A **switch** is the simplest source of a signal. It has no inputs — nothing feeds it — and a single output. You click it, and it stays in whatever state you left it: **on** (emitting a 1, a lit wire) or **off** (emitting a 0, a dark wire). Click it again to flip it.

That word *stays* is the whole point of a switch. It is a toggle: it holds its position until you change it, exactly like the light switch on your wall. This makes it the natural way to feed a steady input into a circuit — to say "A is on, and it will stay on while I think about what happens next."

Almost every circuit in this book begins with a row of switches on the left. They are the questions you pose; the rest of the circuit is the answer.

See it in the bench

Open this: drag a **Switch** from the palette onto the canvas, and wire its output to an **LED**. That is the smallest complete circuit you can make: a thing that emits a signal, and a thing that shows it.



A switch wired to an LED, switch ON so the wire and LED are lit

Click the switch a few times. Each click flips it, and the wire and LED follow instantly. The switch is **holding** its state — it does not spring back.

Try it yourself

> **Try:** 1. Click the switch on, then take your hands off entirely. The LED stays lit — the switch holds. 2. Wire the *same* switch to a second LED as well. Both light together: one source can drive many destinations (this is called **fan-out**). 3. Leave the switch off and look at the dark wire. "Off" is not "disconnected" — the switch is actively emitting a 0. That distinction matters later.

Recap

- A switch is a **source**: no inputs, one output, no feeding required.
- It **holds** its state — on or off — until you click it again.
- It is how you pose steady inputs to a circuit, and where most circuits begin.

Check yourself: If you turn a switch on and walk away, what does its wire carry? *(A steady 1 — it holds until clicked again.)*

Next: Lesson 02 — The Button, a source that does not *hold.*

Lesson 02 — The Button

> **Part I · The Atoms** — native symbol Before this lesson: the Switch (Lesson 01). After this: the Clock (Lesson 03).

What you will learn

- How a button differs from a switch, and why that difference matters.
- What "momentary" means for a signal.
- Where you would reach for a button instead of a switch.

The idea

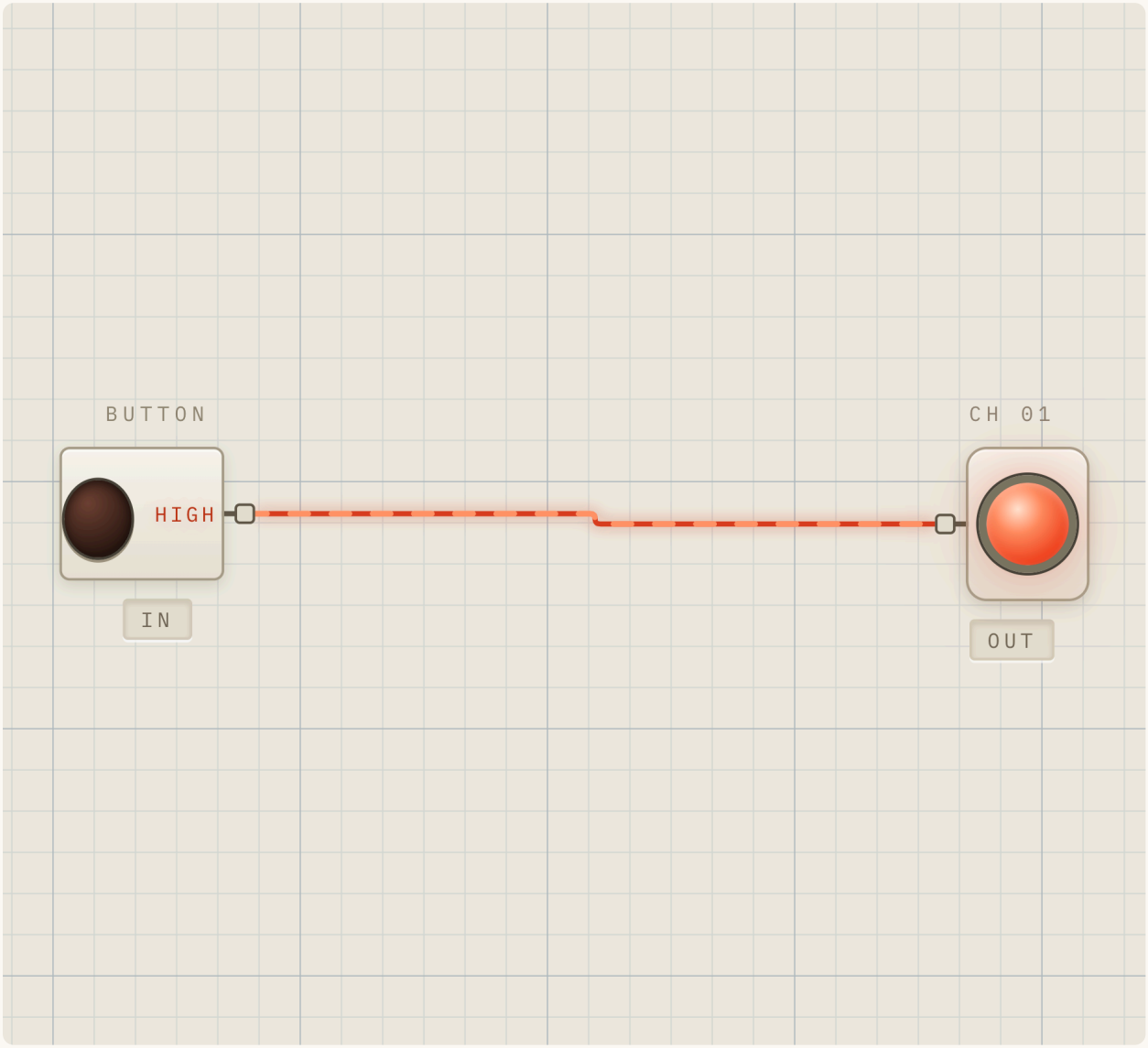
A **button** is a source just like a switch — no inputs, one output — but with one crucial difference: it does **not** hold. A button emits a 1 only *while you are pressing it*, and falls back to 0 the moment you let go. It is momentary.

The wall-switch versus doorbell comparison says it all. A switch is the light on your wall: flip it and it stays. A button is a doorbell: it rings while pressed and goes silent when released. Both put a signal into a circuit; they differ only in whether that signal *holds* or *springs back*.

Why have both? Because some jobs want a steady state ("keep the motor running") and some want a single nudge ("store this value *now*", "reset the counter"). A button is the natural way to deliver a clean one-shot pulse — a momentary "do it once" — without leaving the signal stuck on afterwards.

See it in the bench

Open this: drag a **Button** onto the canvas and wire it to an **LED**.



A button wired to an LED, lit only while pressed

Press and hold — the LED lights. Release — it goes dark at once. There is no flipping; the button always returns to 0 on its own.

Try it yourself

> **Try:** 1. Press and release a few times, watching the LED track your press exactly. 2. Think about the difference from Lesson 01: the switch was a *state* you set; the button is an *event* you trigger. 3. Keep this in mind for later — when you reach memory circuits and the RAM device, a button is exactly what delivers the brief "write it now" pulse (the **WE**, write-enable strobe) without leaving write mode stuck on.

Recap

- A button is a **momentary** source: 1 while pressed, 0 when released.
- A switch *holds*; a button *springs back*.
- Use a button for one-shot actions — store, reset, step — where you do not want the signal to linger.

Check yourself: You want to store a value into memory with a single clean pulse and not stay in write mode. Switch or button? *(Button.)*

Next: Lesson 03 — The Clock, a source that pulses by itself, forever.

Lesson 03 — The Clock

> **Part I · The Atoms** — native symbol Before this lesson: the Button (Lesson 02). After this: HIGH and LOW (Lesson 04).

What you will learn

- What a clock source does and why it is the heartbeat of every sequential machine.
- The meaning of a clock's **edge** — the instant it changes.
- How the clock's speed is something you can set and watch.

The idea

A **clock** is a source that flips itself on and off, over and over, with no help from you. Where a switch holds and a button springs back, a clock *pulses* — 1, 0, 1, 0 — at a steady rhythm forever. To the rest of the circuit it looks like a switch that an invisible hand is flipping at perfectly regular intervals.

This steady beat is the single most important idea in the second half of this book. Everything that *remembers* and *advances in steps* — counters, flip-flops, registers, the entire CPU — marches to a clock. The clock is what turns a pile of logic that merely reacts into a machine that *does one thing, then the next, then the next*. It is the heartbeat.

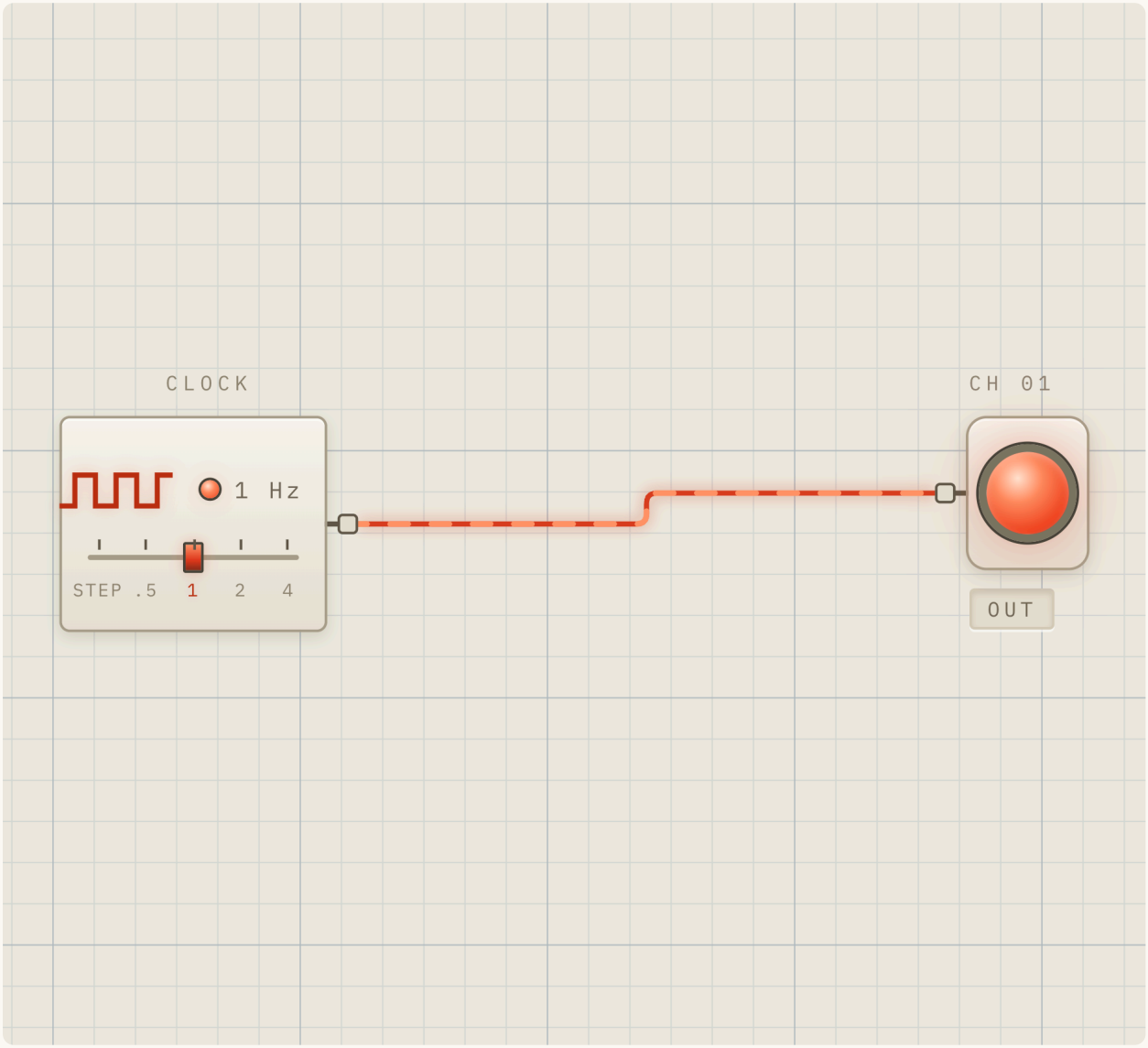
Two moments of a clock matter especially, and they have names:

- The **rising edge** — the instant the clock goes from 0 to 1.
- The **falling edge** — the instant it goes from 1 to 0.

Many circuits do not care whether the clock is currently high or low; they care about the *edge*, the moment of change. You will see this precisely when you meet flip-flops in Part V.

See it in the bench

Open this: drag a **Clock** onto the canvas and wire it to an **LED**. The LED begins blinking on its own — the clock is running.



A clock wired to an LED, mid-blink

The clock has a face you can click to change its **speed** (its frequency — how many beats per second). Slow it down and you can watch each individual tick; speed it up and the LED blurs into a fast flicker.

Try it yourself

> **Try:** 1. Watch the LED blink steadily with no input from you — the clock drives itself. 2. Click the clock face to change its speed. Notice the LED's rhythm change to match. 3. Wire the clock to two LEDs at once — they blink in perfect lockstep, because they share one heartbeat. This is exactly how every part of a CPU stays synchronised.

Recap

- A clock is a self-driving source that pulses 1-0-1-0 at a speed you can set.
- Its **edges** (the moments of change) are what sequential circuits react to.
- It is the heartbeat that lets circuits act in ordered steps — the foundation of everything in Parts V and IX.

Check yourself: What is the difference between a clock being "high" and a clock's "rising edge"? *(High is a state it holds for a while; the rising edge is the single instant it changes from 0 to 1.)*

Next: Lesson 04 — HIGH and LOW, sources that never change at all.

Lesson 04 — HIGH and LOW (constants)

> **Part I · The Atoms** — native symbols Before this lesson: the Clock (Lesson 03). After this: the LED (Lesson 05).

What you will learn

- What the constant sources HIGH and LOW are for.
- Why a circuit sometimes needs a wire that is *always* 1 or *always* 0.
- The difference between "tied low" and "left unconnected".

The idea

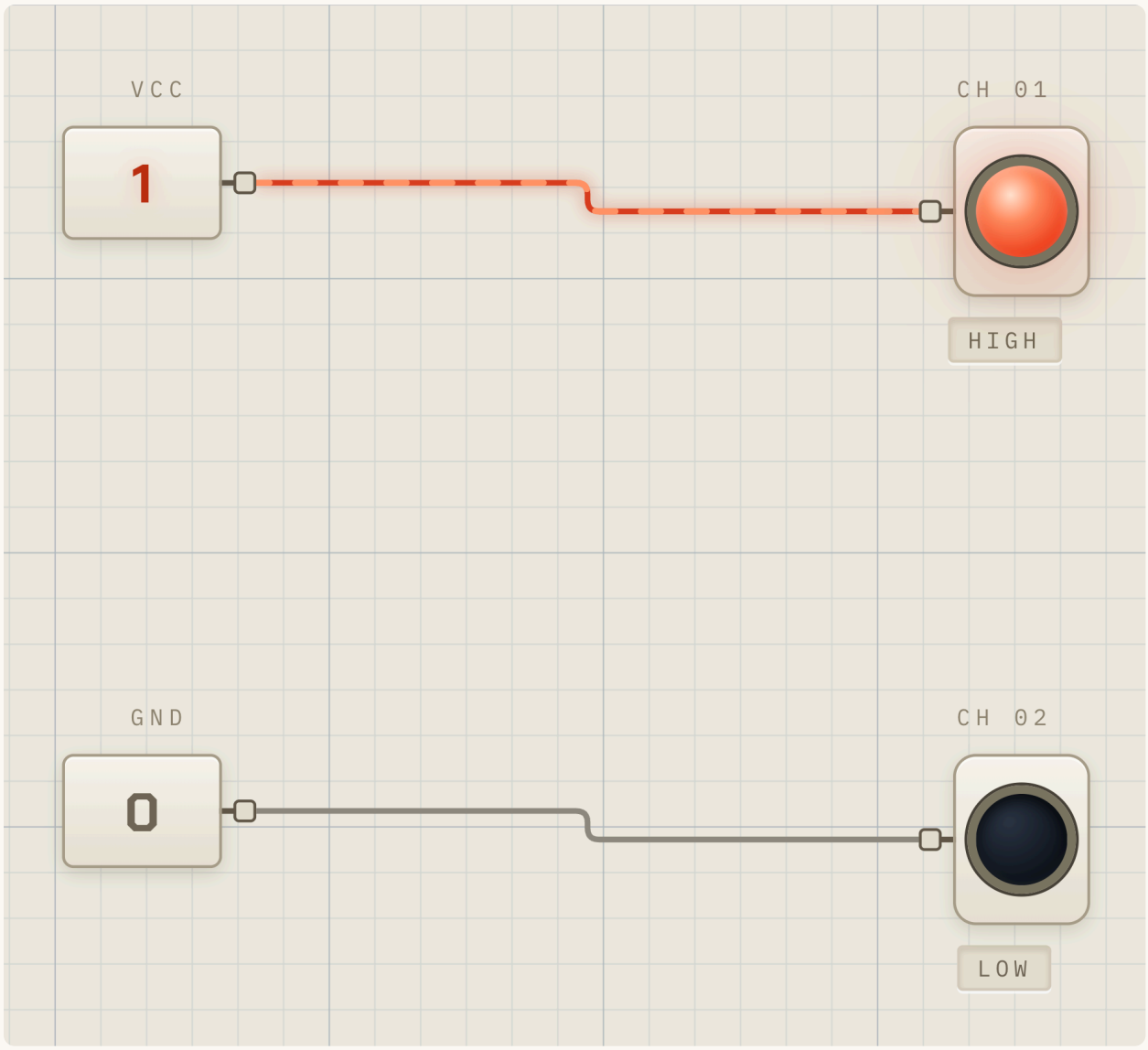
HIGH and **LOW** are the simplest sources of all. They have no inputs and no controls — you cannot click them. **HIGH** always emits 1. **LOW** always emits 0. That is everything they do.

Why would you want a signal that never changes? Because real circuits are full of inputs that need to be pinned to a fixed value. A gate might have an input that, for this particular design, should *always* be 1 — so you tie it to HIGH and forget about it. Another input might need to be held at 0 — tie it to LOW. Constants let you hardwire a decision into the circuit instead of dedicating a switch to something that will never move.

Engineers call this **tying** a line high or low, and it is everywhere in real hardware: enable pins held permanently on, unused inputs pinned to a safe value, carry-in lines fixed at 0 for the first stage of an adder.

See it in the bench

Open this: drag a **HIGH** source and a **LOW** source onto the canvas, and wire each to its own **LED**.



A HIGH constant lighting an LED and a LOW constant leaving one dark

The HIGH wire is permanently lit; the LOW wire is permanently dark. Neither responds to clicking — they are constants.

Watch out for

Pitfall — "tied low" is not the same as "left unconnected", even though both read 0. In this bench an input with no wire also reads as 0 (you will see why in Lesson 14). So why bother with a LOW source? Because tying a line to LOW is a *deliberate statement* — "this input is meant to be 0" — that you and anyone reading the circuit can see. An unconnected input is just as likely to be a forgotten wire. Use LOW when 0 is the intent, not the accident.

Try it yourself

Try: 1. Wire a HIGH source into one input of an AND gate and a switch into the other. Now the AND gate's output simply follows the switch — because "(always 1) AND B" is just "B". You have used a constant to simplify a gate. 2. Do the same with a LOW source into an OR gate: "(always 0) OR B" is also just "B". Constants are a quiet, powerful tool.

Recap

- **HIGH** always emits 1; **LOW** always emits 0; neither can be clicked.
- They let you hardwire a fixed decision into a circuit — "tying" a line high or low.
- Tying to LOW states intent; an unconnected input merely happens to read 0.

Check yourself: What does "(always 1) AND B" simplify to? *(Just B.)*

Next: Lesson 05 — The LED, the simplest way to see a signal.

Lesson 05 — The LED

> **Part I · The Atoms** — native symbol Before this lesson: HIGH and LOW (Lesson 04). After this: the 7-Segment Display (Lesson 06).

What you will learn

- What an LED does in the bench and why it is your primary way of *seeing* a signal.
- The idea of a **sink**: a component that receives but does not emit.
- How one LED reads exactly one bit.

The idea

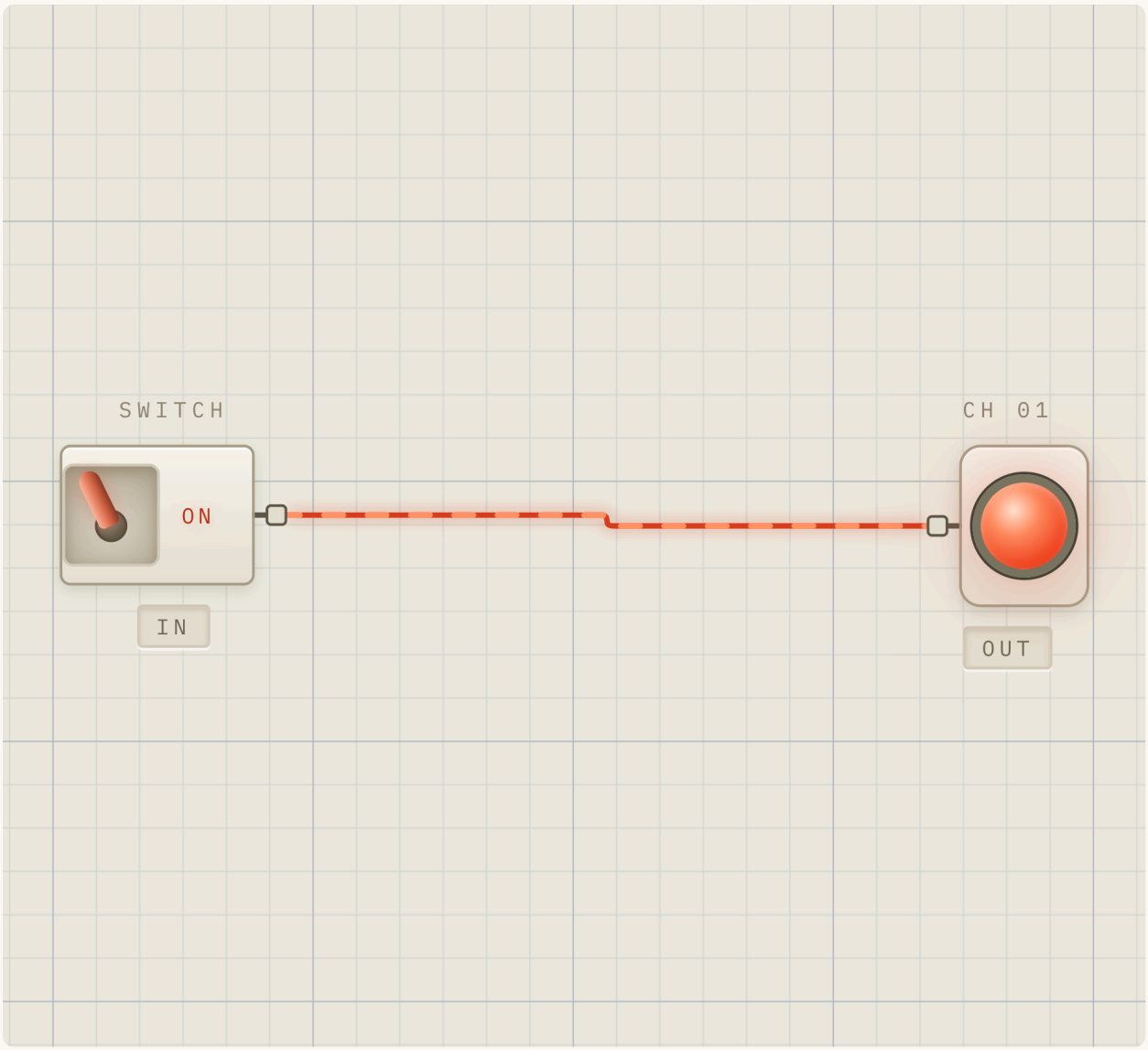
An **LED** is the opposite of a switch. A switch is a source — it emits a signal and has no input. An LED is a **sink** — it has one input and emits nothing onward. Its only job is to *show* you the value arriving on its wire: lit when the wire carries 1, dark when the wire carries 0.

That makes the LED your eyes. Logic happening on a wire is invisible until something displays it, and the LED is the simplest display there is — one lamp, one bit. Every time this book says "watch the output light up", it is an LED you are watching.

Because an LED is a sink, nothing flows out of it. You wire things *into* an LED, never *out of* it. It is the end of a signal's journey.

See it in the bench

Open this: wire a **Switch** to an **LED** (you did exactly this in Lesson 01). Flip the switch and watch the LED echo it.



A switch driving an LED, both shown on and off

The LED reports, faithfully and instantly, the single bit on its input wire. One LED, one bit — that is the unit of seeing in this bench.

Try it yourself

> **Try:** 1. Drive one LED from a clock (Lesson 03) and watch it blink — the LED is showing you the clock's bit changing over time. 2. Wire one switch to four separate LEDs. All four mirror the same bit. To show a *number* rather than a single bit, you need several wires and a smarter display — which is the next lesson.

Recap

- An LED is a **sink**: one input, no output; it only displays.
- Lit = 1, dark = 0. One LED shows exactly one bit.
- It is your fundamental instrument for seeing what a wire is doing.

Check yourself: Can you wire a signal *out* of an LED into another gate? *(No — an LED is a sink; signals only flow in.)*

Next: Lesson 06 — The 7-Segment Display, which reads four bits at once as a hex digit.

Lesson 06 — The 7-Segment Display

> **Part I · The Atoms** — native symbol Before this lesson: the LED (Lesson 05). After this: the Binary Clock instrument (Lesson 07).

What you will learn

- How a 7-segment display turns four bits into a single readable hex digit.
- The binary weights 1, 2, 4, 8 and how they add up.
- Why engineers read four bits as one hex character.

The idea

One LED shows one bit. But circuits quickly deal in groups of four bits at a time (a group of four bits is called a **nibble**), and reading four separate lamps as a number gets tiring. The **7-segment display** solves this: it takes **four input bits** and shows them as a single **hexadecimal digit** — one of `0 1 2 3 4 5 6 7 8 9 A B C D E F`.

The four inputs are not equal — each carries a **weight**:

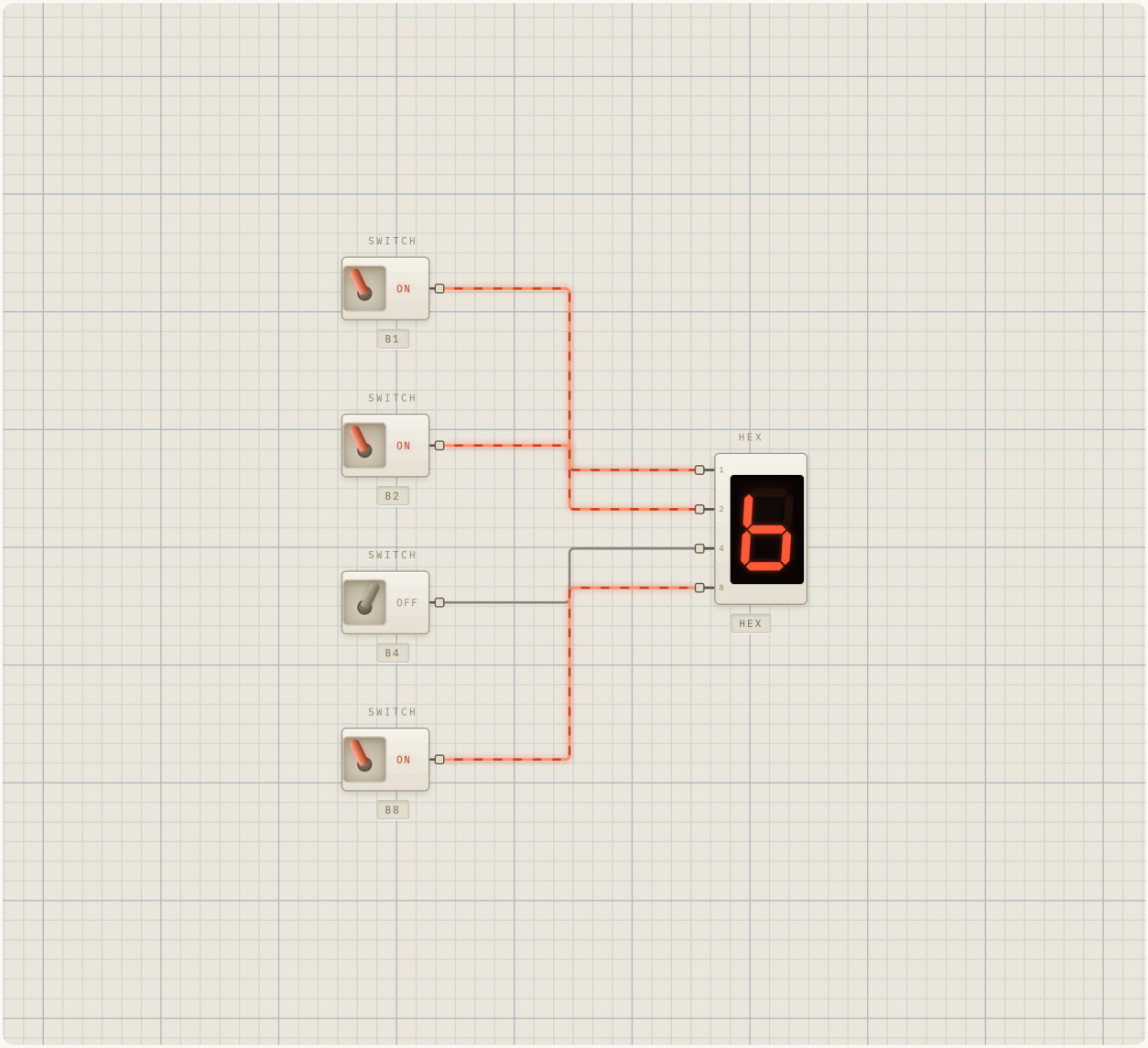
- input bit 1 is worth **1**
- input bit 2 is worth **2**
- input bit 3 is worth **4**
- input bit 4 is worth **8**

The display simply **adds the weights of the inputs that are on** and shows the total as a hex digit. So if the 8-input and the 2-input and the 1-input are high, the display adds 8 + 2 + 1 = 11, and shows **b** (hex for 11). All four on gives 8 + 4 + 2 + 1 = 15 = **F**.

This is exactly how engineers read binary in the real world: four bits at a glance, as one compact hex character, instead of squinting at four lamps.

See it in the bench

Open this: load **Hex Display Demo** from the library (category **ROUTING & CODES**). It wires four switches — labelled by weight, B1 B2 B4 B8 — into one 7-segment display. (We will revisit this circuit properly in Lesson 40; here, just meet the display itself.)



Four weighted switches driving a 7-segment display reading b

Try it yourself

Predict first, then flip.

Try: 1. Turn on **B8, B2, B1** (leave B4 off). Predict the digit, then look: 8 + 2 + 1 = 11 → **b**. 2. Now add **B4** as well — all four on. Predict: 8 + 4 + 2 + 1 = 15 → **F**. 3. Turn everything off, then count upward — B1 alone (1), B2 alone (2), B1+B2 (3), B4 alone (4) — and watch the digit follow. You are reading binary as hex in real time.

Recap

- A 7-segment display reads **four weighted bits** (1, 2, 4, 8) and shows their sum as one **hex digit** 0–F.
- It is the compact way to read a nibble at a glance.
- Binary 1011 = 8 + 2 + 1 = 11 = hex **b**.

Check yourself: Which inputs are on when the display reads **F**? *(All four — 8 + 4 + 2 + 1 = 15.)*

Next: Lesson 07 — The Binary Clock instrument, a self-contained wall clock you read in binary.

Lesson 07 — The Binary Clock instrument

> **Part I · The Atoms** — native symbol Before this lesson: the 7-Segment Display (Lesson 06). After this: the NOT gate (Lesson 08).

What you will learn

- What the Binary Clock instrument is, and how it differs from every other component.
- How to read the current time in binary-coded decimal (BCD).
- The difference between an *instrument* that shows real time and a *circuit* that computes it.

The idea

The **Binary Clock** is unusual: it is a self-contained instrument with **no inputs and no outputs**. You cannot wire anything to it. It simply hangs on the canvas like a wall clock and shows the **real current time** — the actual time of day — in binary.

It displays hours, minutes and seconds as **HH:MM:SS**, but each digit is shown as a column of lamps rather than a printed numeral. You read each column **bottom-up** using the familiar weights **1, 2, 4, 8**, and add the lit lamps to get that decimal digit. This way of showing a decimal number with binary lamps per digit is called **binary-coded decimal**, or BCD.

For example, at **10:37:42** the minute columns read 3 and 7: the "7" column has its 1-, 2- and 4-lamps lit (1 + 2 + 4 = 7). A small selector underneath switches the timezone.

The instrument is a bit of fun — but it also sets up a profound point you will reach in Lesson 34. This lamp display shows the time; it does *not* explain how

a circuit could *generate* the time. Later you will load a **full gate-level circuit** that produces this very same readout from flip-flops and a 1 Hz clock. The instrument is the *what*; the circuit is the *how*.

See it in the bench

> **Open this:** load **Binary Clock (BCD)** from the library (category **ROUTING & CODES**).

The Binary Clock instrument showing the current time in BCD lamp columns

Try it yourself

> **Try:** 1. Read the seconds columns and watch the bottom lamp (weight 1) flip every second — that is real time passing. 2. Add up the lit lamps in each column and check the time against your computer's own clock. They match. 3. Note that nothing is wired to this instrument. Then make yourself a promise: by Lesson 34 you will load the *circuit* that produces this exact display, built entirely from gates and flip-flops you will understand.

Recap

- The Binary Clock is a self-contained **instrument**: no inputs, no outputs, shows real wall time.
- Read each column bottom-up by weights 1-2-4-8 and add the lit lamps (binary-coded decimal).
- It shows the time but does not explain it — Lesson 34 builds the gate-level circuit that does.

Check yourself: A column has its 1-lamp and 4-lamp lit. What digit is it? *(5.)*

Next: Lesson 08 — The NOT gate — our first gate, and the start of real logic.

Lesson 08 — The NOT gate (inverter)

> **Part I · The Atoms** — native symbol Before this lesson: the Binary Clock (Lesson 07). After this: the AND gate (Lesson 09).

What you will learn

- What the NOT gate does — the simplest gate of all.
- Why "invert" is the most basic computation a circuit can perform.
- How a single gate with one input still counts as logic.

The idea

The **NOT gate**, also called an **inverter**, is the only gate with a single input. Its rule is as simple as logic gets: it outputs the **opposite** of its input. Feed it a 1 and it emits 0; feed it a 0 and it emits 1. It flips the bit.

That is genuinely all it does — and yet it is indispensable. Inversion is how a circuit says *not*: "fire when the door is **not** closed", "select channel A when the select line is **not** high". You will see in a moment (Lesson 09 onward) that almost every interesting circuit needs to invert some signal somewhere. The NOT gate is the little word *not* made out of wire.

Its symbol is a triangle pointing in the direction of signal flow, with a small circle (a "bubble") on its nose. That bubble is the universal mark of inversion — keep an eye out for it, because it reappears on the NAND and NOR gates, where it means exactly the same thing: "and then flip the result."

The truth table

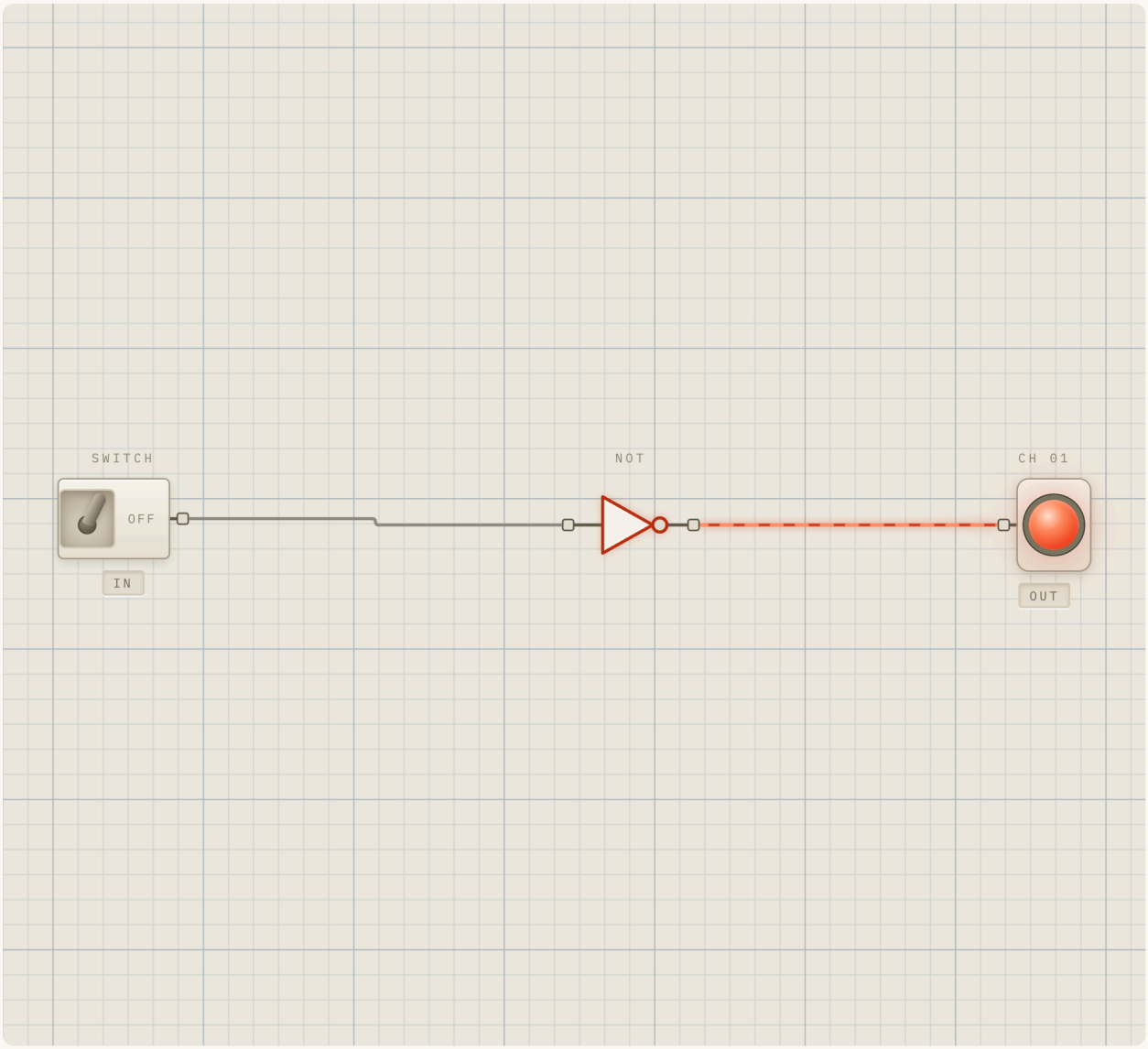
With one input there are only two rows:

A	NOT A
0	1
1	0

Every row is simply the opposite of its input. That is the entire behaviour.

See it in the bench

> **Open this:** drag a **NOT** gate onto the canvas, wire a **Switch** into its input, and wire its output to an **LED**.



A NOT gate inverting its input: switch off, LED on

Notice the surprise: with the switch **off**, the LED is **on**. The gate has inverted the dark input into a lit output. Flip the switch on, and the LED goes dark. The output always contradicts the input.

Try it yourself

Predict first, then flip.

Try: 1. Switch **off** → predict the LED. It is **lit** (0 inverts to 1). This catches everyone the first time. 2. Switch **on** → predict. Now **dark** (1 inverts to 0). 3. Wire a *second* NOT gate after the first. Two inversions cancel: the final LED now matches the switch again. "NOT NOT A" is just "A".

Recap

- A NOT gate (inverter) has one input and outputs its **opposite**.
- It is the hardware word for *not*, and the bubble on its nose marks inversion.
- Two NOTs in a row cancel out.

Check yourself: What does a NOT gate output when its input is 0? And what is "NOT NOT A"? *(1; and just A.)*

Next: Lesson 09 — The AND gate, the gate of "both".

Lesson 09 — The AND gate

> **Part I · The Atoms** — native symbol Before this lesson: the NOT gate (Lesson 08). After this lesson: the OR gate (Lesson 10).

What you will learn

- What an AND gate does, in one sentence and in a full truth table.
- How to read its behaviour off the lit and dark wires in the bench.
- Why "AND" is the hardware word for *both / all*.
- How to wire one yourself and prove its truth table by hand.

The idea

An **AND gate** has two inputs and one output. Its output is **1 only when both inputs are 1**. If either input is 0 — or both are — the output is 0.

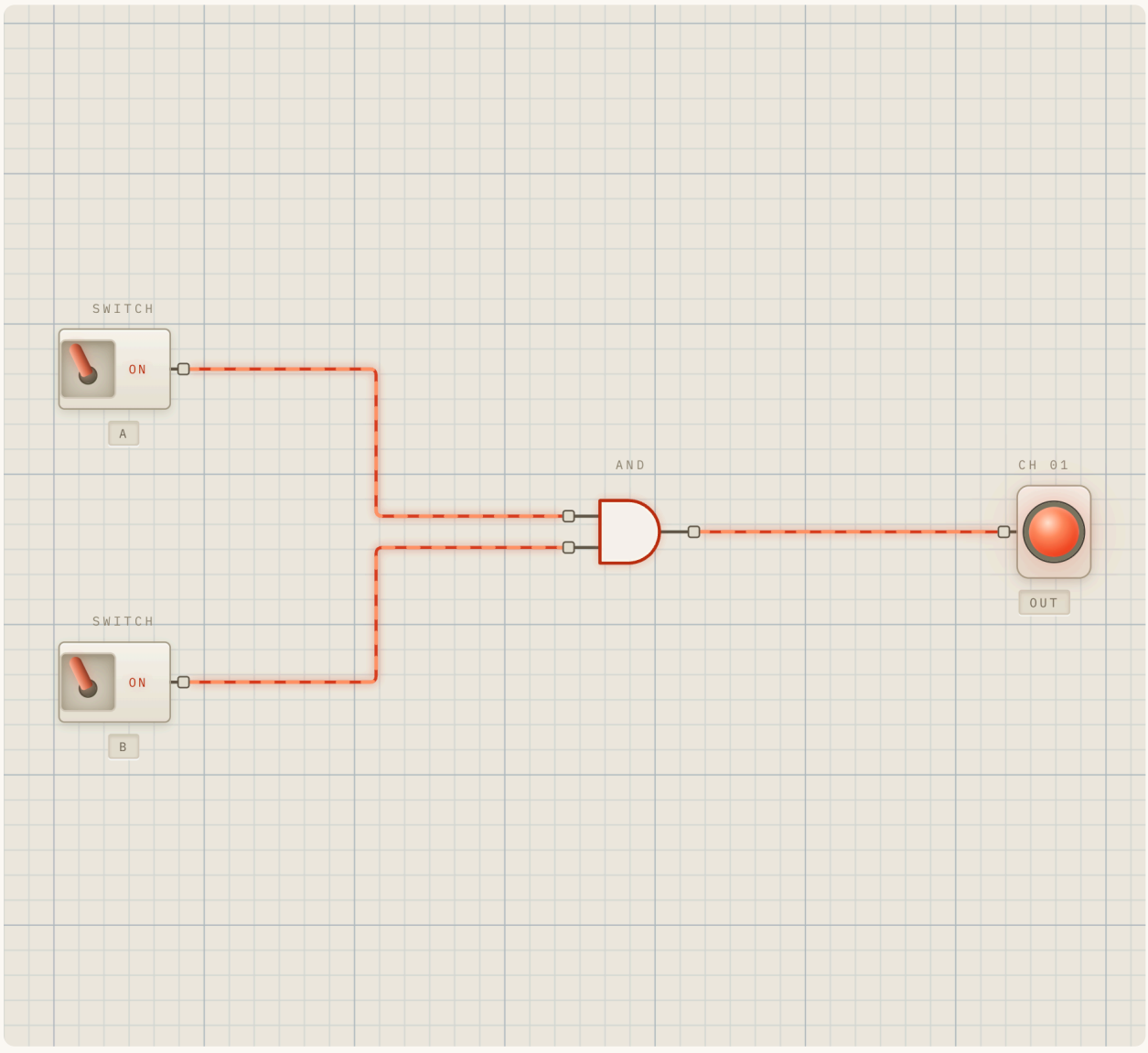
That is the whole rule. In plain language it is the gate of *both*: "light the output **if A and B are both on**." It is how a circuit asks a question with the word *and* in it — "is the door closed **and** the key turned?"

The symbol in the bench is the classic AND shape: a flat back where the two input wires arrive on the left, and a smooth half-round nose pushing the single output wire out to the right.

See it in the bench

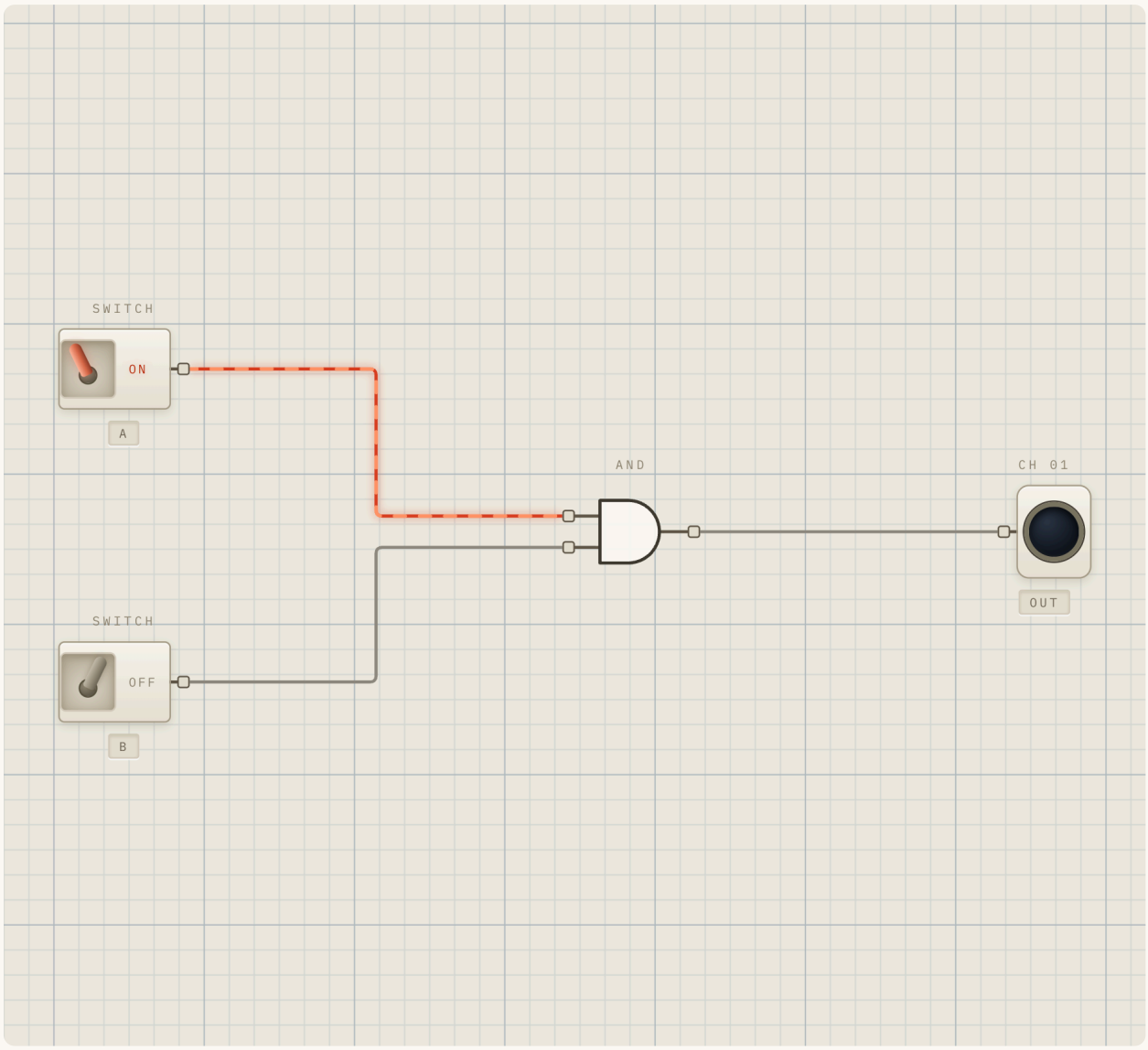
Open this: drag an **AND** gate onto the canvas from the palette (the *Gates* group). Wire a **Switch** to each of its two inputs, and wire its output to an **LED**. You now have a complete, testable AND gate.

If you would rather not wire it from scratch yet, every circuit in Part III uses AND gates you can watch — but building this one yourself takes thirty seconds and teaches more.



The AND gate with both switches ON and the output LED lit

When both switches are on, both input wires glow, and the output wire glows too — the LED is lit. Now switch **one** of them off:



The AND gate with one switch OFF — the output goes dark

The moment either input goes dark, the output goes dark. The gate is unforgiving: it wants **both**.

The truth table

A truth table lists every possible combination of inputs and the output each produces. With two inputs there are exactly four rows:

A	B	A AND B
0	0	**0**
0	1	**0**
1	0	**0**
1	1	**1**

Notice the shape of it: a single lonely **1** at the very bottom. Out of four possible input combinations, only one — both inputs high — lights the output. This rarity is exactly what makes AND useful: it fires *only* in one specific situation, so it is the gate you reach for when you need several conditions to all be true at once.

Try it yourself

> **Predict first, then flip.** For each step, say out loud what the LED will do *before* you touch the switch.
>
> **Try:** 1. Start with both switches **off**. Predict the LED, then look. *(Both inputs 0 → output 0.)* 2. Turn **A** on, leave **B** off. Predict, then look. Did turning one input on do anything? 3. Now turn **B** on as well. The instant the second one comes on, the LED should light. 4. Turn **A** back off. The LED dies immediately — one missing input is enough to break the "both".

You have now walked all four rows of the truth table by hand. You *are* the test bench.

A small experiment that teaches a big idea

Try: Add a **third** switch and a second AND gate. Wire the first AND's output and your new third switch into the second AND. Now the LED lights only when **all three** original conditions are true.

This is your first taste of **composition** — chaining simple gates to ask bigger questions. A 3-input AND is just two 2-input ANDs in a row. Real chips with 8-input gates are built exactly this way.

Watch out for

Pitfall — an unconnected input is not "neutral". If you leave one of the AND gate's inputs with no wire at all, the bench treats it as **0** (LOW), not as "ignore me". So an AND gate with one input wired high and the other left dangling will read **0** forever — it is waiting for a "both" that can never arrive. If a gate seems stuck dark, check that *every* input is actually wired.

Recap

- An AND gate outputs **1 only when all its inputs are 1**.
- Its truth table has a single 1, in the both-high row.
- It is the gate of *both / all* — use it when several things must be true together.
- Unconnected inputs read as 0, which can silently hold an AND gate dark.

Check yourself: Without looking at the table — if A is 1 and B is 0, what is the output? And what is the *only* input combination that lights an AND gate? *(Answers: 0; and only A=1, B=1.)*

Next: Lesson 10 — The OR gate, the gate of "either".

Lesson 10 — The OR gate

> **Part I · The Atoms** — native symbol Before this lesson: the AND gate (Lesson 09). After this: the NAND gate (Lesson 11).

What you will learn

- What an OR gate does and how it complements the AND gate.
- Why "OR" is the hardware word for *either / any*.
- The one row of its truth table that surprises people.

The idea

An **OR gate** has two inputs and one output. Its output is **1 when at least one input is 1** — either A, or B, or both. The only way to get 0 out of an OR gate is for *both* inputs to be 0.

If AND is the gate of *both*, OR is the gate of *either*. It is how a circuit asks a question with the word *or* in it: "sound the alarm if the front door **or** the back door is open." Any one condition being true is enough.

The OR symbol is a curved shield shape: a concave back where the inputs arrive and a pointed nose where the single output leaves. It is worth contrasting it with the AND shape (flat back, round nose) so you can tell them apart at a glance on a busy canvas.

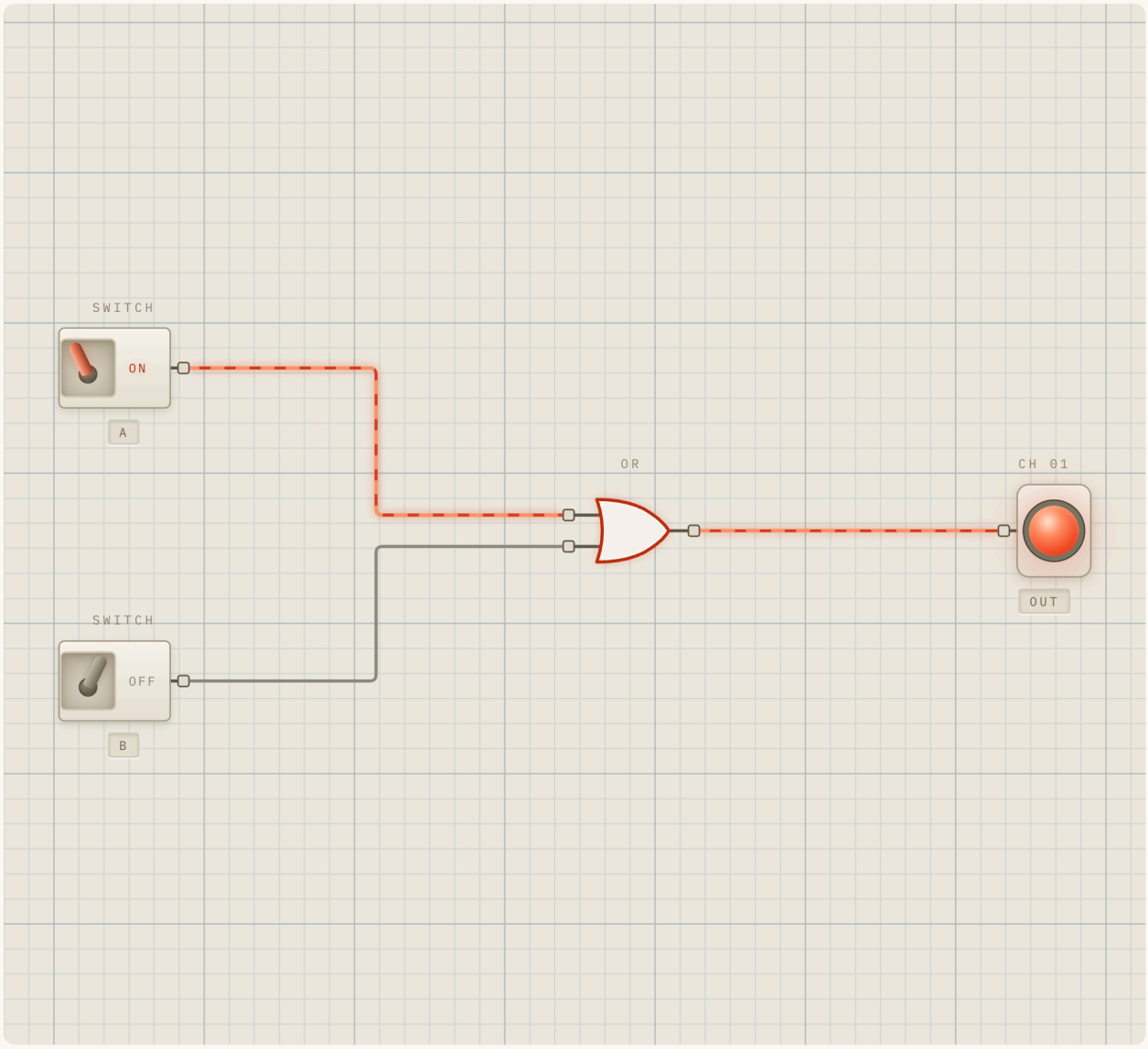
The truth table

A	B	A OR B
0	0	**0**
0	1	**1**
1	0	**1**
1	1	**1**

This is the mirror image of the AND table: AND had a single 1 at the bottom, OR has a single **0** at the top. Three of the four combinations light the output. OR is generous — it fires unless *everything* is off.

See it in the bench

> **Open this:** drag an **OR** gate onto the canvas. Wire a **Switch** to each input and the output to an **LED**.



An OR gate with one input on, output lit

Turn on just one switch — the LED lights. That is the heart of OR: one is enough.

Try it yourself

Predict first, then flip.

Try: 1. Both switches **off** → predict. The *only* dark-output case: LED off. 2. Turn on **just A** → LED lights. Now turn A off and turn on **just B** → still lit. Either alone works. 3. Turn on **both** → still lit. OR does not mind "too much"; it minds "nothing".

The surprising row: people expect "1 OR 1" to somehow be different from "1 OR 0". It is not — both give 1. OR asks only "is *anything* on?"

Recap

- An OR gate outputs **1 when any input is 1**; it outputs 0 only when all inputs are 0.
- It is the gate of *either / any* — the mirror of AND.
- Its truth table is a single 0 (all-off) with 1s everywhere else.

Check yourself: What is the only input combination that gives an OR gate a 0 output? *(Both inputs 0.)*

Next: Lesson 11 — The NAND gate, the "not-and" gate that can build anything.

Lesson 11 — The NAND gate

> **Part I · The Atoms** — native symbol Before this lesson: the OR gate (Lesson 10). After this: the NOR gate (Lesson 12).

What you will learn

- What a NAND gate does — and why it is secretly the most important gate of all.
- How "NAND" is just "AND, then NOT".
- A first hint of the universality you will prove in Part II.

The idea

A **NAND gate** is an AND gate with an inverter stuck on its output. The name is literally **N**ot-**AND**. So its rule is: take AND of the two inputs, then flip the result. Output is **0 only when both inputs are 1**; in every other case the output is **1**.

Compare it directly to AND. The AND gate gave a single 1 (in the both-high row). NAND gives the exact opposite everywhere: a single **0** in the both-high row, 1s elsewhere. The bubble on the NAND's nose — the same inversion bubble you met on the NOT gate — is the visual sign of that final flip.

Here is why NAND deserves special attention: it is **universal**. Using nothing but NAND gates, you can build a NOT, an AND, an OR, an XOR — *every* gate, and therefore every circuit in this entire book, your CPU included. Part II proves this with circuits you can load. For now, just register the claim: this humble "not-and" gate is a complete toolkit on its own. Real chip fabs love it for exactly this reason.

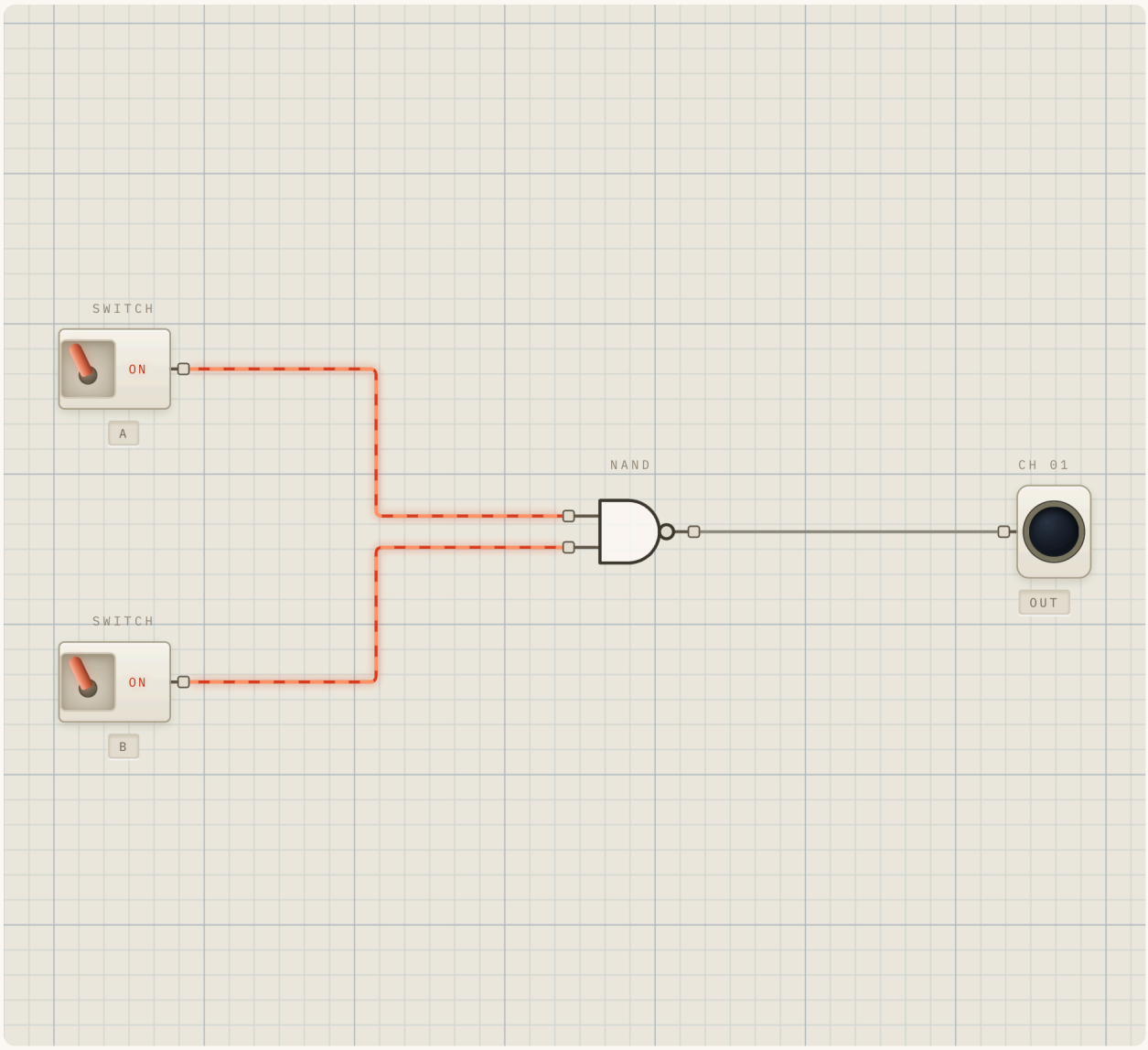
The truth table

A	B	A NAND B
0	0	**1**
0	1	**1**
1	0	**1**
1	1	**0**

Read it as "the AND table, upside down": wherever AND was 0, NAND is 1, and the lone both-high row that AND lit is the one row NAND leaves dark.

See it in the bench

> **Open this:** drag a **NAND** gate onto the canvas, wire a **Switch** to each input, and the output to an **LED**.



A NAND gate with both inputs on, output dark

The striking case is **both inputs on**: unlike AND, the output goes **dark**. Every other combination lights it.

Try it yourself

Predict first, then flip.

Try: 1. Both **off** → LED lit (NAND of 0,0 is 1). Already the opposite of AND. 2. One on, one off → still lit. 3. **Both on** → LED goes dark. The single 0 of the whole table. 4. Tie both inputs to the *same* switch. Now the NAND behaves as a **NOT** gate — when the switch is on, both inputs are 1, output 0; when off, output 1. You have just built an inverter out of a NAND. That is universality starting to show itself.

Recap

- NAND = AND then NOT: output is **0 only when both inputs are 1**, else 1.
- It is the upside-down AND table.
- NAND is **universal** — every other gate can be built from it (proved in Part II). Feeding it one shared input already gives you a NOT.

Check yourself: What is the only input combination that makes a NAND output 0? *(Both inputs 1.)*

Next: Lesson 12 — The NOR gate, the other universal gate.

Lesson 12 — The NOR gate

> **Part I · The Atoms** — native symbol Before this lesson: the NAND gate (Lesson 11). After this: the XOR gate (Lesson 13).

What you will learn

- What a NOR gate does — "OR, then NOT".
- That NOR, like NAND, is universal.
- Why NOR matters for memory, foreshadowing the SR latch.

The idea

A **NOR gate** is an OR gate with an inverter on its output: **N**ot-**OR**. Take OR of the two inputs, then flip it. The result: output is **1 only when both inputs are 0**; any input being 1 forces the output to 0.

It is the mirror of OR exactly as NAND mirrors AND. OR gave a single 0 (all-off); NOR gives a single **1** (all-off), with 0s everywhere a 1 is present. The familiar inversion bubble sits on its nose.

NOR is the **second universal gate**: like NAND, it alone can build every other gate and therefore any circuit. There is a famous proof point — the Apollo Guidance Computer that flew astronauts to the Moon was built almost entirely out of NOR gates. You will recreate the full set of gates from NOR in Lesson 16.

One more reason to pay attention: NOR has a special role in **memory**. When you cross-couple two NOR gates — feed each one's output back into the other — you get a circuit that *remembers* a bit. That is the SR latch of Lesson 26, the first time a circuit gains a past. Keep the NOR gate in mind; you will meet it again at that turning point.

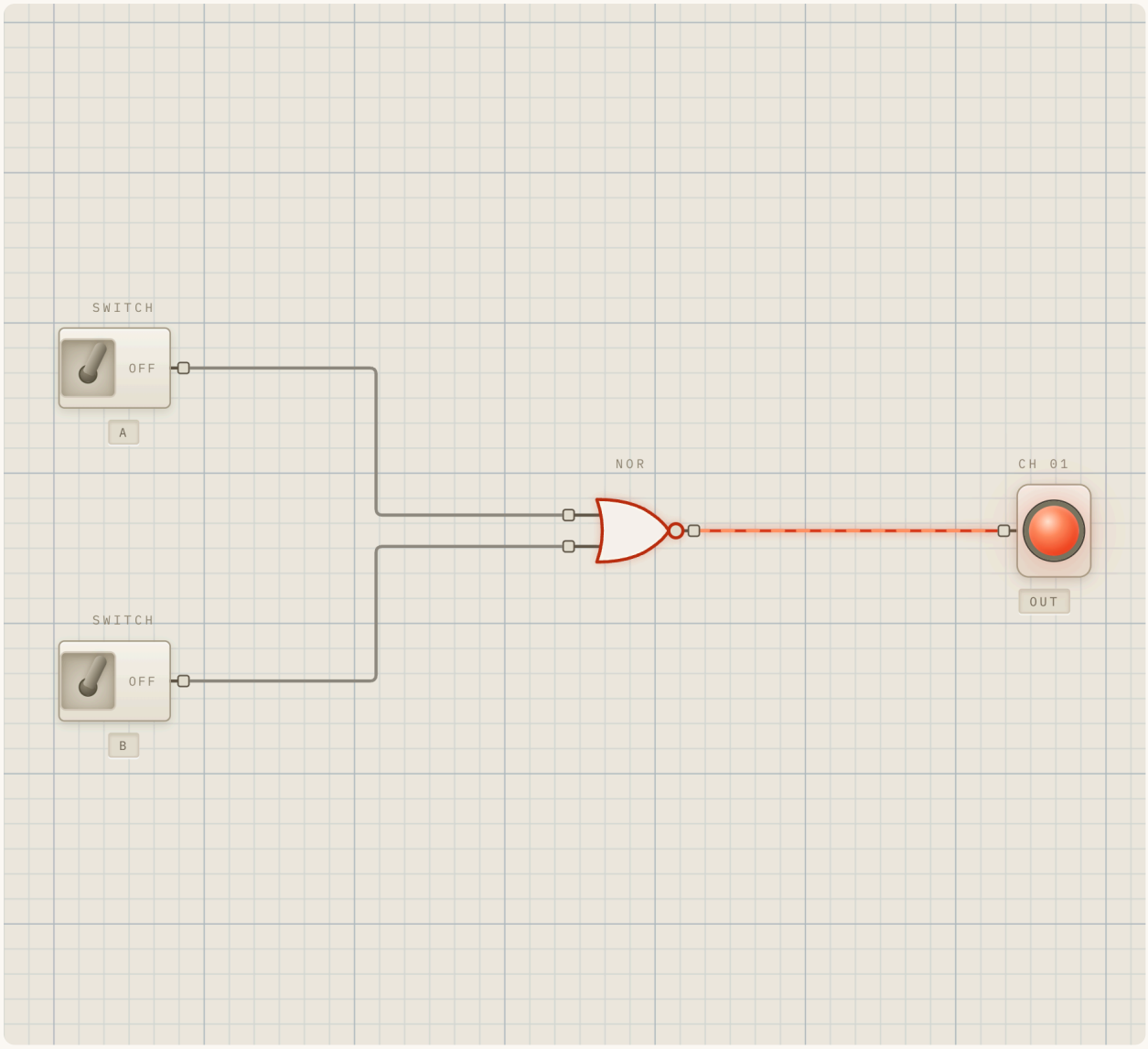
The truth table

A	B	A NOR B
0	0	**1**
0	1	**0**
1	0	**0**
1	1	**0**

The OR table, flipped: a single 1 in the all-off row, 0s elsewhere.

See it in the bench

> **Open this:** drag a **NOR** gate onto the canvas. Wire a **Switch** to each input and the output to an **LED**.



A NOR gate with both inputs off, output lit

With **both switches off**, the LED lights — the single 1 of NOR. Turn on either switch and it goes dark.

Try it yourself

Predict first, then flip.

Try: 1. Both **off** → LED lit (the only on-case). Opposite of OR. 2. Turn on **either** input → LED dark. Any 1 kills the output. 3. Tie both inputs to one switch — the NOR now acts as a **NOT** gate, just like the NAND trick. Another glimpse of universality.

Recap

- NOR = OR then NOT: output is **1 only when both inputs are 0**, else 0.
- It is the upside-down OR table, and the **second universal gate**.
- Cross-coupled NOR gates remember a bit — the basis of the SR latch (Lesson 26).

Check yourself: What is the only input combination that makes a NOR output 1? *(Both inputs 0.)*

Next: Lesson 13 — The XOR gate, the gate of "different".

Lesson 13 — The XOR gate

> **Part I · The Atoms** — native symbol Before this lesson: the NOR gate (Lesson 12). After this: how the bench thinks (Lesson 14).

What you will learn

- What an XOR gate does — the gate of *difference*.
- Why XOR is the secret heart of binary addition.
- How it differs from a plain OR.

The idea

An **XOR gate** — "exclusive OR" — outputs **1 when its two inputs are different**, and 0 when they are the same. Different → 1. Same → 0. That is the whole rule, and it is worth saying out loud because it is unlike the others: XOR cares not about *how many* inputs are on, but about whether they *disagree*.

Contrast it with OR. Plain OR lights up for 0-1, 1-0, *and* 1-1. XOR lights for 0-1 and 1-0 but goes **dark** for 1-1 — the "exclusive" part means "one or the other, but **not both**." When the inputs agree (both 0 or both 1), XOR outputs 0.

Why does this strange "are they different?" gate matter so much? Because it *is* binary addition of one bit. Think: 0+0 = 0, 0+1 = 1, 1+0 = 1, 1+1 = 0-with-a-carry. Ignore the carry for a moment and look at the bottom bit of each: 0, 1, 1, 0 — that is precisely the XOR table. This is why XOR is the **SUM** half of the half adder you will build in Lesson 18, and why it appears in every adder, subtractor and comparator in Part III. XOR is where logic learns to count.

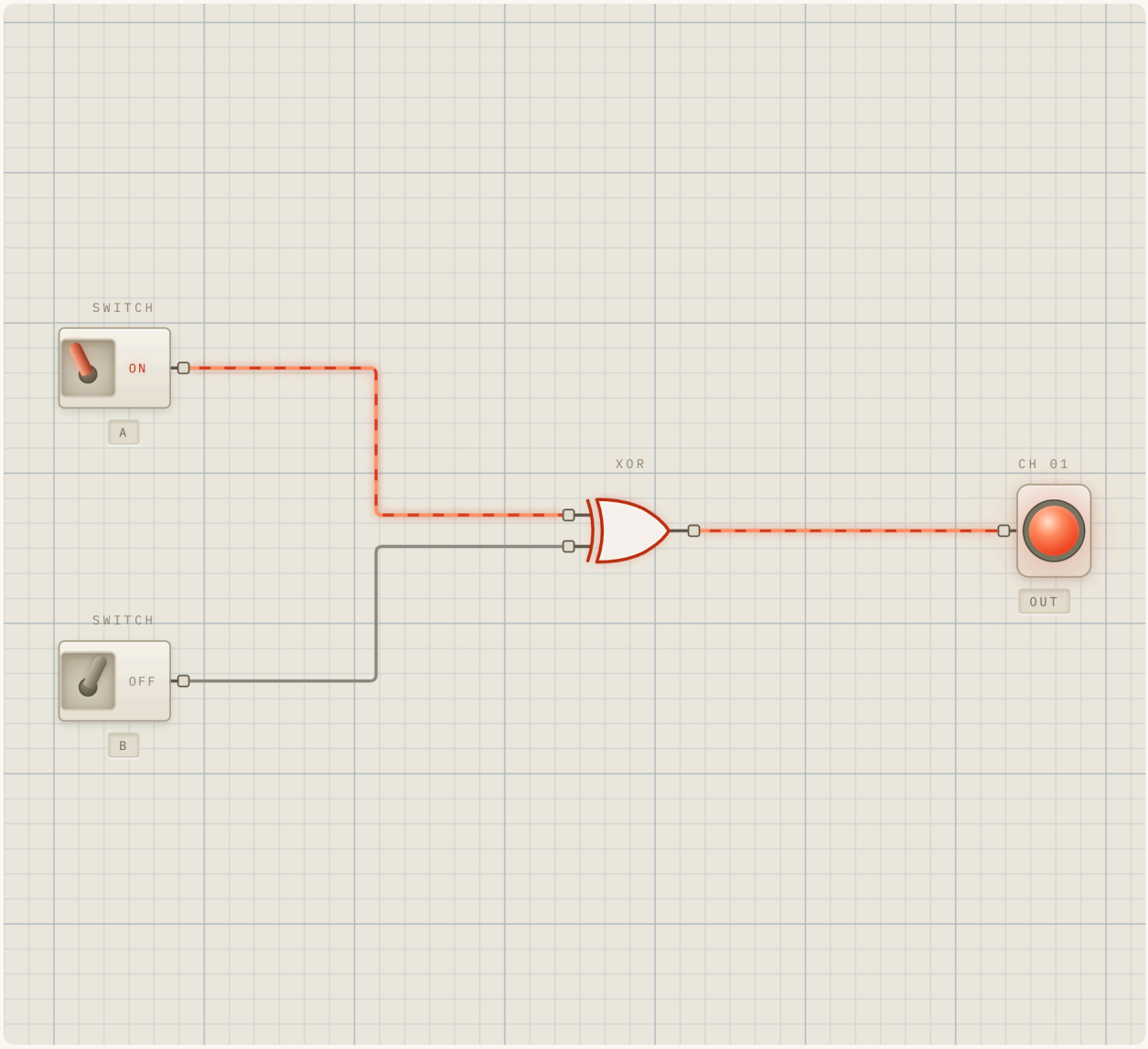
The truth table

A	B	A XOR B
0	0	**0**
0	1	**1**
1	0	**1**
1	1	**0**

The pattern is "1 when they differ." Note the 1-1 row is **0** — that is the difference from OR.

See it in the bench

> **Open this:** drag an **XOR** gate onto the canvas. Wire a **Switch** to each input and the output to an **LED**.



An XOR gate with inputs differing, output lit

Set the two switches to **different** states — one on, one off — and the LED lights. Now make them **the same** (both on, or both off) — the LED goes dark.

Try it yourself

Predict first, then flip.

Try: 1. Both **off** (same) → LED dark. 2. **A on, B off** (different) → LED lit. 3. **Both on** (same again) → LED dark. This is the row that separates XOR from OR — predict it before you flip, and feel the difference. 4. Hold one input on and toggle the other repeatedly: XOR turns that input into a *controlled inverter* — when the held input is 1, the output is the opposite of the other input. (That trick is exactly how the ALU in Lesson 47 flips bits to do subtraction.)

Recap

- An XOR gate outputs **1 when its inputs differ**, 0 when they match.
- The 1-1 row is 0 — the "exclusive" that separates it from OR.
- XOR is one-bit addition (the SUM bit), and a controlled inverter — both crucial in Part III and the CPU.

Check yourself: What does XOR output when both inputs are 1? *(0 — they are the same.)*

Next: Lesson 14 — How the bench thinks: the rules that govern every circuit you have built so far.

Lesson 14 — How the bench thinks: wires, fan-in, and settling

> **Part I · The Atoms** — the engine's rules Before this lesson: all the gates (Lessons 08–13). After this: Gates from NAND (Lesson 15).

What you will learn

- The three rules the simulator uses to decide what every wire carries.
- Why an unconnected input is never "blank".
- What happens when two wires feed the same input.
- What "oscillating" means and why the bench sometimes refuses to settle.

This lesson has no single preset — it explains the rules behind *every* circuit you build. Understanding them turns confusing surprises into predictable behaviour, and lets you experiment with confidence.

Rule 1 — An unconnected input reads 0 (LOW)

If a gate input has **no wire** attached, the bench treats it as **0**, not as "undefined" or "ignore me". This is why, back in Lesson 09, an AND gate with one input left dangling stayed dark forever — it was reading that empty input as a solid 0.

This rule is simple but easy to forget. If a circuit behaves as though some input is stuck at 0, the first thing to check is whether you actually wired that input.

> **Pitfall:** "no wire" looks like nothing, but it *acts* like a LOW source. Always account for every input on every gate.

Rule 2 — Two wires into one input are OR-ed together

What if you wire **two** sources into the *same* input port of a gate? The bench combines them with **OR**: the input reads 1 if *either* incoming wire is 1. They are effectively merged.

This is a deliberate, friendly simplification (real hardware would need a special part to join two outputs safely). For your purposes it means: fanning several signals into one input gives you a free OR. Useful occasionally — but usually you want each input driven by exactly one wire, so you always know what is driving it.

Rule 3 — The circuit "settles", and sometimes cannot

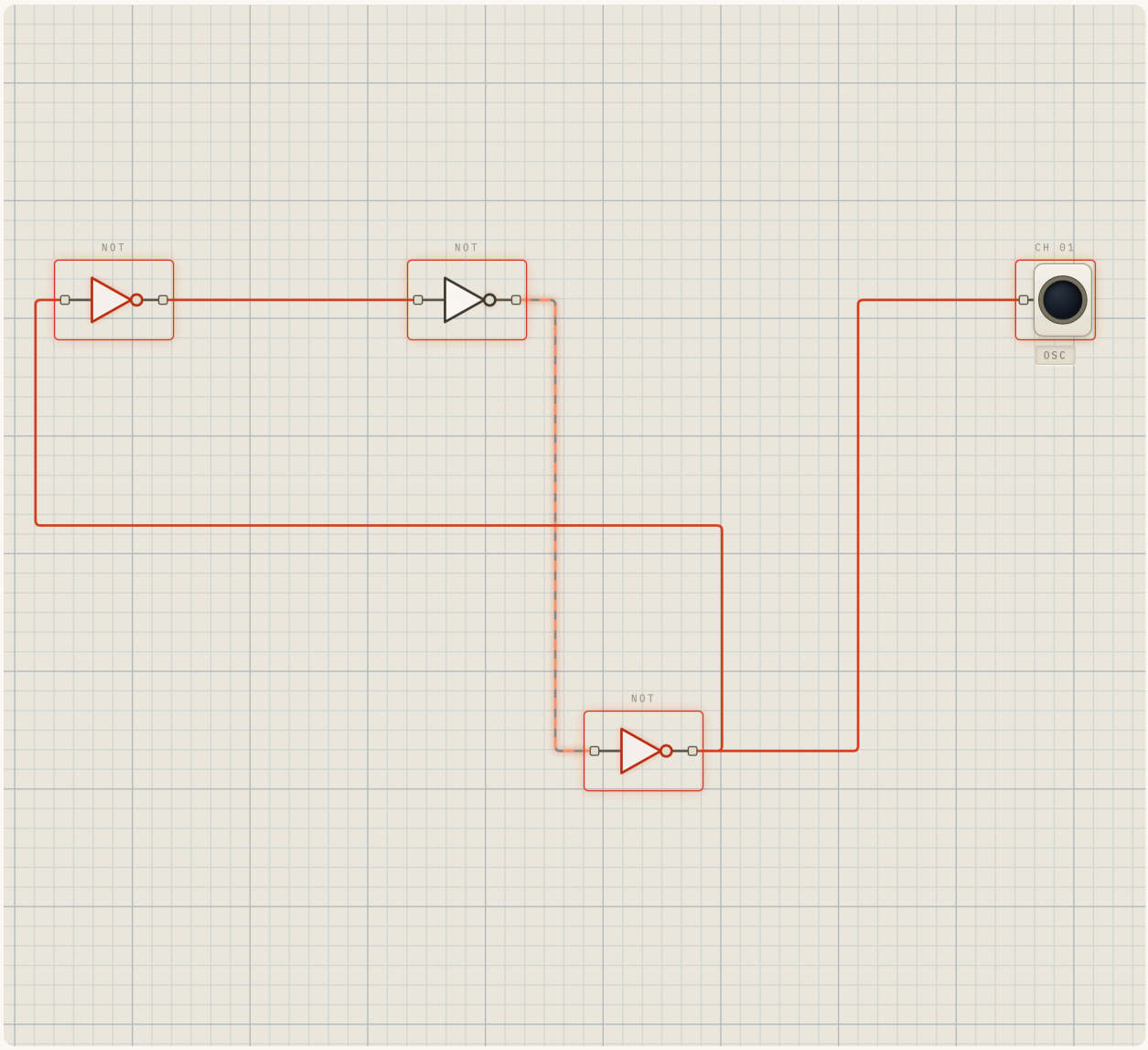
When you flip a switch, the bench does not magically know all the new wire values at once. It **sweeps** through the components repeatedly, updating each from its inputs, again and again, until a full pass produces **no more changes**. That stable end-state is called the **settled** result — it is what you see on screen.

For ordinary circuits (gates feeding forward to LEDs) this happens instantly and invisibly. The settling idea only becomes visible in two situations:

Feedback that settles — this is good. When you cross-couple gates so an output feeds back to an earlier input (as in the SR latch, Lesson 26), the sweeps settle into a stable held value. This is exactly what gives the bench *memory*: a latch keeps its stored bit instead of being recomputed from scratch. The settling process is what makes memory physically possible.

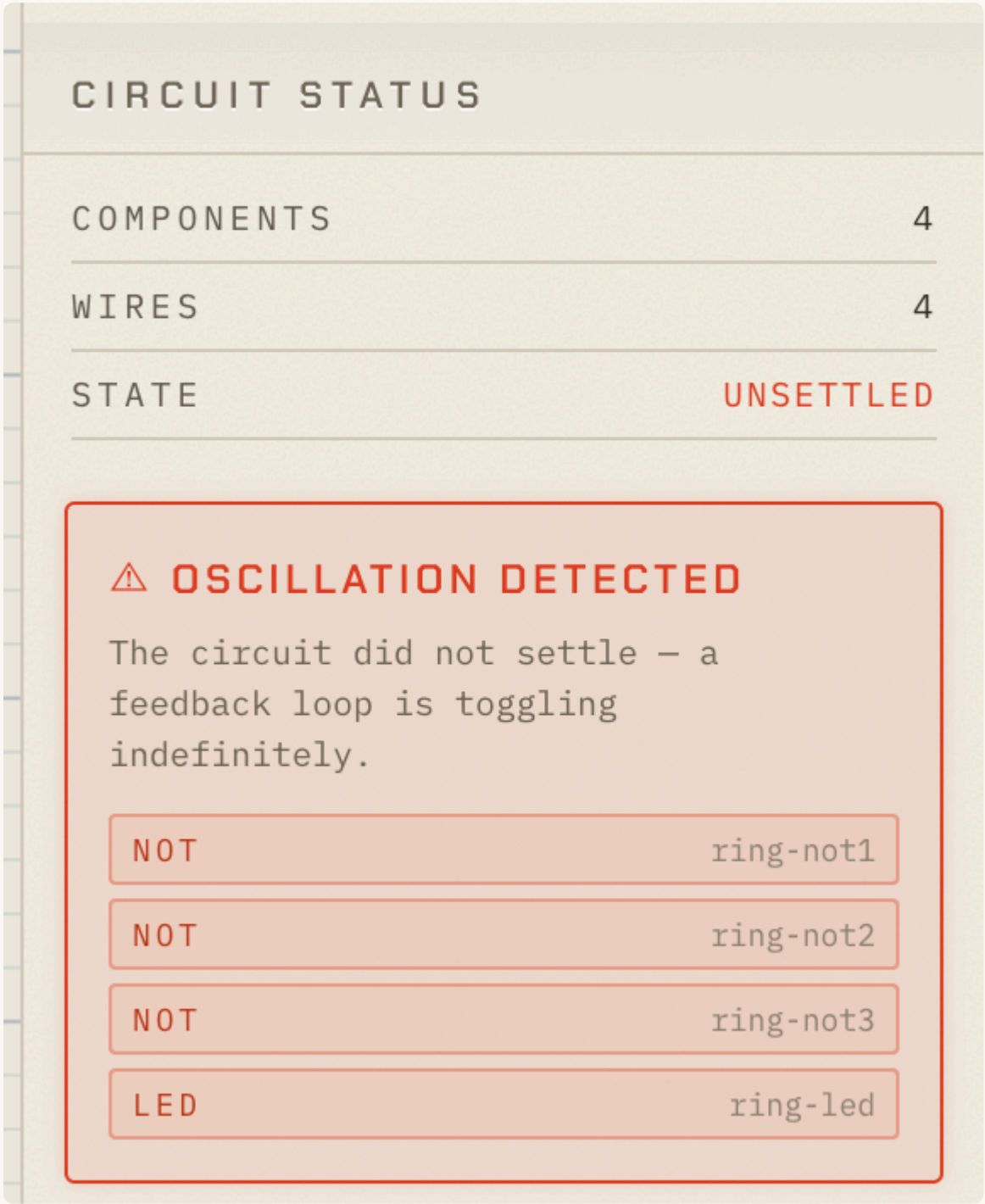
Feedback that never settles — the oscillation warning. Some loops have *no* stable state. The classic case is three inverters in a ring (Lesson 33's cousin): the signal chases its own tail, 0 forcing 1 forcing 0, forever. The bench detects that the sweeps never stop changing and raises an **OSCILLATING** warning instead of pretending there is an answer.

> **See it:** load **Ring Oscillator** from the library (category **BASICS**) — three NOT gates in a loop. Watch the unstable node pulse and the **OSCILLATING** status appear. The bench is being honest: this circuit genuinely has no settled value.



The Ring Oscillator showing the OSCILLATING warning

The **Circuit Status** panel makes the diagnosis explicit — it reports the circuit as *unsettled* and names the offending loop, rather than showing a value that would be a lie:



The Circuit Status panel reporting OSCILLATION DETECTED

This honesty is the point. A lesser simulator would pick an arbitrary value and move on; this one tells you the truth — that the circuit has no settled answer.

Why these rules are worth knowing

Almost every "why is my circuit doing that?" moment traces back to one of these three rules: a forgotten wire reading 0, two wires unexpectedly OR-ing, or a feedback loop that either holds (memory) or oscillates (no answer). Knowing them, you can predict the bench instead of being surprised by it — which is the whole point of experimenting.

Recap

- **Unconnected input → 0.** Empty is not blank; it acts LOW.
- **Two wires into one input → OR.** They merge.
- **The circuit settles** by repeated sweeps; stable feedback gives **memory**, unstable feedback gives the **OSCILLATING** warning.

Check yourself: You wire two switches into the *same* input of an LED. One is on, one off. Is the LED lit? *(Yes — the two wires are OR-ed, and one is 1.)*

Next: Lesson 15 — Gates from NAND: the proof that one gate can build them all.

Part II — Universality

Lesson 15 — Gates from NAND

> **Part II · Universality** — library circuit (category: UNIVERSAL GATES)
> Before this lesson: how the bench thinks (Lesson 14). After this: Gates from NOR (Lesson 16).

What you will learn

- What "functionally complete" (universal) means.
- How NOT, AND, OR, NOR and XOR can each be built from NAND gates alone.
- Why this single fact underpins all of digital hardware.

The idea

In Lesson 11 you met the claim: the NAND gate is **universal**. This lesson makes good on it. *Universal* (or **functionally complete**) means that with copies of just this one gate, you can build every other gate — and therefore any digital circuit at all, up to and including the CPU at the end of this book.

That is a startling economy. You do not need six different gate types in your toolbox; you need **one**, used cleverly. Here is the intuition for the key constructions:

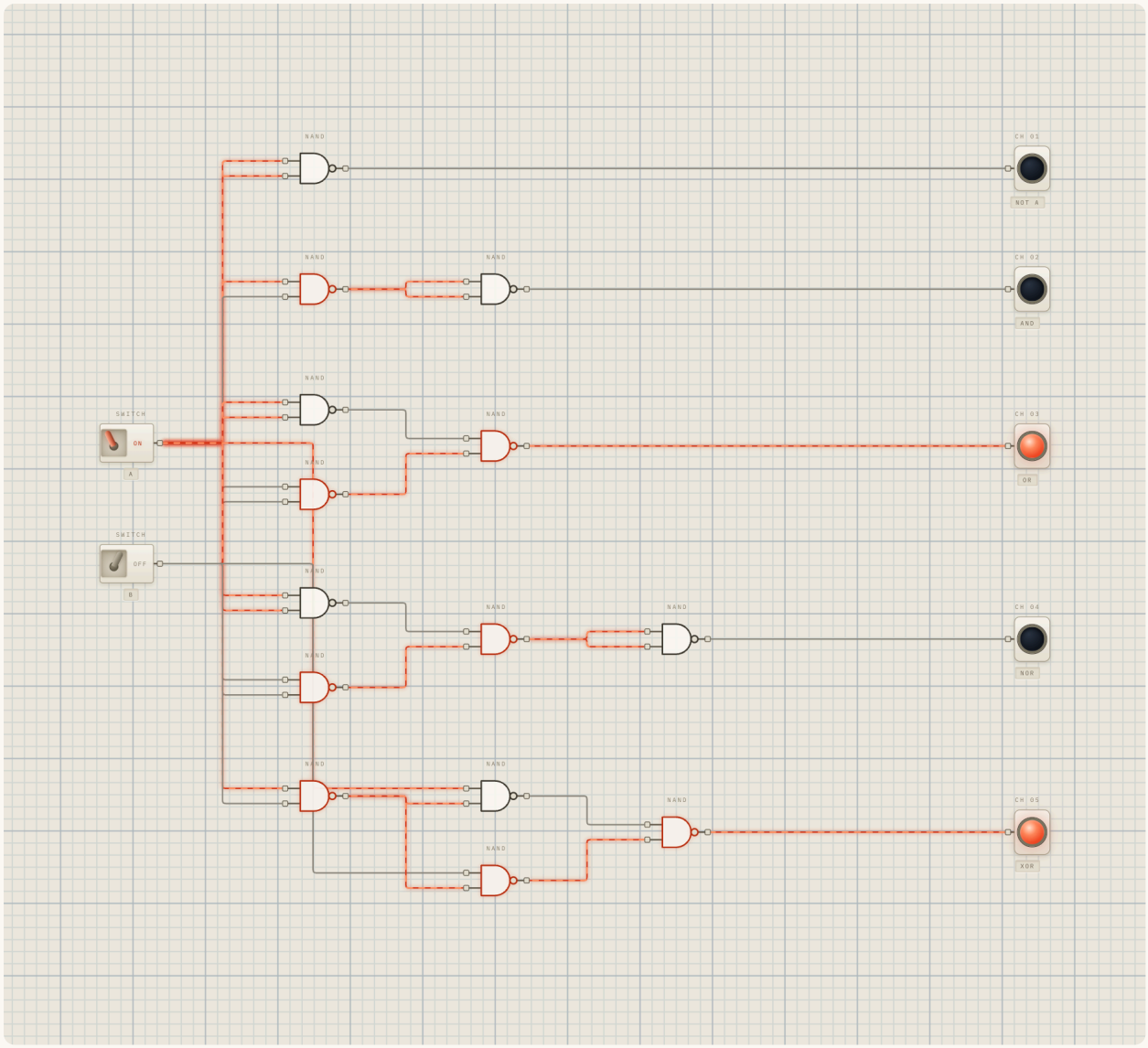
- **NOT from NAND:** tie both inputs of a NAND together. NAND(A, A) = NOT A. (You discovered this in Lesson 11.)
- **AND from NAND:** a NAND followed by a NOT (itself a NAND). NAND then invert = AND.
- **OR from NAND:** invert *both inputs first*, then NAND them. By De Morgan's law, NAND(NOT A, NOT B) = A OR B.
- **NOR from NAND:** build OR as above, then invert once more.

- **XOR from NAND:** four NANDs in the clever arrangement you will study in Lesson 17.

You do not need to memorise these. The point is that they all *exist*, and you can watch them all work at once.

See it in the bench

> **Open this:** load **Gates from NAND** from the library (category **UNIVERSAL GATES**). Two shared switches, A and B, feed five separate NAND-only constructions — one each for NOT, AND, OR, NOR and XOR — with an output LED for each.



Gates from NAND: five NAND-only constructions sharing two inputs

Try it yourself

Predict first, then flip.

Try: 1. Toggle A and B through all four combinations. For each, check every one of the five output LEDs against the truth table you learned for that gate. They all match — built from NAND alone. 2. Focus on the **OR** construction: notice it needs the inputs inverted first. That extra inversion is De Morgan's law made of wire. 3. Appreciate what you are seeing: *every* gate in Part I, reproduced with a single gate type. A chip factory that can print only NAND gates can build anything.

Recap

- **Universal / functionally complete:** one gate type suffices to build all circuits.
- NAND is universal — NOT, AND, OR, NOR and XOR all follow from it.
- This is why NAND is a favourite building block in real silicon.

Check yourself: How do you make a NOT gate from a single NAND? *(Tie both of its inputs to the same signal.)*

Next: Lesson 16 — Gates from NOR, the other universal gate — the one that flew to the Moon.

Lesson 16 — Gates from NOR

> **Part II · Universality** — library circuit (category: UNIVERSAL GATES)
> Before this lesson: Gates from NAND (Lesson 15). After this: XOR from NAND (Lesson 17).

What you will learn

- That NOR is the *other* universal gate.
- How NOT, OR, AND, NAND and XOR are each built from NOR alone.
- A piece of real history that rode on this fact.

The idea

NAND is not the only universal gate. **NOR is universal too** — with copies of just the NOR gate, you can build every other gate and therefore any circuit at all. There are exactly two single-gate universal building blocks in ordinary logic, and these are they: NAND and NOR.

This is not a mere curiosity. The **Apollo Guidance Computer** — the machine that navigated astronauts to the Moon and back — was built almost entirely from NOR gates. When engineers needed a computer they could trust with human lives on limited 1960s technology, building everything from one well-understood gate was a feature, not a limitation.

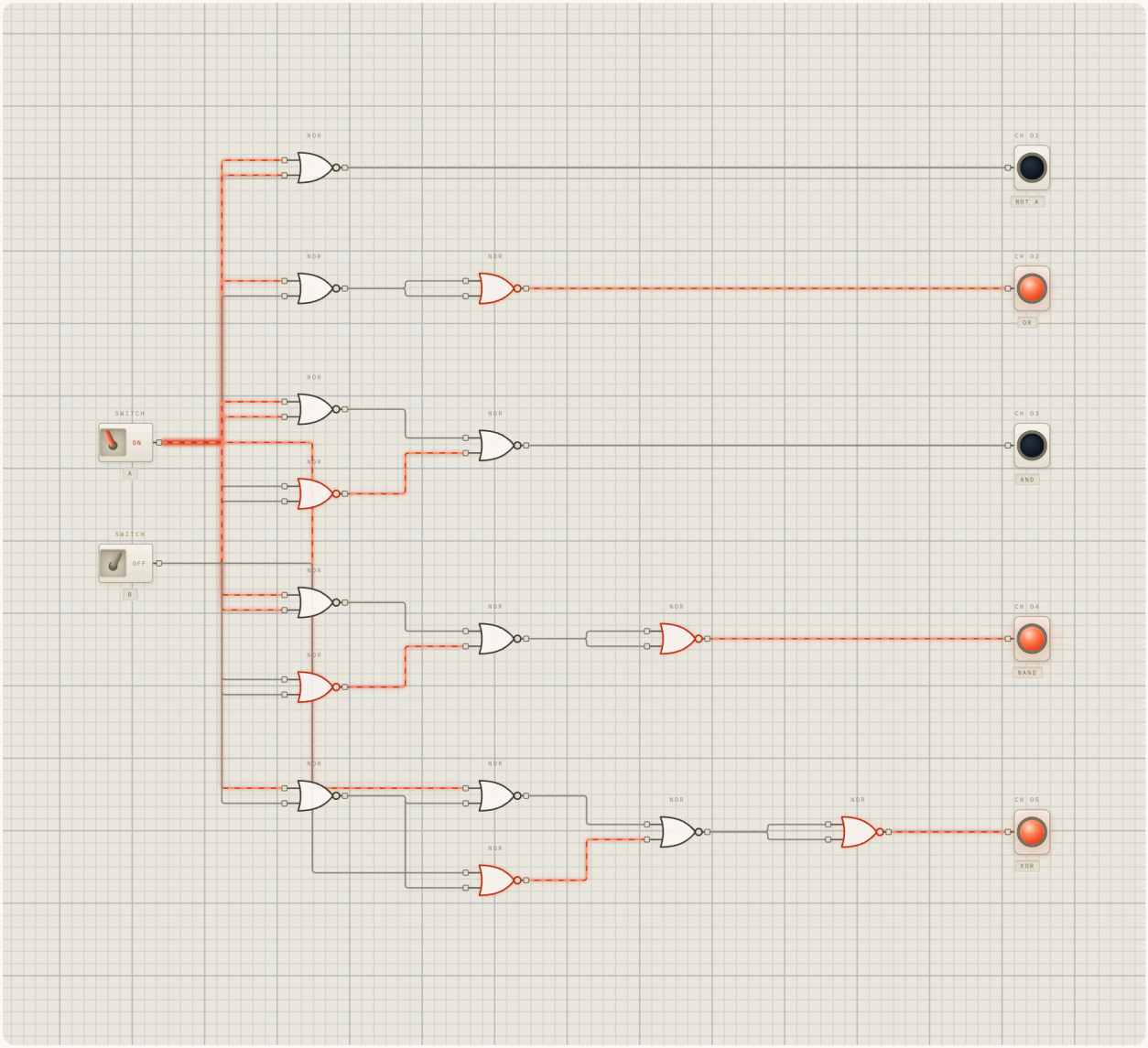
The constructions mirror the NAND ones, with OR and AND swapping roles:

- **NOT from NOR:** tie both inputs of a NOR together. NOR(A, A) = NOT A.
- **OR from NOR:** a NOR followed by a NOT.
- **AND from NOR:** invert both inputs first, then NOR them (De Morgan again).
- **NAND from NOR:** build AND as above, then invert.

- **XOR from NOR:** a NOR-only arrangement.

See it in the bench

Open this: load **Gates from NOR** from the library (category **UNIVERSAL GATES**). The same two shared switches A and B drive five NOR-only constructions for NOT, OR, AND, NAND and XOR, each with its own LED.



Gates from NOR: five NOR-only constructions sharing two inputs

Try it yourself

Predict first, then flip.

Try: 1. Step A and B through all four combinations and verify each of the five outputs against its expected truth table. All correct, from NOR alone. 2. Compare with Lesson 15: notice that NAND made OR awkward (inputs inverted first), while NOR makes **AND** the awkward one. Each universal gate builds its "natural" partner cheaply and its opposite with extra inversions. 3. Reflect: the two circuits in Lessons 15 and 16 together prove that digital logic rests on a remarkably tiny foundation — a single kind of gate is enough.

Recap

- NOR is the **second universal gate**: all gates, hence all circuits, can be built from it.
- The Apollo Guidance Computer was built almost entirely from NOR gates.
- Where NAND builds OR cheaply, NOR builds AND cheaply — mirror images.

Check yourself: Name the two single gates that are each, on their own, enough to build every circuit. *(NAND and NOR.)*

Next: Lesson 17 — XOR from NAND: one universal construction in close-up.

Lesson 17 — XOR from NAND

> **Part II · Universality** — library circuit (category: UNIVERSAL GATES)
> Before this lesson: Gates from NOR (Lesson 16). After this: the Half Adder (Lesson 18).

What you will learn

- How exactly four NAND gates combine to make an XOR.
- How to trace a signal through a small multi-gate circuit.
- Why this particular construction is worth studying on its own.

The idea

In Lesson 15 you saw five gates built from NAND all at once. This lesson zooms in on the most interesting of them: the **XOR built from four NAND gates**. XOR is the trickiest of the basic gates to construct, and the classic four-NAND arrangement is a small masterpiece worth tracing by hand.

Here is the structure. Call the inputs A and B:

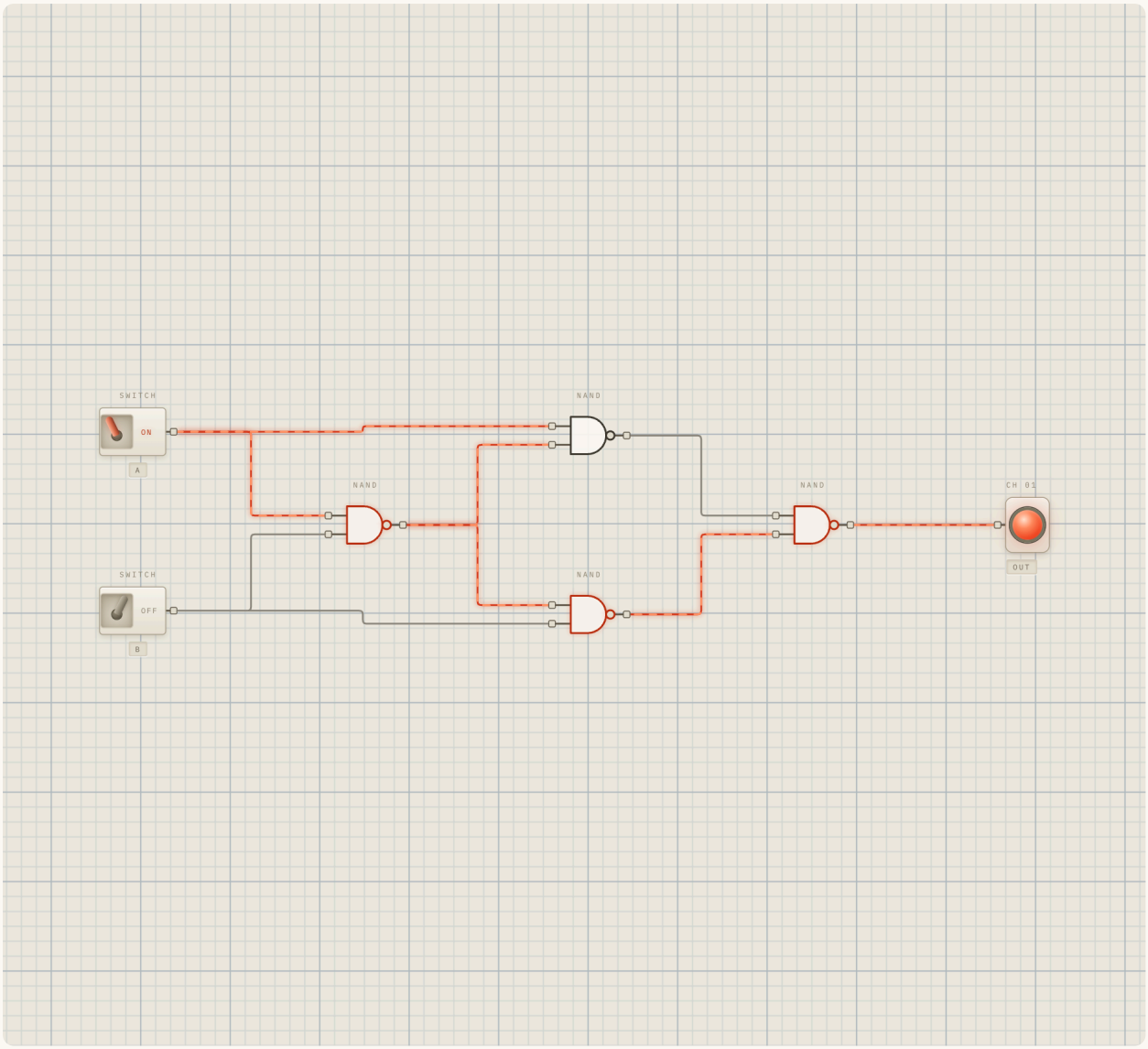
1. **NAND 1** takes A and B. Call its output `m` (for "middle"). It is 0 only when both inputs are 1.
2. **NAND 2** takes A and `m`.
3. **NAND 3** takes `m` and B.
4. **NAND 4** takes the outputs of NAND 2 and NAND 3, and produces the final result.

Work through the four input cases and you will find the output is 1 exactly when A and B differ — the XOR truth table. The shared middle gate `m` is the clever pivot that makes four gates enough.

You do **not** need to hold all of that in your head. The reason to meet it is that XOR appears constantly from here on (every adder uses it), and seeing it dissected into NANDs cements the lesson of Part II: even the "hard" gate is just a handful of one universal gate, wired with care.

See it in the bench

> **Open this:** load **XOR from NAND** from the library (category **UNIVERSAL GATES**). You will see two switches, four NAND gates in the arrangement above, and one output LED.



XOR from four NAND gates, inputs differing, output lit

Try it yourself

Predict first, then flip.

Try: 1. Set exactly one of A or B high. OUT lights — the inputs differ. 2. Make them the same (both on, or both off). OUT goes dark. 3. Now the rewarding part: pick one input combination and **trace it gate by gate**. Read the output of NAND 1 (`m`), then NAND 2 and NAND 3, then NAND 4. Confirm each gate is obeying the plain NAND rule you learned in Lesson 11, and that they compose into XOR. Following one signal all the way through is the single best way to stop seeing a circuit as magic.

Recap

- XOR can be built from exactly **four NAND gates** sharing a clever middle term.
- Tracing the four gates by hand confirms XOR emerges from plain NAND behaviour.
- This closes Part II: even the hardest basic gate is just NANDs, wired with care.

Check yourself: XOR is the "hard" gate to build. How many NAND gates does the classic construction use? *(Four.)*

Next: Lesson 18 — The Half Adder: now we put gates to work doing arithmetic.

Part III — Arithmetic

Lesson 18 — The Half Adder

> **Part III · Arithmetic** — library circuit (category: ARITHMETIC) Before this lesson: the XOR gate (Lesson 13) and the AND gate (Lesson 09). After this lesson: the Full Adder (Lesson 19).

What you will learn

- How two gates you already know combine to do real binary addition.
- What **SUM** and **CARRY** mean, and why one bit of adding needs two output bits.
- Why this tiny circuit is called the *smallest building block of arithmetic*.
- How to read `1 + 1 = 10` off the lights.

The idea

You have met the XOR gate (output 1 when inputs *differ*) and the AND gate (output 1 when *both* inputs are 1). Put them side by side, feed them the same two inputs A and B, and something remarkable happens: **you have built a machine that adds.**

Think about adding two single binary digits. There are only four cases:

- `0 + 0 = 0`
- `0 + 1 = 1`
- `1 + 0 = 1`
- `1 + 1 = 10` ← two! which is *one-zero* in binary

The first three fit in one output bit. The fourth does not — `1 + 1` is **2**, written `10` in binary, so it needs a second bit to carry into. That second bit is the **CARRY**, exactly like the little 1 you carry when 7 + 5 spills past 9 in ordinary decimal addition.

So one-bit addition produces **two** outputs:

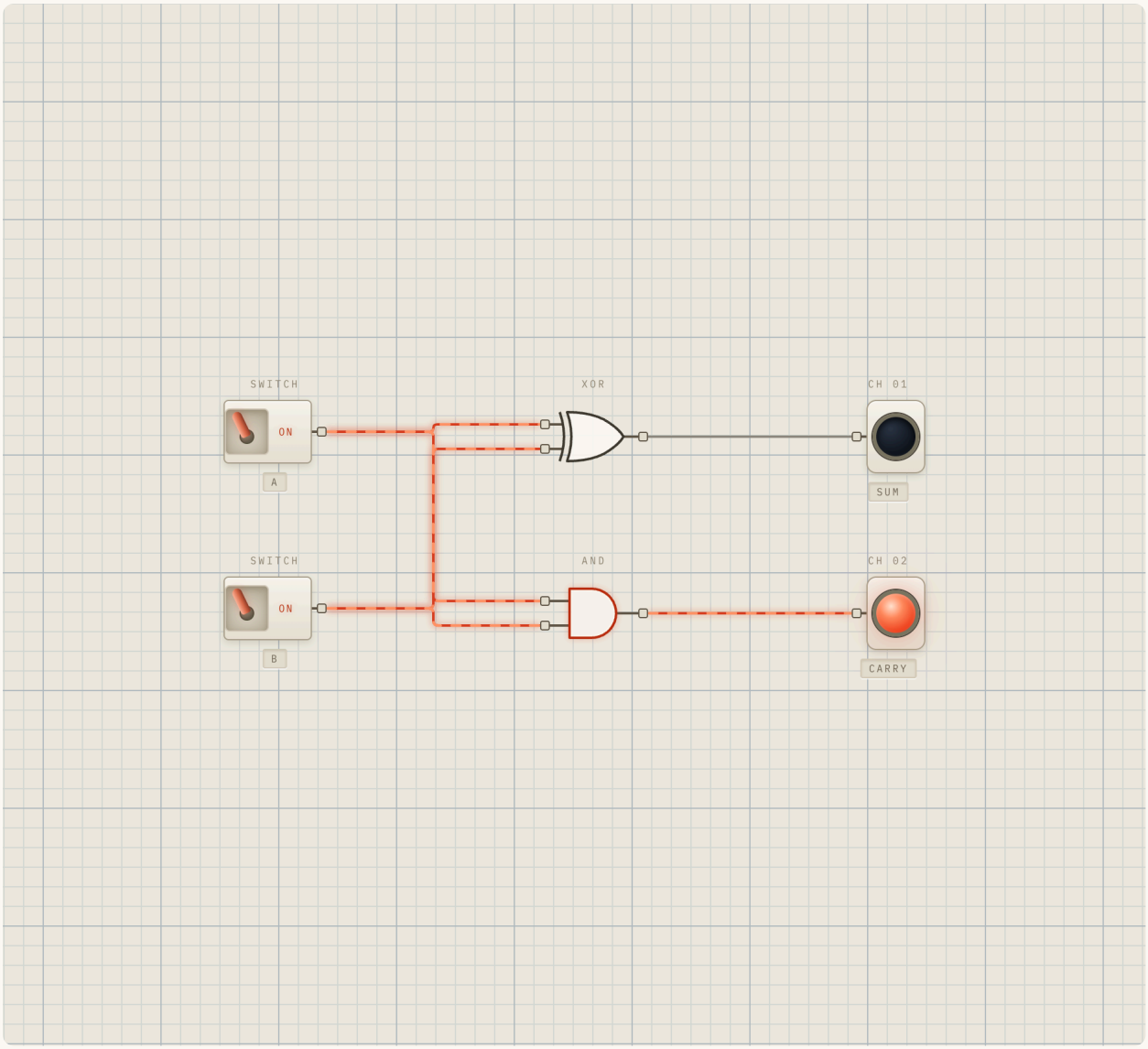
- **SUM** — the bottom bit of the answer. This is precisely XOR(A, B).
- **CARRY** — the overflow bit, 1 only when both inputs are 1. This is precisely AND(A, B).

That is the entire half adder: an XOR for the sum, an AND for the carry, sharing the same two inputs.

See it in the bench

> **Open this:** load **Half Adder** from the library (category **ARITHMETIC**).

You will see two input switches, A and B, each fanning out to *two* gates: an XOR (driving the SUM led) and an AND (driving the CARRY led).



The Half Adder with both inputs ON: SUM dark, CARRY lit — reading binary 10

Watch the four cases

Walk the inputs through all four combinations and watch the two LEDs together as a two-bit number, **CARRY SUM**:

A	B	CARRY	SUM	reads as
0	0	0	0	0
0	1	0	1	1
1	0	0	1	1
1	1	1	0	2 (binary 10)

Look at the SUM column on its own — it is the XOR truth table (1 when the inputs differ). Look at the CARRY column on its own — it is the AND truth table (1 only when both are high). The half adder is nothing more than those two gates you already know, *read together as a number*.

Try it yourself

> **Predict first, then flip.**
>
> **Try:** 1. Both switches **off** — predict both LEDs. *(0 + 0 = 0: both dark.)* 2. Turn on **just A** — SUM should light, CARRY stays dark. You are looking at the answer **1**. 3. Turn on **B as well** (so both are on). Watch carefully: **SUM goes dark and CARRY lights.** This is the heart of the lesson — `1 + 1 = 10`, so the sum bit rolls to 0 and the carry bit becomes 1. 4. Turn **A** back off (just B on) — back to SUM lit, CARRY dark: the answer **1** again.

The step that surprises people is step 3 — adding *more* makes the SUM light go *out*. That is not a glitch; that is binary carrying, exactly like 9 + 1 making the ones digit roll to 0 and a 1 carry left.

Look inside (and look ahead)

This circuit is built from two plain gates, so there is no chip to open *here* — what you see is the whole truth. But hold onto the shape of it, because in the **next lesson** you will chain two half adders together (plus one OR gate) to build a **Full Adder**, which can also accept a *carry coming in* from a previous stage. And once a stage can take a carry in and pass a carry out, you can line them up side by side to add big multi-bit numbers — which is exactly how the 4-Bit and 8-Bit Adders later in this Part are built.

> **The ladder from here:** Half Adder → Full Adder → 4-Bit Adder → 8-Bit Adder → the adder *inside* the CPU's ALU. It genuinely is this circuit, repeated and chained, all the way up.

Recap

- A half adder adds two single bits and produces a two-bit answer: **CARRY** and **SUM**.
- **SUM = XOR(A, B)** (the answer bit); **CARRY = AND(A, B)** (the overflow bit).
- `1 + 1 = 10`, so the both-on case is the one where SUM drops to 0 and CARRY rises to 1.
- It is the smallest building block of binary arithmetic — everything that adds is built from this.

Check yourself: Why does a *single* bit of addition need *two* output wires? *(Because 1 + 1 = 2, which does not fit in one bit — it needs a carry.)*

Next: Lesson 19 — The Full Adder, which adds a carry-in so adders can be chained.

Lesson 19 — The Full Adder

> **Part III · Arithmetic** — library circuit (category: ARITHMETIC) Before this lesson: the Half Adder (Lesson 18). After this: the 4-Bit Adder (Lesson 20).

What you will learn

- Why the half adder is not quite enough to build a real adder.
- What a **carry-in** is and why it changes everything.
- How two half adders plus an OR gate make a full adder you can chain.

The idea

The half adder (Lesson 18) adds two bits. But think about adding two *multi-bit* numbers by hand, column by column: each column receives not just its own two digits, but also a **carry coming in** from the column to its right. The half adder has nowhere to accept that incoming carry. It is "half" an adder precisely because it is missing this third input.

The **full adder** fixes that. It takes **three** inputs — A, B, and a **carry-in (CIN)** — and produces the same two outputs as before: a **SUM** bit and a **carry-out (COUT)**. Now a single stage can absorb the carry from the previous stage and pass its own carry onward. That is the missing piece that lets you chain stages into adders of any width.

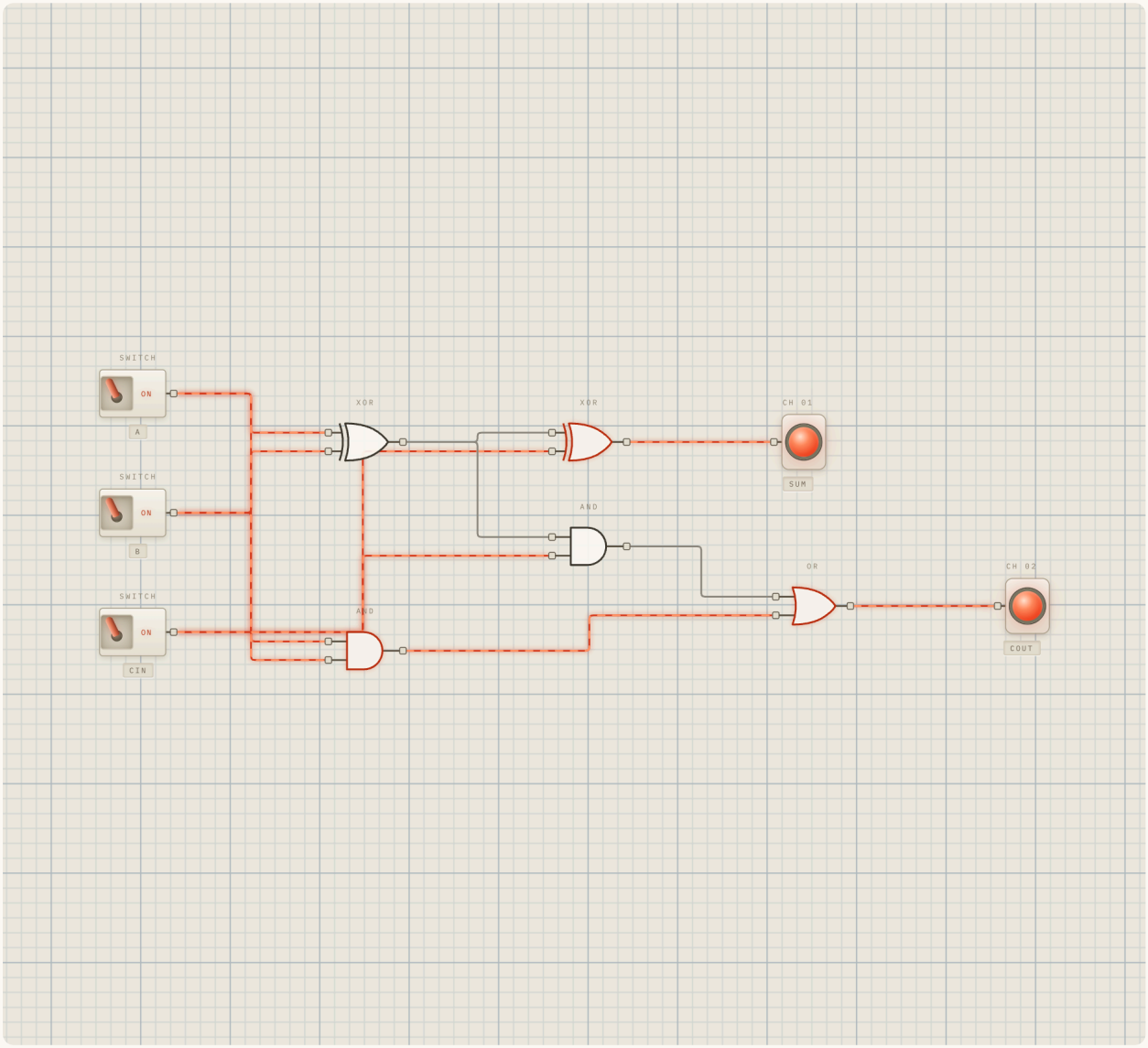
How is it built? From **two half adders and one OR gate**:

1. The first half adder adds A and B, giving a partial sum and a first carry.
2. The second half adder adds that partial sum to CIN, giving the final SUM and a second carry.
3. An OR gate merges the two carries into COUT. (A carry out happens if *either* half-add produced one.)

So the full adder is literally two of the circuit from the previous lesson, stitched together. Hierarchy in action.

See it in the bench

> **Open this:** load **Full Adder** from the library (category **ARITHMETIC**). You will see three input switches — A, B, CIN — feeding two XOR/AND half-adder cores and an OR gate, driving SUM and COUT.



The Full Adder with all three inputs on: SUM and COUT both lit

The truth table

Read the outputs together as the two-bit number **COUT SUM**:

A	B	CIN	COUT	SUM	reads as
0	0	0	0	0	0
0	0	1	0	1	1
0	1	0	0	1	1
0	1	1	1	0	2
1	0	0	0	1	1
1	0	1	1	0	2
1	1	0	1	0	2
1	1	1	1	1	3

Each row is just A + B + CIN, written as a two-bit number. The bottom row — all three inputs 1 — gives 3, which is binary `11`: both outputs lit.

Try it yourself

Predict first, then flip.

Try: 1. Switch on **all three** inputs. SUM and COUT both light: 1 + 1 + 1 = 3 = binary `11`. 2. Set A and B on but CIN off (that is 1 + 1 + 0 = 2): SUM dark, COUT lit — binary `10`. 3. Now imagine the COUT wire of this stage running into the CIN of an identical stage to its left. That single connection is how you go from adding one bit to adding a whole byte — the subject of the next lesson.

Recap

- A full adder adds **three** bits (A, B, carry-in) → SUM and carry-out.
- It is built from **two half adders plus an OR** that merges the carries.
- The carry-in/carry-out pair is what lets stages **chain** into wide adders.

Check yourself: What can a full adder do that a half adder cannot? *(Accept a carry-in, so it can be chained into multi-bit adders.)*

Next: Lesson 20 — The 4-Bit Adder: four full adders in a row.

Lesson 20 — The 4-Bit Adder

> **Part III · Arithmetic** — library circuit (category: ARITHMETIC) Before this lesson: the Full Adder (Lesson 19). After this: the 8-Bit Adder (Lesson 21).

What you will learn

- How four full adders chain into a circuit that adds two 4-bit numbers.
- What "ripple carry" means and why the carry has to travel.
- How to read a 4-bit sum on the hex display.

The idea

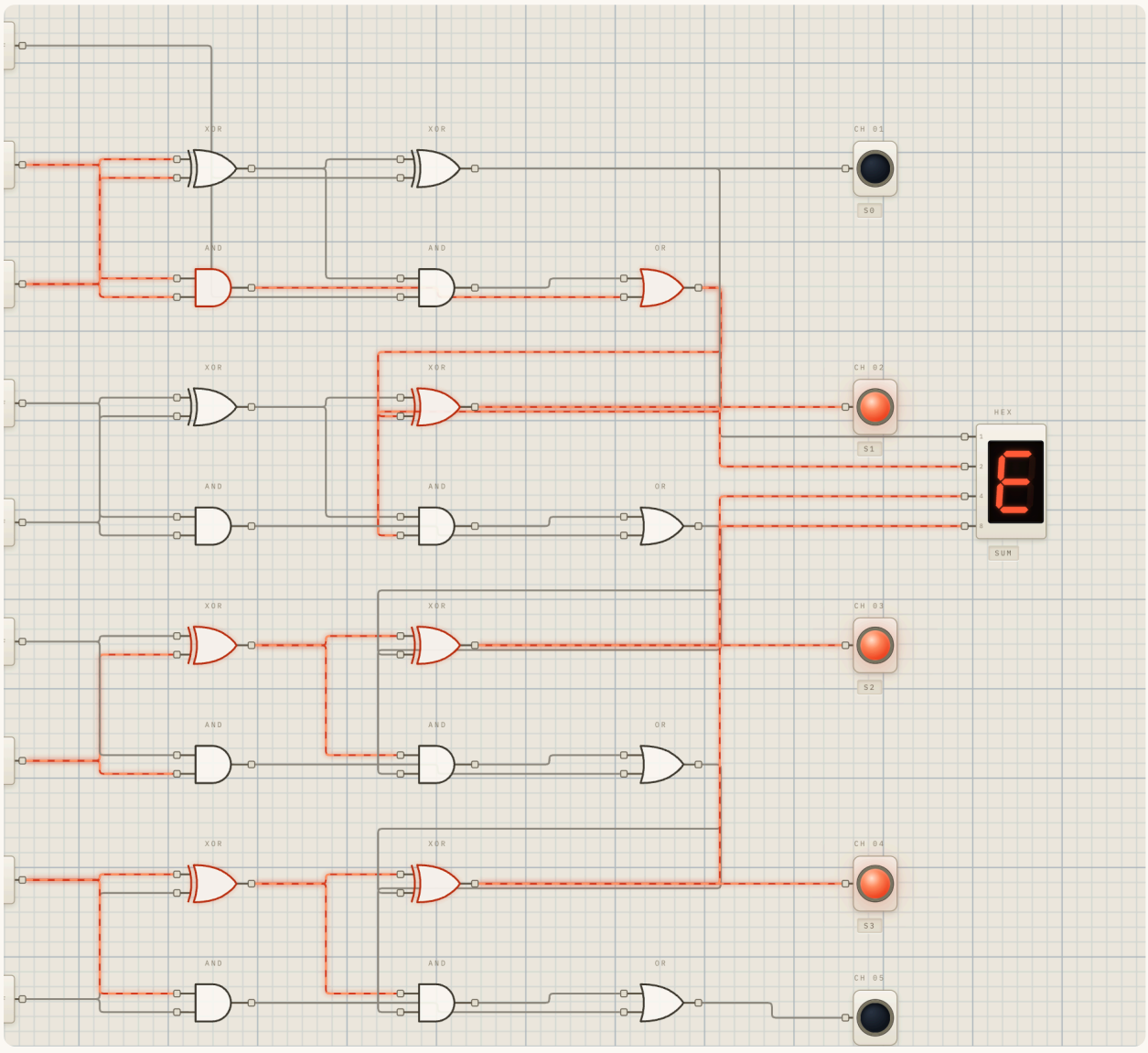
A single full adder handles one column. To add two **4-bit** numbers (values 0–15 each), you line up **four full adders**, one per bit position, and wire each stage's **carry-out into the next stage's carry-in**. The rightmost (least significant) stage takes the overall carry-in — usually tied to 0 — and the leftmost stage's carry-out becomes the final **COUT**, signalling overflow past 15.

This design is called a **ripple-carry adder**, and the name is a vivid description of how it works. When you add, the carry generated in the lowest column may force a carry in the next column, which may force one in the next, and so on — the carry **ripples** left along the chain, one stage at a time, like a row of dominoes. The answer is not final until the ripple has finished travelling.

That rippling is also the circuit's weakness: the more bits, the longer the carry may have to travel, and the longer you must wait for the sum to settle. This is precisely why real CPUs eventually use cleverer (but bulkier) fast-carry schemes. You are meeting the honest, simple version first — the one whose behaviour you can see.

See it in the bench

Open this: load **4-Bit Adder** from the library (category **ARITHMETIC**). Two banks of switches set the 4-bit operands A and B; four chained full adders produce S0–S3 and a final COUT, with the sum also shown on a hex display.



The 4-bit ripple-carry adder computing a sum on the hex display

Try it yourself

Predict first, then flip.

Try: 1. Set **A = 9** (switch on A0 and A3 — that is 1 + 8) and **B = 5** (A0… rather B0 and B2 — that is 1 + 4). Predict the sum: 9 + 5 = 14. The hex display reads **E**. 2. Now push it to overflow: set A and B both near 15 (e.g. A = 12, B = 6). The 4 sum bits cannot hold 18, so **COUT** lights — that is the carry out of the top bit, the circuit telling you the answer spilled past 15. 3. Pick an addition that forces a long ripple (for example 15 + 1) and remember that the carry had to travel through all four stages to reach the top. With four bits it is instant to your eye; with many bits it would not be.

Recap

- Four full adders, carry-out chained to carry-in, add two 4-bit numbers.
- This is a **ripple-carry adder**: the carry travels left through the stages.
- The travelling carry is simple and visible but slow for wide numbers — the seed of why fast adders exist.

Check yourself: Why is it called "ripple" carry? *(Because a carry generated low down can ripple up through each stage in turn before the sum is final.)*

Next: Lesson 21 — The 8-Bit Adder: the same idea, scaled to a full byte.

Lesson 21 — The 8-Bit Adder

> **Part III · Arithmetic** — library circuit (category: ARITHMETIC) Before this lesson: the 4-Bit Adder (Lesson 20). After this: Subtractors (Lesson 22).

What you will learn

- How the ripple-carry idea scales to a full 8-bit byte.
- Why "just add more stages" really is the whole trick.
- How far a carry can travel, and why that matters for speed.

The idea

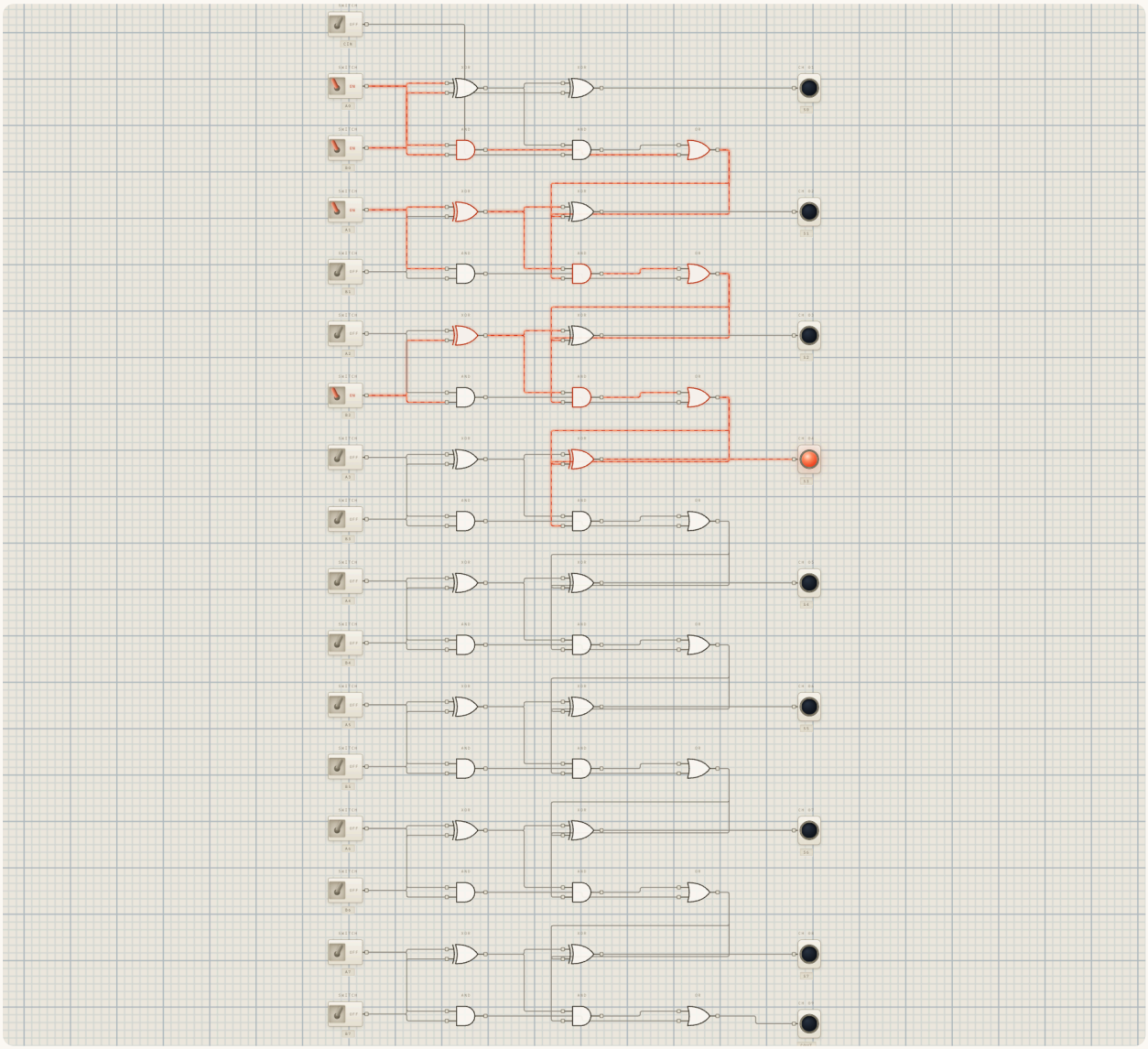
There is almost nothing new here — and that is the lesson. To add two **8-bit** numbers (0–255 each), you take the 4-bit adder of the previous lesson and simply keep going: **eight full-adder stages**, each one's carry-out feeding the next one's carry-in. The structure does not get cleverer; it gets *longer*.

This is one of the most important habits of mind in all of digital design: **once you have a stage that chains, width is free.** The full adder was designed with a carry-in and carry-out precisely so you could line up as many as you like. Four bits, eight bits, thirty-two bits — same circuit, more copies.

The one thing that *does* grow is the **carry's journey**. In the worst case (think 255 + 1), a carry born in the lowest bit must ripple all the way through all eight stages to the top. With eight bits this is still instant to your eye, but you can now feel why a 64-bit ripple adder would be sluggish, and why real processors invest in faster carry logic. You are watching the exact tradeoff that shaped real CPU design.

See it in the bench

Open this: load **8-Bit Adder** from the library (category **ARITHMETIC**). Two banks of eight switches set A and B; eight chained full adders produce S0–S7 plus COUT.



The 8-bit ripple-carry adder summing two bytes

Try it yourself

Predict first, then flip.

Try: 1. Set **A = 3** (A0, A1 on) and **B = 5** (B0, B2 on). Predict 8 — watch S3 light. 2. Try a sum that overflows a byte: **A = 200, B = 100**. The 8 sum bits cannot hold 300, so **COUT** lights and the visible sum wraps to 300 − 256 = 44. Overflow is not an error here — it is the carry-out faithfully reporting "the answer needed a ninth bit." 3. Compare the *layout* to the 4-bit adder. It is visibly the same circuit, just twice as long. Let that sink in: this is how all wide hardware is built — a good small stage, repeated.

Recap

- An 8-bit adder is **eight full adders chained** — the 4-bit adder, extended.
- Once a stage chains, **width is free**; you just add stages.
- The carry's worst-case journey grows with width, which is why fast-carry schemes exist in real CPUs.

Check yourself: What is genuinely different between the 4-bit and 8-bit adders, besides size? *(Almost nothing — the carry just has farther to travel.)*

Next: Lesson 22 — Subtractors: teaching the same gates to subtract.

Lesson 22 — Subtractors (half and full)

> **Part III · Arithmetic** — library circuits (category: ARITHMETIC) Before this lesson: the 8-Bit Adder (Lesson 21). After this: the 2-Bit Multiplier (Lesson 23).

What you will learn

- How subtraction is built from the same XOR core as addition.
- What a **borrow** is — the subtraction mirror of a carry.
- The difference between a half subtractor and a full subtractor.

The idea

Subtraction is addition's mirror, and its hardware mirrors the adder almost exactly. When you subtract by hand and a column's top digit is smaller than the one below it, you **borrow** from the next column. A **borrow** in subtraction plays the same role a **carry** plays in addition — it is the signal that passes between columns.

The half subtractor computes A − B for a single bit:

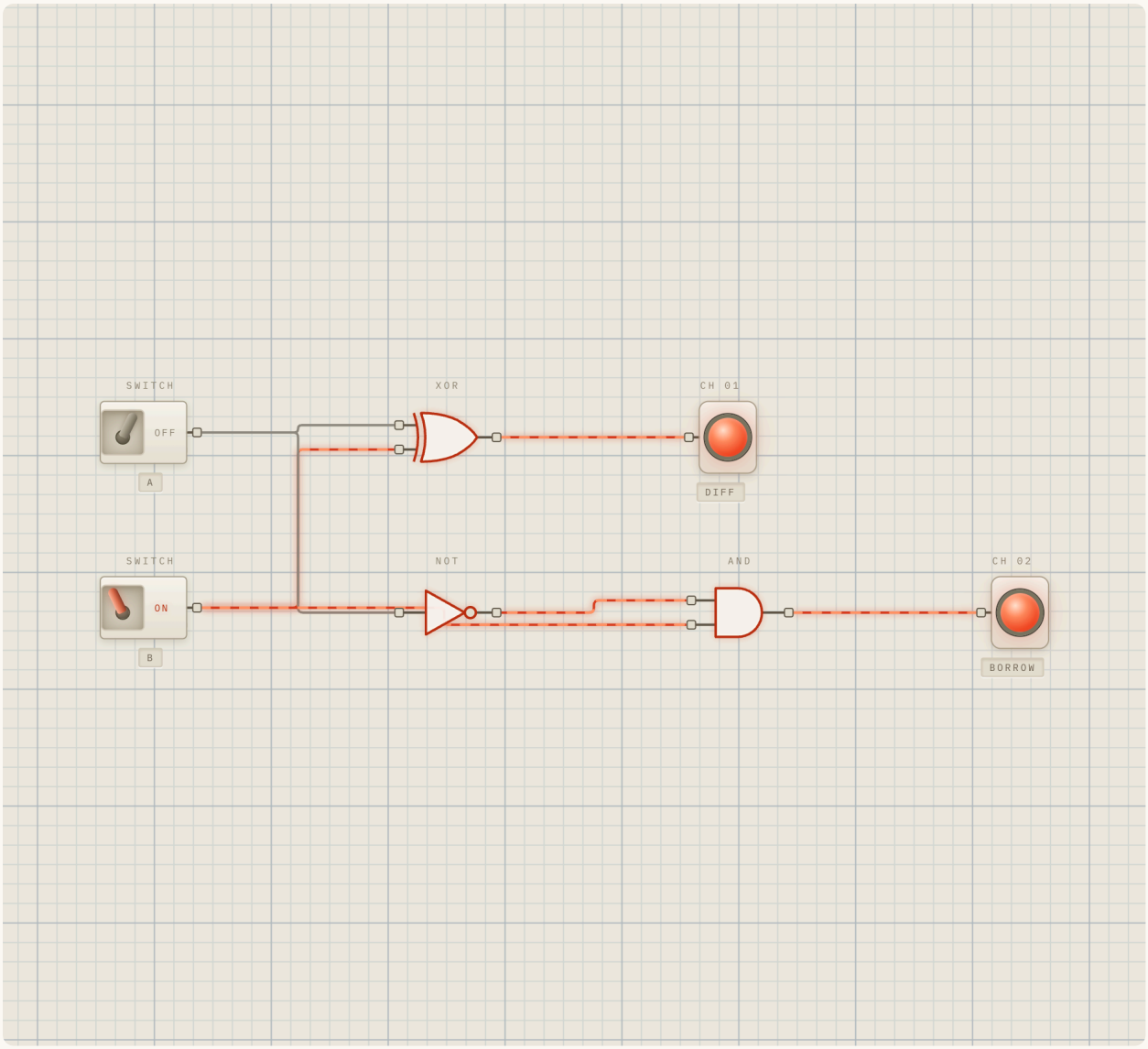
- **DIFF** (the difference bit) = XOR(A, B) — the same XOR you used for SUM.
- **BORROW** = (NOT A) AND B — raised when B is 1 but A is 0, i.e. when you are taking 1 from 0 and must borrow.

The full subtractor adds a third input, a **borrow-in (BIN)** coming from the previous column, and produces DIFF and a **borrow-out (BOUT)** to pass onward — exactly parallel to how the full adder added a carry-in. It shares the adder's XOR core, with inverters steering the borrow logic where the adder steered the carry.

The deep point: the **same XOR gate** is the heart of both adding and subtracting. The difference is only in how the second signal (carry vs borrow) is computed. This is a first hint of something you will see fully in the ALU (Lesson 47): a single piece of arithmetic hardware can both add and subtract, depending on how you steer it.

See it in the bench

> **Open this:** load **Half Subtractor** from the library (category **ARITHMETIC**), then later **Full Subtractor**.



The Half Subtractor computing 0 − 1, raising both DIFF and BORROW

The half subtractor truth table

A	B	BORROW	DIFF
0	0	0	0
0	1	**1**	**1**
1	0	0	1
1	1	0	0

The row that tells the story is **A = 0, B = 1**: you are computing 0 − 1, which you cannot do without help, so BORROW lights and DIFF becomes 1 (you have

effectively borrowed to make it 10 − 1 = 1).

Try it yourself

> **Predict first, then flip.**
>
> **Try (half subtractor):** 1. A on, B off (1 − 0): DIFF lights, BORROW dark — a clean 1. 2. A off, B on (0 − 1): **both** DIFF and BORROW light — you had to borrow. 3. Both on (1 − 1): everything dark — a clean 0.
>
> **Try (full subtractor):** load **Full Subtractor** and turn on **B and BIN** with A off. DIFF lights and **BOUT** signals the borrow being passed to the next column — the chaining mechanism, just like the full adder's carry-out.

Recap

- Subtraction uses the same **XOR** core as addition; only the between-column signal differs.
- A **borrow** is subtraction's carry: DIFF = XOR(A, B), and the borrow logic uses inverters.
- The **full** subtractor adds a borrow-in/borrow-out so it can chain into wide subtractors.

Check yourself: In subtraction, what plays the role that "carry" plays in addition? *(Borrow.)*

Next: Lesson 23 — The 2-Bit Multiplier: from adding to multiplying.

Lesson 23 — The 2-Bit Multiplier

> **Part III · Arithmetic** — library circuit (category: ARITHMETIC) Before this lesson: Subtractors (Lesson 22). After this: the 4-Bit Equality Comparator (Lesson 24).

What you will learn

- That multiplication, too, is built from the gates you already know.
- How "partial products" with AND gates and half adders make a multiplier.
- Why one AND gate already *is* a 1-bit multiplier.

The idea

Here is a small delight: the **AND gate is a one-bit multiplier**. Look at the multiplication table for single bits — 0×0 = 0, 0×1 = 0, 1×0 = 0, 1×1 = 1 — and compare it to the AND truth table. They are identical. Multiplying two bits *is* ANDing them.

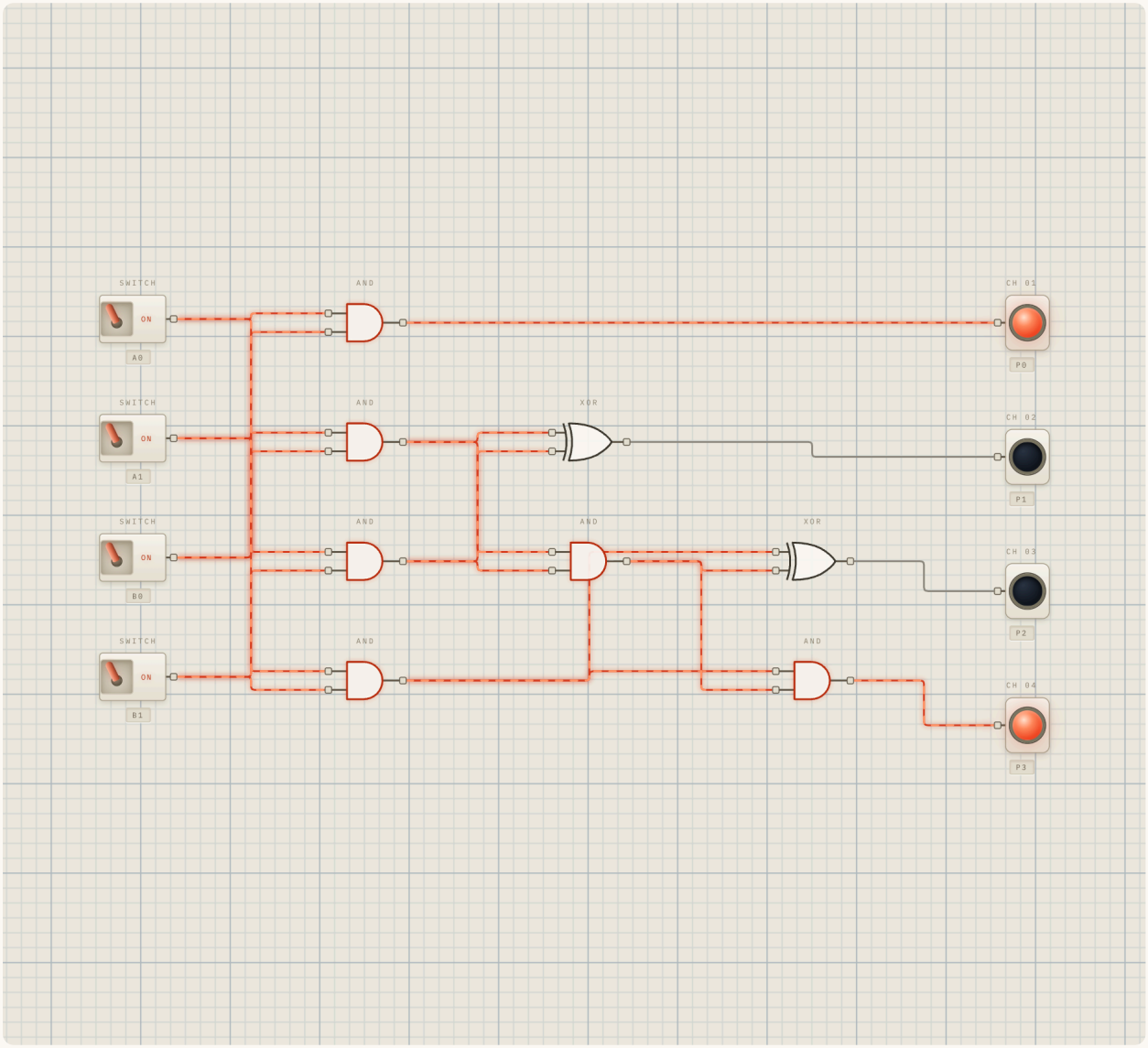
To multiply larger numbers, you do exactly what you learned in school for long multiplication: form **partial products** and add them up. For two 2-bit numbers A (= A1 A0) and B (= B1 B0):

1. **Four AND gates** form the four partial products: A0·B0, A1·B0, A0·B1, A1·B1.
2. **Two half adders** sum those partial products into the proper columns, producing the 4-bit result **P3 P2 P1 P0**.

That is the entire 2-bit multiplier: four ANDs to multiply the bit-pairs, and a little addition (which you already understand) to combine them. A real, if tiny, hardware multiplier — and a perfect example of building a new operation entirely out of the pieces from earlier lessons.

See it in the bench

Open this: load **2-Bit Multiplier** from the library (category **ARITHMETIC**). Two pairs of switches set A (A1 A0) and B (B1 B0); the product appears on P0–P3.



The 2-bit multiplier computing 3 × 3 = 9

Try it yourself

> **Predict first, then flip.**
>
> **Try:** 1. Set **A = 3** (A0, A1 on) and **B = 3** (B0, B1 on). Predict 3 × 3 = 9. The outputs read `1001` (P0 and P3 lit) = 9. The largest product two 2-bit numbers can make is exactly 9, which needs all four output bits. 2. Set **A = 2, B = 2** → predict 4 → P2 lights alone (`0100`). 3. Set **B = 0** (both B switches off). Whatever A is, the product is 0 — because every partial product passes through an AND with a 0, and 0 ANDed with anything is 0. Multiplication by zero, visible as gates going dark.

Recap

- An **AND gate is a 1-bit multiplier** (its table matches single-bit multiplication).
- Bigger multipliers form **partial products** (ANDs) and **add** them (half adders) — long multiplication in hardware.
- The 2-bit multiplier is four ANDs plus two half adders, producing a 4-bit product up to 9.

Check yourself: Which single gate already performs 1-bit multiplication? *(AND.)*

Next: Lesson 24 — The 4-Bit Equality Comparator: checking whether two numbers match.

Lesson 24 — The 4-Bit Equality Comparator

> **Part III · Arithmetic** — library circuit (category: ARITHMETIC) Before this lesson: the 2-Bit Multiplier (Lesson 23). After this: the Majority Voter (Lesson 25).

What you will learn

- How to test whether two multi-bit numbers are equal, in hardware.
- What an **XNOR** gate is (XOR with a NOT) and why it means "same".
- How an AND tree combines per-bit checks into one answer.

The idea

How does a circuit decide whether two 4-bit numbers A and B are **equal**? The trick is to break the big question into small ones: two numbers are equal exactly when **every pair of corresponding bits matches**. So check each bit position for a match, then insist that *all* the checks pass.

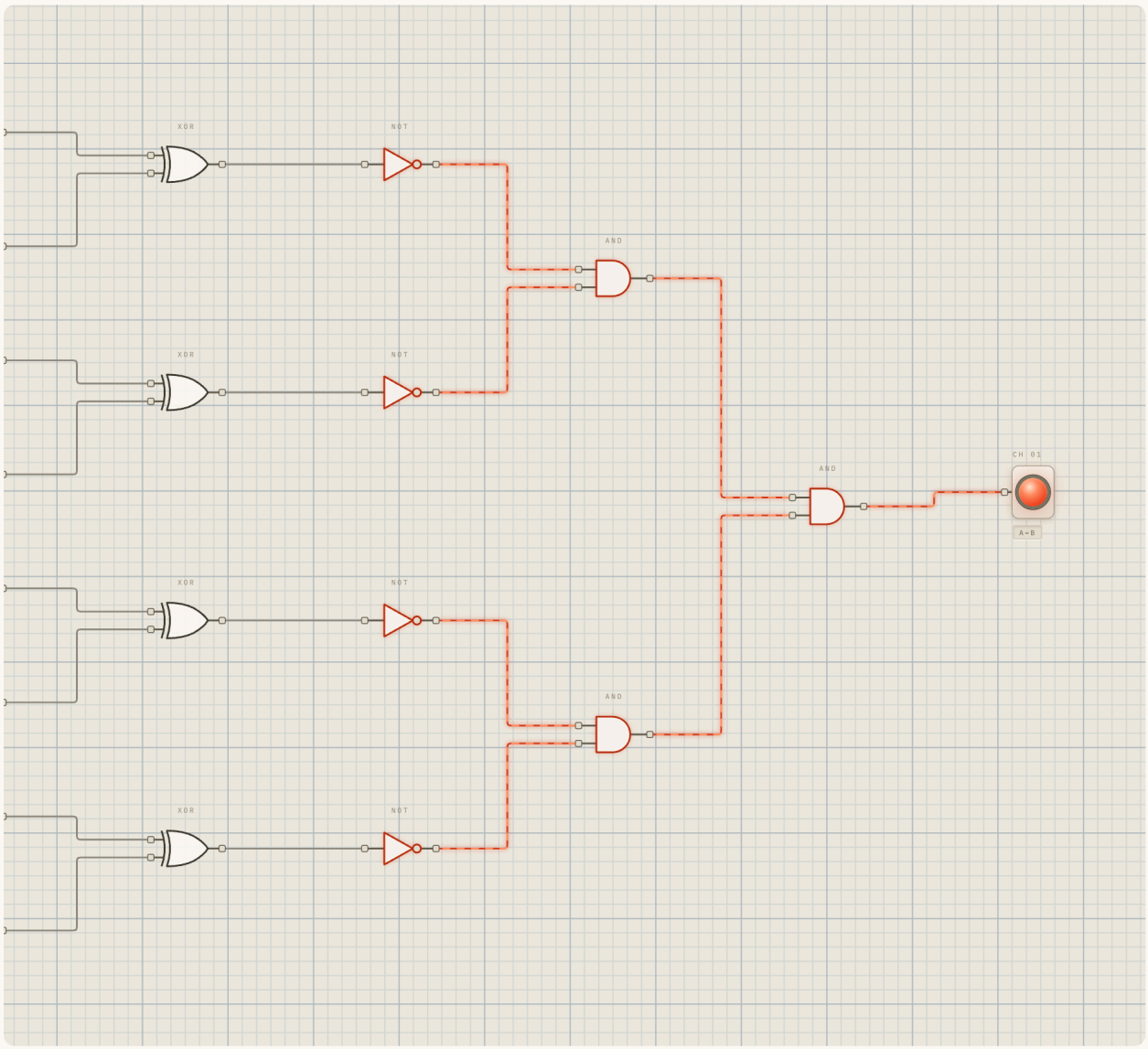
Checking one bit for "same". Recall XOR outputs 1 when its inputs *differ*. Equality wants the opposite — 1 when inputs are the *same*. So put a NOT after the XOR. That combination, XOR-then-NOT, is its own named gate: **XNOR** (exclusive-NOR), and it means "these two bits match." One XNOR per bit position tells you whether that column agrees.

Combining the checks. You now have four "this column matches" signals, and you want to light the output only if *all four* are true. That is the job of AND — specifically an **AND tree** that feeds the four match-signals together. The final output, often labelled **A = B**, lights only when every column matched.

So the comparator is four XNORs (one per bit) feeding an AND tree. Per-bit sameness, combined by *all*.

See it in the bench

Open this: load **4-Bit Equality Comparator** from the library (category **ARITHMETIC**). Two banks of four switches set A and B; an `A = B` LED reports the verdict.



The comparator lighting A=B when both nibbles match

Try it yourself

Predict first, then flip.

Try: 1. Start with everything off — A = 0 and B = 0, which are equal, so **A = B is lit** (0 does equal 0). 2. Flip a **single** switch on one side. Now one column disagrees, the AND tree breaks, and A = B goes **dark** — instantly, on one bit's difference. 3. Match it on the other side (flip the corresponding switch so both sides have that bit). A = B lights again. You can feel the "all columns must agree" rule: one mismatch anywhere is enough to fail.

Recap

- Two numbers are equal when **every bit pair matches**.
- **XNOR** (XOR then NOT) outputs 1 when two bits are the *same* — a per-bit equality check.
- An **AND tree** combines the per-bit checks: A = B lights only if all agree.

Check yourself: Which gate tells you two single bits are the *same*, and what is it made of? *(XNOR — an XOR followed by a NOT.)*

Next: Lesson 25 — The Majority Voter: a circuit that takes a vote.

Lesson 25 — The Majority Voter

> **Part III · Arithmetic** — library circuit (category: BASICS) Before this lesson: the 4-Bit Equality Comparator (Lesson 24). After this: the SR Latch (Lesson 26).

What you will learn

- What a majority voter does and where it is used in the real world.
- How three ANDs and an OR implement "at least two of three".
- The idea of fault tolerance through redundancy.

The idea

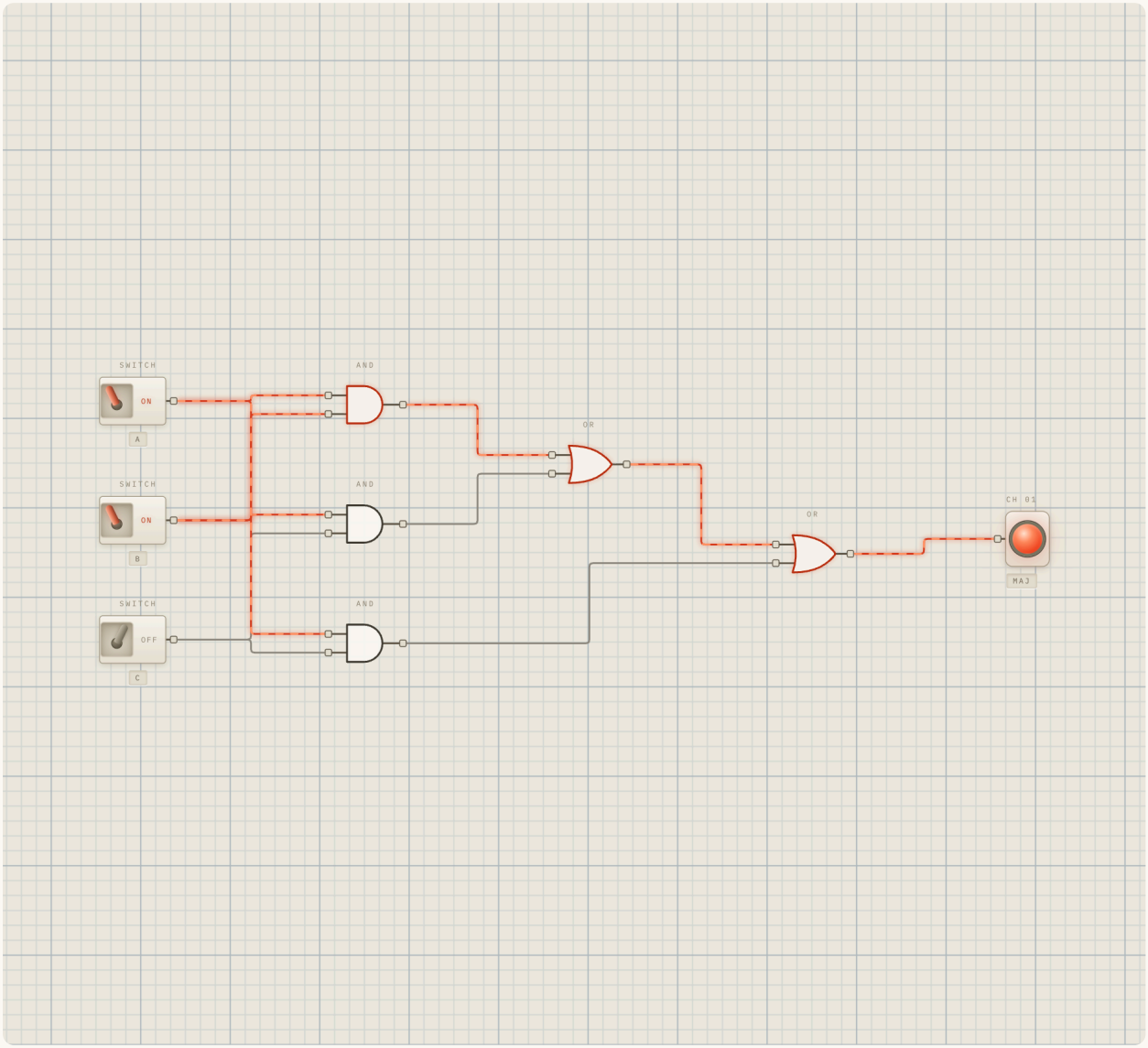
A **3-input majority voter** outputs whatever the **majority** of its three inputs say: if at least two of A, B, C are 1, the output is 1; otherwise it is 0. It takes a vote, and the majority wins.

This sounds simple, but it solves a serious real-world problem: **fault tolerance**. Suppose a critical measurement is so important that you cannot trust a single sensor — what if it fails? The classic solution is to use *three* sensors and a majority voter. If one sensor fails and disagrees with the other two, the voter **outvotes it**, and the system keeps working on the strength of the two that agree. This "triple modular redundancy" flies in spacecraft, runs in safety-critical controllers, and guards against single-point failures everywhere reliability matters.

The implementation is a tidy use of gates you know. "At least two of three are on" is the same as "(A and B) **or** (A and C) **or** (B and C)" — check each *pair*, and if any pair is both-on, that is a majority. So: **three AND gates**, one per pair, feeding an **OR tree**.

See it in the bench

Open this: load **3-Input Majority Voter** from the library (category **BASICS**). Three switches A, B, C feed three pair-checking ANDs and an OR into the MAJ output.



The majority voter with two of three inputs on, output lit

The truth table

A	B	C	MAJ
0	0	0	0
0	0	1	0
0	1	0	0
0	1	1	**1**
1	0	0	0
1	0	1	**1**
1	1	0	**1**
1	1	1	**1**

The output is 1 in exactly the four rows where two or three inputs are high.

Try it yourself

Predict first, then flip.

Try: 1. Turn on any **two** of the three switches — MAJ lights. It does not matter which two. 2. Drop one so only **one** is on — MAJ goes dark. One vote is not a majority. 3. Picture A, B, C as three copies of the same sensor. Turn two on (the sensor reads "1") and then flip the third off as if it had failed and disagreed. MAJ stays lit — the failed sensor was outvoted. That is fault tolerance, working in front of you.

Recap

- A majority voter outputs the value of **at least two of its three** inputs.
- It is built from **three pairwise ANDs feeding an OR**.

- It provides **fault tolerance**: a single disagreeing input is outvoted — the basis of triple-redundant safety systems.

Check yourself: Two sensors say 1 and one says 0. What does the voter output, and why? *(1 — the majority rules and the lone 0 is outvoted.)*

Next: Lesson 26 — The SR Latch: the moment a circuit first gains a memory.

Part IV — Memory

Lesson 26 — The SR Latch

> **Part IV · Memory** — library circuit (category: MEMORY) Before this lesson: the NOR gate (Lesson 12) and the bench's settling rules (Lesson 14). After this: the Gated D Latch (Lesson 27).

What you will learn

- The single most important leap in this book: a circuit that **remembers**.
- How two cross-coupled NOR gates store one bit.
- The meaning of **Set**, **Reset**, and "holding" a value.

The idea

Every circuit so far has been **combinational**: its outputs depend only on its inputs *right now*. Turn the inputs off and the outputs forget everything. This lesson breaks that — it is where a circuit gains a **past**.

The **SR latch** stores one bit. It is built from **two NOR gates cross-coupled**: each gate's output is fed back into the other gate's input. That feedback loop is the whole trick. Thanks to the bench's settling behaviour (Lesson 14), this loop has *two* stable resting states — output Q high, or output Q low — and it will sit in whichever one it was last put into, even after you stop touching the inputs.

It has two control inputs:

- **S (Set):** pulse this high to force Q to **1**.
- **R (Reset):** pulse this high to force Q to **0**.

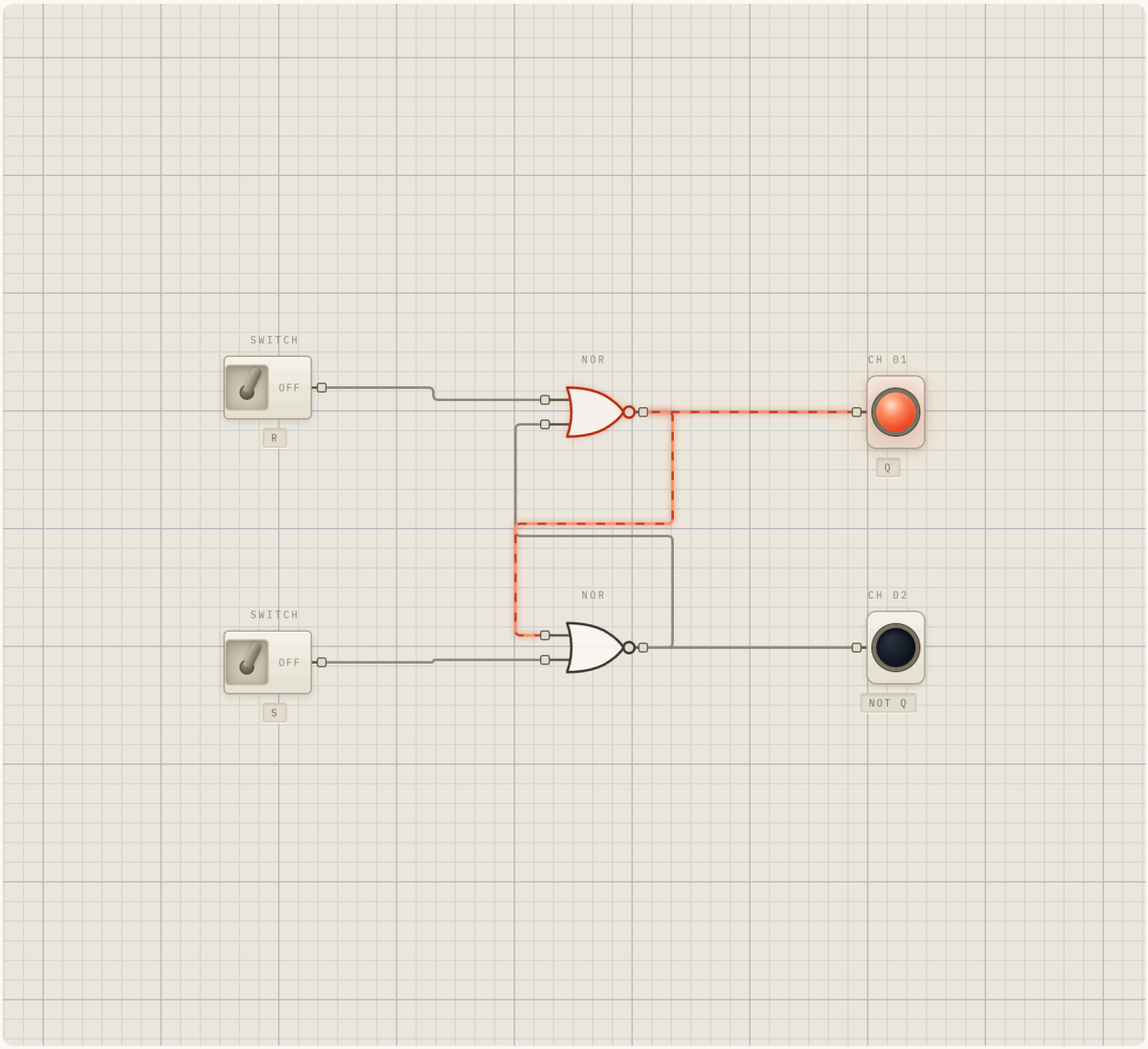
And here is the magic: after you pulse S and then *let go* (S back to 0), Q **stays** at 1. The latch remembers that it was set. Pulse R and release, and Q stays at 0.

The latch holds its bit with no ongoing input — it has memory. (There is also a companion output, NOT Q, which is simply the opposite of Q.)

This is not a simulation trick layered on top; it is the genuine behaviour of cross-coupled feedback, the same physics that makes real latches work. Every register, every byte of RAM, every stored value in the entire machine descends from this circuit.

See it in the bench

> **Open this:** load **SR Latch (NOR)** from the library (category **MEMORY**). Two switches, S and R, feed two cross-coupled NOR gates driving the Q and NOT-Q LEDs.



The SR latch holding Q high after S was pulsed and released

Try it yourself

Predict first, then flip — this is the key experiment of the whole Part.

Try: 1. Pulse **S** on, then off again. Watch Q: it goes to 1 when you set it — and **stays at 1 after you release S**. That "stays" is memory. Sit with it for a moment. 2. Now pulse **R** on, then off. Q drops to 0 and **stays** at 0. The latch now remembers the opposite bit. 3. Pulse S again — back to 1, held. You can store and re-store a bit as many times as you like. The latch always holds the *last* thing you told it. 4. Notice NOT-Q is always the opposite of Q.

Watch out: turning **both** S and R on at once is a contradictory command ("set and reset simultaneously") and is avoided in practice — it is the one input combination an SR latch is not designed for. Explore it if you are curious, but the meaningful operations are set, reset, and hold.

Recap

- The SR latch is the first **sequential** circuit — it has a past, it remembers.
- **Two cross-coupled NOR gates** create a feedback loop with two stable states.
- **S** sets Q to 1, **R** resets it to 0, and the latch **holds** the last value after the input returns to 0.

Check yourself: You pulse S high and then return it to 0. What is Q now, and why? *(Q is 1 and stays 1 — the cross-coupled loop holds the stored bit.)*

Next: Lesson 27 — The Gated D Latch: adding control over when *the latch listens.*

Lesson 27 — The Gated D Latch

> **Part IV · Memory** — library circuit (category: MEMORY) Before this lesson: the SR Latch (Lesson 26). After this: the 4-Bit Register (Lesson 28).

What you will learn

- How to fix the SR latch's two awkward inputs into one clean data input.
- What an **enable** line does — control over *when* memory listens.
- The meaning of "transparent" versus "holding".

The idea

The SR latch works, but it is a little awkward to use: two separate inputs (S and R), and a forbidden both-on combination. What you usually want is simpler: **one data input D** ("here is the bit I want to store") and a separate control that says **"store it now."** The **gated D latch** provides exactly that.

It is built from **four NAND gates** and has two inputs:

- **D (Data):** the bit you want to store — just 0 or 1.
- **EN (Enable):** the control line that decides whether the latch is listening.

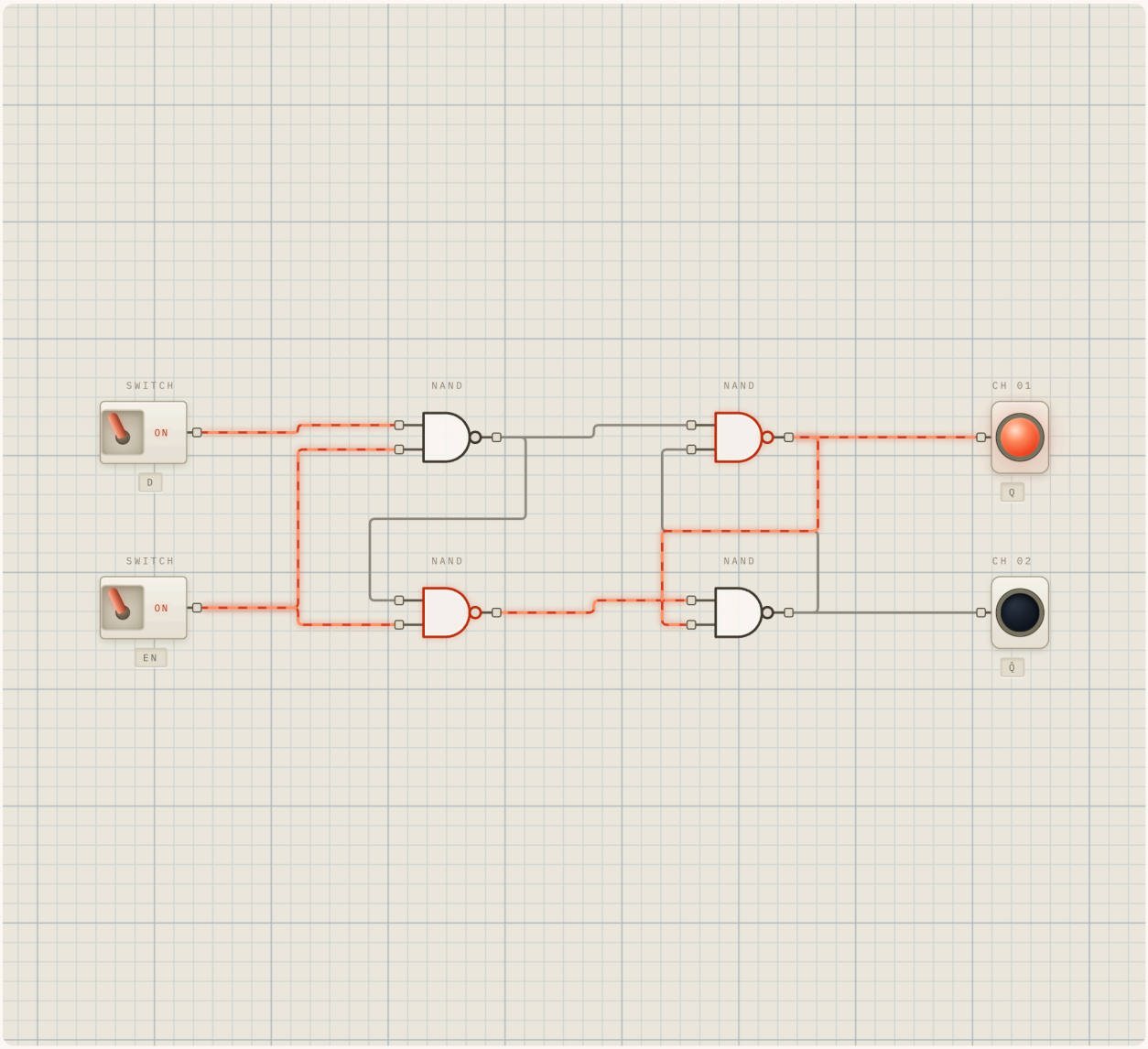
Its behaviour has two modes:

- **While EN is high, the latch is "transparent":** Q simply follows D. Change D and Q changes with it, immediately. The latch is wide open, passing the data through.
- **When EN goes low, the latch "holds":** it freezes whatever value D had at that moment, and from then on ignores D entirely. Q stays put no matter how you wiggle D.

This is a profound upgrade. The enable line gives you **control over time** — you decide the exact moment the latch captures its value. "Watch this input… now stop watching and remember what you saw." That ability to say *when* is what turns raw memory into a useful, controllable building block, and it leads directly to registers (next lesson) and the clocked flip-flops of Part V.

See it in the bench

> **Open this:** load **D Latch (Gated)** from the library (category **MEMORY**). Switches for D and EN feed the four-NAND latch driving Q.



The gated D latch holding its value after EN went low

Try it yourself

Predict first, then flip.

Try: 1. Turn **EN on**. Now toggle **D** back and forth — Q follows it exactly. This is transparent mode: the latch is just passing D through. 2. Set **D = 1** with EN on (so Q = 1), then turn **EN off**. The latch is now holding. 3. With EN still off, flip **D to 0**. Watch Q: **it does not budge.** It is holding the 1 it captured, deaf to D. This is the experiment that defines the latch — predict it, then confirm it. 4. Turn EN back on. Q immediately snaps to whatever D currently is. The latch is listening again.

Recap

- The gated D latch has a clean **D** (data) input and an **EN** (enable) control.
- **EN high → transparent** (Q follows D); **EN low → hold** (Q freezes, D ignored).
- The enable line gives **control over when** a value is captured — the key to useful memory.

Check yourself: EN is low and you change D. What happens to Q? *(Nothing — the latch is holding and ignores D until EN goes high again.)*

Next: Lesson 28 — The 4-Bit Register: four latches sharing one control, storing a whole nibble.

Lesson 28 — The 4-Bit Register

> **Part IV · Memory** — library circuit (category: MEMORY) Before this lesson: the Gated D Latch (Lesson 27). After this: the Clock Divider (Lesson 29).

What you will learn

- How several latches combine to store a whole multi-bit value.
- What a **register** is — the CPU's basic unit of storage.
- Why one shared control line lets a group of bits move together.

The idea

One gated D latch stores one bit. Stack **four** of them side by side, give them four data inputs D0–D3, and wire them all to **one shared LOAD line**, and you have a **4-bit register** — a circuit that stores a whole nibble at once.

The shared control is the key idea. Because all four latches share a single LOAD line:

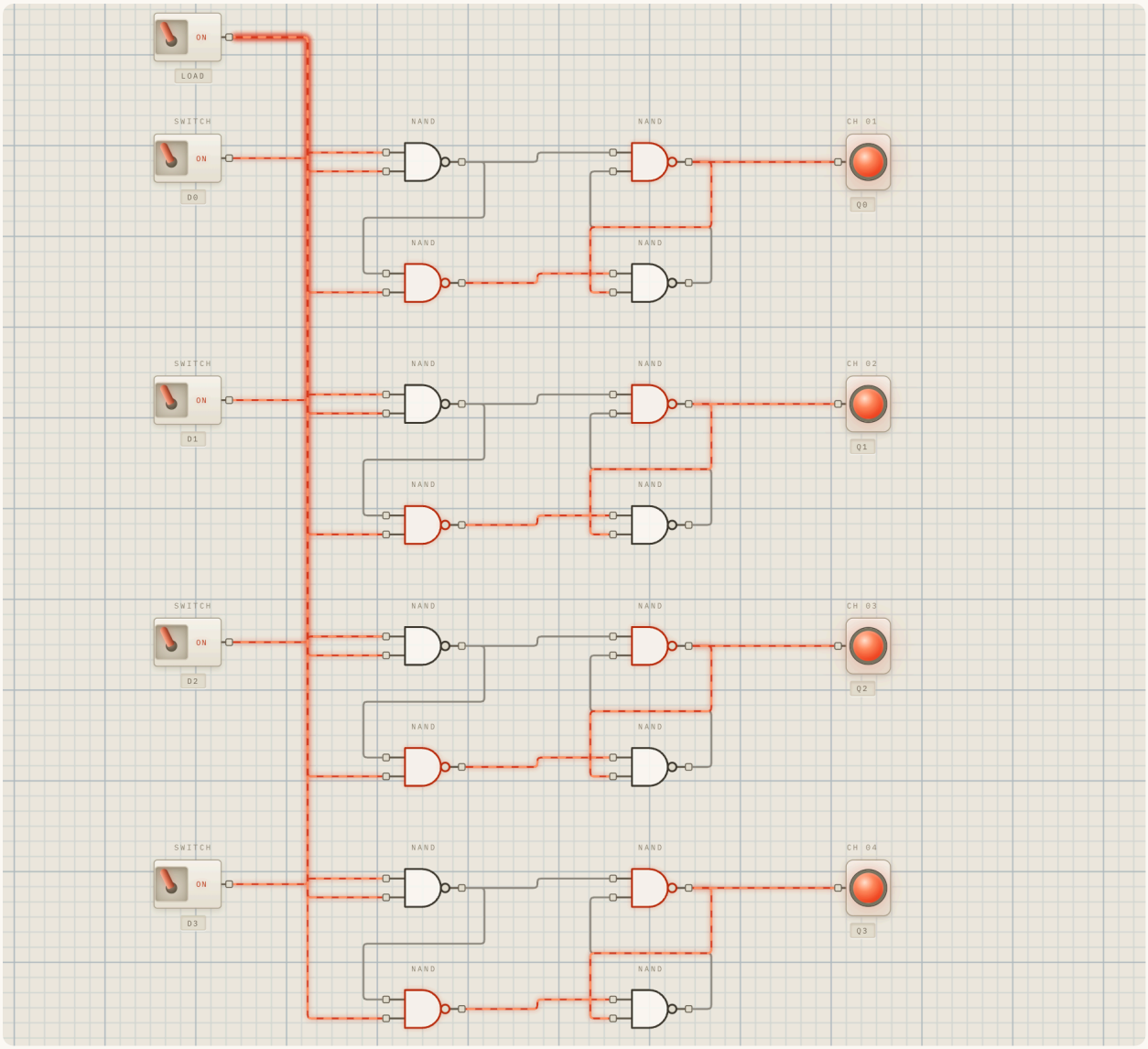
- **While LOAD is high**, all four Q outputs follow their D inputs — the register is transparent, showing whatever you present.
- **When LOAD drops**, all four bits freeze together, capturing the nibble at that instant, and hold it.

A register is therefore just "the gated D latch, four wide, sharing one enable." But it is one of the most important structures in the whole machine. A CPU is full of registers: the accumulator that holds the number you are working on, the program counter that holds your place in the program, the registers that shuttle data between operations. When you hear that a processor "loads a value into a register," this — latches sharing a LOAD line — is the literal hardware that happens.

The shared line is also what lets a group of bits act as a single unit. You do not store a byte one bit at a time; you store all eight (or here, four) **simultaneously**, on one signal. That simultaneity is what makes a register a place where a *number* lives, not just a pile of separate bits.

See it in the bench

> **Open this:** load **4-Bit Register** from the library (category **MEMORY**). Four data switches D0–D3 and one LOAD switch feed four gated D latches driving Q0–Q3.



The 4-bit register holding 1010 after LOAD went low

Try it yourself

Predict first, then flip.

Try: 1. Turn **LOAD on**. Set the data switches to **D = 1010** (D1 and D3 on). The Q outputs mirror it. 2. Turn **LOAD off.** The register now holds 1010. 3. With LOAD off, **scramble all the D switches** to something completely different. The Q outputs do not move — all four bits are held together, frozen at 1010. You stored a number, and it is staying put. 4. Turn LOAD on again — the register immediately captures the new D value. You have just performed "load a new value into the register," exactly as a CPU does.

Recap

- A **register** is several D latches sharing **one LOAD line**, storing a multi-bit value at once.
- **LOAD high → transparent**; **LOAD low → all bits hold together**.
- Registers are the CPU's basic storage — accumulator, program counter, and more are registers.

Check yourself: Why do all four bits of a register capture their values at the same instant? *(They share a single LOAD line, so one signal freezes all four latches together.)*

Next: Lesson 29 — The Clock Divider: adding the clock's heartbeat to memory, and meeting the flip-flop.

Part V — Sequential machines & the clock

Lesson 29 — The Clock Divider (T flip-flop)

> **Part V · Sequential machines & the clock** — library circuit (category: SEQUENTIAL) Before this lesson: the 4-Bit Register (Lesson 28) and the Clock (Lesson 03). After this: the D Flip-Flop (Lesson 30).

What you will learn

- The difference between a latch (level-sensitive) and a flip-flop (edge-triggered).
- What "toggle on every edge" means and why it divides frequency by two.
- The master-slave structure that makes edge-triggering possible.

The idea

The latches of Part IV listened *whenever* their enable was high — they were **level-sensitive**. But circuits that march in step with a clock need something sharper: a memory that updates only at the precise **instant** the clock changes, not during the whole time it is high. That sharper device is a **flip-flop**, and it is **edge-triggered** — it acts on the clock's *edge* (the moment of change) rather than its level.

This lesson's flip-flop is a **T (toggle) flip-flop**. Its behaviour is beautifully simple: on every clock edge, it **flips** its output to the opposite of what it was. High becomes low, low becomes high, once per beat.

Two consequences follow, both worth noticing:

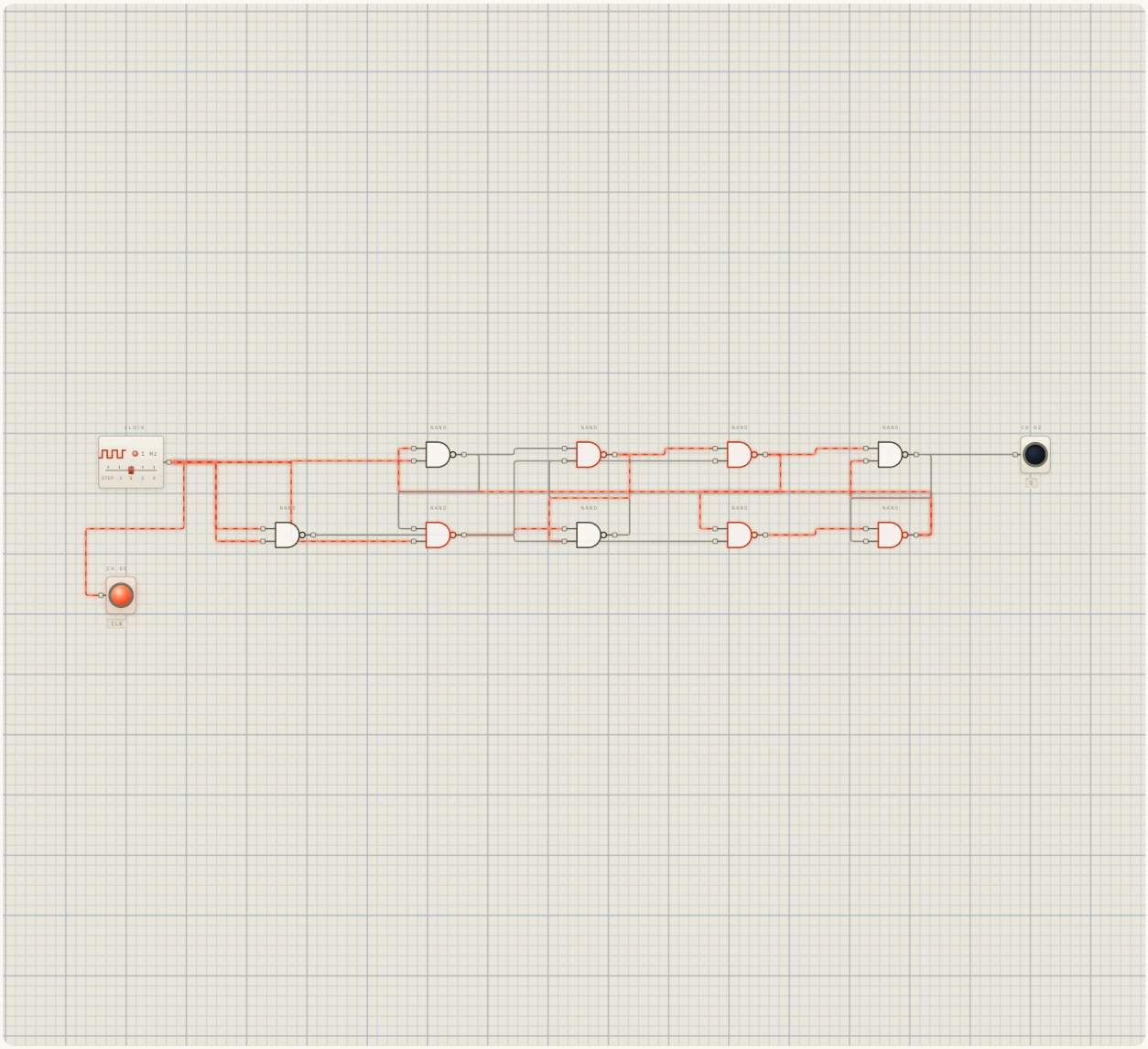
It is a frequency divider. Because Q flips once per clock edge, it completes a full on-off cycle every *two* edges. So Q blinks at exactly **half** the clock's rate. Feed in a clock, get out a signal at half the frequency — hence "clock divider."

Chain several and each halves again (the basis of the counter two lessons from now).

It is built master-slave. To update cleanly on an edge rather than smearing during a level, the flip-flop uses *two* latch stages in series — a "master" that listens while the clock is high and a "slave" that listens while the clock is low. The value can only ripple all the way through at the moment the clock hands off between them: the **edge**. This master-slave pair (here, nine NAND gates) is the standard way edge-triggering is achieved, and feeding $\bar{Q}$ back as the input is what makes it toggle.

See it in the bench

Open this: load **Clock Divider (T Flip-Flop)** from the library (category **SEQUENTIAL**). A clock drives the master-slave T flip-flop; Q drives an LED.



The T flip-flop's Q blinking at half the clock rate

Try it yourself

Try: 1. Watch Q blink. Compare its rhythm to the raw clock — Q changes **half** as often. You are seeing divide-by-two. 2. Click the **clock face** to change its speed. Q's rate tracks it, always at half. 3. Think ahead: if one T flip-flop halves the frequency, what would a *second* one, clocked by the first one's Q, do? (Halve it again — quarter the original.) That stacking is the ripple counter of Lesson 32.

Recap

- A **flip-flop** is **edge-triggered** (acts on the clock's moment of change), unlike a level-sensitive latch.
- A **T flip-flop toggles** Q on every edge, so Q runs at **half** the clock frequency.
- It is built **master-slave** (two latch stages) so it updates cleanly on the edge.

Check yourself: If you feed a 4 Hz clock into a T flip-flop, how fast does Q blink? *(2 Hz — half.)*

Next: Lesson 30 — The D Flip-Flop: edge-triggered storage of a chosen bit.

Lesson 30 — The D Flip-Flop

> **Part V · Sequential machines & the clock** — library circuit (category: SEQUENTIAL) Before this lesson: the Clock Divider (Lesson 29) and the Gated D Latch (Lesson 27). After this: the JK Flip-Flop (Lesson 31).

What you will learn

- How the D flip-flop differs from the D *latch* you met earlier.
- Why "samples on the edge" is more disciplined than "transparent while enabled."
- Why the D flip-flop is the workhorse of synchronous design.

The idea

You already know the gated D latch (Lesson 27): while its enable is high, Q *follows* D continuously. The **D flip-flop** does the same job — store the bit on D — but with crucial discipline: it samples D **only at the clock edge**, and ignores it the rest of the time.

Put the two side by side, because the distinction is the heart of this lesson:

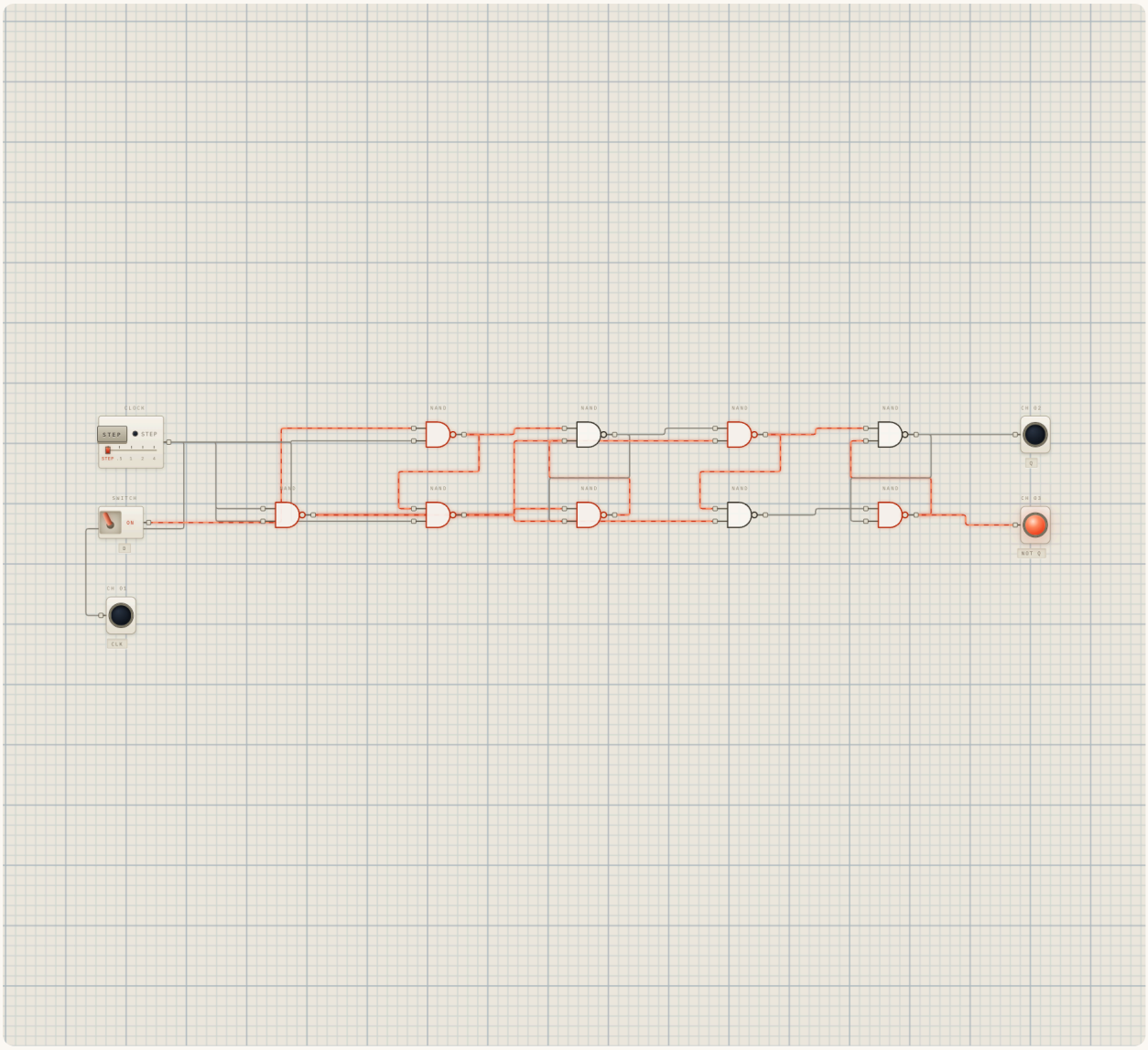
- **D latch (level-sensitive):** while enabled, Q tracks every wiggle of D. Q can change at any moment the enable is high.
- **D flip-flop (edge-triggered):** Q changes *only* at the instant of the clock edge. Between edges, you can wiggle D all you like — the flip-flop is not looking. At the next edge, it takes one clean snapshot of D and holds it until the edge after.

This "one snapshot per beat" behaviour is what makes large synchronous systems possible. When every flip-flop in a machine samples on the *same* clock edge, the entire machine advances in one coordinated step: everybody

reads their inputs at the same instant, everybody updates together, and nothing is half-changed in between. That orderliness is why the edge-triggered D flip-flop — built master-slave from nine NANDs, exactly like the T flip-flop — is the single most-used memory element in real digital design.

See it in the bench

> **Open this:** load **D Flip-Flop (Edge-Triggered)** from the library (category **SEQUENTIAL**). A D switch and a clock feed the master-slave flip-flop; Q drives an LED.



The D flip-flop sampling D only on the clock edge

Try it yourself

Predict first, then flip.

Try: 1. Set **D = 1** and watch across a clock edge — at the next falling edge, Q becomes 1. 2. Now **change D between edges** — flip it to 0 and back to 1 a few times *while watching Q*. Q does **not** react; it is waiting for the edge. This is the defining contrast with the latch, which *would* have followed every change. 3. Leave D at some value and let an edge pass — Q snaps to D's value at that instant, then holds. One snapshot per beat.

Recap

- A D flip-flop stores D but **samples only at the clock edge**, ignoring D between edges.
- The D **latch** is transparent while enabled; the D **flip-flop** takes one snapshot per edge.
- Edge-triggering lets a whole machine advance in **one coordinated step** — why the D flip-flop is the workhorse of synchronous design.

Check yourself: You change D three times between two clock edges. How many times does Q change? *(At most once — only at the edge, capturing D's value at that instant.)*

Next: Lesson 31 — The JK Flip-Flop: one flip-flop with four modes.

Lesson 31 — The JK Flip-Flop

> **Part V · Sequential machines & the clock** — library circuit (category: SEQUENTIAL) Before this lesson: the D Flip-Flop (Lesson 30). After this: the 4-Bit Ripple Counter (Lesson 32).

What you will learn

- How a JK flip-flop packs four behaviours into one device.
- How simple steering logic in front of a D flip-flop creates those modes.
- Why the J=K=1 "toggle" mode connects back to the T flip-flop.

The idea

The **JK flip-flop** is the most versatile of the basic flip-flops. It has two control inputs, **J** and **K**, and on each clock edge it does one of **four** things depending on their combination:

J	K	on the next edge, Q…
0	0	**holds** (stays as it was)
1	0	**sets** (becomes 1)
0	1	**resets** (becomes 0)
1	1	**toggles** (flips to the opposite)

So a single JK flip-flop can hold, set, reset, *or* toggle — all the useful one-bit operations — selectable by two control lines. That flexibility made it a favourite in classic logic design.

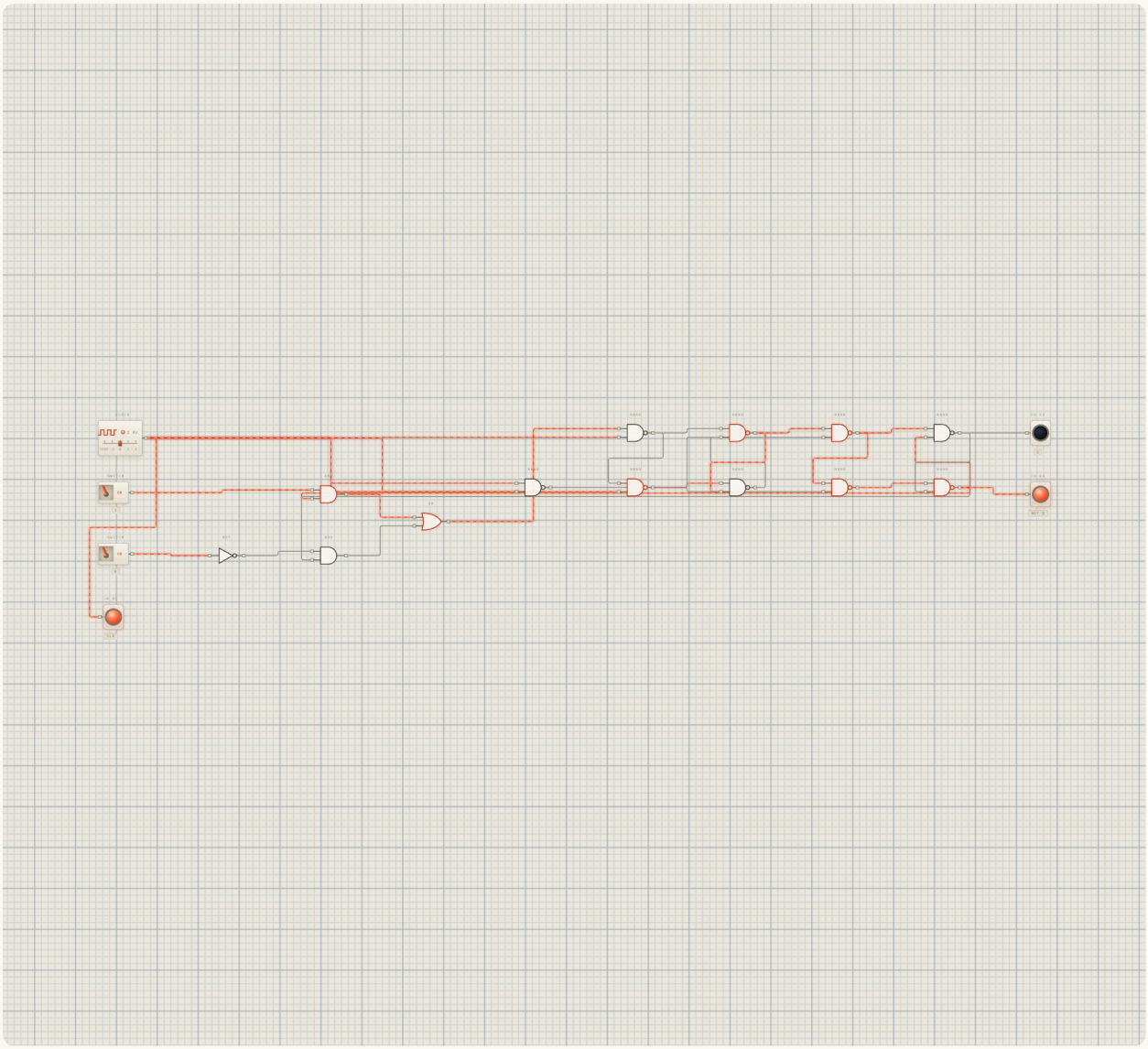
How is it built? Cleverly, and economically: **steering logic in front of a plain D flip-flop**. The circuit computes a D value from J, K and the current Q using the rule `D = (J AND NOT Q) OR (NOT K AND Q)`, then feeds that into the same

master-slave D core you already know. Work through that little formula for each J/K combination and the four modes fall out. You do not have to derive it — the point is that the JK flip-flop is not a new kind of magic, just smart wiring around the D flip-flop from the previous lesson.

Notice the last row: when **J = K = 1**, the JK toggles on every edge — which is exactly the **T flip-flop** behaviour from Lesson 29. The JK contains the T as a special case. The family of flip-flops is more connected than it first appears.

See it in the bench

> **Open this:** load **JK Flip-Flop** from the library (category **SEQUENTIAL**). Switches J and K plus a clock feed the steering logic and the D core; Q drives an LED.



The JK flip-flop in toggle mode with J and K both high

Try it yourself

Predict first, then flip.

Try: 1. **J = 1, K = 0** — across an edge, Q sets to 1 and stays. (Set mode.) 2. **J = 0, K = 1** — Q resets to 0. (Reset mode.) 3. **J = 0, K = 0** — Q holds whatever it was, edge after edge. (Hold mode.) 4. **J = 1, K = 1** — Q toggles on every edge, blinking at half the clock rate. Compare this to the Clock Divider of Lesson 29: identical behaviour. You have found the T flip-flop hiding inside the JK.

Recap

- A JK flip-flop offers **four edge-triggered modes**: hold (00), set (10), reset (01), toggle (11).
- It is built from **steering logic in front of a D flip-flop** — no new magic.
- The **J = K = 1** toggle mode *is* a T flip-flop; the family is interconnected.

Check yourself: What does a JK flip-flop do on each clock edge when J = K = 1? *(Toggles — exactly like a T flip-flop.)*

Next: Lesson 32 — The 4-Bit Ripple Counter: chaining flip-flops to count.

Lesson 32 — The 4-Bit Ripple Counter

> **Part V · Sequential machines & the clock** — library circuit (category: SEQUENTIAL) Before this lesson: the Clock Divider (Lesson 29). After this: the Ring Counter (Lesson 33).

What you will learn

- How chaining toggle flip-flops produces a binary counter.
- Why each bit runs at half the rate of the one before it.
- How to read a counter's value climbing on the hex display.

The idea

In Lesson 29 you saw a single T flip-flop halve a clock's frequency. Now chain four of them: clock the first from the main clock, then clock each subsequent flip-flop from the **previous one's output Q**. The result is a **4-bit binary counter** that counts 0, 1, 2, 3, … up to 15 (F) and wraps around.

Why does chaining toggles produce *counting*? Think about how a binary number increments. The lowest bit flips on every step (0,1,0,1,…). The next bit flips half as often (0,0,1,1,…). The next, half as often again. That is precisely what the chain produces: each flip-flop toggles, and each is clocked by the one below it, so:

- **Q0** runs at half the clock — the ones bit.
- **Q1** runs at half of Q0 — the twos bit.
- **Q2** at half of Q1 — the fours bit.
- **Q3** at half of Q2 — the eights bit.

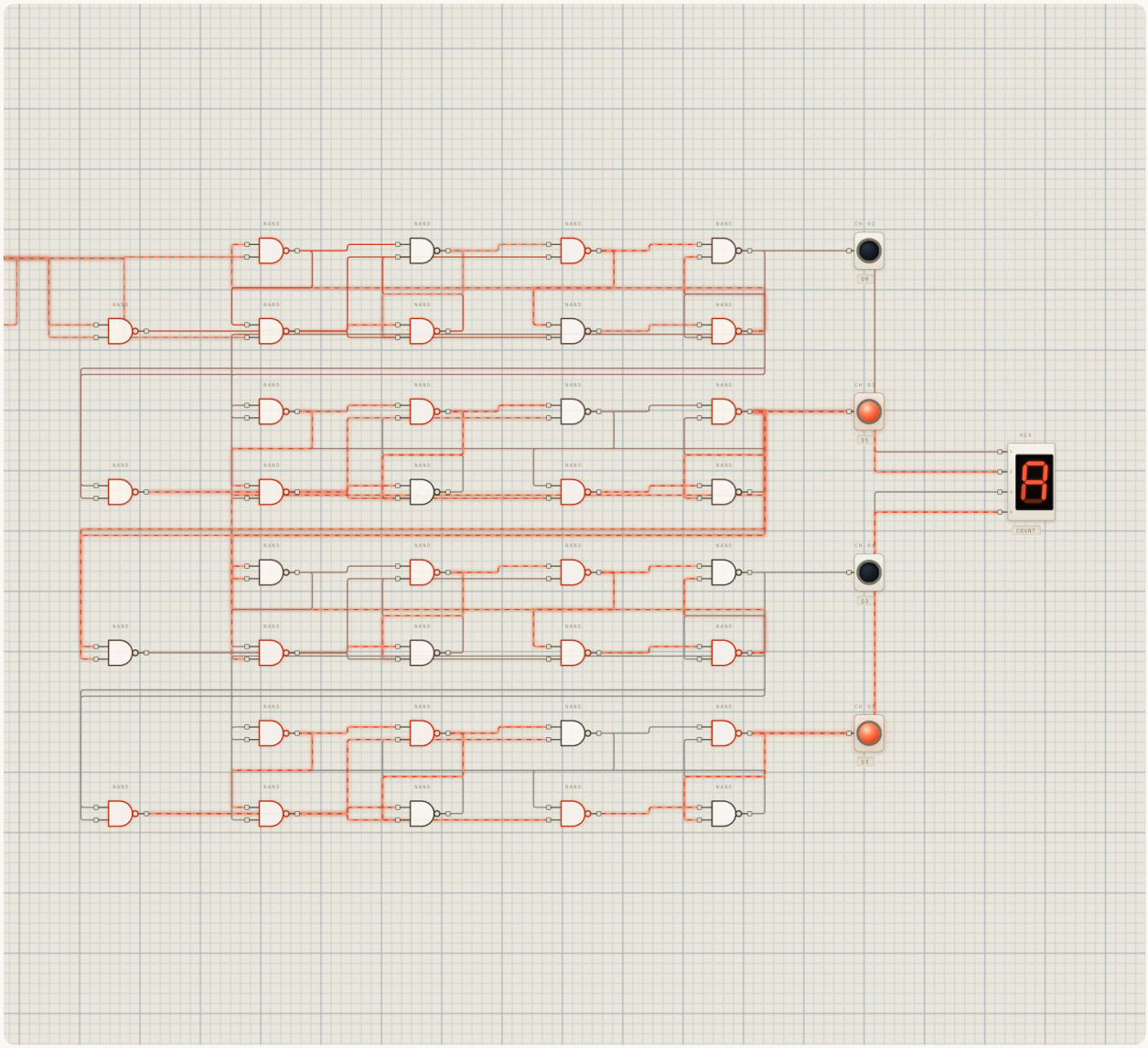
Read Q3 Q2 Q1 Q0 together as a binary number and it counts upward, one per clock tick. The frequency-halving you saw in isolation, chained, *is* binary

counting.

Because each stage is clocked by the previous stage's output, the change "ripples" up the chain (hence **ripple counter**), the same travelling-signal idea you met in the ripple-carry adder. It is simple and beautiful; its only cost is that the bits do not all flip at the exact same instant, which matters in high-speed designs but not to your eye here.

See it in the bench

> **Open this:** load **4-Bit Ripple Counter** from the library (category **SEQUENTIAL**). A clock drives four chained T flip-flops; Q0–Q3 feed a hex display labelled COUNT.



The ripple counter climbing through hex values

Try it yourself

> **Try:** 1. Load it and watch **COUNT** tick upward: 0, 1, 2, 3 … E, F, then wrap to 0. A counter, built from toggles. 2. Watch **Q0** (the ones bit) — it flips every tick. Now watch **Q3** (the eights bit) — it flips only once every eight ticks. Each bit is half the speed of the one before, exactly as binary counting requires. 3. Click the **clock face** to count faster or slower. Slow it right down and you can verify each increment by hand against the binary pattern.

Recap

- Four chained **toggle flip-flops** make a **4-bit binary counter** (0–15, then wrap).
- Each bit runs at **half the rate** of the bit below it — that frequency cascade *is* counting.
- It is a **ripple** counter: each stage clocks the next, so the change travels up the chain.

Check yourself: In a binary counter, how does the rate of the eights bit (Q3) compare to the ones bit (Q0)? *(Q3 flips one-eighth as often — each higher bit is half the rate of the one below.)*

Next: Lesson 33 — The Ring Counter: a different kind of counter that walks a single lit bit.

Lesson 33 — The Ring Counter (running light)

> **Part V · Sequential machines & the clock** — library circuit (category: SEQUENTIAL) Before this lesson: the 4-Bit Ripple Counter (Lesson 32). After this: the Binary Clock full circuit (Lesson 34).

What you will learn

- A second style of counter: one that circulates a single lit bit.
- What "self-starting" and "self-correcting" mean.
- How feedback shapes a circuit's long-term behaviour.

The idea

The ripple counter counted in binary. A **ring counter** counts differently: instead of cycling through binary values, it passes a **single lit bit** along a row of flip-flops, like a baton handed down a line. The German word for this marquee effect — *Lauflicht*, "running light" — captures it perfectly: one lamp lit, chasing along the strip and wrapping back to the start.

Here, **eight D flip-flops** are arranged so that on each clock edge, each one passes its value to the next, and the last wraps around to the first — a closed ring. Drop a single 1 into the ring and it circulates forever: L0, L1, L2 … L7, L0, again.

Two clever properties make this particular ring robust, and they are the real lesson:

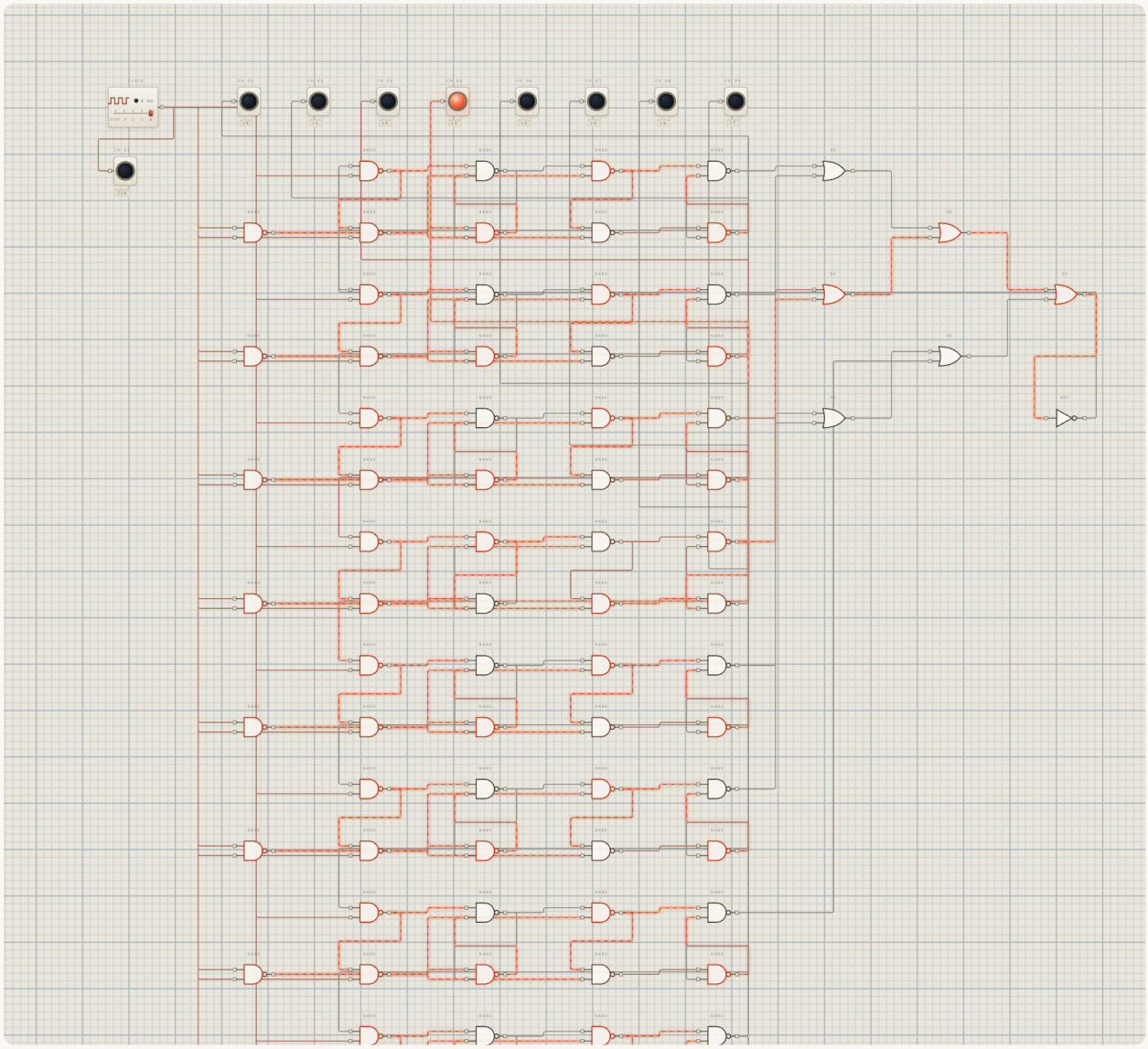
- **Self-starting.** From a cold start every flip-flop is 0 — so where does the first lit bit come from? Stage 0's input is wired to `NOR(Q0…Q6)`, which is 1 exactly when all those stages are empty. So at power-up, with everything empty, the ring *injects its own* first bit. It starts itself with no help.

- **Self-correcting.** If noise ever produced *two* lit bits, that same NOR feedback starves the extra one out over the next few cycles, returning the ring to exactly one circulating bit. The circuit heals itself back to correct behaviour.

These properties come entirely from one feedback wire. It is a small, elegant demonstration that **feedback does not only create memory (as in the latch) — it can also shape and stabilise a circuit's whole behaviour over time.**

See it in the bench

> **Open this:** load **8-LED Running Light (Ring Counter)** from the library (category **SEQUENTIAL**). A clock drives eight chained D flip-flops with the NOR feedback into stage 0; L0–L7 are the LED strip.



The ring counter with one lit LED travelling along the strip

Try it yourself

> **Try:** 1. Load it and watch the single lit LED chase across L0 → L7 and wrap back to L0, one step per clock edge. 2. Note that you did **nothing** to start it — it was all zeros at load, yet a bit appeared and began circulating. That is the self-starting NOR feedback at work. 3. Click the **clock face** to speed the marquee up or slow it down. 4. Compare the two counters you now know: the ripple counter encodes its count as a binary *number* across all bits; the ring counter encodes it as the *position* of one lit bit. Two very different ways to count, both built from flip-flops.

Recap

- A **ring counter** circulates a single lit bit around a closed ring of flip-flops.
- This one is **self-starting** (NOR feedback injects the first bit from an empty state) and **self-correcting** (it starves out any extra bits).
- Feedback shapes long-term behaviour, not just memory.

Check yourself: How does a ring counter represent its current count, compared with a binary ripple counter? *(By the position of a single lit bit, rather than as a binary number across all bits.)*

Next: Lesson 34 — The Binary Clock (full circuit): the machinery behind the instrument from Lesson 07.

Lesson 34 — The Binary Clock (full circuit)

> **Part V · Sequential machines & the clock** — library circuit (category: SEQUENTIAL) Before this lesson: the Ripple Counter (Lesson 32) and the Binary Clock instrument (Lesson 07). After this: the 2:1 Multiplexer (Lesson 35).

What you will learn

- How a real, working 24-hour clock is built entirely from flip-flops and gates.
- How counters chain through a **carry chain** to roll seconds into minutes into hours.
- The payoff of a promise made back in Lesson 07.

The idea

Back in Lesson 07 you met the Binary Clock *instrument* — a pretty lamp display that simply showed the real time, with nothing wired to it. This lesson delivers the promise made there: here is the **full circuit** that actually *generates* a clock's display from scratch, using only the flip-flops and gates you have spent this Part learning.

It is a genuine 24-hour clock built from **17 master-slave D flip-flops** plus next-state logic, all sharing a single **1 Hz clock** (one tick per second). The structure is a stack of counters connected by a **carry chain**, exactly mirroring how time itself rolls over:

- A seconds counter ticks 0→59 on the 1 Hz beat. When it rolls past 59, it **carries** into…
- …the minutes counter, which advances by one. When minutes roll past 59, they carry into…

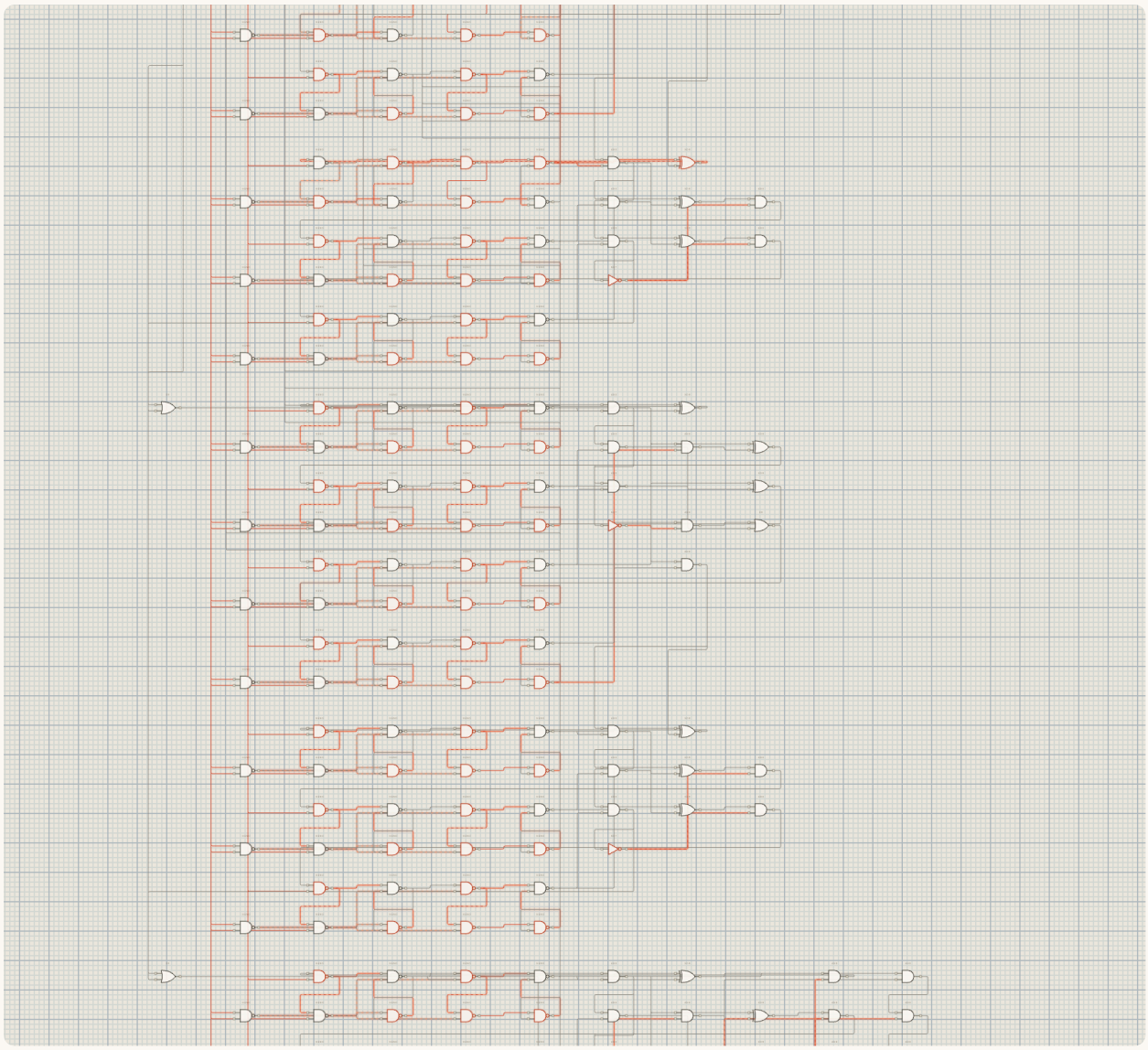
- …the hours counter, which advances by one and forces the famous rollover **23 → 00** at midnight.

This is the same carry idea you met in the ripple-carry adder and the ripple counter, now used to make a clock count in proper hours, minutes and seconds (as binary-coded-decimal digits — read each LED column bottom-up as 1-2-4-8 and add). It powers up at 00:00:00, just like the real desk gadget, and while you hold a SET button, each tick fast-advances that field so you can set the time.

This is a milestone lesson. Everything in Parts IV and V — memory, the clock, edge-triggering, counting, carry chains — converges into one circuit that does something genuinely useful and recognisable. You are looking at the *how* behind the *what* you admired in Lesson 07.

See it in the bench

> **Open this:** load **Binary Clock (Full Circuit)** from the library (category **SEQUENTIAL**). A 1 Hz clock drives six BCD digit counters (the 17 flip-flops) feeding a 20-LED BCD matrix, with SET M and SET H controls and a speed fader.



The full gate-level binary clock counting real seconds

Try it yourself

Try: 1. Load it and watch the seconds column advance once per second — real counting, produced by flip-flops you understand, not a pre-made instrument. 2. Crank the **speed fader** up (e.g. to 4 Hz) and watch the carry chain in action: seconds roll into minutes, minutes into hours. Watch a minute roll over and push the minutes column up by one. 3. Hold **SET M** to fast-advance the minutes, or **SET H** for hours, and set it to a time you choose. 4. Open it beside the **Binary Clock (BCD)** instrument from Lesson 07. Same readout — but now you can see every flip-flop that produces it. That is the whole spirit of this bench: the box opens, and there is no magic inside.

Recap

- The full binary clock is a real 24-hour clock made of **17 D flip-flops** plus logic on a **1 Hz clock**.
- Counters chain through a **carry chain**: seconds → minutes → hours, rolling 23→00 at midnight.
- It is the *how* behind the Lesson 07 instrument — Parts IV and V converging into one useful machine.

Check yourself: What makes the minutes column advance? *(A carry out of the seconds counter when it rolls past 59 — the carry chain linking the counters.)*

Next: Lesson 35 — The 2:1 Multiplexer: we turn to routing signals.

Part VI — Routing & codes

Lesson 35 — The 2:1 Multiplexer

> **Part VI · Routing & codes** — library circuit (category: ROUTING & CODES)
> Before this lesson: the AND, OR and NOT gates (Part I). After this: the 4:1 Multiplexer (Lesson 36).

What you will learn

- What a multiplexer (mux) does — choosing one of several inputs.
- How a NOT, two ANDs and an OR build a data selector.
- Why muxes are the plumbing that routes data through a computer.

The idea

A **multiplexer** — "mux" for short — is a **data selector**: it has several data inputs, one **select** input, and one output, and it routes *the chosen* input through to the output. A 2:1 mux is the smallest: two data inputs (A and B), one select line (SEL), one output.

The rule:

- When **SEL = 0**, the output follows **A**.
- When **SEL = 1**, the output follows **B**.

It is a signal-controlled switch — but where a physical switch is thrown by your hand, a mux is thrown by *another signal*. That is what makes it powerful: a circuit can decide, on the fly, which of two data streams to pass along. This is the fundamental act of **routing**, and it is everywhere inside a computer — choosing which register feeds the ALU, choosing whether the program counter advances normally or jumps, choosing which value to write back. Wherever a machine "picks one of these," a mux is doing it.

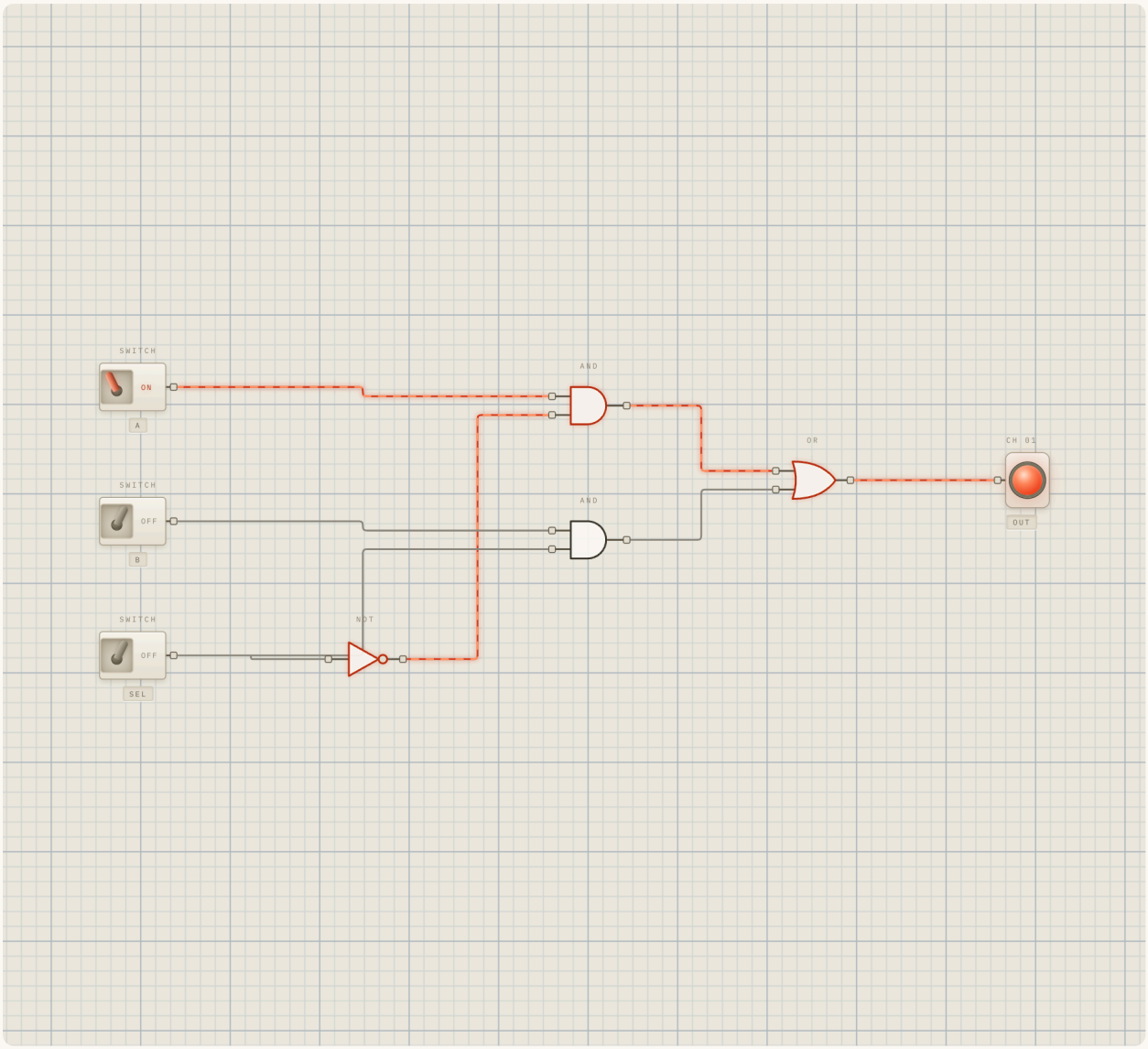
The construction is a clean little use of three gate types you know:

- A **NOT** inverts SEL, so you have both SEL and NOT-SEL available.
- **AND gate A** passes input A only when NOT-SEL is 1 (i.e. SEL = 0).
- **AND gate B** passes input B only when SEL is 1.
- An **OR** merges the two AND outputs — and since only one AND is ever "open" at a time, the OR cleanly carries whichever input was selected.

"Gate the one you want, block the other, merge." That pattern — used here for two inputs — scales up to any number, as the next lesson shows.

See it in the bench

> **Open this:** load **2:1 Multiplexer** from the library (category **ROUTING & CODES**). Switches A, B and SEL feed the NOT/AND/AND/OR network into an OUT LED.



The 2:1 mux routing the selected input to the output

Try it yourself

Predict first, then flip.

Try: 1. Set **A = 1** and **B = 0**. With **SEL = 0**, OUT follows A → lit. Flip **SEL to 1**, and OUT now follows B → dark. You just rerouted the output by changing only the select line. 2. Reverse the data: **A = 0, B = 1**, and sweep SEL again. OUT always tracks whichever input SEL points at. 3. Hold SEL fixed and toggle the *unselected* input. OUT does not react — the mux is ignoring the channel it did not pick. That indifference is exactly what makes it a clean selector.

Recap

- A **multiplexer** routes one of several data inputs to the output, chosen by a **select** line.
- A 2:1 mux: **SEL = 0 → A**, **SEL = 1 → B**.
- Built from a **NOT, two ANDs and an OR** — "gate the chosen input, block the other, merge."
- Muxes are the routing plumbing of a CPU.

Check yourself: A 2:1 mux has A = 1, B = 0, SEL = 1. What is OUT? *(0 — SEL = 1 selects B, which is 0.)*

Next: Lesson 36 — The 4:1 Multiplexer: choosing one of four with a two-bit select.

Lesson 36 — The 4:1 Multiplexer

> **Part VI · Routing & codes** — library circuit (category: ROUTING & CODES) Before this lesson: the 2:1 Multiplexer (Lesson 35). After this: the 2-to-4 Decoder (Lesson 37).

What you will learn

- How a mux scales from two inputs to four.
- Why two select bits address four channels.
- The relationship between number of select lines and number of inputs.

The idea

A 2:1 mux chose between two inputs with one select line. A **4:1 multiplexer** chooses among **four** data inputs (D0, D1, D2, D3) — and to address four things you need **two select bits**, S1 and S0, read together as a 2-bit number 00, 01, 10, 11:

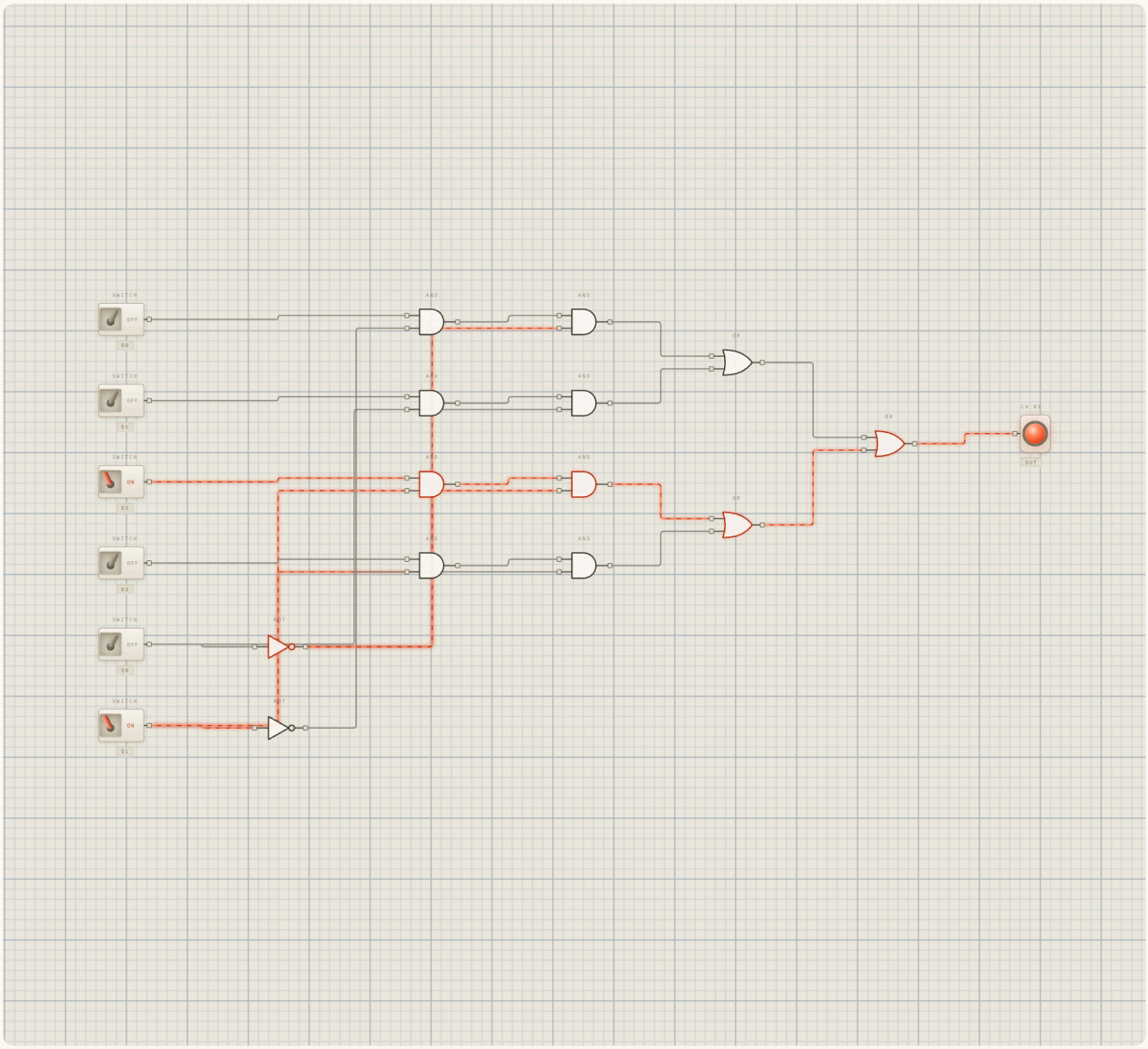
- **S1 S0 = 00** → output follows **D0**
- **S1 S0 = 01** → output follows **D1**
- **S1 S0 = 10** → output follows **D2**
- **S1 S0 = 11** → output follows **D3**

This reveals a tidy and important pattern: **n select bits address 2^n inputs.** One select bit chose between 2; two bits choose among 4; three bits would choose among 8; and so on. The select lines are simply a binary *address* naming which input to pass through. This doubling is a rhythm you will see again and again in addressing — including how memory picks one location out of many (Part VIII).

The construction extends the 2:1 pattern directly: each of the four inputs gets an AND gate that opens only for its own select combination (built from S1, S0 and their inverses), and an OR tree merges the four lanes. Exactly one AND is open at a time, so the OR passes the selected channel cleanly. "Gate the chosen one, block the rest, merge" — just wider.

See it in the bench

> **Open this:** load **4:1 Multiplexer** from the library (category **ROUTING & CODES**). Four data switches D0–D3 and two select switches S1, S0 feed the AND/OR network into OUT.



The 4:1 mux selecting channel 2 with S1 S0 = 10

Try it yourself

Predict first, then flip.

Try: 1. Switch on **only D2**. Set **S1 = 1, S0 = 0** (that is binary 10 = channel 2). OUT lights — you selected the one channel that is on. 2. Now leave the data switches and step the select **S1 S0 through 00, 01, 10, 11**. OUT lights only at 10, because only D2 is on. The select bits are walking an address across the four inputs. 3. Turn on a different data input and find the select value that reaches it. You are learning to *address* one of four things with two bits.

Recap

- A **4:1 mux** selects one of **four** inputs using **two** select bits (a 2-bit address).
- **n select bits address 2^n inputs** — the doubling rule of addressing.
- Same construction as the 2:1, widened: gate the chosen channel, block the rest, OR them together.

Check yourself: How many select bits does a mux need to choose among 8 inputs? *(Three, because $2^3 = 8$.)*

Next: Lesson 37 — The 2-to-4 Decoder: the mux's mirror image — one address, one hot output.

Lesson 37 — The 2-to-4 Decoder

> **Part VI · Routing & codes** — library circuit (category: ROUTING & CODES)
> Before this lesson: the 4:1 Multiplexer (Lesson 36). After this: 4-Bit Parity (Lesson 38).

What you will learn

- What a decoder does — turning an address into a single selected line.
- The meaning of **one-hot** output.
- How decoders drive chip-select and address logic.

The idea

A multiplexer took many inputs and an address, and passed *one* input through. A **decoder** is its mirror image: it takes just an **address** and lights up *exactly one* of its many output lines — the one the address names. No data flowing through; it simply *points*.

A **2-to-4 decoder** takes a 2-bit address (A1 A0) and has four outputs (Y0, Y1, Y2, Y3). It activates the single output whose number matches the address:

- **A1 A0 = 00** → Y0 high (only)
- **A1 A0 = 01** → Y1 high (only)
- **A1 A0 = 10** → Y2 high (only)
- **A1 A0 = 11** → Y3 high (only)

Exactly one output is high at any time — a pattern called **one-hot** (one line "hot," the rest cold). It also has an **enable (EN)** input that gates everything: with EN low, *all* outputs stay dark regardless of the address.

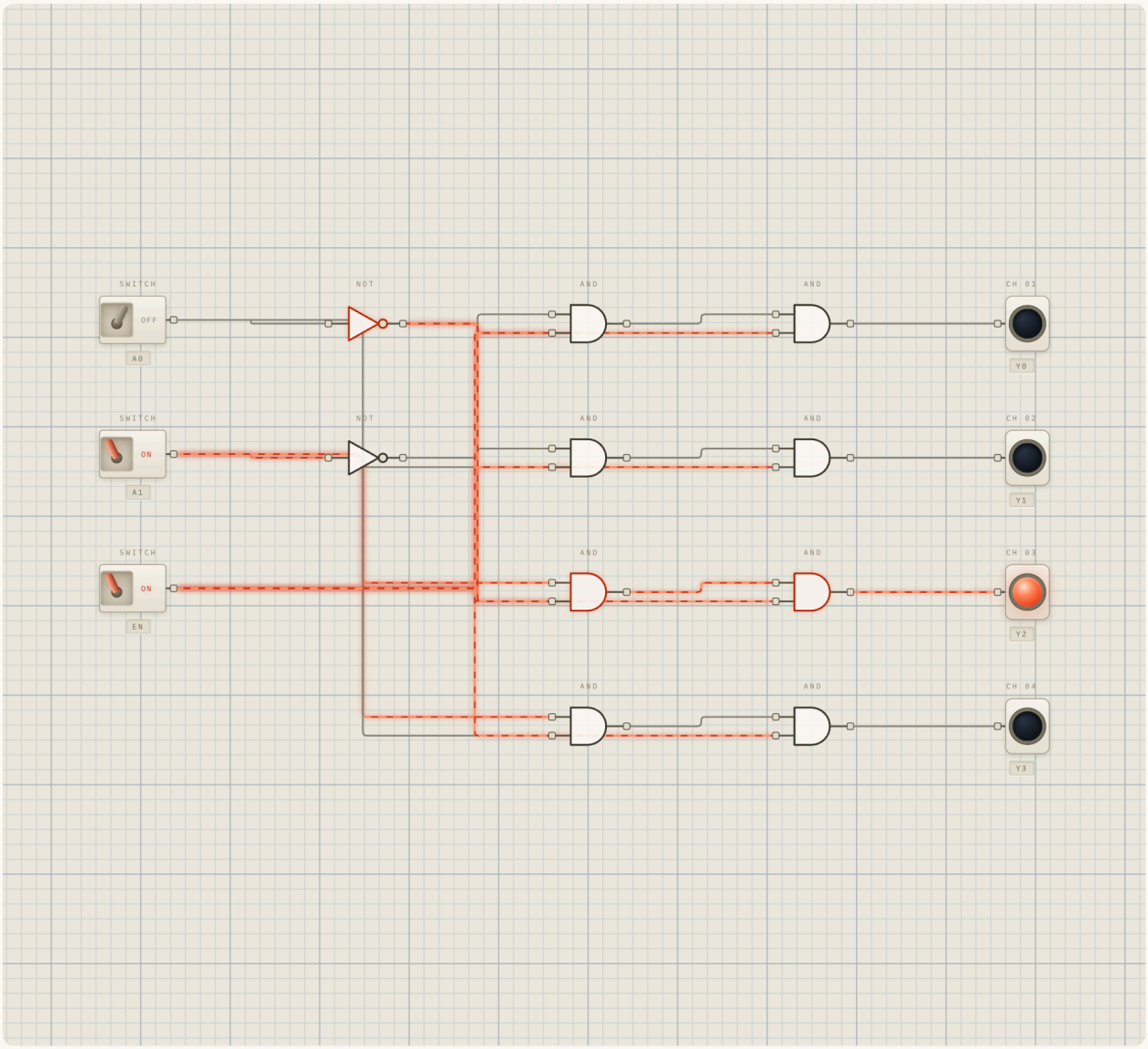
Where is this used? Everywhere a system must select *one of many* things by number. The most important example is **memory addressing**: an address

arrives, and a decoder activates the one memory location (or the one chip) it names, while all others stay quiet. The "chip-select" lines in real computers are decoder outputs. You will see this exact role when RAM is built from gates in Lesson 44 — a decoder turns the address into the single word-line that gets read or written.

The construction: three NOTs supply the inverted address bits, and each output is an AND of the right combination of address bits (and EN). Each output AND recognises exactly one address.

See it in the bench

Open this: load **2-to-4 Decoder** from the library (category **ROUTING & CODES**). Switches A1, A0 and EN feed the NOT/AND network into Y0–Y3.



The 2-to-4 decoder lighting Y2 for address 10

Try it yourself

Predict first, then flip.

Try: 1. Turn **EN on**. Now count the address **A1 A0 through 00, 01, 10, 11** and watch the lit output walk Y0 → Y1 → Y2 → Y3. Exactly one is ever hot — the one the address names. 2. Turn **EN off**. Every output goes dark, whatever the address says. The enable is a master gate over the whole decoder. 3. Hold the address fixed and toggle EN. The selected line blinks on and off while its neighbours stay dark. That EN-gated single line is exactly a memory chip-select.

Recap

- A **decoder** turns an **address** into a **one-hot** output: exactly one line high, naming the addressed item.
- A 2-to-4 decoder maps a 2-bit address to one of four outputs; **EN** gates them all.
- Decoders drive **address and chip-select** logic — central to memory (Lesson 44).

Check yourself: With EN high and address A1 A0 = 11, which output is hot? *(Y3 — and only Y3.)*

Next: Lesson 38 — 4-Bit Parity: a single bit that guards against errors.

Lesson 38 — 4-Bit Parity

> **Part VI · Routing & codes** — library circuit (category: ROUTING & CODES)
> Before this lesson: the XOR gate (Lesson 13). After this: Binary to Gray Code (Lesson 39).

What you will learn

- What a parity bit is and how it catches errors.
- How an XOR tree counts "oddness" across several bits.
- Where parity is used in real systems.

The idea

When data travels — across a wire, through memory, over a link — bits occasionally get corrupted. A **parity bit** is the simplest possible guard against this: a single extra bit that records whether the number of 1s in the data is **odd or even**. If a single bit flips in transit, the oddness changes, and the parity check notices that something is wrong.

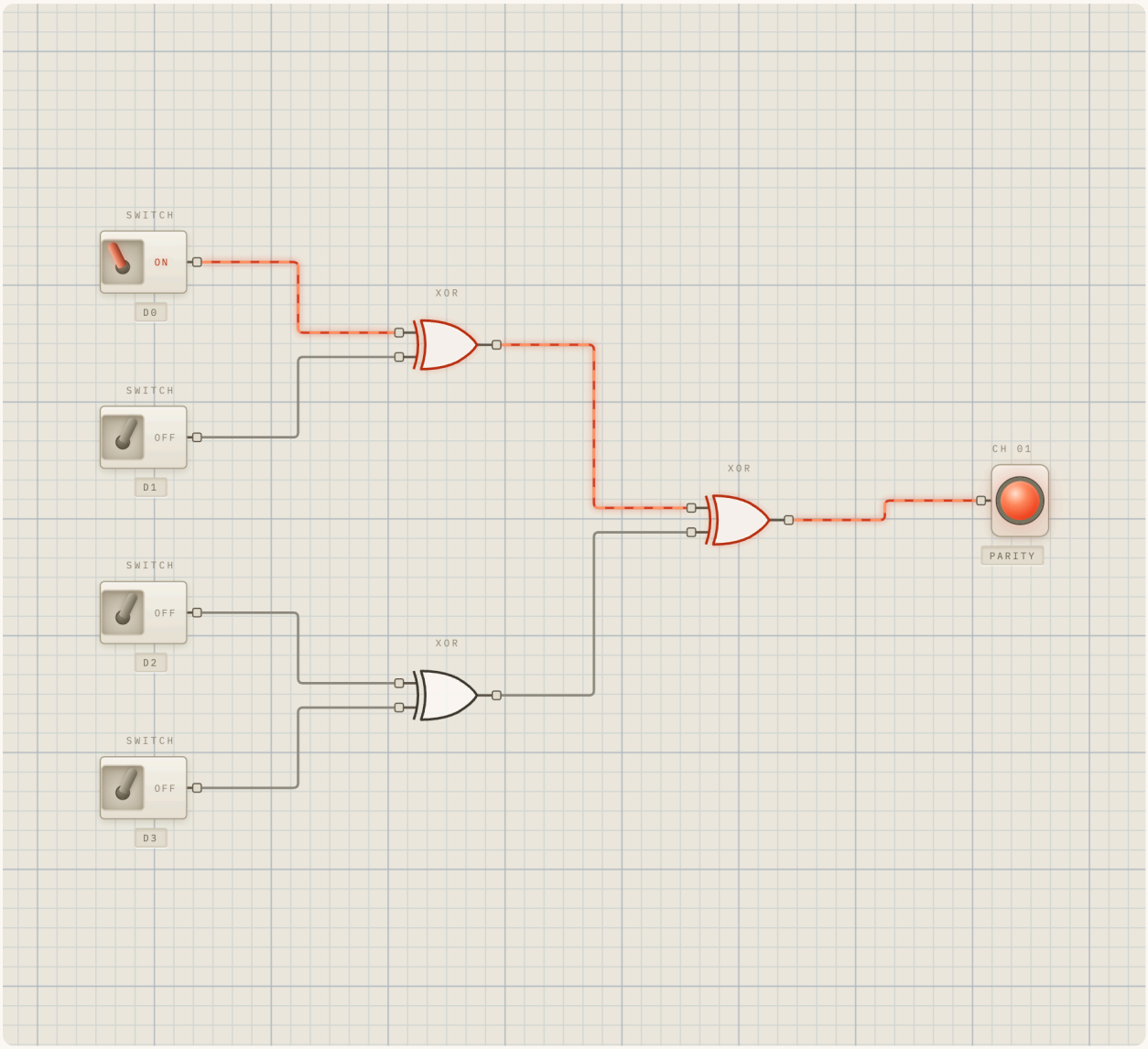
This circuit computes **even parity** over four data bits D0–D3: its single PARITY output is **1 when an odd number of the inputs are high**. (Pair it with the data and the *total* number of 1s becomes even — hence "even parity," the check bit that makes the whole group even.)

The mechanism is a lovely use of XOR. Recall XOR outputs 1 when its two inputs *differ* — which is the same as saying it computes the "odd-ness" of two bits. Cascade XORs into a **tree** — XOR the bits together in pairs, then XOR the results — and the final output is 1 exactly when an **odd** number of *all* the inputs were 1. An XOR tree is an odd-counter. That is the whole circuit: fold four bits down to one "is the count odd?" bit.

Parity is the humble workhorse of error detection: it has guarded memory chips, serial ports, and communication links for decades. It cannot *fix* an error, and it misses errors that flip two bits at once, but as a cheap first line of defence — one gate-tree, one bit — it is everywhere.

See it in the bench

> **Open this:** load **4-Bit Even Parity** from the library (category **ROUTING & CODES**). Four switches D0–D3 feed an XOR tree into the PARITY LED.



The 4-bit parity circuit lighting PARITY for an odd number of inputs

Try it yourself

Predict first, then flip.

Try: 1. Start with all off (zero 1s — even). PARITY is dark. Turn on **one** input — now one 1, which is odd — PARITY lights. 2. Turn on a **second** input — two 1s, even again — PARITY goes dark. Keep going: every single flip you make toggles PARITY, because each flip changes the count's oddness by one. 3. That toggle-on-every-flip behaviour *is* the error-detection property: if exactly one data bit got corrupted in transit, the parity would no longer match — and the receiver would know.

Recap

- A **parity bit** records whether the number of 1s is odd or even, catching single-bit errors.
- An **XOR tree** computes oddness: PARITY is 1 when an odd number of inputs are high.
- Parity is a cheap, ubiquitous first line of error **detection** (it cannot correct, and misses double flips).

Check yourself: Three of the four inputs are high. Is PARITY lit? *(Yes — three is odd.)*

Next: Lesson 39 — Binary to Gray Code: a different way to count where only one bit changes at a time.

Lesson 39 — Binary to Gray Code

> **Part VI · Routing & codes** — library circuit (category: ROUTING & CODES)
> Before this lesson: the XOR gate (Lesson 13). After this: the Hex Display Demo (Lesson 40).

What you will learn

- What Gray code is and the one special property that defines it.
- How a chain of XOR gates converts binary to Gray.
- Why this code matters for rotary encoders and other physical sensors.

The idea

Ordinary binary counting has a hidden hazard: between some consecutive values, **several bits change at once**. Going from 3 (`011`) to 4 (`100`), all three bits flip simultaneously. In a physical device reading those bits — say a rotary position sensor — the bits never flip at *exactly* the same instant, so for a fleeting moment the reading could be garbage (`111`? `000`?). At a boundary, that glitch could read a wildly wrong position.

Gray code solves this elegantly: it is an ordering of values in which **consecutive numbers differ in exactly one bit**. Only ever one bit changes from each value to the next. So a sensor reading Gray code can never momentarily show a wild value at a boundary — at worst it is briefly one step behind. This is why Gray code is beloved of **rotary encoders**, shaft-position sensors, and similar hardware where a clean transition matters.

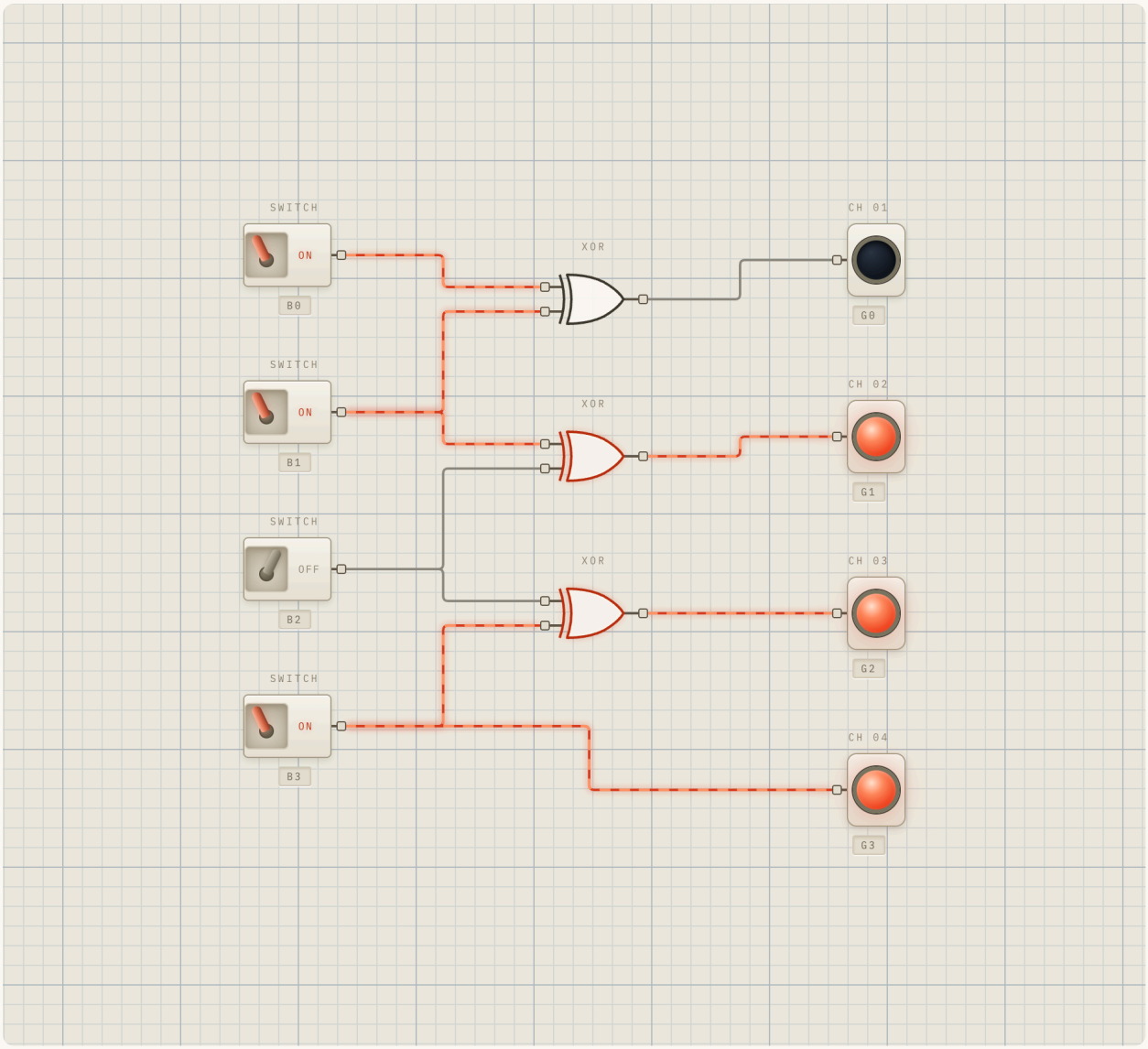
Converting a 4-bit binary value to Gray code turns out to need almost nothing — just **XOR**:

- The top Gray bit (G3) is the same as the top binary bit (B3), passed straight through.
- Each lower Gray bit is the **XOR of two adjacent binary bits**: G2 = B3 XOR B2, G1 = B2 XOR B1, G0 = B1 XOR B0.

A simple chain of XOR gates, each comparing neighbouring binary bits. XOR appears yet again — here as the tool that detects "where the binary value changes between adjacent bit positions," which is exactly what produces the one-bit-at-a-time Gray ordering.

See it in the bench

Open this: load **Binary → Gray (4-Bit)** from the library (category **ROUTING & CODES**). Four binary switches B0–B3 feed the XOR chain into Gray outputs G0–G3.



The binary-to-Gray converter translating a value through its XOR chain

Try it yourself

Predict first, then flip.

Try: 1. Set the binary input to **1011** (B0, B1, B3 on). Read the Gray outputs: **1110**. (G3 = B3 = 1; G2 = B3⊕B2 = 1⊕0 = 1; G1 = B2⊕B1 = 0⊕1 = 1; G0 = B1⊕B0 = 1⊕1 = 0.) 2. Now count the **binary** input slowly upward — 0000, 0001, 0010, 0011, 0100 … — and watch the **Gray** outputs. Each single step of the binary count changes exactly **one** Gray output bit. That one-bit-at-a-time property is the entire point of the code. 3. Find the binary step where the *most* binary bits change at once (e.g. 0111 → 1000, all four flip). Notice the Gray output still changes only one bit. The glitch-prone moment in binary is smooth in Gray.

Recap

- **Gray code** orders values so that consecutive numbers **differ in exactly one bit**.
- This avoids the multi-bit-change glitches of plain binary — ideal for rotary encoders and position sensors.
- Conversion is an **XOR chain**: each Gray bit is the XOR of two adjacent binary bits (top bit passes through).

Check yourself: What is the defining property of Gray code? *(Consecutive values differ in exactly one bit.)*

Next: Lesson 40 — The Hex Display Demo: reading four bits as a single hex digit.

Lesson 40 — The Hex Display Demo

> **Part VI · Routing & codes** — library circuit (category: ROUTING & CODES) Before this lesson: the 7-Segment Display (Lesson 06). After this: Splitters and Mergers (Lesson 41).

What you will learn

- How four weighted switches map to a single hex digit (revisiting Lesson 06 in depth).
- A solid feel for reading binary nibbles as hexadecimal.
- Why hex is the everyday shorthand for binary.

The idea

You met the 7-segment display as a component back in Lesson 06. This short lesson is the hands-on companion: a circuit built purely to let you *practise* reading four bits as one hex digit until it becomes second nature.

The setup is four switches, each labelled by its **binary weight** — **B1, B2, B4, B8** (worth 1, 2, 4, and 8). The display adds the weights of whichever switches are on and shows the total as a single hexadecimal digit, 0 through F. So:

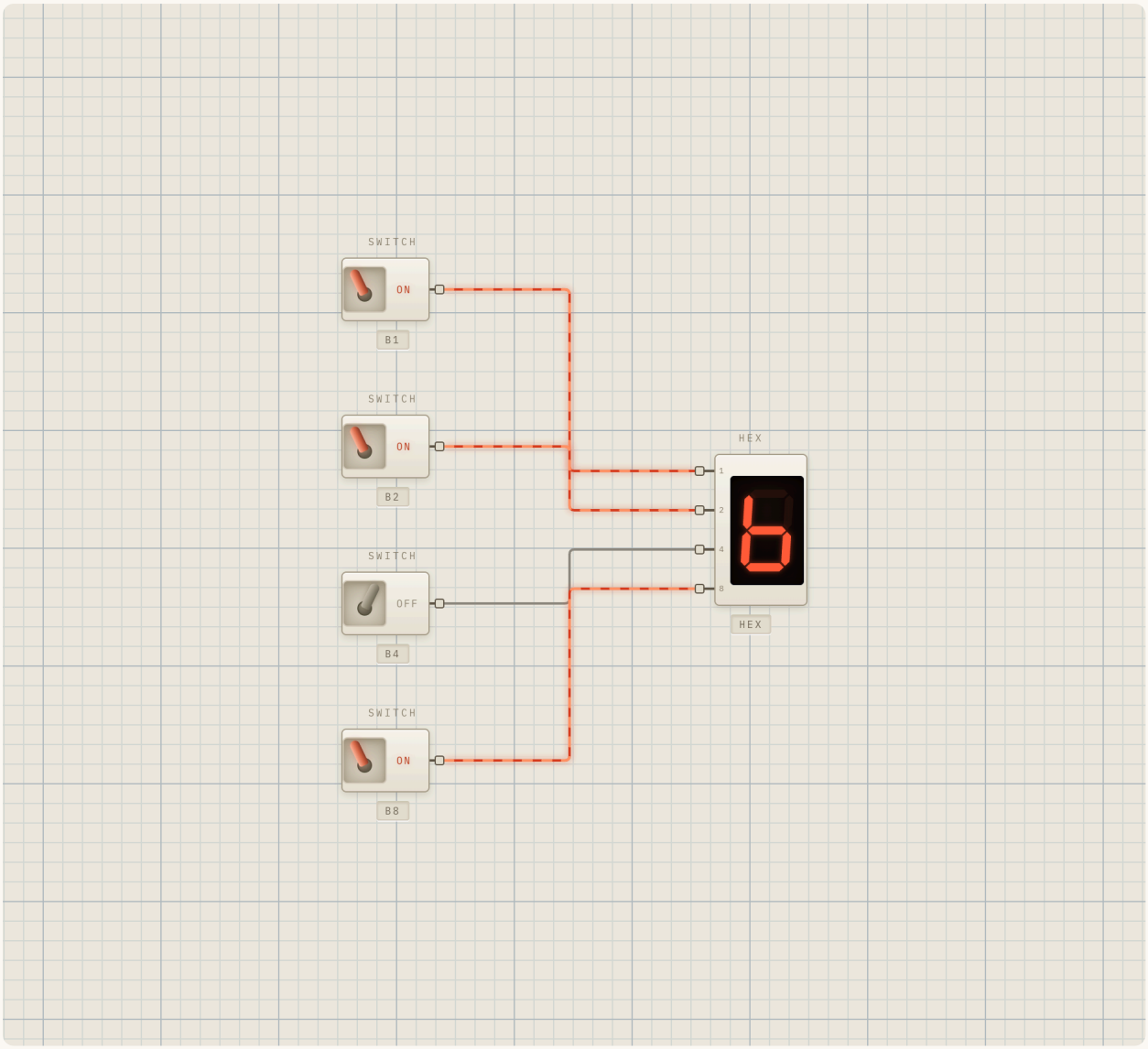
- `B1` alone → 1
- `B2 + B1` → 3
- `B8 + B2 + B1` → 11 → shown as **b**
- all four → 15 → shown as **F**

Why does this matter enough for its own lesson? Because **hexadecimal is how humans actually read binary.** Four bits — a nibble — collapse neatly into one hex character (0–F), and a byte into just two. Rather than squint at `10110010`, an engineer reads `B2`. Every memory address, every byte you will

inspect in the RAM and ROM of Part VIII, every opcode in the CPU lessons, is shown in hex. Getting fluent at "four bits ↔ one hex digit" now will pay off in every lesson that follows.

See it in the bench

> **Open this:** load **Hex Display Demo** from the library (category **ROUTING & CODES**). Four weighted switches B1, B2, B4, B8 drive one 7-segment display.



The hex display reading b from weights 8, 2 and 1

Try it yourself

Predict first, then flip.

Try: 1. Turn on **B8, B2, B1** (leave B4 off). Add the weights: 8 + 2 + 1 = 11 → display shows **b**. 2. Add **B4** so all four are on: 8 + 4 + 2 + 1 = 15 → **F**, the largest single hex digit. 3. Now drill it: count from 0 to F by switching the weights to make each value in turn — 0, 1, 2, 3 (B2+B1), 4, 5 (B4+B1) … up to F. By the time you reach F, reading a nibble as hex should feel automatic. 4. Recognise the letters: 10 = **A**, 11 = **b**, 12 = **C**, 13 = **d**, 14 = **E**, 15 = **F**. Hex borrows six letters to name the values past 9.

Recap

- Four weighted bits (1, 2, 4, 8) sum to a single **hex digit** 0–F.
- **Hexadecimal** is the standard human shorthand for binary: one digit per nibble, two per byte.
- Fluency here pays off in every later lesson, where addresses, bytes and opcodes are all shown in hex.

Check yourself: Which switches are on to display **A** (decimal 10)? *(B8 and B2 — 8 + 2 = 10.)*

Next: Lesson 41 — Splitters and Mergers: bundling many wires into one bus.

Part VII — Buses & larger structure

Lesson 41 — Splitters and Mergers

> **Part VII · Buses & larger structure** — native symbols Before this lesson: the adders (Part III). After this: the 4-Bit Adder from Chips (Lesson 42).

What you will learn

- What a **bus** is and why wide circuits need one.
- What splitters and mergers do.
- How bundling wires keeps large circuits readable.

The idea

By now you have built circuits with eight separate wires carrying eight separate bits — the 8-bit adder had sixteen input wires alone. That works, but it becomes a tangle. Real designs group related bits into a single fat strand called a **bus**: instead of eight loose 1-bit wires, you draw one `/8` bus that carries all eight bits together.

A bus is purely an organisational idea — it does not change the logic, only the tidiness. To move between "eight separate bits" and "one bus," you use two adapter components:

- A **MERGER** takes several individual bits and **bundles** them into one bus. Eight 1-bit wires go in; one `/8` bus comes out.
- A **SPLITTER** does the reverse: it takes a bus and **fans it back out** into its individual bits. One `/8` bus goes in; eight 1-bit wires come out.

Think of a bus as a multi-lane cable and these adapters as the connectors at each end — a merger gathers the loose wires into the cable, a splitter breaks them back out where you need them individually. Between the two, the eight

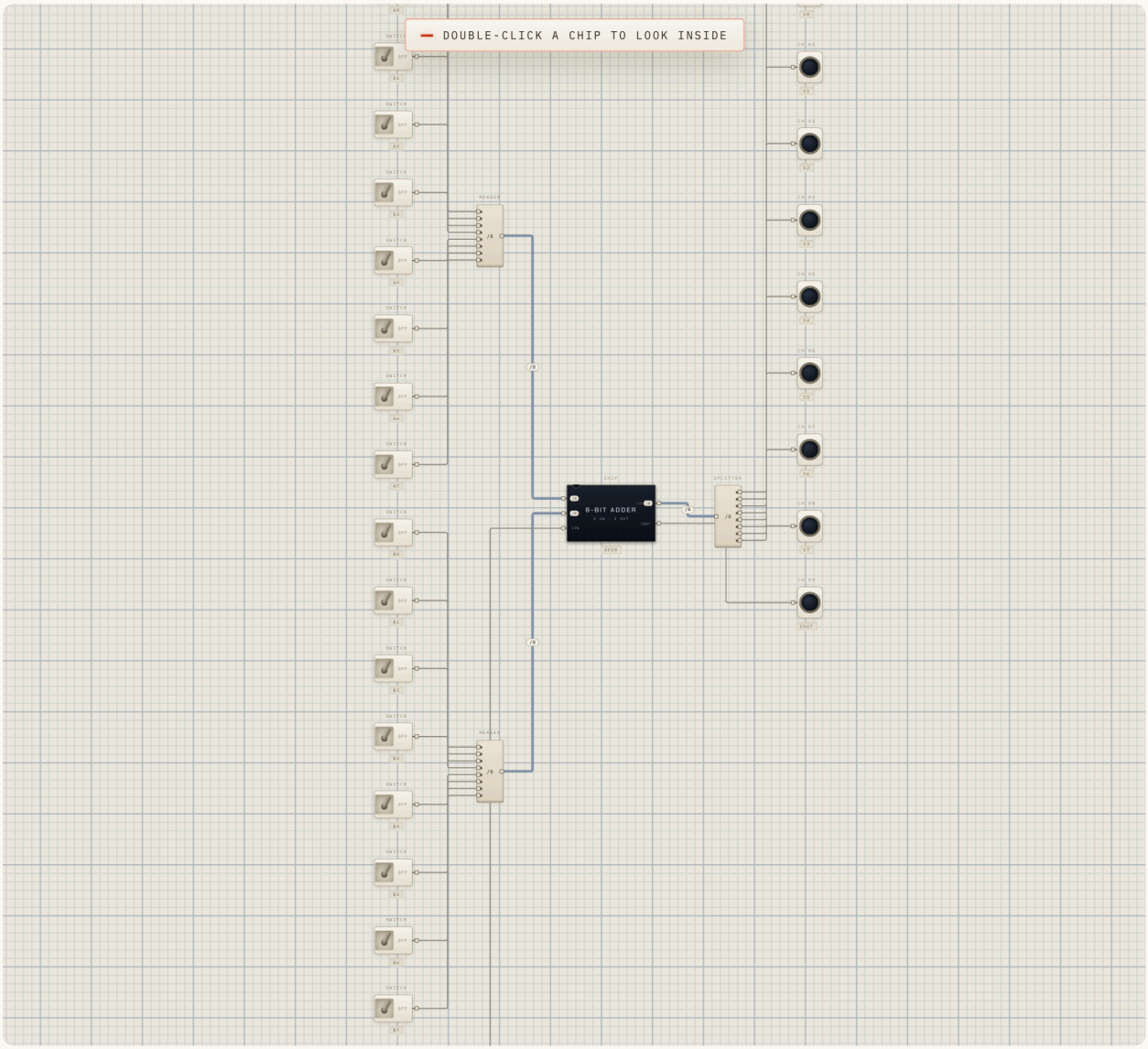
bits travel as one neat line you can route across the canvas without a thicket of parallel wires.

> **A note on how the bench handles this:** splitters, mergers, and buses are a *drawing* convenience. Before the simulator actually computes anything, every bus is expanded back into its individual 1-bit wires behind the scenes — the engine itself only ever deals in single bits. So a bus is real and useful for *you*, the builder, while the underlying logic stays exactly as honest and bit-precise as everything in Part I. Nothing is hidden; the wires are just bundled for readability.

This matters most as circuits get big. The next two lessons rebuild adders you already know using buses and packed chips — and you will see the same logic become dramatically easier to read.

See it in the bench

> **Open this:** the clearest place to see splitters and mergers at work is the **8-Bit Adder (Bus)** preset (Lesson 43), where two operand banks are merged onto buses and the result is split back out. For now, just look for the two adapter shapes: a merger gathering bits into a fat line, and a splitter fanning a fat line back into bits.



A merger bundling bits into a bus and a splitter fanning them back out

Try it yourself

> **Try:** 1. Open the **8-Bit Adder (Bus)** preset and find where eight switch bits enter a **merger** and become one bus. Trace that bus across to where a **splitter** breaks the result back into eight LED bits. 2. Compare its tidiness to the plain **8-Bit Adder** from Lesson 21, with its sixteen loose input wires. Same logic, far less clutter. 3. Remember the key reassurance: the bus is for *your* eyes. Underneath, it is still the same individual bits you have understood all along.

Recap

- A **bus** bundles several related bits into one fat strand for readability.
- A **MERGER** gathers individual bits into a bus; a **SPLITTER** fans a bus back into bits.
- Buses are a drawing convenience — the engine expands them back to single bits, so the logic stays bit-precise and honest.

Check yourself: Which adapter turns eight separate bit-wires into one `/8` bus? *(A merger.)*

Next: Lesson 42 — The 4-Bit Adder from Chips: packing a circuit into a reusable chip you can open.

Lesson 42 — The 4-Bit Adder from Chips

> **Part VII · Buses & larger structure** — library circuit (category: ARITHMETIC) Before this lesson: the 4-Bit Adder (Lesson 20) and bus adapters (Lesson 41). After this: the 8-Bit Adder on a Bus (Lesson 43).

What you will learn

- What a **chip** is: a circuit packed into a reusable, sealed block.
- The single most important idea in hardware design: **hierarchy**.
- How to open a chip and watch its internal gates work — the bench's signature move.

The idea

In Lesson 20 you built a 4-bit adder from loose gates — four full adders, each made of XORs, ANDs and an OR, all spread across the canvas. It worked, but it was a lot of gates to look at. This lesson rebuilds the *exact same adder* using **chips**.

A **chip** is a circuit you have **packed into a single sealed block** with labelled pins, so it can be dropped in and reused like a single component. Here, one full adder — the two-half-adders-plus-OR circuit of Lesson 19 — is packaged into a **FULL ADDER** chip. The 4-bit adder then becomes simply **four instances of that one chip**, chained carry-to-carry. Instead of dozens of scattered gates, you see four tidy labelled blocks.

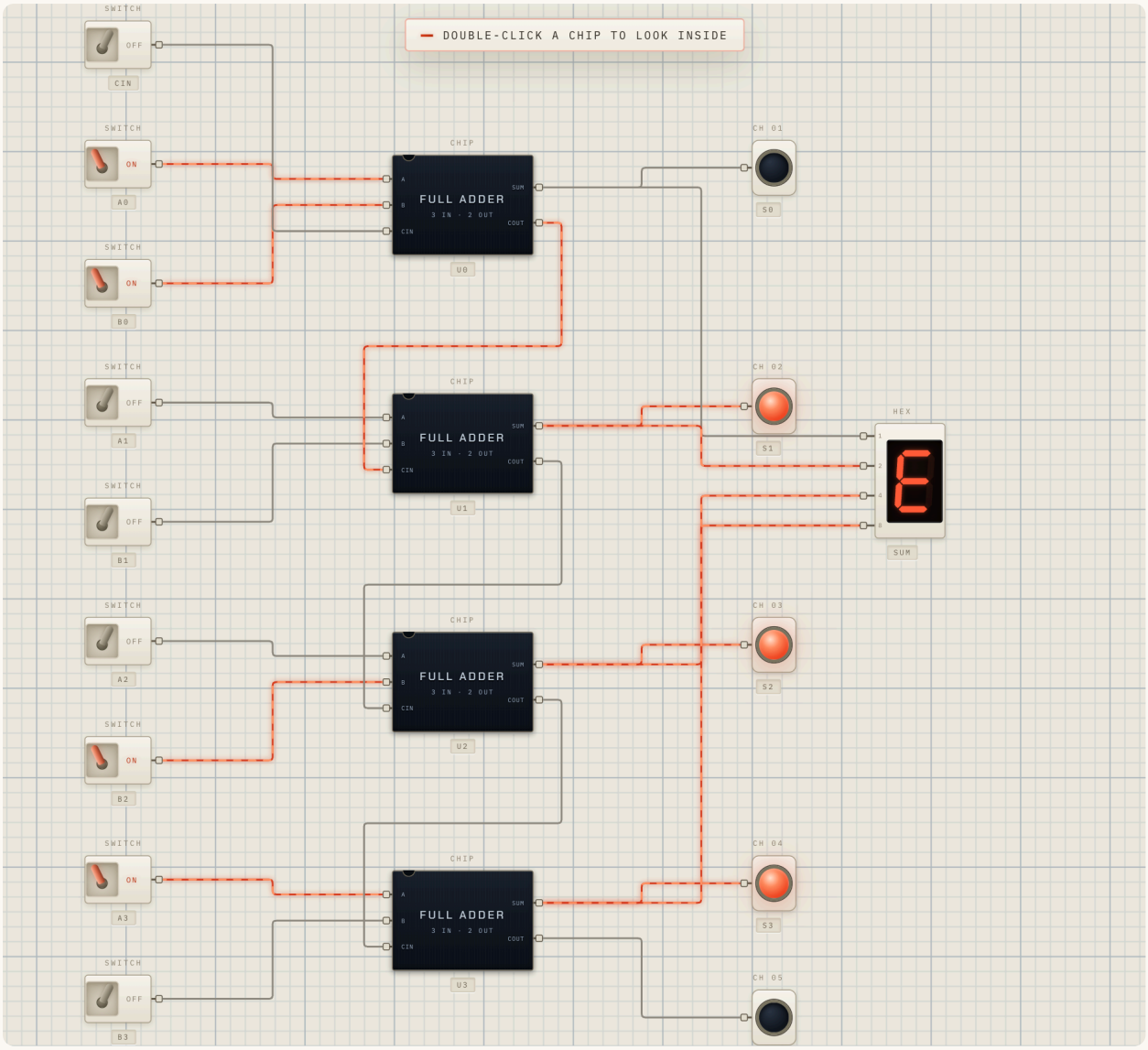
This is **hierarchy**, and it is *the* central idea that makes complex hardware possible. You design a thing once, prove it works, seal it into a chip, and from then on treat it as a single building block — without ever again worrying about its insides. Then you build bigger chips out of *those*, and bigger ones out of *those*. A CPU is not designed gate-by-gate by one person holding

20,000 gates in their head; it is built in layers, each layer trusting the sealed correctness of the layer below. You have actually been relying on this all along — every preset that "uses a full adder" was using this idea informally. Now it is explicit.

And here is the bench's promise, made concrete: **the chip is sealed, but not opaque.** You can **double-click any chip instance and drop inside it** to watch its own gates working live, with real signals flowing — then step back out. The abstraction is real, but it is never a lie. There is no level at which "and then magic happens." Open the FULL ADDER chip and you find the honest XORs and AND and OR you built in Lesson 19, computing away.

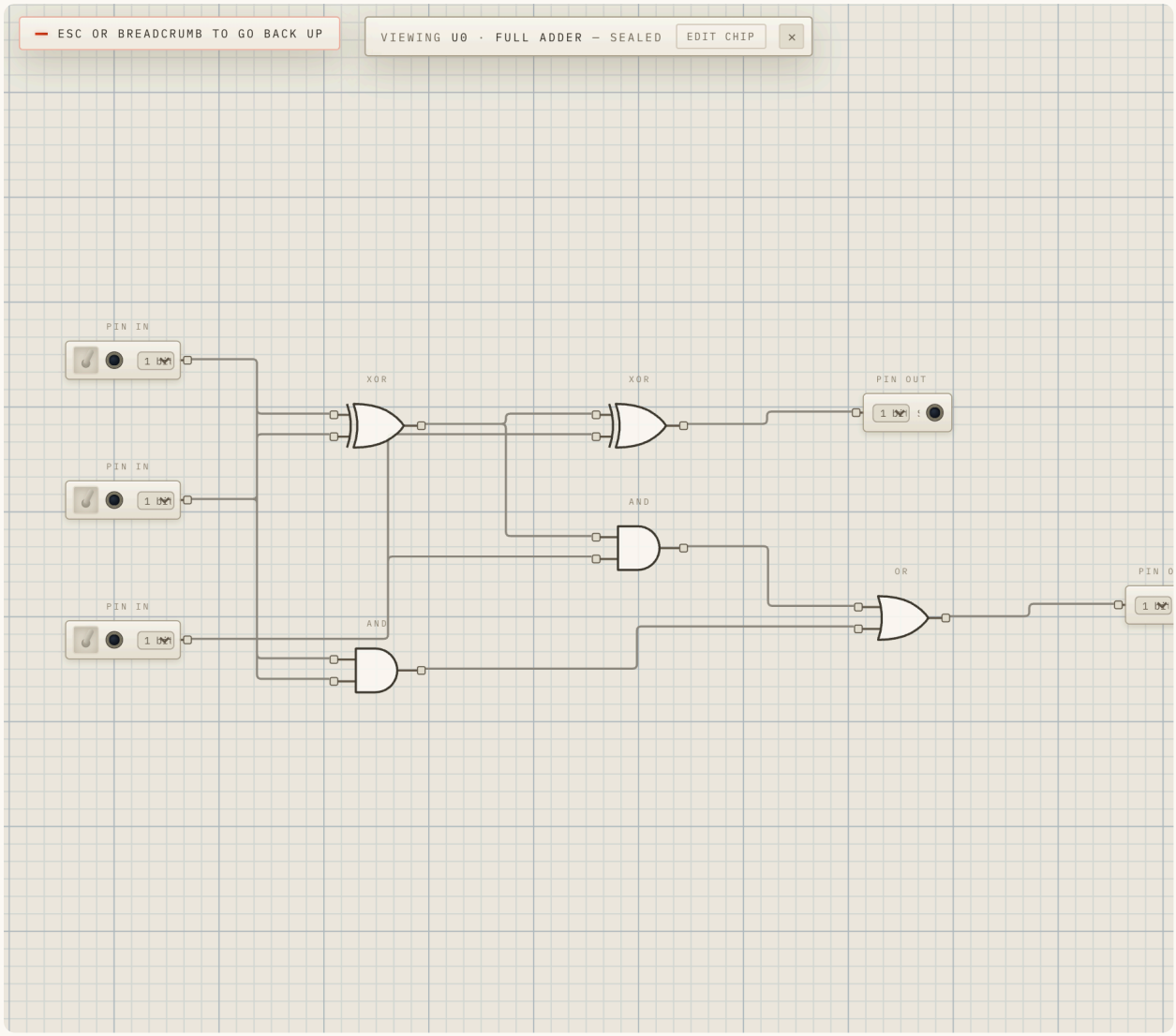
See it in the bench

> **Open this:** load **4-Bit Adder from Chips** from the library (category **ARITHMETIC**). You will see four **FULL ADDER** chip instances (U0–U3) chained by their carries, with operand switches and a hex SUM display. (Loading it also places FULL ADDER on the CHIPS shelf.)



Four FULL ADDER chips chained into a 4-bit adder

Then open a chip: double-click any FULL ADDER instance to drop inside and watch its gates.



Inside the FULL ADDER chip: the live gates one level down

Try it yourself

Predict first, then flip.

Try: 1. Set **A = 9** (A0, A3) and **B = 5** (B0, B2). The display reads **E** (14) — same arithmetic as Lesson 20, far fewer visible parts. 2. **Double-click U0**, the lowest full-adder chip, and drop inside. Watch its internal gates compute the lowest bit's sum and carry, live. The carry you see leaving it was computed by these gates, one level down. 3. Step back out and double-click a *different* instance. Same insides — because all four are instances of one definition. Change a definition once and every instance changes; that reuse is the power of chips.

Recap

- A **chip** packs a proven circuit into a sealed, reusable block with labelled pins.
- **Hierarchy** — building big things from sealed smaller things — is the central idea that makes complex hardware buildable.
- The bench's signature move: **double-click a chip to watch its real gates live.** The abstraction is real but never opaque.

Check yourself: Why can a designer use a FULL ADDER chip without thinking about XORs and ANDs — yet still trust it? *(Because hierarchy lets you treat a proven sealed block as one part; and you can always open it to verify the honest gates inside.)*

Next: Lesson 43 — The 8-Bit Adder on a Bus: chips and buses together.

Lesson 43 — The 8-Bit Adder on a Bus

> **Part VII · Buses & larger structure** — library circuit (category: ARITHMETIC) Before this lesson: the 4-Bit Adder from Chips (Lesson 42) and bus adapters (Lesson 41). After this: the 4×4 RAM (Lesson 44).

What you will learn

- How chips and buses combine to make a wide circuit genuinely readable.
- How an 8-bit operand travels as one bus instead of eight loose wires.
- A consolidation of every structural idea in Part VII.

The idea

This lesson brings together the two tools you just met — **chips** (Lesson 42) and **buses** (Lesson 41) — on a single circuit, and the result is the cleanest wide adder yet. It is the 8-bit addition you already understand from Lesson 21, but instead of sixteen loose input wires and a sprawl of gates, you see something close to how a real engineer would draw it.

Two things make it tidy:

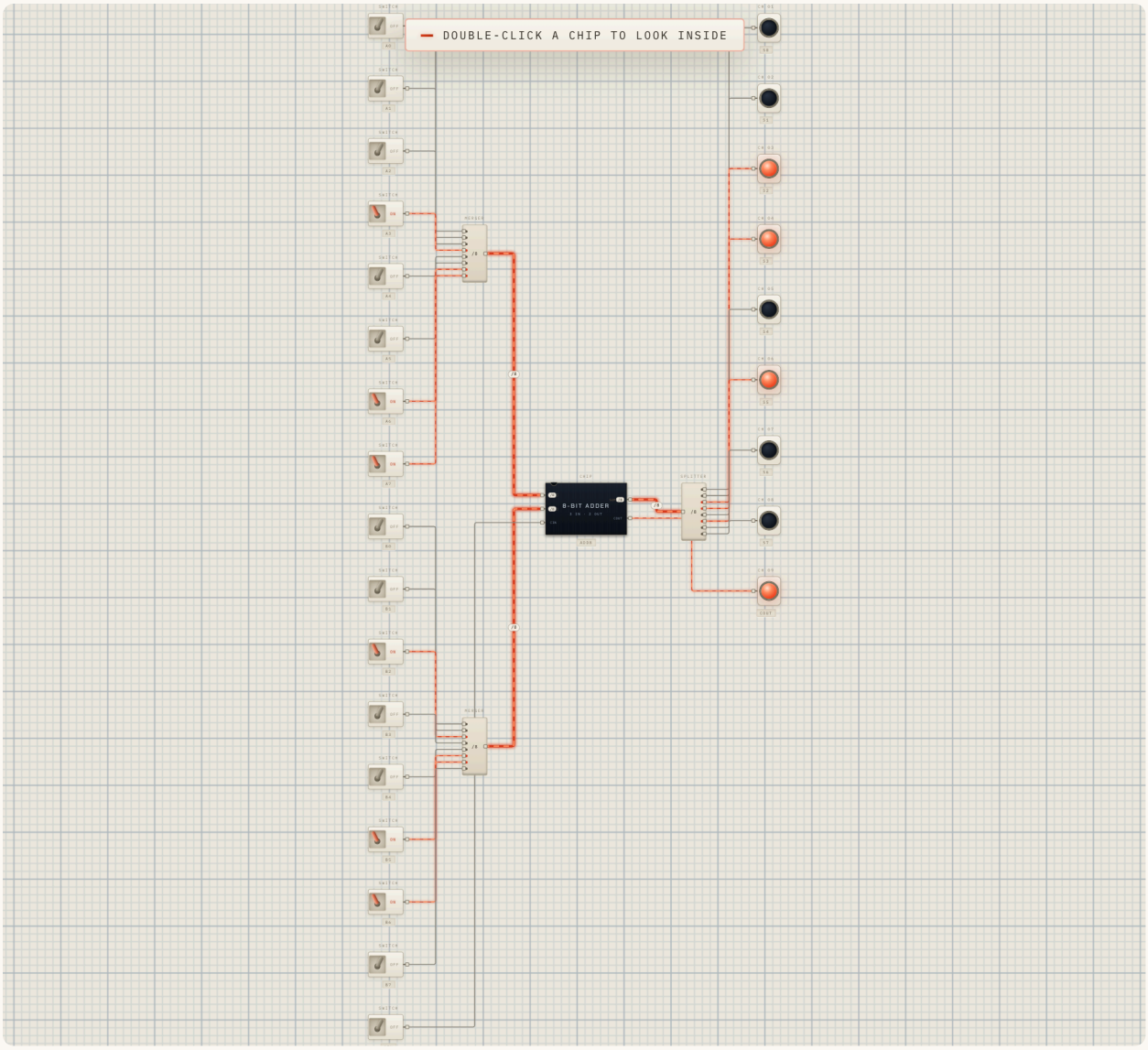
- **The operands ride on buses.** Each 8-bit number is gathered by a **merger** into one `/8` bus, routed as a single fat line, and fanned back out by a **splitter** where the bits are needed. The result bus is split out to the LEDs and hex displays. The sixteen-wire tangle becomes two clean lines in and one out.
- **The adding is done by packed adder chips.** The full-adder logic is sealed into chip blocks, chained carry-to-carry, exactly as in the previous lesson — just eight bits wide.

The payoff is conceptual, not arithmetic: the *math* is identical to Lesson 21, but the *drawing* now scales. This is the lesson where "how do real designers manage thousands of wires?" gets its answer — **bundle the wires (buses) and seal the logic (chips)**, then reason about the tidy blocks and fat lines instead of the underlying thousands of bits. And, as always, nothing is hidden: the buses expand to real bits in the engine, and the chips open to real gates under a double-click.

This circuit is the bridge between the small hand-wired pieces of Parts I–III and the large structured machines — RAM, ROM, and the CPU — coming next. From here on, you will increasingly read circuits as *blocks and buses*, trusting (and able to verify) the honest gates beneath.

See it in the bench

Open this: load **8-Bit Adder (Bus)** from the library (category **ARITHMETIC**). Two operand banks merge onto `/8` buses into chained adder chips; the sum is split out to a hex readout and COUT.



The 8-bit adder drawn as buses and adder chips

Try it yourself

Predict first, then flip.

Try: 1. Set **A = 3, B = 5** and confirm the hex readout shows **8** — the same result the loose 8-bit adder gave in Lesson 21. 2. Push it to overflow: **A = 200, B = 100**. The visible sum wraps to 44 and **COUT** lights — the carry out of the top bit, reporting that the true answer (300) needed a ninth bit. 3. Trace one operand from its switches, into the **merger**, along the **bus**, and into the adder chip. Then double-click an adder chip to drop into its gates. Bus on the outside, honest gates on the inside — the whole of Part VII in one view.

Recap

- The 8-bit bus adder combines **buses** (tidy wide wiring) and **chips** (sealed logic) on one circuit.
- The arithmetic equals Lesson 21's; the **drawing** is what scales — bundle the wires, seal the logic.
- It is the bridge from small hand-wired circuits to the large structured machines ahead.

Check yourself: What two structural tools make this wide adder readable, and does either change the logic? *(Buses and chips — and no, both are organisational; the engine still computes the same individual bits.)*

Next: Lesson 44 — The 4×4 RAM: building read/write memory from gates.

Part VIII — Stored memory: RAM & ROM

Lesson 44 — The 4×4 RAM (built from gates)

> **Part VIII · Stored memory** — library circuit (category: MEMORY) Before this lesson: the Gated D Latch (Lesson 27) and the 2-to-4 Decoder (Lesson 37). After this: the 256×8 RAM device (Lesson 45).

What you will learn

- How read/write memory is built entirely from gates you already know.
- How a decoder picks which word to access, and a mux reads it back.
- What "address", "data in", "data out" and "write enable" really are in hardware.

The idea

This is one of the most satisfying circuits in the book: **genuine read/write memory, built from nothing but gates and latches you have already met.** No new magic — just a clever arrangement of old parts. This RAM holds **4 words of 4 bits each** (sixteen bits of storage), and every piece of it is something you understand.

Three familiar ideas combine:

- **Storage: gated D latches.** Each of the sixteen bits is held in a gated D latch (Lesson 27). Sixteen latches, arranged as four rows (words) of four bits.
- **Choosing a word: a decoder.** A 2-bit **ADDRESS** feeds a **2-to-4 decoder** (Lesson 37), whose one-hot output is the set of **word lines**. Exactly one word is selected at a time — the decoder lights the row you addressed and leaves the other three alone.
- **Writing and reading.** A **WRITE ENABLE** signal, ANDed with the selected word line, opens the latches of *only* the addressed word so your **DATA IN**

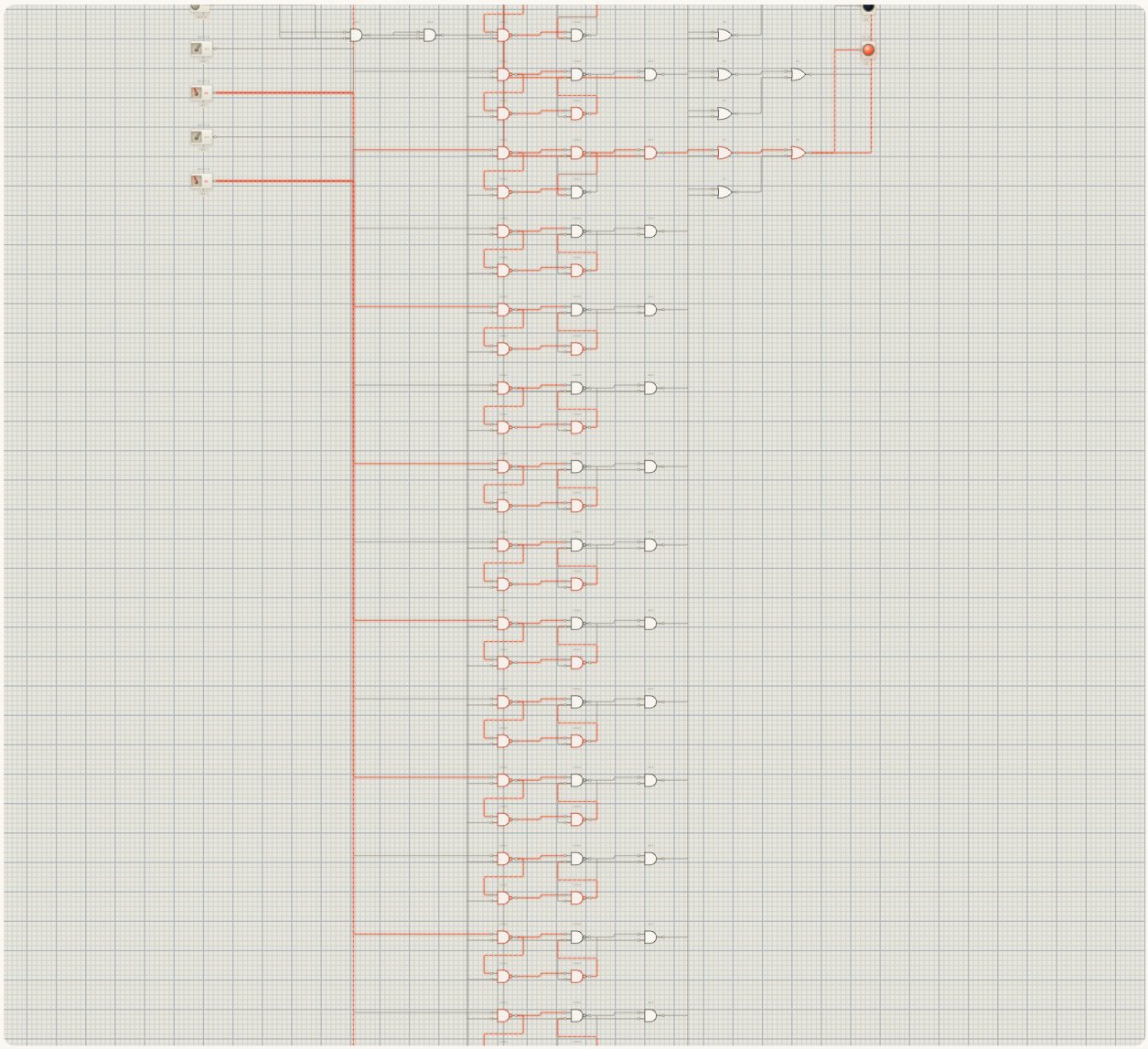
bits get stored there — every other word is left untouched. To read, a **4:1 multiplexer** (Lesson 36), driven by the same address, selects the addressed word's bits onto **DATA OUT**.

Step back and look at what that gives you: put an address on the address lines and a value on data-in, pulse write-enable, and that value is stored at that location. Later, put the same address back and the stored value appears on data-out — even though you have since stored other things at other addresses. The circuit *remembers a table of values you can write and re-read at will.* That is RAM, and you can see every gate of it.

This is the honest heart of computer memory. The RAM in a real machine is this same idea — address decoder, storage cells, write logic, read mux — repeated billions of times with denser cells. Here, at 4×4, it is small enough to watch.

See it in the bench

Open this: load **4×4 RAM (from gates)** from the library (category **MEMORY**). You will find ADDRESS (2 bits), DATA IN (4 bits), WRITE ENABLE, and DATA OUT (4 bits), wrapped around a decoder, sixteen latches, and a read mux.



The 4×4 gate-built RAM with its decoder, latch array and read mux

Try it yourself

Predict first, then flip.

Try: 1. **Write a value.** Set **ADDRESS = 00**, set **DATA IN = 1010**, then pulse **WRITE ENABLE** on and off. You have stored 1010 at address 0. DATA OUT should now show 1010. 2. **Store something elsewhere.** Set **ADDRESS = 01**, **DATA IN = 0011**, pulse WRITE ENABLE. Address 1 now holds 0011. 3. **Read back.** Set ADDRESS back to **00** (do *not* pulse write). DATA OUT shows **1010** again — address 0 kept its value while you wrote to address 1. Switch to **01**: DATA OUT shows **0011**. The memory is holding a table you can revisit. 4. Watch the **decoder** output as you change the address — exactly one word line is hot, naming the row you are touching. That one-hot line is what protects the other words from your writes.

Watch out: writing happens only while WRITE ENABLE is active on the addressed word. If you change DATA IN without pulsing write enable, nothing is stored — you are just changing what *would* be written. This is the difference between presenting data and committing it.

Recap

- A 4×4 RAM stores four 4-bit words, built from **gated D latches**, a **decoder**, write logic, and a **read mux** — all parts you already knew.
- **ADDRESS** picks a word (via the decoder's one-hot word lines); **WRITE ENABLE** commits **DATA IN** to that word; the read mux puts the addressed word on **DATA OUT**.
- This is the honest core of all computer memory, small enough to watch gate by gate.

Check yourself: Why does writing to address 1 not disturb the value stored at address 0? *(The decoder makes only address 1's word line hot, so write-enable*

opens only that word's latches; address 0's latches stay closed and hold their value.)

Next: Lesson 45 — The 256×8 RAM device: the same idea as a larger behavioural device you can inspect.

Lesson 45 — The 256×8 RAM device

> **Part VIII · Stored memory** — library circuit (category: MEMORY / DEVICES) Before this lesson: the 4×4 RAM from gates (Lesson 44). After this: the Lookup ROM (Lesson 46).

What you will learn

- Why some components are modelled as **behavioural devices** rather than gate-by-gate.
- How a 256-byte RAM works: 8-bit address, 8-bit data, write-enable.
- The honest tradeoff the bench makes here, stated plainly.

The idea

In Lesson 44 you built RAM from sixteen visible latches. That is wonderful for *understanding*, but it does not scale: a realistic memory has thousands or millions of cells, and drawing every latch would be impossible to build with or even see. So for practical memory, the bench provides a **device**.

A **device** is a component whose behaviour is modelled directly — it *acts* exactly like the real thing, with honest inputs and outputs, but its internals are described by a behavioural rule rather than drawn as individual gates. The **256×8 RAM device** stores **256 words of 8 bits** (256 bytes). Its pins are the grown-up versions of the ones you just met:

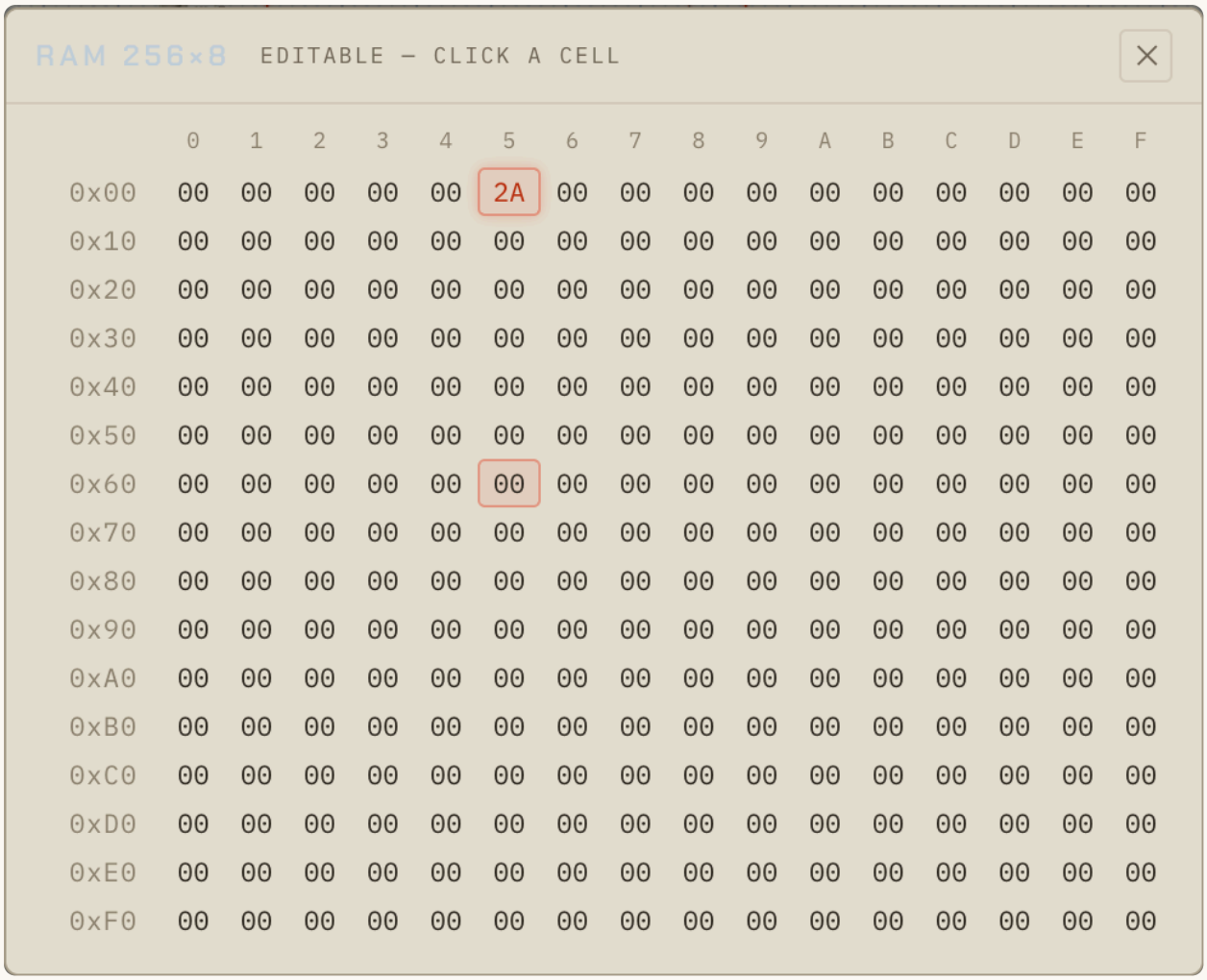
- An **8-bit ADDRESS** (so it can name any of 256 = 2^8 locations).
- An **8-bit DATA IN** and **8-bit DATA OUT**.
- A **WRITE ENABLE**, which captures DATA IN into the addressed byte on its rising edge (a clean one-shot write, like the button-pulse of Lesson 02).

Functionally it is precisely the 4×4 RAM scaled up: address in, decoder picks a location, data is written or read. Nothing about the *idea* changed — only the size, and the decision to model it behaviourally so it is usable.

Here is where the bench keeps faith with you. This is one of the few places it does *not* show you the gates inside — and it is honest about that being a deliberate, stated boundary, not a hidden cheat. Crucially, **the device is not a black box you must take on faith: its stored contents are fully inspectable.** You can open the device and view its memory — all 256 bytes laid out — and watch a cell change the instant you write to it. You have already proven, in Lesson 44, that this behaviour *is* buildable from gates; the device simply lets you use that proven idea at a useful scale while still seeing every value it holds.

See it in the bench

> **Open this:** load **256×8 RAM (Device)** from the library (category **MEMORY** / devices). Set the 8-bit ADDRESS and DATA IN, use WRITE ENABLE to store, and open the device's memory inspector to view its contents.



The 256-byte RAM device with its memory contents inspector open

Try it yourself

Predict first, then flip.

Try: 1. Set **ADDRESS = 0x00** and **DATA IN = 0x2A** (binary 00101010). Pulse **WRITE ENABLE**. Open the memory inspector and find that byte 0 now holds **2A**. 2. Change **ADDRESS = 0x01**, **DATA IN = 0xFF**, pulse write. Byte 1 becomes **FF**, byte 0 still reads 2A in the inspector. Two locations, two independent values. 3. Set ADDRESS back to **0x00** without writing — DATA OUT shows **2A** again. Reading does not disturb storage. 4. Scroll the inspector and appreciate the scale: 256 bytes, each one a location you could write — and every one visible. This is the same circuit as Lesson 44, just 64× bigger and modelled as a device.

Watch out: the write happens on the **rising edge** of WRITE ENABLE — the moment it goes from 0 to 1 — capturing DATA IN at that instant. Holding it high does not keep re-writing in a way you can see; think of each write as one clean strobe, the button-pulse discipline from Lesson 02.

Recap

- A **device** models a component's behaviour directly instead of drawing every internal gate — necessary once memory gets large.
- The **256×8 RAM** is the 4×4 RAM scaled up: 8-bit address (256 locations), 8-bit data, edge-triggered **WRITE ENABLE**.
- The bench is honest about this boundary: the gates aren't shown, but the **contents are fully inspectable**, and you already proved the design from gates in Lesson 44.

Check yourself: Why does the bench model the 256-byte RAM as a device rather than from latches like the 4×4? *(Drawing thousands of individual*

latches would be unusable; a behavioural device acts identically at a practical scale — and its contents stay inspectable.)

Next: Lesson 46 — The Lookup ROM: memory that only reads, holding a fixed table.

Lesson 46 — The Lookup ROM

> **Part VIII · Stored memory** — library circuit (category: MEMORY / DEVICES) Before this lesson: the 256×8 RAM device (Lesson 45). After this: the 4-Bit ALU (Lesson 47).

What you will learn

- The difference between RAM and ROM.
- How a ROM acts as a fixed lookup table that converts inputs to outputs.
- Why "precompute and store" is a powerful alternative to "calculate with gates."

The idea

RAM can be both written and read. A **ROM** — Read-Only Memory — can only be **read**: its contents are fixed in advance and do not change as the circuit runs. You give it an address, and it gives back the value stored there. That is all it does, and it turns out to be enormously useful.

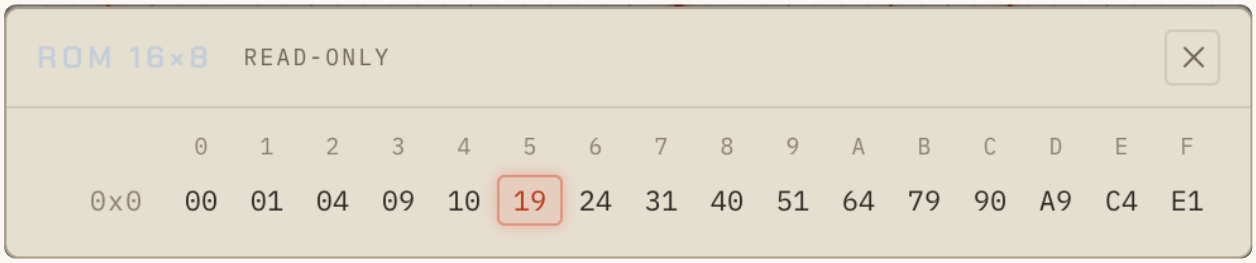
The key insight is that **a ROM is a lookup table frozen into hardware.** Instead of *computing* an answer with gates, you can *precompute* every answer ahead of time, store the results in a ROM, and then just look them up. The address is the question; the stored byte is the pre-prepared answer.

This example ROM stores **squares**: at address n it holds the value n^2. Feed in the address 5 and out comes 25; feed in 9 and out comes 81. No multiplier gates are involved at lookup time at all — the squares were all worked out in advance and stored, and the ROM simply fetches the one you asked for. Its contents are inspectable, exactly like the RAM device, so you can open it and read the whole table of squares.

This "trade computation for storage" idea is everywhere in real systems: ROMs hold character-shape tables for displays, trigonometry tables, colour palettes, and — most importantly for the next Part — the **microcode** that tells a CPU what to do for each instruction. In fact a ROM addressed by an input pattern can implement *any* logic function you can write a truth table for: just store the output column and look it up. A ROM is a universal "answer table." Keep that in mind, because the LB-8 CPU's control logic (Lesson 53) is built exactly this way — its behaviour lives in a ROM.

See it in the bench

Open this: load **Lookup ROM (Squares)** from the library (category **MEMORY** / devices). Set the ADDRESS and read the squared value on the output; open the ROM's inspector to see the full table.



The squares ROM returning n-squared for the addressed input

Try it yourself

Predict first, then flip.

Try: 1. Set **ADDRESS = 5**. The output reads **25** — the ROM looked up 5^2, it did not calculate it. 2. Set **ADDRESS = 9** → output **81**. Step the address up one at a time and watch the output trace the squares: 0, 1, 4, 9, 16, 25, 36, 49, 64, 81 … 3. Open the ROM's inspector and read the stored table directly. Every answer was sitting there in advance; the address just chose which to reveal. 4. Notice what you *cannot* do: there is no write-enable that changes the table as it runs. The ROM is read-only — its knowledge is fixed.

Recap

- **ROM** is read-only: fixed contents, addressed to **look up** a stored value (RAM, by contrast, can be written).
- A ROM is a **precomputed lookup table in hardware** — trade computation for storage; the squares ROM returns n^2 with no multiplier at lookup time.
- A ROM can implement any truth-table function, which is why CPU **microcode** lives in ROM (Lesson 53).

Check yourself: A ROM stores n^2 at address n. How does it "compute" $7^2 = 49$ when you address it with 7? *(It doesn't compute at all — 49 was precomputed and stored at address 7, and the ROM simply fetches it.)*

Next: Lesson 47 — The 4-Bit ALU: the calculating heart of a CPU.

Part IX — The CPU

Lesson 47 — The 4-Bit ALU

> **Part IX · The CPU** — library circuit (category: CPU) Before this lesson: the adders (Part III), subtractors (Lesson 22), and multiplexer (Lesson 36). After this: the Instruction Decoder (Lesson 48).

What you will learn

- What an ALU is and why it is the calculating heart of every processor.
- How one circuit performs several operations, chosen by a select code.
- How a single adder is made to subtract, using the XOR trick from Lesson 13.

The idea

The **ALU** — Arithmetic Logic Unit — is the part of a CPU that actually *computes*. Everything else in a processor exists to feed the ALU operands and route its results; the ALU is where numbers are added, subtracted, and combined. This one is **4 bits wide** and performs **four operations**, selected by a 2-bit operation code **OP1 OP0**:

OP1	OP0	Operation
0	0	**ADD** (A + B)
0	1	**SUB** (A − B)
1	0	**AND** (A AND B, bitwise)
1	1	**OR** (A OR B, bitwise)

The structure pulls together much of this book. Internally, all the operation results are computed, and a **multiplexer** (Lesson 36), driven by the OP code,

selects which one reaches the output — exactly the "compute several, select one" pattern you have seen before, now doing real CPU work.

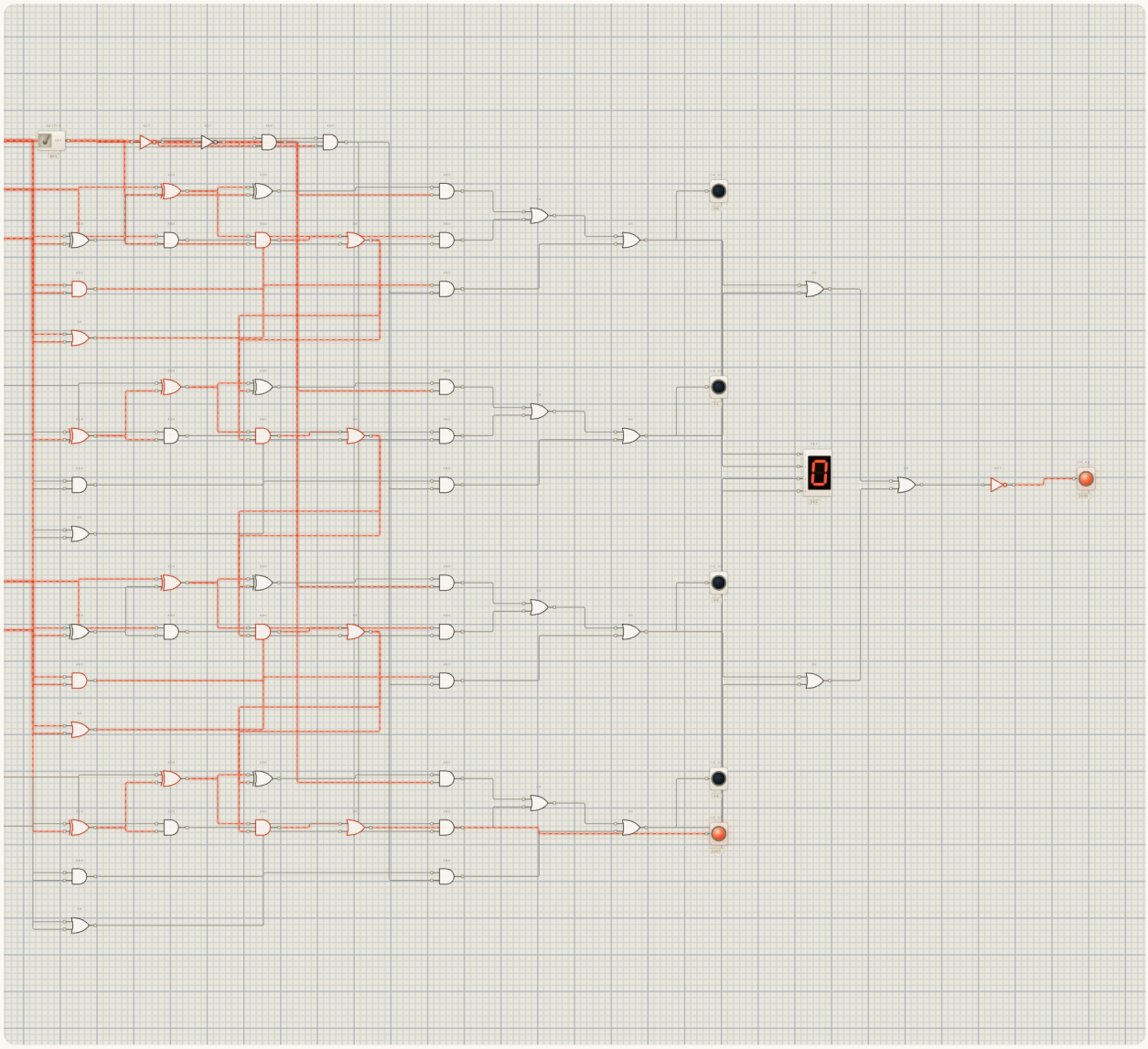
The cleverest part is how **one adder does both ADD and SUB**. Recall from Lesson 13 that XOR acts as a *controlled inverter*: XOR a bit with 1 and it flips; with 0 and it passes through. Subtraction A − B in binary is done by adding A to the **two's complement** of B — which is "flip every bit of B, then add 1." So the ALU feeds B through a row of XOR gates controlled by the operation: in SUB mode it flips B's bits and forces a carry-in of 1 (the "+1"), turning the *same adder* into a subtractor. Add and subtract share one piece of hardware, steered by a control line. That economy is exactly why the XOR-as-controlled-inverter idea was worth planting earlier.

The ALU also produces **flags** — status bits describing the result:

- **COUT** (carry-out): the carry off the top of an add (or the borrow indicator of a subtract).
- **ZERO**: high when the entire result is 0 — computed by NOR-ing all the result bits together (Lesson 12's "all inputs 0" gate). The ZERO flag is how a CPU later answers questions like "did that subtraction come out equal?", which drives conditional jumps.

See it in the bench

> **Open this:** load **4-Bit ALU** from the library (category **CPU**). Set operands A and B (4 bits each) and the operation OP1 OP0; read the 4-bit RESULT plus the COUT and ZERO flags.



The 4-bit ALU subtracting two equal operands, ZERO flag lit

Try it yourself

Predict first, then flip.

Try: 1. **ADD:** OP = 00, A = 0011 (3), B = 0101 (5). RESULT = 1000 (8). The ALU is using the ripple adder you know. 2. **SUB:** OP = 01, A = 0101 (5), B = 0011 (3). RESULT = 0010 (2). Behind the scenes B was bit-flipped and a carry-in of 1 added — the same adder, subtracting. 3. **The ZERO flag:** keep OP = SUB and set **A = B** (say both 0101). RESULT = 0000 and **ZERO lights**. This is how a CPU tests equality: subtract, then look at ZERO. You will use exactly this in the jump instructions ahead. 4. **Logic ops:** OP = 10 (AND) and OP = 11 (OR) with various A, B — the RESULT is the bitwise gate applied across all four bit positions.

Recap

- The **ALU** is the CPU's calculating core: here, 4 bits wide with four operations (ADD, SUB, AND, OR) chosen by **OP1 OP0**.
- A **mux** selects which operation's result is output — "compute several, select one."
- One adder does **both add and subtract** via XOR controlled-inversion (two's complement); **flags** COUT and ZERO describe the result, and ZERO drives later conditional jumps.

Check yourself: How does the ALU subtract using an adder, and what does the ZERO flag let a CPU decide? *(It flips B's bits and adds 1 — two's complement — so the adder computes A − B; ZERO tells the CPU whether the result was zero, e.g. whether two values were equal.)*

Next: Lesson 48 — The Instruction Decoder: turning an opcode into control signals.

Lesson 48 — The Instruction Decoder

> **Part IX · The CPU** — library circuit (category: CPU) Before this lesson: the 2-to-4 Decoder (Lesson 37) and the ALU (Lesson 47). After this: the Program Counter (Lesson 49).

What you will learn

- How a CPU turns a numeric **opcode** into the control signals that make things happen.
- The idea of one-hot **control lines**, one per instruction.
- Why this is the "translator" between a program and the hardware.

The idea

A program is a list of numbers. But wires do not understand numbers — they understand *signals*. Something has to stand between "the instruction is number 3" and "therefore the hardware should do an ADD." That something is the **instruction decoder**.

It is a decoder exactly in the sense of Lesson 37, scaled up: it takes a **3-bit opcode** (so it can name $2^3 = 8$ different instructions) and produces **8 one-hot control lines** — exactly one line high, identifying which instruction this is. This example uses an 8-instruction set:

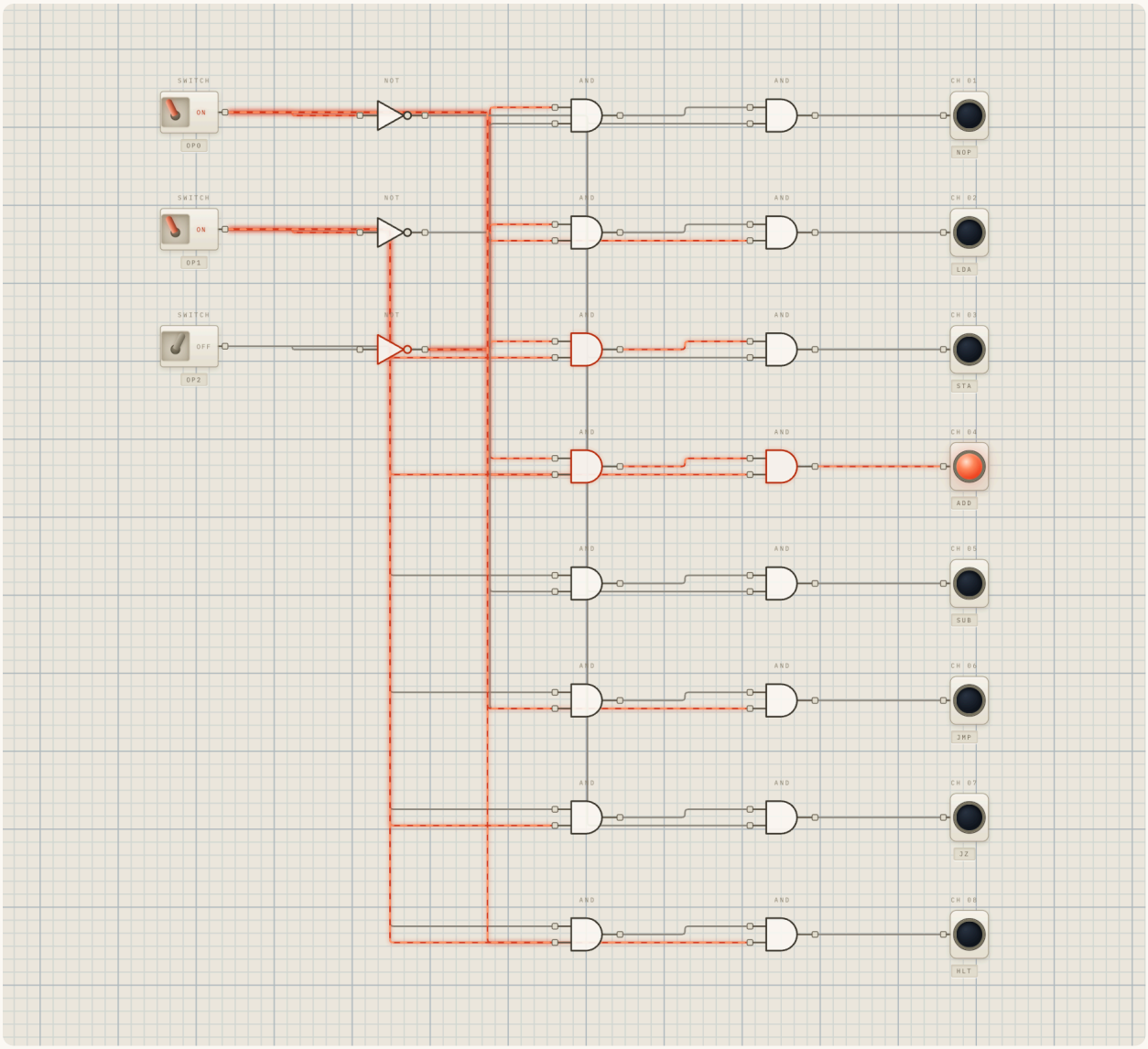
Opcode	Instruction	Meaning
000	**NOP**	do nothing
001	**LDA**	load a value into the accumulator
010	**STA**	store the accumulator to memory
011	**ADD**	add a value to the accumulator
100	**SUB**	subtract a value from the accumulator
101	**JMP**	jump to an address
110	**JZ**	jump, but only if the ZERO flag is set
111	**HLT**	halt the machine

When opcode 011 arrives, the **ADD** line — and only the ADD line — goes high. That single hot wire is what will, in a full CPU, switch the ALU to its add operation, open the right register, and so on. Each control line is the hardware's name for one instruction.

This is the **translator** at the centre of a CPU. On one side is software — numbers in memory. On the other side is hardware — gates that need to be told what to do. The decoder converts the former into the latter, opcode by opcode. Every instruction your real computer runs passes through a decoder like this one, turning a stored number into a pattern of control signals. It is the precise point where a *program* becomes *action*.

See it in the bench

> **Open this:** load **Instruction Decoder (3-to-8)** from the library (category **CPU**). Three opcode switches feed the decoder; eight labelled control-line LEDs (NOP…HLT) show the one-hot output.



The instruction decoder lighting the ADD line for opcode 011

Try it yourself

Predict first, then flip.

Try: 1. Set the opcode to **011**. Predict which line lights — **ADD**, and only ADD. The opcode is a number; the lit line is its meaning. 2. Step the opcode through **000 → 111** and watch the hot line walk down the instruction list: NOP, LDA, STA, ADD, SUB, JMP, JZ, HLT. Eight numbers, eight meanings. 3. Pick any opcode and notice that *exactly one* control line is ever active. A CPU does one instruction at a time, and this one-hot output is the hardware enforcing that. 4. Look ahead: imagine the **ADD** line wired to switch the ALU (Lesson 47) into add mode, and the **JZ** line wired to consult the ZERO flag. That wiring is what the tiny CPUs ahead actually do.

Recap

- The **instruction decoder** turns a numeric **opcode** into **one-hot control lines**, one per instruction.
- It is the **translator** between software (numbers in memory) and hardware (gates that act) — where a program becomes action.
- Exactly one control line is hot at a time, naming the current instruction (here, one of NOP/LDA/STA/ADD/SUB/JMP/JZ/HLT).

Check yourself: The opcode 110 arrives. Which control line lights, and what makes that instruction special? *(JZ — and it is conditional: it jumps only if the ZERO flag is set.)*

Next: Lesson 49 — The Program Counter: the register that keeps the CPU's place in the program.

Lesson 49 — The Program Counter

> **Part IX · The CPU** — library circuit (category: CPU) Before this lesson: the Ripple Counter (Lesson 32) and the Register (Lesson 28). After this: the Tiny 4-Bit Computer (Lesson 50).

What you will learn

- What a program counter (PC) does and why every CPU has one.
- How a counter keeps the machine's place in a program.
- How RUN and HLT start and stop the march of execution.

The idea

A program lives in memory as a numbered list of instructions: address 0, address 1, address 2, and so on. To run the program, the CPU must keep track of **which instruction it is up to** — and step forward to the next one each cycle. The register that holds that place is the **program counter**, or **PC**.

It is, at heart, a **4-bit counter** (Lesson 32) with a job title. Each clock cycle, the PC supplies the current address to memory ("fetch the instruction here"), and then **increments** — 0, 1, 2, 3 … — advancing to the next instruction. That steady climb is the literal mechanism of a program "running": the PC walking up through the addresses, pulling out one instruction after another. The PC *is* the machine's place-marker in the program.

Two control ideas make it a usable CPU part rather than a free-running counter:

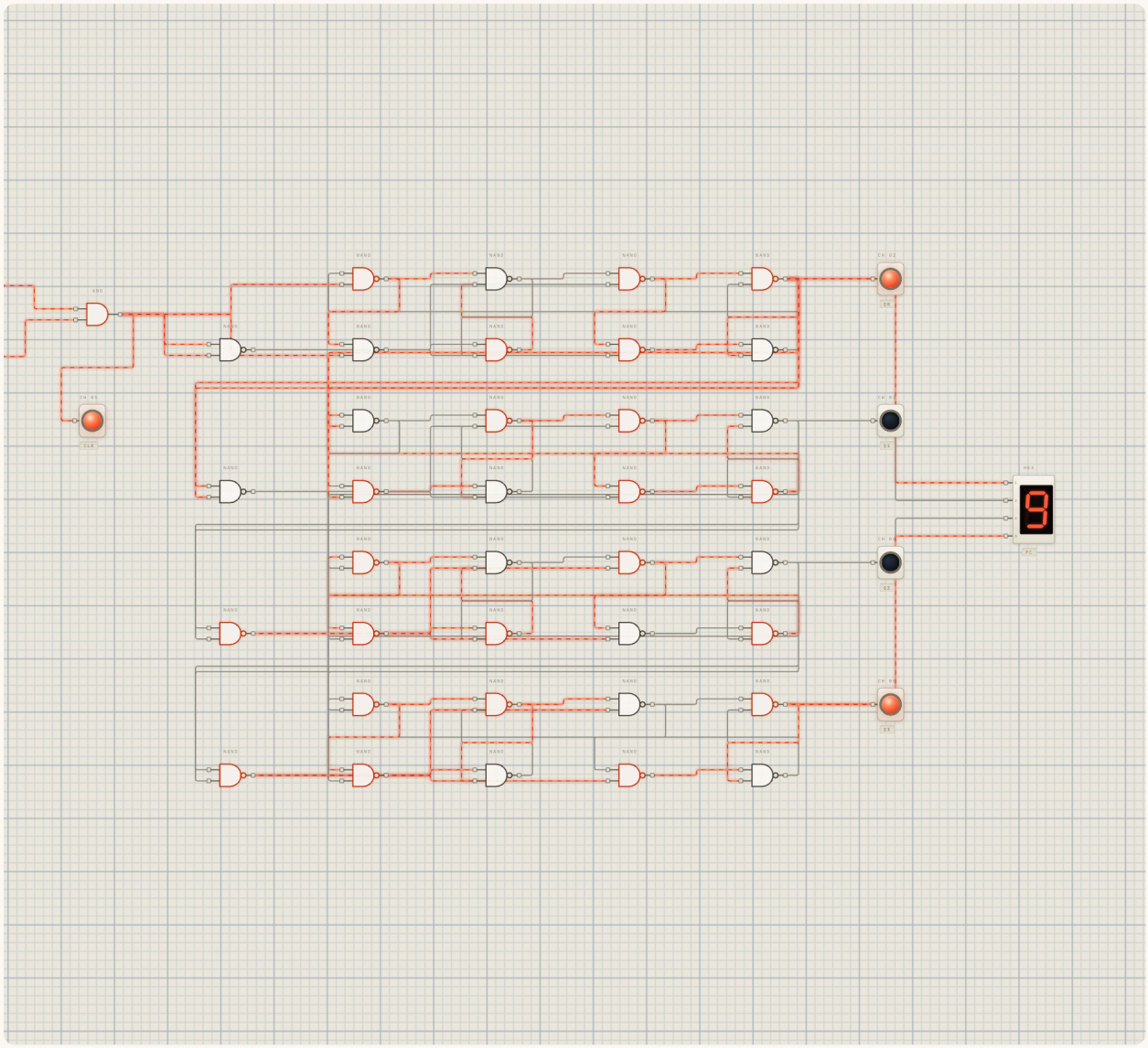
- **RUN-gated clock.** The PC only advances while a **RUN** signal is active. Gate the clock with RUN (an AND gate) and you can start and pause execution. No RUN, no stepping — the machine waits.

- **HLT parks it.** When the program reaches a **HLT** (halt) instruction — opcode 111 from the previous lesson — the control logic drops RUN, which freezes the PC. The machine stops cleanly on the halt and stays put. That is what "the program ended" looks like in hardware: the program counter parked, no longer advancing.

There is one more power the PC needs, which you will see used fully in the next lessons: a program counter can also be **loaded** with a new address instead of just incrementing. That is how a **jump** works — JMP and JZ from Lesson 48 force a new value into the PC, so execution continues from somewhere else. Increment is the normal step; load is the jump. With those two behaviours, a counter becomes the controller of program flow.

See it in the bench

> **Open this:** load **Program Counter (4-Bit)** from the library (category **CPU**). A clock gated by RUN drives the counter; its 4-bit output is the current address. A HLT/stop control parks it.



The program counter stepping through addresses, gated by RUN

Try it yourself

Try: 1. Turn **RUN** on and watch the PC climb 0, 1, 2, 3 … — it is generating the stream of addresses a program would be fetched from. 2. Turn **RUN** off. The PC freezes in place. Turn it back on — it resumes from where it paused. That start/pause is the RUN-gated clock at work. 3. Trigger the **HLT/stop** condition and watch the PC park and stay parked. This is exactly how a CPU stops when a program finishes. 4. Hold the bigger picture: this counter, supplying addresses and able to be loaded for jumps, is the engine of program flow. The next lesson drops it into a complete CPU where its output feeds program memory.

Recap

- The **program counter (PC)** holds the address of the current instruction and **increments** each cycle to advance through the program.
- A **RUN-gated clock** lets execution start and pause; a **HLT** instruction drops RUN and **parks** the PC — how a program stops.
- The PC can also be **loaded** with a new address, which is how **jumps** redirect program flow.

Check yourself: What does incrementing the program counter actually accomplish, and what happens to it on HLT? *(Incrementing moves the CPU to the next instruction's address, advancing the program; on HLT, RUN drops and the PC freezes, stopping execution.)*

Next: Lesson 50 — The Tiny 4-Bit Computer: every piece so far, assembled into a working CPU.

Lesson 50 — The Tiny 4-Bit Computer

> **Part IX · The CPU** — library circuit (category: CPU) Before this lesson: the ALU (47), the Instruction Decoder (48), and the Program Counter (49). After this: the Tiny CPU II with RAM (Lesson 51).

What you will learn

- How every part you have built assembles into a **complete, working CPU**.
- The **fetch–decode–execute** cycle that defines how a computer runs.
- That there is no magic anywhere — you can open every block down to gates.

The idea

This is the summit the whole book has been climbing toward. The **Tiny 4-Bit Computer** is a complete, single-cycle CPU that runs a program — and **every single piece of it is something you have already built and understood.** Nothing new is introduced here; this lesson is the moment all the parts click together into a machine.

Look at what is inside and recognise each piece:

- A **program counter** (Lesson 49) holds the current address and steps through the program.
- A small gate-built **ROM** (Lesson 46) holds the program — the instructions, burned in.
- An **instruction decoder** (Lesson 48) turns each opcode into control lines.
- An **ALU** (Lesson 47) does the arithmetic.
- An **accumulator** — a **register** (Lesson 28) — holds the value being worked on.

Wire those together and they perform, over and over, the **fetch–decode–execute cycle** that is the heartbeat of every computer that has ever existed:

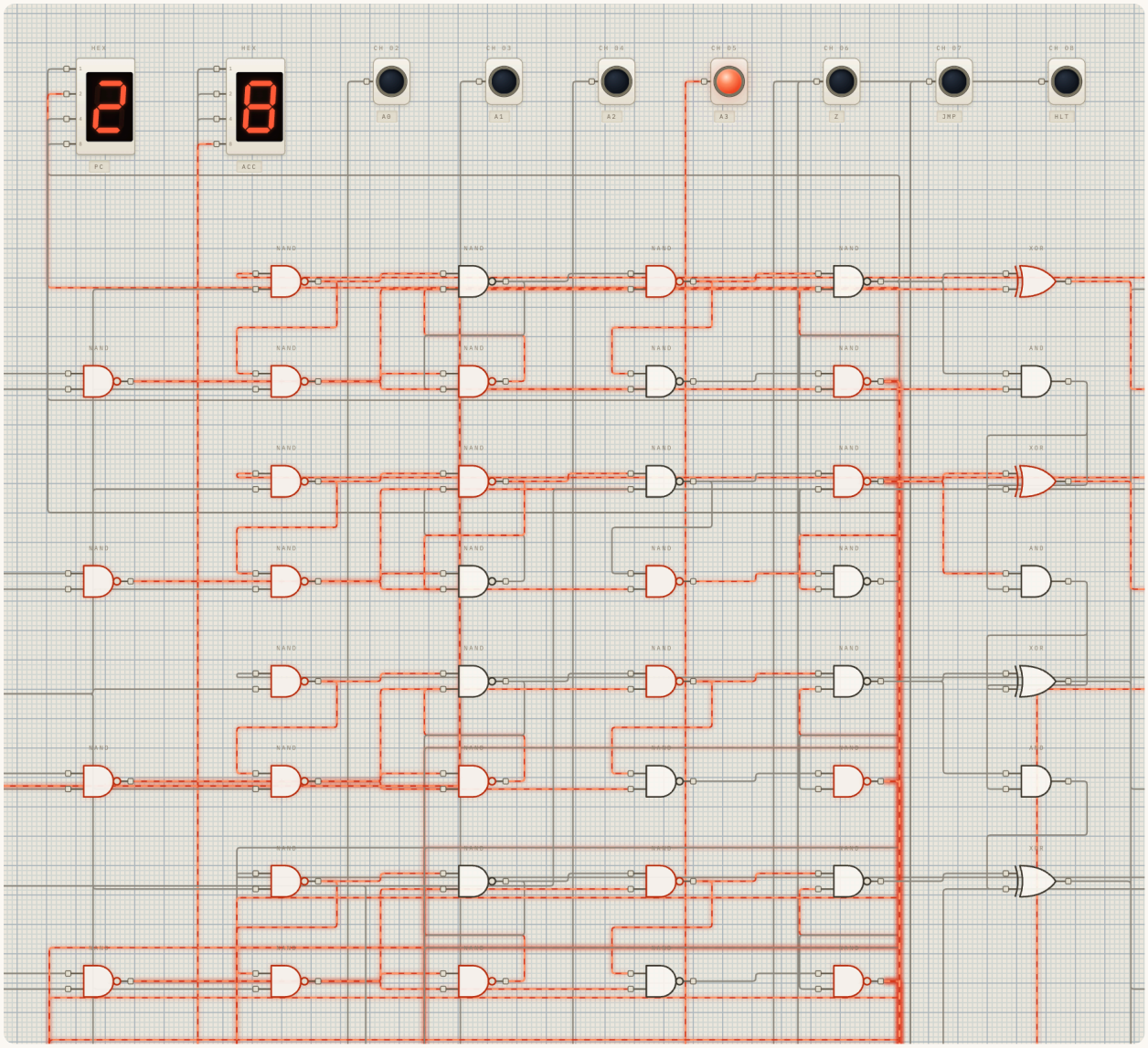
1. **Fetch** — the PC's address selects an instruction from the ROM.
2. **Decode** — the decoder turns that instruction's opcode into control signals.
3. **Execute** — the signals steer the ALU and accumulator to actually do the operation.
4. The PC advances, and the cycle repeats with the next instruction — until a HLT parks it.

That loop, running on the clock, *is* a computer running a program. The tiny CPU here executes a short burned-in program automatically, step by step, until it halts — and you can watch the accumulator change as each instruction runs.

And here is the promise this entire book has been building toward, now fully paid off: **you can open every block.** Double-click the ALU and watch its adder's gates compute a sum. Drop into the decoder and see the one-hot line light. Open the ROM and read the program. There is no level at which understanding runs out and magic begins — it is gates all the way down, from this running CPU to the single AND gate of Lesson 09. Sit with that for a moment. You now understand, concretely and completely, how a computer works.

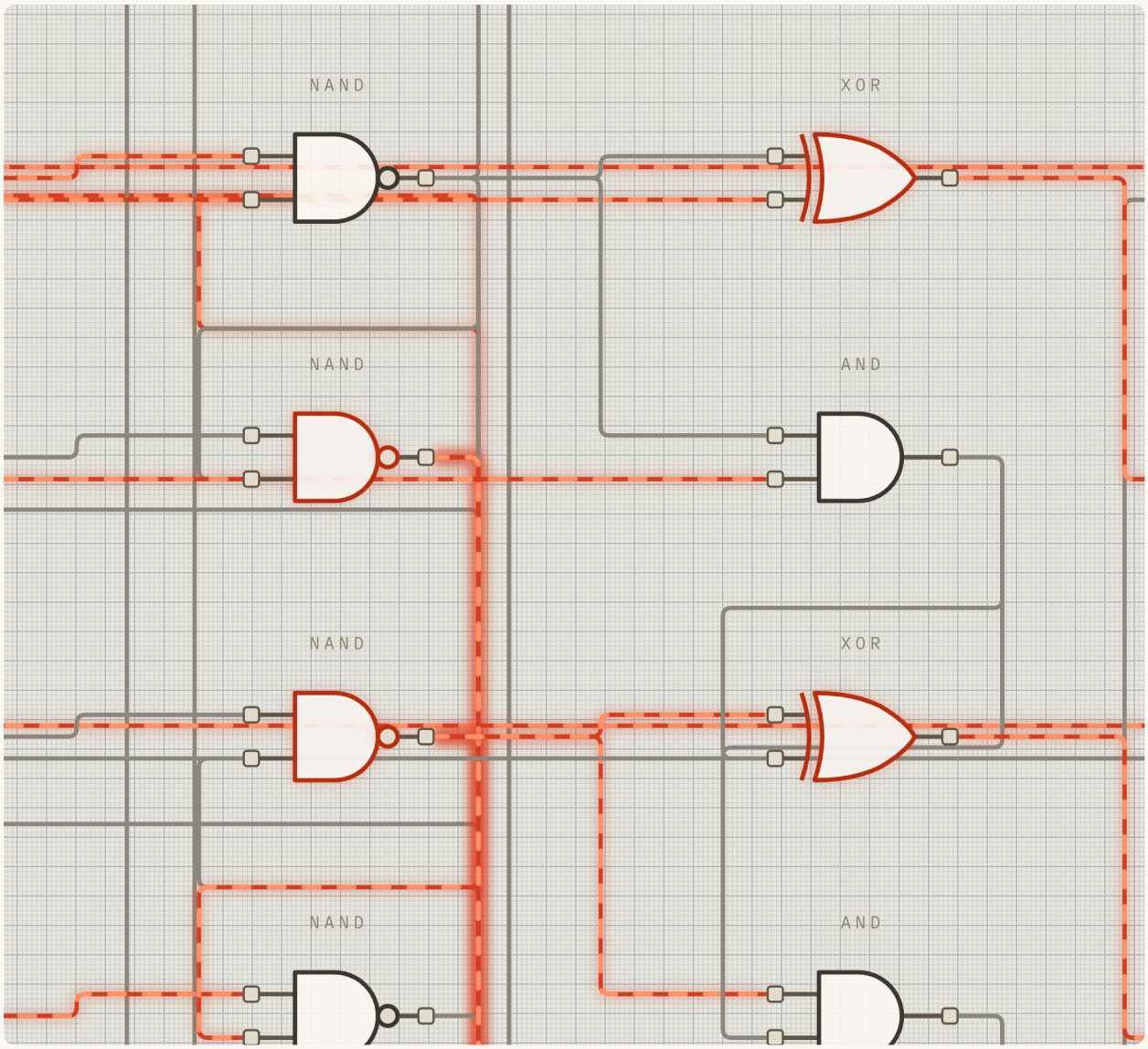
See it in the bench

> **Open this:** load **Tiny 4-Bit Computer** from the library (category **CPU**). It runs its built-in program on the clock; watch the PC, the decoded instruction, the ALU, and the accumulator as it executes.



The Tiny 4-Bit Computer running its program, accumulator updating

Then open a block: double-click the ALU (or the decoder, or the ROM) to drop inside and watch its gates work while the CPU runs.



Inside a CPU block: live gates one level down

Try it yourself

> **Try:** 1. Let it **run** and watch the **accumulator** change as each instruction executes. Slow the clock down (the clock face) so you can follow one fetch–decode–execute step at a time. 2. Watch the **program counter** climb with each instruction, then **park** when the program hits HLT. That is the program beginning, running, and ending — visibly. 3. **Open the ALU** mid-run and watch its gates compute. Step back out, open the **decoder**, and watch the control line for the current instruction light. The CPU keeps running while you look inside. 4. Trace one full instruction end to end: PC supplies an address → ROM gives an instruction → decoder lights a control line → ALU/accumulator act → PC advances. One lap of the cycle, every piece familiar.

Recap

- The **Tiny 4-Bit Computer** is a complete CPU assembled entirely from parts you already built: PC, ROM, decoder, ALU, accumulator.
- It runs the **fetch–decode–execute cycle** — fetch an instruction, decode it to control signals, execute it, advance, repeat until HLT.
- **Every block opens to its gates** while it runs: there is no magic layer, from the whole CPU down to a single gate.

Check yourself: Name the four steps the CPU repeats to run a program. *(Fetch the instruction at the PC's address, decode its opcode into control signals, execute the operation, then advance the PC — repeating until HLT.)*

Next: Lesson 51 — The Tiny CPU II (with RAM): a CPU that can store results back into memory.

Lesson 51 — The Tiny CPU II (with RAM)

> **Part IX · The CPU** — library circuit (category: CPU) Before this lesson: the Tiny 4-Bit Computer (Lesson 50) and the 4×4 RAM (Lesson 44). After this: the Tiny CPU II loadable (Lesson 52).

What you will learn

- What changes when a CPU can **write** to memory, not just read instructions.
- The **stored-program** (von Neumann) idea, made concrete.
- How read/write memory lets a program compute a real sequence.

The idea

The Tiny CPU of Lesson 50 could run a program, but its memory was a **ROM** — read-only. It could compute, but it had nowhere to *put* a result that it could later read back. **Tiny CPU II** adds that missing power: it includes a gate-built **4×4 RAM** (the very circuit you built in Lesson 44), so the machine can now both **read from and write to** memory as it runs.

This small change is a giant conceptual step. A CPU that can write to its own memory and read it back has working **variables** — named places to keep intermediate results. With the STA (store) and LDA (load) instructions from your instruction set (Lesson 48), the program can save a value, do more work, and come back for it later. That is the difference between a fixed calculator and a genuine computer.

This is also a clean illustration of the **stored-program computer** — the von Neumann idea that program and data live together in addressable memory. Instructions are fetched from memory; results are written back to memory; it

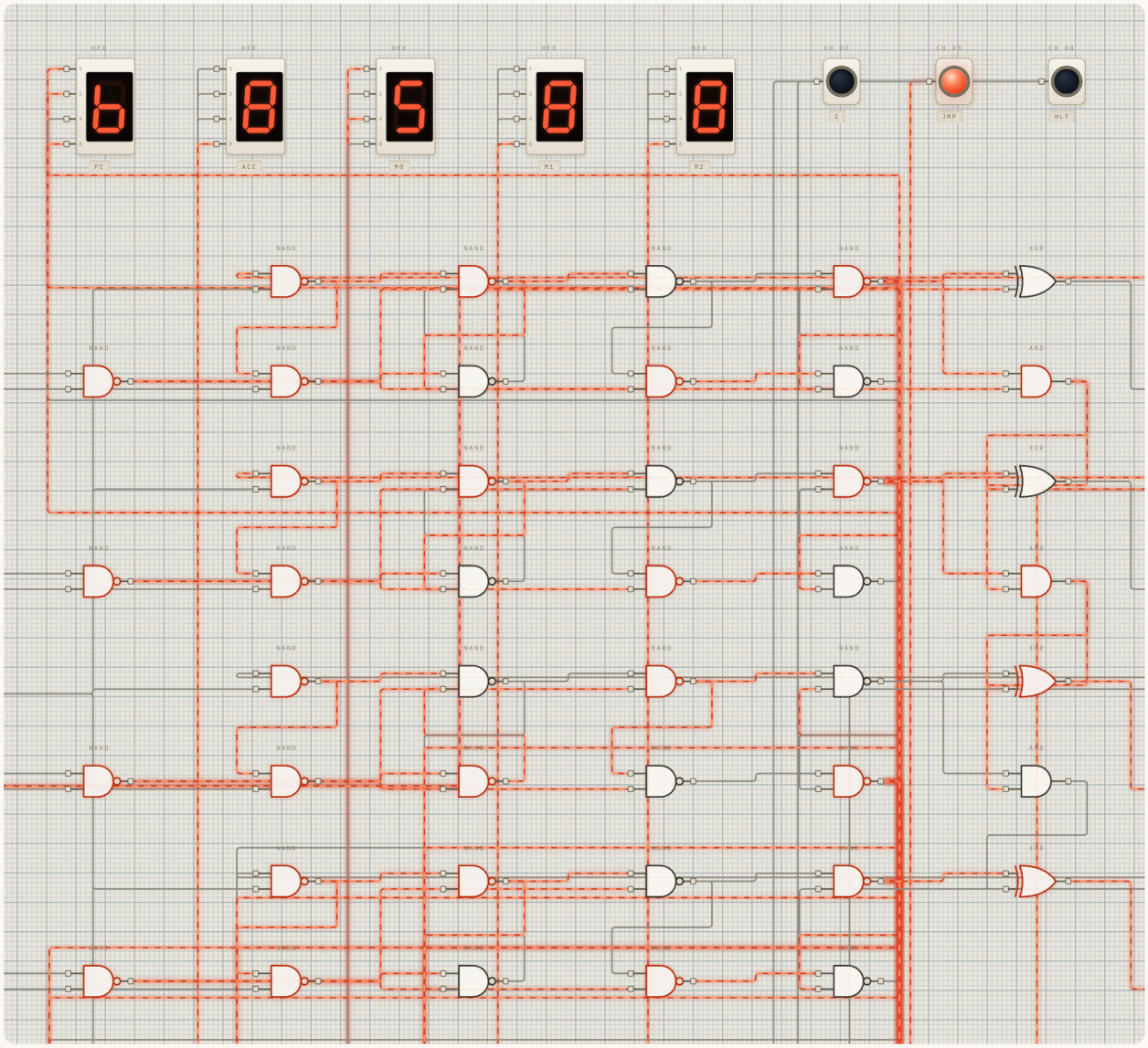
is all one addressable space the CPU manipulates. Nearly every computer in existence works this way, and here it is small enough to watch.

To show it off, Tiny CPU II runs a real program: it **computes the Fibonacci sequence mod 16** (each number is the sum of the previous two, kept to 4 bits, so it wraps around at 16). Watch it run and you will see the accumulator and RAM holding the running pair of Fibonacci numbers, each new term computed by adding the last two and stored back — exactly the read-compute-write loop that read/write memory makes possible. The program *uses its memory as scratchpad*, which the ROM-only machine never could.

As always, every block still opens. The RAM here is the gate-built 4×4 from Lesson 44 — double-click in and you will find the decoder, latches and read mux you already understand, now doing real work inside a running CPU.

See it in the bench

> **Open this:** load **Tiny CPU II (with RAM)** from the library (category **CPU**). It runs the Fibonacci-mod-16 program; watch the accumulator, the RAM contents, and the program counter as it executes.



Tiny CPU II computing Fibonacci, with values held in its RAM

Try it yourself

> **Try:** 1. Run it with the clock slowed down and watch the **accumulator** and **RAM** together. You will see the sequence build: 1, 1, 2, 3, 5, 8, 13, then wrapping mod 16 (21 → 5, …). 2. Watch a **STA** instruction store the accumulator into RAM, then later an **LDA** read it back. Those two instructions, plus the RAM, are what give the program a memory of its own results. 3. **Open the RAM block** (double-click) while it runs and watch a cell change at the moment of a store — the gate-built 4×4 RAM of Lesson 44, alive inside a CPU. 4. Compare with Lesson 50: same fetch–decode–execute cycle, but now execution can *write back* to memory. That write-back is the whole upgrade.

Recap

- **Tiny CPU II** adds a gate-built **4×4 RAM**, so the CPU can **write to memory and read it back**, not just read instructions.
- This gives the machine **variables** (via STA/LDA) and makes it a true **stored-program (von Neumann)** computer — program and data in one addressable memory.
- It runs a real **Fibonacci-mod-16** program, using RAM as scratchpad; every block, including the RAM, still opens to its gates.

Check yourself: What can Tiny CPU II do that the ROM-only Tiny CPU could not, and why does it matter? *(It can write results to RAM and read them back — giving it variables, so a program can store intermediate values and use them later, the essence of a stored-program computer.)*

Next: Lesson 52 — The Tiny CPU II (loadable): the same CPU, but now you *can load programs into it.*

Lesson 52 — The Tiny CPU II (loadable)

> **Part IX · The CPU** — library circuit (category: CPU) Before this lesson: the Tiny CPU II (Lesson 51) and the Lookup ROM (Lesson 46). After this: the LB-8 (Lesson 53).

What you will learn

- How a CPU becomes truly programmable — by *you*, without rewiring.
- What an **assembly** panel does and how a program gets loaded into memory.
- How to read a running CPU's internals through probes and a register panel.

The idea

Tiny CPU II ran a program, but that program was fixed in its design. The **loadable** version closes the final gap between "a circuit that computes something" and "a computer you program": it lets **you write a program and load it in**, then watch the same hardware run *your* code. This is the moment the machine stops being a demonstration and becomes a tool.

A few things make this work, and each is a familiar idea given a practical face:

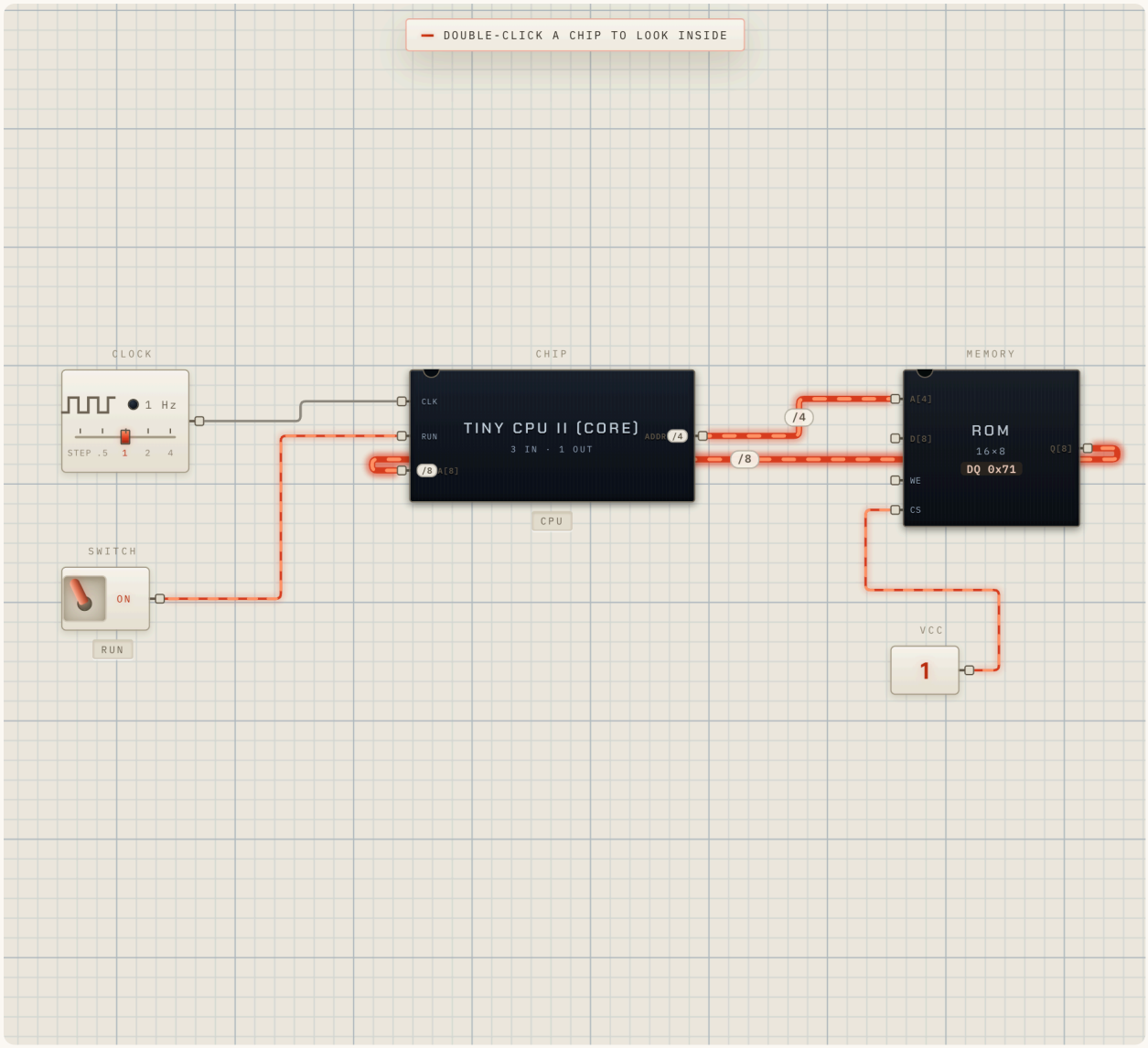
- **The program lives in a loadable ROM/memory.** Rather than being burned into the wiring, the program is written into a memory **device** (Lessons 45–46) — its contents can be set without changing a single gate. The CPU core itself is packaged as a **chip** (Lesson 42), with the program memory sitting on a **bus** (Lesson 41) beside it. Core plus loadable memory, cleanly separated.
- **An assembly (ASM) panel with LOAD.** You type a short program in a simple assembly language — `LDA`, `ADD`, `STA`, `JMP`, `JZ`, `HLT`, the

instruction set from Lesson 48 — and a **LOAD** action writes the assembled bytes into the program memory. No rewiring; you are filling memory, exactly as loading software onto a real machine fills its memory.
- **Probes and a registers panel.** To follow execution, the CPU exposes **probes** on its key signals — the **PC** (program counter), the **ACC** (accumulator), the **Z** (zero) flag, and a **FETCH** signal marking each instruction fetch — surfaced in a **REGISTERS** panel. Instead of guessing, you read the machine's state directly as it steps.

Put together, this is recognisably how you use a real computer: write code, load it, run it, and watch the registers. The hardware underneath is unchanged from Lesson 51 — same fetch–decode–execute CPU, same openable blocks — but now *the program is yours*. That separation of fixed hardware from loadable software is one of the most important ideas in all of computing, and here you can hold both halves in view at once.

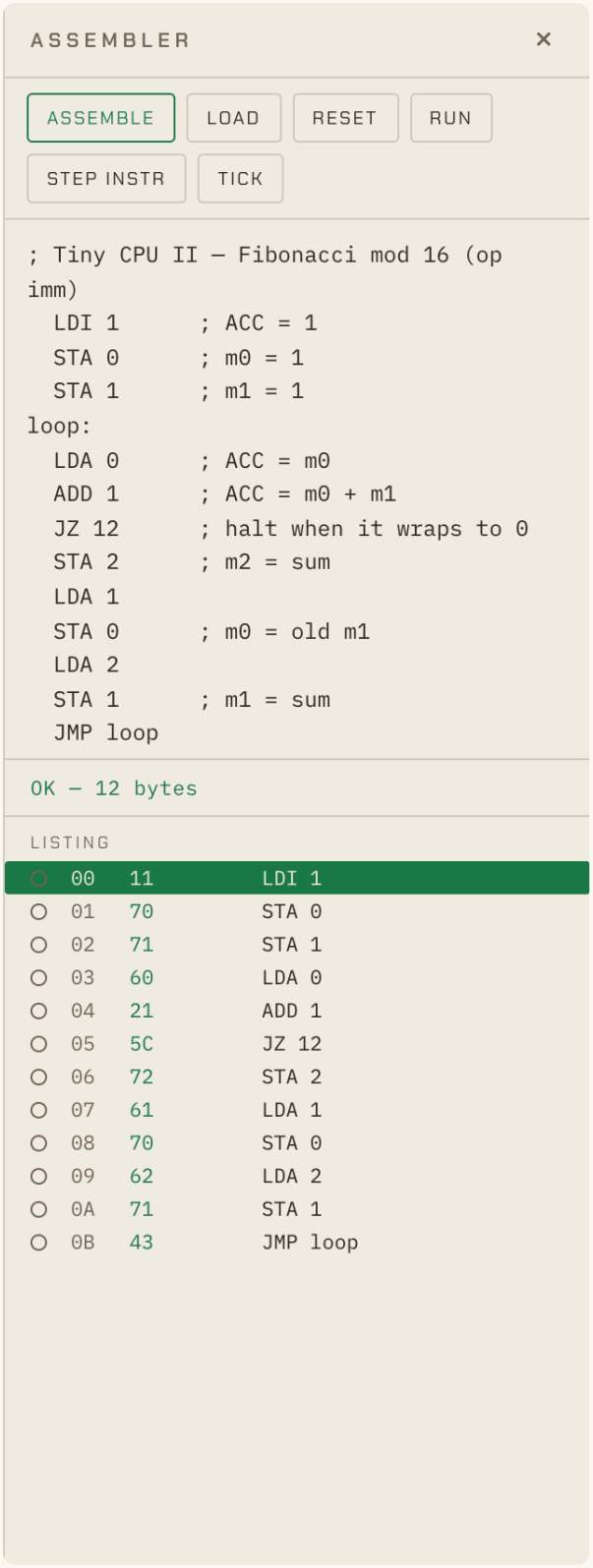
See it in the bench

Open this: load **Tiny CPU II (Loadable)** from the library (category **CPU**). Use the ASM panel to enter a short program, press LOAD, then run — watching the PC, ACC, Z and FETCH probes in the REGISTERS panel.



The loadable Tiny CPU II with its ASM panel and registers readout

The **ASM panel** is where you write the program. It shows your assembly source, the **ASSEMBLE / LOAD / RUN** controls, and — once assembled — a **listing** of each instruction with the exact bytes and address it was placed at:



The ASM panel with a program assembled to bytes

Try it yourself

Try: 1. In the **ASM panel**, enter a short program — for example: load a number, add another, store the result, then HLT. Press **LOAD** to write it into program memory. 2. **Run** with the clock slowed, and watch the **REGISTERS** panel: the **PC** climbing, the **ACC** changing as your ADD executes, the **FETCH** probe pulsing once per instruction, the **Z** flag reflecting results. 3. Write a second, different program and load it. Same hardware, new behaviour — because only memory changed. *That* is software. 4. Try a program with a **JZ** (jump if zero): do a subtraction that lands on zero and let JZ redirect the PC. Watch the program counter jump instead of simply incrementing — program flow under your control. 5. Open the CPU **core chip** to confirm the familiar ALU, decoder and accumulator are still in there, running your program on real gates.

Recap

- The **loadable** Tiny CPU II lets **you** write a program in an **ASM panel** and **LOAD** it into program memory — programmable without rewiring.
- It surfaces **probes** (PC, ACC, Z, FETCH) in a **REGISTERS** panel so you can read execution directly.
- It cleanly separates **fixed hardware** (the CPU core chip) from **loadable software** (the program in memory) — a foundational idea of computing — while every block still opens to its gates.

Check yourself: When you load a new program into this CPU, what physically changes in the hardware? *(Nothing in the gates — only the contents of the program memory change; the CPU core is unchanged. That is exactly what running different software means.)*

Writing your own programs? See Appendix A — The Assembler for the full language reference, the Tiny CPU II instruction set, and worked example programs with exact assembled bytes.

Next: Lesson 53 — The LB-8: a real, multi-cycle 8-bit CPU — the book's capstone.

Lesson 53 — The LB-8: a real 8-bit CPU

> **Part IX · The CPU** — library circuit (category: CPU) Before this lesson: the whole book — especially the Tiny CPUs (50–52), the ROM (46), and microcode-as-lookup. The capstone.

What you will learn

- How a real CPU runs each instruction over **several clock cycles** (microsequencing).
- What **registers, flags, and addressing modes** give a processor its power.
- How **microcode in a ROM** controls the whole machine — the idea from Lesson 46, fully realised.
- That even this, the most complex machine in the bench, opens all the way down to gates.

The idea

This is the summit. The **LB-8** is a genuine **8-bit CPU** — wider, more capable, and more realistic than the tiny CPUs before it. Where those executed each instruction in a single cycle, the LB-8 works the way real processors do: each instruction unfolds over **several clock cycles**, marched along by a **microsequencer**. It is the bridge from "teaching toy" to "this is how real chips are organised."

Everything you have learned converges here. Recognise the pieces:

- **Registers.** The LB-8 has more than one working register — an accumulator **A** and an index register **X** — so it can juggle several values at once, the way real CPUs do.
- **Flags.** It maintains status flags, including **Z** (zero) and **C** (carry), set by the ALU and used to make decisions — the conditional-branch machinery

you first glimpsed with the ZERO flag in Lesson 47.

- **Addressing modes.** This is a major step up. An instruction can get its operand in three different ways: **immediate** (the value is right there in the instruction), **absolute** (the value lives at a named memory address), and **absolute-indexed** (the address is computed by adding the X register — the key to walking through arrays and tables). Addressing modes are what let a small instruction set do rich, real work.
- **Board RAM on a bus.** Program and data live in memory reached over an **8-bit address bus** and **8-bit data bus** (Lesson 41's buses, at full CPU scale).

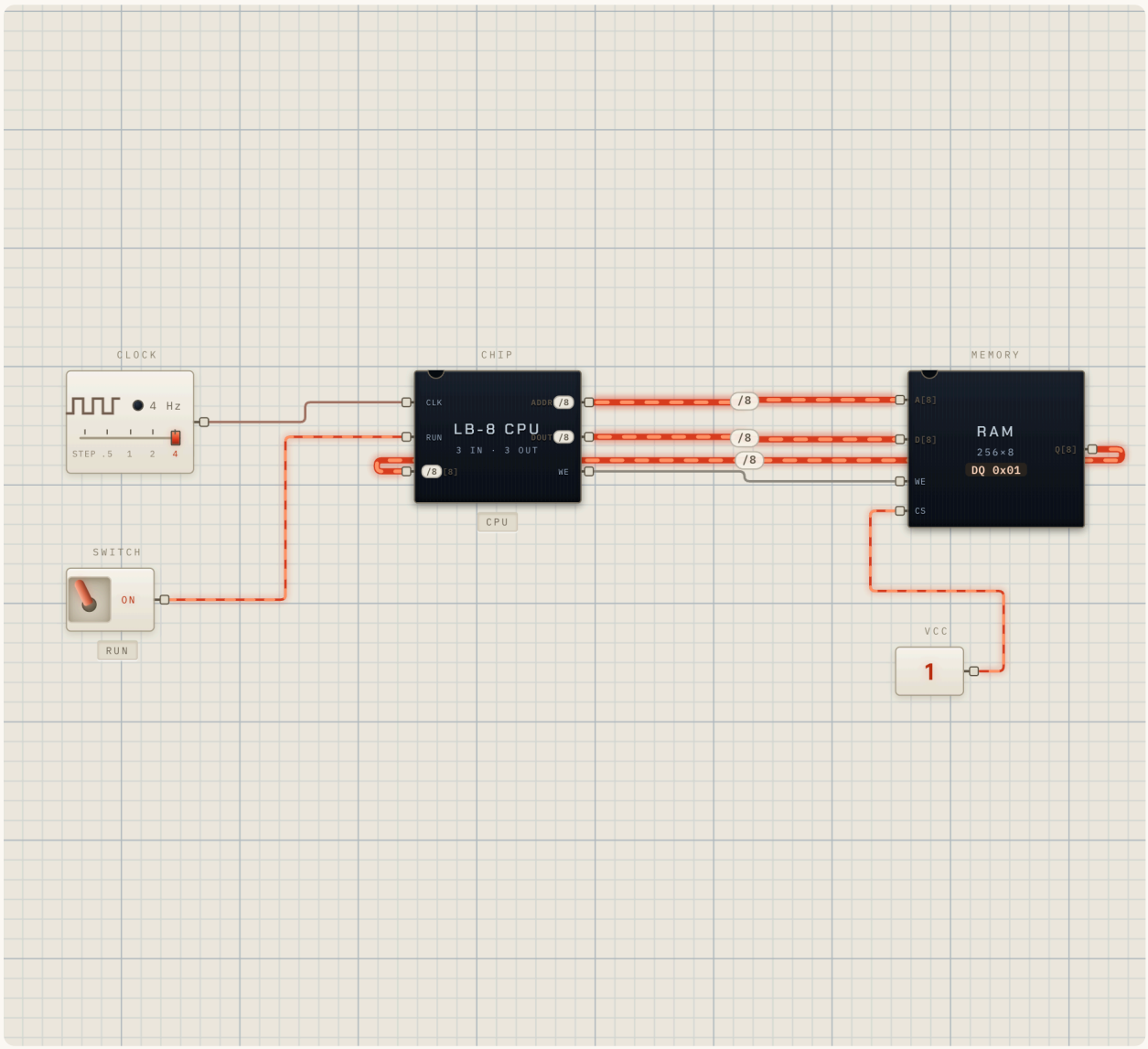
And at the centre, the masterstroke: the LB-8 is controlled by **microcode held in a ROM**. Remember Lesson 46's lesson that a ROM can implement *any* function as a lookup table, and that this is how CPUs are controlled? Here it is, real. The microcode ROM is addressed by the combination of **{opcode, T-state}** — *which instruction* you are running and *which cycle* of that instruction you are in — and at each address it stores the bundle of control signals for that exact step. The CPU runs by walking the T-states of each instruction, and at every step the ROM hands out the right control signals like a player-piano roll dictating the tune. The entire behaviour of the processor lives in that table. This is **microsequencing**, and it is genuinely how many real CPUs are built.

You also get an **assembly panel** to write and load LB-8 programs, just like the loadable Tiny CPU — so this is a CPU you can actually program, watch step through its T-states, and inspect as it runs.

Take a moment to weigh what this circuit is. It has registers, flags, three addressing modes, a bus-connected memory, and ROM-driven microcode sequencing each instruction across multiple cycles. That is not a metaphor for a CPU — it is the real organisation of one, built in front of you. And the book's promise holds even here, at the top: **every block opens.** Drill into the ALU and you reach the adder; into the adder and you reach a full adder; into that, the XOR and AND and OR of Part III; into those, single gates; into a single gate, the simple truth table of Lesson 09. From a programmable 8-bit microcoded CPU down to one AND gate, there is an unbroken chain of understanding, and not one link is magic. You have climbed the whole ladder.

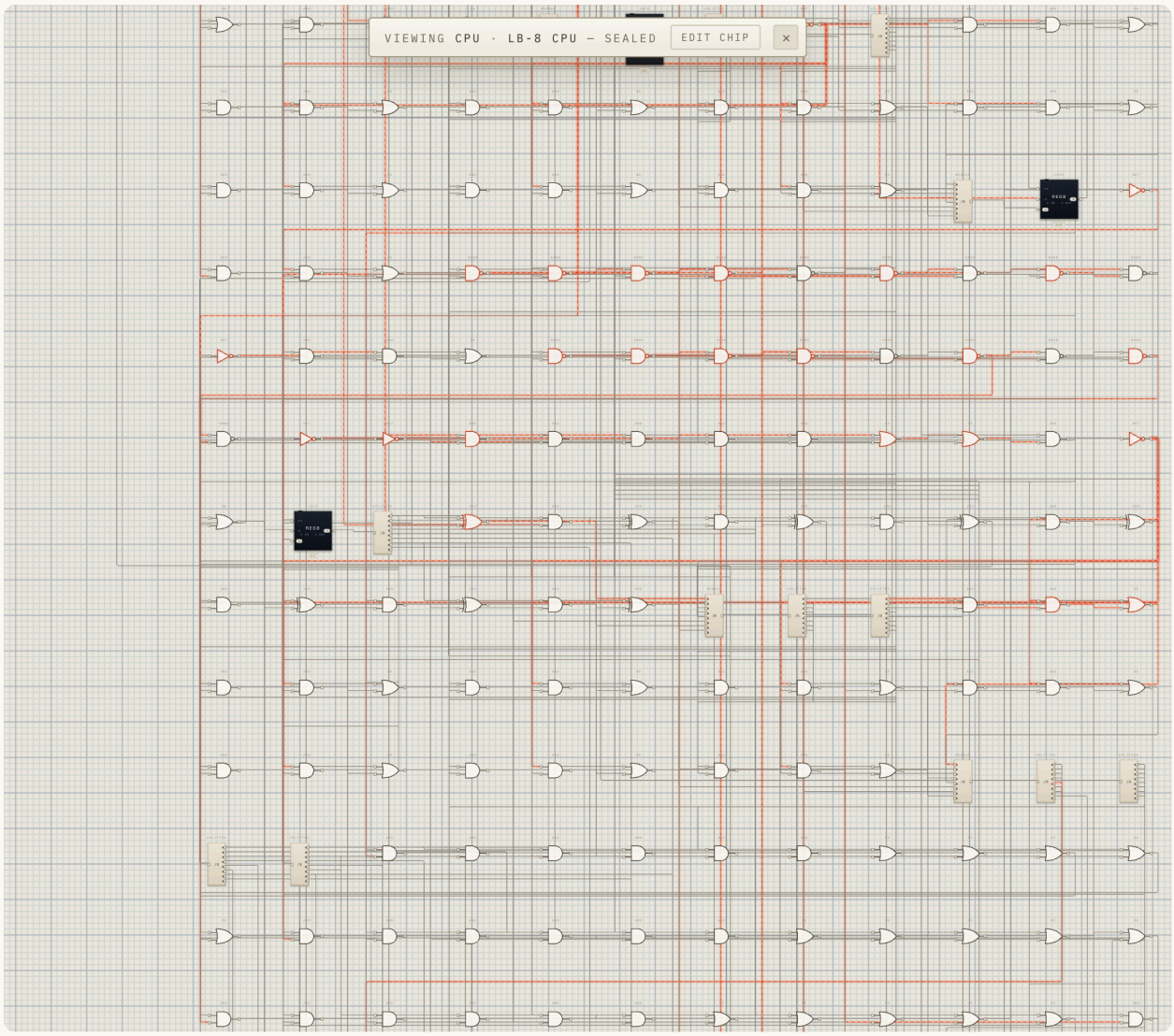
See it in the bench

Open this: load **LB-8 (Board)** from the library (category **CPU**). Use the ASM panel to load a program; run it and watch the registers (A, X), the flags (Z, C), the T-state sequencer, the address/data buses, and the board RAM.



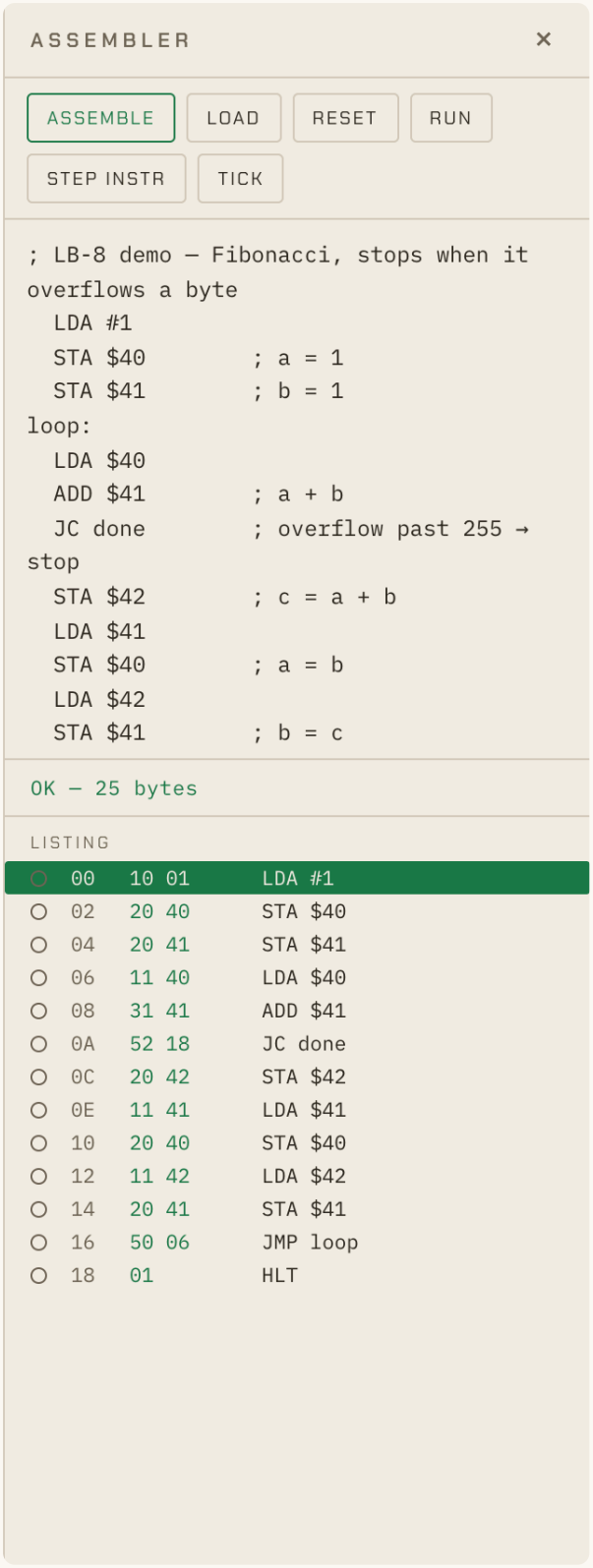
The LB-8 CPU running, registers and microsequencer in view

Then drill down: double-click into the ALU (then its adder, then a full adder, then a gate) to follow the chain from CPU to single gate.



Drilling from the LB-8 down toward a single gate

You write and load programs through the **ASM panel**, which assembles your source into bytes and lists each instruction with its address — the same assembler documented in Appendix A:



The LB-8 ASM panel with the demo program assembled

And the **REGISTERS** panel is where the machine's state is legible at a glance — the program counter, both registers, and the flags, updating as it runs:

The LB-8 REGISTERS panel showing PC, A, X, Z, C

Try it yourself

Try: 1. **Write and load a program** in the ASM panel — start simple: load A with an immediate value, add another, halt. Run it slowed down and watch register **A** change. 2. **Watch the T-states.** Slow the clock right down and observe a single instruction taking *several* cycles — fetch, then decode, then execute steps — as the microsequencer walks its T-states. Compare with the single-cycle Tiny CPU: this is the realistic, multi-cycle way. 3. **Try the addressing modes.** Use an **absolute** load to pull a value from a memory address; then an **absolute-indexed** access and change **X** to watch the effective address move — the mechanism behind stepping through an array. 4. **Use the flags.** Do an operation that sets **Z** or **C**, and a conditional branch that consults it. Watch a decision get made in hardware. 5. **Open the microcode ROM** and see the control-signal table addressed by {opcode, T-state}. Then **drill into the ALU** and keep going down until you reach a single gate. End where the book began — at one honest gate — now seen as the foundation of a working computer.

Recap

- The **LB-8** is a real **8-bit, multi-cycle CPU**: registers **A** and **X**, **Z/C** flags, three **addressing modes** (immediate, absolute, absolute-indexed), and bus-connected board RAM.
- Each instruction is **microsequenced** across **T-states**, with control signals supplied by a **microcode ROM** addressed by **{opcode, T-state}** — Lesson 46's "ROM as universal table," fully realised.
- It is programmable via an ASM panel, and — like everything in the bench — **opens all the way down to single gates**. From microcoded CPU to one AND gate, the chain of understanding is unbroken.

Check yourself: What addresses the LB-8's microcode ROM, and why does that combination make sense? *(The pair {opcode, T-state} — which instruction*

is running and which cycle of it — so the ROM can output exactly the right control signals for each step of each instruction.)

A real CPU — and a doorway

You began with a single switch lighting a single LED, and you have arrived at a programmable 8-bit CPU whose every component you can open and understand — gates, adders, latches, registers, memory, ALU, microcode — with no magic at any level. That unbroken path from one gate to a working computer is the spine of this book.

And the LB-8 is not quite the end. The next lessons give it eyes and hands — **input, output, and a screen** — and then a final Part rebuilds this whole idea as a real, historical processor: the **MOS 6502**, made of gates, running a program you can watch and take apart.

Return to the Contents at any time — or open any chip in the bench and keep exploring. Every box still opens.

> **Ready to write LB-8 programs?** Appendix A — The Assembler documents the full language, the complete LB-8 instruction set with all three addressing modes, and worked example programs (including array-walking with `X` indexing) — every listing verified against the bench's own assembler.

Next: Lesson 54 — Memory-Mapped I/O: how a CPU talks to the outside world.

Lesson 54 — Memory-Mapped I/O

> **Part IX · The CPU** — library circuit (category: CPU) Before this lesson: the LB-8 (Lesson 53). After this: the Framebuffer (Lesson 55).

What you will learn

- How a CPU talks to the outside world — switches, buttons, lights — without any new instructions.
- The idea of **memory-mapped I/O**: hardware that lives at memory addresses.
- Why "everything is memory" is one of the most powerful simplifications in computing.

The idea

The LB-8 you built in Lesson 53 could compute and remember, but it lived in a sealed box — no way to read a switch the user flips, no way to light a lamp. A real computer must talk to **hardware**. The astonishing part is *how few* new parts that takes: with the right trick, the answer is **none**. No new instructions, no new wires into the CPU — just a convention about addresses.

That trick is **memory-mapped I/O**. The idea: reserve a handful of memory addresses and wire them not to RAM cells, but to *real hardware*. Then talking to that hardware is just reading and writing those addresses with the same `LDA` and `STA` you already know. The CPU thinks it is touching memory; in fact it is touching the world.

In this board, the top of the address space is wired to I/O devices:

- `STA $F0` lights an **8-LED row**.
- `STA $F1` drives a **7-segment display**.

- `LDA $F3` reads **8 switches**.
- `LDA $F4` reads a **button**.

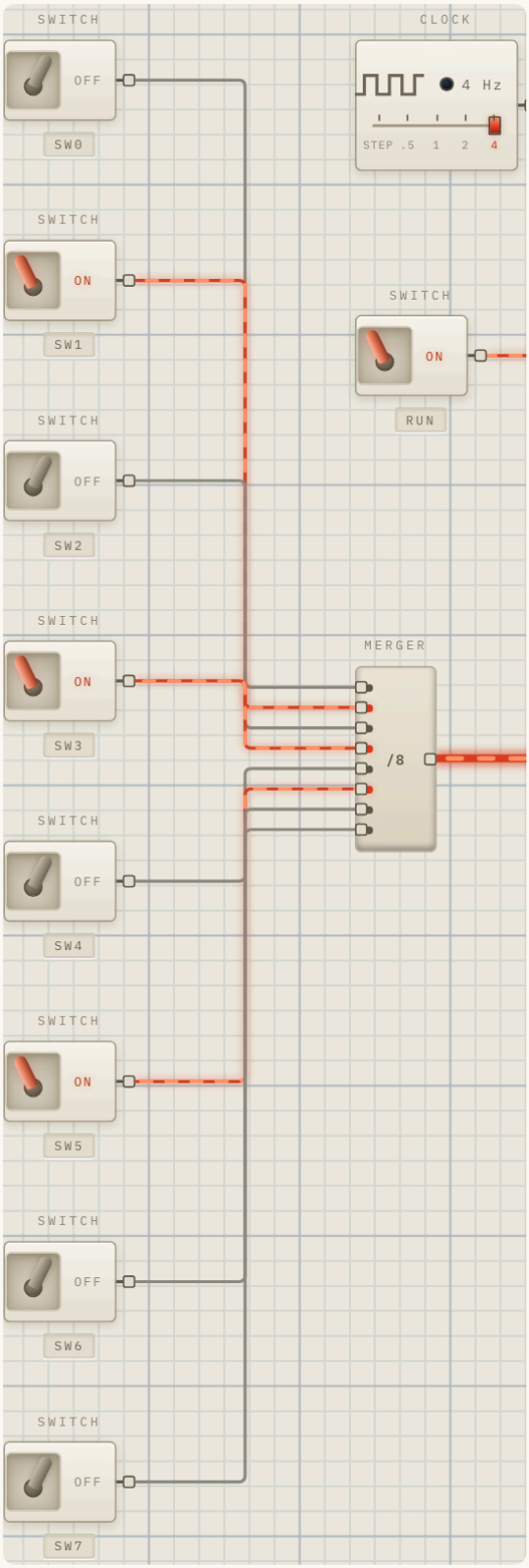
Read that list again and notice what is *not* there: no "read switch" instruction, no "light LED" opcode. Writing to address `$F0` lights LEDs for the same reason writing to any address stores a value — except an address decoder noticed the address was `$F0` and routed the write to the LED register instead of to RAM. The hardware *is* memory, as far as the program is concerned.

This is one of the great unifying ideas in computer design. A whole category of complexity — "how does the CPU communicate with dozens of different devices?" — collapses into "it reads and writes addresses, and a decoder sends each address to the right place." The keyboard, the screen, the disk, the network card in a real machine are, at bottom, addresses the CPU reads and writes.

See it in the bench

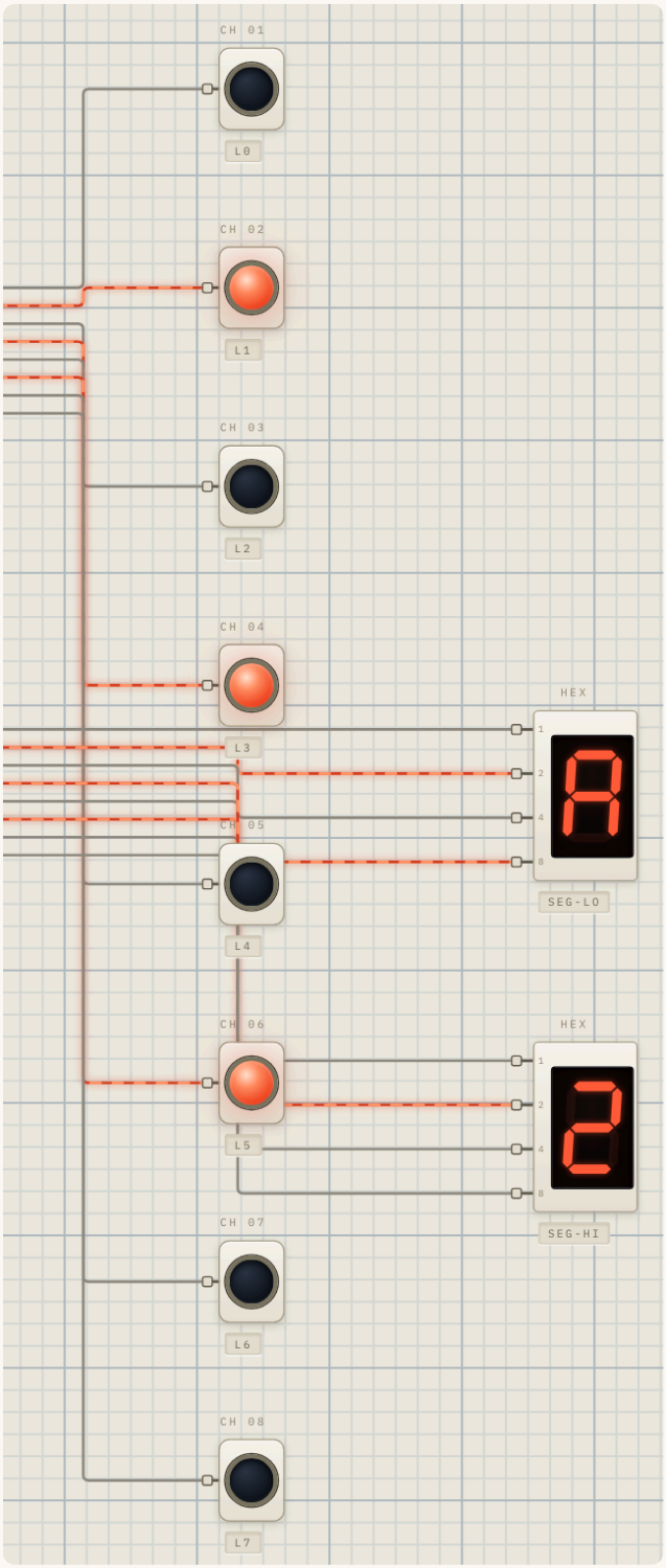
> **Open this:** load **LB-8 (I/O board)** from the library (category **CPU**). It is the same gate CPU as Lesson 53, but the top 16 addresses now reach real I/O through a drillable **IO controller** chip. The loaded demo loops `LDA $F3` / `STA $F0` / `STA $F1` forever — reading the switches and mirroring them to the LEDs and the hex display.

On the left of the board are the **8 input switches**. Set them to `0x2A` (binary `00101010`) — this is the value the program reads with `LDA $F3`:



The 8 input switches set to 0x2A on the LB-8 I/O board

On the right are the outputs the program writes — the **8-LED row** (`STA $F0`) and the **7-segment display** (`STA $F1`). They mirror the switches: the LED row shows the same `00101010` pattern, and the display reads `2A`:



The LED row and 7-segment mirroring 0x2A as the output

Read the two halves together: the program did nothing but copy address `$F3` to `$F0` and `$F1` — yet a switch on one side moved a lamp on the other. That is memory-mapped I/O: the world, reached through addresses.

Try it yourself

Try: 1. Flip **RUN** on and raise the clock. The demo is now looping: read switches, write LEDs, write display. 2. Set the **switches** to `0x2A` (binary 00101010). Watch the **LED row** mirror that exact bit pattern, and the **7-segment** show `2A`. You are watching a program read the world and write it back — live. 3. Change the switches to anything else. The LEDs and display follow within one loop. The program never "knew" about hardware; it just copied `$F3` to `$F0` and `$F1`. 4. **Double-click the IO controller chip** to see how it works: an address decoder picks out `$F0`–`$FF`, registers latch the output writes, and a mux steers reads from the switches or button. Same gate logic you met in the decoder (Lesson 37) and register (Lesson 28) lessons — now used to touch the world.

Recap

- **Memory-mapped I/O** wires hardware to memory addresses, so the CPU talks to devices using the **same** `LDA` / `STA` it uses for memory — no new instructions.
- An **address decoder** routes each address either to RAM or to a device; writing `$F0` lights LEDs because the decoder sent that write to the LED register.
- "Everything is memory" collapses device communication into reading and writing addresses — a foundational simplification in real machines.

Check yourself: How does the LB-8 light an LED without a "light LED" instruction? *(It does `STA $F0` — a normal store — and the address decoder routes that write to the LED register instead of to RAM.)*

Next: Lesson 55 — The Framebuffer: when a block of memory becomes a picture.

Lesson 55 — The Framebuffer

> **Part IX · The CPU** — library circuit (category: CPU) Before this lesson: Memory-Mapped I/O (Lesson 54). After this: the Anatomy of the 6502 (Lesson 56).

What you will learn

- The oldest idea in computer graphics: a block of memory that *is* the picture.
- How writing a byte paints a row of pixels.
- Why drawing, at bottom, is just `STA`.

The idea

In the last lesson, memory-mapped I/O let the CPU touch switches and lamps by reading and writing addresses. Now we point that same idea at a *screen*, and something wonderful falls out: **a block of memory becomes a picture.**

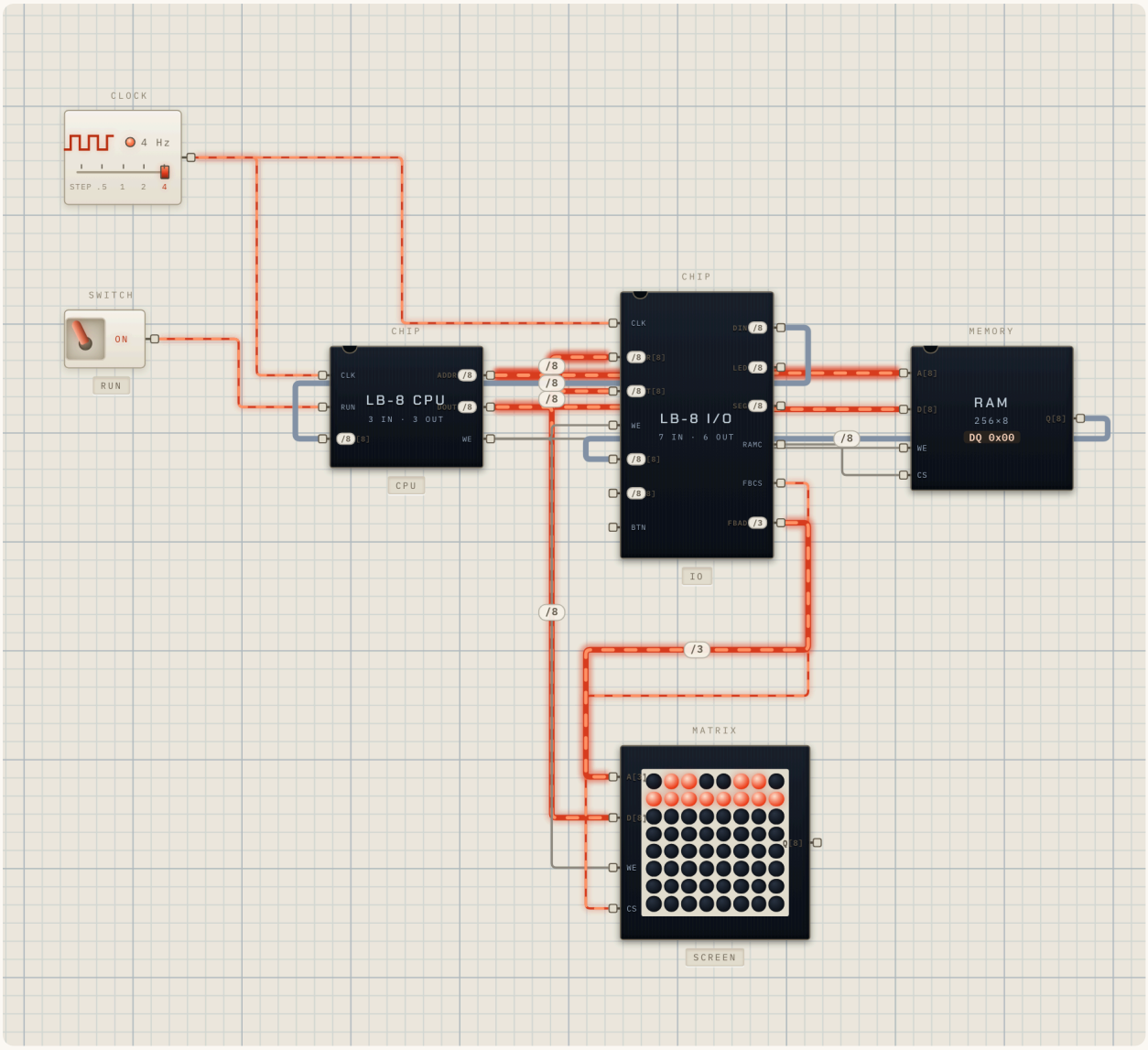
This is a **framebuffer**, and it is the oldest idea in computer graphics. The principle: a region of memory *is* the image. Each bit (or group of bits) in that memory is one pixel on the display. To draw, you do not call a "draw" routine — you simply write bytes into the framebuffer memory, and the screen shows them. The memory and the picture are the same thing, viewed two ways.

Here the **8 bytes at `$F8`–`$FF`** are the 8 rows of an 8×8 LED panel. Each bit is one pixel. So drawing a row is a single store: `STA $FA` with `A = 0x7E` paints row 2 as `.XXXXXX.` — because the bits of `0x7E` (`01111110`, MSB on the left) *are* the pixels of that row. There is no translation layer; the byte and the row of light are literally the same eight bits.

The loaded demo paints a **heart**, one `STA` per row, then halts. Eight stores, eight rows, a picture. That is the whole of it — and it is genuinely how the screen you are reading this on works, scaled up: millions of bytes in memory, each driving pixels, redrawn many times a second. The deep idea is identical to the 8×8 version you can hold in your head here.

See it in the bench

> **Open this:** load **LB-8 (graphics)** from the library (category **CPU**). The IO controller now decodes `$F8`–`$FF` to an 8×8 **SCREEN** device, passing the low 3 address bits as the row number — the same decoder logic as the I/O board, pointed at a panel.



The LB-8 graphics board with a heart painted on the 8×8 screen

Each lit pixel on that matrix is one bit of the eight bytes the program stored at `$F8`–`$FF`. Reading the picture back into bytes is just the drawing in reverse: row 2 showing `...XX...` *is* the byte `0x18` (`00011000`), because the bits and the pixels are the same eight values. The memory and the picture are not two things linked by a translation step — they are one thing, seen as numbers or seen as light.

Try it yourself

Try: 1. Flip **RUN** on and raise the clock. Watch the heart draw, one row per `STA`, then the machine halts. 2. **Read the matrix as bytes.** Pick a row of the heart and work out its eight bits from the lit and dark pixels, MSB on the left — that binary value is exactly the byte the program stored to that row's address. Confirm a clear one: a row showing `...XX...` is `0x18`. 3. **Map rows to addresses.** The top row is `$F8`, the next `$F9`, down to `$FF` for the bottom — the low three address bits *are* the row number. Trace which `STA` in the demo painted which row. 4. For a challenge: open the ASM panel and write a short program that animates a single bouncing pixel by storing a moving bit pattern to successive rows each frame.

Recap

- A **framebuffer** is a block of memory that *is* the picture — each bit is a pixel.
- Drawing is just `STA`: writing a byte to a framebuffer address lights that row's pixels, because the bits are the pixels.
- This is the real mechanism behind every screen, scaled down to 8×8 so you can see every bit.

Check yourself: To paint row 2 of the panel as `.XXXXXX.`, what do you do? *(Store the byte `0x7E` to that row's address — `STA $FA` — because its bits are the row's pixels.)*

Next: Lesson 56 — The Anatomy of the 6502: we leave the LB-8 behind and meet a real, historical processor.

Part X — The 6502

Lesson 56 — The Anatomy of the 6502

> **Part X · The 6502** — library circuit (category: CPU) Before this lesson: the whole book, especially the LB-8 (Lesson 53). The opening of the final Part.

What you will learn

- What the MOS 6502 is, and why it earns its own Part in this book.
- The map of a real processor: its registers, its datapath, its control.
- How the machine you are about to take apart relates to everything you have already built.

The idea

Everything so far has built *toward* a real computer. The LB-8 was the first real CPU — multi-cycle, microsequenced, with addressing modes. But it was ours, a teaching machine. Now we meet a CPU that **shipped to the world**: the **MOS 6502**, the processor at the heart of the Apple II, the Commodore 64, the BBC Micro, the Nintendo Entertainment System. Millions of people's first computer ran a 6502. It is one of the most important chips ever made — and in this bench you can build it from gates and watch it run.

This Part takes the 6502 apart and puts it back together. This first lesson is the **map** — the whole machine seen at once, so that when we open each block in the lessons that follow, you know where it sits and what it is for. (We start with the *behavioral* 6502: a working model whose insides are a verified emulator. The gate-built version, where every one of these blocks is real logic, is the climax in Lesson 62.)

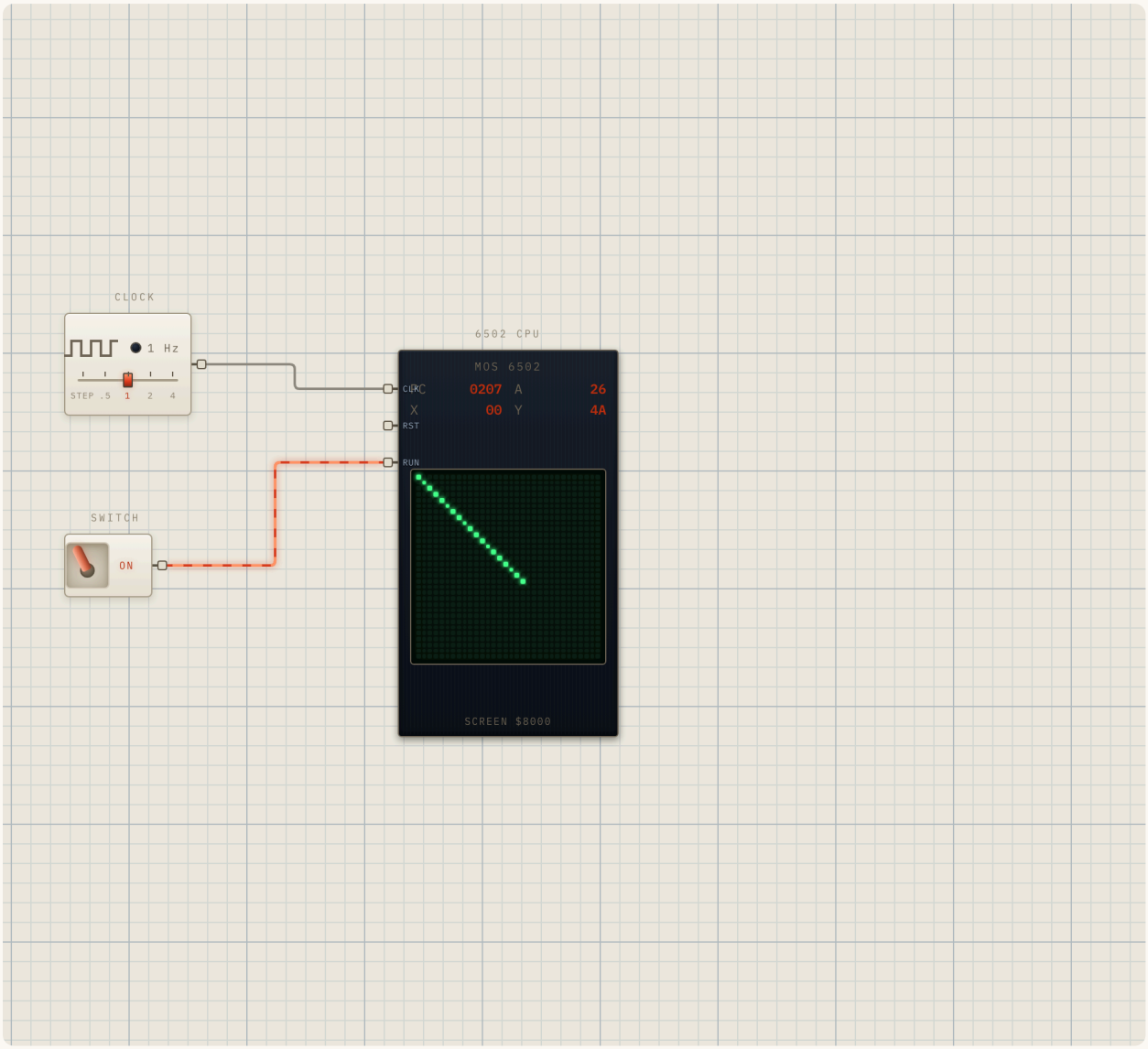
A 6502 is organised around a few parts, and you have met the principle behind every one of them:

- **Registers** — small stores of state. The 6502 has an accumulator **A** (the working value), two index registers **X** and **Y** (for stepping through data), a stack pointer **S**, the program counter **PC**, and the status register **P** (the flags). Every one is a *register* in the sense of Lesson 28.
- **The ALU** — the calculator (Lesson 47), here 8 bits wide with the 6502's own operation set.
- **The program counter** — your place in the program (Lesson 49), here a full **16 bits** so it can address 64K of memory.
- **The stack pointer** — a new idea, coming in Lesson 60.
- **The flags** — status bits (Lesson 47's Z and C), here a richer set including a famous *decimal mode*.
- **The control unit** — microcode in ROM, addressed by (opcode, T-state), sequencing the datapath one step at a time. Exactly the LB-8's mechanism (Lesson 53), now driving a real ISA.

Look at that list and feel what it means: **you already understand every part of a real, historical CPU.** The 6502 is not a new kind of thing — it is the registers, ALU, counter, flags, and microcode you have built, arranged with care and proven against the original. This Part simply opens each box.

See it in the bench

Open this: load **6502 CPU (behavioral)** from the library (category **CPU**). It is a working 6502 — its "inside" is an emulator verified against the Klaus Dörmann test suite, the standard proof-of-correctness for 6502 implementations. Its registers (PC, A, X, Y, S, P) are live in the **REGISTERS** panel, and its face carries a 32×32 screen showing the framebuffer at `$8000`.



The behavioral 6502 with its registers panel and 32×32 screen

Try it yourself

Try: 1. Flip **RUN** at 1 Hz and watch the demo draw a diagonal on the 32×32 screen, pixel by pixel, while the **REGISTERS** panel shows PC, A, X, Y, S, P moving. This is a real 6502 program running. 2. Drop the clock to .5 Hz to go slower, or crank it to 4 Hz to fill the screen at once. The clock is the speed knob, exactly as on every machine in this book. 3. Set the clock to **STEP** and use **STEP INSTR** to execute one instruction at a time. Watch a single instruction change exactly the registers it should — the current instruction is marked in the ASM listing. 4. Read the register names and match each to a lesson: A → the accumulator (L28/L47), PC → the program counter (L49), P → the flags (L47). You are looking at an old friend in a famous form. 5. For a livelier demo, load **6502 — Bouncing Ball** — the same behavioral 6502 running a real program that bounces a pixel around the screen. It is the playful preview of Lesson 63, where that very program runs on the *gate-built* 6502.

Recap

- The **MOS 6502** is a real, historically pivotal CPU — and every one of its parts is something you have already built.
- Its anatomy: registers (A, X, Y, S, PC, P), an 8-bit ALU, a 16-bit PC, a stack pointer, a rich flag set, and microcode control.
- This Part opens each block in turn; the **behavioral** 6502 here is the map, the **gate-built** one (Lesson 62) is the territory.

Check yourself: The 6502's program counter is 16 bits wide, not 8. What does that width buy it? *(A 16-bit address can reach 2^{16} = 65,536 locations — the 6502's full 64K memory.)*

Next: Lesson 57 — The 6502 ALU: the calculator, revisited at 8 bits.

Lesson 57 — The 6502 ALU

> **Part X · The 6502** — building block (chip: ALU6502) Before this lesson: the Anatomy (Lesson 56) and the 4-Bit ALU (Lesson 47). After this: the 16-Bit Adder (Lesson 58).

What you will learn

- How the 6502's ALU relates to the 4-bit ALU you already built.
- What its operation set is.
- Why this is a short lesson — you already know the idea.

The idea

You built an ALU in Lesson 47: a calculator that selects among operations with a control code, does add and subtract through one adder using the two's-complement trick, and raises flags. The **6502's ALU is that same circuit, grown up.** If Lesson 47 made sense, this one is mostly a matter of scale and a slightly larger menu.

What is the same: it is an arithmetic-logic unit driven by an operation select, with a shared adder that subtracts via B-invert and carry-in, producing a result plus status flags. Everything you understood at 4 bits holds.

What is new, and worth noting:

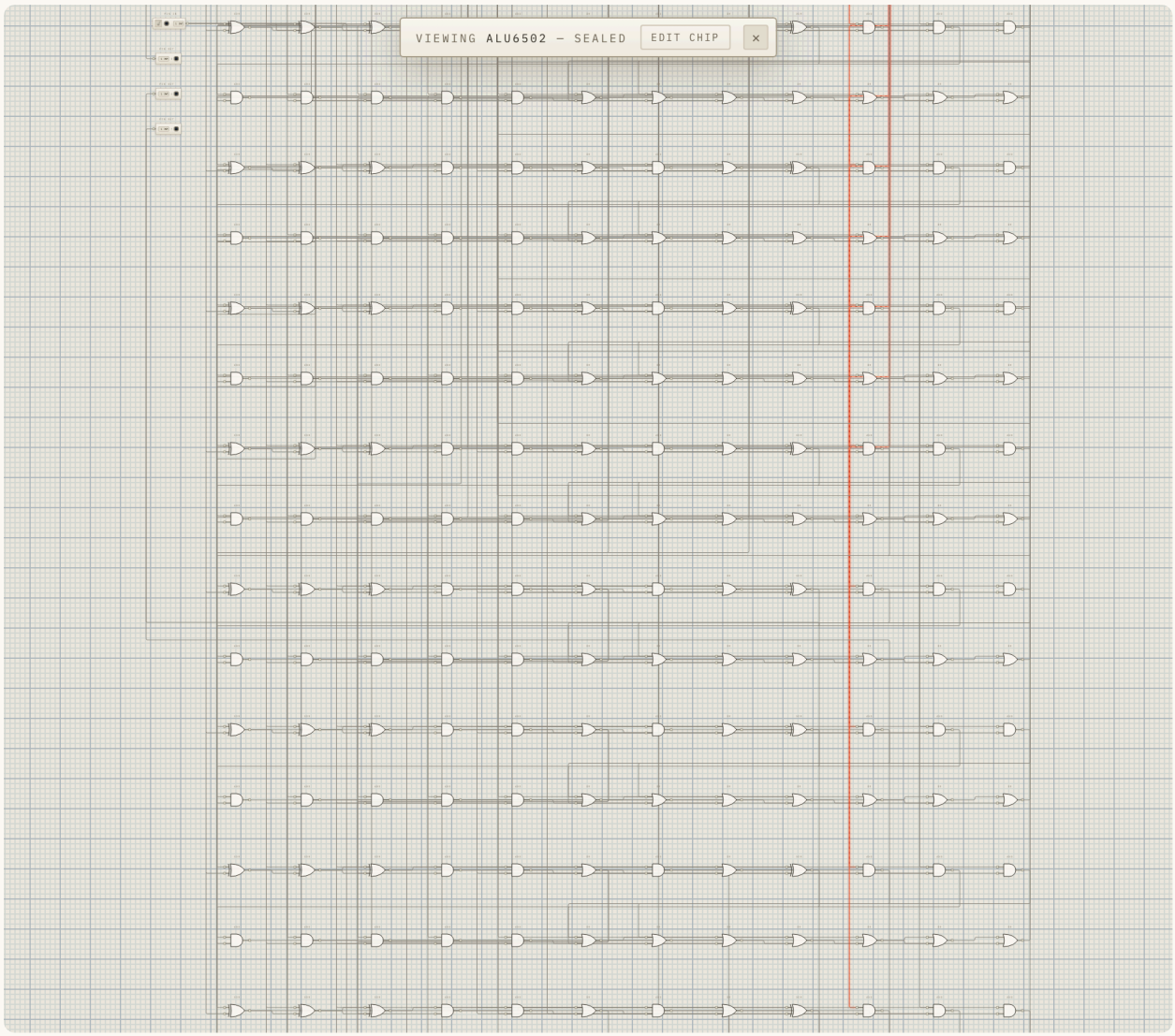
- **It is 8 bits wide.** Wider operands, same structure — exactly the "once a stage chains, width is free" lesson from the 8-bit adder (Lesson 21).
- **Its operation menu is the 6502's own:** add, AND, OR, EOR (exclusive-or), and the shifts (shift-left, shift-right), selected by a 3-bit operation code. The logic operations are the bitwise gates you have known since Part I, applied across all eight bits.

- **It outputs both C (carry) and V (overflow).** You met C in Lesson 47. **V**, the signed-overflow flag, is new here — it signals when a signed addition produced a result too large to fit, which matters for signed arithmetic. It is computed from the carry into and out of the top bit.

That is the whole lesson. The 6502 ALU is not a new idea; it is the ALU you built, eight bits wide, with the 6502's operation set and one extra flag.

See it in the bench

> **Open this:** the 6502 ALU lives inside the gate-built 6502 (Lesson 62). To see it on its own, load the **6502 — Gate-Built** preset and **double-click into the M6502 chip**, then into the **ALU6502** block; or find the ALU6502 chip on the CHIPS shelf once a gate-6502 board is loaded.



The ALU6502 block: an 8-bit arithmetic-logic unit

Try it yourself

Try: 1. Compare it to the 4-bit ALU from Lesson 47 side by side. Same shape — operands in, operation select, result and flags out — just wider and with EOR and shifts added. 2. Find the **B-invert** path: this is how one adder does both add and subtract, exactly as in Lesson 47. 3. Look for the **V** output and recall it is the *signed* overflow flag — the new one. C answers "did it carry past 8 bits?"; V answers "did a signed result overflow?"

Recap

- The **6502 ALU** is the Lesson 47 ALU at **8 bits**, with the 6502's operation set (add, AND, OR, EOR, shifts) and an added **V** (signed overflow) flag alongside **C**.
- Its add/subtract still rides one adder via B-invert and carry-in; its logic ops are the bitwise gates from Part I.
- Nothing here is a new principle — it is a known circuit, scaled and outfitted for a real ISA.

Check yourself: The 6502 ALU outputs both C and V. What is the difference? *(C is the unsigned carry out of bit 7; V is the signed-overflow flag, set when a signed result is too large to fit.)*

Next: Lesson 58 — The 16-Bit Adder: how the 6502 computes addresses.

Lesson 58 — The 16-Bit Adder

> **Part X · The 6502** — building block (chip: ADD16) Before this lesson: the 6502 ALU (Lesson 57) and the adders of Part III. After this: the Program Counter (Lesson 59).

What you will learn

- Why the 6502 has a *second* adder, separate from its ALU.
- What an "effective-address" calculation is.
- Why this, too, is a short lesson built on what you know.

The idea

The 6502 has two adders. One is inside the ALU (Lesson 57), for arithmetic on data. The other — **ADD16** — does a different job: it computes **addresses**. This lesson is short because the adder itself is no mystery; what is worth understanding is *why a separate one exists*.

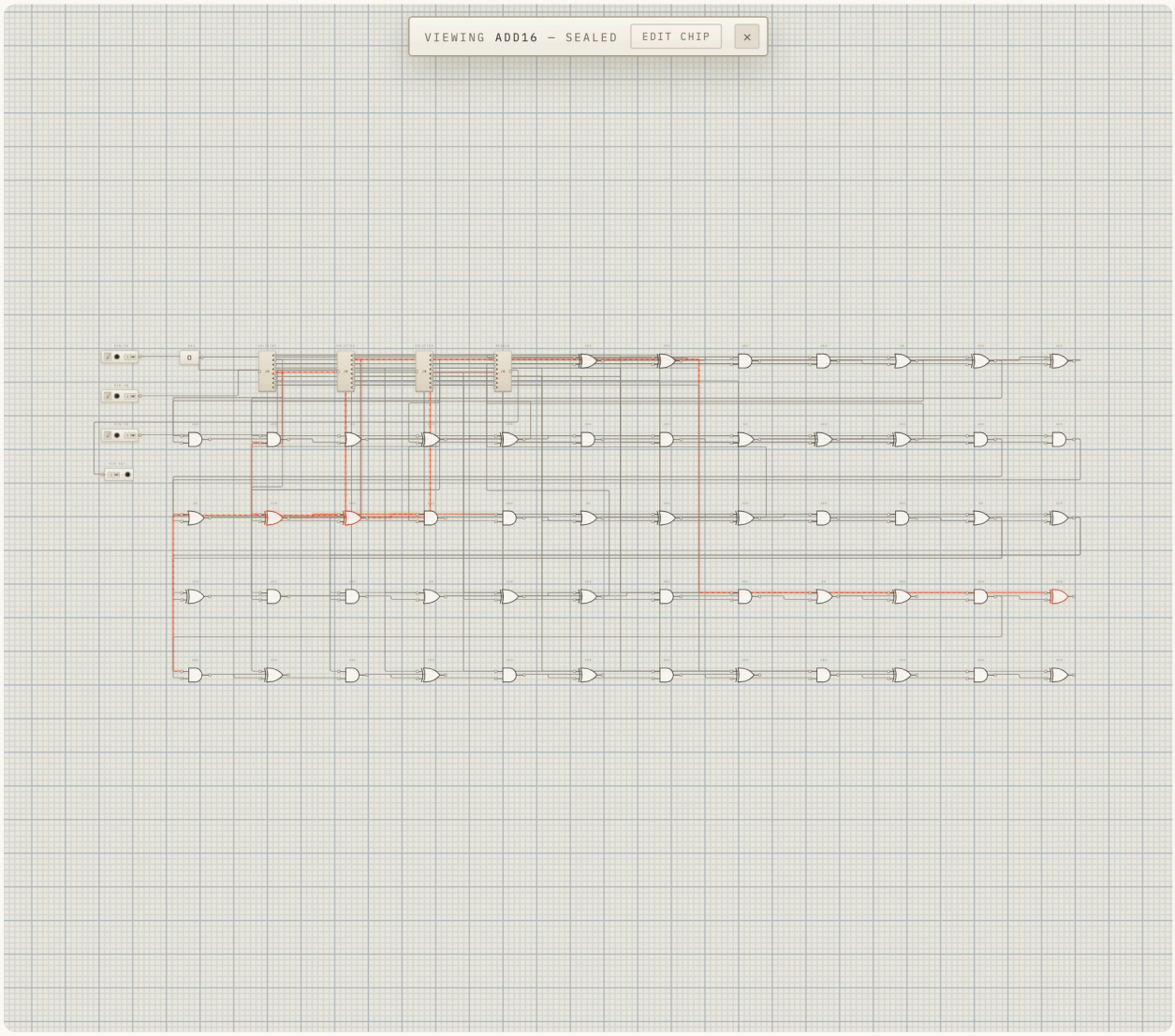
Recall the addressing modes from the LB-8 (Lesson 53): absolute-indexed access reaches `address + X`. To find that final address — the **effective address** — the CPU must *add*. When a program says "load from the table at `$2000`, offset by the index in X," the hardware computes `$2000 + X` to know where to actually look. That sum is 16 bits wide (addresses span 64K), and computing it is the ADD16's job: it takes a 16-bit base `{ADH, ADL}` and adds a (zero-extended) index, producing the 16-bit effective address.

Why not reuse the ALU? Because address arithmetic and data arithmetic often need to happen close together, and a dedicated address adder keeps the datapath clean and fast. This is a real design pattern in processors: a separate path for computing *where*, distinct from the path computing *what*.

And the adder itself? It is the ripple-carry adder of Part III (Lessons 20–21), 16 bits wide. You already know exactly how it works — carry rippling from the low byte into the high byte. The "once a stage chains, width is free" lesson applies one more time: 16 full-adder stages, the same as 8, the same as 4.

See it in the bench

> **Open this:** drill into the **6502 — Gate-Built** preset (Lesson 62) and open the **M6502** chip, then the **ADD16** block.



The ADD16 block: a 16-bit effective-address adder

Try it yourself

> **Try:** 1. Recognise the structure: a 16-bit ripple-carry adder, the Part III circuit at double the 8-bit width. The carry travels from the low byte up through the high byte. 2. Think through *why it is here*: every indexed memory access (`LDA $2000,X`) needs `$2000 + X` computed before the load can happen. This adder does that, every time, separately from the ALU. 3. Contrast the two adders in the 6502: the ALU's adds *data*; the ADD16 adds *addresses*. Same circuit, two jobs, two locations.

Recap

- The 6502 has a **dedicated 16-bit adder (ADD16)** for computing **effective addresses** (e.g. `base + index`), separate from the ALU's data adder.
- It is the **ripple-carry adder** of Part III, 16 bits wide — nothing new in the adder, only in its purpose.
- Separating "compute where" from "compute what" is a real processor design pattern.

Check yourself: Why does the 6502 need to *add* in order to do `LDA $2000,X`? *(It must compute the effective address `$2000 + X` before it knows where to load from — that 16-bit sum is the ADD16's job.)*

Next: Lesson 59 — The 16-Bit Program Counter: keeping your place across 64K.

Lesson 59 — The 16-Bit Program Counter

> **Part X · The 6502** — building block (chip: PC16) Before this lesson: the 16-Bit Adder (Lesson 58) and the Program Counter (Lesson 49). After this: the Stack Pointer (Lesson 60).

What you will learn

- How the 6502's program counter extends the one you built.
- The three things a real PC must do: increment, jump, and branch.
- Why branches use a *signed* offset.

The idea

You built a program counter in Lesson 49: a counter that holds the current address and increments to the next instruction, and that can be *loaded* with a new value for jumps. The **6502's PC16 is that same idea at 16 bits**, with one addition worth a closer look — branches.

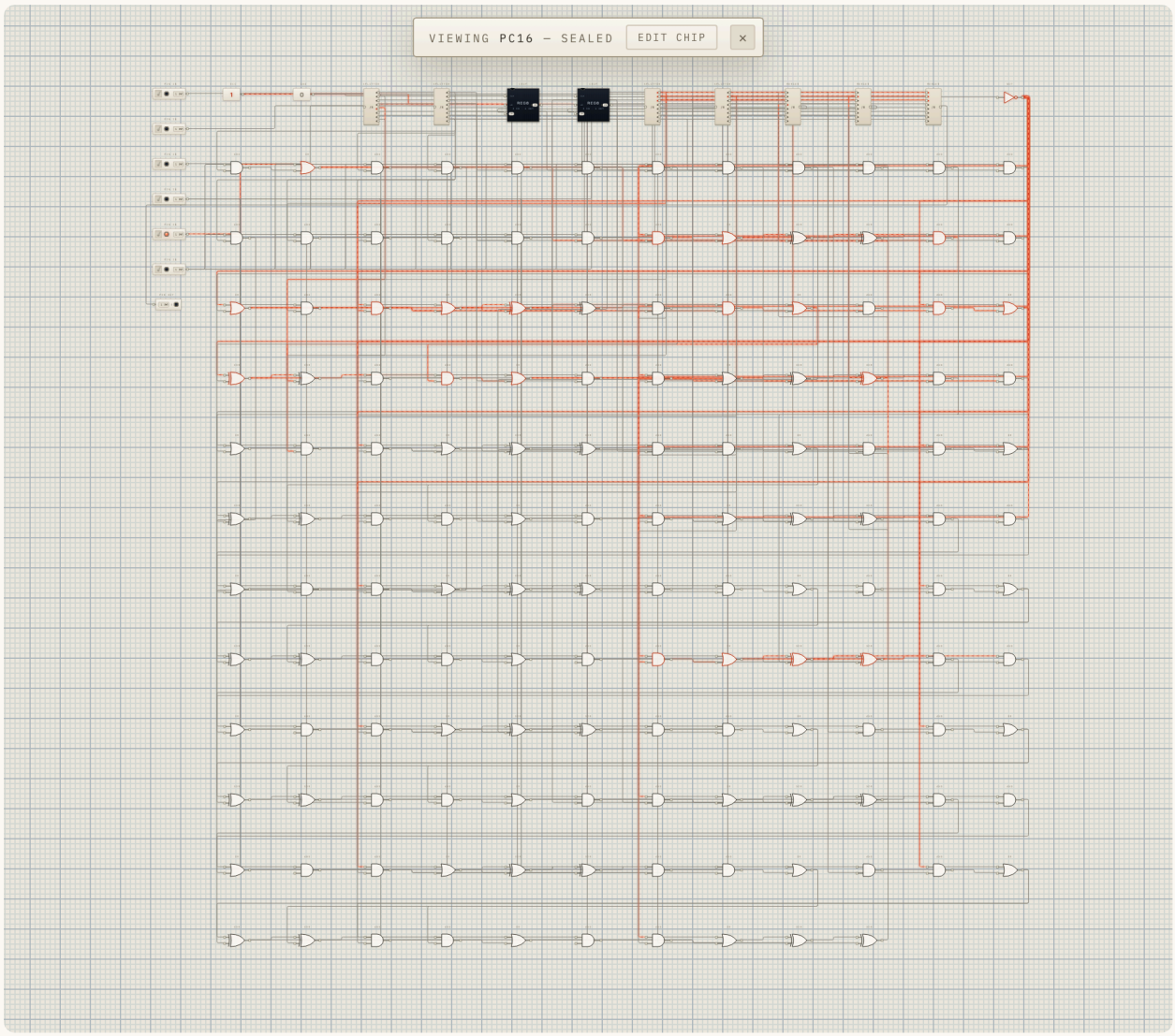
A real program counter must do three things:

- **Increment** — advance to the next instruction during fetch. This is the counter behaviour from Lesson 49, sixteen bits wide so it can walk all of 64K.
- **Load** — take a whole new address at once, for `JMP`, `JSR` (jump to subroutine) and `RTS` (return). This is the "load instead of increment" power you already met; it is how jumps redirect the flow.
- **Branch** — and this is the new one. A 6502 *branch* (like "branch if zero") does not carry a full address. It carries a small **signed offset** — a single byte interpreted as a number from −128 to +127 — and the PC adds it to itself. Branch forward with a positive offset, backward with a negative one.

Why signed, and why small? Because branches are almost always *nearby* — the top of a loop a few instructions back, the end of an `if` a few instructions ahead. A one-byte signed offset reaches either direction within a short range, cheaply. This is a real and clever economy: most control flow is local, so the common case is made compact. (Long-distance jumps use the full-address `JMP` instead.) The signed offset is exactly the two's-complement idea from the arithmetic Part, now steering *where the program goes* rather than computing a sum.

See it in the bench

> **Open this:** drill into the **6502 — Gate-Built** preset (Lesson 62), open the **M6502** chip, then the **PC16** block.



The PC16 block: a 16-bit program counter with load, increment and branch

Try it yourself

Try: 1. Recognise the **increment** path — this is the Lesson 49 counter, 16 bits wide, advancing each fetch. 2. Find the **load** path: a whole new 16-bit address dropped in at once. That is `JMP` / `JSR` / `RTS` — the jump mechanism you already understand. 3. Find the **branch** path: here the PC *adds* a signed byte to itself. Picture a loop — the branch at the bottom carries a small negative offset that sends the PC back up to the top. Local control flow, made cheap.

Recap

- The **PC16** is the Lesson 49 program counter at **16 bits**, doing three jobs: **increment** (fetch), **load** (JMP/JSR/RTS), and **branch**.
- A **branch** adds a **signed one-byte offset** (−128…+127) to the PC — compact because control flow is usually local.
- The signed offset is two's-complement (Part III) applied to *program flow* instead of arithmetic.

Check yourself: Why can a 6502 branch only reach nearby instructions, and why is that usually fine? *(Its offset is a single signed byte, ±127 — and most branches target nearby code, like the top of a loop, so the short reach is rarely a limit; distant targets use JMP.)*

Next: Lesson 60 — The Stack Pointer: a new idea — memory that remembers how to come back.

Lesson 60 — The Stack Pointer

> **Part X · The 6502** — building block (chip: SP8) Before this lesson: the Program Counter (Lesson 59) and the Register (Lesson 28). After this: the Flags & Decimal Mode (Lesson 61).

What you will learn

- What a **stack** is, and the problem it solves that nothing in this book has solved yet.
- How a single counting register — the stack pointer — implements it.
- How the stack lets a program *call a subroutine and find its way back*.

The idea

Here is a genuinely new idea — the first in a while. Every part of the 6502 so far has been a wider or better-dressed version of something you already built. The **stack** is different: it solves a problem the book has not yet touched, and it does so with surprising economy.

The problem: how does a program come back? Suppose your program calls a subroutine — a reusable piece of code, say "draw a pixel." When the subroutine finishes, it must return to *wherever it was called from*. But it might be called from many different places. So the CPU must, at the moment of the call, *remember the address to come back to* — and if the subroutine itself calls another subroutine, it must remember *those* return addresses too, in the right order, like a stack of notes where you always take the top one first.

The solution: a stack. A stack is a region of memory used in **last-in, first-out** order — the last thing you put on is the first thing you take off, exactly like a stack of plates. To call a subroutine, the CPU **pushes** the return address onto the stack; to return, it **pops** the top one back off. Nested calls just stack up and

unwind in reverse. This is how a program can dive several subroutines deep and always find its way back out.

The clever part: it takes just one register. The whole machinery is a single 8-bit register — the **stack pointer (S)** — that holds the address of the top of the stack. The SP8 chip can do exactly three things to it:

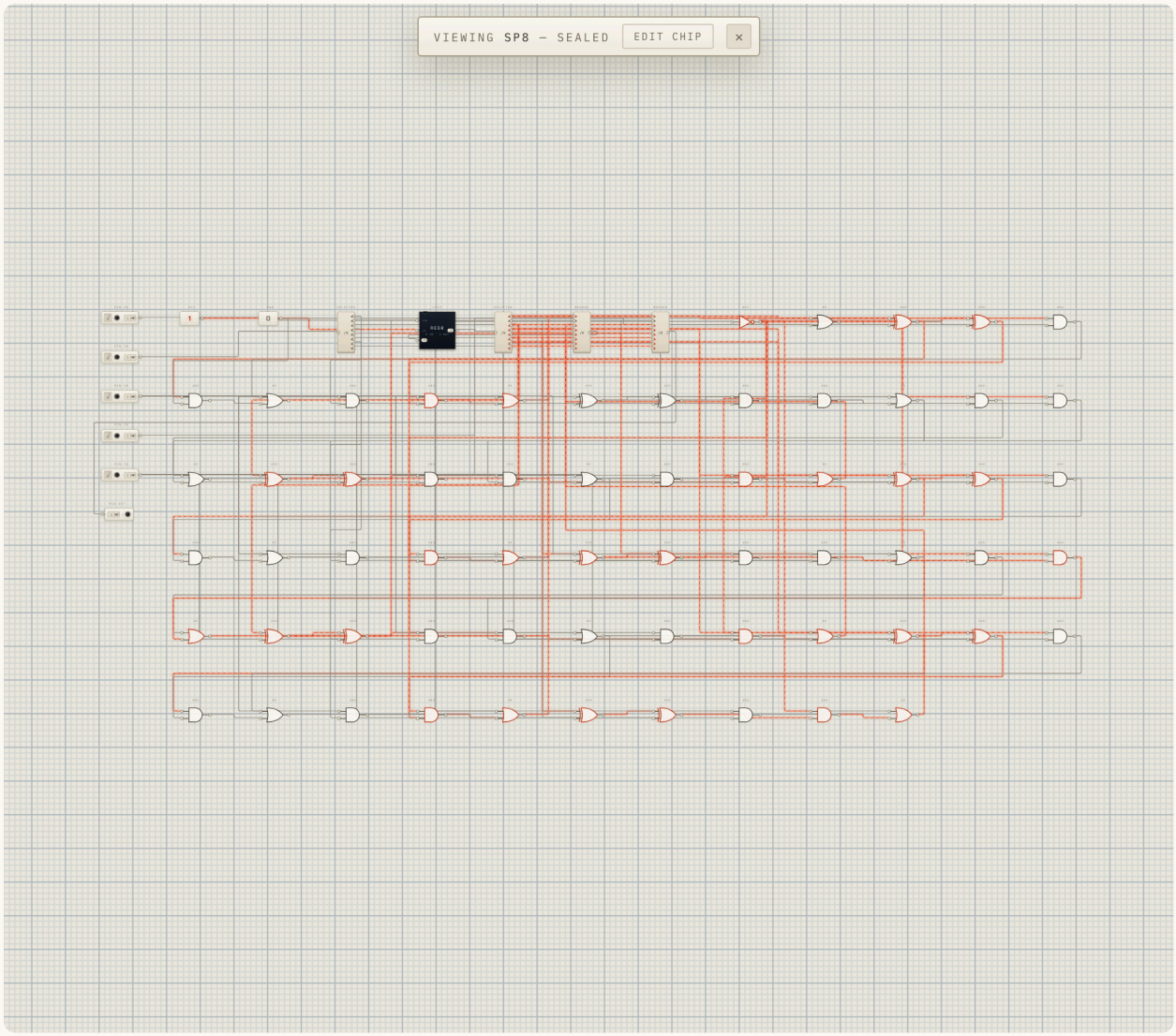
- **Load** it (the `TXS` instruction sets the stack pointer from X — used to initialise the stack).
- **Decrement** it (a *push* — the stack grows downward in memory as you add to it).
- **Increment** it (a *pop* — the stack shrinks as you remove from it).

That is the entire stack pointer: a register that counts down when you push and up when you pop. Combined with normal memory writes at the address it points to, those three operations give you the complete last-in-first-out stack — and therefore subroutines, nested calls, and return addresses. One small counting register unlocks one of the most important structures in all of programming.

This is also why the 6502 reserves a special place — "page one," addresses `$0100`–`$01FF` — for its stack: the 8-bit S register names a location within that 256-byte page. A small register, a reserved page, and the whole call-and-return mechanism falls out.

See it in the bench

> **Open this:** drill into the **6502 — Gate-Built** preset (Lesson 62), open the **M6502** chip, then the **SP8** block.



The SP8 block: an 8-bit stack pointer with load, push and pop

Try it yourself

Try: 1. Identify the three operations on the SP8: **load** (TXS, to set up the stack), **decrement** (push), **increment** (pop). It is a register (Lesson 28) that can also count up or down. 2. Picture a subroutine call: the CPU pushes the return address (S decrements), the subroutine runs, then a return pops it back (S increments) and the program counter jumps there. Trace that in your head — push, run, pop, return. 3. Picture *nested* calls: each push stacks another return address; each return pops the most recent. Last in, first out — the plates analogy, in hardware. 4. Recall from Lesson 56 that the behavioral 6502 powers up with S winding down to `$FD` — that is the stack pointer initialising near the top of page one, ready for the first push.

Recap

- A **stack** is last-in-first-out memory; it solves how a program **returns** from subroutines, including nested ones.
- The **stack pointer (S)** is a single 8-bit register that **decrements on push** and **increments on pop**, plus a load (TXS) to initialise it.
- Those three operations, over a reserved memory page, give the complete call-and-return mechanism — a huge capability from one small counting register.

Check yourself: A program calls subroutine A, which calls subroutine B. When B returns, how does the CPU know to go back into A and not somewhere else? *(B's return address was pushed last, so it is popped first — last-in, first-out — returning control to A, whose return address is still waiting underneath.)*

Next: Lesson 61 — The Flags & Decimal Mode: the status register, and a famously quirky feature.

Lesson 61 — The Flags & Decimal Mode

> **Part X · The 6502** — building blocks (chips: FLAGS, DECADJ) Before this lesson: the Stack Pointer (Lesson 60) and the ALU's flags (Lesson 47). After this: the Gate-Built 6502 (Lesson 62).

What you will learn

- The 6502's full status register, beyond the Z and C you already know.
- How flags get *set* — some from the ALU, some from the data bus, some by direct command.
- A famously quirky feature: **decimal mode**, and the dedicated hardware that makes it work.

The idea

In Lesson 47 you met two flags: **Z** (zero) and **C** (carry). The 6502 keeps a whole **status register**, called **P**, and the **FLAGS** chip is where those status bits live and update. Most of them you can already reason about; one of them is strange and wonderful, and gets its own chip.

The 6502's flags, and where each comes from:

- **N (negative)** and **Z (zero)** — set from the value on the data bus: N is just the top bit (bit 7) of a result, Z is "the result was zero" (the NOR-of-all-bits idea from Lesson 47).
- **C (carry)** and **V (overflow)** — set from the ALU (Lesson 57): C is the unsigned carry, V the signed overflow.
- **I (interrupt disable)** and **D (decimal)** — these are not computed from results; they are *modes* you set or clear directly, with instructions like `SEI` / `CLI` (interrupt) and `SED` / `CLD` (decimal). The FLAGS chip lets each be commanded on or off.

So the status register is a small cluster of one-bit latches (Lesson 26's territory), each fed from the right source — bus, ALU, or a direct set/clear command. Nothing exotic about *storing* them; the interest is in *where each value comes from*, and the FLAGS chip wires each to its proper source.

Decimal mode — the quirk worth its own chip

The **D (decimal)** flag turns on one of the 6502's most distinctive features, and the reason for the separate **DECADJ** chip.

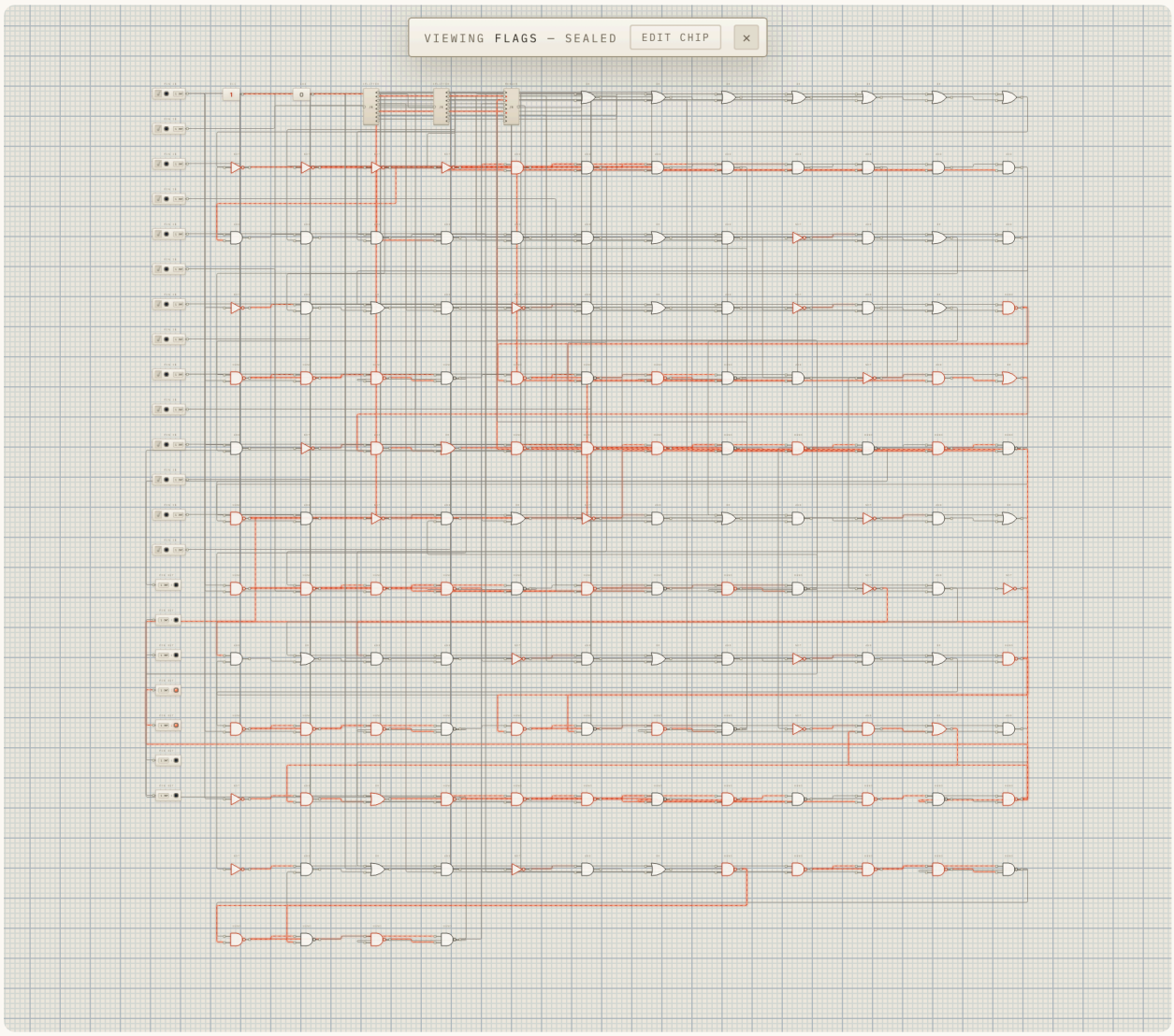
Normally a computer adds in pure binary. But humans count in decimal, and some applications — calculators, cash registers, clocks — want to work directly in decimal digits without converting back and forth. The 6502 supports this with **BCD (binary-coded decimal) arithmetic**: when the D flag is set, `ADC` (add-with-carry) and `SBC` (subtract-with-carry) treat each byte as **two decimal digits** (0–9 in each nibble) rather than a binary number, and produce a properly *decimal* result — so `0x09 + 0x01` gives `0x10` (decimal ten), not `0x0A`.

Making binary adder hardware produce *decimal* answers takes a correction step, and that is exactly what **DECADJ** does: after the normal binary add, two chained nibble-correction units (PLAs — programmable logic arrays, structured gate networks) **adjust** each nibble back into valid decimal range, carrying between digits as decimal carrying requires. It is a small, special-purpose gate network whose only job is "fix a binary sum into a decimal one."

This is a lovely example of the book's whole thesis at the level of a single feature: a quirky, real behaviour of a famous chip — one that puzzled programmers for decades — turns out to be *just gates*, a correction network you can open and inspect. No magic; a PLA that nudges nibbles.

See it in the bench

Open this: drill into the **6502 — Gate-Built** preset (Lesson 62), open the **M6502** chip, then the **FLAGS** block (and, near the ALU, the **DECADJ** block).



The FLAGS block: the 6502 status register, with each flag's source

Try it yourself

> **Try:** 1. In the FLAGS block, match each flag to its source: **N, Z** from the bus; **C, V** from the ALU; **I, D** set/cleared by command. Each is a one-bit latch — the SR/D-latch territory of Part IV. 2. Recall how flags are *used*: a branch (Lesson 59) tests one of these bits to decide whether to jump. The flags are where "did something happen?" is recorded so the program can react. 3. Open the **DECADJ** block and appreciate what it is: a correction network that turns a binary sum into a decimal one when D is set. Its whole existence is to make `9 + 1 = 10` come out as decimal `0x10`, not binary `0x0A`.

Recap

- The 6502 **status register (P)** holds **N, V, Z, C, I, D** — set from the **bus** (N, Z), the **ALU** (C, V), or **direct command** (I, D).
- Each flag is a **one-bit latch** (Part IV) wired to its proper source by the FLAGS chip.
- **Decimal mode** (the D flag) makes `ADC` / `SBC` work in **BCD**, and the **DECADJ** chip is the correction network that adjusts binary sums into decimal — a famous quirk that is, in the end, just gates.

Check yourself: With decimal mode on, what is `0x09 + 0x01`, and what makes that happen? *(`0x10` — decimal ten — because the DECADJ chip corrects the binary sum `0x0A` back into valid BCD, carrying into the next decimal digit.)*

Next: Lesson 62 — The Gate-Built 6502: every block you just opened, assembled into a real processor made of gates.

Lesson 62 — The Gate-Built 6502

> **Part X · The 6502** — library circuit (category: CPU) Before this lesson: every block of Part X (Lessons 56–61). After this: the Bouncing Ball (Lesson 63).

What you will learn

- How every block you just opened assembles into a complete, real processor — made of gates.
- How the 6502 boots like real silicon, through a reset vector.
- That the most famous chip of its era is, all the way down, the logic you have built since Lesson 09.

The idea

This is the monument. The **gate-built MOS 6502** is the same instruction set as the behavioral model from Lesson 56 — but here the CPU is **one chip built entirely from logic gates**. Every block you opened in the last five lessons is *real* here: a 16-bit program counter, the 8-bit ALU, the effective-address adder, the stack pointer, the flag latches, the A/X/Y/IR/DL registers — all sequenced by **control ROMs addressed by (opcode, T-state)**, exactly the microcode mechanism from the LB-8 (Lesson 53), now driving a real, historical ISA.

It sits on a real bench — a **64K RAM** wired over a 16-bit address bus and an 8-bit data bus, the same SBC (single-board-computer) topology as the LB-8. And it **boots like real silicon**: on power-up it does not start at address 0. It reads the **reset vector** from `$FFFC` / `$FFFD` — two bytes that hold the address where execution should begin — and jumps there, winding the stack pointer down to `$FD`. This is precisely how every real 6502 started: a Commodore 64,

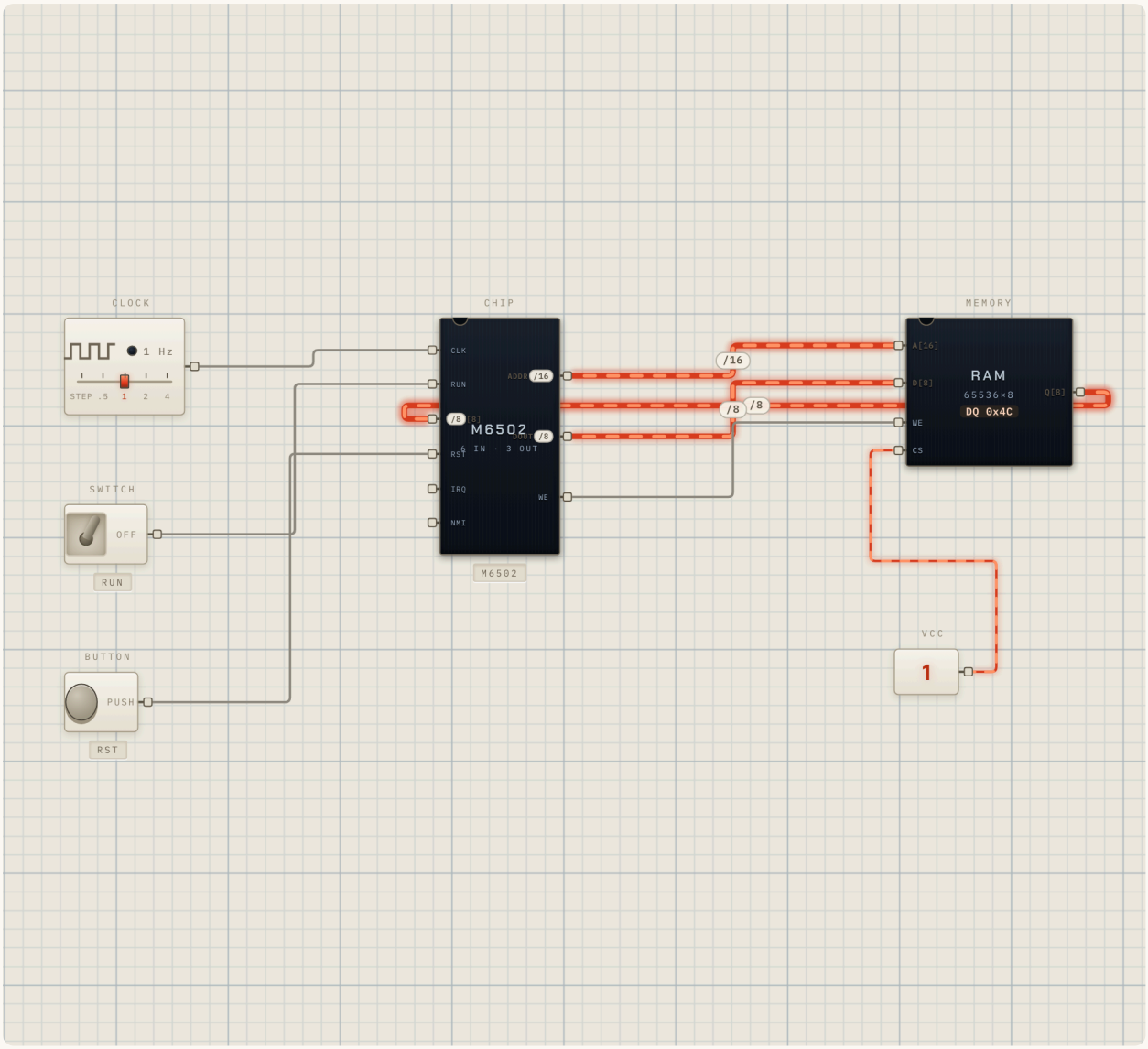
an Apple II, an NES, all began by reading those two bytes and going where they pointed. You can watch your gate-built 6502 do the same.

Step back and weigh what this is. It is not a simulation of a 6502 *wearing* a gate costume — it is a 6502 *made of gates*, its correctness proven against the original (the behavioral model it matches was verified against the Klaus Dörmann suite, the standard 6502 test). When it runs a program, real gates are switching: the adder you understand, the registers you understand, the microcode you understand. There is no level at which it stops being logic you have built.

And here is the honest engineering note the bench makes openly: clocking ~6,000 gate nodes every cycle is far too slow to animate at any real speed. So when you press **RUN**, the bench **fast-forwards a verified-equivalent emulator** (proven instruction-for-instruction identical to the gates) and mirrors the result onto the screen and the gate flip-flops — so things move at a watchable pace while the registers stay live. But the gates are not bypassed as a teaching matter: **STEP runs the real gates**, one T-state at a time, and you can drill into the chip and watch the microcode ROMs and datapath move per step. The fast path is an honesty about performance, not a hiding of the machine — the real gates are always one STEP away.

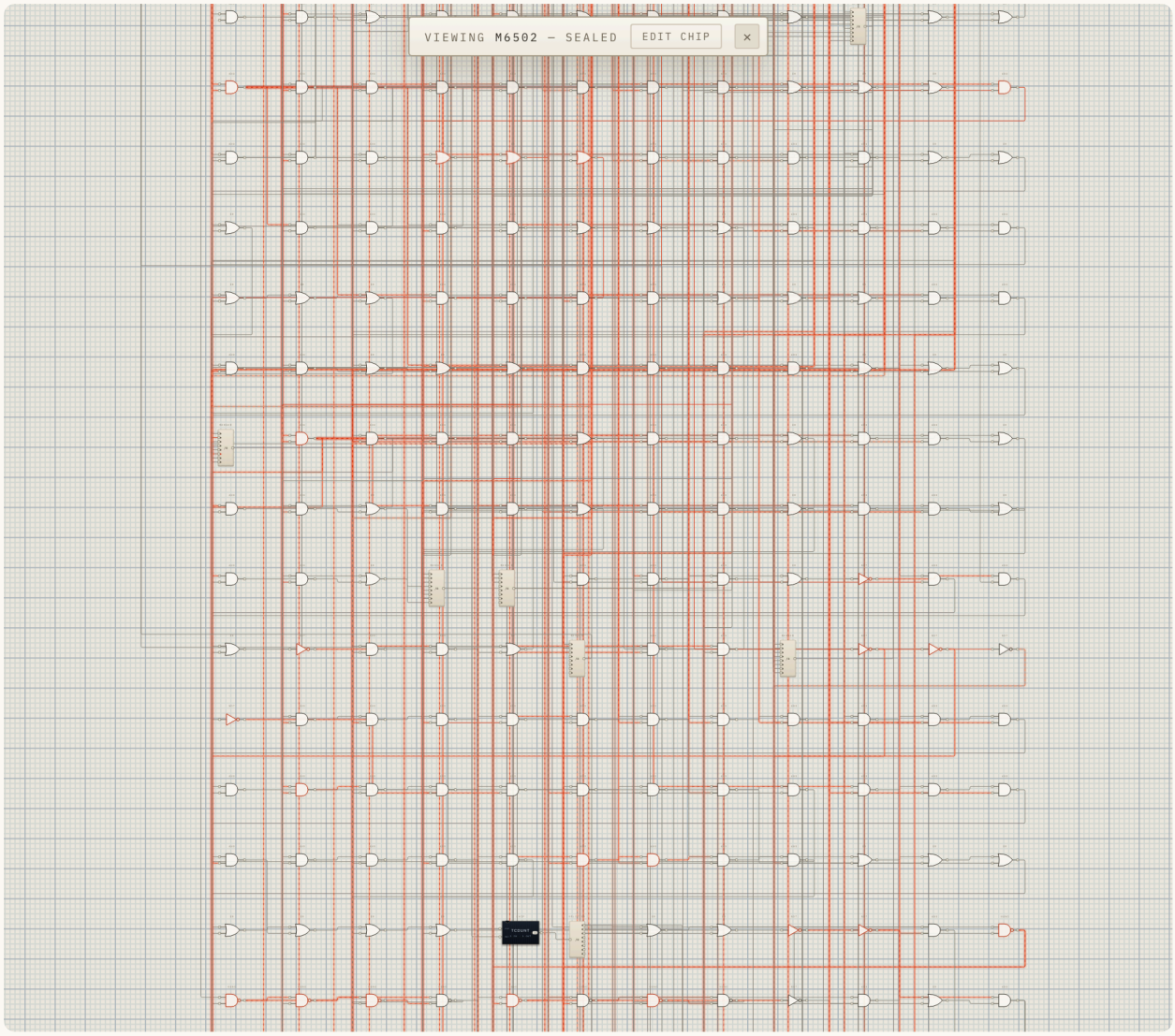
See it in the bench

> **Open this:** load **6502 — Gate-Built** from the library (category **CPU**). On power-up it resets through `$FFFC` and runs the loaded program (a diagonal draw) from its 64K RAM. The **REGISTERS** panel shows PC/A/X/Y/S/P live; the screen is in the OUTPUT RACK.



The gate-built 6502 running, registers live, drilled one level in

Then drill in: double-click the **M6502** chip and watch the microcoded datapath — the control ROMs, the T-counter, the registers and ALU — move per T-state.



Inside the gate-built 6502: the microcoded datapath on real gates

Try it yourself

> **Try:** 1. Load it and watch it **boot**: it reads the reset vector at `$FFFC` and begins. At 1 Hz, RUN draws the diagonal pixel by pixel while the REGISTERS panel shows the machine working. 2. Set the clock to **STEP** and use **STEP INSTR** — now the **real gates** execute, one instruction at a time. This is the gate-true machine, not the fast-forward. 3. **Drill into the M6502 chip** and, while stepping, watch the **control ROMs** select the next micro-operation and the datapath respond per T-state. The microcode mechanism from Lesson 53, driving a real ISA. 4. Keep going down: open the **ALU**, then its adder, then a full adder, then a single gate. Trace the chain from "a CPU that ran the Apple II" to one NAND. Not one link is magic.

Recap

- The **gate-built 6502** is the real 6502 ISA built **entirely from gates**: PC16, ALU6502, ADD16, SP8, FLAGS, registers, all sequenced by **microcode ROMs** (opcode, T-state).
- It **boots like silicon** — reading the **reset vector** at `$FFFC` and jumping there — and runs from a 64K RAM over the SBC bus.
- **RUN fast-forwards a verified-equivalent emulator** for watchable speed; **STEP runs the real gates**, always one step away, drillable to a single NAND.

Check yourself: When the gate-built 6502 powers on, why doesn't it start executing at address 0? *(It reads the reset vector at `$FFFC`/`$FFFD` — two bytes holding the start address — and jumps there, exactly as every real 6502 boots.)*

Next: Lesson 63 — The Bouncing Ball: the whole journey in one bench — a real program, on a real gate CPU, that you can watch and take apart.

Lesson 63 — The Bouncing Ball

> **Part X · The 6502** — library circuit (category: CPU) Before this lesson: the Gate-Built 6502 (Lesson 62). The final lesson of the book.

What you will learn

- How a real, complete program runs on a processor you built from gates.
- How an animated picture emerges from nothing but stores to memory.
- That you have walked the entire path from one gate to a working computer — and can prove it.

The idea

This is the north star — the lesson the whole book has been climbing toward since the first switch lit the first LED.

A pixel bounces around a screen. It moves smoothly, tracks its position and velocity, erases and redraws itself each frame, flips direction when it hits an edge. It looks like the simplest possible video game. And it is running on a **MOS 6502 built entirely from logic gates** — the same machine code that ran on real 6502 silicon, executing on a processor whose every adder, register, flag and microcode step you can open and watch.

Sit with the layers of that for a moment, because they are the whole book:

- The **ball** is plain 6502 code. It keeps its position and velocity in zero-page memory, redraws itself each frame, and paces itself with a busy-wait delay loop — the authentic way a fixed-clock machine controls its speed. (No new tricks: it is `LDA`, `STA`, branches, and the addressing modes you learned.)

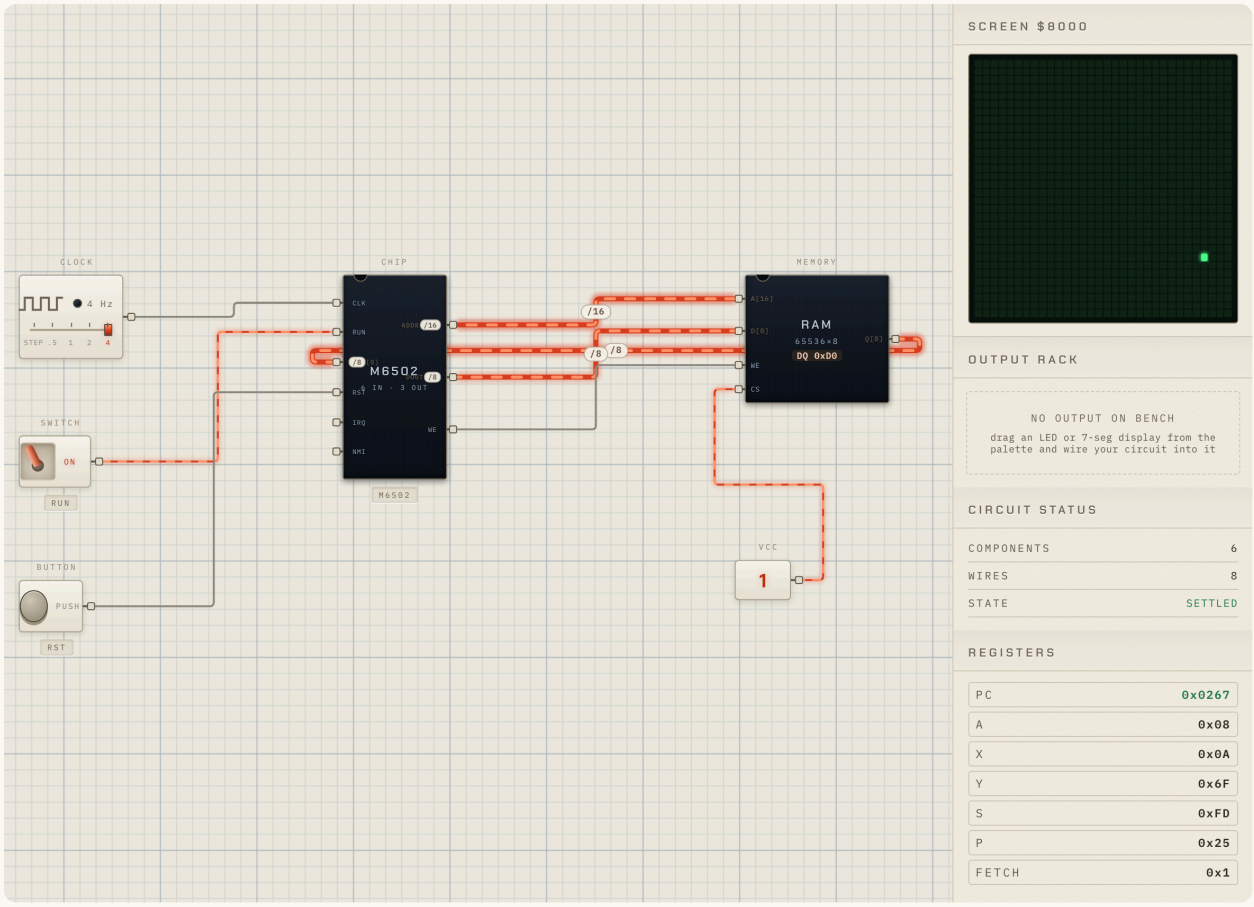
- The **picture** is a framebuffer (Lesson 55): the ball appears because the program *stores* a byte to a screen address, and erases because it stores a zero. Animation is just stores over time.
- The **processor** is gates (Lesson 62): every instruction the ball program runs is executed by the ALU, registers, PC, stack and microcode you opened block by block in this Part.
- And those blocks are **adders, latches, muxes and gates** (Parts I–VII): open any of them and you reach the half-adder of Lesson 18, the SR latch of Lesson 26, the NAND of Lesson 11.
- Which are, at the very bottom, the **single gate** of Lesson 09.

From a bouncing ball down to one AND gate, in an unbroken chain, with no magic at any level. That is the thing this whole book set out to make true and visible — and here it is, running in front of you, every layer openable.

As with the gate-built 6502, the bench is honest about speed: **RUN fast-forwards a verified-equivalent emulator** so the ball flies at a watchable frame rate (clocking thousands of gates per cycle would be far too slow), while the registers stay live. Then **pause and STEP** to hand control back to the actual gates and walk the program one instruction at a time, or **drill into the chip** to watch the microcoded datapath. The motion is fast-forwarded; the machine is real, and always one step away.

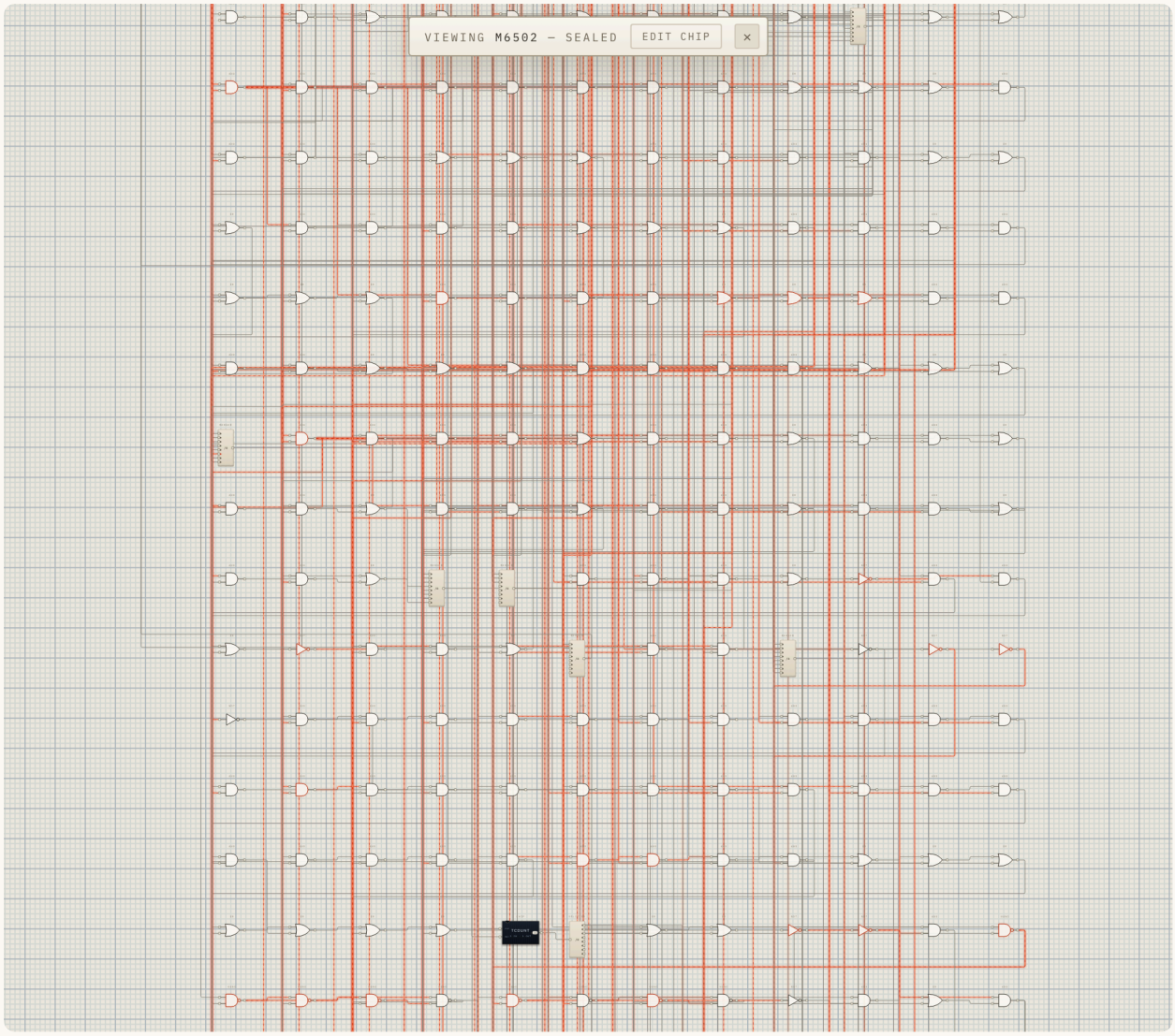
See it in the bench

> **Open this:** load **6502 — Gate-Built Ball** from the library (category **CPU**). Press **RUN** and watch the ball bounce — running on the gate-true 6502, the registers live in the panel.



The bouncing ball running on the gate-built 6502

Then take it apart: pause, STEP through the code on the real gates, and drill into the M6502 chip to watch the datapath move.



Pausing the ball and stepping the real gates inside the 6502

Try it yourself

Try: 1. Press **RUN** at 4 Hz and watch the ball bounce. A real program, on a real gate CPU. Lower the clock for slow motion. 2. Open the **ASM panel** and read the ball's code. Recognise it: `LDA` / `STA` to move and draw, branches to flip direction at the edges, a delay loop to pace it. Every instruction is one you have met. 3. **Pause and STEP INSTR** — now the actual gates execute the ball, one instruction at a time. Watch a single `STA` light a pixel via the framebuffer. 4. **Drill all the way down.** From the running ball: into the M6502 chip, into the ALU, into its adder, into a full adder, into a single gate. Arrive at one NAND — and remember Lesson 09, where you flipped two switches and watched one lamp. You have climbed the entire ladder.

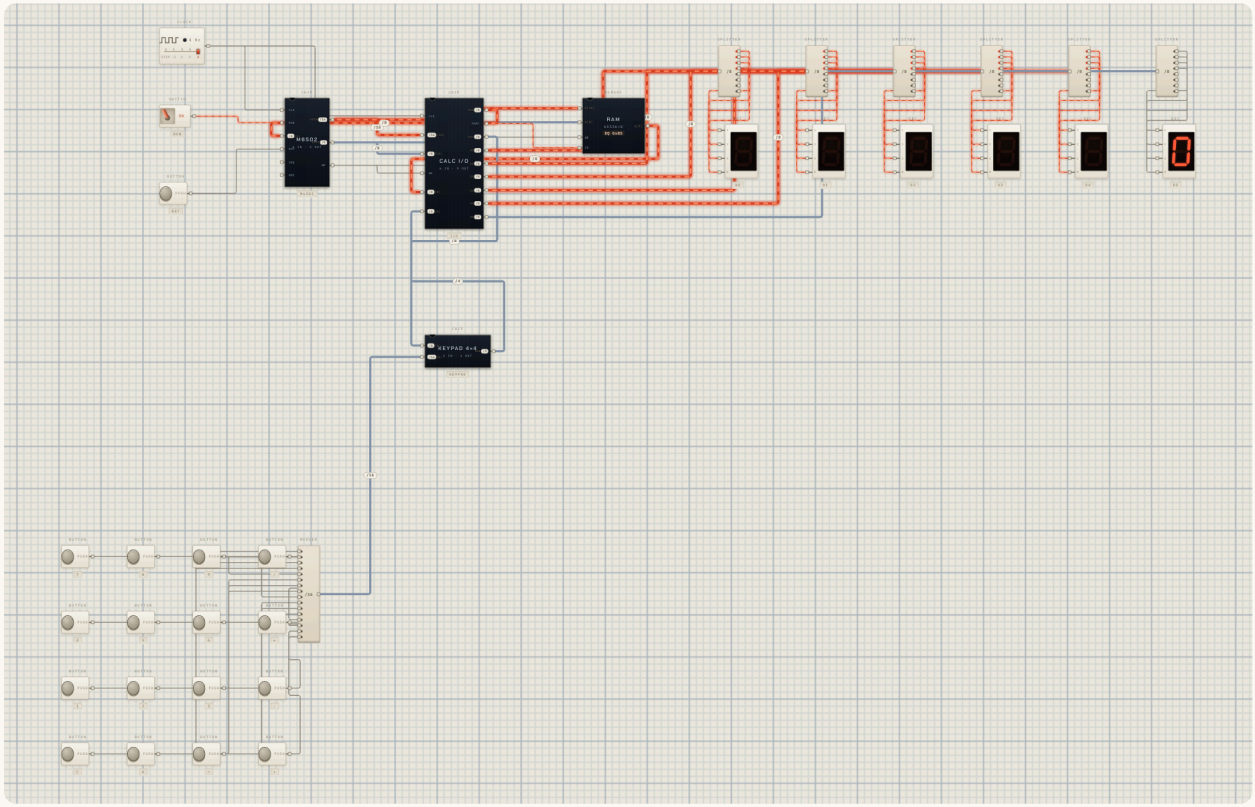
You reached the summit

You began with a single switch and a single LED. You met the gates, proved they could build anything, taught them to add and remember and count, gave them a clock and a memory, assembled them into CPUs, and arrived here: a real, historical processor — the chip behind the machines that taught a generation what a computer was — built from gates, running a program, animating a picture, and openable all the way down to the one lamp you lit in Lesson 05.

There was never a magic layer. There was only logic, arranged with care, all the way up and all the way down. Now you have seen the whole of it — and you understand, concretely and completely, how a computer works.

Where to go next

The ladder is climbed, but the bench keeps going. One board worth loading after the summit is **6502 — Pocket Calculator** (category **CPU**): the very same gate-built 6502, now running calculator firmware and wired to a **4×4 keypad** and a **6-digit display**. Where the ball only *draws*, the calculator *reads input* — proving the processor you built can do real, interactive I/O. It talks to the keypad and display through memory-mapped addresses on the `$D0xx` page (no new instructions, just `STA` / `LDA`), decoded by a gate-built I/O controller you can drill into, beside a scanned keypad matrix that is itself just `AND`s and `OR`s. Press **RUN**, tap `7 + 8 =`, and watch a number you typed appear on a display driven by gates you understand — the smallest honest thing that feels like a *device*.



The Pocket Calculator board: the gate-built 6502 with its I/O controller and RAM, the six-digit display, and the 4x4 keypad below

---

This is the final lesson. Return to the Contents to revisit any rung — or open any chip in the bench and keep exploring. Every box still opens.

Appendices

Appendix A — The Assembler: Reference & Example Programs

> **Companion to Part IX.** Use this alongside Lessons 52 (Tiny CPU II loadable) and 53 (LB-8), where the ASM panel first appears.

This appendix documents the assembler built into Logic Bench: the language it accepts, the full instruction sets of both programmable CPUs, and a set of complete, ready-to-run example programs. Every byte listing in this appendix was produced by the bench's own assembler, so the encodings are exact — type a program in, press **LOAD**, and it will assemble to the bytes shown.

A.1 — What the assembler is

The bench includes a small **two-pass assembler**. You write a program in readable text — mnemonics like `LDA` and `ADD` — press **LOAD**, and the assembler translates it into the bytes that fill the CPU's program memory. It also works in reverse: a **disassembler** turns raw bytes back into readable mnemonics, which is how the bench can show you a program that was loaded as a blob with no source.

One assembler serves **both** CPUs. It is *table-driven*: the rules of each processor — its mnemonics, opcodes, and operand encodings — live in a data table called an **ISA** (Instruction Set Architecture), and the same assembler reads whichever table applies. Tiny CPU II and the LB-8 are two different tables, not two different assemblers. (This is the same "behaviour expressed as data" idea you met with the lookup ROM in Lesson 46.)

Because there are two instruction sets, **a program is always written for one specific CPU.** A few things that are legal on the LB-8 (such as the `#` immediate marker) are not part of Tiny CPU II's language, and vice versa. Where this matters below, it is called out explicitly.

A.2 — Language reference

The general grammar — comments, labels, numbers, directives — is shared by both CPUs. What differs between the two is only the **instruction set** (Sections A.3 and A.4) and one point about operand syntax, noted under *Instructions* below.

Comments

A semicolon begins a comment that runs to the end of the line:

```
HLT         ; this is a comment
```

Labels

A **label** is a name ending in a colon. It marks a position in the program so you can refer to it by name instead of counting addresses by hand — which is what makes jumps manageable. A label may sit on its own line or in front of an instruction:

```
loop:  ADD 1        ; "loop" names the address of this instruction
       JMP loop      ; jump back to it — the assembler fills in the address
```

Label names start with a letter or underscore and may contain letters, digits, and underscores. During assembly each label is resolved to the address it marks; the resulting name-to-address map is the **symbols** table (shown after a successful assemble). Referring to a label you never define is an error, and defining the same label twice is an error.

Instructions

One instruction per line: a **mnemonic**, optionally followed by an **operand**.

```
MNEMONIC              ; no operand   — e.g. HLT, NOP, INX
MNEMONIC operand      ; with operand — e.g. ADD 1, JMP loop, LDA #5
```

Whether a mnemonic takes an operand is fixed by the instruction set; supplying one where none is allowed, or omitting a required one, is an error.

How the operand is written can also carry meaning — **but only on the LB-8.** There, the operand's *syntax* chooses the addressing mode: `#5` means "the value 5", `5` means "memory address 5", and `5,X` means "address 5 plus the X register" (see A.4). **On Tiny CPU II there is only one form** — a bare operand like `ADD 1` — and its meaning (immediate, memory address, or jump target) is fixed by the mnemonic itself (see A.3). Tiny CPU II does not use the `#` or `,X` syntax at all.

Number formats

Wherever a numeric value is expected — an operand, or a value in a data directive — it may be written in any of four forms, and you can mix them freely within a program:

Format	How to write it	Example	Value
Decimal	digits, no prefix	`10`	10
Hexadecimal	`$` or `0x` prefix	`$0A`, `0x0A`	10
Binary	`%` or `0b` prefix	`%1010`, `0b1010`	10
Character	a character in single quotes	`'A'`	65

All four rows above describe the **same value, 10** (except the character example, whose value is the character code of `A`, which is 65). The format is purely how *you* prefer to write a number; the assembler produces identical bytes regardless. For instance, on the LB-8 the four immediate loads `LDA #10`, `LDA #$0A`, `LDA #%1010`, and `LDA #0x0A` all load the value ten — they differ only in notation. (The leading `#` there is the LB-8's immediate marker from A.4, not part of the number itself.)

Directives

Directives begin with a dot and instruct the assembler rather than emitting a CPU instruction:

Directive	Meaning
`.org N`	Set the assembly address to **N**; the following code or data is placed starting there.
`.byte v[, v…]`	Emit one or more raw bytes directly into memory.
`.word v[, v…]`	Emit one or more 16-bit values, **little-endian** (low byte first).

For example, this places four raw bytes at address 64 and names that location `data`:

```
      .org $40
data:  .byte 3, 1, 4, 1     ; four bytes at $40, $41, $42, $43
```

Errors the assembler reports

If a program is malformed, **LOAD** fails and the assembler reports why, with the line number. The messages you are most likely to meet:

- `unknown mnemonic "…"` — the mnemonic is not in this CPU's instruction set (for example, using an LB-8 instruction while assembling for Tiny CPU II).
- `"…" needs an operand` / `"…" takes no operand` — the operand presence does not match the instruction.
- `"…" has no <mode> addressing mode` — the mnemonic does not support the mode your syntax implies; for example `STA #5` on the LB-8 (you cannot store *into* an immediate value).
- `operand N out of range 0..M` — the value does not fit the field: 0–15 for a Tiny CPU II operand, 0–255 for an LB-8 operand byte.
- `undefined symbol "…"` — a label was referenced but never defined.
- `duplicate label "…"` — the same label was defined twice.

A.3 — Tiny CPU II instruction set

Tiny CPU II (Lessons 51–52) uses a **packed** encoding: every instruction is a **single byte**, with a 3-bit opcode in the high bits and a 4-bit operand in the low nibble, combined as `opcode << 4 | operand`. The operand is therefore always a value from **0 to 15**. The machine has one accumulator (**ACC**).

A point worth understanding clearly: although the operand field is 4 bits wide (0–15), the **data memory (RAM) has only 4 words**, addressed by the operand's low 2 bits. So a memory operand of 0, 1, 2, or 3 selects RAM word 0–3 directly; an operand of 4–15 wraps (operand **mod 4**) onto those same four words. Three of the four words (0, 1, 2) have an on-screen display; word 3 has none. The examples below stay within words 0–2 so every value is visible.

Mnemonic	Opcode (high nibble)	Operand means	Effect
`HLT`	`0`	—	Halt the machine.
`LDI n`	`1`	immediate 0–15	ACC = **n** (the literal value).
`ADD a`	`2`	RAM word (low 2 bits)	ACC = ACC + mem[a].
`SUB a`	`3`	RAM word (low 2 bits)	ACC = ACC − mem[a].
`JMP a`	`4`	program address 0–15	Jump to address **a**.
`JZ a`	`5`	program address 0–15	Jump to **a if the zero flag is set** (see below).
`LDA a`	`6`	RAM word (low 2 bits)	ACC = mem[a].
`STA a`	`7`	RAM word (low 2 bits)	mem[a] = ACC.

Reading the encoding. `LDI 3` is opcode `1`, operand `3` → byte `0x13`. `STA 0` is opcode `7`, operand `0` → `0x70`. `HLT` is `0x00`.

The zero flag and `JZ`. The machine continuously computes a **zero flag** as the NOR of the four accumulator bits — it is 1 exactly when ACC is currently 0. `JZ`

jumps when that flag is set, i.e. when the accumulator holds zero. (In practice you reach a zero ACC by doing arithmetic, such as a `SUB` that cancels out, then testing it with `JZ`.)

`HLT` is the all-zero byte (`0x00`), and that is deliberate. Because an empty, unprogrammed memory word is all zeros, it decodes as `HLT`. So if the program ever jumps into memory you never wrote, the machine simply halts rather than running garbage. Execution begins at **address 0** after reset.

A.4 — LB-8 instruction set

The LB-8 (Lesson 53) is a fuller 8-bit CPU. It uses a **byte-opcode** encoding: the opcode occupies the whole first byte, and an instruction is **one or two bytes** long (a one-byte operand follows when the instruction needs one). It has two registers — accumulator **A** and index register **X** — and two flags, **Z** (zero) and **C** (carry).

Its expressive power comes from **three addressing modes**, which the assembler selects from the operand's syntax:

You write	Mode	Meaning
`#v`	**immediate**	the operand *is* the value `v`
`v`	**absolute**	use the value stored in memory at address `v`
`v,X`	**absolute-indexed**	use the value at address `v + X` (X is the index register)

The same mnemonic may appear in several modes; each mode is a **distinct opcode**, chosen automatically from how you write the operand. (This is the syntax-selects-mode behaviour mentioned in A.2 — and it applies to the LB-8 only.)

Implied instructions (1 byte, no operand)

Mnemonic	Opcode	Effect
`NOP`	`$00`	Do nothing.
`HLT`	`$01`	Halt.
`INX`	`$02`	X = X + 1.

Operand instructions (2 bytes: opcode + one operand byte)

Written as	Opcode	Effect
`LDA #v`	`$10`	A = v (immediate)
`LDA v`	`$11`	A = mem[v] (absolute)
`LDA v,X`	`$12`	A = mem[v + X] (indexed)
`LDX #v`	`$18`	X = v (immediate)
`LDX v`	`$19`	X = mem[v] (absolute)
`STA v`	`$20`	mem[v] = A (absolute)
`STA v,X`	`$21`	mem[v + X] = A (indexed)
`ADD #v`	`$30`	A = A + v (immediate)
`ADD v`	`$31`	A = A + mem[v] (absolute)
`ADD v,X`	`$32`	A = A + mem[v + X] (indexed)
`SUB #v`	`$38`	A = A − v (immediate)
`SUB v`	`$39`	A = A − mem[v] (absolute)
`SUB v,X`	`$3A`	A = A − mem[v + X] (indexed)
`AND #v`	`$40`	A = A AND v (immediate)
`AND v`	`$41`	A = A AND mem[v] (absolute)
`JMP v`	`$50`	Jump to address v.
`JZ v`	`$51`	Jump to v **if the Z (zero) flag is set.**
`JC v`	`$52`	Jump to v **if the C (carry) flag is set.**

Flags. The arithmetic and logic instructions (`ADD`, `SUB`, `AND`) set the result-describing flags: **Z** is 1 when the result is zero (the NOR of all eight result bits), and **C** is the carry out of the top bit of an add (or the borrow indicator of a

subtract). You make a decision by performing an operation that sets a flag and then branching on it with `JZ` or `JC`.

Operand values are full bytes, **0–255**. Execution begins at **address 0** after reset, and the program lives in a 256-byte board RAM (Lesson 53).

> **What the LB-8 deliberately lacks.** There is no compare instruction and no "decrement-and-branch." To act on a condition you set a flag (for example, `SUB` produces a zero result that sets `Z`, or an add that overflows a byte sets `C`) and then use `JZ`/`JC`. A counted loop is often clearest if you simply **unroll** it — write the body out once per iteration — since there is no loop-counter-compare instruction. The demo program in A.5.4 instead loops by watching for a carry, which is the idiomatic LB-8 way.

A.5 — Example programs

Every listing below was produced by the bench's own assembler, so the bytes are exact. Type a program into the ASM panel, press **LOAD**, then run with the clock slowed (or single-stepped) so you can follow each instruction. Addresses and bytes are shown in hex.

> **A note on "Watch" descriptions.** The byte encodings here are verified by the assembler. The accompanying "what happens when it runs" notes describe the intended behaviour of each program; watch the registers and memory in the bench to confirm them for yourself — that is exactly the read-then-verify habit the whole workbook is built on.

A.5.1 — Tiny CPU II: add two constants

Computes 3 + 4 = 7. Because `ADD` takes a *memory* operand, the first value is stored in RAM word 0 so the second can add it.

```
LDI 3        ; ACC = 3
STA 0        ; word 0 = 3   (so ADD can reference it)
LDI 4        ; ACC = 4
ADD 0        ; ACC = 4 + word 0 = 7
STA 1        ; word 1 = 7
HLT
```

Assembles to: `13 70 14 20 71 00` — 6 bytes at addresses 0–5.

Watch: ACC goes 3 → (stored) → 4 → 7 on the `ADD`. After `HLT`, the M1 display (word 1) shows 7.

A.5.2 — Tiny CPU II: countdown loop

Counts 5 down to 0 with a labelled loop and `JZ`. The decrement amount (1) is held in RAM word 0.

```
       LDI 1        ; ACC = 1
       STA 0        ; word 0 = 1   (the amount to subtract each pass)
       LDI 5        ; ACC = 5      (start value)
loop:  SUB 0        ; ACC = ACC - 1
       JZ done      ; if ACC reached 0, exit
       JMP loop     ; else go round again
done:  HLT
```

Assembles to: `11 70 15 30 56 43 00` — 7 bytes at addresses 0–6. **Symbols:** `loop` = 3, `done` = 6.

Watch: ACC steps 5 → 4 → 3 → 2 → 1 → 0; when it hits 0 the zero flag sets, `JZ` takes the branch to `done` (address 6), and the machine halts. Notice the assembler turned `JZ done` into the byte `56` (opcode 5, operand 6) and `JMP loop` into `43` (opcode 4, operand 3) — it resolved the labels to addresses for you.

A.5.3 — Tiny CPU II: the built-in Fibonacci program

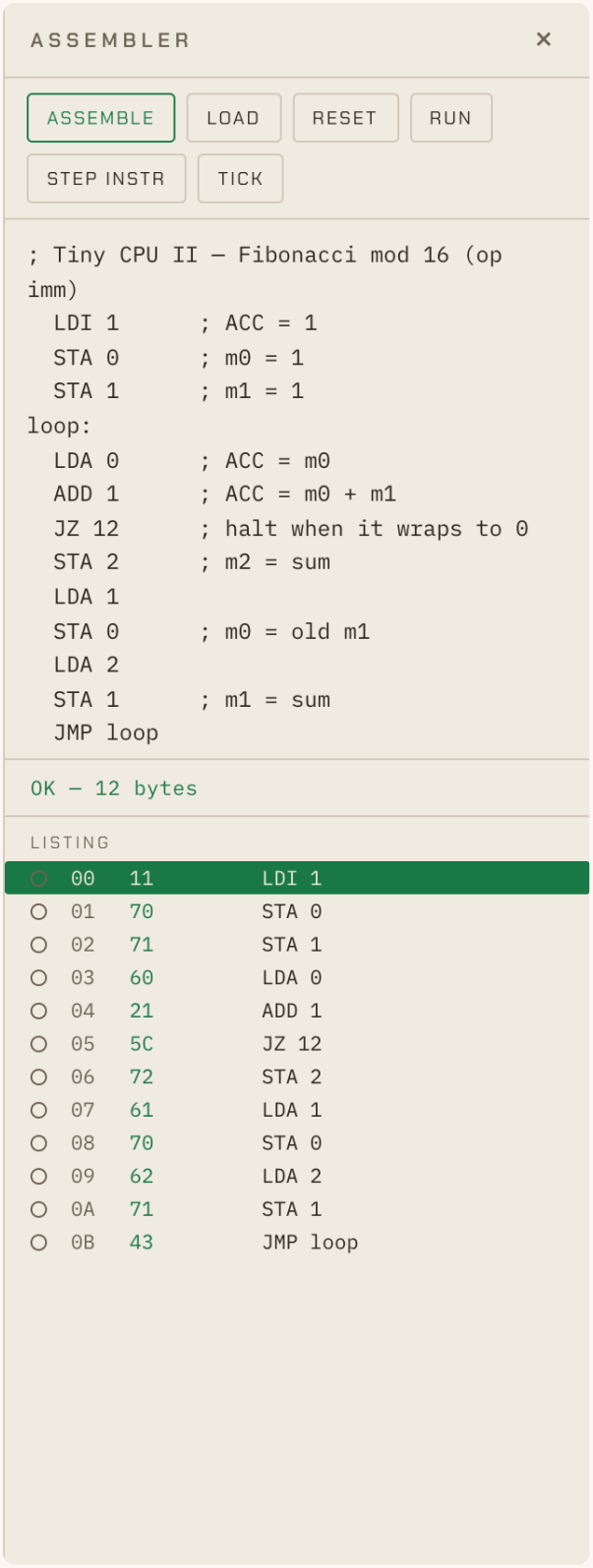
This is the program that ships loaded in the Tiny CPU II (Lessons 51–52) — Fibonacci mod 16, kept in RAM words 0, 1, and 2. It is reproduced here as a worked reading exercise; it is already in the ASM panel when you open the preset.

```
       LDI 1        ; seed
       STA 0        ; word0 = 1
       STA 1        ; word1 = 1
loop:  LDA 0        ; ACC = word0
       ADD 1        ; ACC = word0 + word1
       JZ 12        ; when the sum wraps to 0 (mod 16), halt
       STA 2        ; word2 = sum
       LDA 1
       STA 0        ; word0 = old word1
       LDA 2
       STA 1        ; word1 = sum
       JMP loop
```

Assembles to: `11 70 71 60 21 5c 72 61 70 62 71 43` — 12 bytes at addresses 0–11. **Symbols:** `loop` = 3.

Watch: the M0/M1/M2 displays churn through the Fibonacci pairs 1,1 → 1,2 → 2,3 → 3,5 → 5,8 → … Because the values are kept mod 16, a sum eventually wraps to 0; `JZ 12` then jumps to address 12, which is empty (`HLT`), and the clock parks. The `JZ 12` target is written as a bare number here because address 12 has no label — both styles work.

Here is this program in the ASM panel, assembled — note the listing on the right showing each instruction's address and bytes, matching the encoding above:



The Tiny CPU II ASM panel with the Fibonacci program assembled

A.5.4 — LB-8: immediate add

The LB-8 version of the first example. Immediate mode (`#`) means the constants are in the instructions themselves, so no memory stashing is

needed.

```
LDA #5       ; A = 5
ADD #3       ; A = 8
STA $40      ; mem[$40] = 8
HLT
```

Assembles to: `10 05 30 03 20 40 01` — 7 bytes at addresses 0–6.

Watch: A becomes 5, then 8; `STA` writes 8 to address $40 (visible in the memory inspector). Each two-byte instruction advances the PC by 2 — the PC probe steps 0 → 2 → 4 → 6.

A.5.5 — LB-8: the built-in Fibonacci demo

This is the program that ships loaded in the LB-8 (Lesson 53). It computes Fibonacci in memory and stops when a sum overflows a byte — detected with the **carry flag** and `JC`, which is the idiomatic way to end a loop on this CPU (there is no compare instruction). It uses two labels and absolute addressing throughout.

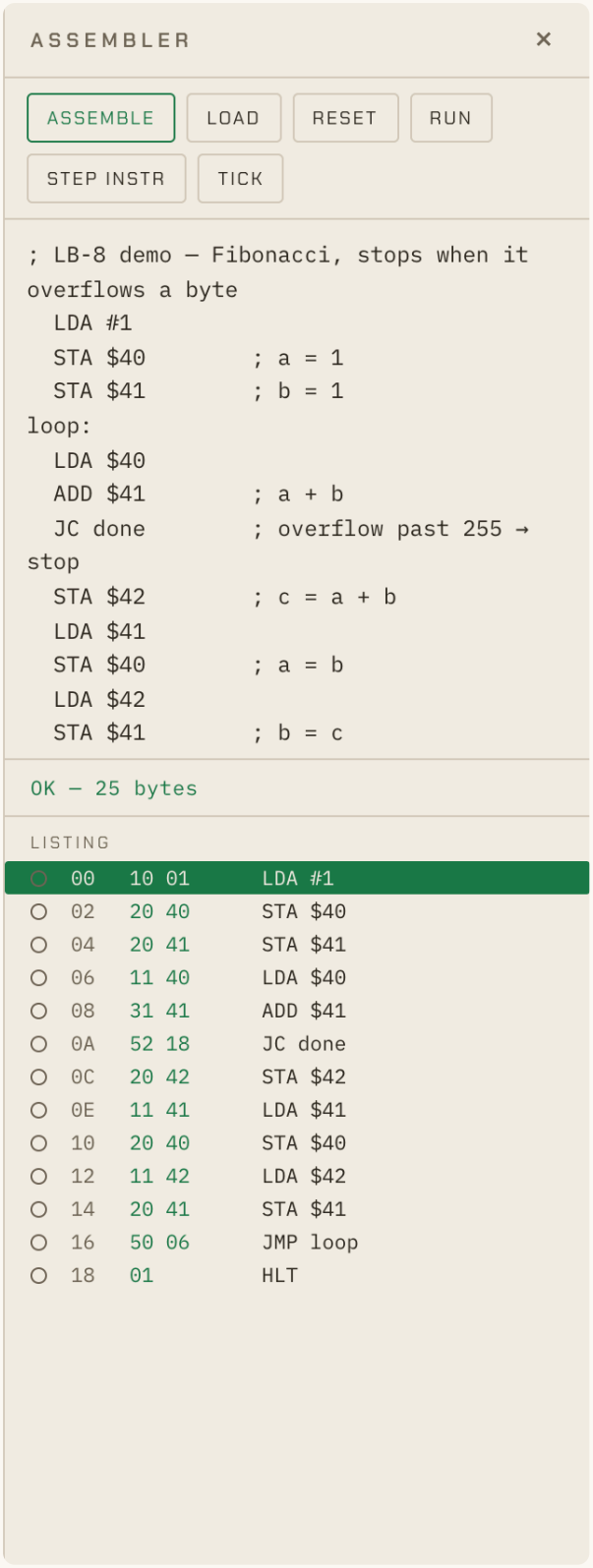
```
      LDA #1
      STA $40        ; a = 1
      STA $41        ; b = 1
loop: LDA $40
      ADD $41        ; a + b
      JC done        ; carry set → the sum passed 255, so stop
      STA $42        ; c = a + b
      LDA $41
      STA $40        ; a = b
      LDA $42
      STA $41        ; b = c
      JMP loop
done: HLT
```

Assembles to: `10 01 20 40 20 41 11 40 31 41 52 18 20 42 11 41 20 40 11 42 20 41 50 06 01` — 25 bytes at addresses 0–24. **Symbols:** `loop` = 6, `done` = 24.

Watch: A climbs through the Fibonacci numbers — 1, 2, 3, 5, 8, 13, 21, … — in the REGISTERS panel, with `$40` / `$41` holding the running pair. When an `ADD` finally pushes the result past 255 the carry flag sets, `JC done` branches to address 24 (note the assembler encoded `JC done` as `52 18`, where `$18` = 24),

and `HLT` stops the machine. The `JMP loop` became `50 06` — a jump to address 6, where the label sits.

This is the program in the LB-8 ASM panel. The listing confirms the bytes above exactly — `00: 10 01 LDA #1`, `0A: 52 18 JC done`, `16: 50 06 JMP loop`, `18: 01 HLT` — the assembler and this appendix agree, because both come from the same source:



The LB-8 ASM panel with the Fibonacci demo assembled to bytes

A.6 — Staying faithful to the machine

Two habits will keep your programs honest:

Write for the right CPU. Tiny CPU II and the LB-8 have different instruction sets and slightly different languages — most visibly, the `#` immediate marker and `,X` indexing exist only on the LB-8, and Tiny CPU II's operands are limited to 0–15. If the assembler reports `unknown mnemonic` or `has no … addressing mode`, the usual cause is writing one CPU's style for the other.

Let the assembler check you. The byte listings in this appendix came straight from the assembler, not from hand calculation, and you should trust the bench the same way: if a program loads, its encoding is correct by construction; if it does not, the error message and line number tell you what to fix. The common ones — a mode a mnemonic does not support (e.g. `STA #5`), an operand past its range, or a label you forgot to define — are all listed in A.2.

Everything documented here is what the bench actually implements today. As the project grows toward a gate-built 6502 (the path sketched in Lesson 53), new instruction tables will be added — but they will be new *data* for the same assembler, documented the same way.

Return to the Contents, or back to Lesson 53 — The LB-8.