THE LOGIC BENCH

User Guide

A complete guide to the bench — components, chips, buses, memory devices, programs, and the full 48-circuit library.

Logic Bench · every box opens, nothing inside is fake

Logic Bench is a web-based graphical logic simulator. You drag gates, switches, clocks and indicators onto a grid canvas, wire them together, and watch signals propagate in real time: wires glow when they carry HIGH, LEDs light, 7-segment displays count. The simulation re-evaluates instantly on every change, genuinely stores state in feedback circuits (latches and flip-flops behave like real memory), lets you pack any selection into a reusable **chip** you can drill into live (see *Chips: boxing circuits*), bundle parallel signals into **buses** (see *Buses: one wire, many signals*), drop in behavioral **memory devices** you inspect and edit as bytes (see *Memory devices*), write and **assemble** programs and **load** them into a board CPU you can run, single-step and breakpoint (see *Programs*), and ships with a 48-circuit library that scales from a half adder all the way up to a complete, programmable 8-bit CPU and a **MOS 6502 built entirely from gates** — booting through its reset vector, running real machine code, and openable down to a single NAND.

---

1. Quick start

1. **Place a component.** Drag AND from the GATES section of the left palette onto the canvas (or just click/tap the palette item — it lands at the center of the view). Components snap to the 16 px grid.
2. **Add inputs and an output.** Place two TOGGLE SWITCHes and one LED the same way.
3. **Wire it up.** Drag from a switch's **output pin** (right side) to one of the AND gate's **input pins** (left side), then from the AND's output to the LED's input. Wires always run *out* → *in*.
4. **Toggle.** Click a switch lever. The wire from it glows hot; with both switches ON, the AND output wire glows and the LED lights — both on the canvas and as a channel row in the OUTPUT RACK on the right.
5. **Load something bigger.** Scroll the left rail down to LIBRARY and click any entry, e.g. *Half Adder* under ARITHMETIC. It replaces the bench (you are asked to confirm if the bench is not empty) and the view zooms to fit. Hover an entry (or tap its **i** button on touch devices) for an explanation card with a "Try:" hint.

---

2. The workbench

The screen has three areas plus a toolbar:

- **Component palette (left rail)** — GATES, SOURCES, CONSTANTS, OUTPUT, BUS and MEMORY sections (the BUS section holds the SPLITTER and MERGER adapters — see *Buses: one wire, many signals*; the MEMORY section holds RAM, ROM and an 8×8 LED-matrix framebuffer device — see *Memory devices*), the CHIPS shelf (your document's chip definitions — see *Chips: boxing circuits*), and the LIBRARY (built-in presets grouped by category, plus your own saved circuits). Items are draggable; clicking/tapping places them at the viewport center.
- **Canvas (center)** — the bench itself. Pan, zoom (0.25×–2.5×), fine and coarse background grids, zoom controls bottom-left. Connections are drawn as orthogonal (smooth-step) wires: dim when LOW, glowing with an animated current-flow dash when HIGH.
- **Output rack (right rail)** — mirrors every LED as a channel row (`CH 01`, `CH 02`, … or the component's label) with a lamp and HIGH/LOW state, and every 7-segment display as a row showing its hex/decimal value and digit. Below that, a CIRCUIT STATUS block counts components and wires and reports SETTLED/UNSETTLED, and an oscillation warning panel appears when needed.

Toolbar. The header shows the LOGIC BENCH brand, a **STABLE / OSCILLATING** status chip, and a **BENCH:** chip naming the circuit currently on the bench (UNTITLED if none). Actions:

Button	Effect
☾ / ☀	Toggle dark/light theme. Dark is the default; the choice is persisted and applied before first paint.
Export JSON	Downloads the bench as `logic-bench-YYYY-MM-DD.json`.
Import JSON	Loads a circuit file (validated; invalid files are rejected with an alert).
Clear	Empties the bench after confirmation.

Responsive and touch behavior. At ≤1100 px width the side rails become overlay drawers, opened by **≡ PARTS** and **RACK** buttons in the toolbar (the RACK button shows a badge while any output is active or the circuit oscillates). At ≤700 px the Export/Import/Clear actions collapse into a menu. On touch devices: tap a palette item to place it, **long-press** (≈0.5 s) a component or wire for the context menu, and use the **i** affordance on library entries to read the explanation cards that desktop users see on hover.

---

3. Components reference

Gates

Six two-state logic gates, drawn with standard distinctive-shape symbols that light when their output is HIGH. All have one output; NOT has one input, the rest have two.

Gate	Output is HIGH when…
AND	both inputs are HIGH
OR	at least one input is HIGH
NOT	the input is LOW
XOR	exactly one input is HIGH
NAND	NOT both inputs are HIGH
NOR	neither input is HIGH

Sources

- **Toggle Switch** — latching 0/1 source. Click the lever to flip between OFF and ON; the state readout next to it shows which. Switch positions survive autosave and reload.
- **Push Button** — momentary source. Outputs HIGH only while pressed (press-and-hold works with mouse and touch); releasing, or sliding off the button, returns it to LOW. Always restored OFF on load.
- **Clock** — free-running square-wave source with a five-detent fader: **STEP, 0.5, 1, 2, 4 Hz** (1 Hz = one full cycle per second, i.e. a flip every 500 ms). A wave

glyph and lamp show the current output. Slide the fader to **STEP** and the clock stops free-running and turns into a **STEP pad**: press for a clean rising edge, release for a clean falling edge — one manual clock pulse per press, perfect for single-stepping sequential circuits. Clocks restart from LOW (phase reset) when a circuit is loaded.

Constants

- **HIGH (VCC)** — always outputs 1.
- **LOW (GND)** — always outputs 0. Useful for tying inputs down explicitly (though unconnected inputs already read LOW).

Outputs

- **LED** — single-input indicator; the lens lights when its input is HIGH. Each LED also appears as a row in the output rack, auto-numbered `CH 01`, `CH 02`, … unless you give it a label.
- **7-Seg Display (HEX)** — four inputs with binary weights **1, 2, 4, 8** (printed beside the pins, `in-0` = least significant bit). The display sums the high inputs and shows the value as one hex digit 0–F; its rack row additionally shows `0xN · decimal`. Note it renders lowercase **b** and **d** for 11 and 13, so they can't be confused with 8 and 0.
- **Binary Clock (BIN CLOCK)** — a self-contained wall instrument with no ports: six bottom-aligned BCD lamp columns spell HH:MM:SS of real wall time (bottom lamp = 1, then 2, 4, 8 upward; read each column bottom-up and add the lit lamps to get the digit). A **timezone select** underneath switches between a dozen curated IANA zones (UTC, Madrid, Berlin, London, New York, Chicago, Los Angeles, São Paulo, Tokyo, Shanghai, Kolkata, Sydney — default Europe/Madrid); an imported circuit's custom zone is kept as an extra option. It displays real time and takes no part in the simulation.

---

4. Working with circuits

Moving, rotating, deleting, renaming

- **Move** — drag a component; it snaps to the grid.
- **Context menu** — right-click (or long-press on touch) a component for **Rename…**, **Duplicate**, **Rotate 90°**, **Delete**; right-click a wire for **Delete wire**. Selected nodes/wires can also be removed with Backspace/Delete. Duplicate places a fresh copy one grid step down-right — duplicated chips share their definition but simulate independently.
- **Rotate** — each rotation turns the body and its pins 90° clockwise; wires follow.
- **Labels** — *Rename…* opens an inline tag editor below the component (Enter/blur commits, Escape cancels, empty clears, max 24 characters). Existing labels can be edited by **double-clicking the tag**. Labels replace the auto channel names in the output rack and are saved with the circuit.

Wiring rules

- Wires run from a component's single **out** handle to a target's **in** handle. Any output can fan out to many inputs.
- **Fan-in is OR-ed:** if several wires feed the same input port, the port reads HIGH when *any* of them is HIGH.
- **Unconnected inputs read LOW.** A two-input gate with one wire behaves as if the other input were 0 — e.g. a NAND with both inputs tied to the same signal acts as a NOT, but a NAND with one input left open is stuck HIGH.

The simulation model in plain words

The engine is pure and synchronous: the whole circuit is re-evaluated on every change (a switch flip, a clock tick, a wire added) by sweeping over the components repeatedly until a full sweep changes nothing — settling is instantaneous from your point of view, with no gate propagation delay.

- **Feedback circuits store state (warm start).** Each evaluation starts from the previous result rather than from all-zero. That is what makes an SR latch, D latch or flip-flop genuinely *remember*: when its inputs go idle, the cross-coupled pair

holds whichever stable state it was in, across any number of interactive changes.
- **Oscillation detection.** If the circuit has not settled after a capped number of sweeps (2 × component count + 8), the engine flags it as oscillating: the toolbar chip flips to **OSCILLATING**, the rack shows an ⚠ **OSCILLATION DETECTED** warning listing the unstable components, and the unstable nodes themselves are highlighted on the canvas. The trigger is any feedback loop with no stable state — the classic example being an odd number of inverters in a ring (see the *Ring Oscillator* preset). Logic Bench does not animate such loops; it tells you they never settle.

Persistence

- **Autosave** — the bench (components, wires, positions, rotations, labels, switch states, clock speeds, binary-clock timezones) is saved to browser localStorage automatically (debounced) and restored on the next visit. Buttons and clocks always come back OFF; switches keep their position.
- **Export / Import JSON** — the same validated format, as a file. Use it for backup or sharing; importing sets the bench name to the file name.
- **Save to library** — type a name into **SAVE CURRENT AS...** at the top of the LIBRARY section and press SAVE (or Enter). Your circuit appears in a SAVED group, sorted by name, loadable like a built-in; saving an existing name asks before overwriting, and the × button deletes an entry (with confirmation). Saved circuits live in localStorage on this browser.
- **ON BENCH indicator** — the library entry whose circuit is currently on the bench (built-in or saved) is marked **ON BENCH**, matching the toolbar BENCH chip. It is display-only: edits to the bench do not modify the library entry until you save again.

---

5. Chips: boxing circuits

Any part of a circuit can be boxed into a **chip**: a reusable sub-circuit that appears on the bench as a single DIP-package component with its own pins. A chip is one **definition** (the inner circuit) plus any number of placed **instances** that all share it. Before every evaluation the simulator flattens chips away, so a bench built from chips

behaves exactly like the same bench built from loose gates — including feedback state, which is kept per instance.

Packing and unpacking

- **Pack** — multi-select components on the bench, then right-click → **Pack n into chip...** (or use the floating **PACK** button that appears with a multi-selection). Every wire crossing the selection boundary becomes a pin — inbound wires become inputs, outbound ones outputs. The dialog asks for a name (unique per document, shown as the silkscreen), an optional description, and lets you rename and reorder the pins. On confirm the selection is replaced by one instance, wired identically, and the definition appears on the CHIPS shelf.
- **Unpack** — instance context menu → **Unpack** replaces the instance with a fresh copy of the definition's gates, wired identically. The definition stays on the shelf.
- **Undo** — pack, unpack and SAVE CHIP are each a single undo step: press **Ctrl/Cmd+Z** or the toast's **UNDO** button (pressing again redoes).

The CHIPS shelf

A CHIPS section sits in the left rail between OUTPUT and LIBRARY. Each row shows the chip's name and its pin/gate counts; drag or tap a row to place an instance, ✎ opens the definition editor, **×** deletes the definition, and **+ NEW CHIP...** creates an empty definition (pre-placed with one example pin per direction) and opens it for editing. Hover a row (or tap its **i**) for the description and full pin list.

Looking inside (sealed views)

Double-click an instance (select + **Enter**, context menu → **Look inside**, or double-tap on touch) to drill into it. The canvas zooms into the definition's circuit, rendered **live with this instance's real signals** — the PIN markers at the edges carry the boundary values, wires glow, nested chips can be drilled into further. A breadcrumb bar (`BENCH ▸ U2 ▸ …`) tracks where you are; click any crumb, press **Esc**, or use the banner's ✕ to go back up (each level's viewport is restored).

Inside views are **sealed**: you can pan, zoom, hover and select freely, but nothing can be moved, wired, deleted or toggled — in-view switches render inert, and any mutation attempt nudges the banner (`VIEWING U2 · FULL ADDER — SEALED`). Looking is always zero-risk.

Editing a definition

Press **EDIT CHIP** in the sealed banner (or **Edit chip…** in an instance's menu, or the shelf's ✎). The definition opens as a normal, fully editable bench with the banner `EDITING DEFINITION — USED BY n INSTANCES`. Two things are special in the editor:

- A **PINS** section appears at the top of the palette: **PIN IN** and **PIN OUT** markers define the chip's interface. A pin's name is the marker's label; pin order is the markers' top-to-bottom position.
- The draft simulates **standalone**: every PIN IN grows a switch-style **test toggle**, so the editor doubles as the chip's test bench. The toggles exist only here — placed instances are driven purely by their wires.

SAVE CHIP (or **Ctrl/Cmd+S**) applies the edit to every instance in one undoable step. Wires connected to deleted pins are dropped (you are warned with the affected wire/instance count beforehand); reordered pins keep their wires. Leaving the editor with unsaved changes prompts first.

Delete protection

A definition can only be deleted while it is unused: if instances exist on the bench or inside other chips, deletion is blocked and the toast offers **SHOW ME** to select the bench instances. Deletion is also blocked while the definition's view or editor is open, and while the open editor draft places it.

Oscillation inside chips

If a feedback loop inside a chip never settles, the instance shows the pulsing red halo on the bench, and the CIRCUIT STATUS panel lists the path into the chip (e.g. `U2 / not1`; hover for the full path). Drill in to see the actual unstable gates highlighted.

Persistence

Autosave keeps every definition, used or not. **Export JSON** and **library saves** embed exactly the definitions the bench uses, so files are self-contained; importing or loading replaces bench and shelf wholesale. Old v1 files (from before chips existed) load forever.

6. Buses: one wire, many signals

Eight signals drawn as eight separate wires is visual noise. A **bus** is the fix: one thick strand on the canvas that carries *n* parallel 1-bit lines underneath. It is the same honest abstraction as a chip — nothing is hidden, only bundled. Before every evaluation the simulator expands each bus back into its individual wires, so the engine stays strictly 1-bit and a bus behaves exactly like the loose wires it stands in for.

A bus wire draws thicker than a normal wire and carries a small `/n` tick showing its width. It glows in proportion to how many of its bits are HIGH, and **hovering it shows the live value in hex** (e.g. `0x6B`). Bit 0 is the least-significant bit (LSB).

Splitters and mergers

Buses are bridged to individual wires by two adapters in the **BUS** section of the palette:

- **SPLITTER** — one bus in, *n* single-bit outputs (`out-0` = LSB at the top). Use it to drive individual LEDs or gates from a bus.
- **MERGER** — *n* single-bit inputs (`in-0` = LSB), one bus out. Use it to gather loose signals (switches, gate outputs) into a bus.

A splitter is where the abstraction is made visible: it literally shows that one fat strand *is* these eight wires, each lighting with its own bit. Each adapter's width is set when you place it (default 8) and shown on its face as `/n`.

Bus pins on chips

A chip pin can itself be a bus. In the definition editor, a PIN marker has a **width** control: set it above 1 and the pin becomes a bus pin, drawn on instances as a fat pin labelled like `DATA[8]` with a `/8` tick. Inside the definition you typically split the incoming bus into bits, do the work, and merge the result back — exactly what you see when you drill into the **8-Bit Adder (Bus)** preset.

Connecting buses

Wires only connect ports of **matching width**: you cannot drop a 1-bit wire onto an 8-bit pin or vice-versa. The bench refuses the connection and shows a hint that names the fix — add a **SPLITTER** to break a bus into wires, or a **MERGER** to combine wires into a bus. (If you later change a pin's width in the definition editor, any instance wires that no longer fit are dropped when you save, with a count in the toast — the same way removing a pin drops its wires.)

---

7. Memory devices: the box that opens to bytes

Everything else on the bench is gates, all the way down. **RAM and ROM are the one deliberate exception.** A 256-byte memory built from real latches would be ~20,000 gates that dominate every simulation and teach nothing that sixteen latches don't — so bulk memory is a **behavioral device**: a single chip whose *contents* you inspect instead of its *gates*. It's the same bargain a real retro kit makes — you study the CPU; the RAM chip beside it stays a labelled rectangle you trust. The box still opens, it just opens to **bytes**.

Place a **RAM** or **ROM** from the **MEMORY** section of the palette (both default to 256 words × 8 bits). The device has bus pins **ADDR[n]** and **DATA[n]**, single-bit **WE** (write-enable) and **CS** (chip-select), and a bus output **DQ[n]** — wire them with M2 buses (mergers/splitters or bus pins). The data output shows the **addressed byte** combinationally whenever CS is high; a **write** commits `DATA → [ADDR]` on the **rising edge of WE** (so the new byte appears on the next step, exactly like a real synchronous RAM). **ROM ignores writes** and carries its program as fixed contents.

The hex inspector

Double-click a device to open its contents as a hex grid — this is the device's "inside", the way drilling into a chip shows its gates. The grid shows every byte (16 per row, with address gutters); the cell at the **currently-driven address glows** as the machine reads it. While the bench is **paused** (no clock free-running), a RAM's cells are **editable**: click a cell, type a new hex value, press Enter — the byte is written into the device and held. ROM is always read-only. Esc or ✕ closes the inspector; "look freely, edit deliberately" is the same rule chips follow.

The gate-built **4×4 RAM** preset stays in the library as the companion exhibit: when you want to see how memory *actually* works from latches, open that one and drill in.

The LED-matrix framebuffer

The **LED MATRIX** in the MEMORY section is the same bargain one step further — memory that *is* a picture. It's a behavioral device like RAM (same **ADDR/DATA/WE/CS/DQ** bus pins, written on the rising edge of WE), but it holds just **8 bytes** and renders them as an **8×8 LED panel**: each byte is one row, each bit one pixel (the most significant bit lights the leftmost pixel). The panel updates live as the bytes change. **Double-click it** and the "inside" is the **byte↔pixel view**: each row shows its address, hex byte, bit pattern and pixels aligned side by side — so `0x7E = 01111110 = .XXXXXX.` reads straight across. While the bench is **paused** (clock at STEP), **click any pixel to toggle it** and edit the picture by hand. This is a *framebuffer*, the oldest idea in computer graphics — and wired to a CPU's address bus it becomes a memory-mapped screen (see the *LB-8 (graphics)* preset under CPU).

---

8. Programs: assemble, load, run, step

Once a bench has a **board CPU** — a chip whose definition exports a PC probe plus the memory device it fetches from — you can stop hand-wiring ROMs and just *write programs*. The **Tiny 4-Bit Computer II (loadable)** preset is the reference board; load it, then click **ASM** in the toolbar to open the assembler panel on the right.

Writing and assembling

Type assembly in the editor. The grammar is small and CPU-agnostic (it reads the CPU's instruction table, not hard-coded opcodes):

- **Comments** start with `;` and run to end of line.
- **Labels** are a name ending in `:` (e.g. `loop:`); reference one as an operand and the assembler fills in its address.
- **Directives**: `.org N` sets the assembly address; `.byte v, v…` / `.word v, v…` emit raw data.

- **Instructions** are `MNEMONIC [operand]`. Operands are decimal (`5`), hex (`$0A` or `0x0A`), binary (`%0101` or `0b0101`), a character (`'A'`), or a label.

Press **ASSEMBLE**. The status line reads `OK — N bytes` or the **first error** (`L4: operand 20 out of range 0..15`); errors also list inline. On success a **listing** appears: one row per instruction showing `addr: bytes mnemonic` — the exact bytes the machine will run.

Loading and running

- **LOAD** writes the assembled image into the board's ROM (and cold-restarts). It's the program's *init image*, so it survives **RESET** and is saved when you export the circuit. Double-click the ROM to see the loaded bytes in the hex grid. LOAD is enabled only once you've assembled with a board CPU present.
- **RESET** restarts the machine from cold — every register clears and the device re-seeds from its loaded image (the CPU's reset rule; Tiny CPU II resets to PC = 0).
- **RUN / PAUSE** free-runs the clock or freezes it. The **REGISTERS** panel shows PC, ACC, flags and the fetch tap live, flashing the ones that change.
- **STEP INSTR** executes exactly one instruction (one clock cycle on this single-cycle machine), so you can walk the control flow by hand.

Breakpoints and the PC highlight

While a program is listed, the row at the **current PC is highlighted** — that's the instruction about to execute. Click a row's **gutter dot** to arm a **breakpoint** (it turns red); during RUN the machine drops to PAUSE the moment PC reaches it. Click again to clear. Between the highlight and the registers you can follow a program one instruction at a time without ever leaving the panel.

---

9. Circuit library

All **48 built-in presets**, in sidebar order. Loading a preset replaces the bench and resets simulation memory, so every circuit starts from its intended pristine state.

BASICS (2)

Ring Oscillator

Three inverters in a closed loop — 4 components. The signal chases its own tail forever; there is no stable state. Nothing to drive: just load it and watch the unstable-node highlight pulse, the rack warning, and the toolbar chip read OSCILLATING. Teaching point: this is exactly what the oscillation detector exists for.

3-Input Majority Voter

MAJ = AB | BC | AC: three ANDs detect each pair, an OR tree merges the votes. Turn on any two of switches A, B, C — MAJ lights; drop one and it goes dark. Teaching point: the fault-tolerance classic used in triple-redundant systems to outvote a failing sensor.

UNIVERSAL GATES (3)

XOR from NAND

The classic 4-NAND construction of exclusive-OR. Set exactly one of switches A or B high — OUT lights only while the inputs differ. Teaching point: the standard proof step that NAND alone suffices to build anything.

Gates from NAND

Two shared switches A and B feed five NAND-only rows (21 components) that recreate NOT, AND, OR, NOR and XOR, each with its own result LED. Toggle A and B and read all five truth tables at once. Teaching point: NAND is functionally complete — any digital circuit, your CPU included, can be built from NAND alone. Note the NOT row: a NAND with *both* inputs tied to one signal.

Gates from NOR

The exact dual bench (22 components): the same five-row topology built from NOR gates yields NOT, OR, AND, NAND and XOR (the XOR row needs one extra NOR-as-NOT, since the 4-gate dual construction gives XNOR). Same drive: toggle A and B. Teaching point: NOR is the other universal gate — the Apollo Guidance Computer was built entirely from NOR gates.

ARITHMETIC (10)

Half Adder

The smallest unit of binary arithmetic: XOR makes SUM, AND makes CARRY. Switch on both A and B — SUM goes dark while CARRY lights, because 1 + 1 = 10 in binary.

Full Adder

Adds three bits (A, B, carry-in CIN) from two half adders plus an OR merging the carries. Switch all three inputs on — SUM and COUT both light (1 + 1 + 1 = 11 binary). Teaching point: chain these and you get a multi-bit adder.

4-Bit Adder

Four full-adder stages (35 components) whose carries ripple downward; A + B + CIN appears on LEDs S0–S3, on COUT, and on a SUM hex display fed by the four sum bits. Try A = 0110 and B = 0101 — the display reads **b** (decimal 11).

8-Bit Adder

The same ripple-carry idea scaled to a full byte: 8 stages, 66 components, up to 255 + 255 + 1. Try A = 3 (A0, A1) and B = 5 (B0, B2) — S3 lights, reading 8. Teaching point: watch how far the carry has to travel from bit 0 to bit 7 — the reason real CPUs use faster adder schemes.

Half Subtractor

A – B in one bit: XOR makes DIFF, NOT + AND raise BORROW when B exceeds A. With A off and B on, DIFF and BORROW both light (0 – 1 needs a borrow). The mirror image of the half adder.

Full Subtractor

A – B – BIN → DIFF and a borrow-out BOUT for chaining. It shares the XOR core of the full adder, with inverters steering the borrow instead of the carry. Try B and BIN on with A off — DIFF lights and BOUT signals the borrow.

2-Bit Multiplier

A1A0 × B1B0 → P0–P3: four ANDs form the partial products, two half adders sum the middle column. Set A = 3 and B = 3 — P0 and P3 light, reading 1001 = 9. A real, if tiny, hardware multiplier.

4-Bit Equality Comparator

Per-bit XNOR (XOR + NOT) checks each position; an AND tree lights **A=B** only when all four agree. With everything off, A=B is already lit (0 equals 0); flip any single switch and it goes dark until you match it on the other side.

4-Bit Adder from Chips

The pyramid demo — gates → chip → board. The exact ripple-carry adder again, but each full-adder band is one **FULL ADDER** chip (2 XOR, 2 AND and an OR behind pins A, B, CIN → SUM, COUT): four instances U0–U3 of a single definition, chained COUT → CIN, with the A/B switches, S0–S3 LEDs, SUM display and COUT LED of the classic preset. Loading it puts FULL ADDER on the CHIPS shelf. Set A = 9 and B = 5 — the display reads E (14). Double-click any instance to watch its gates work live; edit the definition once and all four instances change. Teaching point: building sealed abstractions and composing them is how all real hardware design works — hierarchy.

8-Bit Adder (Bus)

The same byte-wide adder, but the wiring stays humane. A **bus** is one wire on screen carrying *n* signals underneath: the two operand banks are merged onto single **/8 buses** that ride into one **8-BIT ADDER** chip as two fat strands instead of sixteen loose wires, and the SUM bus is split back onto the LEDs. Bus wires draw thicker with a `/8` tick — hover one to read its live value in hex; the chip's pins are bus pins (`A[8]`, `B[8]`). Set A = 60 and B = 9 — the LEDs read 69 (0x45). Double-click the chip to watch the bus split into eight FULL ADDER chips, chained by their carry, then merged back. Teaching point: a bus is the same honest abstraction as a chip, applied to wires — nothing is hidden, only bundled (drop a **SPLITTER** or **MERGER** from the BUS palette section to go between a bus and its bits).

MEMORY (6)

SR Latch (NOR)

The simplest stored bit: two cross-coupled NOR gates remember which of S or R was pulsed last. Flip S on then off again — Q stays lit until you pulse R. Teaching point: thanks to warm-start simulation, the latch genuinely holds its state across changes.

D Latch (Gated)

The classic 4-NAND gated latch: while EN is high, Q transparently follows D; drop EN and the cross-coupled pair freezes. Set D and EN on, switch EN off, then flip D — Q does not budge. The level-sensitive counterpart of the edge-triggered D flip-flop in SEQUENTIAL.

4-Bit Register

Four gated D latches sharing one LOAD line (25 components) — exactly how a CPU register stores a nibble. With LOAD on, Q0–Q3 follow D0–D3; drop LOAD and the value is held. Try: set D = 1010 with LOAD on, switch LOAD off, then scramble the D switches — Q keeps 1010.

4×4 RAM

Four 4-bit words of genuine read/write memory built from sixteen gated D latches (114 components) — addressed storage, the step beyond a single register. Dial a word with the ADDR switches (A1 A0), set DATA on D0–D3, and hold the WRITE button: the addressed word captures the value and keeps it when you release. The Q0–Q3 LEDs and the OUT display always show the *currently addressed* word. Try: write 0xA to word 0 and 0x5 to word 1, then flip ADDR back and forth — both values persist, and scrambling the DATA switches with WRITE released changes nothing. This is exactly the memory wired into Tiny 4-Bit Computer II. (For memory at practical scale, see the behavioral *256×8 RAM (device)* below — same idea, inspected as bytes instead of latches.)

256×8 RAM (device)

Random-access memory at practical scale: a 256-byte **behavioral RAM device** wired over /8 buses — eight ADDR switches and eight DATA switches feed it through

mergers, a WRITE button strobes WE, and the data bus reads out on eight LEDs and a two-digit hex display. Unlike the gate-built 4×4 RAM above you don't see latches inside — you **double-click it to inspect and edit the 256 bytes** in a hex grid (the device's "inside" is its contents). Try: set ADDR = 5 (A0, A2), DATA = 0x2A (D1, D3, D5), tap WRITE — the readout shows 2A and holds; double-click to see byte 05 = 2A. See *Memory devices*.

Lookup ROM (squares)

A 16-byte **read-only** memory holding the squares n^2 for n = 0..15 — the kind of preprogrammed table real machines use for fast lookups. Dial the 4-bit address on A0–A3 and the stored byte appears on the data bus, LEDs and hex display; the contents never change. Double-click to view the whole table. Try: ADDR = 12 (A2, A3) → 144 (0x90).

SEQUENTIAL (6)

All sequential presets include their own CLOCK component with a CLK monitor LED; use the clock's fader to slow things down or to single-step in STEP mode.

Clock Divider (T Flip-Flop)

A master-slave T flip-flop from nine NANDs: the master latch is open while CLK is high, the slave while CLK is low, so Q updates exactly on the falling edge — and feeding the slave $\overline{Q}$ back as D makes it toggle. Watch Q blink at half the rate of CLK. Teaching point: edge-triggering and divide-by-two in one circuit; this 9-NAND core is the building block of everything below.

4-Bit Ripple Counter

Four falling-edge T flip-flops in a chain (43 components): each stage is clocked by the previous stage's Q, so every bit runs at half the rate of the bit before. Q0–Q3 drive a hex display that counts 0 through F and wraps. Load it, watch COUNT tick upward, and change the clock speed to taste.

D Flip-Flop (Edge-Triggered)

The 9-NAND master-slave core with a D switch on the master input. Unlike the gated D Latch, Q never follows D directly — it samples D only at the falling clock edge.

Change D between edges and watch Q wait for the next falling edge before updating.

JK Flip-Flop

Steering logic D = (J AND NOT Q) OR (NOT K AND Q) in front of the same D core gives four modes per falling edge: J=K=0 hold, J=1 K=0 set, J=0 K=1 reset, J=K=1 toggle. Set J=K=1 and watch Q toggle every cycle, exactly like a T flip-flop.

8-LED Running Light (Ring Counter)

A ring counter (89 components): eight D flip-flops pass one circulating hot bit along the L0–L7 LED strip on every falling edge of the 4 Hz clock. Stage 0 samples NOR(Q0…Q6), so the all-zero cold start injects the bit by itself and stray extra bits die out within a lap — self-starting and self-correcting. Watch the marquee chase, then play with the clock fader.

Binary Clock (Full Circuit)

The largest preset bar the CPU: a gate-level 24-hour BCD clock — six synchronous digit counters built from **17 master-slave D flip-flops** plus next-state logic (273 components, 511 wires), all on one shared 1 Hz clock. The carry chain rolls seconds into minutes into hours and forces 23 → 00 at midnight. Read each of the 20 LED-matrix columns bottom-up as 1-2-4-8. It powers up at 00:00:00 like the real desk gadget; while you hold **SET M** or **SET H**, every tick advances that field. Try: crank the fader to 4 Hz and hold SET M to set the minutes fast, or use STEP mode for precise single increments. This preset *is* the machinery behind the Binary Clock (BCD) instrument under ROUTING & CODES.

CPU (14)

4-Bit ALU

An arithmetic logic unit (77 components) — the part of a CPU that actually computes. OP1 OP0 select the operation: **00 ADD, 01 SUB, 10 AND, 11 OR**; results go to LEDs R0–R3, a RES hex display, and COUT/ZERO flag LEDs. SUB uses the classic trick: each B bit is XORed with OP0 and OP0 also feeds the carry-in, so the same adder computes A + NOT B + 1 (two's complement). Try A = 9, B = 5, OP0 on (SUB) — RES

shows 4 with COUT lit, meaning no borrow. The same B-invert adder executes ADD and SUB inside the Tiny 4-Bit Computer.

Instruction Decoder (3-to-8)

How a CPU turns an opcode into action: three opcode switches OP2 OP1 OP0 decode to exactly one hot output LED, labelled with the mnemonics NOP, LDA, STA, ADD, SUB, JMP, JZ, HLT. Dial in opcode 011 (OP0 and OP1 on) — the ADD line lights, all others stay dark. Each hot line would enable that instruction's control signals in a real machine.

Program Counter (4-Bit)

The register that holds the next instruction's address. An AND gates the clock with a RUN switch: RUN on → four chained T flip-flops count 0–F on the PC display; RUN off → the gated clock (CLK LED) parks low and the count holds. Flip RUN off mid-count — the address freezes; flip it back and counting resumes. Teaching point: gating the clock is exactly what HLT does to a real machine.

Tiny 4-Bit Computer

A complete single-cycle stored-program accumulator CPU — **216 components, 413 wires**, everything built from gates; the only sequential primitives are eight of the proven 9-NAND master-slave D flip-flops (4 for the program counter PC, 4 for the accumulator ACC). On the bench: PC and ACC hex displays, A0–A3 accumulator LEDs, Z/JMP/HLT flag LEDs, a RUN switch and a 1 Hz clock. The instruction at PC is decoded combinationally from a mask ROM and executes on the same falling clock edge that moves PC.

ISA — each 7-bit word is `op[2:0] ++ imm[3:0]`:

Opcode	Mnemonic	Effect
000	NOP	nothing
001	LDI	ACC ← imm
010	ADD	ACC ← ACC + imm
011	SUB	ACC ← ACC − imm (two's complement)
100	JMP	PC ← imm
101	JZ	PC ← imm if ACC = 0
111	HLT	freeze the internal clock

(110 is unused and decodes to no control lines, i.e. behaves like NOP.)

The burned-in program:

Addr	Instruction	Comment
0	LDI 0	clear ACC
1	ADD 4	count up by 4
2	JZ 4	wrapped to 0? move on
3	JMP 1	keep adding
4	LDI 9	start the countdown
5	SUB 1	decrement
6	JZ 8	reached 0? halt
7	JMP 5	keep subtracting
8	HLT	freeze
9–15	(zero)	NOP

Expected run: switch RUN on and watch ACC step **4, 8, C, 0** (the add loop, wrapping mod 16), then load **9 and count down 8, 7, … 0**, after which the **HLT** lamp lights and the gated clock parks low — the machine is frozen for good, exactly like a halted CPU. Restarting requires reloading the preset.

Single-stepping advice: slide the clock fader to **STEP** and press-and-release the pad to execute exactly one instruction per pulse, watching PC and ACC move in

lockstep, the Z flag track ACC = 0, and JMP flash on every taken jump. This is by far the best way to follow the control flow.

Reprogramming = rewiring the ROM. The ROM is not a black box: each *nonzero* program word gets one address-decode AND cascade off the PC's Q/Q̄ lines, and each of the 7 instruction-word bits is an OR tree over its contributing addresses (a bit with no contributors is wired from the LOW constant). Change which address lines feed each word-bit OR tree and you have written a new program. The building blocks star solo in the 4-Bit ALU, Instruction Decoder (3-to-8) and Program Counter (4-Bit) presets.

Tiny 4-Bit Computer II (RAM)

The full von Neumann upgrade — **350 components, 691 wires**: the Tiny CPU's fetch/decode/execute machinery plus a gate-built 4-word × 4-bit RAM (the same latch design as the 4×4 RAM preset). The ISA is reworked so arithmetic operates on *memory*, not immediates:

Opcode	Mnemonic	Effect
000	HLT	freeze — so **empty ROM words halt the machine** by design
001	LDI	ACC ← imm
010	ADD	ACC ← ACC + mem[imm]
011	SUB	ACC ← ACC − mem[imm]
100	JMP	PC ← imm
101	JZ	PC ← imm if ACC = 0
110	LDA	ACC ← mem[imm]
111	STA	mem[imm] ← ACC

The burned-in program computes Fibonacci mod 16 using memory words m0 and m1 as the variables:

Addr	Instruction	Comment
0–2	LDI 1 · STA 0 · STA 1	m0 = m1 = 1
3–4	LDA 0 · ADD 1	ACC = m0 + m1
5	JZ 12	wrapped to 0? jump into empty ROM = halt
6	STA 2	m2 = next value
7–10	LDA 1 · STA 0 · LDA 2 · STA 1	shift the pair: (m0, m1) ← (m1, m2)
11	JMP 3	next lap

Expected run: the M0/M1/M2 displays show the RAM live; switch RUN on and watch the pair (M0, M1) walk **(1,1) → (1,2) → (2,3) → (3,5) → (5,8) → (8,D) → (D,5) → (5,2) → (2,7) → (7,9)** while ACC carries each fresh Fibonacci value. On the tenth lap 7 + 9 = 16 ≡ 0, the Z flag fires JZ into address 12 — an empty (all-zero) ROM word, which decodes to HLT: the machine has halted by *falling off the end of its program*, a genuine real-world failure mode turned into a feature. STA writes are level-timed: the addressed word is transparent during the clock's high phase (capturing the stable ACC) and latches on the same falling edge that advances PC — single-cycle store semantics, exactly as in real hardware.

Tiny 4-Bit Computer II (loadable)

The same single-cycle Tiny CPU II, re-cast as a **single-board computer**: the CPU is one CHIP (`CPU`) and its program lives in a **separate ROM device** on the bench, wired over a `/4` **ADDR** bus (PC out) and a `/8` **DATA** bus (instruction word back). The chip **exports probes**, so the **REGISTERS** panel shows PC, ACC, the Z flag and a FETCH tap live — no drilling required. Because the ROM is a board device, the program is **loadable** and **inspectable** (double-click the ROM). This is the machine the **ASM** panel targets: open ASM, ASSEMBLE the seeded Fibonacci program (or your own), LOAD it, then RUN / STEP / breakpoint — see *Programs: assemble, load, run, step*.

LB-8 (8-bit CPU)

The first **real** CPU on the bench, and the biggest jump in the library. Where the Tiny machines do one instruction per clock with a hard-wired mask ROM, the LB-8 is a

multi-cycle, microsequenced 8-bit computer: a control unit reads **microcode** and walks the datapath through *fetch* → *operand-fetch* → *execute*, **one micro-step per clock edge**. It has an 8-bit accumulator **A**, an 8-bit index register **X**, a **Z** (zero) and **C** (carry) flag pair, and addresses a full **256-byte** RAM. Like the loadable Tiny CPU II it is a true single-board computer — the CPU is one CHIP and its program lives in a **board RAM device**, here over an `/8` **ADDR** bus and an `/8` **DATA** bus (both directions: instruction/operand fetch *and* stores write back over the same data path with a WE strobe).

Three addressing modes, the thing that makes it feel like a 6502 rather than a toy: immediate `#$05` (the byte is in the program), absolute `$40` (operate on `mem[0x40]`), and absolute-indexed `$40,X` (operate on `mem[0x40 + X]` — the basis of array/table loops). The **ASM** panel targets the LB-8 automatically when this board is loaded (the assembler/disassembler are driven by the LB-8 ISA table), so `LDA #5`, `STA $40,X` and friends assemble straight onto the bench.

ISA (byte opcode + optional trailing operand byte; ~21 encodings):

Mnemonic	Modes	Effect
NOP / HLT	—	do nothing / freeze the CPU
LDA	#, abs, abs,X	A ← operand
LDX	#, abs	X ← operand
STA	abs, abs,X	mem[addr] ← A
ADD / SUB	#, abs, abs,X	A ← A ± operand, sets Z and C
AND	#, abs	A ← A & operand, sets Z
INX	—	X ← X + 1
JMP	abs	PC ← addr
JZ / JC	abs	branch if Z=1 / C=1

How the microsequencer works (drill into the `CPU` chip to see it). A 3-bit **T-state counter** (`TCOUNT`) counts micro-steps within an instruction. The address `{IR << 3 | T}` indexes **three control-word ROMs** whose 21 output bits *are* the control signals for that step — which register loads, what the ALU does, which source drives the single internal **IBUS** (a one-hot set of source-enables: PC, A, X, ALU, memory,

operand latch). Fetch is **T0**; the last micro-step of every instruction asserts `tReset`, which zeroes the T-counter and starts the next fetch. This `{opcode, T} → control word` table is generated from the *same* ISA spec that drives the assembler, so the encoder, disassembler and the hardware can never disagree. Two `ALU8` instances do the work: one computes `A op operand`, the other adds `base + X` to form an indexed effective address.

The loaded demo computes a Fibonacci sequence in memory and **halts on overflow**: it keeps `a` and `b` in `mem[0x40]` / `mem[0x41]`, and each lap does `LDA $40 · ADD $41`; the first sum that carries past 255 trips **JC done → HLT**. The pair that overflows is 144 + 233 = 377, so the machine stops with A = 121 (377 mod 256) and C = 1.

Expected run: load *LB-8 (8-bit CPU)* under CPU, open **REGISTERS** (it shows PC, A, X, Z, C and a FETCH tap), flip **RUN** and raise the clock — watch **A** climb 1, 2, 3, 5, 8, 13, 21, 34, 55, 89, 144 and then **HLT** when the next sum carries. **Double-click the RAM** to watch the bytes change as STA writes land. **Drill into `CPU`** (it's a CHIP) to see the T-counter, the three microcode ROMs and the two ALUs in action. Use **STEP** on the clock to advance the datapath one micro-step at a time — the cleanest way to see fetch/operand/execute unfold. This is the stepping stone toward a gate-built 6502.

LB-8 (I/O board)

The LB-8 wired to the outside world through **memory-mapped I/O** — the idea that talking to hardware needs *no new instructions*, just `STA` / `LDA` at special addresses. The top of memory becomes doorways to devices: `STA $F0` lights an **8-LED row**, `STA $F1` drives a **7-seg** (shown as two hex digits), `LDA $F3` reads **8 switches** into A, `LDA $F4` reads a **button**. A single gate-built **IO controller** chip makes it work, and you can **drill into it**: an address decoder picks out `$F0–$FF`, two `REG8` registers latch the output writes, a per-bit mux steers reads from the switches/button or the RAM, and it drives the RAM's CS so memory cleanly ignores the I/O page. The loaded demo loops `LDA $F3 · STA $F0 · STA $F1` forever. **Expected run:** open ASM, click **RUN**, raise the clock, then flip the switches — the LED row and the hex display **mirror them live**. Try: set the switches to 0x2A and read it back as the LED pattern and “2A” on the 7-seg.

LB-8 (graphics)

The LB-8 driving an **8×8 LED-matrix framebuffer** at `$F8–$FF` — a *framebuffer* is the oldest idea in computer graphics: a block of memory **is** the picture. The 8 bytes at `$F8–$FF` are the 8 rows of the panel (each bit a pixel, MSB on the left), so drawing is just `STA` at the right address: `STA $FA` with A = `0x7E` paints row 2 as `.XXXXXX.` — the bits *are* the pixels. The same IO controller decodes `$F8–$FF` to the **SCREEN** device exactly as it decodes the LEDs/switches — drill in to see the shared gate logic. The loaded demo paints a **heart** row by row, then halts. **Expected run:** open ASM, **RUN**, raise the clock, and watch it draw. **Double-click the SCREEN** for the **byte↔pixel inspector** (address · hex · bits · pixels aligned per row); pause the clock (set it to **STEP**) and **click pixels to edit the picture** by hand. Try: pause and draw your own 8×8 sprite, or write a program that animates a bouncing pixel.

6502 CPU (behavioral)

The first **real, historical** processor on the bench: a working **MOS 6502** — the chip behind the Apple II, the Commodore 64, the BBC Micro and the NES. Here it is a single bus-master leaf whose "inside" is a cycle-correct emulator (verified against the **Klaus Dörmann functional suite** — 30,646,176 instructions, zero errors), owning its 64 KB memory and exposing **PC, A, X, Y, S, P** live to the REGISTERS panel. Its face carries a live **32×32 screen** — a window into the framebuffer at `$8000`, where each `STA` lights a pixel. The loaded demo draws a corner-to-corner diagonal. **Expected run:** flip RUN at 1 Hz and watch the diagonal draw pixel by pixel while the registers move; drop to .5 Hz for slow motion, or crank to 4 Hz to fill at once; set **STEP** and use **STEP INSTR** to execute one instruction at a time (the current one is marked in the ASM listing). Open ASM to write your own program and LOAD it. Teaching point: every register here — A (accumulator), X/Y (index), S (stack pointer), PC, P (flags) — is something you have already built; this is the map for the gate-built version below.

6502 — Bouncing Ball

The flagship demo: a real 6502 program bounces a pixel around the 32×32 screen, on the behavioral 6502. The clock ships at **4 Hz** (full speed) so RUN animates it smoothly; lower it for slow motion, or **STEP** to single-step (STEP INSTR walks one instruction at a time, the REGISTERS panel showing PC/A/X/Y/S/P move). The ball is

plain 6502 code — it keeps position and velocity in zero page, erases and redraws itself each frame, flips direction at the edges, and paces itself with a busy-wait delay loop (the authentic way a fixed-clock machine controls speed). **Expected run:** press RUN and watch it bounce. Open ASM to read or edit the code, then re-LOAD. Teaching point: the very same machine code runs unchanged on the gate-true 6502 (below) — a real program, ready to meet real gates.

6502 — Gate-Built

The monument. The same 6502 instruction set as the behavioral leaf, but here the CPU is one **CHIP built entirely from logic gates**: a microcoded datapath — a 16-bit PC, the 8-bit ALU, the effective-address adder, the stack pointer, the flag latches and the A/X/Y/IR/DL registers — sequenced by control ROMs addressed by (opcode, T-state), exactly the microcode mechanism of the LB-8 driving a real ISA. It sits on a real bench: a **64K RAM** over a 16-bit address bus and an 8-bit data bus, with CLOCK, RUN and **RST**. On power-up it **boots like silicon** — reading the reset vector from `$FFFC/$FFFD` and jumping there, winding the stack pointer down to `$FD`. The loaded program is the diagonal. **Expected run:** at 1 Hz, RUN draws it pixel by pixel on the OUTPUT-RACK screen; **STEP INSTR** walks it one instruction at a time. **Drill into the M6502 CHIP** to watch the microcode ROMs and datapath move per T-state — then keep going: into the ALU, its adder, a full adder, a single NAND. Honest note: clocking ~6,000 gate nodes per cycle is too slow to animate, so **RUN fast-forwards a verified-equivalent emulator** and mirrors the result onto the screen and gate flip-flops, while **STEP runs the real gates** — always one step away. Teaching point: the most famous chip of its era is, all the way down, the logic you have built since the AND gate.

6502 — Gate-Built Ball

The north star. The bouncing-ball program — the very same 6502 machine code from the behavioral demo — executing on the CPU built entirely from gates. It boots through `$FFFC` and runs from a 64K RAM over the SBC bus. Because clocking ~6,000 gate nodes per frame is far too slow to animate, **RUN fast-forwards the cycle-correct emulator** (proven instruction-for-instruction identical to the gates) and mirrors the result onto the screen and gate flip-flops — so the ball flies while the **REGISTERS** stay live. Then **PAUSE and STEP INSTR** to hand control back to the actual gates and walk it one instruction at a time, or **drill into the M6502 CHIP** to

watch the microcoded datapath. The clock ships at 4 Hz; lower it for slow motion, set STEP to single-step. **Expected run:** press RUN, watch the ball bounce on a gate-built 6502; pause, step the real gates, and open the box all the way down to one NAND. Teaching point: this is the whole journey in one bench — a real program, on a real gate CPU, that you can watch run and take apart; from a bouncing ball to a single gate, with no magic at any level.

6502 — Pocket Calculator

The $3 Woolworth calculator, rebuilt honestly: a real 4-function calculator (**+ − × ÷**) whose brain is the **same gate-true 6502** as the monument, now running calculator firmware and wired to the one piece of hardware a calculator needs — a **4×4 keypad** in and a **6-digit decimal display** out, over a drillable memory-mapped I/O controller. This is the board that proves the 6502 can do real interactive I/O, not just draw to a screen: keypad and display are plain `STA` / `LDA` at special addresses on the **`$D0xx`** page (`$D000` selects a keypad column, `$D001` reads the rows back, `$D010`–`$D015` latch the six digits). **Expected run:** flip **RUN**, then tap the keypad — `7 + 8 =`, `9 × 6 =` — and watch the digits appear (overflow or ÷0 shows **E**). The math is plain 6502 machine code (open the ASM panel to read it): it scans the keypad column by column, debounces and edge-detects each press, builds the operands, and runs binary multiply/divide and add/subtract routines, converting to decimal only to drive the seven-segment digits. **Two new drillable chips:** double-click the **I/O controller** to see the `$D0xx` address decoder, the `REG8` latches for the column-select and the six digits, and the per-bit mux steering the keypad rows onto the data bus (modeled exactly on the LB-8 I/O controller); double-click the **4×4 keypad** to see a real scanned matrix — each key is `AND(column, button)`, OR-ed per row, the literal gate model of a diode keypad. Like the other gate-6502 boards, **RUN fast-forwards a verified-equivalent emulator** so typing stays snappy, while **PAUSE + STEP** hands control back to the actual gates — or drill the **M6502** to watch the datapath compute a digit. Teaching point: this is the interactive bookend to the screen demos — the same gate CPU, now reading input and driving a real display, the smallest honest computer that does something you'd recognise as a *device*.

ROUTING & CODES (7)

2:1 Multiplexer

A data selector: with SEL low, OUT follows input A; with SEL high, it follows B (NOT + two gating ANDs + OR). Set A on and B off, then flip SEL — OUT tracks whichever input is selected.

2-to-4 Decoder

Address bits A1 A0 light exactly one of Y0–Y3, with an EN switch gating every output. Switch EN on, then count A1 A0 through 00–11 and watch the lit output walk from Y0 to Y3. Teaching point: the circuit behind address decoding and chip-select lines.

4-Bit Even Parity

A XOR tree folds D0–D3 down to one PARITY bit that lights when an odd number of inputs is high. Turn inputs on one at a time and watch PARITY toggle on every single flip. The check bit used for simple error detection on memory and serial links.

4:1 Multiplexer

Routes one of four data lines D0–D3 to OUT, addressed by S1 S0: cascaded ANDs pass only the selected channel, an OR tree merges the lanes (20 components). Switch on D2 only, then set S1 on and S0 off — OUT lights because channel 2 (binary 10) is selected.

Binary → Gray (4-Bit)

Converts a 4-bit binary value to Gray code, where neighbouring values differ in exactly one bit — beloved by rotary encoders. Each Gray bit is the XOR of two adjacent binary bits; G3 passes straight through. Set B = 1011 (B0, B1, B3) — G reads 1110.

Hex Display Demo

Four switches with binary weights 1, 2, 4, 8 drive a 7-seg hex display directly — the compact way engineers read four bits at a glance. Switch on B8, B2 and B1 — the display reads **b** (11); add B4 and it shows F (15).

Binary Clock (BCD)

A single Binary Clock instrument (1 component, 0 wires) preset to Madrid time. Each lamp column is one decimal digit of HH:MM:SS, read bottom-up as 1, 2, 4, 8. Use the select underneath to switch timezones, and check the minutes against your OS clock. For the same readout built gate by gate, load *Binary Clock (Full Circuit)* under SEQUENTIAL.

---

10. Tips & limits

- **Clocks belong at board level.** A CLOCK placed inside a chip definition does **not** free-run — the ticker only drives clocks sitting on the bench itself (the same applies inside the definition editor; use the PIN test toggles to step a clocked chip by hand there). Feed clocked chips through a PIN IN and keep the clock on the board, where you can see and control it. This is a known limitation, not a feature.
- **STEP mode is your debugger.** Any clock can be parked at the STEP detent and pulsed by hand — one press-and-release is one clean clock cycle. Use it on counters, the binary clock, and especially the Tiny 4-Bit Computer to watch one instruction at a time.
- **Latch states reset on load and clear.** Loading a library circuit, importing a file, or clearing the bench drops the warm-start memory, so feedback circuits come up in their cold-start state (the presets are arranged so this is the intended state, e.g. counters start at 0). Switch positions, by contrast, are part of the saved circuit.
- **HLT is final.** Once the Tiny 4-Bit Computer fetches HLT, its internal clock is gated off permanently — flipping RUN does not revive it. Reload the preset to run again.
- **Self-loop wires look degenerate.** Wiring a component's output directly back into its own input is simulated correctly, but the wire renders as a tiny squashed path. Cosmetic only; route feedback through another gate if you want to see it.
- **LEDs and displays mirror in the rack.** Every LED and 7-seg display on the canvas also appears as a row in the OUTPUT RACK, so you can read final

outputs without hunting across a big circuit. Label your outputs (Rename…) to get meaningful rack names instead of `CH 01` / `DSP 01`.

- **7-seg digits b and d are lowercase** on purpose, so 11 ≠ 8 and 13 ≠ 0 at a glance.
- **Unconnected ≠ floating.** An unwired input is a hard LOW. If a gate "mysteriously" outputs HIGH, check for a NAND/NOR with an open input.
- **The oscillation warning is a feature.** If the status chip flips to OSCILLATING after a wiring change, you have created a loop with no stable state; the rack lists the unstable components so you can find it.