CSSE2010 — Week 3 Notes

Table of contents

Lecture 5 — Introduction to Sequential Circuits **2**
- Outline . . . 2
- Recap . . . 3
- Worked: multiplexer implementing a given function . . . 3
- Circuits that remember values . . . 5
- The D flip-flop . . . 5
- Worked: D flip-flop output waveform . . . 6
- SR latch from NOR gates . . . 7
- Flip-flops vs latches . . . 8
- Combinational vs sequential circuits . . . 9
- Reminders . . . 9
- See also . . . 9

Lecture 6 — Sequential Circuits 1: Shift Registers **10**
- Outline . . . 10
- Recap: flip-flops and sequential circuits . . . 10
- Registers . . . 11
- Shift registers . . . 11
- Exercise: bidirectional shift register . . . 12
- Universal shift register . . . 13
- Wide shift registers . . . 13
- Lab 06 preparation task . . . 14
- Reminders . . . 15

Lab 4 Pre-Work **15**
- Warm-up: three recall questions from Lecture 4 . . . 16
- Circuit 1 — a 2-to-1 multiplexer from 2-input NAND gates only . . . 18
- Circuit 2 — a 2-to-4 decoder . . . 19
- Circuit 3 — a 4-bit two's complement adder/subtractor . . . 20
- In the session . . . 25

Next . . . 26

Logisim and Flip-Flops **26**
Recall . . . 26
Worked: which is NOT a combinational circuit? . . . 27
Worked: which waveform captures the behaviour of this circuit? . . . 27
Worked: which of these is FALSE about a negative-edge-triggered D flip-flop? . . . 28
Floating pins . . . 28
Tasks 1 and 2 — simulate the S-R latches . . . 30
Task 3 — build a `74HCT74` circuit . . . 31
Kit borrowing . . . 32
Next . . . 34

Reference material **34**
Binary Number Representations . . . 34
Boolean Algebra . . . 42
Circuit Schematics . . . 45
Combinational Logic Blocks . . . 48
Counters . . . 59
CSSE2010 Diagram Notation . . . 65
Device Pinouts . . . 81
Flip-Flops and Latches . . . 91
The I/O Board . . . 99
Logic Gates . . . 107
Sequential Circuits . . . 112
Shift Registers . . . 115
Timing Diagrams . . . 123

Lecture 5 — Introduction to Sequential Circuits

See flip-flops-and-latches[1] and sequential-circuits[2] for the reference definitions, characteristic tables and symbols.

Outline

- Recap of combinational logic
- Circuits that remember values
- The D flip-flop: symbol, operation, characteristic table, waveforms
- The SR latch built from NOR gates

[1] courses/csse2010/flip-flops-and-latches.pdf
[2] courses/csse2010/sequential-circuits.pdf

- Flip-flops vs latches; a real D flip-flop and its chips
- Combinational vs sequential circuits; synchronous sequential circuits

Recap

The course so far stacks up in three layers, each built on the one below:

1. **Binary representations** — unsigned, sign-magnitude, 1's complement, 2's complement, excess-2^{N-1} (see binary-number-representations[3], 2026-08-04-binary-arithmetic[4]).
2. **Logic gates** — NOT, AND, OR, NAND, NOR, XOR, XNOR — and Boolean algebra (see logic-gates[5], boolean-algebra[6]).
3. **Combinational logic circuits** — adder, adder/subtractor, multiplexer, decoder (see combinational-logic-blocks[7]).

The circuits built so far are the 4-bit ripple carry adder (four full adders chained by their carries), the 4-bit adder/subtractor with its mode select M ($M = 0$ add, $M = 1$ subtract), the 4:1 MUX with selects S_1S_0, and the 3:8 decoder with inputs A, B, C and outputs $D_0 \dots D_7$. All of them are combinational. Sequential circuits start here.

Worked: multiplexer implementing a given function

Consider a 4:1 multiplexer with data inputs A, B, C, D on lines 0, 1, 2, 3 respectively and select inputs S_1S_0. What must A, B, C, D be so that the multiplexer output is

$$X = S_1.S_0 + \bar{S}_1.G$$

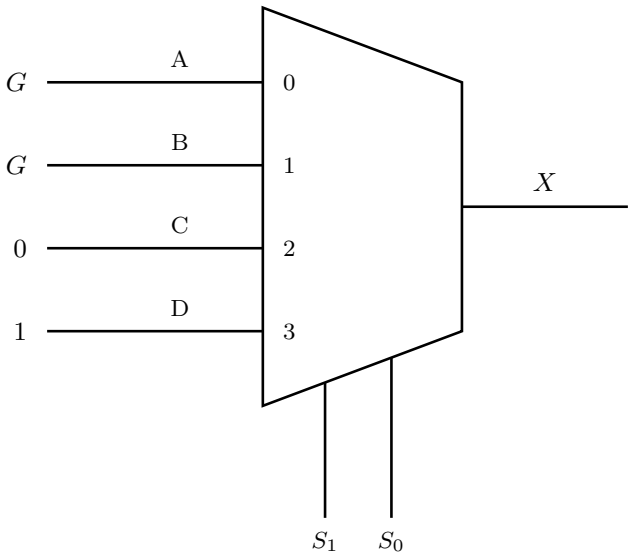
The options: (1) $A = 0, B = 0, C = 0, D = 1$; (2) $A = G, B = G, C = 0, D = 1$; (3) $A = G, B = 0, C = 0, D = 1$; (4) $A = 0, B = G, C = 0, D = 1$; (5) none of the above.

By select code. Walk the four select codes and ask what the multiplexer has to be passing in each case. A 4:1 MUX passes exactly one data input per select code, so whatever X evaluates to for a given S_1S_0 *is* the value that must be sitting on the addressed input.

The same thing algebraically. A 4:1 MUX passes whichever data input its select code addresses, so in general

$$X = \bar{S}_1\bar{S}_0.A + \bar{S}_1S_0.B + S_1\bar{S}_0.C + S_1S_0.D$$

[3] courses/csse2010/binary-number-representations.pdf
[4] courses/csse2010/2026-08-04-binary-arithmetic.pdf
[5] courses/csse2010/logic-gates.pdf
[6] courses/csse2010/boolean-algebra.pdf
[7] courses/csse2010/combinational-logic-blocks.pdf



S_1	S_0	$X = S_1.S_0 + \overline{S_1}.G$	input
0	0	$0.0 + \overline{0}.G = G$	$A = G$
0	1	$0.1 + \overline{0}.G = G$	$B = G$
1	0	$1.0 + \overline{1}.G = 0$	$C = 0$
1	1	$1.1 + \overline{1}.G = 1$	$D = 1$

Figure 1: The multiplexer with the answer wired on, and the working beside it. Each row substitutes one select code into the target expression and reads off the value the addressed data input must carry; the four results down the right-hand column are the answer. Rows 1 and 2 both give G — the term S1-bar . G says nothing about S0, so both of the S1 = 0 inputs have to carry G.

Expand the target into the same four select terms. The $\bar{S}_1.G$ term covers *both* $\bar{S}_1\bar{S}_0$ and $\bar{S}_1 S_0$, and the $S_1.S_0$ term is 1 for that one select code only:

$$X = \bar{S}_1\bar{S}_0.G + \bar{S}_1 S_0.G + S_1\bar{S}_0.0 + S_1 S_0.1$$

Matching term by term against the MUX expression: $A = G$, $B = G$, $C = 0$, $D = 1$.

Answer: option 2.

The trap: it's tempting to put G on only one of the two $\bar{S}_1$ lines, but $\bar{S}_1.G$ says nothing about S_0, so *both* $\bar{S}_1$ inputs must carry G. That is what separates option 2 from options 3 and 4.

Two wider points:

- Each of the four data inputs is one of $\{1, 0, G, \bar{G}\}$. Those are the only four things a data input can ever need to be when a multiplexer is used to implement a function of its select variables plus one extra variable G — a constant, that variable, or its complement.
- Using lookup/selection hardware to realise arbitrary logic is the idea programmable logic (FPGAs) is built on.

Circuits that remember values

- The output of any logic gate or **combinational circuit** depends on the **current value of the inputs only**. A two-input adder block is an example: x and y in, S out.
- If an input changes, the output can change too — and the previous value is **lost forever**.
- In a **sequential circuit**, the current output depends not only on the current inputs but also on the **past** outputs.
- Circuits with **memory** can remember values even when the input changes.

That motivates the whole rest of the course: to build anything that accumulates, counts, or holds a result, you need a storage element.

The D flip-flop

The first memory element is the D flip-flop: **D** is the data input, **Q** the output, and **CLK** the control input — the clock. Q copies the value of D — and remembers it — whenever CLK goes from 0 to 1 (the *rising edge*). A D flip-flop stores 1 bit of information.

Everything below depends on telling the two kinds of clock edge apart.

The full symbol set, the characteristic table and a first worked waveform are in flip-flops-and-latches[8].

[8] courses/csse2010/flip-flops-and-latches.pdf

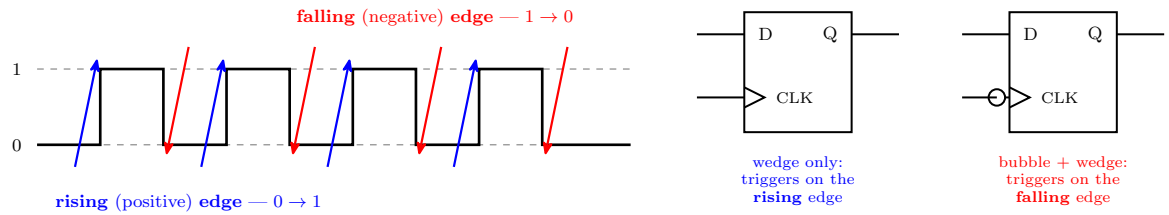


Figure 2: A clock waveform with an up-arrow on every 0-to-1 transition and a down-arrow on every 1-to-0 transition, beside the flip-flop symbols that trigger on each. Colour is only to keep the two edge types apart; it carries no other meaning. A D flip-flop responds to exactly one of these two events, and which one is settled by whether its clock pin carries a bubble.

The characteristic table is the tabular definition of a flip-flop's operation. Its right-hand column $Q(t+1)$ is the *next* output: what the output will be on the next clock edge.

A D flip-flop remembers a single bit, so n D flip-flops remember n bits and an n-bit **register** is by definition n D flip-flops. JK and T flip-flops also exist but are not covered in this course. Flip-flops can be made out of logic gates.

Worked: D flip-flop output waveform

For the rising-edge D flip-flop above, what is the output waveform for Q, given the D and CLK waveforms in the figure?

Reading the figure. The clock has four rising edges, each marked with an arrow. D is low across the first clock pulse, rises after that pulse has ended, stays high across the second and third clock pulses, and falls again *after* the third rising edge but before the fourth.

Rising clock edge	D at that edge	Q after the edge
1st	0	0
2nd	1	1
3rd	1	1 (no change)
4th	0	0

So Q goes high at the **2nd** rising clock edge and stays high until the **4th** rising clock edge, where it returns to 0. Q is *not* a copy of D: it lags D's rise (waiting for the next clock edge) and lags D's fall (holding the 1 until the next clock edge).

Answer: option 3 — the trace that rises at the second rising clock edge and falls at the fourth rising clock edge.

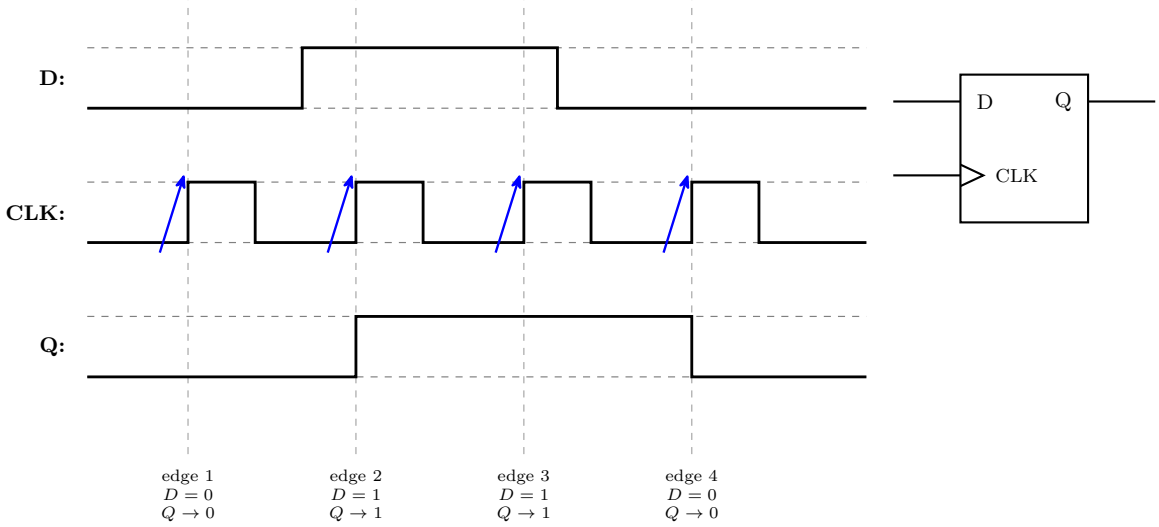


Figure 3: The D and CLK waveforms with the answer, option 3, drawn in as Q. The dashed verticals are the four rising clock edges, each also marked with a blue arrow. Everything about Q is decided at those four instants and nowhere else — which is why Q's edges never line up with D's.

Why the others are wrong:

- **Option 1** rises and falls exactly with D — that's D itself, i.e. no clocking at all.
- **Option 2** rises correctly at the second rising edge but falls at the *falling* edge of the third clock pulse — reacting to the wrong edge, as a negative-edge-triggered device would.
- **Option 4** goes high and low several times inside the window where D is high, dropping back to 0 between clock pulses instead of holding — it is following the clock rather than storing anything, so nothing is being remembered at all.

The general check, worth applying to any question of this shape: a rising-edge flip-flop's output can only change *at* a marked rising edge. Any candidate trace with a transition anywhere else is wrong on sight, whatever D is doing there.

SR latch from NOR gates

Two cross-coupled NOR gates make a storage element: S into the top gate (output $\bar{Q}$), R into the bottom gate (output Q), each gate's output fed back into the other gate's spare input. This circuit is the **active high SR latch**.

The truth table: $SR = 00$ holds (Q unchanged, $Q = Q_{n-1}$ — the storage, or remember, row), 01 resets to 0, 10 sets to 1, and 11 is *invalid*. With $S = R = 1$ the outputs are $Q = \bar{Q} = 0$, and

the two outputs coming out equal is what makes the combination invalid.

The analysis turns on the identity $\overline{0 + x} = 1 \cdot \bar{x} = \bar{x}$ — a NOR with a 0 on one input is an inverter for the other. Its companion is $\overline{1 + x} = 0$ (a NOR with a 1 on one input is pinned to 0); it is the same De Morgan step and the other half of the analysis, and flip-flops-and-latches[9] carries both.

Circuit, both identities and the full row-by-row derivation are in flip-flops-and-latches[10].

Homework: re-analyse the same S-R latch with the NOR gates replaced by NAND gates, and complete the truth table. The result is the **active low SR latch**: the inputs become $\bar{S}$ and $\bar{R}$ and, importantly, the gate outputs swap over so that the *top* gate now carries Q. That relabelling is what keeps "set" meaning set. Drawn and worked through in flip-flops-and-latches[11].

Flip-flops vs latches

Latches are **level** triggered — they respond to changes of logic level on their inputs. Modify a latch so it responds instead to an **edge** (a momentary transition) of a control input, and you have a **flip-flop**: it stores 1 bit while being sensitive only to a clock edge, so it changes its output only *at* the edges and holds the value in between.

A clock has a positive edge (0→1) and a negative edge (1→0), so a D flip-flop can be positive- or negative-edge triggered — see Figure 2 above. In a **positive edge triggered D flip-flop**, input D is copied to output Q at the positive edge of the clock. In between clock edges the flip-flop is not responsive, so it stores the value.

JK and T flip-flops exist but are **not covered in this course**. The **state** of a flip-flop is the value it is storing. Flip-flops are more useful than latches.

The four schematic symbols (plain / bubble / wedge / bubble-and-wedge on the clock pin, with the wedge meaning edge-triggered and the bubble meaning falling edge) are drawn and tabulated in flip-flops-and-latches[12].

A real D flip-flop's internal schematic is six NAND gates in three cross-coupled pairs. Its $\overline{\text{PRE}}$ and $\overline{\text{CLR}}$ inputs are **active low** and **asynchronous**: they reach into the latches directly and do not wait for a clock edge. The symbol is in flip-flops-and-latches[13].

The chips that package them: the **74HCT74** (dual D flip-flop) and the **74HCT273** (eight flip-flops, i.e. one byte). See device-pinouts[14] for the 74HCT74 pinout — the 74HCT273 is not in the course's device-symbols document, so there is no pinout for it in the vault.

[9] courses/csse2010/flip-flops-and-latches.pdf
[10] courses/csse2010/flip-flops-and-latches.pdf
[11] courses/csse2010/flip-flops-and-latches.pdf
[12] courses/csse2010/flip-flops-and-latches.pdf
[13] courses/csse2010/flip-flops-and-latches.pdf
[14] courses/csse2010/device-pinouts.pdf

Combinational vs sequential circuits

The formal contrast:

	Combinational	Sequential
Contents	Logic gates only, no flip-flops	Includes flip-flops
Output	Uniquely determined by the inputs — the same inputs always give the same output	Determined by the current inputs **and** the current **state** (the values in the flip-flops)
When it can change	Whenever an input changes	Only when the clock 'ticks'
Blocks met so far	Adders, multiplexers, decoders, demultiplexers, encoders	Counters, registers

A combinational example, first met in Lecture 2: A into an AND gate whose other input is $B + C$, giving $A(B + C)$ — a circuit with no memory at all.

The general block diagram (combinational logic + flip-flops + feedback path, with the clock entering the flip-flops only) and the definition of a **synchronous** sequential circuit — storage elements can only change at discrete instants of time, namely the clock edges — are drawn in sequential-circuits[15].

Reminders

Coming up:

- **Lab 4** (Mon–Tue this week) — combinational logic. Make sure you attempt the preparation task. Use logic ICs or Logisim software to test your circuits.
- **Lab 5** (Thu–Fri this week) — flip-flops. Use Logisim software to test the circuits. **Kit loaning happens in Lab 5.**

See also

- flip-flops-and-latches[16] · sequential-circuits[17] — the reference notes for everything above
- combinational-logic-blocks[18] — the multiplexer in the first worked question
- timing-diagrams[19] — how the waveform figures are read

[15] courses/csse2010/sequential-circuits.pdf
[16] courses/csse2010/flip-flops-and-latches.pdf
[17] courses/csse2010/sequential-circuits.pdf
[18] courses/csse2010/combinational-logic-blocks.pdf
[19] courses/csse2010/timing-diagrams.pdf

- week3-lab-logisim-and-flip-flops[20] — Lab 5
- 2026-08-13-shift-registers[21] — Lecture 6, the next one

Lecture 6 — Sequential Circuits 1: Shift Registers

See shift-registers[22] for the reference material, and sequential-circuits[23] and flip-flops-and-latches[24] for the material recapped at the start.

Outline

- Admin
- Sequential circuits
- Shift registers

Recap: flip-flops and sequential circuits

First met in Lecture 5[25]; the full detail is in the concept notes.

- **D flip-flop** — D input, Q output, CLK control input; Q copies (and remembers) D on the **rising edge** of CLK. Only D flip-flops are used in this course. Optional **asynchronous SET and CLR** inputs set/clear Q outside clock edges and are typically **active-low**. D flip-flops are what you build sequential circuits from — e.g. [counters]. Full detail in flip-flops-and-latches[26].
- **Combinational vs sequential circuits** — combinational = logic gates only, output uniquely determined by the inputs, so you always get the same output for a given set of inputs. An example is the pair $AB + AC$ and $A(B + C)$ — two ANDs into an OR beside an OR into an AND, with truth tables that agree; see logic-gates[27]. Sequential = includes flip-flops, output determined by current inputs *and* current **state**, and can change when the clock ticks. See sequential-circuits[28].

[20] courses/csse2010/week3-lab-logisim-and-flip-flops.pdf
[21] courses/csse2010/2026-08-13-shift-registers.pdf
[22] courses/csse2010/shift-registers.pdf
[23] courses/csse2010/sequential-circuits.pdf
[24] courses/csse2010/flip-flops-and-latches.pdf
[25] courses/csse2010/2026-08-10-introduction-to-sequential-circuits.pdf
[26] courses/csse2010/flip-flops-and-latches.pdf
[27] courses/csse2010/logic-gates.pdf
[28] courses/csse2010/sequential-circuits.pdf

- **Sequential circuits** — state = values in the flip-flops; present state vs next state; the output depends on input and state (or sometimes just the state), and the next state depends on inputs and state. Mano's block diagram is a combinational circuit feeding flip-flops, with the flip-flop outputs routed back round to the combinational circuit's inputs and `Clock pulses` entering the flip-flops directly. See sequential-circuits[29], and the counter-structure figure in [counters] for the same diagram with its external inputs removed.
- **Synchronous sequential circuit** — storage elements can only change at discrete instants. The clock is a train of narrow pulses, and those instants are its rising edges. Outputs of storage elements change **only on the edges** of the control signal, unlike logic gates whose output changes whenever the input does. The rule: **in a synchronous sequential circuit, all the sequential elements share a common clock signal.**

Registers

A **register** is a group of flip-flops: an n-bit register is n flip-flops storing n bits. Two points:

- a register is a sequential circuit **without any combinational logic** — unlike the general sequential-circuit block diagram above, which has a combinational block in the loop;
- registers store binary information (data/instructions) **inside a processor**.

An example is a 4-bit register with parallel inputs $I_0..I_3$, parallel outputs $A_0..A_3$, and common `Clock` and (active-low) `Clear` lines. It is drawn as the register figure in the shift-registers note[30].

Shift registers

A **shift register** is a register capable of shifting its binary information in one or both directions. The example is Mano's Fig. 6-3: serial input $SI \rightarrow$ four D flip-flops in series $\rightarrow$ serial output SO, all on a common CLK. It is drawn, with a clock-by-clock trace, in the shift-registers note[31].

Two applications:

- **Serial parallel conversion** — shift registers can do serial-to-parallel conversion and vice-versa. The starting point is the plain chain of four D flip-flops on a common `clk`: a lead into the first D on the left, each stage's output running straight into the next stage's D. The conversion adds parallel taps to that chain.

[29] courses/csse2010/sequential-circuits.pdf
[30] courses/csse2010/shift-registers.pdf
[31] courses/csse2010/shift-registers.pdf

- **Parallel load and serial shift** — four D flip-flops on a common `clk` again, with a **multiplexer in front of each flip-flop's D input** to choose between the shift source and the parallel input. See combinational-logic-blocks[32] for the mux, and the shift-registers note[33] for the construction, drawn out.

Exercise: bidirectional shift register

Using the same multiplexer concept, draw a **3-bit shift register which allows data to be shifted in either direction**.

Hint: build it from the element below, where **DIRN is 0 for left shift, 1 for right shift**.

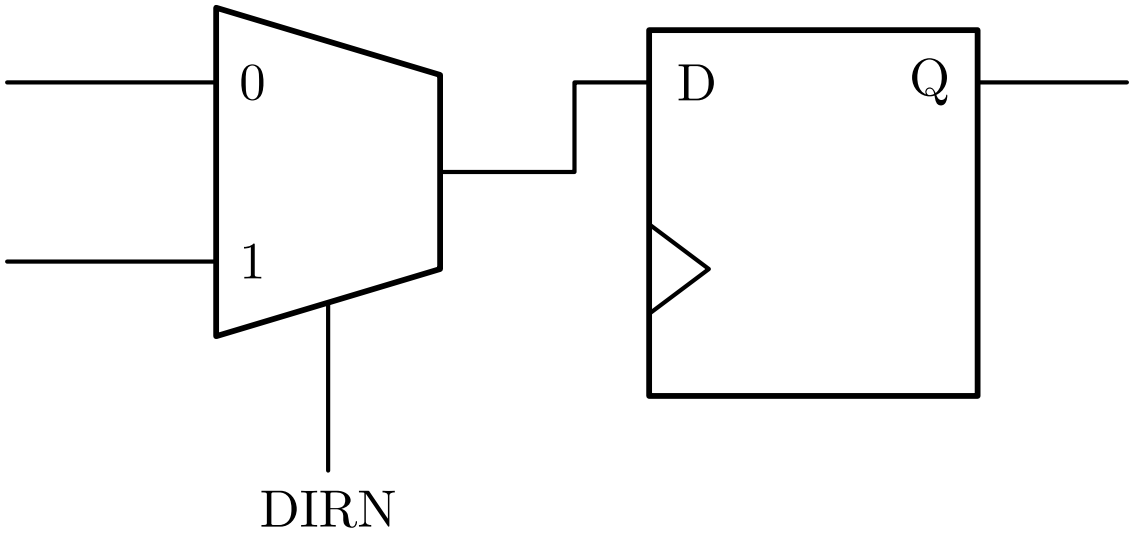


Figure 4: The hint element: a 2-to-1 multiplexer whose data inputs are labelled only 0 and 1, whose select input DIRN enters through the bottom sloping edge, and whose output drives the D input of one edge-triggered D flip-flop. The mux inputs have no names and the clock input no label. One mux plus one flip-flop is a stage, and DIRN picks which neighbour that stage listens to.

Answer. Instantiate three copies of the element, share one `DIRN` line and one clock across all three, and for each stage wire

- **mux input 0** (selected when DIRN = 0, left shift) to the neighbouring stage on the left-shift *source* side, and

[32] courses/csse2010/combinational-logic-blocks.pdf
[33] courses/csse2010/shift-registers.pdf

- **mux input 1** (selected when DIRN = 1, right shift) to the neighbouring stage on the right-shift *source* side,

with an external serial input supplying whichever end of the register has no neighbour in that direction. Writing the bits as $Q_2Q_1Q_0$ with Q_2 most significant, and taking "left shift" to move bits towards the more-significant end:

Stage	Mux input 0 (DIRN=0, left)	Mux input 1 (DIRN=1, right)
Q_2	Q_1	SI_R
Q_1	Q_0	Q_2
Q_0	SI_L	Q_1

Sanity check on $Q_2Q_1Q_0 = 110$ with both serial inputs at 0: one edge with DIRN = 0 gives 100; one edge with DIRN = 1 gives 011. The full wired-up register is drawn in the shift-registers note[34].

The point of the exercise: **a bidirectional shift register is just a unidirectional one with a mux per flip-flop**, and the direction control is a single line shared by every mux, so the whole register commits to one direction per clock edge.

Universal shift register

A universal shift register does hold / shift left / shift right / parallel load in one device, selected by control inputs. See the shift-registers note[35].

Wide shift registers

Shift registers can shift **multiple bits at a time**. An example is a **4-stage, 8-bit queue**: four 8-bit registers in a row (inputs $A..H$, outputs $Q_1..Q_8$, each with an `ENB` enable), each register's outputs feeding the next register's inputs as an 8-bit bus, all on a common clock. A byte at `Input` emerges at `Output` four clock edges later. It is drawn in the shift-registers note[36].

This is also the shape the lab preparation task takes, two stages deep and four bits wide instead of four stages of eight.

[34] courses/csse2010/shift-registers.pdf
[35] courses/csse2010/shift-registers.pdf
[36] courses/csse2010/shift-registers.pdf

Lab 06 preparation task

> **2-digit lock/unlock circuit**: the user inputs two decimal digits (4 bits each) **AB** in serial. The circuit matches the two input digits with a code (say **CD**) and unlocks if the input matches the code (i.e. **AB = CD**).

The **preparation-task handout** specifies the architecture precisely, and it is not what "in serial" might suggest:

- A **two-stage, 4-bit-wide shift register — 8 flip-flops.** The register is *digit*-serial, not bit-serial: each shift moves a whole 4-bit digit from stage 1 to stage 2.
- **Four switches** set a digit value in parallel (0_{10}–15_{10}, i.e. 0000_2–1111_2); a **push button** clocks that digit into stage 1, pushing stage 1's previous contents into stage 2.
- Two LEDs — one for **unlocked**, one for **locked** — plus a **reset** push button. After reset both stages hold `0000`; at all times other than a match, "locked" is on and "unlocked" is off.
- The lock remembers **the last two digits entered**, so it opens on any sequence ending in the code. If the code is 34 and you enter 5, 3, 4, it unlocks on the third digit.

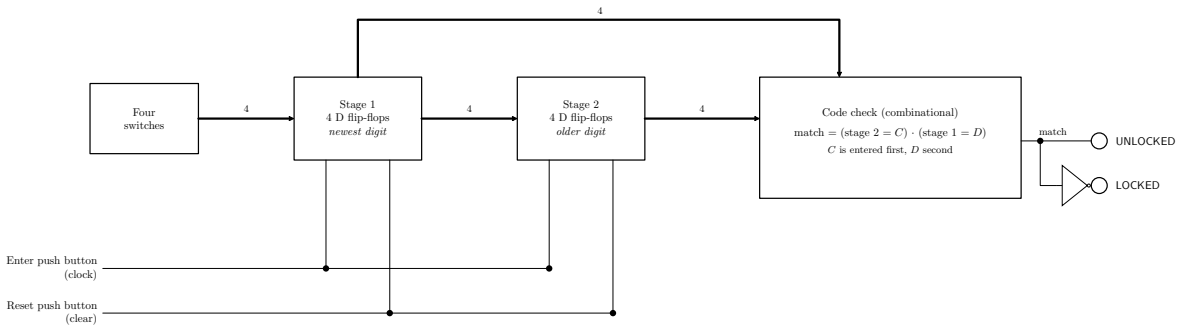


Figure 5: The combination lock at block level. Inputs on the left feed flip-flops in the centre, which feed flip-flops on the right, so the older digit ends up on the right. Hence the match condition shown: the first code digit C is compared against stage 2 and the second digit D against stage 1, not the other way round. Bus widths are lettered beside the heavy lines.

Your code is the **last two digits of your student number**, with any zeroes changed to $10_{10} = 1010_2$; and if the two digits are identical, replace the second with $11_{10} = 1011_2$. Enter the left digit first. (E.g. 40123456 → 5 then 6; 46543210 → 1 then 10; 44444444 → 4 then 11.)

The ordering trap, from the handout's closing NB:

> NB: The natural and eminently reasonable way to begin laying out the logic diagram and/or circuit schematic is to have the inputs on the left, feeding into a set of

> flipflops in the centre, which in turn feed into a set of flipflops on the right. In this case, the entered code moves left to right, such that the older of the two values is stored in the flipflops on the right and the newer on the left. Ensure that your code checking logic reflects this ordering.

Since you enter the left code digit first, that digit is the *older* one by the time the lock should open, so it is the one sitting in **stage 2**. That is the asymmetry Figure 5 spells out; getting it backwards is a lock that opens on the code reversed.

Deliverables: a **logic diagram** *and* a **circuit schematic**, using the device pinouts from Blackboard (see device-pinouts[37] and circuit-schematics[38]). The handout offers Logisim as an option, not a requirement — *"You may also simulate your logic diagram in Logisim before coming to lab sessions"* — and says the circuit is **built during the practical session**, along with variations on it. The comparison itself is combinational — see combinational-logic-blocks[39]. The lab session this feeds is week4-lab-sequential-circuits-shift-registers[40].

The handout recommends watching the week 3, Lecture 6 recording before attempting this task.

Reminders

- **Labs 6 and 7** (next week, week 4) have **preparation tasks** which should be attempted **before** coming to the labs.
- Attempt the weekly exercise and quizzes.

Lab 4 Pre-Work

Lab 4 is the combinational-logic lab, and almost all of its work happens **before** the session. The preparation sheet asks for three circuit schematics; the session itself is spent peer-checking them, getting them signed off by a tutor, and building the third one. Pairs with 2026-08-06-combinational-logic[41] and combinational-logic-blocks[42].

> The device pinout information is on Blackboard (**learning resources → lab resources → digital logic**) and in device-pinouts[43]. The drawing rules every one of the three schematics is marked against are in circuit-schematics[44].

[37] courses/csse2010/device-pinouts.pdf
[38] courses/csse2010/circuit-schematics.pdf
[39] courses/csse2010/combinational-logic-blocks.pdf
[40] courses/csse2010/week4-lab-sequential-circuits-shift-registers.pdf
[41] courses/csse2010/2026-08-06-combinational-logic.pdf
[42] courses/csse2010/combinational-logic-blocks.pdf
[43] courses/csse2010/device-pinouts.pdf
[44] courses/csse2010/circuit-schematics.pdf

Warm-up: three recall questions from Lecture 4

Three multiple-choice questions on combinational logic. They are worth doing before you look at the circuits, because all three are about reading a block symbol correctly, which is exactly what Circuit 3 needs.

What does this symbol represent?

A trapezoid with the **short edge on the left** carrying three inputs A, B, C and the **long edge on the right** carrying eight outputs $D_0 \ldots D_7$.

Answer: a 3:8 decoder. The flare direction is the whole tell — see csse2010-diagram-notation[45]: a multiplexer's wide edge is on the **left** (many data inputs funnelling to one output, selects entering the bottom), a decoder's wide edge is on the **right** (n inputs fanning out to 2^n outputs).

The distractors are worth naming:

- *3:8 multiplexer / 8:3 multiplexer* — a multiplexer has exactly **one** output whatever its size, so eight output stubs rule it out immediately.
- *3:8 de-multiplexer* — the near-miss. A demultiplexer routes a separate **data** input to one of 2^n outputs under n selects; this symbol has only the three address lines and no data input, and no select stubs on the bottom edge.
- *8:3 encoder* — that would be eight in, three out; this is the other way round.

See combinational-logic-blocks[46] for what each block actually does.

Which diagram implements $Z = \overline{S} \cdot X + S \cdot Y$?

Five 2-to-1 multiplexers, differing only in what is wired to the `0` and `1` data inputs.

Answer: A. The mux's data input labels *are* the value of the select that picks them, so $\overline{S}X + SY$ means "X on 0, Y on 1". No complement appears in the expression, so any option with a bar on a data input is out (B, C, E), and D has X and Y the wrong way round.

What should be connected to A, B, C, D so that $X = \overline{S_1 \oplus S_0}$?

A 4-to-1 multiplexer with data inputs A, B, C, D on `0`–`3`, selects $S_1 S_0$, output X.

[45] courses/csse2010/csse2010-diagram-notation.pdf
[46] courses/csse2010/combinational-logic-blocks.pdf

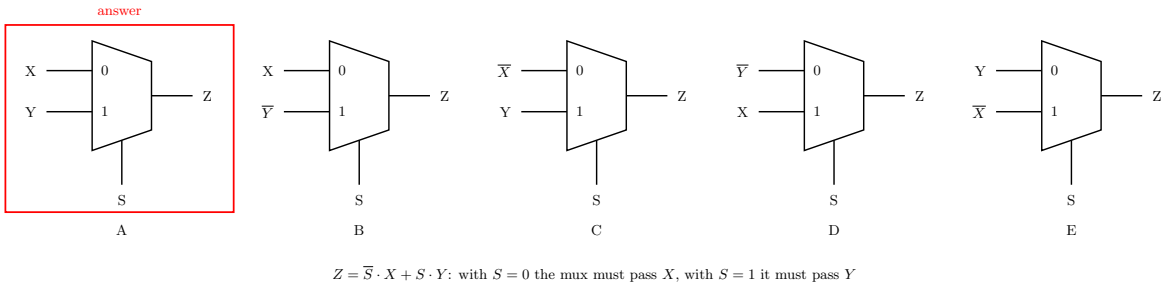
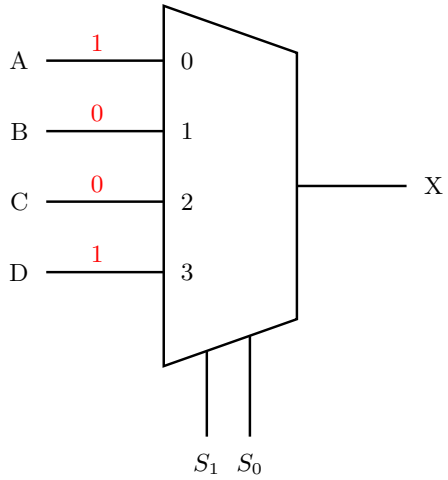


Figure 6: The five candidate multiplexers, lettered A to E. The data input labelled 0 is the one selected when S = 0, so the expression S-bar.X + S.Y is read straight off: X on input 0, Y on input 1, and nothing complemented. Option A is the only one that matches; every other option either swaps X and Y or complements a data input that the expression does not complement.



S_1	S_0	selects	$X = \overline{S_1 \oplus S_0}$
0	0	A	1
0	1	B	0
1	0	C	0
1	1	D	1

The select pair addresses the data input by its binary index, so the column of required output values *is* the list of constants to tie A, B, C, D to.

Figure 7: The 4-to-1 multiplexer with the answer written onto its data inputs in red, and the table that produces it. Using a mux as a function generator is mechanical: the select pair addresses each data input by its binary index, so tabulate the function against every select combination and the output column is literally the list of constants to tie the data inputs to. Here the function is the complement of XOR, so the constants are 1, 0, 0, 1. The 0/1 written at the end of a data input is the course's logic-diagram notation for tying that input to logic low or logic high (device-pinouts); on a real schematic they become the ground and Vcc symbols.

Answer: (d) $A = 1$, $B = 0$, $C = 0$, $D = 1$ — the XNOR column. This is the technique of using a mux to implement a logic function; see combinational-logic-blocks[47].

Circuit 1 — a 2-to-1 multiplexer from 2-input NAND gates only

> Draw a circuit schematic diagram for a 2-to-1 multiplexer that uses only 2-input NAND gates. The data inputs should be connected to push buttons. The select input should be connected to a switch. The output should be connected to an LED.
>
> Hint: do the truth table (or a function table) for a 2-to-1 multiplexer, obtain the logic expression, then draw a logic circuit and convert it to the NAND equivalent circuit by replacing each gate with its NAND version. If you use Logisim, **do not use the MUX module**.

1. Function table

S_0	D_0	D_1	F
0	0	0	0
0	0	1	0
0	1	0	1
0	1	1	1
1	0	0	0
1	0	1	1
1	1	0	0
1	1	1	1

2. Sum of products and simplification

Deriving the unsimplified sum of products from the true outputs in the table:

- $S_0 = 0$, $D_0 = 1$, $D_1 = 0 \rightarrow \bar{S_0} \cdot D_0 \cdot \bar{D_1}$
- $S_0 = 0$, $D_0 = 1$, $D_1 = 1 \rightarrow \bar{S_0} \cdot D_0 \cdot D_1$
- $S_0 = 1$, $D_0 = 0$, $D_1 = 1 \rightarrow S_0 \cdot \bar{D_0} \cdot D_1$
- $S_0 = 1$, $D_0 = 1$, $D_1 = 1 \rightarrow S_0 \cdot D_0 \cdot D_1$

[47] courses/csse2010/combinational-logic-blocks.pdf

Using Boolean algebra, we can simplify this expression:

$$\begin{aligned} F &= \bar{S_0} D_0 \bar{D_1} + \bar{S_0} D_0 D_1 + S_0 \bar{D_0} D_1 + S_0 D_0 D_1 \\ &= \bar{S_0} D_0 (\bar{D_1} + D_1) + S_0 D_1 (\bar{D_0} + D_0) \\ &= \bar{S_0} D_0 (1) + S_0 D_1 (1) \\ &= \bar{S_0} D_0 + S_0 D_1 \end{aligned}$$

3. NAND equivalent conversion

To implement this circuit using *only* 2-input NAND gates, convert the simplified AND-OR expression into a NAND-NAND expression by applying double negation and De Morgan's laws:

$$\begin{aligned} F &= \overline{\overline{\bar{S_0} D_0 + S_0 D_1}} \\ &= \overline{(\overline{\bar{S_0} D_0}) \cdot (\overline{S_0 D_1})} \end{aligned}$$

4. Circuit schematic

Exactly **four** 2-input NAND gates are needed, so the whole circuit fits on one 74HCT00. Using circuit-schematics[48] labelling conventions they become `U1:A`–`U1:D`:

1. One NAND acting as an inverter for S_0 (both inputs tied to S_0) to make $\bar{S_0}$.
2. One NAND computing $\overline{\bar{S_0} D_0}$.
3. One NAND computing $\overline{S_0 D_1}$.
4. One final NAND combining the outputs of 2 and 3.

Circuit 2 — a 2-to-4 decoder

> Draw a circuit schematic diagram for a 2-to-4 decoder, using whichever logic gates you prefer (from those available in the CSSE2010/CSSE7201 lab). The two inputs should be connected to switches. The outputs should be connected to LEDs.

The truth table (inputs A_1, A_0; outputs $X_0 \ldots X_3$):

A1	A0	X0	X1	X2	X3
0	0	1	0	0	0

[48] courses/csse2010/circuit-schematics.pdf

A1	A0	X0	X1	X2	X3
0	1	0	1	0	0
1	0	0	0	1	0
1	1	0	0	0	1

1. Boolean expressions

Since A_1, A_0 can only take four combinations, each output's truth-table row is already a single minterm — no Boolean simplification is needed, just one 2-input AND term per output:

- $X_0 = \bar{A}_1 \cdot \bar{A}_0$
- $X_1 = \bar{A}_1 \cdot A_0$
- $X_2 = A_1 \cdot \bar{A}_0$
- $X_3 = A_1 \cdot A_0$

2. Circuit schematic

Because every term is exactly two literals, this is the only one of the three circuits that needs no gate-building tricks at all: a 74HCT04 to make the two complements once each, and a 74HCT08 for the four products.

Circuit 3 — a 4-bit two's complement adder/subtractor

> Using a 74HCT04, 74HCT157 and a 74HCT283, draw a circuit schematic diagram for a circuit which can act as a 4-bit two's complement adder or subtractor. The output (4 bits from the 74HCT283, shown on LEDs) will have the value $A + B$ or $A - B$, where A and B are the 4-bit inputs taken from switches.
>
> A push button input (M, mode) determines whether the circuit adds or subtracts — if the value is 0, addition is performed; if the value is 1, subtraction is performed by calculating $A + (-B)$.
>
> The circuit is similar to the adder-subtractor in **Lecture 4**, except that the selection of the B input bits as B or $\overline{B}$ is to be performed using **multiplexers instead of XOR gates**. (The 74HCT157 is a quad 2-to-1 multiplexer with a shared select input.) The carry output should be shown on an LED also.

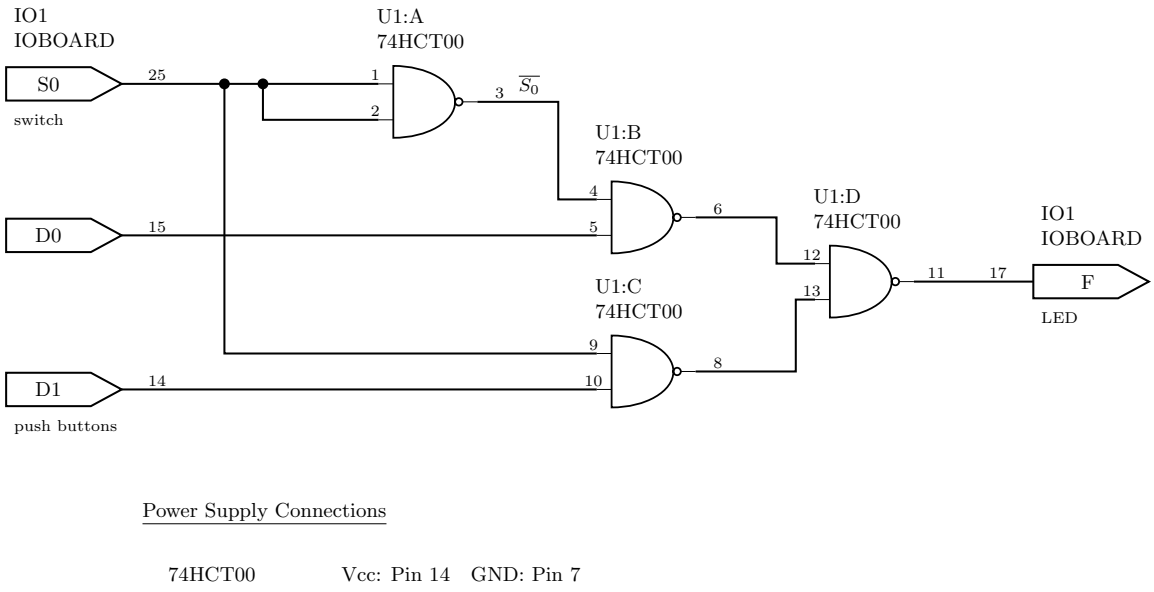
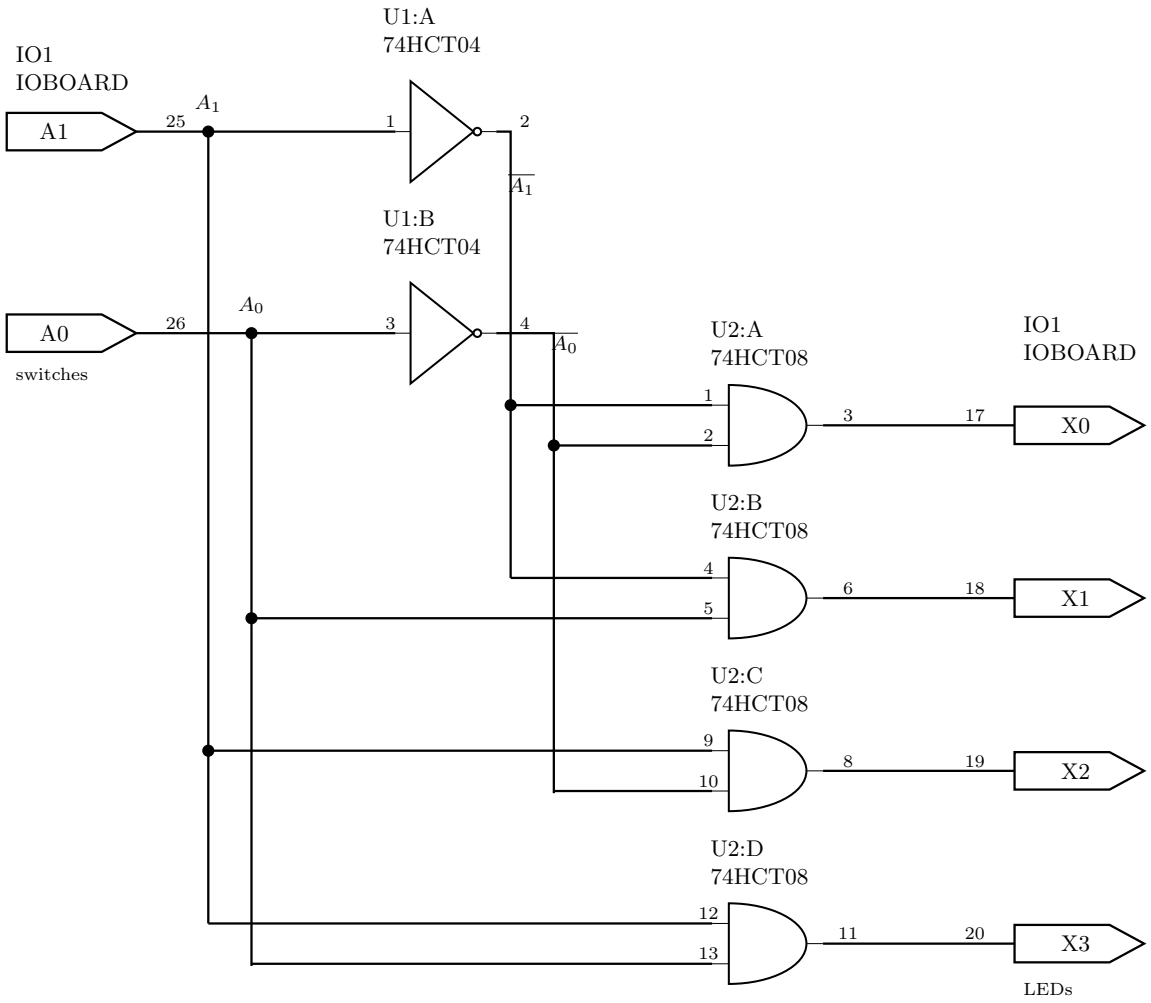


Figure 8: Circuit 1 as a schematic. The one thing to look at is U1:A: its two inputs are tied together by the wire carrying the junction dot, which is what turns a NAND into an inverter and is the move that keeps the whole design inside one package. Note also that S0 fans out twice - to U1:A and to U1:C - so the switch wire carries a second dot. The four gates use the 74HCT00's pin sets 1,2 to 3; 4,5 to 6; 9,10 to 8; 12,13 to 11, and one power block covers the single chip type. The task only says switch, push buttons and LED, so the pin assignments for the switch, the two buttons and the LED are chosen from the IO board list in device-pinouts.



Power Supply Connections

74HCT04 74HCT08	}	Vcc: Pin 14 GND: Pin 7

Figure 9: Circuit 2 as a schematic. Four vertical rails carry A1, A0 and their complements past all four AND gates, and each gate taps the one pair of rails that spells its minterm - the same rail idiom as the 3-to-8 decoder. Only the dots are connections: every other place a tap crosses a rail it is two wires passing. Exactly one AND gate can be high at a time, which is the decoder's defining property and the thing to check when testing. The 74HCT04's second half runs backwards (pin 9 in, pin 8 out), but this circuit only needs the first two inverters, so it is not exercised here.

The multiplexer IC in the kit, and its active-low ENABLE

Pin 15 of the 74HCT157 is its active-low ENABLE input. Two questions about it:

- **What does active low mean?** The pin performs its function when it is driven **low**, not high. That is why the device sheet writes the name overlined, $\overline{\text{ENABLE}}$, and draws a bubble on that pin in the schematic symbol — the bubble is the notation for "inverted here" (see logic-gates[49]).
- **What should be connected to pin 15 for the IC to work? Logic 0 — tie it to GND.** Left floating it is undriven and can read as either level at random, so the multiplexer intermittently stops passing its inputs through. This is the same failure mode the flip-flop lab returns to; see week3-lab-logisim-and-flip-flops[50].

Design reasoning

In two's complement, $-B = \bar{B} + 1$, so subtraction can be built from the same adder used for addition: $A - B = A + \bar{B} + 1$. This means mode input M needs to do two things at once:

- Select whether the adder's B input is B (add) or $\bar{B}$ (subtract) — done with the 74HCT157.
- Inject the "+1" that turns one's complement into two's complement — done by wiring M straight into the adder's carry-in, since $C_0 = 0$ for addition (no extra bit needed) and $C_0 = 1$ for subtraction (adds the required 1).

So M is wired to **both** the 74HCT157's select line and the 74HCT283's C_0 pin — no separate control logic at all. That single fan-out is the whole idea of the circuit; everything else is bit-plumbing. Compare the XOR version of the same trick in combinational-logic-blocks[51] — the multiplexer is doing exactly the job the controlled inverter did there, and the mode line still doubles as the carry-in either way.

Note also what the 74HCT157 being a **quad** mux with a **shared** select buys you: one chip conditions all four bits of B together, which is what you want, because a per-bit select would let the operand be half-complemented.

> **Select polarity.** Wiring M straight to `SEL` (rather than through an inverter) assumes `SEL` **low** selects the **A** channel and `SEL` high selects **B**. That is the 74HCT157's actual behaviour, but it comes from the part's datasheet — the *Device Pinouts and Symbols* sheet (device-pinouts[52]) names both channels and the select line without saying which level picks which. So put the raw B switches on the `A` inputs and the inverted ones on `B`, and check the polarity on the bench before you trust it: if the modes come out swapped, that one assumption is why.

[49] courses/csse2010/logic-gates.pdf
[50] courses/csse2010/week3-lab-logisim-and-flip-flops.pdf
[51] courses/csse2010/combinational-logic-blocks.pdf
[52] courses/csse2010/device-pinouts.pdf

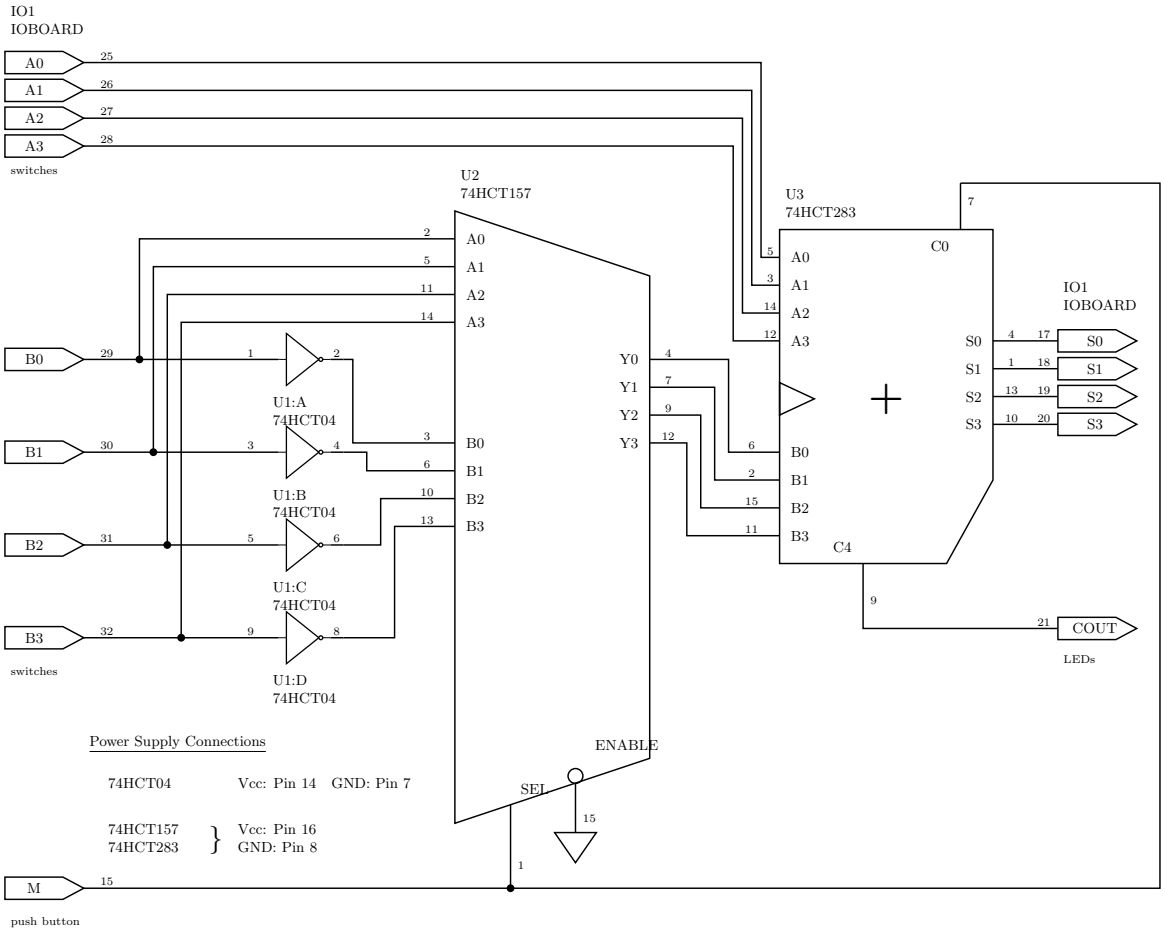


Figure 10: Circuit 3 as a schematic. Follow the mode line M along the bottom: it forks once, at the junction dot, into the 74HCT157's SEL (pin 1) and the 74HCT283's carry-in C0 (pin 7) - one push button doing both halves of A + not(B) + 1. Channel A of the multiplexer takes the raw B switches and channel B takes them inverted by the 74HCT04, so SEL = 0 passes B and SEL = 1 passes B-bar. ENABLE (pin 15) is tied to ground because it is active low. The power block is split in two because the 74HCT04 is a 14-pin part while the 157 and the 283 are 16-pin. The task only names the three chips and the input and output devices, so the pin assignments for the switches, button and LEDs are chosen from the IO board list in device-pinouts.

The wiring, chip by chip

- `U1` (74HCT04, hex inverter) — generates $\bar{B}_0$–$\bar{B}_3$ for the multiplexer's B channel:
  - `U1:A` (1→2): $B_0 \to \bar{B}_0$
  - `U1:B` (3→4): $B_1 \to \bar{B}_1$
  - `U1:C` (5→6): $B_2 \to \bar{B}_2$
  - `U1:D` (9→8): $B_3 \to \bar{B}_3$ — note the direction reversal in the second half of the package
- `U2` (74HCT157, quad 2-to-1 MUX) — selects B_i (channel A, $M = 0$) or $\bar{B}_i$ (channel B, $M = 1$) per bit. `SEL` (pin 1) = M; `ENABLE` (pin 15) tied to GND:
  - `A0`–`A3` = pins 2, 5, 11, 14 = the B switches
  - `B0`–`B3` = pins 3, 6, 10, 13 = `U1`'s inverted outputs
  - `Y0`–`Y3` = pins 4, 7, 9, 12 → the adder's B inputs
- `U3` (74HCT283, 4-bit adder):
  - `A0`–`A3` (pins 5, 3, 14, 12) = the A switches (unchanged by mode)
  - `B0`–`B3` (pins 6, 2, 15, 11) = `U2`'s `Y0`–`Y3`
  - `C0` (pin 7, carry-in) = M (the same signal as `U2`'s `SEL`)
  - `S0`–`S3` (pins 4, 1, 13, 10) → LEDs
  - `C4` (pin 9, carry-out) → LED

Power supply connections are shown once per chip **type**, and this circuit has two different pin pairs: the `74HCT04` is a 14-pin package (Vcc 14, GND 7), while the `74HCT157` and `74HCT283` are 16-pin (Vcc 16, GND 8). Do not group all three together.

In the session

The session is spent working on the three schematics above:

- Complete the preparation tasks, and make sure the three schematics are **correctly drawn**, not just roughly right.
- **Exchange your solutions with others at your table for peer evaluation.**
- Get a tutor to verify your schematic diagrams.
- **Construct the Circuit 3 (adder/subtractor) circuit on the breadboard and verify the functionality.** Logisim simulation is allowed as well.

Construction advice

- Keep circuits neat — **use the shortest wires you can**.
- **Turn the power off before you change the circuit.**

- Debugging: use the **logic probe** (the `PR` pin on the IO board — pin 16, see io-board[53]), and be systematic: **check power and ground connections first**, then start at the inputs and work towards the outputs.

For each circuit

1. Make sure you have a correct schematic **before** you build.
2. Decide your test procedure — **what truth table are you expecting?**
3. Build or simulate.
4. Test systematically.

   Don't expect things to always work in the first instance. If things don't work, troubleshoot systematically. Troubleshooting is also a part of your learning. **A neatly drawn circuit schematic will always help when troubleshooting.**

Stages of completion for the build: logic diagram only / schematic only / simulation only / attempted construction with incorrect output / fully working circuit with functionality verified.

Next

Lab 5 is the Logisim and flip-flops session, and the one where kits are loaned out — see week3-lab-logisim-and-flip-flops[54].

Logisim and Flip-Flops

Learning Lab 5 — verify the operation of latches and the D flip-flop in Logisim, then build one. It is also the session where **kits are loaned out**. Pairs with 2026-08-10-introduction-to-sequential-circuits[55] and flip-flops-and-latches[56].

Recall

There are two types of logic circuit: **combinational** and **sequential**. The basic building block of sequential circuits is the **flip-flop**, which stores 1 bit of information. See flip-flops-and-latches[57] and sequential-circuits[58].

[53] courses/csse2010/io-board.pdf
[54] courses/csse2010/week3-lab-logisim-and-flip-flops.pdf
[55] courses/csse2010/2026-08-10-introduction-to-sequential-circuits.pdf
[56] courses/csse2010/flip-flops-and-latches.pdf
[57] courses/csse2010/flip-flops-and-latches.pdf
[58] courses/csse2010/sequential-circuits.pdf

Worked: which is NOT a combinational circuit?

A counter. Adders, multiplexers, decoders and adder/subtractors are all combinational — their output is a function of the current inputs alone, and every one of them appeared in Lab 4 (see week3-lab-lab-4-pre-work[59]). A counter has to *remember* where it is, so it is sequential. See [counters].

Worked: which waveform captures the behaviour of this circuit?

The circuit is a **rising-edge-triggered** D flip-flop (a wedge on the clock input with **no** bubble in front of it) with A on D, CLK on the clock, and Q taken out. `SET` and `CLR` are lettered inside the box but not driven.

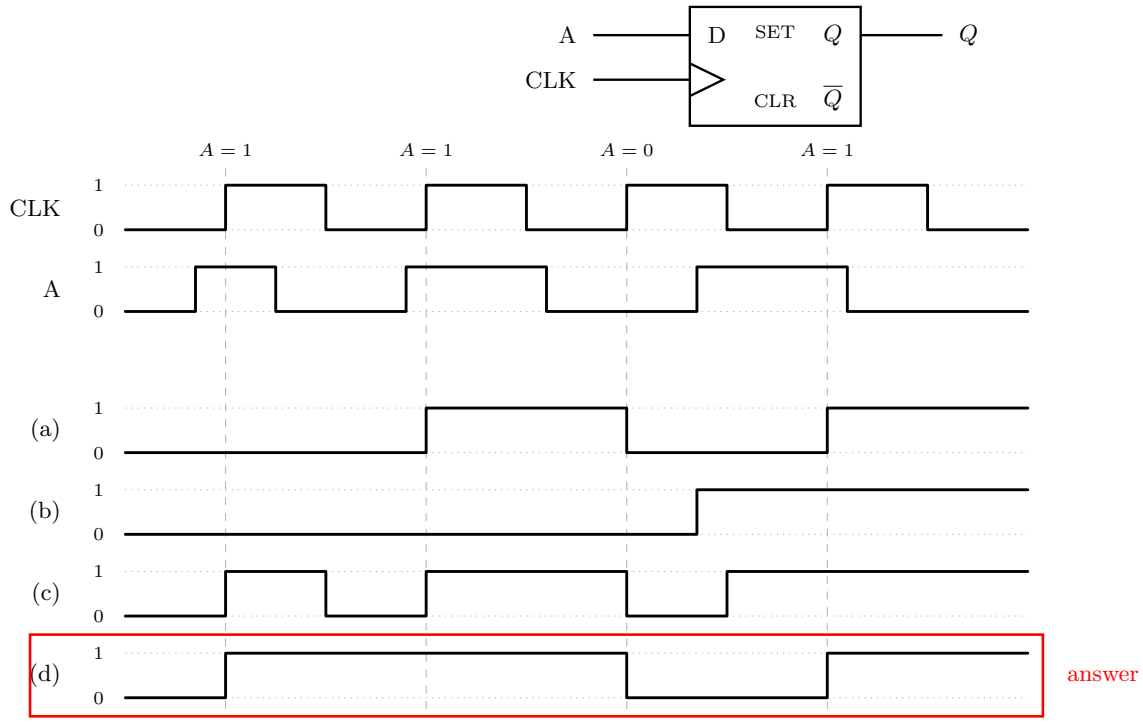


Figure 11: The circuit, its two input waveforms CLK and A, and the four candidate outputs. The answer is (d). The dashed verticals through the four rising clock edges, and the A = 1/1/0/1 labels above them, are what makes the answer mechanical, since a rising-edge flip-flop copies D to Q at those four instants and nowhere else.

[59] courses/csse2010/week3-lab-lab-4-pre-work.pdf

Answer: (d). Sample A at each rising edge of CLK and nowhere else: the values are 1, 1, 0, 1, so Q goes high at the first edge, holds through the second, drops at the third and rises again at the fourth. Between edges Q does nothing at all, however much A moves.

The three wrong waveforms each fail in a different, recognisable way:

- **(a)** agrees with (d) from the second clock edge onwards but misses the first. A is already high when the first edge arrives, so the flip-flop does capture it — this is the answer you get by mentally starting one edge late.
- **(b)** changes exactly once, at the moment A last rises. That is a wire, not a flip-flop: no clock edge is involved.
- **(c)** changes at instants where the clock is **falling** as well as where it is rising, which no single-edge-triggered device does.

Worked: which of these is FALSE about a negative-edge-triggered D flip-flop?

"Changes on the D input affect the output when the clock signal is at logic 0."

That describes a *level*-triggered latch — the thing an edge-triggered flip-flop is defined in contrast to. An edge-triggered device responds only at the transition; for a negative-edge part, when the clock goes $1 \to 0$. The other three statements (stores 1 bit, is a sequential circuit, changes when the clock goes from 1 to 0) are all true.

Contrast the previous question: the symbol there had a bare wedge (rising edge), and this question is about a bubbled wedge (falling edge). The wedge and the bubble read independently — the wedge says edge-triggered, the bubble says inverted; see the symbol table in flip-flops-and-latches[60].

Floating pins

> If a pin isn't being driven it is **floating**, and a floating pin can act as 0 or 1 at random.

The question is whether it matters that a pin is left disconnected, in three cases on one specific wired-up `74HCT74` (pinout in device-pinouts[61]). The answers differ sharply.

Reading the figure: the circuit uses **only the first flip-flop**. The second one (pins 8–13) is wired to nothing at all, and of the first flip-flop's six pins only three are driven or read.

[60] courses/csse2010/flip-flops-and-latches.pdf
[61] courses/csse2010/device-pinouts.pdf

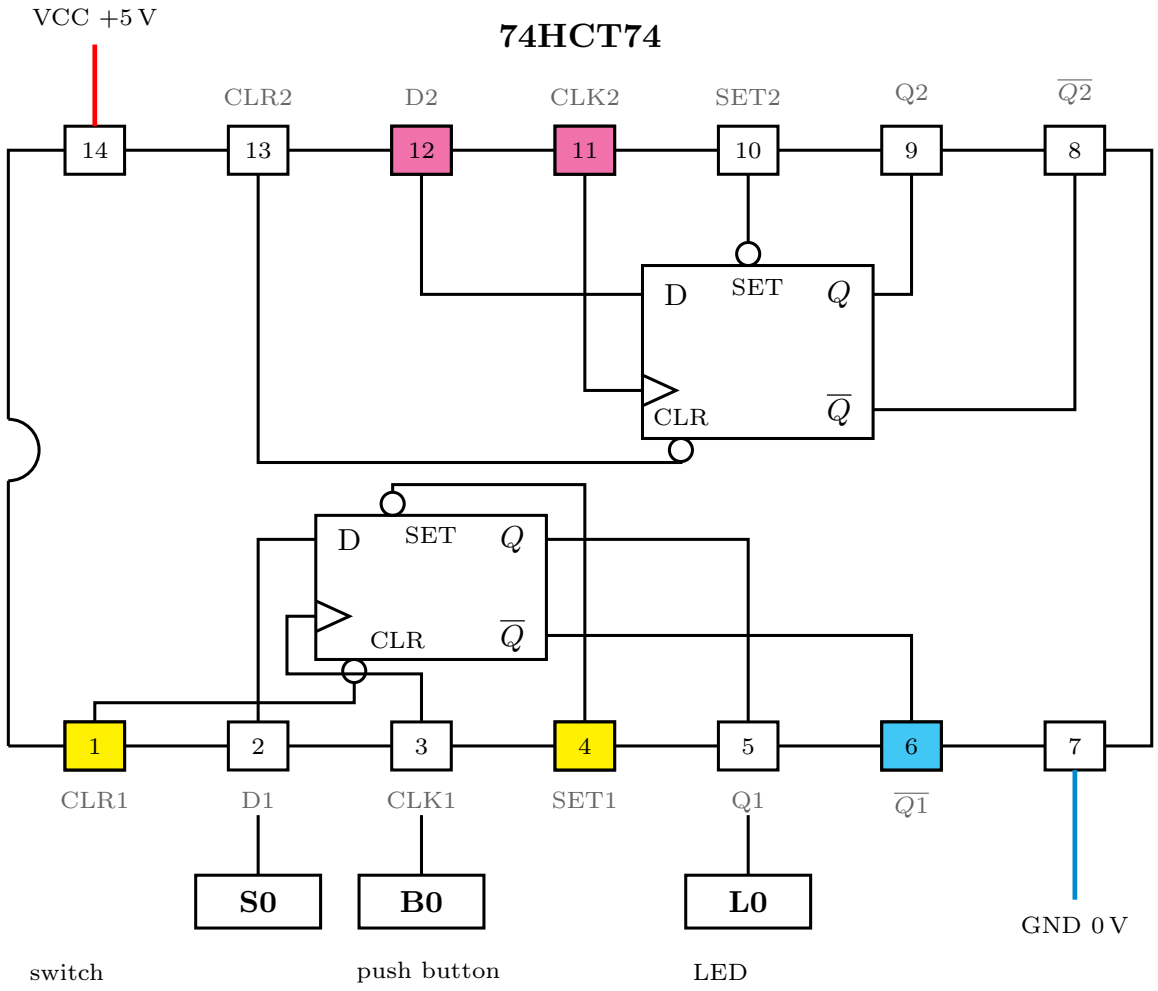


Figure 12: The wired-up 74HCT74: the 14-pin package with both flip-flops inside, the three highlighted pin groups, and the only three external connections the circuit actually makes. The fill colours mark the three cases, one colour each - magenta for pins 11 and 12, yellow for pins 1 and 4, cyan for pin 6 - and mean nothing beyond that grouping; red and blue mark the supply stubs. The three connections are switch S0 to D1 (pin 2), push button B0 to CLK1 (pin 3) and LED L0 to Q1 (pin 5). The signal names outside each pin box are from the device sheet's pin table (device-pinouts).

Disconnected	Matters?
Pins 11 and 12 — `CLK2` and `D2`, the clock and data of the *second, entirely unused* flip-flop	**No.** That half's outputs (pins 8 and 9) go nowhere, so whatever its flip-flop does is invisible. Untidy rather than wrong; good practice is to tie unused inputs to a fixed level so the chip isn't switching at random and drawing current.
Pin 6 — $\overline{Q1}$, the complement output of the flip-flop you *are* using	**No.** It is an **output**. Leaving an output unconnected is always safe; it is only ever a problem to leave an *input* undriven.
Pins 1 and 4 — `CLR1` and `SET1` of the flip-flop you are using	**Yes, badly.** These are active-low asynchronous inputs. Floating, they can assert at random and clear or set the flip-flop behind your back, so Q changes with no clock edge in sight. They must be tied **high**.

The distinction to take away: a floating **input on a device you are using** is a real fault; a floating **output** is harmless; a floating input on a section you are not using is merely untidy.

> **A naming clash worth knowing about.** The device sheet calls pins 4 and 1 `SET` and `CLR`; Task 3 below calls the same two pins `PRE` and `CLR`; flip-flops-and-latches[62] writes them $\overline{\text{PRE}}$ and $\overline{\text{CLR}}$. The course is inconsistent about this (see csse2010-diagram-notation[63]). The pinout figure uses the device sheet's names, and the Task 3 section below uses `PRE`/`CLR` for the two push buttons while the chip symbol inside its schematic keeps the device sheet's `SET`/`CLR` on pins 4 and 1. Same two pins, two names, one figure.

The same trap comes back in Lab 4 as the 74HCT157's active-low `ENABLE`, which must be tied low for the chip to work at all — see week3-lab-lab-4-pre-work[64] — and again in Lab 7, see week4-lab-lab-7-exercises[65].

Tasks 1 and 2 — simulate the S-R latches

1. Simulate the **active-high** S-R latch (cross-coupled **NOR** gates) in Logisim and verify its operation.
2. Simulate the **active-low** S-R latch (cross-coupled **NAND** gates) and verify its operation.

[62] courses/csse2010/flip-flops-and-latches.pdf
[63] courses/csse2010/csse2010-diagram-notation.pdf
[64] courses/csse2010/week3-lab-lab-4-pre-work.pdf
[65] courses/csse2010/week4-lab-lab-7-exercises.pdf

Both circuits, both truth tables, and why the NAND version's set/reset roles invert and its inputs are drawn overlined, are in flip-flops-and-latches[66].

The thing to actually verify in the simulator is the part a truth table hides: the **hold** row, where both inputs are inactive and the latch keeps whatever it had. Drive it into set, return the inputs to their inactive values, and check the output stays put — that is the memory, and it is the only behaviour that distinguishes a latch from a gate.

Task 3 — build a `74HCT74` circuit

The circuit to implement is a 2-input **AND** gate fed by A and B, whose output drives the D input of a rising-edge-triggered D flip-flop.

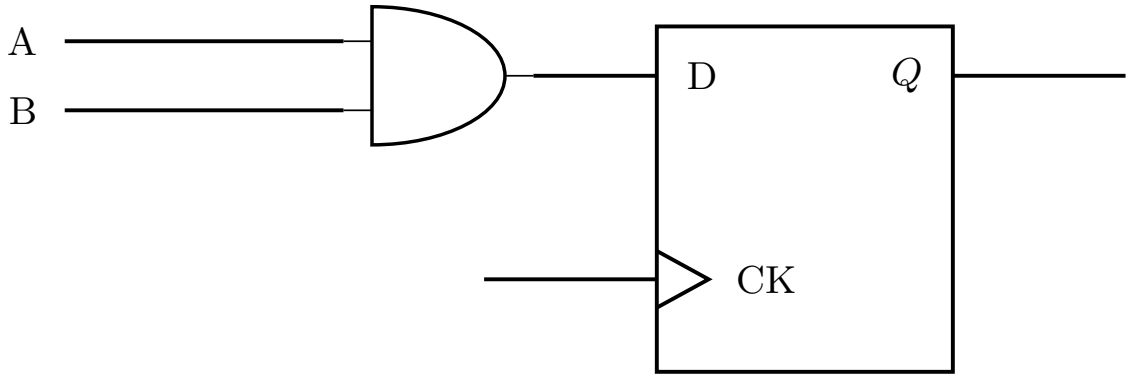


Figure 13: The Task 3 circuit - two switch inputs ANDed into D, one clock input on a wedge (so edge-triggered), Q out. The clock is labelled CK, which is one of the four spellings this course uses for that pin. Nothing else is drawn: the asynchronous inputs and Q-bar are left off, even though the task requires both to be wired.

So Q takes the value of $A \cdot B$ on each rising clock edge and holds it until the next one. Drawing the schematic means adding what this logic diagram leaves out. The task asks for a schematic that:

- uses a `74HCT74` (and other chips as needed) — the AND gate needs a second chip;
- uses **push buttons** for the CLK (CK), PRE and CLR inputs;
- uses **switches** for the A and B inputs;
- uses **LEDs** for the Q and $\overline{Q}$ outputs.

Have it checked by a tutor, then build and test it.

[66] courses/csse2010/flip-flops-and-latches.pdf

Why put inverters between the buttons and the PRE/CLR inputs?

Because $\overline{\text{PRE}}$ and $\overline{\text{CLR}}$ are **active low**, and a push button is the wrong shape for an active-low input. What the flip-flop needs is a pin that sits reliably **high** whenever you are not asking for a preset or clear, and goes low only while you press — otherwise the asynchronous input is asserted at rest and the clocked behaviour you are trying to test never happens. Putting an inverter between the button and the pin turns the button's natural "pressed = high" into the "pressed = low" the pin wants, and holds the pin high the rest of the time. (Asserting $\overline{\text{PRE}}$ and $\overline{\text{CLR}}$ at once is an illegal input; see flip-flops-and-latches[67].)

This is the same problem as the floating-pin question above. Pins 1 and 4 must be **held high** and only pulsed low, and neither leaving them floating nor wiring a bare button to them achieves that.

Test it the way Lab 4 asks you to test everything (see week3-lab-lab-4-pre-work[68]): decide the expected behaviour first. Here that is four rows of $A \cdot B$ crossed with "did a clock edge happen", plus two asynchronous checks — press PRE with $A \cdot B = 0$ and confirm Q goes high *without* a clock edge, then press CLR and confirm it goes low the same way.

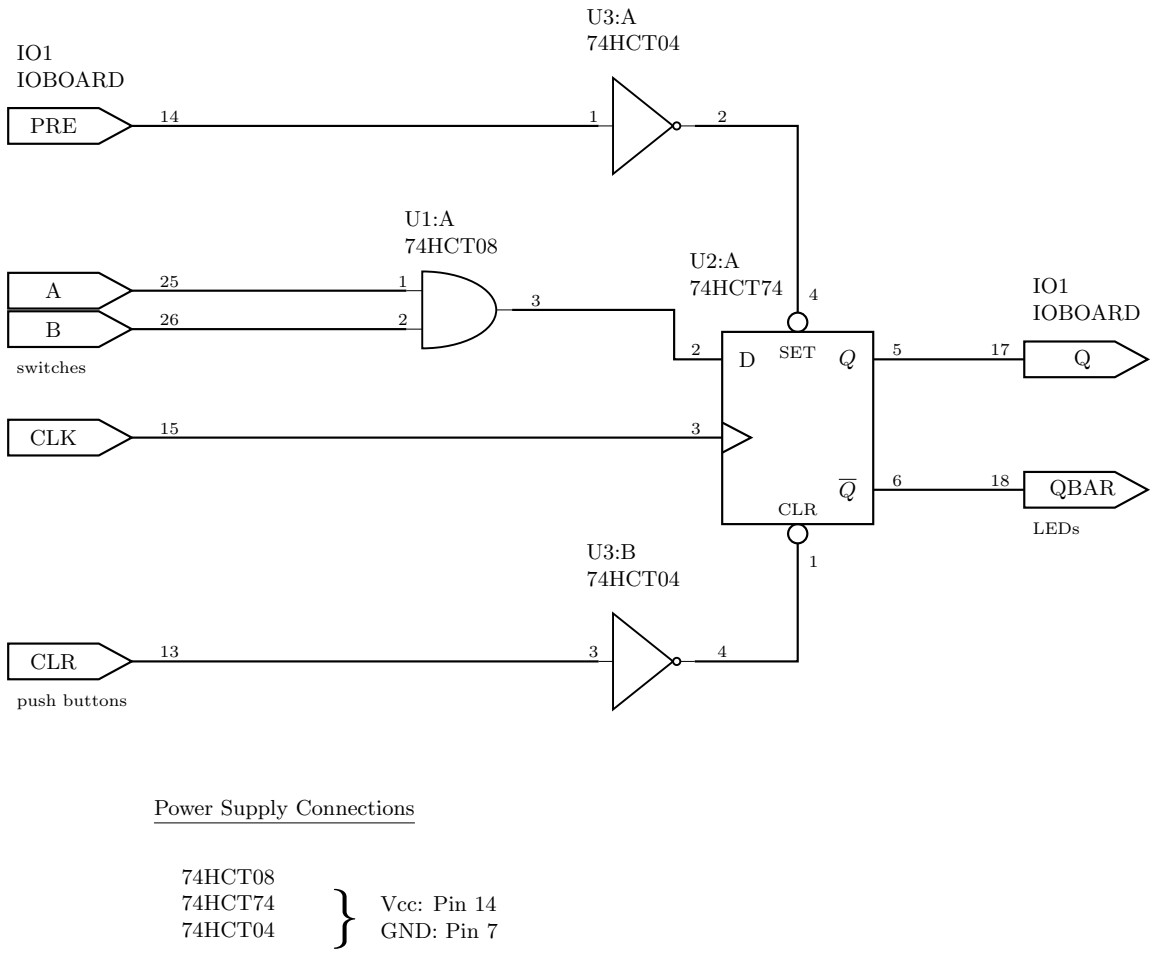
Kit borrowing

- Fill in, sign and submit the kit borrowing agreement (also on Blackboard), then collect your kit. **Wait at your table; don't queue.**
- Check all the parts are there and **write your student number on the kit**.
- **Don't use the kit until advised** — some parts break easily, especially the **USB connector on the I/O board** (see io-board[69]).
- You'll start with the logic chips and breadboard, and will be advised how to use them. The **AVR microcontroller is only needed in the second half** of the course — leave those parts alone until then.
- Drop the course after borrowing? Return the kit within **3 business days** to the EECS school office, building 78 room 425. (If you're thinking of dropping, don't borrow one.)
- The kit must be returned **by the end of semester in working order**.

Dongles

By the end of semester you'll be using **two USB-A cables** for data — one for programming, one for serial communication — and one of them from about halfway through. On your own laptop you may need a dongle. There are workarounds but they all have downsides, so **get one sooner rather than later**.

[67] courses/csse2010/flip-flops-and-latches.pdf
[68] courses/csse2010/week3-lab-lab-4-pre-work.pdf
[69] courses/csse2010/io-board.pdf



Power Supply Connections

74HCT08
74HCT74
74HCT04
} Vcc: Pin 14
GND: Pin 7

Figure 14: The Task 3 schematic. Three chips: U1 for the AND gate, U2 for the flip-flop, U3 for the two inverters that make the active-low PRE and CLR usable from push buttons. All three are 14-pin parts sharing Vcc 14 and GND 7, so one power block covers them. The pin assignments and the IO board pin numbers are from device-pinouts.

Next

Lab 6 has a **preparation task** — the combination lock, spelled out in 2026-08-13-shift-registers[70] — to be done before the session. See week4-lab-sequential-circuits-shift-registers[71].

Reference material

Binary Number Representations

Bits, bytes, and words

- **Bit** = binary digit (0 or 1).
- **Nibble** = 4 bits. It is the unit that makes hexadecimal convenient (one hex digit = one nibble).
- **Byte** = 8 bits, e.g. `01010111`.
- Modern computers deal with **words**, usually a power-of-2 number of bytes: 1, 2, 4, or 8 bytes = 8, 16, 32, 64 bits. The **AVR** used in this course is at the small end — an **8-bit** (1-byte) word; **modern CPUs** sit at the other end, 32- or 64-bit.

Representing whole (unsigned) numbers

Each bit position has a value — a power of 2, increasing from right (least significant) to left (most significant). **Binary → decimal**: add the values of each position where the bit is 1.

Most and least significant bits

Converting decimal to binary

Two equivalent methods, both worked on the same number, 53.

Method 1 — sum of powers of 2. Rewrite n as a sum of powers of 2, by repeatedly subtracting the largest power of 2 not greater than n ($53 - 32 = 21$, $21 - 16 = 5$, $5 - 4 = 1$, $1 - 1 = 0$). Then assemble the binary number from 1's in the bit positions corresponding to those powers of 2, and 0's everywhere else.

Method 2 — repeated division. Build the bits up from the right (least significant): divide n by 2, the remainder (0 or 1) is the next bit, then repeat with n = the quotient. Stop when the quotient reaches 0, and read the remainders **upwards**.

[70] courses/csse2010/2026-08-13-shift-registers.pdf
[71] courses/csse2010/week4-lab-sequential-circuits-shift-registers.pdf

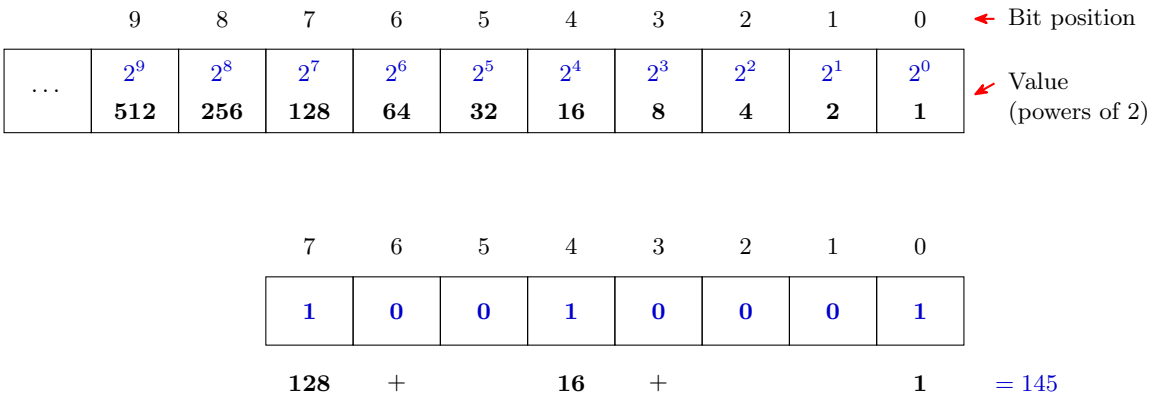


Figure 15: Place values for an unsigned binary number, and a worked conversion. Position i is worth 2^i, so reading a binary number is just adding the place values of the positions holding a 1: $10010001_2 = 128 + 16 + 1 = 145$.

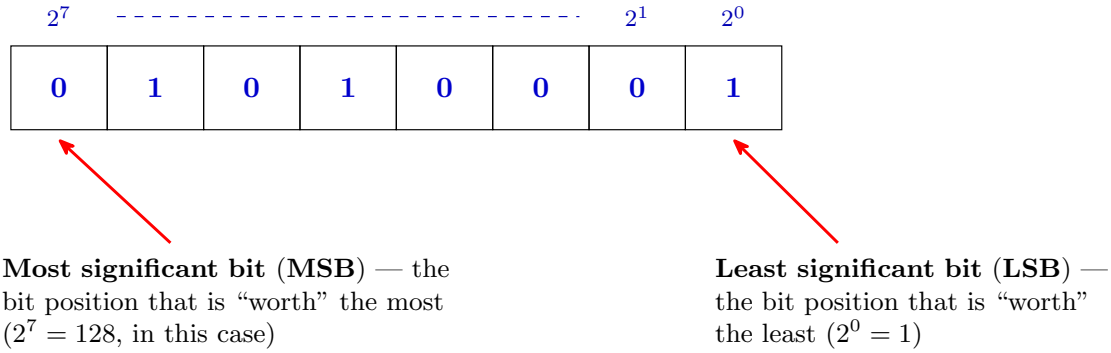


Figure 16: The two ends of a word get names, because almost every later convention (shifting, sign bits, bit numbering in AVR registers) is stated in terms of them. For an n-bit unsigned word the MSB is worth 2^{n-1} — here $2^7 = 128$ — and the LSB is always worth $2^0 = 1$.

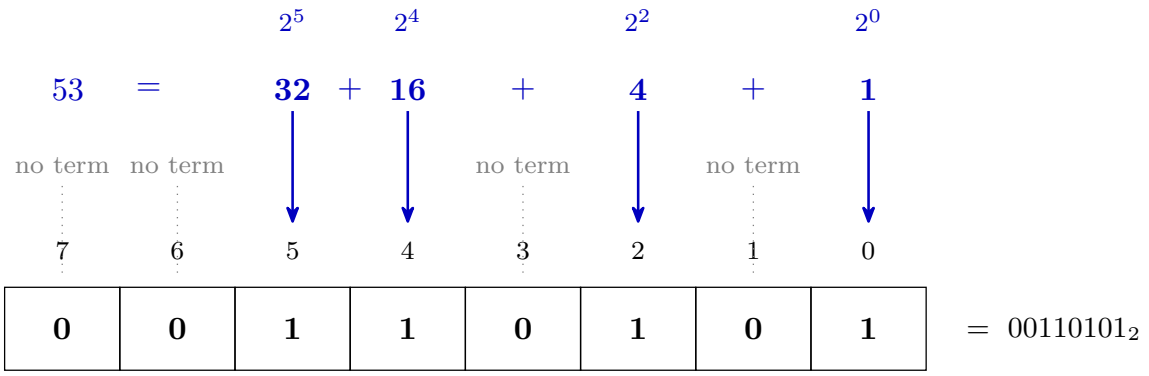


Figure 17: Method 1 on 53. Each power of two in the decomposition names one bit position — 2^5 is position 5, 2^4 is position 4, and so on — and every position with no term in the sum gets a 0. Reading the row off left to right gives 00110101_2.

Number range (unsigned)

- Smallest representable value: all 0's $\rightarrow$ 0.
- Largest representable value: all 1's $\rightarrow$ for an n-bit word, $2^n - 1$ (e.g. 255 for 8 bits).

The -1 is the part worth understanding: an n-bit word has 2^n distinct patterns, but one of them is spent on zero.

Other radices

Radix = number system base. A radix-k number system has k distinct symbols for digits 0 to $k-1$, and the value of each digit (from the right) is $k^0, k^1, k^2, \ldots$ — so a radix-k number $d_m \ldots d_1 d_0$ is worth $\sum_i d_i k^i$. E.g. $1\text{A}_{16} = 1 \cdot 16^1 + 10 \cdot 16^0 = 16 + 10 = 26$.

- **Octal** (radix-8): symbols `0`–`7`. One octal digit corresponds to exactly 3 bits.
- **Hexadecimal** (radix-16): symbols `0`–`9`, `A`–`F`. One hex digit corresponds to exactly 4 bits — very convenient for grouping binary.

Dec	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17
Oct	0	1	2	3	4	5	6	7	10	11	12	13	14	15	16	17	20	21
Hex	0	1	2	3	4	5	6	7	8	9	A	B	C	D	E	F	10	11

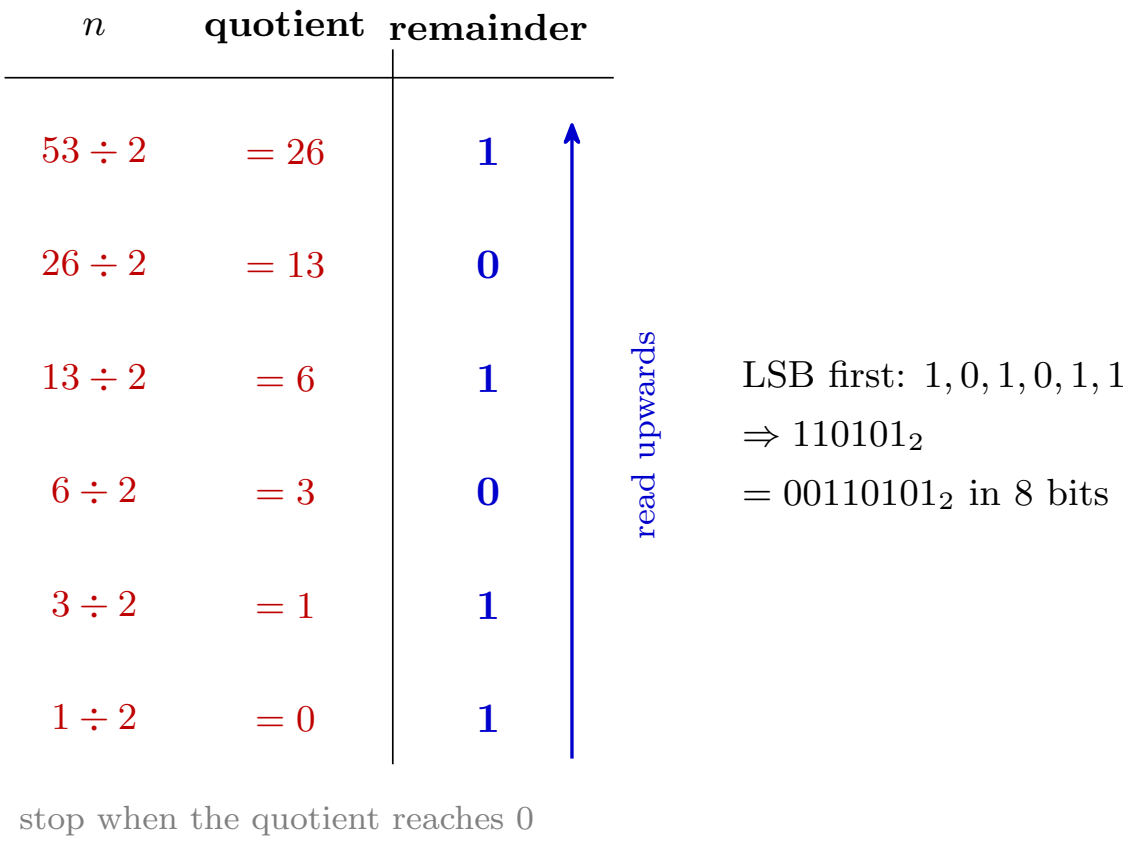


Figure 18: Method 2 on 53. The remainders come out LSB-first, which is why the answer has to be read back up the column: 110101_2, or 00110101_2 padded to a byte. The same procedure with a divisor of 16 or 8 converts to hexadecimal or octal instead.

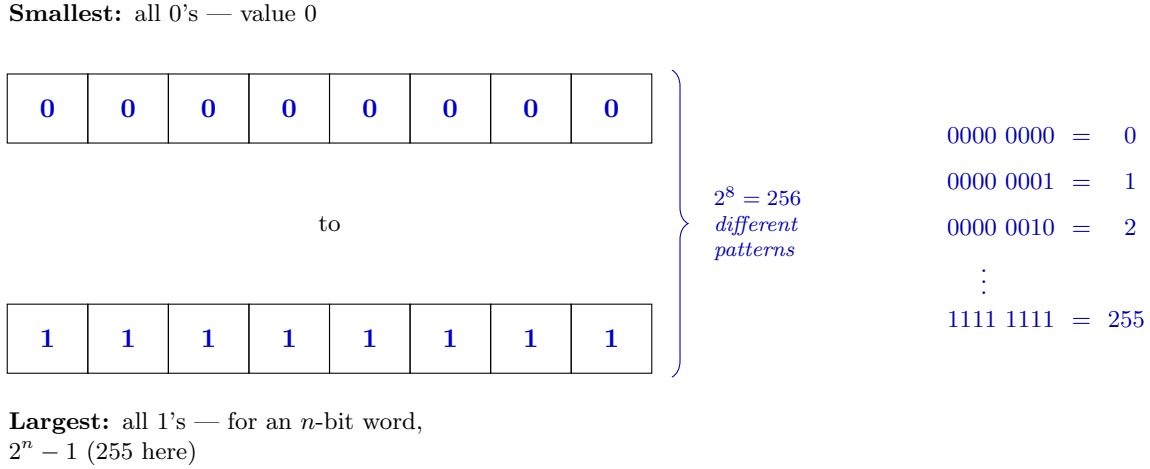


Figure 19: Why the largest unsigned value is $2^n - 1$ and not 2^n. Counting up from all-zeros steps through every pattern in turn — each extra bit doubles how many there are, so 8 bits give $2^8 = 256$ patterns — and because the count starts at 0, the last one is 255.

Converting binary ⇄ hex/octal by grouping

Because $16 = 2^4$ and $8 = 2^3$, no arithmetic is needed: **group the bits from the right** — four at a time for hex, three at a time for octal — and translate each group with the table above. Pad the leftmost group with leading zeros.

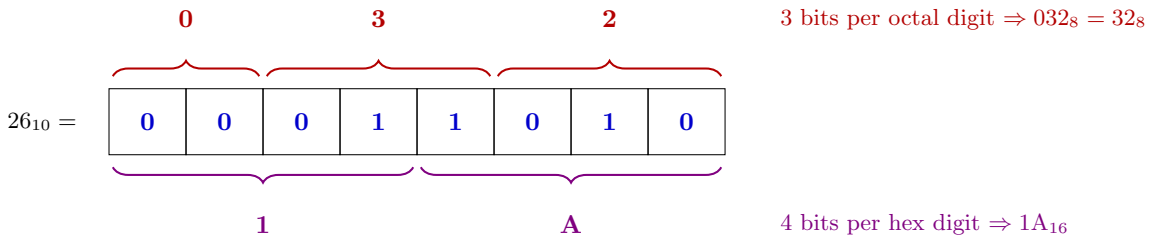


Figure 20: One bit pattern read two ways. Grouping 00011010_2 into nibbles from the right gives 1 and A, i.e. $1A_{16}$; grouping the same bits in threes gives 0, 3 and 2, i.e. 32_8. The grouping must start at the right — starting at the left gives the wrong answer whenever the width is not a multiple of 3 or 4.

The same trick read backwards is why hex is used for bit patterns at all: a full byte of 1's, `1111 1111`, is `F F` = `0xFF` = 255, and any byte can be written as two hex digits.

To convert **decimal → any radix**, use Method 2 with that radix as the divisor: $26 \div 16$

gives quotient 1 remainder 10 (= `A`), so $26_{10} = 1A_{16}$; $26 \div 8$ gives quotient 3 remainder 2, so $26_{10} = 32_8$. See 2026-07-27-course-intro-bits-bytes-binary[72] for that worked exercise.

Radix notation conventions

Since a bare number like `101` or `747` is ambiguous, a radix indicator is needed. A subscript works generically (e.g. 101_2, 101_{16}); in code/assembly, conventions vary. **Bolded rows are the ones this course mostly uses**:

Radix	Convention	Example	Where used
any	**subscript**	101_2, 101_{16}	**generic/mathematical**
Hex	**leading `0x`**	`0x101`	**C, Atmel AVR**
Hex	trailing `h`	`101h`	some assembly languages
Hex	**leading `$`**	`$747`	**Atmel AVR assembly**
Octal	**leading `0`**	`0101`	**C, Atmel AVR**
Octal	trailing `q`	`101q`	some assembly languages
Octal	leading `@`	`@747`	some assembly languages
Binary	**leading `0b`**	`0b101`	**Atmel AVR assembly, some C**
Binary	trailing `b`	`101b`	some assembly languages
Binary	leading `%`	`%101`	some assembly languages

The octal convention is the trap: in C, a leading `0` silently changes the radix, so `0101` is 65_{10}, not 101_{10}.

Representing signed numbers

Computers don't store + and − signs; the sign has to be encoded in the bits themselves. Four formats have been used, and **all four — plus unsigned — represent positive numbers identically.** Only the negatives differ.

Signed magnitude

The leftmost bit is a **sign bit** (0 positive, 1 negative) and **contributes nothing to the magnitude**. Negate by inverting the sign bit alone.

Ones' complement

The sign bit now also carries weight: its position value is $-(2^{N-1} - 1)$. **Negate by inverting all the bits.**

[72] courses/csse2010/2026-07-27-course-intro-bits-bytes-binary.pdf

Two's complement

The sign bit's position value is -2^{N-1}. **Negate by inverting all the bits and adding 1.** This is the one actually used in practice.

Compare the position values, 8-bit:

	bit 7	6	5	4	3	2	1	0
Signed magnitude	sign	64	32	16	8	4	2	1
Ones' complement	−127	64	32	16	8	4	2	1
Two's complement	−128	64	32	16	8	4	2	1

Excess-2^{N-1}

For 8 bits, "excess 128": the value stored is the true value **plus 128**. So −3 is stored as 125 and 3 as 131.

It is exactly **two's complement with the sign bit reversed** — a 0 in the MSB means bit value −128, a 1 means 0. Its practical use is that comparing two signed numbers becomes simple in hardware. The exponent of an IEEE floating-point number is stored in excess notation for exactly that reason, with a bias one smaller: excess-127 for 8 bits (floating-point-numbers[73]).

Ranges

Format	8-bit	n-bit
Unsigned	0 to 255	0 to $2^n - 1$
Signed magnitude	−127 to 127	$-(2^{n-1} - 1)$ to $2^{n-1} - 1$
Ones' complement	−127 to 127	$-(2^{n-1} - 1)$ to $2^{n-1} - 1$
Two's complement	−128 to 127	-2^{n-1} to $2^{n-1} - 1$
Excess-2^{n-1}	−128 to 127	-2^{n-1} to $2^{n-1} - 1$

Worked example — ±47 in 8 bits

+47 = `00101111` in every format.

Format	−47	How
Signed magnitude	`10101111`	flip the sign bit
Ones' complement	`11010000`	invert all bits

[73] courses/csse2010/floating-point-numbers.pdf

Format	−47	How
Two's complement	`11010001`	invert all bits, add 1
Excess-128	`01010001`	$-47 + 128 = 81$

And in 4 bits, −3 is `1100` in ones' complement and `1101` in two's complement.

Why "complement"?

A thing and its complement together make a whole — an angle and its complement make a right angle, a set and its complement a universal set.

- **Ones' complement**: a number plus its negative gives a string of ones (`11111111` in 8 bits).
- **Two's complement**: a number plus its negative gives a power of two, with overflow (`1 00000000`). It would more accurately be called *power-of-two's complement*.

Base 10 has the same pair — nines' and ten's complement — used on early mechanical adding machines to do subtraction.

Why two's complement wins

Two representations of zero, which wastes a bit pattern that could have extended the range:

- Signed magnitude: `00000000` and `10000000`
- Ones' complement: `00000000` and `11111111`

Asymmetric range: two's complement and excess-2^{n-1} (one more negative than positive).

Arithmetic: signed-magnitude addition requires taking cases; ones' complement addition requires an extra addition step (the end-around carry); sign extension and comparison are non-trivial in both. Two's complement has a single zero and its addition is *just* binary addition — no special cases.

Format matters

The same eight bits mean different things under different formats. `10010010` reads as:

Read as	Value
Unsigned	146
Two's complement	−110
Ones' complement	−109

Read as	Value
Signed magnitude	-18
Excess-128	18

Or `01010101`: 85 unsigned, -43 in excess-128, `'U'` in ASCII, or 10.625 as fixed point with 5 bits before the binary point and 3 after.

The format is not encoded in the data. It has to be known separately — by a human, by a program, or by the hardware. See week1-lab-number-representations[74] for the full 4-bit table across all five formats, and 2026-08-04-binary-arithmetic[75] for what happens when you add these.

Fractions: fixed and floating point

Everything above is whole numbers. Place values carry on past a **binary point** as $\frac{1}{2}, \frac{1}{4}, \frac{1}{8}, \ldots$, and there are two ways to use them.

- **Fixed point**: agree where the point sits, as in the 5.3 reading of `01010101` above, and read the bits as before. Ordinary integer hardware does the arithmetic, but the format fixes the range and the precision once and for all. See fixed-point-numbers[76].
- **Floating point**: store the digits and the position of the point separately, as binary scientific notation: a sign, a biased exponent and a mantissa. This is IEEE 754, the `float` and `double` of C. See floating-point-numbers[77].

Boolean Algebra

Logic functions can be expressed as expressions of **variables** (literals, e.g. A, B, X) and **functions** (the operators $+$, $\cdot$, $\oplus$, $\bar{}$) — i.e. an output written in terms of its inputs, $X = f(A, B)$. Variables and functions can only take the values 0 or 1. **Boolean algebra** is the set of rules governing how such expressions can be manipulated.

Notation conventions

- **Inversion**: overline, e.g. $\mathrm{NOT}(A) = \bar{A}$ (pronounced "A bar").
- **AND**: dot, or implied by adjacency, e.g. $\mathrm{AND}(A, B) = AB = A \cdot B$.
- **OR**: plus sign, e.g. $\mathrm{OR}(A, B, C) = A + B + C$.

[74] courses/csse2010/week1-lab-number-representations.pdf
[75] courses/csse2010/2026-08-04-binary-arithmetic.pdf
[76] courses/csse2010/fixed-point-numbers.pdf
[77] courses/csse2010/floating-point-numbers.pdf

- Other examples: $\text{XOR}(A, B) = A \oplus B = \bar{A}B + A\bar{B}$; $\text{NAND}(A, B, C) = \overline{ABC}$; $\text{NOR}(A, B) = \overline{A + B}$.

$+$ and $\cdot$ are **OR** and **AND**, not arithmetic addition and multiplication — $1 + 1 = 1$ here.

Boolean identities

Name	AND form	OR form
Identity law	$1A = A$	$0 + A = A$
Null law	$0A = 0$	$1 + A = 1$
Idempotent law	$AA = A$	$A + A = A$
Inverse law	$A\bar{A} = 0$	$A + \bar{A} = 1$
Commutative law	$AB = BA$	$A + B = B + A$
Associative law	$(AB)C = A(BC)$	$(A + B) + C = A + (B + C)$
Distributive law	$A + BC = (A + B)(A + C)$	$A(B + C) = AB + AC$
Absorption law	$A(A + B) = A$	$A + AB = A$
De Morgan's law	$\overline{AB} = \bar{A} + \bar{B}$	$\overline{A + B} = \bar{A}\bar{B}$

Two rows in particular do the heavy lifting when simplifying: the **inverse law** ($A\bar{A} = 0$, $A + \bar{A} = 1$) collapses whole terms to a constant, and the **absorption law** ($A + AB = A$) deletes a term outright.

The **distributive law in its AND form is not the arithmetic one** — $A + BC = (A + B)(A + C)$ has no algebraic analogue and is easy to misremember.

De Morgan's law and gate equivalents

De Morgan's law says AND and OR can be **interchanged if you invert both the inputs and the output**. Drawn as gates, that gives four equivalences — every gate has a second symbol, and pushing an inversion through a gate always flips its shape:

Exercise: **verify each of the four with a truth table.**

This is also why NAND-only and NOR-only circuits can implement any function (see logic-gates[78]) — an AND feeding an OR can always be redrawn as a NAND feeding a NAND.

[78] courses/csse2010/logic-gates.pdf

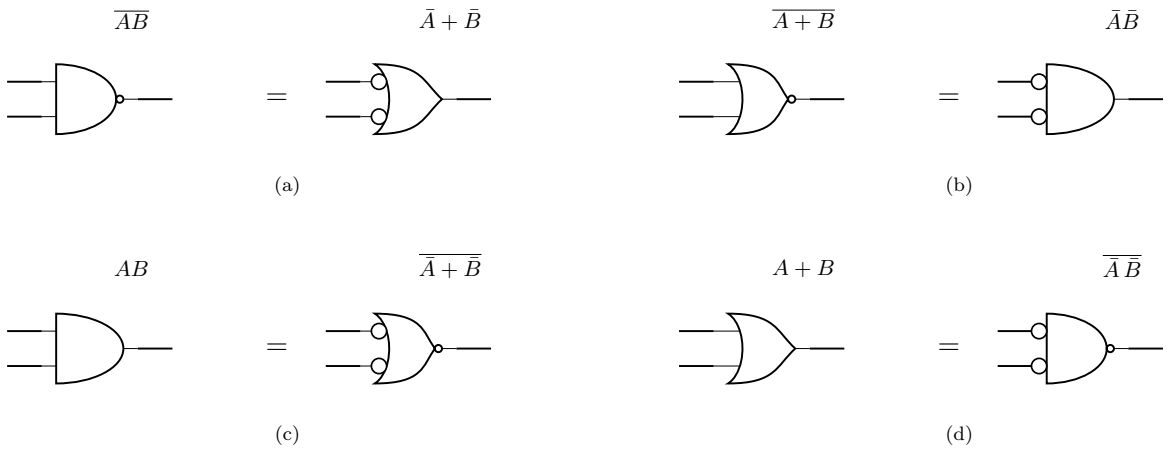


Figure 21: De Morgan's law drawn as gate symbols. (a) a NAND is an OR with inverted inputs; (b) a NOR is an AND with inverted inputs; (c) and (d) are the same two statements read the other way. Each pair is the same function, so either symbol may be used - pick whichever makes the surrounding bubbles cancel.

Simplifying an expression

Worked example:

$$\text{CODE} = C + CD + \overline{(D + \bar{D})}$$

1. Apply **De Morgan's law** to the last term: $\overline{D + \bar{D}} = \bar{D} \cdot \bar{\bar{D}} = \bar{D} \cdot D = 0$ (the double overline cancels, then the **inverse law** $A\bar{A} = 0$ applies).
2. The **identity law** $0 + A = A$ drops that term: $\text{CODE} = C + CD$.
3. The **absorption law** $A + AB = A$ finishes it: $\text{CODE} = C$.

The whole expression is just C — neither D nor the CD term affects the output at all.

Sum of products

Any logic function can be implemented as the **OR of AND combinations** of its inputs (a *sum of products*):

1. For each row where the truth table output is 1, write the AND term of the inputs (complementing wherever the input is 0) that produces that row.
2. OR all of those terms together.

Because every 1-row gets its own term, and each term is true for exactly that one row, the result is guaranteed to reproduce the table. Worked through in full, with the resulting circuit drawn out, in 2026-07-30-intro-to-logic-gates[79].

Sum of products always works, but does **not** necessarily give the minimum number of gates — the resulting expression can usually be simplified further with the identities above. E.g. $Z = AB + AC = A(B + C)$ (distributive law) needs two gates rather than three.

Circuit Schematics

A **logic diagram** shows the *idea* of a circuit and is hardware-independent (see logic-gates[80]). A **circuit schematic** goes further: it tells you how to actually **build** the circuit on real hardware — labelled ICs, pin numbers, power connections — so it's hardware-specific. Schematics are drawn for practicals and assessment using 74-series logic chips and the CSSE2010/CSSE7201 IO Board (see device-pinouts[81] for pin layouts). The rules are set out in the *CSSE2010/CSSE7201 - Guide to Drawing Circuit Schematics* (Blackboard, v2.3).

Required elements

Every circuit schematic must show:

- **Labelled inputs and outputs** — every input/output needs a sensible, unique name. Inputs and outputs use different symbols, and by convention inputs are drawn on the left, outputs on the right.
  - Both symbols are the same right-pointing pentagon; what differs is **which end the wire attaches to**. An **input** has its wire leaving the *point*, so the signal flows out of the tip into the circuit. An **output** has its wire arriving at the flat *back* edge, with the point left free. Getting this backwards is one of the common errors below.
- **Labelled devices** — each physical device (chip or IO board) gets a unique identifier: one or more letters for the device type, then a sequential number starting from 1.
  - Logic chips: `U1`, `U2`, `U3`, …
  - IO board: `IO1` (`IO2` for a second board, etc.) — when a group of inputs/outputs all belong to the same IO board, it only needs to be labelled once for the whole group, not per signal.

[79] courses/csse2010/2026-07-30-intro-to-logic-gates.pdf
[80] courses/csse2010/logic-gates.pdf
[81] courses/csse2010/device-pinouts.pdf

- **Gate labels within a chip** — where a chip contains multiple gates, each gate is labelled `A`, `B`, `C`, … after the chip ID, separated by a colon (e.g. `U1:A`, `U1:B`). Labels can be assigned to gates in any order. A chip with only one gate/device doesn't need a gate letter (just `U3`).
- **Device type** — written on the line below the device ID (e.g. `74HCT00`, `74HCT04`, `IOBOARD`).
- **Pin numbers** — every pin used must be numbered per the actual device pinout (see device-pinouts[82]).
- **Power supply connections** — shown once per device *type*, not per individual chip. Where two+ chip types share the same power pins, they can be grouped together. The IO board doesn't need power connections shown.

Worked example

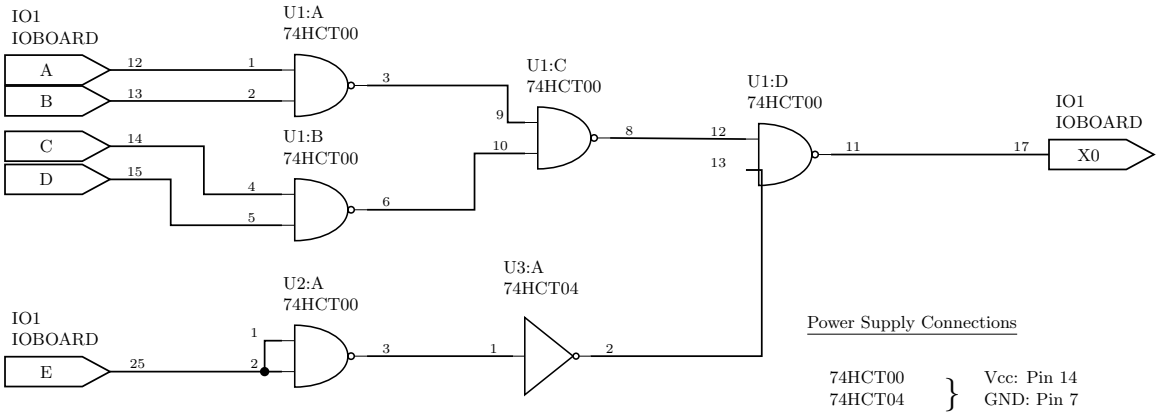


Figure 22: A correct schematic. Every rule above is visible here at once: five named inputs (A-E) and one named output (X0) on the IO board, four gates of one 74HCT00 labelled U1:A to U1:D, a fifth NAND that had to move to a second chip U2, an inverter on a third chip U3 because it is a different gate type, a pin number on every terminal, a junction dot where E splits to both inputs of U2:A, and one power-supply block covering both chip types.

Reading it against the rules:

- **Naming**: the inputs are `A`, `B`, `C`, `D`, `E` and the output is `X0`. The two IO-board groups are each labelled `IO1`/`IOBOARD` once, not per signal.
- **Chip identifiers**: `U1` supplies four NAND gates (`U1:A`–`U1:D`). A fifth NAND is needed, so it comes from a *second* chip, `U2:A`. The inverter is a different gate type, so it needs a *third* chip, `U3:A`, of type 74HCT04.

[82] courses/csse2010/device-pinouts.pdf

- **Pin numbers**: the 74HCT00's four gates use inputs 1&2, 4&5, 9&10, 12&13 with outputs 3, 6, 8, 11 respectively — so those are the numbers written on `U1:A`–`U1:D`. `U3:A` uses 1 → 2.
- **The junction dot**: `E` (pin 25) drives *both* inputs of `U2:A`, which is what turns a NAND into an inverter. The dot at the split is what says the two wires are joined.
- **Power**: `74HCT00` and `74HCT04` have the same power pins, so they are braced together into one block: Vcc on pin 14, GND on pin 7. Nothing is shown per chip, and nothing at all for the IO board.

Common errors

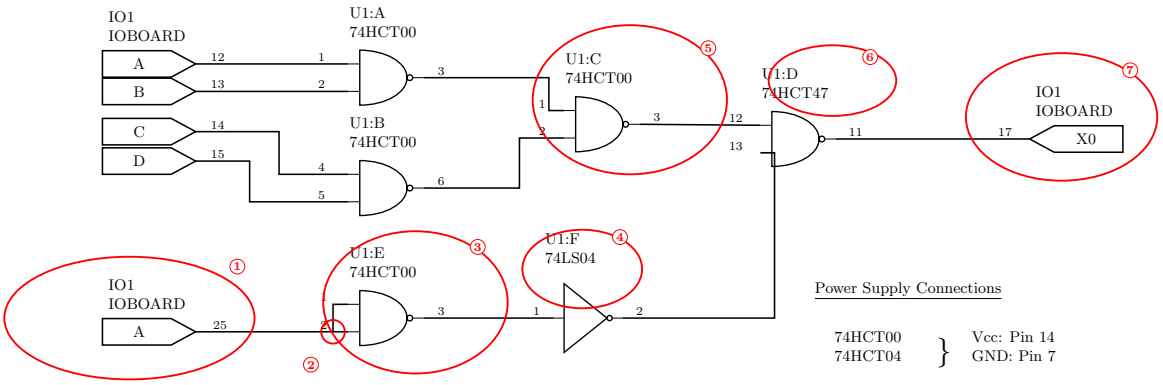


Figure 23: The same circuit with seven deliberate mistakes ringed and numbered left to right, matching the list below. Compare each ring against the corresponding part of the correct schematic above.

1. **Duplicate names** — two inputs given the same name (both called `A`). Every input needs a unique name.
2. **Missing junction dots** — where wires join or split, a dot must mark the connection. Wires that merely cross on the page, with no dot, are *not* electrically connected. Here the split feeding both inputs of the NAND-as-inverter has no dot.
3. **Gate label exceeds chip capacity** — `U1:E` on a 74HCT00, which only has four 2-input NAND gates (`A`–`D`). A fifth gate needs a new chip ID (`U2:A` or similar).
4. **Wrong device for the gate type** — `U1:F` is a NOT gate, a different type from the rest of `U1`, so it needs its own chip identifier (`U3:A`). (The figure also gives it as a `74LS04` while the rest of the schematic uses `74HCT` parts.)
5. **Reusing pin numbers** — `U1:C` is drawn with pins 1, 2 → 3, the same physical gate as `U1:A`. You can't use one gate twice.
6. **Inconsistent device type** — `U1:D` is labelled `74HCT47` when the rest of `U1` is a `74HCT00`. Every gate labelled `U1:something` must show the same chip type.
7. **Output errors** — two distinct mistakes on `X0`:

a. The *input* symbol is used (wire attached to the point) where the output symbol is needed (wire attached to the flat back) — see Figure 22.
b. An input is wired to the output of a gate. An input should never be tied straight to a gate's output; **this may destroy the device**. Gate outputs should only feed output devices (e.g. LEDs) or the inputs of other gates.

See also

- device-pinouts[83] — the pin numbers and symbols every schematic has to match
- logic-gates[84] — the gate symbols themselves, and the logic-diagram level above this one
- week3-lab-lab-4-pre-work[85] — the preparation task that asks for schematics of a 2-to-1 mux, a 2-to-4 decoder and a 4-bit adder/subtractor

Combinational Logic Blocks

Reference note for the standard combinational building blocks: the adder-subtractor, the multiplexer, and the decoder. See logic-gates[86] for the gate symbols/truth tables and boolean-algebra[87] for the notation used below.

What makes a circuit combinational

A **combinational circuit** is a combination of logic gates with n inputs and m outputs:

- Each output can be expressed as a function of the n input variables.
- The **output depends on the current inputs only** — there is no memory of previous inputs. (Contrast with sequential-circuits[88], where the output also depends on stored state.)
- It can always be written as a truth table with n input columns, m output columns, and 2^n rows (one per possible input combination).

[83] courses/csse2010/device-pinouts.pdf
[84] courses/csse2010/logic-gates.pdf
[85] courses/csse2010/week3-lab-lab-4-pre-work.pdf
[86] courses/csse2010/logic-gates.pdf
[87] courses/csse2010/boolean-algebra.pdf
[88] courses/csse2010/sequential-circuits.pdf

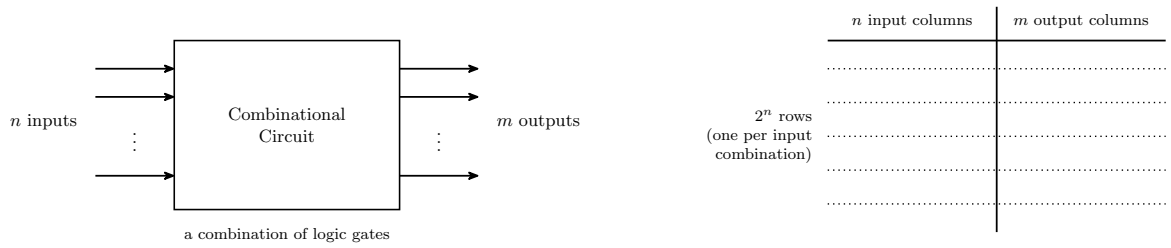


Figure 24: The general shape of a combinational circuit and the truth table it is always equivalent to. The row count 2^n depends only on the number of inputs, and the output column count m only on the number of outputs — so a block with six inputs needs 64 rows however few gates are inside it.

Binary subtraction

$A - B$ is usually implemented as $A + (-B)$, where:

- A and B are multi-bit quantities;
- "+" here means **addition**, not OR;
- $-B$ means the **two's complement** of B (see binary-number-representations[89]).

The two's complement of B is calculated by **flipping all the bits and adding 1**:

$$-B = \bar{B} + 1$$

So a subtractor is just an adder with two extra pieces: something that conditionally inverts B, and something that adds the extra 1.

The controlled inverter (XOR trick)

We need a gate that flips a bit *only sometimes*: given a mode bit M,

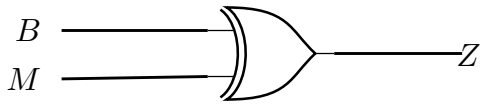
$$Z = \bar{B} \text{ when } M = 1, \qquad Z = B \text{ when } M = 0$$

From the XOR truth table: XOR with one input held at 0 passes the other input through unchanged, and XOR with one input held at 1 inverts it. So the gate is a **2-input XOR**:

$$Z = B \oplus M$$

[89] courses/csse2010/binary-number-representations.pdf

M	B	$Z = B \oplus M$	Effect
0	0	0	pass through
0	1	1	pass through
1	0	1	invert
1	1	0	invert



$$M = 0: \; Z = B \oplus 0 = B \quad \text{(pass through)}$$
$$M = 1: \; Z = B \oplus 1 = \bar{B} \quad \text{(invert)}$$

Figure 25: The controlled inverter: one XOR gate per bit, with the mode line on the second input. The two identities on the right are the whole trick, and they hold for any signal X — XOR against a constant 0 is a wire, XOR against a constant 1 is an inverter.

This is called a **controlled inverter**.

Adder-subtractor circuit

A 4-bit adder-subtractor is built from a 4-bit binary adder (a ripple-carry adder — see 2026-08-04-binary-arithmetic[90]) plus four XOR gates:

- Data input A ($A_0 \ldots A_3$) goes **straight** into the adder's A inputs.
- Data input B ($B_0 \ldots B_3$) goes into the adder's B inputs **via one XOR gate per bit**; the second input of every XOR gate is the shared **mode select** line M.
- $M = 0$ means **add**, $M = 1$ means **subtract**.
- The adder produces the data output $S_0 \ldots S_3$, plus a carry-out C_4 and taking a carry-in C_0.

The carry-in. With $M = 1$ the XOR gates supply $\bar{B}$, and two's complement needs $\bar{B} + 1$ — the extra $+1$ is supplied by feeding it in as the carry-in. With $M = 0$ we want plain addition, which needs a carry-in of 0. Both cases are satisfied by

$$C_0 = M$$

so the mode select line is wired to both the XOR gates and the adder's carry-in.

The cost of subtraction is therefore **four XOR gates and one wire** on top of an adder — a 74HCT86 (quad 2-input XOR) plus a 74HCT283 (4-bit adder), see device-pinouts[91].

[90] courses/csse2010/2026-08-04-binary-arithmetic.pdf
[91] courses/csse2010/device-pinouts.pdf

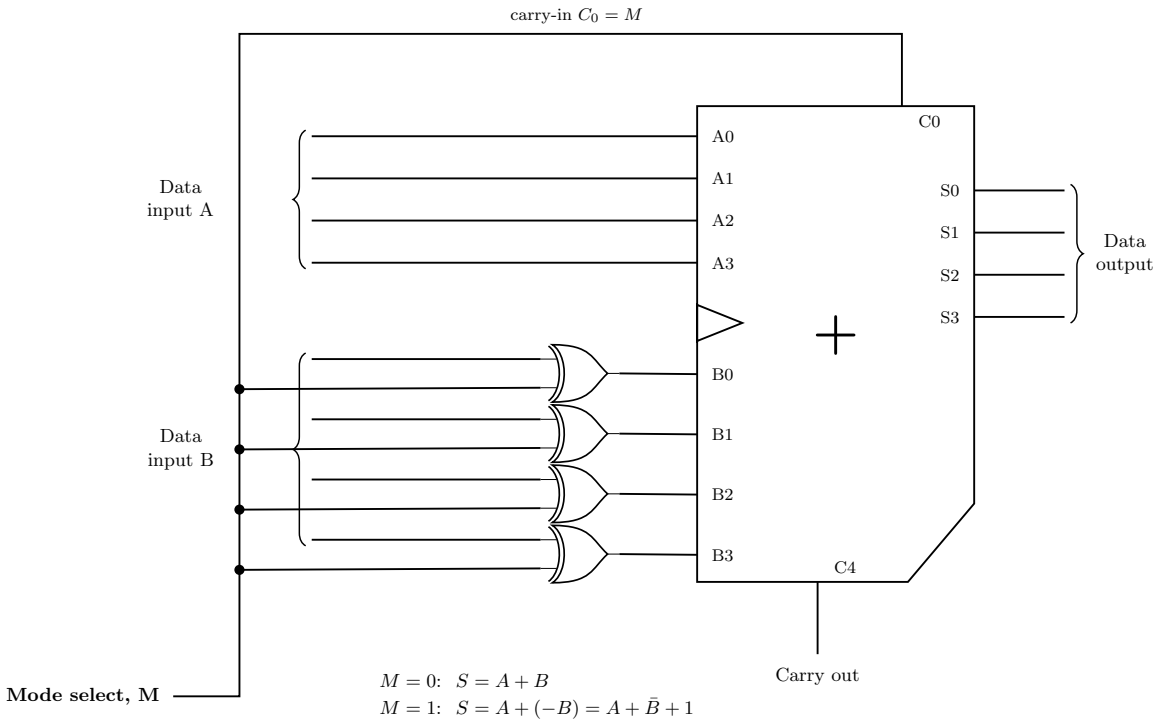


Figure 26: The 4-bit adder-subtractor. One mode line does two jobs: it conditions every bit of B through an XOR, and it is the adder's carry-in. With M = 0 the XORs pass B through and the carry-in is 0, so the adder computes A + B; with M = 1 the XORs deliver B-bar and the carry-in is 1, so it computes A + B-bar + 1, which is A + (-B). No extra hardware is spent on the +1 at all.

Multiplexer (mux)

A **multiplexer** — also called a **data selector** — has:

- 2^n **data** inputs,
- **1** output,
- n **control** (or **select**) inputs, which *select* one of the data inputs to be "sent" or "steered" to the output.

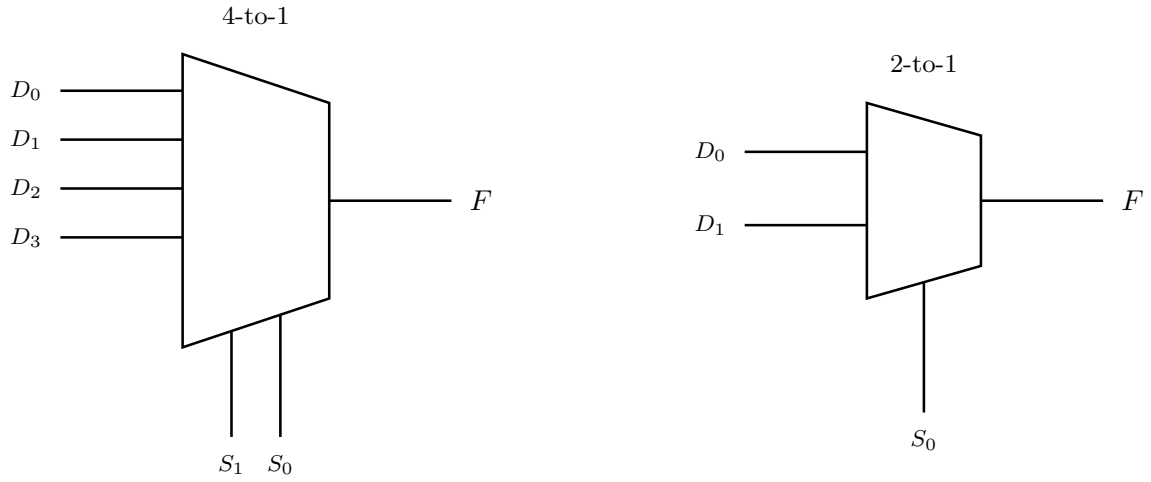


Figure 27: The multiplexer logic symbol at two sizes, 4-to-1 and 2-to-1. Data inputs enter the tall left edge, the single output leaves the short right edge, and the select inputs enter from underneath. The number of select lines is what fixes the size: n selects steer 2^n data inputs.

4-to-1 multiplexer

Data inputs $D_0 \ldots D_3$, select inputs $S_1 S_0$, output F. The logic symbol is the characteristic trapezoid with the data inputs on the wide (left) edge, F on the narrow (right) edge, and the select inputs entering the bottom.

Function table:

S_1	S_0	F
0	0	D_0
0	1	D_1
1	0	D_2
1	1	D_3

This is a **function table**, not a full truth table — the full truth table would need $2^6 = 64$ rows for the six inputs $S_1, S_0, D_0, D_1, D_2, D_3$. The function table's four rows say the same thing far more compactly, which is why it is what gets drawn.

4-to-1 mux logic circuit implementation

- S_1 and S_0 each feed an inverter, giving $\bar{S}_1$ and $\bar{S}_0$ as well as the true forms.
- Four **3-input AND gates**, one per data input. Each AND gate takes its data input plus the select-literal pair that identifies it: D_0 with $\bar{S}_1\bar{S}_0$, D_1 with $\bar{S}_1 S_0$, D_2 with $S_1\bar{S}_0$, D_3 with $S_1 S_0$.
- The four AND outputs feed a single **4-input OR gate** whose output is F.

That is exactly the sum-of-products form:

$$F = D_0\bar{S}_1\bar{S}_0 + D_1\bar{S}_1 S_0 + D_2 S_1\bar{S}_0 + D_3 S_1 S_0$$

Only the AND gate whose select literals match the current S_1S_0 can be 1, so exactly one data input reaches the OR gate at a time. The other three AND gates output 0, and 0 is the identity for OR, so they contribute nothing.

The gate-level version needs **3-input ANDs and a 4-input OR**, neither of which exists in the lab kit (see device-pinouts[92]) — building this from the available 2-input chips means cascading them.

2-to-1 multiplexer

Data inputs D_0 and D_1, one control input S_0, output F.

Function table:

S_0	F
0	D_0
1	D_1

Expanded to a full truth table over all three inputs ($2^3 = 8$ rows):

S_0	D_0	D_1	F
0	0	0	0
0	0	1	0
0	1	0	1

[92] courses/csse2010/device-pinouts.pdf

S_0	D_0	D_1	F
0	1	1	1
1	0	0	0
1	0	1	1
1	1	0	0
1	1	1	1

i.e. $F = D_0\bar{S}_0 + D_1 S_0$. When $S_0 = 0$ the output tracks D_0 and ignores D_1; when $S_0 = 1$ it tracks D_1 and ignores D_0.

A quad 2-input multiplexer is available as an off-the-shelf IC — the 74HCT157, see device-pinouts[93].

Using a mux to implement a logic function

Because the data inputs can be tied to constant 0 or 1, an n-select mux with its selects wired to the function's variables can implement *any* function of those n variables: set data input D_i to the value the function should take when the selects equal i. The function table **is** the truth table of the function, read off directly. See the worked question in 2026-08-06-combinational-logic[94].

Decoder

A **decoder** converts an n-bit input to a logic 1 on **exactly one** of its 2^n outputs (the one whose index equals the input value); all other outputs are 0.

3-to-8 decoder

Inputs A, B, C (with A the most significant), outputs $D_0 \dots D_7$.

A	B	C	D_0	D_1	D_2	D_3	D_4	D_5	D_6	D_7
0	0	0	**1**	0	0	0	0	0	0	0
0	0	1	0	**1**	0	0	0	0	0	0
0	1	0	0	0	**1**	0	0	0	0	0
0	1	1	0	0	0	**1**	0	0	0	0
1	0	0	0	0	0	0	**1**	0	0	0
1	0	1	0	0	0	0	0	**1**	0	0
1	1	0	0	0	0	0	0	0	**1**	0
1	1	1	0	0	0	0	0	0	0	**1**

[93] courses/csse2010/device-pinouts.pdf
[94] courses/csse2010/2026-08-06-combinational-logic.pdf

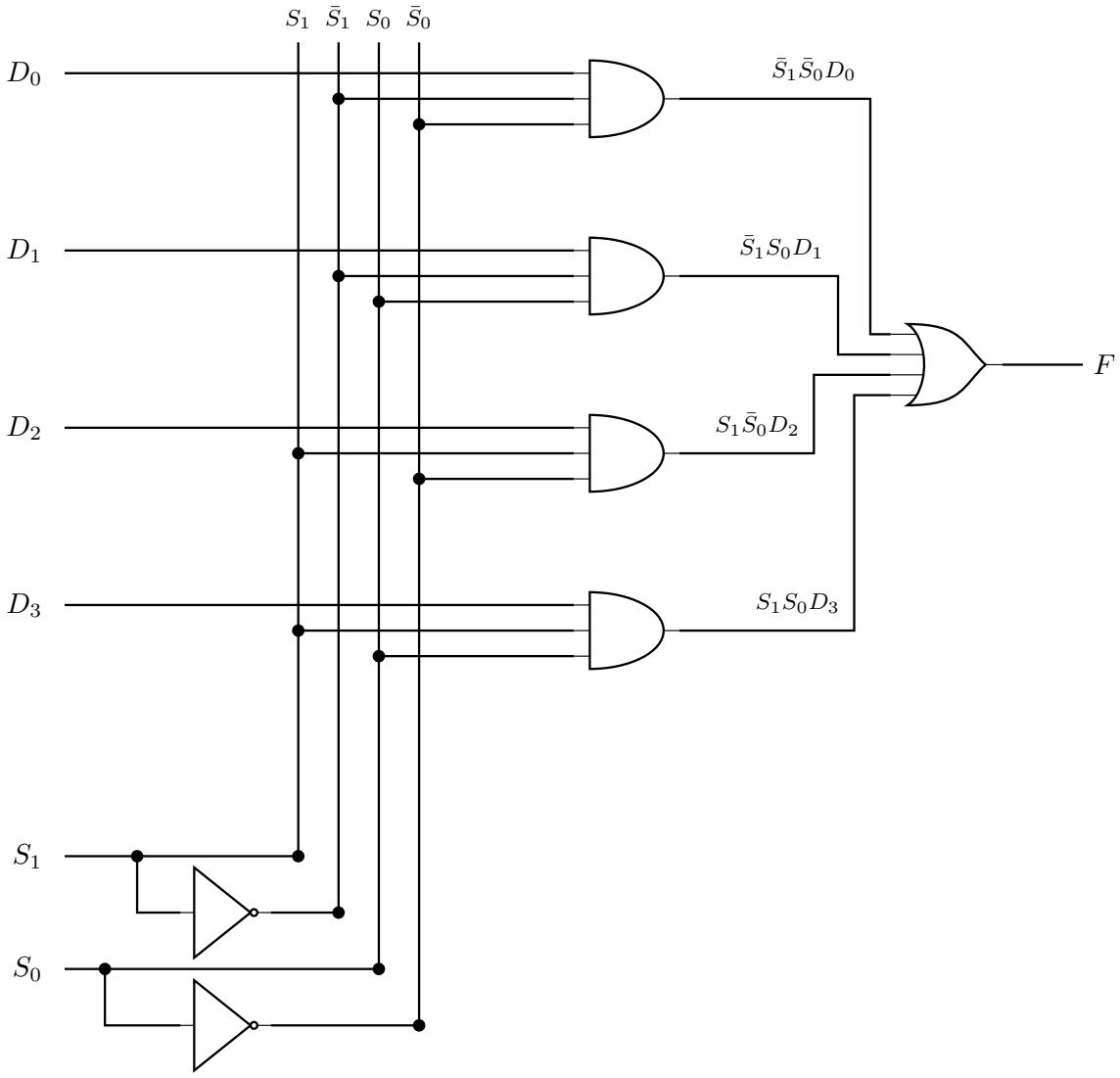


Figure 28: A 4-to-1 multiplexer built from gates. The two inverters generate the complemented selects; the four vertical rails carry S1, S1-bar, S0 and S0-bar past every AND gate; each AND gate is one data input gated by the one select combination that names it, and the OR gate collects them. Only the dots are connections — every other crossing is two wires passing.

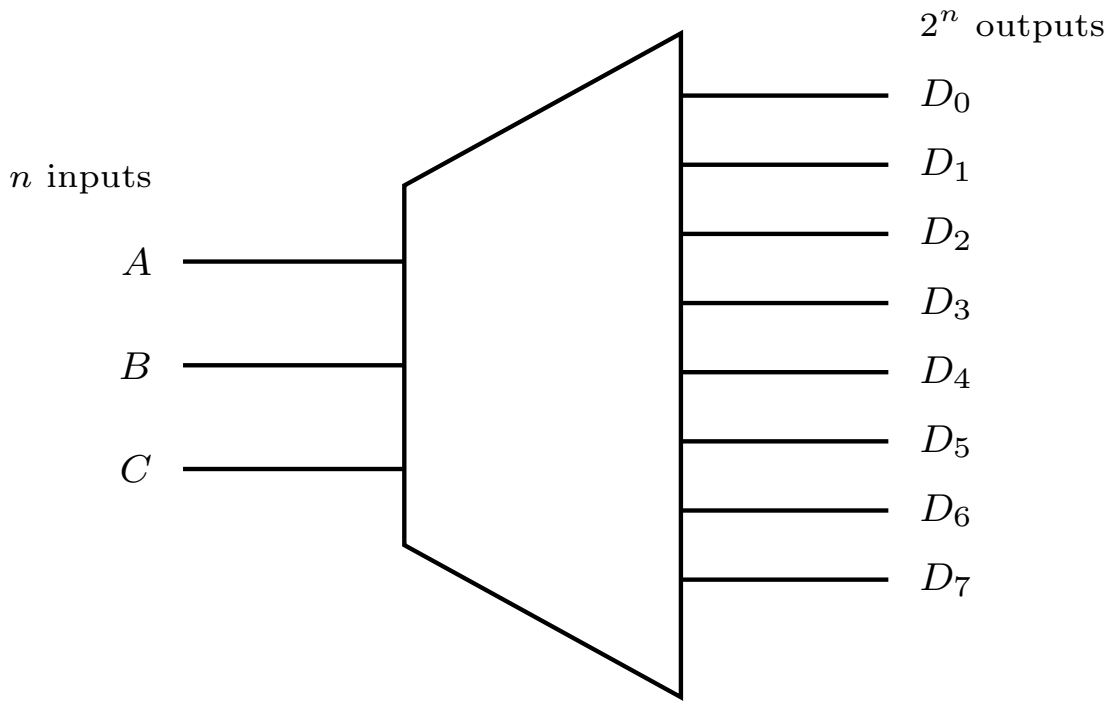


Figure 29: The decoder as a block: n inputs in, 2^n outputs out, and the shape widens accordingly. It is the mux's opposite — a mux funnels 2^n inputs down to one output under n selects, a decoder fans n inputs out to 2^n outputs.

The 1 always sits in the column whose index is the binary value of ABC.

Each row of the table is one **minterm**: the top row is $\bar{A}\bar{B}\bar{C}$ and the bottom row is ABC.

3-to-8 decoder logic circuit implementation

- Each of A, B, C is fanned out to an inverter, so all six literals $A, \bar{A}, B, \bar{B}, C, \bar{C}$ are available on a vertical bus.
- **Eight 3-input AND gates**, one per output. Each taps the three literals matching its index:

$$D_0 = \bar{A}\bar{B}\bar{C}, \quad D_1 = \bar{A}\bar{B}C, \quad D_2 = \bar{A}B\bar{C}, \quad D_3 = \bar{A}BC$$
$$D_4 = A\bar{B}\bar{C}, \quad D_5 = A\bar{B}C, \quad D_6 = AB\bar{C}, \quad D_7 = ABC$$

Each AND gate is the minterm for one input combination, so exactly one is high at any time.

Using a decoder to implement a logic function

Because a decoder hands you every minterm of its inputs on a separate wire, any function of those inputs is just an **OR of the minterm outputs whose truth-table rows are 1**. For example, take

$$F = \bar{A}\bar{B}\bar{C} + ABC$$

which is $D_0 + D_7$ — one OR gate on two decoder outputs, with nothing else needed.

This is the same idea as the mux trick above, approached from the other side: a mux picks one of 2^n *values*, a decoder hands you all 2^n *minterms* and you OR the ones you want.

See also

- 2026-08-06-combinational-logic[95] — Lecture 4, where these blocks were introduced
- device-pinouts[96] — the off-the-shelf parts: 74HCT157 (mux), 74HCT283 (adder), 74HCT86 (XOR)
- arithmetic-logic-unit[97] — the ALU bit slice, which is a mux selecting between several function blocks
- binary-number-representations[98] — two's complement, on which the subtractor depends

[95] courses/csse2010/2026-08-06-combinational-logic.pdf
[96] courses/csse2010/device-pinouts.pdf
[97] courses/csse2010/arithmetic-logic-unit.pdf
[98] courses/csse2010/binary-number-representations.pdf

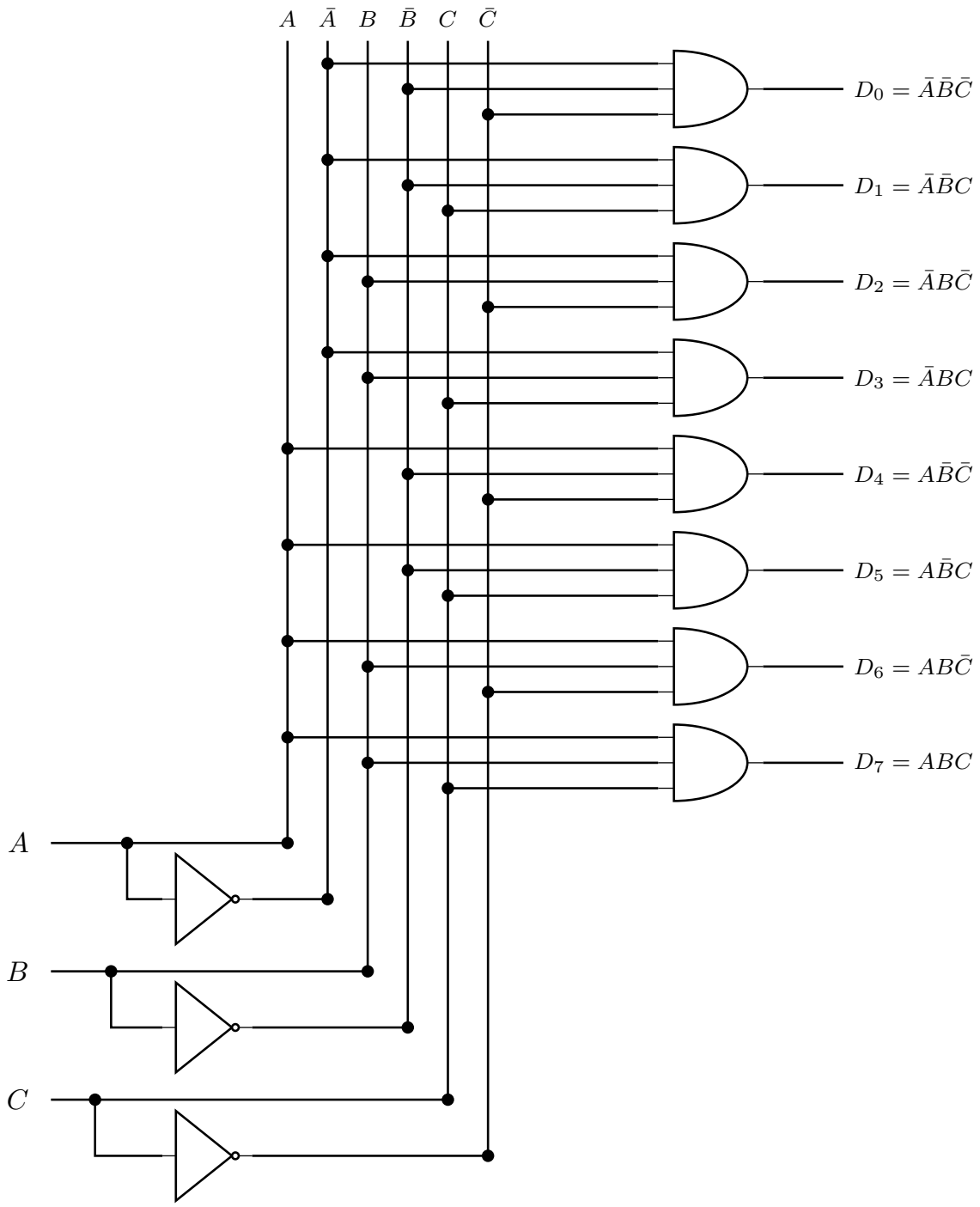


Figure 30: A 3-to-8 decoder built from gates. Three inverters generate the complements onto six vertical rails; each of the eight AND gates taps the one literal triple that matches its index, so it is the minterm for that input combination. Exactly one AND gate can be high at a time, which is the decoder's defining property.

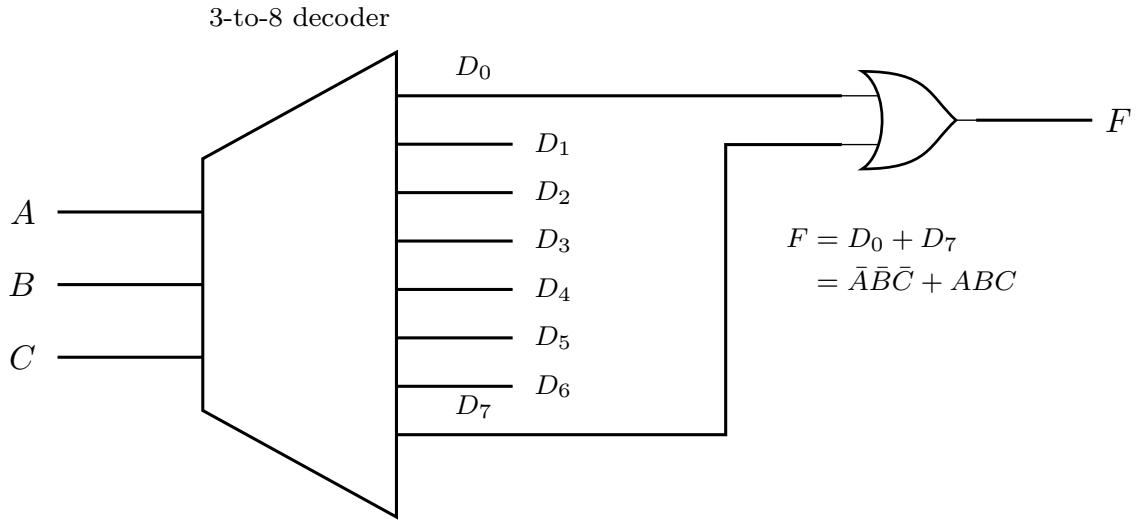


Figure 31: A function built on a decoder: F = A-bar B-bar C-bar + ABC is D0 + D7, so it costs a single OR gate on top of the decoder. Any function of A, B and C can be built this way — pick out the outputs for its true rows and OR them together.

Counters

A **counter** is a multi-bit register that goes through a *determined sequence of states* (values) upon the application of input (clock) pulses. It is a synchronous sequential circuit[99] built from D flip-flops[100] plus some combinational logic in the feedback path.

Binary counters

A counter which follows the binary number sequence is a **binary counter**.

An n**-bit binary counter**:

- has n flip-flops;
- has 2^n different states;
- counts from 0 to $2^n - 1$, then wraps back to 0.

For example, a 2-bit binary counter counts

$$00 \to 01 \to 10 \to 11 \to 00 \to ...$$

[99] courses/csse2010/sequential-circuits.pdf
[100] courses/csse2010/flip-flops-and-latches.pdf

i.e. $0 \to 1 \to 2 \to 3 \to 0 \to ...$

4-bit binary counter — counting sequence

Bit 3	Bit 2	Bit 1	Bit 0	Value
0	0	0	0	0
0	0	0	1	1
0	0	1	0	2
0	0	1	1	3
0	1	0	0	4
0	1	0	1	5
0	1	1	0	6
0	1	1	1	7
1	0	0	0	8
1	0	0	1	9
1	0	1	0	10
1	0	1	1	11
1	1	0	0	12
1	1	0	1	13
1	1	1	0	14
1	1	1	1	15
0	0	0	0	0

Each bit toggles at half the rate of the bit to its right — bit 0 changes every count, bit 1 every second count, bit 2 every fourth, bit 3 every eighth.

Key points

- The **next state is a function of the present state** (and possibly of external inputs).
- The count sequence **can** be the binary numbers but **does not have to be** — if it is, the counter is a binary counter; otherwise it is just some arbitrary cycle through states.
- These circuits are **synchronous**: all flip-flops share the same clock.
- **Asynchronous** counters exist too, but are not discussed in this course.

State: the design idea

This is the whole trick, and it is what makes counter design mechanical:

- The values stored in the flip-flops are the **current (present) state** of the circuit.

- The D inputs to the flip-flops are the **next state** — whatever is sitting on D at the clock edge becomes the new state.
- The D inputs are some **combinational function of the current state** (and of any external inputs).

So for a counter with state bits $Q_{n-1} \ldots Q_1 Q_0$, designing the counter means finding n Boolean expressions

$$D_i = f_i(Q_{n-1}, \ldots, Q_1, Q_0)$$

and building each one out of gates[101].

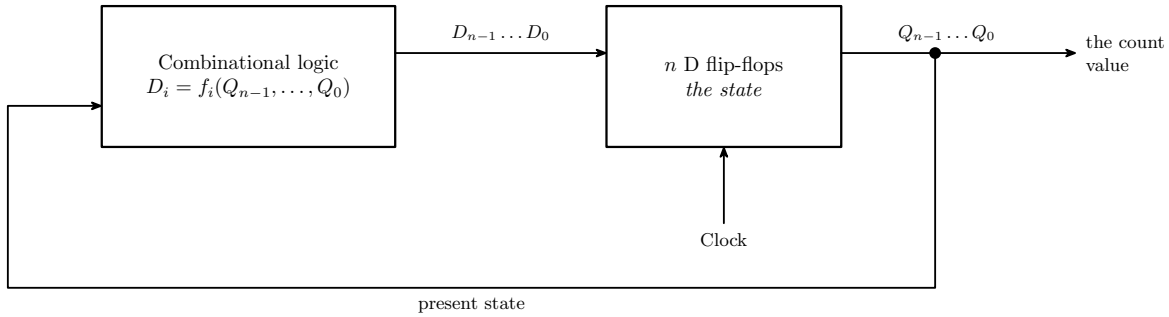


Figure 32: A counter is the general sequential-circuit block diagram with the external inputs deleted. That diagram, as Mano draws it, is a combinational circuit feeding flip-flops, the flip-flop outputs routed back round to the combinational circuit, and the clock entering the flip-flops on its own labelled stub. A counter may or may not have external inputs; removing them, as here, is what makes the design mechanical: with nothing else to depend on, each D is a function of the Q values alone, so the present-state/next-state table below is a complete specification of the circuit.

Design recipe

1. **Write down the count sequence** you want, as a cycle of state values.
2. **Build a present-state / next-state table.** One row per possible state — all 2^n of them, not just the ones in your sequence. Columns: present state $Q_{n-1} \ldots Q_0$, then next state $D_{n-1} \ldots D_0$. Fill each next-state entry by reading off what follows that state in your sequence.
3. **Handle unused states.** If your sequence doesn't use all 2^n states, the next state for the unused ones is a **don't-care** (X). You may set each X to whatever makes the algebra simplest — but check afterwards where the unused state actually goes, because a

[101] courses/csse2010/logic-gates.pdf

self-correcting counter (one whose unused states lead back into the cycle) is nicer than one that can get stuck.

4. **Read each D column as a Boolean function of the Qs.** The table *is* a truth table with the Qs as inputs and D_i as the output, so write it in sum-of-products form: OR together one AND term per row where $D_i = 1$. See boolean-algebra[102].
5. **Simplify** each expression using the Boolean algebra laws (and the don't-cares), then draw the circuit: the combinational logic for D_i feeding flip-flop i, with every flip-flop on a **common clock**. See circuit-schematics[103] for schematic conventions and device-pinouts[104] for the 74HCT74 / 74HCT175 D flip-flop packages.

The smallest case falls straight out of the recipe: a **1-bit counter** counts $0 \to 1 \to 0 \to \ldots$, so its next state is always the complement of its present state, giving $D = \bar{Q}$ — a single flip-flop with its $\bar{Q}$ output wired back to its own D input (a *toggle*).

Worked example — a 2-bit counter for $00 \to 10 \to 01 \to 00$

Step 1–2 — present-state / next-state table. The sequence uses three of the four states; 11 is unused, so its next state is a don't-care.

Q_1	Q_0	D_1	D_0	comment
0	0	1	0	$00 \to 10$
0	1	0	0	$01 \to 00$
1	0	0	1	$10 \to 01$
1	1	X	X	unused state

Step 4 — read off the columns.

D_1 is 1 only in the row $Q_1Q_0 = 00$. Making the don't-care 1 would give $Q_1 \odot Q_0$ (XNOR), which is no simpler, so take the don't-care as 0 ($X_1 = 0$):

$$D_1 = \bar{Q}_1\bar{Q}_0$$

D_0 is 1 only in the row $Q_1Q_0 = 10$, which gives $D_0 = Q_1\bar{Q}_0$. Here the don't-care *does* help: setting it to 1 ($X_0 = 1$) makes $D_0 = 1$ for both rows with $Q_1 = 1$, so

$$D_0 = Q_1$$

[102] courses/csse2010/boolean-algebra.pdf
[103] courses/csse2010/circuit-schematics.pdf
[104] courses/csse2010/device-pinouts.pdf

Step 5 — the circuit. Two D flip-flops on a common clock. Flip-flop 0's D input is wired directly to flip-flop 1's Q_1 output — no gate at all. Flip-flop 1's D input is driven by a 2-input NOR of Q_1 and Q_0 (since $\bar{Q}_1\bar{Q}_0 = \overline{Q_1 + Q_0}$), or equivalently by an AND of the two $\bar{Q}$ outputs.

Check, including the unused state:

present	$D_1 D_0$	next
00	1 0	10
10	0 1	01
01	0 0	00
11	0 1	01

The cycle is right, and the unused state falls into the cycle on the next clock edge, $11 \to 01$, so the counter is **self-correcting** — a robust design. (Choosing $D_0 = Q_1\bar{Q}_0$ instead — don't-care set to 0 — would send $11 \to 00$, also fine, at the cost of a whole extra gate: an AND fed from Q_1 and the flip-flop's $\bar{Q}_0$ pin, where $D_0 = Q_1$ needs no gate at all.)

Applied exercise

week4-lab-lab-7-exercises[105] is exactly this recipe at $n = 3$: design a 3-bit synchronous counter that steps through an allocated 7-state sequence, with the eighth (missing) state treated as a don't-care.

Related

- flip-flops-and-latches[106] — the D flip-flop and its characteristic table.
- sequential-circuits[107] — state, present state / next state, synchronous sequential circuits.
- shift-registers[108] — the other family of registers built from a flip-flop chain.
- boolean-algebra[109] — sum of products and the simplification laws used in step 4–5.
- binary-number-representations[110] — the binary counting sequence itself.
- state-machines[111] — the same design recipe generalised: a counter is a state machine whose transitions carry no input condition.

[105] courses/csse2010/week4-lab-lab-7-exercises.pdf
[106] courses/csse2010/flip-flops-and-latches.pdf
[107] courses/csse2010/sequential-circuits.pdf
[108] courses/csse2010/shift-registers.pdf
[109] courses/csse2010/boolean-algebra.pdf
[110] courses/csse2010/binary-number-representations.pdf
[111] courses/csse2010/state-machines.pdf

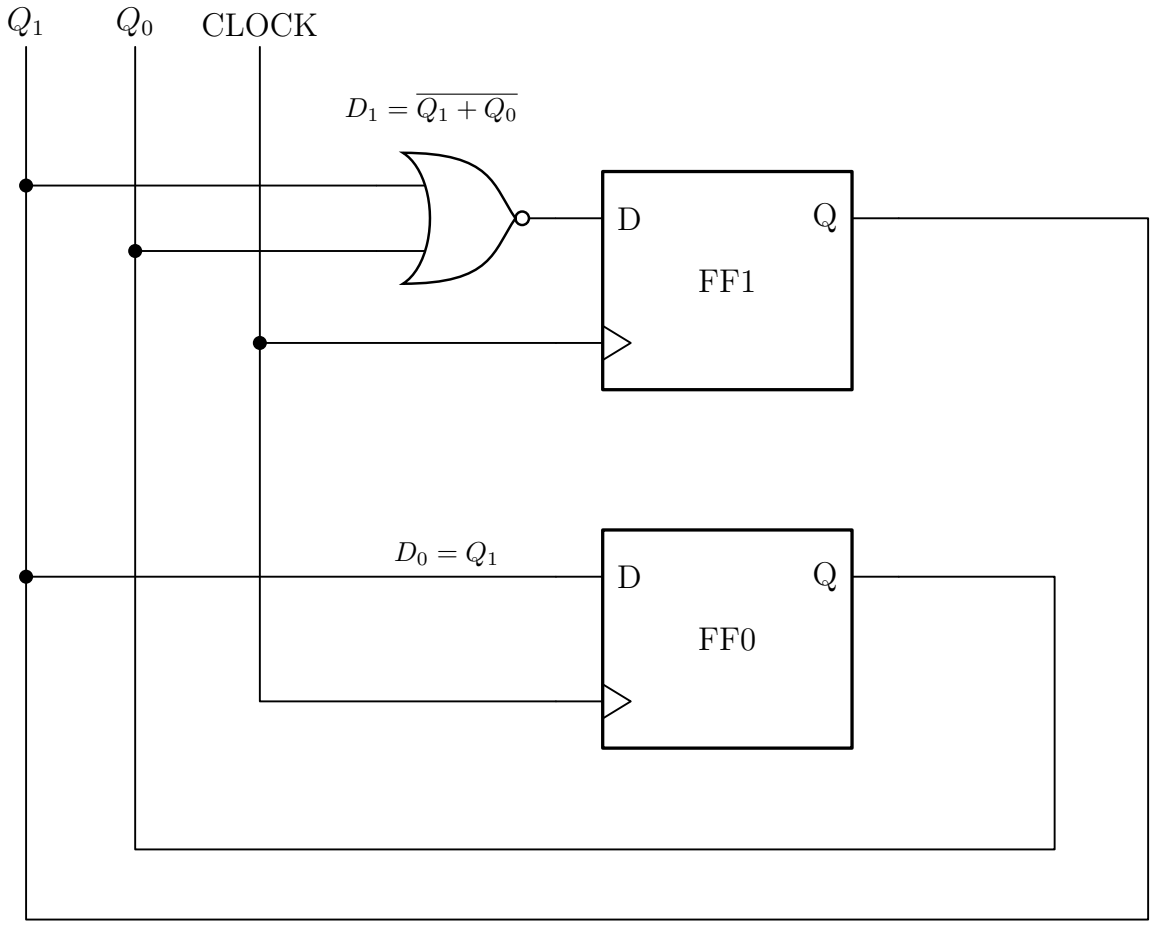


Figure 33: The whole counter, drawn from the derivation above. Each Q runs back to a vertical rail that the next-state logic taps, which is the physical form of the feedback loop in the block diagram earlier: D0 is FF1's Q wired straight across with no gate at all, and D1 is one NOR gate. Dots are connections; where the Q0 rail and the CLOCK rail cross the D0 wire there is no dot and no connection. The circuit takes the NOR form of D1; an AND of the two Q-bar outputs is the same function and would need the flip-flops' Q-bar pins instead.

- 2026-08-17-counters[112] — Lecture 7.
- csse2010-diagram-notation[113] — the drawing conventions the figures above follow.

CSSE2010 Diagram Notation

The drawing conventions this course actually uses, inferred from its own materials. This is the reference every later diagram in CSSE2010 should be checked against.

Scope. This note covers the diagram families that the two existing notation notes do *not*: gate symbols and logic diagrams, timing/waveform diagrams, state diagrams and state tables, block and datapath diagrams, register/memory layout diagrams, and the AVR architecture figures. It also records the algebraic and textual notation the figures depend on. For circuit-schematic rules (device IDs, gate letters, pin numbers, junction dots, power blocks, IO-board input/output pentagons) see circuit-schematics[114]; for the 74-series pinouts and their schematic symbols see device-pinouts[115]. Those two are not repeated here.

What was read

An honest statement of the sample, because most of what follows is inferred from drawings rather than stated in prose.

Guides — all three read in full (9 of 9 pages):

- *CSSE2010/CSSE7201 Device Pinouts and Symbols* (v4), pp. 1–2
- *CSSE2010/CSSE7201 — Guide to Drawing Circuit Schematics* (v2.3, Semester One 2016), pp. 1–2
- *I/O Board User Manual*, pp. 1–5

Lecture decks — 221 of 246 pages:

Deck	Pages read	of
L01 Intro and Binary Numbers	1–20	21
L02 Intro to Logic Gates	1–24	24
L03 Binary Arithmetic	8–17	17
L04 Combinational Logic	1–19	19
L05 Flip-Flops	4–19	19
L06 Shift Registers	3–17	17
L07 Counters	3–17	17

[112] courses/csse2010/2026-08-17-counters.pdf
[113] courses/csse2010/csse2010-diagram-notation.pdf
[114] courses/csse2010/circuit-schematics.pdf
[115] courses/csse2010/device-pinouts.pdf

Deck	Pages read	of
L08 State Machines	2–21	21
L09 ALU and Control Unit	3–22	22
L10 AVR Introduction	3–19	19
L11 ISA Part 1	3–28	28
L12 ISA Part 2	2–20	22

Unread: L01 p21, L03 pp1–7, L05 pp1–3, L12 pp21–22 — title, outline and admin pages in every case except L03 pp1–7 (number-representation recap, no new diagram family). Labs, past papers and the course profile were not sampled at all.

Citation format below: "L04 s10" means lecture deck 04, slide 10. Slide numbers printed on the slides match the PDF page numbers in every deck checked.

The course ships no notation guide for these diagram families

The three guides are *hardware* documents. Between them they settle schematic layout, device labelling, pin numbering and the physical IO board — nothing about gate-symbol family, waveform drawing, state diagrams, block diagrams or memory layout. The *Device Pinouts and Symbols* sheet comes closest: it has a "Logic Diagram Symbol" column (p. 1–2) that shows the gate shapes and the flip-flop/mux/adder block symbols, but it gives no rules, only examples.

So: **everything in the sections below except the schematic rules is inferred from the drawings**, and every entry is cited so you can check the inference.

Figure sources the slides cite

This matters more than it looks. L01 s8 says **"No textbook required"** — yet four different figure sources are credited across the semester, each bringing its own drawing style. There is no single house style.

Source, quoted as printed	Where	What it supplies
`[Figure from Mano, "Digital Design", 3rd edition]`	L03 s16, L04 s4, L04 s10, L05 s17, L08 s3	Ripple-carry adder, 4-to-1 mux circuit, sequential-circuit block diagram
`[Mano, Digital Design, 3rd ed.]` (short form)	L06 s6, L06 s8, L06 s9	4-bit register, 4-bit shift register

Source, quoted as printed	Where	What it supplies
`Images taken from Instructor Resources of Digital Fundamentals, Thomas L. Flyod, Pearson Educations Inc., 2015` (misspelling in the original)	L07 s4, s5, s7	Shift-register type diagrams, digital-system application block diagram
`[Figure from Tanenbaum, "Structured Computer Organization"]` — no edition given	L09 s8	Full ALU internal logic diagram
`[From ATmega324A datasheet]`	L11 s14	AVR data and program memory maps
`Some datasheet extracts follow` (prose, not a caption)	L10 s9, covering s10–s11	AVR CPU architecture, ATmega324A chip block diagram
`Source: Google Images`	L01 s4	Photo collages only, not line art

Tanenbaum is also cited in prose twice as the source of *terminology*, not figures: `(Tanenbaum calls this Program Status Word)` (L11 s19) and `These addressing mode names are as per Tanenbaum "Structured Computer Organization"` (L12 s16).

Consequence for a new diagram: the block-diagram idiom the course leans on most is Mano's (five citations across four decks). If you redraw a block diagram and want it to look native, match Mano, not Floyd. Floyd appears in exactly one deck. See Inconsistencies — this is not a settled question.

Gate symbols and logic diagrams

Symbol family

Distinctive-shape symbols throughout — no IEC rectangles anywhere. AND is a flat-backed body with a semicircular nose; OR has a concave back and a pointed nose; NOT is a triangle with a bubble on its tip; NAND/NOR/XNOR are the base shape with a bubble on the output; XOR/XNOR add a second concave arc set back from the input edge.

Evidence: the definitional table at L02 s6–s8 (all seven gates), then every working diagram in the course — L02 s14, s18, s19, s22, s23; L03 s15; L04 s10, s14; L05 s10, s11, s13; L07 s3, s12; L09 s6, s7, s8, s10; L11 s6. Also the "Logic Diagram Symbol" column of *Device Pinouts and Symbols* p. 1–2.

Consistency: **holds without exception across all 221 deck pages and all three guides.** The course never names the family; the identification as ANSI/IEEE Std 91 distinctive-shape is mine, from the shapes.

Inversion bubble

A small **open** circle at the symbol boundary, always. On outputs it makes NAND/NOR/XNOR/NOT (L02 s6–s8). It is also legitimate on **inputs**, to express a De Morgan equivalent: L02 s22 draws OR-with-inverted-inputs = NAND, and AND-with-inverted-inputs = NOR, with bubbles sitting on the input leads at the body edge. Active-low control pins carry it too (74HCT157 `ENABLE`, *Device Pinouts* p. 2). *Consistency:* every occurrence, throughout.

Flow direction and wiring

- **Inputs on the left, outputs on the right.** Stated in words, twice: L02 s14 ("Inputs are conventionally on left" / "Outputs are conventionally on right") and the *Guide to Drawing Circuit Schematics* p. 1. Held in every logic diagram sampled.
- **Orthogonal routing only** — wires run horizontally and vertically, turning at right angles; no diagonal segments in any figure. (L02 s14, s18; L04 s14; L05 s13; L11 s6.)
- **Junction dot = connection. Crossing with no dot = no connection.** L02 s14 labels both cases explicitly on the same figure ("Indicates Electrical Connection", "No Connection"). Restated as error 2 of the schematics guide p. 2. *Consistency:* the single most-repeated drawing rule in the course.
- One line weight for all signals in a gate-level diagram. Thick lines appear only in datapath block diagrams (see below).

Labelling inside a logic diagram

- **Gate-type names are written above the symbol only in the definitional table** (L02 s6–s8: "AND", "OR", "NAND", "NOR", "XOR", "XNOR"; the NOT symbol on s6 is left unnamed). In every *working* diagram the shape alone carries the function — no type label. (L02 s14, s18, s19; L03 s15; L04 s14; L09 s6.)
- **Signal names**: single uppercase letters for primary inputs (`A`, `B`, `C`, `D`, `E`), and `X`, `M`, `F` or `Z` for outputs (L02 s6–s8, s14, s18; L03 s15; L04 s14). Complements carry an overline ($\overline{\texttt{A}}$).
- **Intermediate signals are labelled on the wire, immediately at the gate's output**, with the expression they carry: `AB`, `AC`, `AB + AC`, `B + C`, `A(B + C)` (L02 s19; L05 s16; L06 s5), or written above the wire ($\overline{\texttt{A}}$`BC`, `ABC`, `ABC`, `ABC` — L02 s18).
- **Numbered gates:** where a figure needs to refer to individual gates it puts a bare arabic numeral inside the gate body (L02 s18: inverters 1–3, ANDs 4–7, OR 8). **Seen once.** Extrapolate with caution.

- **Signal rails:** when one variable feeds many gates, it is run as a vertical rail crossing the whole figure, tapped with junction dots, and labelled at the top of the rail (L02 s18 — rails `A B C Ā B C`; L04 s14, the 3-to-8 decoder, same idiom). Two instances, two decks.
- **Sub-figure labels** are lowercase letters in parentheses, centred under each variant: `(a) (b) (c) (d)` (L02 s22; L05 s14, s15).

Block and MSI symbols

Multiplexer — trapezoid, wide edge left

Long parallel edge on the **left** carrying the data inputs stacked top-to-bottom (`D0 … D3`, or bare `0/1/2/3` when the inputs are anonymous); short parallel edge on the **right** with the single output; **select inputs enter through the bottom sloping edge**, labelled outside the body (`S1 S0`, `SEL`, `DIRN`, `F1 F0`).

Evidence: L04 s9, s10, s11, s12; L06 s12; L09 s6, s7, s10; L11 s6; *Device Pinouts* p. 2 (74HCT157, with `SEL` and `ENABLE` on the bottom edge). *Consistency:* **five decks plus the guide — solid.**

Decoder — trapezoid, wide edge right

The mirror flare: short edge on the **left** with the n inputs (`A B C`), long edge on the **right** with the 2 outputs (`D0 … D7`). (L04 s13.) *Consistency:* **one figure only.** The orientation is unmistakable in it, but it is a single instance.

Adder — trapezoid with a + in the body

Long edge left with the two operand groups and the carry-in at the top corner; short edge right with the sum group and the carry-out. A large `+` sits in the middle of the body. (L04 s7; *Device Pinouts* p. 2, 74HCT283.) The Floyd figure instead labels its adder block `A Σ / B C_out` (L07 s7) — see Inconsistencies.

D flip-flop — plain rectangle

`D` inside at the top-left, `Q` inside at the top-right, clock entering the **lower-left** edge, `Q` under `Q` when the complement is needed. The **triangular wedge (>) at the clock input means edge-triggered**; no wedge means a level-triggered latch; **a bubble in front of the wedge means falling-edge**, no bubble means rising-edge. L05 s14 states all four cases explicitly as figures (a)–(d) with the caption "Triangle indicates edge-triggered (therefore flip-flop)".

Evidence: L05 s5, s6, s14; L06 s4, s10, s11, s12; L07 s3, s8, s10, s11, s12; *Device Pinouts* p. 1–2. *Consistency:* the rectangle-with-wedge is universal; only the clock **label** varies (see Inconsistencies).

Asynchronous set/clear, when shown, are written `SET` above and `CLR` below inside the box (L07 s3, s12) and **may be omitted entirely when unused** — *Device Pinouts* p. 1 says so in words: "(Set and clear inputs can be omitted if not needed)". Active-low asynchronous inputs are marked by overlining the name (`PRE̅`, `CLR` — L05 s13) or by a bubble on the pin (*Device Pinouts* p. 1, 74HCT74). L06 s4 states the rule in prose: "The SET and CLR inputs are typically active-low".

Register — rectangle with a name plate

Rectangle with the word `Register` written **above** the box, input pin names down the left edge, output pin names down the right, a plain vertical line joining the first and last name of a contiguous range (meaning "and all the ones between"), enable name centred at the bottom inside, clock wedge on the left edge. (L06 s15, the 4-stage 8-bit queue.) **One figure.**

ALU

Two shapes, both used, both in the same deck — see Inconsistencies. The **notched trapezoid** (wide edge up, a V-notch cut into the top edge, operands entering the top, result leaving the bottom, `ALU` lettered in the body) is the more common: L09 s15, s16, s20; L10 s10. The **plain rectangle** with `ALU` centred and heavy arrows on all four sides is used at L09 s5 and s9.

Timing and waveform diagrams

See timing-diagrams[116] for what a timing diagram *means*; this is how it is drawn.

- **Time runs left to right.** Where the axis is marked at all it is a short solid horizontal arrow labelled `Time` placed under or to the right of the waveform block (L04 s15, s18). Most figures omit it (L04 s16, s17; L05 s6, s8).
- **One horizontal track per signal, stacked vertically.** The signal name sits to the **left** of its track, **followed by a colon**: `Input:`, `Output:`, `D:`, `CLK:`, `Q:`, `A:`, `B:`. (L04 s15, s16, s17; L05 s6, s8.) *Consistency:* every timing figure sampled.
- **Two horizontal dashed reference lines bound each track** — upper = logic 1, lower = logic 0. Where they are labelled, the labels read `Logic 1` and `Logic 0` and sit at the end of the rails (L04 s15, s18). In most figures the rails are drawn but left unlabelled (L04 s16, s17; L05 s6, s8).
- **The waveform is a heavy solid black line**, noticeably heavier than the dashed rails, riding on the lower rail for 0 and the upper for 1.
- **Idealised transitions are vertical** (zero rise/fall time): L04 s15, s16, s17; L05 s6, s8. **Sloped edges appear only in the one figure whose subject is real timing** — L04 s18, "Gates Aren't Perfect … The Reality of Timing", where the slope is the point.

[116] courses/csse2010/timing-diagrams.pdf

- **Input waveforms enumerate every input combination exactly once, in equal-width intervals.** L04 s16 states it: "All combinations of inputs are covered". For two inputs the pattern is A = `0 0 1 1`, B = `0 1 0 1` across four equal intervals (L04 s16, s17).
- **Measured intervals** are marked by dropping vertical dashed lines from the two instants and drawing a **red double-headed arrow** between them, labelled in **blue** (`Propagation delay`, `Fall time`, `Rise time`). L04 s18 — **one figure only**, so treat the colour choice as that figure's, not a rule.
- **Clock waveforms** come two ways, both consistent within their purpose: a **train of narrow rectangular pulses** (mark shorter than space, no dashed rails) when the clock is the whole subject (L05 s18; L06 s7), and a **full-height square wave on its own labelled track with rails** when it sits alongside data signals (L05 s6, s8).
- **A vertical dashed grid aligned to the clock edges** may be laid over a multi-signal figure so the waveforms can be read against each other (L05 s8). **One figure.**

State diagrams

L08 s9 is the one slide in the whole course that *states* a diagram convention rather than merely demonstrating one. Everything here comes from it, plus the worked example on s18.

- **A state is a circle divided by a horizontal chord.** The **state name** goes in the upper half (bold); the **output value(s)** go in the lower half. Nothing else goes in the circle. (L08 s9, which prints the empty template `State Name` / `Output Value(s)` next to two filled examples, `S0`/`101` and `Alex`/`01`.)
- **State names are free-form.** The same slide sanctions both a systematic `S0`, `S1`, `S2`, `S3` scheme (used in the worked example, L08 s18) and an arbitrary word (`Alex`). Outputs are written as a bare bit string with no separators or subscripts: `101`, `01`, `00`, `11`.
- **A transition is a plain single-headed arrow**, labelled with a **Boolean logic expression written above the arrow**. "If expression is true, state transition is made." (L08 s9.)
- **An unlabelled arrow means "true" — always transition.** Stated in words on L08 s9.
- **Special case, exactly one input:** the label degenerates to a bare `0` or `1`. L08 s9 boxes this case in red and annotates it "Special case for only one input".
- **Transition-label algebra is the ordinary Boolean notation** of L02 s12: overbar for NOT, `.` or juxtaposition for AND, `+` for OR. L08 s9's own examples are `Z`, `RE̅ SE̅T`, and `A+B.C`.
- **Completeness:** "Each possible combination of inputs should be addressed exactly once for each state — i.e. transition arrows from each state must encompass all possibilities (exactly once)." (L08 s14.) This is a correctness rule, not a drawing rule, but it constrains what you draw.

- **Layout:** the one worked diagram (L08 s18) puts the states in a **left-to-right row** with forward transitions between neighbours and the wrap-around transition drawn as a **long arc below the row**. Single instance — the row layout is an example, not established as required.

State tables

Two forms, both defined on their own slide, both with an empty template printed so the column structure is unambiguous.

One-dimensional (L08 s15) — one row per (current state, input) combination, annotated "Every combination of state and inputs":

Current State	Input U	Next State	Outputs	
			Q1	Q0

Two-dimensional (L08 s16) — one row per state ("One row per state"), with the next-state column split by input combination ("Every combination of inputs"):

Current State	Next State		Outputs	
	$\bar{U}$	U	Q1	Q0

Structural conventions common to both: headers bold and centred; a **heavy rule under the header block** and **heavy vertical rules between column groups**, thin rules inside a group; the individual output bits are sub-columns under a single spanning `Outputs` header; the input name appears in the header, not just its values.

For a counter with **no external input**, the same table is written with the next-state columns named after the flip-flop D inputs rather than the state: `Current State (Q , Q )` | `Next State (D , D )` (L07 s11). L08 s21 lists the design pipeline these tables sit in: state definition → state diagram → state table (1D or 2D) → state encoding → Boolean expressions → logic diagram → Logisim → hardware.

The characteristic table of a flip-flop uses different headers again: `D` | `Q(t+1)`, with the right-hand column glossed "What the output will be on the next clock edge" (L05 s7).

Block and datapath diagrams

- **Plain rectangles.** No rounded corners, no drop shadows, no fill in the line-art figures. The block name is centred inside, on one or two lines. (L04 s8; L05 s17; L06 s6; L07 s16; L08 s3, s7; L09 s15, s16.)
- **Flow left to right; feedback routed around the outside**, usually below the blocks, returning to the left edge of the first block (L05 s17; L06 s6; L07 s16; L08 s3, s7). *Consistency:* every sequential-circuit block diagram in the course, across four decks.
- **Every connection gets an arrowhead at its destination.**
- **The clock enters on its own labelled stub**, not bundled with the data inputs — `Clock pulses` entering from below-left in the Mano figures (L05 s17; L06 s6; L08 s3), `Clock` entering from above in the UQ redraw (L08 s7).
- **Multi-bit paths are drawn heavy, single-bit control thin.** L09 s15/s16 draw the A, B and Dest buses as thick black arrows against thin control lines; the caption "Buses (A,B,C)" points at them. L09 s5 and s9 draw the ALU's `A`, `B`, `Output` and `Function/Control` as heavy arrows with the width lettered beside them (`n`, `n`, `n`, `m`).
- **Grouped signals are collected with a curly brace and one text label** rather than labelled individually: `Data input A`, `Data output` (L04 s7); `Control registers`, `To and from memory` (L09 s15); `Before` / `Today` (L12 s2); the IO-board input/output groups (*Device Pinouts* p. 2).
- **Anonymous repetition uses a vertical ellipsis** between the first and last drawn line, with the count given in words beside the group: "n inputs", "m outputs" (L04 s8).
- **Bus width** is marked two ways — see Inconsistencies.

Register and memory layout diagrams

Bit fields

A single row of equal-width boxes, MSB on the left, LSB on the right. Bit positions are numbered **above** the boxes, descending left to right and ending at `0`. Values or field names go inside the boxes.

Evidence: L01 s12 (a byte, `0 1 0 1 0 1 1 1`), s13 (positions `9 8 7 … 0` above, powers `2 … 2` and decimal weights `512 … 1` inside), s16 (`MSB`/`LSB` called out by red arrows to the extreme boxes), s17; L10 s15. *Consistency:* every bit-field figure. L01 s16 states the underlying rule: "the MSB is worth 2^(n-1)".

The AVR-datasheet register diagram

For a named hardware register the three-row datasheet form is used: **bit numbers `7 … 0` above the row of boxes, flag names inside the boxes (bold), an `R/W` access row underneath, and an initial-value row (`0`) under that.** (L10 s15, the AVR status register

I T H S V N Z C.) One figure in the sampled decks, but it is verbatim datasheet form and is the obvious template for any other AVR register.

Memory maps

L11 s14, the one memory map with an explicit source (`[From ATmega324A datasheet]`):

- **Vertical stack of rectangles, lowest address at the top, highest at the bottom.**
- **Hex addresses written outside, to the right**, aligned with the region boundaries — as a range for a fully-specified region (`0x0000 - 0x001F`) or as the single start address of an open region (`0x0100`), with the final address at the bottom (`0x08FF`).
- **Hex is written `0x` + a fixed number of digits**, four for the address space (`0x0000`), two for the register file (`0x00` … `0x1F`, L11 s22).
- **Region names inside the boxes**, with the size in parentheses where useful (`Internal SRAM (1024/2048/4096/16384 x 8)`).
- **Grey fill for the large or coarsely-described regions, white for the enumerated ones.**
- **A diagonal break across the block** shows the drawing is not to scale (between Application Flash Section and Boot Flash Section).
- Column heading above each stack (`Data`, `Program Memory`).

The generic memory-organisation figure (L11 s5) agrees on direction — addresses `0 1 2 3 …` down the **left** side, increasing downward — and adds a double-headed arrow across the top labelled `Cell width`. **The alignment figure disagrees; see Inconsistencies.**

The register file

L11 s22: vertical stack, `R0` at the top through `R31` at the bottom, hex address to the right of each row, an ellipsis row `...` for the elided middle, a **heavy horizontal rule at the R15/R16 boundary** marking the split that matters for `ldi`, and free text further right for the X/Y/Z aliases (`X-register Low Byte`, etc.).

Shift and rotate

L12 s10, s11: **the printed figure is only a row of empty boxes**. The direction arrows inside the boxes, the `0` entering the vacated end, the separate small box labelled `C` for the carry, and the wrap-around arc for a rotate are all drawn in the lecturer's ink. Record this as the idiom the course *uses*, but note that **nothing about it is printed** — see Not established by the source.

AVR architecture figures

- **The AVR CPU architecture diagram (L10 s10) and the ATmega324A chip block diagram (L10 s11) are datasheet reproductions**, introduced only by the prose line "Some datasheet extracts follow" on L10 s9. Neither carries a caption. The only in-figure datasheet attribution anywhere is on the memory maps, L11 s14.
- **CPU diagram style:** black line art, no fill. The **8-bit data bus is a heavy vertical black bar** running the full height, labelled `Data Bus 8-bit` at the top. Blocks attach to the bus with **double-headed arrows**; instruction and control flow use **thin single-headed arrows**. The ALU inside is the notched trapezoid. Path labels that run vertically are set as **rotated text** (`Direct Addressing`, `Indirect Addressing`). Block names are two or three centred lines (`Flash Program Memory`, `32 x 8 General Purpose Registers`, `Instruction Decoder`, `Data SRAM`).
- **Chip-level diagram style (L10 s11):** a **dashed rectangle is the package boundary**, with pin and port names outside it (`VCC`, `GND`, `RESET`, `XTAL1`, `XTAL2`, `PA7..0`, `PB7..0`, `TOSC2/PC7`). A **heavy solid outline inside is the internal bus**, drawn with an arrowhead to show its direction. Peripherals are plain rectangles named for the function, with the width in parentheses for ports (`PORT A (8)`). Pin ranges are written `PA7..0`, high first.
- **The generic CPU figure the AVR specialises** (L09 s3, s4; L10 s3) is a heavy outer rectangle labelled `CPU` containing three filled boxes — `Control Unit`, `Arithmetic Logic Unit (ALU)`, `Registers` — with `Primary Memory` as a separate box outside it. This one *is* filled with colour (green, cyan, lilac), used only to separate adjacent blocks. See avr-architecture[117] and cpu-organisation[118].

Truth tables and function tables

- **Inputs first (left), outputs last (right), separated by a heavier vertical rule.** (L02 s6–s10; L03 s11, s14; L04 s11, s13; L05 s10.) *Consistency:* every truth table sampled.
- **Rows in ascending binary counting order of the input columns, starting from all-zeros**, all 2 of them. (Every table sampled.) L02 s13 states the underlying instruction: "Lists output for all 2 combinations of inputs — Best to list inputs in a systematic way".
- **Column headers are the bare signal names**, bold. No units, no types.
- **Function tables for MSI blocks** list only the selector combinations, and the output column holds the **name of the selected signal**, not a bit: `S1 S0 | F` with entries `D0`, `D1`, `D2`, `D3` (L04 s9, s11). The header is called out as `Function table` (L04 s9) or `Function Table` (L04 s11).
- **A highlighted cell or row** uses a pale fill — pale orange for the active decoder output (L04 s13), pale grey for the changing bits of a counting sequence (L07 s9).

[117] courses/csse2010/avr-architecture.pdf
[118] courses/csse2010/cpu-organisation.pdf

- **Visual styling of tables is not fixed** — see Inconsistencies. Only the column semantics and row ordering are stable.
- **Multi-column comparison tables** (Intel-64 vs AVR data-type support, L12 s18) use a check mark for "supported" and an empty cell for "not supported" — no cross, no "N/A".

Boolean and textual notation

L02 s12 is titled "Boolean Algebra conventions" and states them outright — the only place in the course where notation is legislated rather than demonstrated:

- **Inversion: overline.** `NOT(A) = Ā`, "pronounced as A bar".
- **AND: a dot, or implied by adjacency.** `AND(A,B) = AB = A.B`.
- **OR: a plus sign.** `OR(A,B,C) = A+B+C`.
- **XOR: .** `XOR(A,B) = A B = ĀB + AB`.
- The overline extends over exactly the sub-expression negated: `NAND(A,B,C) = A B C` as one bar over `ABC`; `NOR(A,B) = A + B` as one bar over `A + B`. Nested negation stacks bars (L02 s22: `A  + B`).

Held throughout the semester (L02 s16–s23; L03 s14; L04 s10; L07 s11, s14; L08 s9; L09 s11).

Beware the operator clash the course flags itself: in an arithmetic context `+` means addition, not OR — L04 s5 says " '+' in this case means addition (not OR)".

Indexed signals take subscripts: `Q` , `Q` , `D` , `D` , `S` , `S` , `A` …`A` , `C` , `F` , `F` (L03 s16; L04 s9; L07 s10–s12; L09 s6, s10). Not universally — see Inconsistencies.

Numbers

- **Radix by subscript:** `26` , `101` , `101` , `8  - 9` (L01 s19, s20; L09 s22).
- **Radix by prefix:** `0x` (C and AVR), `$` (AVR assembly), `0b` (binary), leading `0` (octal). L01 s20 lists every alternative in a table and marks these ones "We'll mostly use these", ruling out the trailing-`h`, trailing-`q`, trailing-`b`, `@` and `%` forms.
- **Binary literals are grouped in fours with a space:** `0000 1101 0111 0111`, `00001010`, `1110 KKKK dddd KKKK` (L01 s18; L03 s8, s9; L10 s4, s16–s18). Short literals under about 8 bits are often left ungrouped (`00001111`, L11 s16).

AVR assembly and machine code

- **Machine-code layouts write fixed bits as `0/1` and each variable field as a repeated letter, one letter per bit** — `0000 11rd dddd rrrr` (ADD), `1001 010d dddd 0000` (COM), `1110 KKKK dddd KKKK` (LDI). `d` = destination register, `r` = source

register, `K` = constant. Note the case split: register-field letters lowercase, the constant uppercase. (L10 s16, s17, s18.)
- **Register-transfer notation uses a left arrow:** `r3 ← r3 + r4`, `r22 ← r22 + 1` (L10 s8, s12, s13).
- **Multi-byte quantities are written as a `high:low` register pair:** `r5:r4`, `r11:r10`, `r13:r14`, `R31:R30` (L11 s22; L12 s12, s20). High byte first, always.
- **Code is set in a monospace font**, and concrete examples in the body text are coloured red (`mov r3, r14`, `add r5,r6`, `ldi r16,73` — L10 s12–s14).
- **Mnemonic case is not settled** — see Inconsistencies.

Colour and annotation

Colour carries no defined meaning anywhere in this course. No slide and none of the three guides ever says what a colour signifies. What can be observed:

- **Red is the usual colour for a callout pointing into a figure** — an arrow plus a short label, sitting outside the drawing (L01 s13 `Bit position`, `Value (Powers of 2)`; L01 s16 `MSB`/`LSB`; L02 s14 the four connection callouts; L04 s4 `Full-adders`; L09 s5 `Function/Control`; L09 s15 `Buses (A,B,C)`; L11 s5 `Address/Cell width`).
- **Blue is the usual colour for an explanatory label attached to a figure element** (L04 s15 `Logic 1`/`Logic 0`, s18 `Propagation delay`; L05 s15, s17 `Logic Gates`; L09 s10 `Carry ripples through from right to left`).
- **But the two are swapped freely.** L04 s15 uses red for `Change in input causes output to change`; L08 s15/s16 use red for both the brace and its explanation; L09 s20 uses red for the `Status` callout with a green-filled fact box beside it; L09 s15 uses purple.
- **Fills are decorative,** used only to separate adjacent blocks: yellow for the levels-of-abstraction stack (L01 s9; L02 s5), green/cyan/lilac for the three CPU parts (L09 s3, s4; L10 s3), pale blue/green/lilac for the fetch-decode-execute fact boxes (L09 s13).
- **Table highlighting** is a pale fill on the cell or row of interest (L04 s13; L07 s9).

Handwritten ink on the slides is the lecturer's live annotation, not part of the published figure. It is present on most technical slides, in red, blue, purple, green and orange, and the colour there tracks nothing but which pen was in hand — L11 s13 uses three ink colours in one worked example purely to separate three sub-workings. **Nothing that exists only in ink should be treated as an established figure convention**; where a convention below is ink-only it is flagged as such and repeated under *Not established*.

Inconsistencies

Places where the course does the same thing more than one way. **None of these is resolved here — a decision is needed before a new diagram commits to one.**

1. **The ALU has two symbol shapes, in one deck.** A **plain rectangle** with `ALU` centred and heavy arrows on four sides (L09 s5, s9), and a **notched trapezoid** with operands in the top and result out the bottom (L09 s15, s16, s20; also the AVR datasheet figure, L10 s10). Both are the ALU of the same machine, ten slides apart. *Decision needed.*

2. **The flip-flop clock input has four labels.** `CLK` (L05 s5, s6; L06 s4; L07 s10, s11, s12; *Device Pinouts* p. 2 on the 74HCT175), `CK` (L05 s14 — the very slide titled "D Latches and Flip-flops — Symbols Used"), italic serif `C` (L06 s8, s9 — the Mano figures), and lowercase italic `clk` as the net name on the wire (L06 s10, s11; L07 s15). *Decision needed.*

3. **The asynchronous inputs have three naming schemes.** `SET` / `CLR` (*Device Pinouts* p. 1, 74HCT74 schematic symbol; L07 s3, s12), `PR̄Ē` (preset) / `CLR` (L05 s13), and `R` with a bubble (Mano's register, L06 s8). *Device Pinouts* p. 2 gives the 74HCT175 only a `CLR`. *Decision needed.*

4. **"Current state" vs "present state", and "next state" vs "Q(t+1)".** Printed slides say `Current State` (L07 s11; L08 s15, s16); the lecturer's worked tables say `Present State` (L07 s8, s14 — ink). The flip-flop characteristic table calls the next-state column `Q(t+1)` (L05 s7) where the state table calls it `Next State` (L07 s11; L08 s15). *Decision needed.*

5. **Three textbook figure styles coexist, none declared canonical. Mano**, *Digital Design* 3rd ed. — black-and-white line art, thin rules, italic serif signal names with subscripts (`B` , `A` , `C` , `S` , `SI`, `SO`, `CLK`), plain square blocks with italic `FA` inside (L03 s16; L04 s4, s10; L05 s17; L06 s6, s8, s9; L08 s3). **Floyd**, *Digital Fundamentals* — pale-blue and beige filled blocks, magenta arrows and magenta text labels, serif `(a) Serial in/shift right/serial out` sub-captions (L07 s4, s5, s7). **Tanenbaum**, *Structured Computer Organization* — dense line art with dashed boxes grouping sub-circuits (L09 s8). And L01 s8 says there is no textbook. *Decision needed:* which family a redrawn figure should imitate.

6. **The same block diagram exists in two drawing styles.** The sequential-circuit block diagram appears as the **Mano original** — thin rules, serif lowercase labels `Combinational circuit` / `Flip-flops` / `Clock pulses` (L05 s17; L06 s6; L07 s16; L08 s3) — and as a **UQ redraw** in bold sans-serif with heavy borders (L08 s7, the Moore/Mealy slide). Same content, same course, different look. *Decision needed.*

7. **Memory addresses increase downward in some figures and upward in others.** Downward: the generic cell diagram (L11 s5, addresses `0 1 2 3 …` down the left), the AVR address-space maps (L11 s14, `0x0000` at the top), the register file (L11 s22, `R0` at the top), the endianness examples (L11 s16, s17). Upward: the alignment figure (L11 s10), where the row addresses `0 8 16 24` run bottom to top and the byte numbers within a row run high-to-low left to right (`15 14 13 12 11 10 9 8`). *Decision needed.*

8. **Hex is prefixed `0x` in some places and `$` in others — sometimes on the same slide.** `0x`: L10 s15 area, L11 s14 (`0x0000`), L11 s22 (`0x1F`), L11 s27 ("0 to 0x3F"), L12 s15 (`andi r17, 0xBA`). `$`: L11 s26 ("$20 to $FF"), L12 s15 (`lds r15, $1234`). L01 s20 blesses both. *Decision needed.*

9. **Assembly mnemonic case in the generic form.** Uppercase: `MOV rd, rr`, `ADD rd, rr`, `AND rd, rr`, `LDI rd, K` (L10 s12, s13, s14; L11 s25; L12 s4, s9). Lowercase bold-italic: `in rd, P`, `out P, rr` (L11 s27; L12 s5), and lowercase throughout the addressing-modes slide (`andi`, `and`, `lds`, `ld` — L12 s15, s16). Concrete examples are lowercase in print (`mov r3, r14`) but uppercase in the lecturer's ink (`ADD r10, r4`, L12 s20). *Decision needed.*

10. **Register names: `R0`–`R31` or `r0`–`r31`.** Uppercase in the register-file figure (L11 s22) and the AVR architecture block (L10 s10, `32 x 8 General Purpose Registers`); lowercase everywhere in prose and code (L10 s12–s14; L11 s27; L12 s4–s6). *Decision needed.*

11. **Truth-table visual styling has three treatments.** Dark-red header row with pink zebra striping (L02 s9, s10; L03 s11; L04 s11); plain black-on-white with a bold header and a heavy input/output divider (L02 s6, s7; L03 s14; L04 s13; L05 s10); light-grey header with grey cell highlighting (L07 s9). Column semantics and row order are stable across all three; only the appearance differs. *Decision needed for a house table style.*

12. **Bus width is marked two ways.** A **short slash across the line with the width beside it** (*Device Pinouts* p. 2, the `4`-marked buses of the 74HCT157 and 74HCT283 logic-diagram symbols). Or the **width simply lettered next to a plain arrow, no slash** (L09 s5, s9 — `n`, `n`, `m`, `n`). *Decision needed.*

13. **The adder's function glyph.** A large `+` in the trapezoid body (L04 s7; *Device Pinouts* p. 2). Or `Σ` on the block's output pin with `A`, `B`, `C_out` on the others (L07 s7, the Floyd figure). *Decision needed.*

14. **Indexed signals are subscripted inconsistently.** Subscripted `Q` , `Q` , `D` , `D` in the counter state table and its circuit (L07 s10, s11, s12). Unsubscripted `Q1`, `Q0` in the state-table templates (L08 s15, s16) and `S0`, `D0` in the mux function table (L04 s9). *Decision needed.*

Not established by the source

Things a CSSE2010 diagram may well need that the materials simply do not settle. **These are where invention risk lives.**

- **Karnaugh maps: entirely absent.** No K-map appears on any of the 221 deck pages sampled, nor in any of the three guides. The course's simplification path runs truth table → sum of products → Boolean identities → algebraic manipulation (L02 s16–s21), and De Morgan is done algebraically (L02 s21, s22). If a K-map is needed, **everything about it is invention**: cell ordering and whether the axes are Gray-coded, which variable goes

on which axis, how variable labels are written along the edges, the style of the grouping loops, and the don't-care mark.

- **The initial/reset state of a state diagram.** L08 s9 defines only the state circle and the transition arrow. The one worked diagram (s18) has no incoming arrow from nowhere, no double circle, no `reset` label. **How to mark the start state is unspecified.**
- **Mealy outputs on a state diagram.** L08 s6, s7 define both machine types, then s7 says "We'll stick to Moore machines in this course". The notation slide therefore supplies only the two-half circle, which is a Moore output. **There is no notation for an output written on a transition** (no `input/output` slash form anywhere).
- **Self-loop transitions.** None is drawn anywhere sampled, although the completeness rule (L08 s14) guarantees most machines need them. Loop shape, side of the circle, and label placement are all unspecified.
- **The don't-care symbol.** `X` is used for don't-care next-state entries at L07 s14 and for "byte not part of this word" at L11 s10 — **both in the lecturer's ink, never in print.** No printed definition of the symbol exists.
- **The shift/rotate register idiom.** L12 s10, s11 print only a row of empty boxes; the direction arrows, the `0` entering the vacated end, the separate `C` carry box and the rotate wrap-arc are all ink. The idiom is clear, but it has **no printed authority**.
- **Numeric values written inside timing-diagram segments.** The `0/1` written in the middle of each waveform segment is always the lecturer's ink (L04 s15, s16, s17; L05 s6), never printed. Whether a finished diagram should carry them is unsettled.
- **Setup and hold time.** Only propagation delay, rise time and fall time are named and drawn (L04 s18). Setup/hold appear nowhere, so their marking is unspecified.
- **Waveforms for multi-bit buses.** The "hexagon" (split-and-rejoin) idiom for a bus carrying a value is never drawn. Every waveform in the course is a single-bit two-level trace.
- **Whether a multi-bit signal in a gate-level logic diagram should be thickened or slashed.** The datapath figures use heavy arrows (L09 s15, s16) and the guide's MSI symbols use slashes (*Device Pinouts* p. 2), but gate-level diagrams use exactly one line weight and no slashes. Nothing bridges the two.
- **Tri-state, open-collector and bidirectional signal marking.** Not drawn anywhere. The AVR datasheet figures use double-headed arrows for bidirectional bus attachment (L10 s10, s11), but that idiom is inherited, not taught.
- **Symbols for gates with three or more inputs.** L02 s9, s10 ask for 3-input NAND and XOR truth tables but never draw a 3-input symbol, and the parts list contains only 2-input gates plus the hex inverter (*Device Pinouts* p. 1; see device-pinouts[119]). How to extend the body and space the input leads is unspecified.
- **Figure numbering and captioning.** The decks caption almost nothing. The only figure caption in any deck is Mano's own, retained inside a reproduced image: `Fig. 6-3 4-Bit Shift Register` (L06 s9). Source credits sit as a bracketed line under the figure. **There is no course figure-numbering scheme**, so any new note must choose its own.

[119] courses/csse2010/device-pinouts.pdf

- **Stroke weights, arrowhead geometry, fonts and sizes.** No slide or guide specifies any of these numerically.
- **What colour means.** Nothing in any slide or guide assigns meaning to a colour. The near-patterns recorded above are observations, not rules — and they are broken more than once. **Do not attach semantic meaning to colour in a new diagram** unless you define it in that diagram's own caption.

See also

- circuit-schematics[120] — the schematic rules this note deliberately does not repeat: device IDs, gate letters, pin numbers, junction dots, power-supply blocks, the seven common errors
- device-pinouts[121] — 74-series pinouts, the logic-diagram vs circuit-schematic symbol columns, the IO board
- logic-gates[122] — the seven gate types and their truth tables
- timing-diagrams[123] — what a timing diagram means, as opposed to how it is drawn
- state-machines[124] — state diagrams and state tables as content
- boolean-algebra[125] — the identities the transition-label algebra uses
- avr-architecture[126] — the AVR figures described here, read for content

Device Pinouts

Pin numbers for the 74-series logic chips and IO board used in circuit-schematics[127]. All quad 2-input gate chips (`74HCT00`, `08`, `32`, `86`) share the **same** pin layout — only the gate function differs.

> **Source:** *CSSE2010/CSSE7201 Device Pinouts and Symbols* (Blackboard, v4). This is the whole parts list: **every gate chip in it is 2-input** (plus the hex inverter), so a design needing a 3-or-more-input gate has to be built up from 2-input ones. There is no 3-input AND/OR/NAND in the kit.

The source sheet gives up to three things per device, and they are not interchangeable:

[120] courses/csse2010/circuit-schematics.pdf
[121] courses/csse2010/device-pinouts.pdf
[122] courses/csse2010/logic-gates.pdf
[123] courses/csse2010/timing-diagrams.pdf
[124] courses/csse2010/state-machines.pdf
[125] courses/csse2010/boolean-algebra.pdf
[126] courses/csse2010/avr-architecture.pdf
[127] courses/csse2010/circuit-schematics.pdf

Column	What it is	Where it is used
Device pinout	the physical package: which signal is on which numbered pin	wiring the breadboard; checking a schematic's pin numbers
Logic diagram symbol	the hardware-independent shape	logic-gates[128] and logic diagrams
Circuit schematic symbol	the shape *plus* device ID, device type and pin numbers	circuit-schematics[129]

Power and ground

Power and ground connections for devices are shown **separately** from the gate pins on a schematic, once per device type. The symbols below are also what you use to tie a device input directly to logic low or logic high.

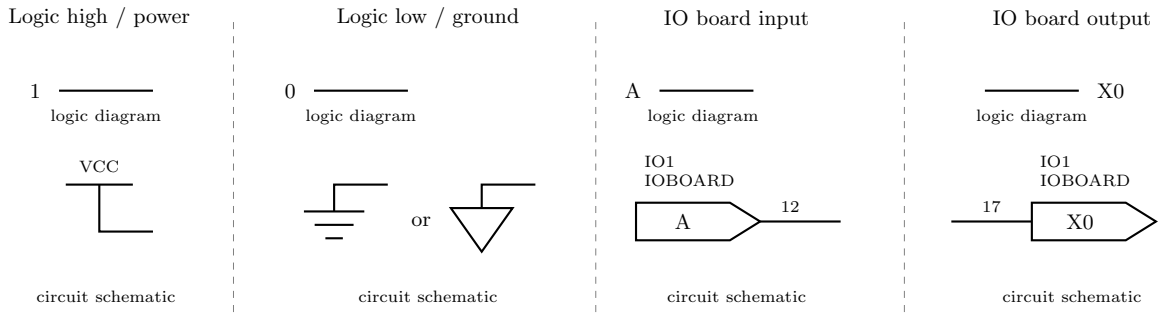


Figure 34: The four non-gate symbols. Logic high and logic low are a bare 1 or 0 on a logic diagram, and a VCC bar or a ground symbol (either of the two accepted forms) on a circuit schematic. The IO board input and output symbols are the same pentagon; the difference is that an input's wire leaves the point and an output's wire arrives at the flat back.

Quad 2-input gates (74HCT00 / 08 / 32 / 86)

[128] courses/csse2010/logic-gates.pdf
[129] courses/csse2010/circuit-schematics.pdf

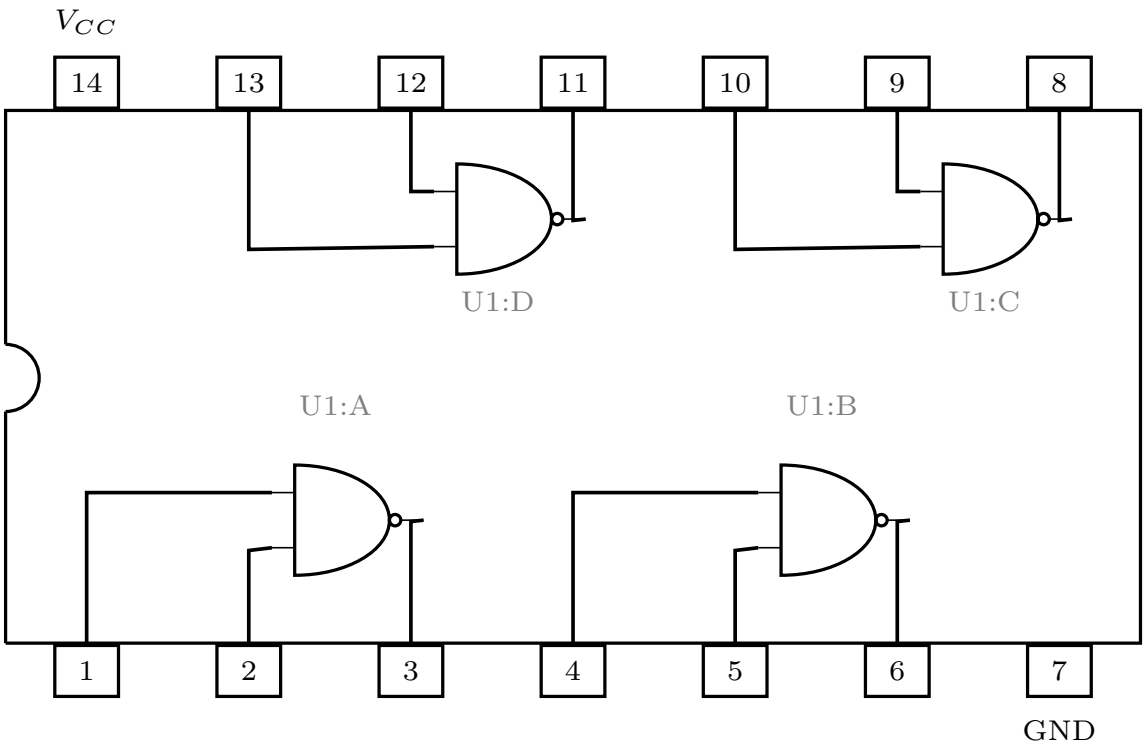


Figure 35: The 14-pin DIP pinout, drawn for the 74HCT00 but identical for the 08, 32 and 86 — only the gate shape inside changes. The notch marks the pin-1 end; pins run 1-7 along the bottom and 14-8 along the top. Reading the internal wiring is what tells you the gates pair pins 1,2 to 3, 4,5 to 6, 9,10 to 8 and 12,13 to 11, and that Vcc (14) and GND (7) are the two pins with no gate on them.

Chip	Function	Gate A	Gate B	Gate C	Gate D	VCC	GND
74HCT00	Quad 2-input NAND	in 1,2 → out 3	in 4,5 → out 6	in 9,10 → out 8	in 12,13 → out 11	14	7
74HCT08	Quad 2-input AND	in 1,2 → out 3	in 4,5 → out 6	in 9,10 → out 8	in 12,13 → out 11	14	7
74HCT32	Quad 2-input OR	in 1,2 → out 3	in 4,5 → out 6	in 9,10 → out 8	in 12,13 → out 11	14	7
74HCT86	Quad 2-input XOR	in 1,2 → out 3	in 4,5 → out 6	in 9,10 → out 8	in 12,13 → out 11	14	7

The letters `A`–`D` are **yours to assign** — the chip does not label its own gates. What is fixed is which *pins* belong to the same gate. Whichever slot you call `U1:A`, its pin numbers must be one of the four sets above.

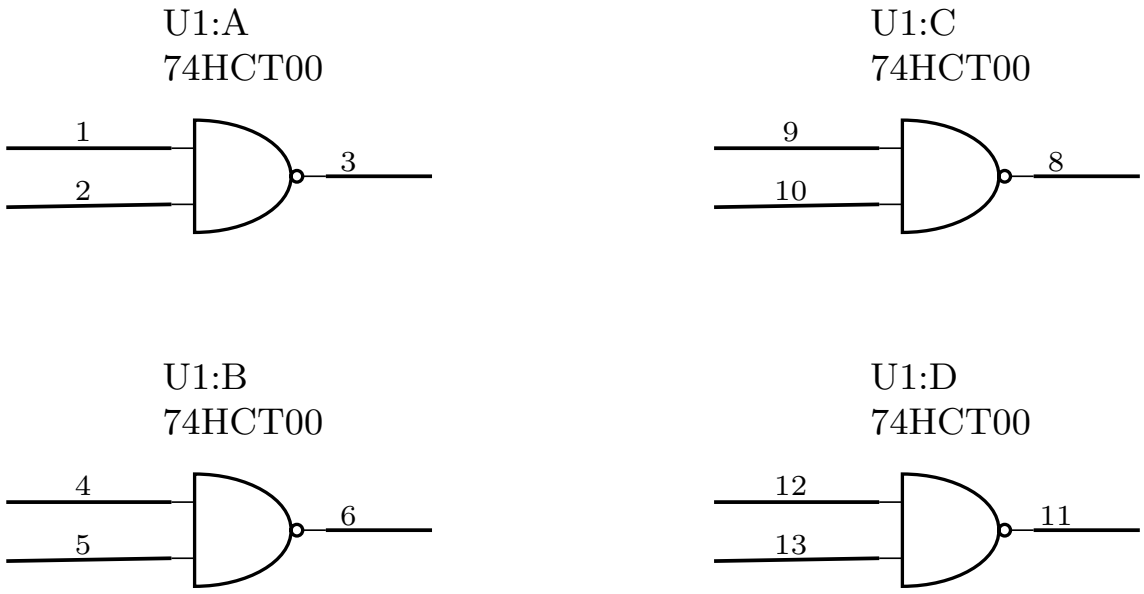


Figure 36: The four gates of a 74HCT00 as circuit schematic symbols: device ID, device type, and a pin number on every terminal. For a 74HCT08, 74HCT32 or 74HCT86 the pin numbers are exactly these; only the gate body changes to AND, OR or XOR.

74HCT04 — Hex inverter

Six independent NOT gates. VCC = pin 14, GND = pin 7.

Gate	A	B	C	D	E	F
in → out	1→2	3→4	5→6	9→8	11→10	13→12

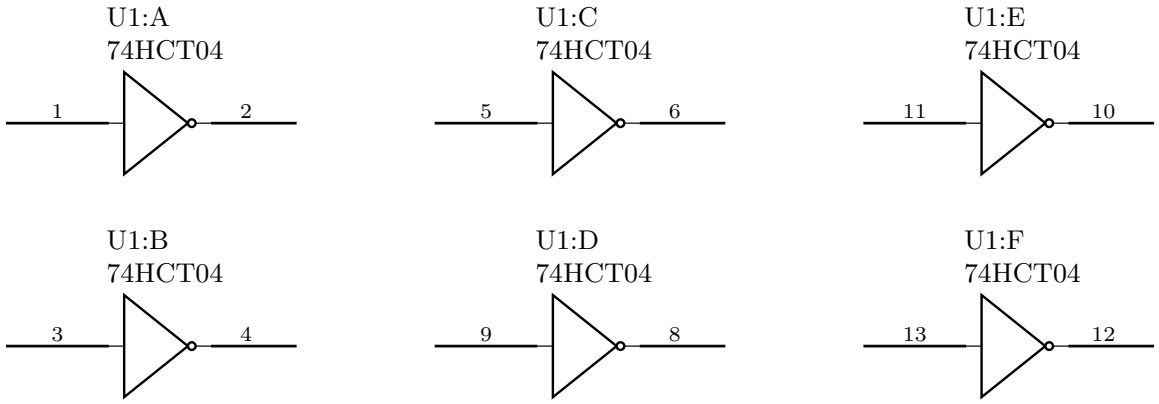


Figure 37: All six inverters of a 74HCT04. The numbering is not a simple run: the first three gates go low-to-high (1 to 2, 3 to 4, 5 to 6) but the last three run the other way (9 to 8, 11 to 10, 13 to 12), because the package is symmetric about its centre.

Note the direction reversal in the second half of the package — `U1:D` is pin **9 in, 8 out**, not 8 in, 9 out. Getting this backwards is an easy way to lose marks on a schematic.

74HCT74 — Dual D-type flip-flop (14-pin)

Pin	1	2	3	4	5	6	7	8	9	10	11	12	13	14
Signal	CLR1	D1	CLK1	SET1	Q1	$\overline{Q1}$	GND	$\overline{Q2}$	Q2	SET2	CLK2	D2	CLR2	VCC

Set and clear inputs can be **omitted** from the schematic if not needed — the source sheet says so explicitly, and its own logic-diagram symbol is drawn without them.

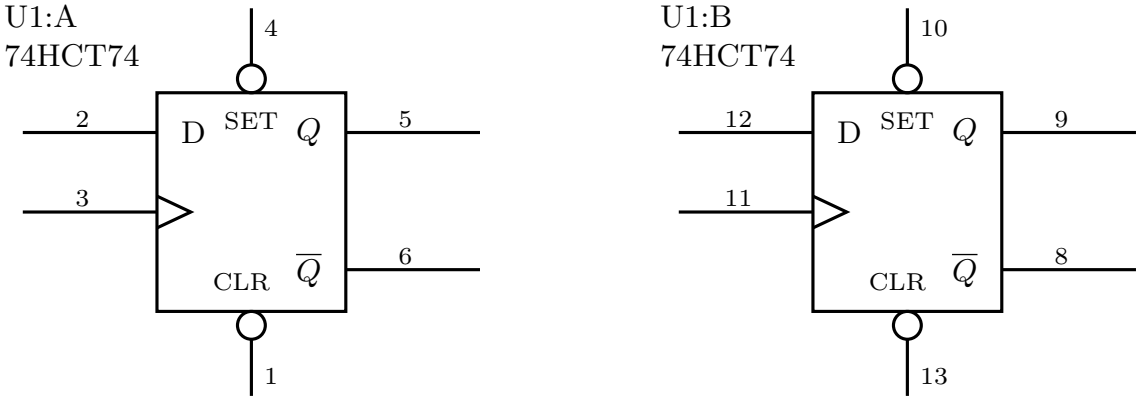


Figure 38: The two flip-flops of a 74HCT74 as circuit schematic symbols. Unlike the 74HCT175 below, each flip-flop has its own clock, its own set and its own clear — nothing is shared. SET and CLR carry bubbles, and the wedge on the left-hand lower input marks the clock.

74HCT157 — Quad 2-input multiplexer (16-pin)

Pin	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16
Signal	SEL	A0	B0	Y0	A1	B1	Y1	GND	Y2	B2	A2	Y3	B3	A3	$\overline{\text{ENABLE}}$	VCC

SEL (pin 1) and ENABLE (pin 15) are common to all four multiplexers — one select line for the whole package. That is what makes this chip a 4-bit-wide 2-to-1 mux rather than four independent ones. ENABLE is **active low** — the source sheet writes it $\overline{\text{ENABLE}}$ and draws a bubble on that pin.

At the logic-diagram level the source sheet allows **several options**: one wide block with `A`, `B`, `Y` buses and slashes marking the 4-bit widths, or four separate 2-to-1 mux symbols drawn side by side, with the enable input added only if it is needed. See combinational-logic-blocks[130] for what a multiplexer does.

74HCT175 — Quad D-type flip-flop (16-pin)

Pin	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16
Signal	$\overline{\text{CLR}}$	Q0	$\overline{Q0}$	D0	D1	$\overline{Q1}$	Q1	GND	CLK	Q2	$\overline{Q2}$	D2	D3	$\overline{Q3}$	Q3	VCC

The ordering is not uniform — check it before wiring. The package is mirror-symmetric about each D-pair, so the true outputs sit at pins **2, 7, 10, 15** and the complements at **3, 6, 11, 14**. Reading "Q then $\overline{Q}$" left-to-right across all four flip-flops gets flip-flops 1 and 3 backwards. The source's own schematic symbol settles it: D1 (pin 5) → Q1 (pin **7**), D3 (pin 13) → Q3 (pin **15**).

CLK (pin 9) and CLR (pin 1) are common to all four flip-flops — one clock and one clear for the whole package. The source sheet's logic-diagram column shows this two ways: as a single D flip-flop with 4-bit-wide D and Q buses, **or** as four separate flip-flops whose clock inputs are all tied together.

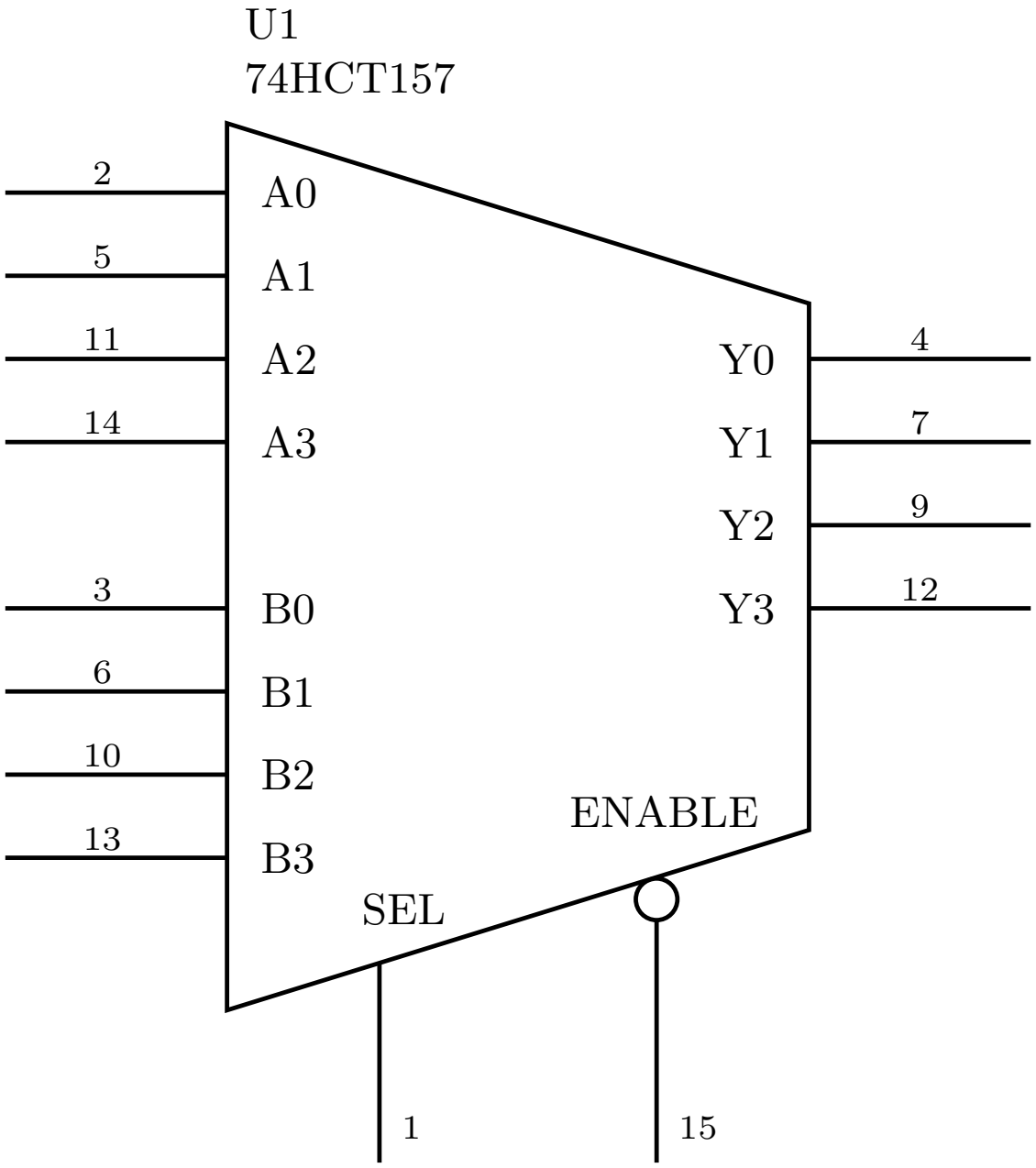


Figure 39: The 74HCT157 as one circuit schematic symbol — the multiplexer trapezoid, tall edge for the eight data inputs and short edge for the four outputs. It is four 2-to-1 multiplexers sharing a single SEL line and a single active-low ENABLE, so it selects between two whole 4-bit words rather than between two individual bits: one value of SEL puts A0-A3 on Y0-Y3 and the other puts B0-B3 there.

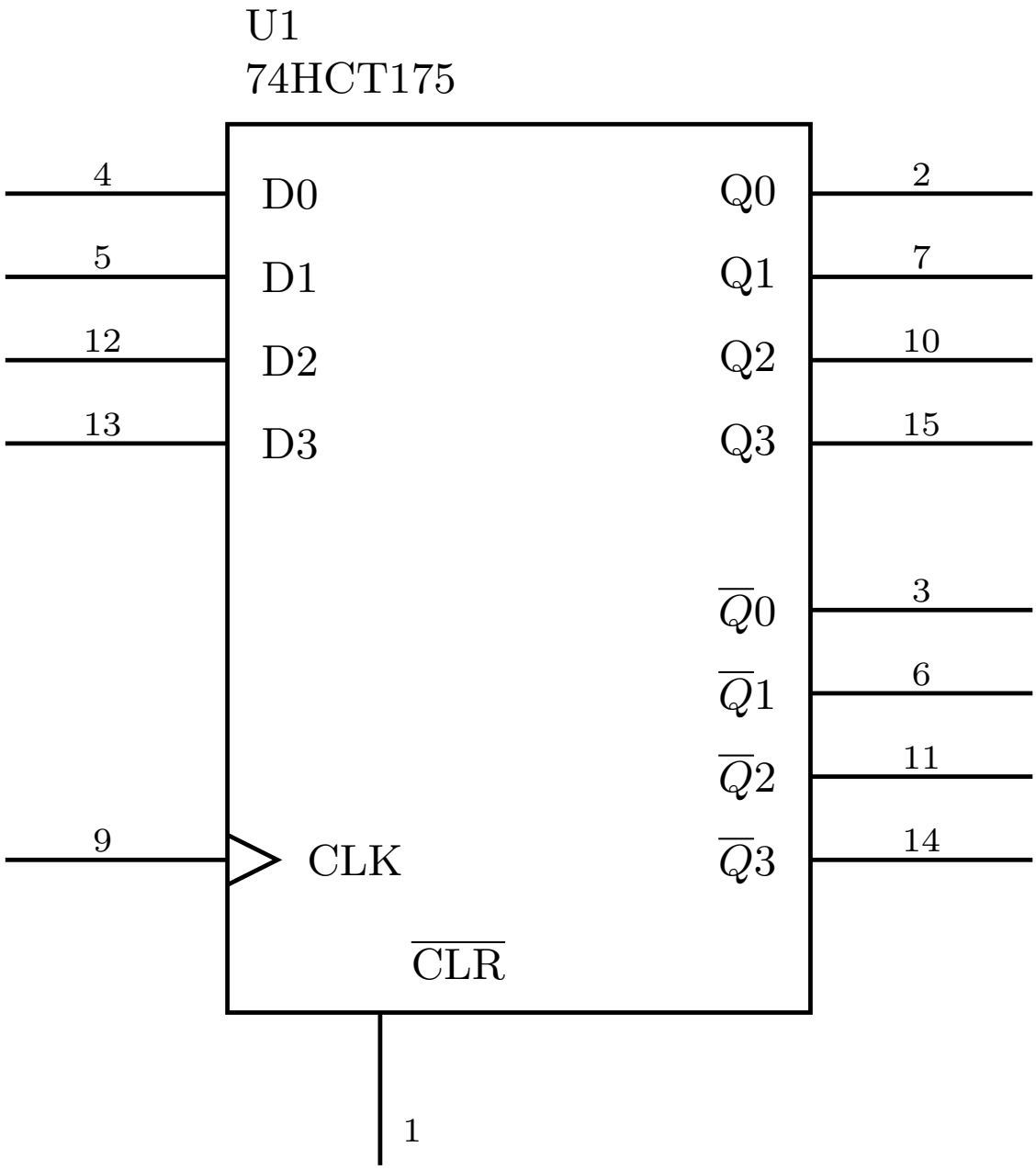


Figure 40: The 74HCT175 as one circuit schematic symbol. CLK (pin 9) and CLR (pin 1) enter once and serve all four flip-flops, which is exactly what a flat pin table hides: this is a 4-bit register, not four independent flip-flops.

Pin	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16
Signal	S1	B1	A1	S0	A0	B0	C0	GND	C4	S3	B3	A3	S2	A2	B2	VCC

$C0$ is the carry-in, $C4$ the carry-out; A_i, B_i are the two 4-bit operands and S_i the 4-bit sum.

The source sheet's logic-diagram column shows the internals as **four cascaded full adders**, with the intermediate carries C_1, C_2, C_3 entirely inside the chip. It also offers a compact alternative symbol with A, B and S drawn as 4-bit buses. This chip is the adder half of the adder-subtractor in combinational-logic-blocks[131].

IO board

Pins	Function
1	TX
2	RX
3	SSD cathode
4	SSD decimal point
5–11	SSD segments G to A
12–15	Buttons B3 to B0
16	Logic probe
17–24	LEDs L0 to L7
25–32	Switches S0 to S7

On a schematic the IO board is labelled `IO1` (`IO2` for a second board), with inputs using the point-attached symbol and outputs the back-attached symbol — see Figure 34 and circuit-schematics[132]. Use sensible signal names rather than the raw pin labels where possible: the source sheet's own example names buttons `B3`–`B0` on pins 12–15, switches `A`–`H` on pins 25–32, and LED outputs `X0`–`X5`, `COUT`, `OFLOW` on pins 17–24. There is no need to show power supply connections for the IO board.

Buttons and switches are **inputs** to your circuit; LEDs and the seven-segment display are **outputs**. See io-board[133] for what the board does electrically.

[131] courses/csse2010/combinational-logic-blocks.pdf
[132] courses/csse2010/circuit-schematics.pdf
[133] courses/csse2010/io-board.pdf

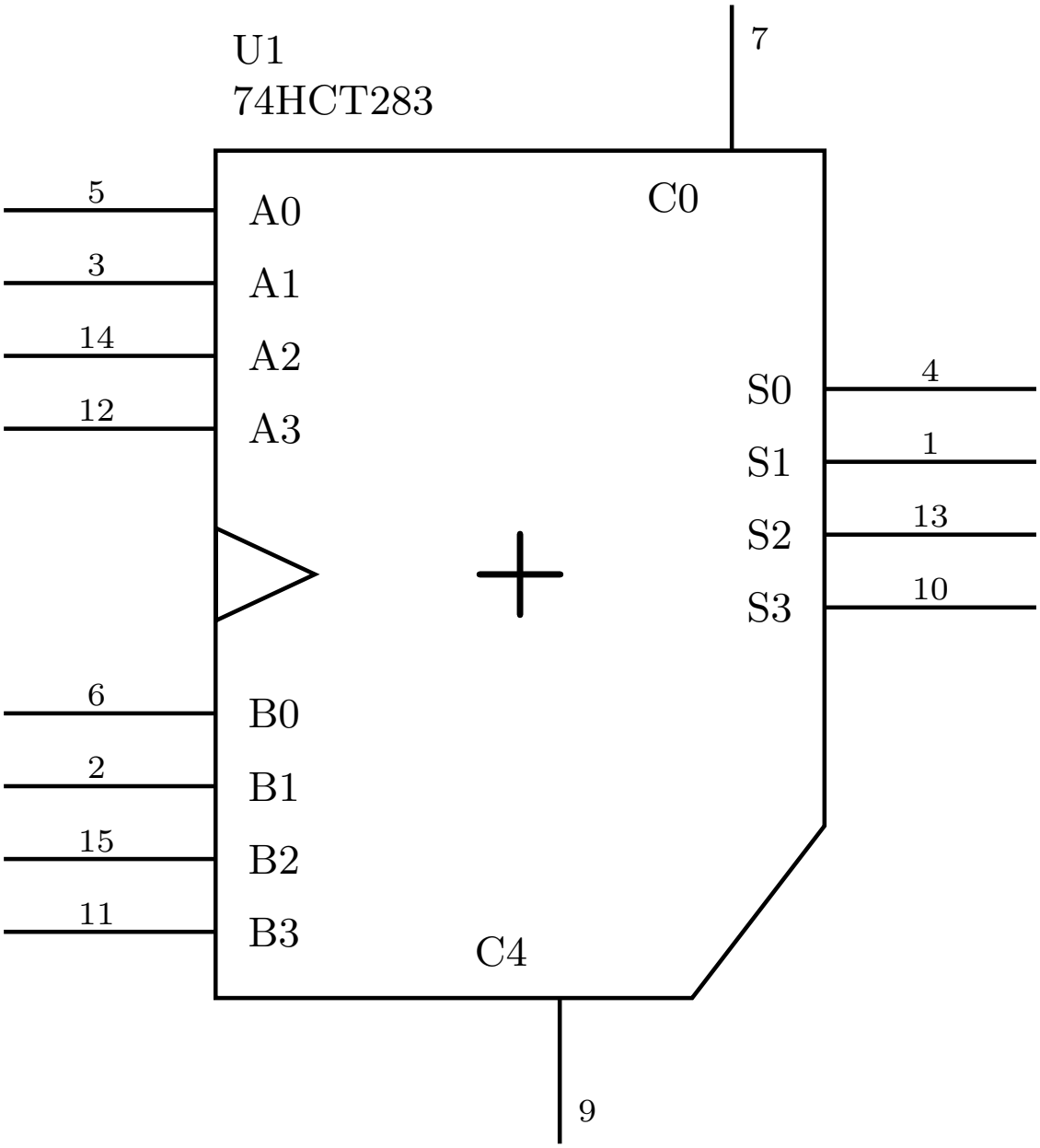


Figure 41: The 74HCT283 as one circuit schematic symbol. Internally it is four full adders with C1, C2 and C3 wired stage to stage inside the package, so only C0 and C4 are brought out to pins.

Flip-Flops and Latches

Flip-flops and latches are the **storage elements** — the circuits that let a design remember a value even after the input that produced it has changed. They're what turns a combinational circuit into a sequential circuit[134].

Introduced in Lecture 5, 2026-08-10-introduction-to-sequential-circuits[135].

The D flip-flop

The D flip-flop is the **only** flip-flop type covered in this course. Its symbol is a rectangle with:

- **D** — the data input, on the left.
- **Q** — the output, on the right.
- **CLK** — the clock, a *control* input on the left, drawn with a small triangular wedge where it meets the box.

How it works: **Q copies the value of D (and remembers it) whenever CLK goes from 0 to 1** — the *rising edge*. In between clock edges the flip-flop is not responsive, so it holds the stored value. Q changes **only** on that 0→1 transition.

Summary: a D flip-flop remembers a single bit — either a "1" or a "0". It keeps that value until the next clock edge, at which point the D input is transferred to output Q.

- To remember n bits you need n D flip-flops.
- An **n-bit register** is (by definition) n D flip-flops. See shift-registers[136].
- Flip-flops can themselves be made out of logic gates — see the SR latch and A real D flip-flop below.

Characteristic table

A **characteristic table** defines the operation of a flip-flop in tabular form. The left column is the input; the right column, $Q(t+1)$, is the *next* value of Q: what the output will be **on the next clock edge** (i.e. when CLK goes 0→1).

D	Q(t+1)	
0	0	Reset
1	1	Set

[134] courses/csse2010/sequential-circuits.pdf
[135] courses/csse2010/2026-08-10-introduction-to-sequential-circuits.pdf
[136] courses/csse2010/shift-registers.pdf

That is the whole behaviour: the flip-flop is *reset* (to 0) or *set* (to 1) according to D, at the clock edge.

A flip-flop's next-state column is written $Q(t+1)$ here. The state tables of state-machines[137] and [counters] call the same thing *Next State*. The course uses both names (see csse2010-diagram-notation[138]).

Worked waveform

D is 0, 0, then 1, 1 across a clock with two rising edges. CLK starts low, pulses high, goes low, then pulses high again after D has risen.

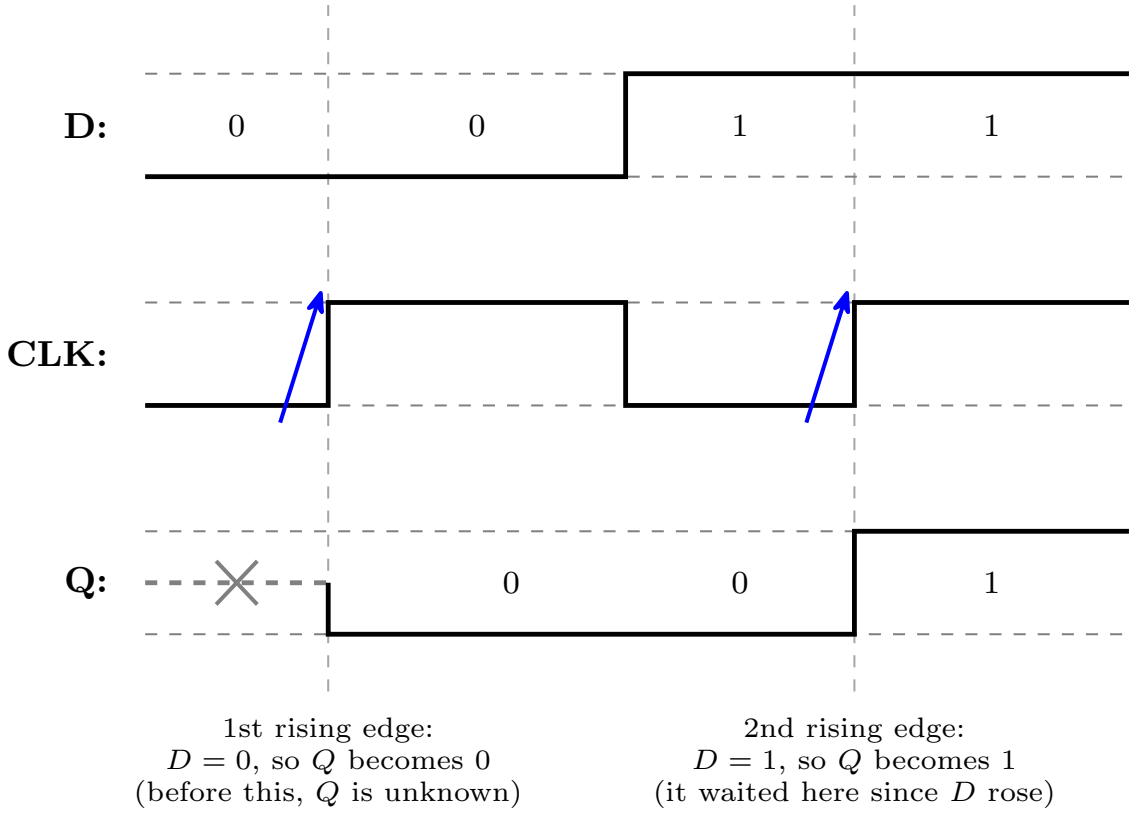


Figure 42: A D flip-flop waveform. The dashed verticals are the two rising clock edges, and the arrows mark them. Q is undefined (drawn dashed, with a cross) until the first edge. The point of the figure is the middle: D rises well before the second edge, and Q does not follow until that edge arrives.

[137] courses/csse2010/state-machines.pdf
[138] courses/csse2010/csse2010-diagram-notation.pdf

Clock edge	D at that edge	Q after
(before the first edge)	—	unknown/undefined
1st rising edge	0	0
2nd rising edge	1	1

Before the first rising edge Q is undefined, because nothing has been clocked in yet; an "X" marks that stretch. Between the two rising edges Q stays 0 even though D has already risen to 1: the flip-flop is only sensitive at the edge. Remember: Q changes **only** when the clock (CLK) goes from 0 to 1.

A second, harder worked waveform — four clock edges, with D both rising and falling between them — is in 2026-08-10-introduction-to-sequential-circuits[139].

Other flip-flop types

Other types exist — **JK flip-flops** and **T flip-flops** — but they are **not covered in this course**. Only the D flip-flop is used.

Latches vs flip-flops

- **Latches** are **level triggered** devices — they latch the output and respond to changes of logic *levels* on the inputs.
- A latch circuit can be modified so that it becomes sensitive to an **edge** (a momentary transition) of a control input, i.e. a clock signal. Such circuits are called **flip-flops**.
- A flip-flop can store 1 bit of information while being sensitive to a clock edge — it changes its output only *at* the clock edges, based on the inputs.
- So: **latches are level triggered; flip-flops are edge triggered.**

A clock signal has two edges:

Edge	Transition
Positive (rising) edge	$0 \rightarrow 1$
Negative (falling) edge	$1 \rightarrow 0$

A D flip-flop can therefore be **positive edge triggered** or **negative edge triggered**. In a positive edge triggered D flip-flop, D is copied to Q at the positive edge of the clock; in between clock edges the flip-flop is not responsive, thus stores the value.

The **state** of a flip-flop is the value it is storing. Flip-flops are more useful than latches in practice.

[139] courses/csse2010/2026-08-10-introduction-to-sequential-circuits.pdf

Symbols used

Four related symbols, all drawn as a rectangle with **D** in at the top left, **Q** out at the top right, and a clock input **CK** at the bottom left. What distinguishes them is the marking on that CK input.

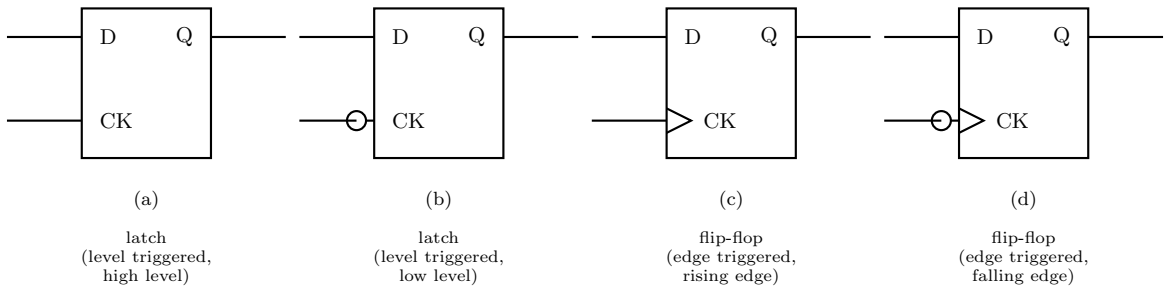


Figure 43: The four latch and flip-flop symbols, with the clock pin labelled CK (the course writes this pin CLK elsewhere). The wedge and the bubble read independently: the wedge says edge-triggered, therefore flip-flop; the bubble says inverted, therefore falling edge or low level. The reading under each symbol follows from those two rules.

Symbol	CK input marking	Device	Triggered on
(a)	plain input	Latch	High level of CK
(b)	**bubble** (inversion circle)	Latch	Low level of CK
(c)	**triangle/wedge**	Flip-flop	**Rising** (positive) edge of the clock
(d)	**bubble + triangle**	Flip-flop	**Falling** (negative) edge of the clock

The two markings read independently and are the whole key to the notation:

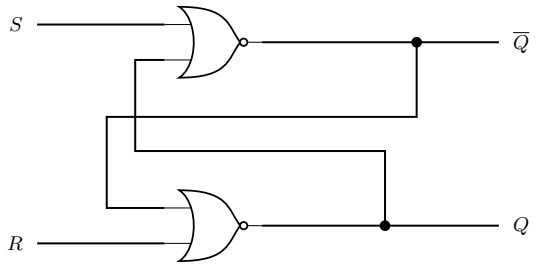
- The **triangle** indicates **edge-triggered**, and therefore that the device is a flip-flop rather than a latch. No triangle means level-triggered, i.e. a latch.
- The **bubble** indicates inversion, i.e. the *falling* edge (or the low level) rather than the rising edge (or the high level).

The SR latch (cross-coupled NOR gates)

Structure: two 2-input NOR gates cross-coupled.

- The **top** NOR gate takes S as one input and the *bottom* gate's output as its other input; its output is $\bar{Q}$.
- The **bottom** NOR gate takes R as one input and the *top* gate's output ($\bar{Q}$) as its other input; its output is Q.

So each gate's output feeds back into the other gate's input — that feedback loop is what stores the bit.



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

$\overline{0+x} = \bar{0}\cdot\bar{x} = 1\cdot\bar{x} = \bar{x}$

a NOR gate with one input held at 0 is an inverter for the other input

$\overline{1+x} = \bar{1}\cdot\bar{x} = 0$

a NOR gate with one input held at 1 has output 0, whatever the other input

Figure 44: The SR latch: two cross-coupled NOR gates, with the NOR truth table to remember and the two special cases the analysis needs. Between them they are the whole analysis: a NOR with a 0 on one input passes the other input inverted, and a NOR with a 1 on one input is pinned to 0. Where the two feedback wires cross there is no junction dot, so it is two wires passing, not a connection.

The analysis needs both identities in the figure. Together they are what makes each row fall out in a line. Both are just De Morgan applied to a NOR (see boolean-algebra[140]): $\overline{A+B} = \bar{A}\cdot\bar{B}$.

The truth table:

S	R	Q	$\bar{Q}$	Meaning
0	0	Q_{n-1}	—	**Storage / remember**
0	1	0	1	**Reset**
1	0	1	0	**Set**
1	1	—	—	**Invalid** $\times$

Q_{n-1} means "whatever Q was before" — the hold row's entry is not a bit but a reference to the previous value. Three cells are left blank: $\bar{Q}$ in the hold row, and both outputs in the last row, which is *Invalid*.

Reasoning for each row, using the two identities above:

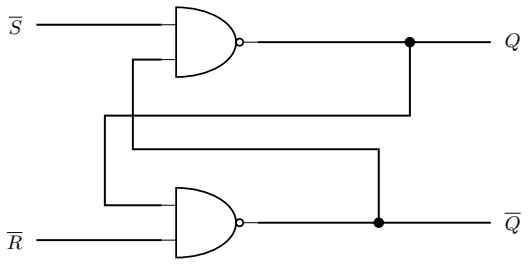
[140] courses/csse2010/boolean-algebra.pdf

- $S = 0, R = 0$: each gate has a 0 on its external input, so each is an inverter of the other's output — $\bar{Q} = \overline{Q}$ and $Q = \overline{\bar{Q}}$. Both $Q = 1$ and $Q = 0$ satisfy that, so the latch simply holds whatever it already had. **This row is the whole point of the circuit**: it is the row in which the latch remembers.
- $S = 1, R = 0$: the top gate has a 1 on an input, so $\bar{Q} = 0$ regardless. The bottom gate then sees $R = 0$ and $\bar{Q} = 0$, so $Q = 1$. S **sets** the latch.
- $S = 0, R = 1$: mirror image — the bottom gate has a 1 on an input, so $Q = 0$; the top gate then sees $S = 0$ and $Q = 0$, so $\bar{Q} = 1$. R **resets** the latch.
- $S = 1, R = 1$: both gates have a 1 on an input, so both outputs are 0: $Q = 0$ and $\bar{Q} = 0$. That result is what condemns the combination: the two outputs are supposed to be complements and here they are equal, so the labels Q and $\bar{Q}$ stop meaning anything. Worse, what happens *after* both inputs return to 0 simultaneously is not uniquely determined (the two gates race), so the resulting state is **indeterminate**. Hence **invalid** — a combination you must not apply.

This circuit is the **active high SR latch**: the inputs do their job when driven to 1, and the resting/hold condition is $S = R = 0$. That name is what distinguishes it from the NAND version below.

The SR latch from NAND gates (active low)

Exercise: analyse the SR latch circuit when the NOR gates are replaced with NAND gates, and complete the truth table. The answer circuit is below. The gate symbols are not merely swapped: **the inputs are relabelled $\bar{S}$ and $\bar{R}$, and which gate's output is called Q is swapped**, so that the set/reset roles survive the change.



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

$\overline{1 \cdot x} = \overline{1} + \overline{x} = 0 + \overline{x} = \overline{x}$

a NAND gate with one input held at 1 is an inverter for the other input

$\overline{0 \cdot x} = \overline{0} + \overline{x} = 1$

a NAND gate with one input held at 0 has output 1, whatever the other input

Figure 45: The NAND SR latch, the active low SR latch, with the NAND truth table and the two identities that drive the analysis. Compare against the NOR version: the gate outputs have swapped names, so the top gate now produces Q, and the inputs are drawn overlined because they are active low.

Worked through with the two NAND identities in the figure:

$\bar{S}$	$\bar{R}$	Q	$\bar{Q}$	Meaning
0	0	1	1	**Invalid** — outputs not complementary
0	1	1	0	**Set** ($\bar{S}$ asserted)
1	0	0	1	**Reset** ($\bar{R}$ asserted)
1	1	Q_{n-1}	—	**Storage / remember**

(Rows in ascending binary order of the inputs, which is what every truth table in the course does — see csse2010-diagram-notation[141].)

- $\bar{S} = \bar{R} = 1$: each gate has a 1 on its external input, so each inverts the other's output — the same self-consistent pair as the NOR latch's $S = R = 0$ row. **Hold.**
- $\bar{S} = 0$: the top gate is pinned to 1, so $Q = 1$; the bottom gate then sees $\bar{R} = 1$ and $Q = 1$, so $\bar{Q} = 0$. **Set.**
- $\bar{R} = 0$: mirror image — $\bar{Q} = 1$, and the top gate then gives $Q = 0$. **Reset.**
- $\bar{S} = \bar{R} = 0$: both gates pinned to 1, so $Q = \bar{Q} = 1$. Same objection as before, so **invalid**.

The headline result, and the reason the inputs are drawn overlined: with NAND gates the inputs become **active low**. The latch rests (holds) at $\bar{S} = \bar{R} = 1$ instead of at $S = R = 0$, and the forbidden combination moves from both-high to both-low. Everything else — set, reset, hold, one invalid corner — is the same circuit doing the same job.

This active-low convention is general: it is why the 74HCT74's asynchronous pins are $\overline{\text{SET}}$ and $\overline{\text{CLR}}$ (see device-pinouts[142]). The SET and CLR inputs are typically active low.

A real D flip-flop

A real D flip-flop is built from cross-coupled NAND gates — **six** NAND gates in three cross-coupled pairs: two input latches driven by D and CLK, and a third pair forming the output latch that carries Q and $\bar{Q}$. That is the same storage idea as the SR latch above, wrapped in enough extra gating to make it respond to an *edge* rather than a *level*.

As well as D, CLK and the outputs Q and $\bar{Q}$, it has two extra inputs:

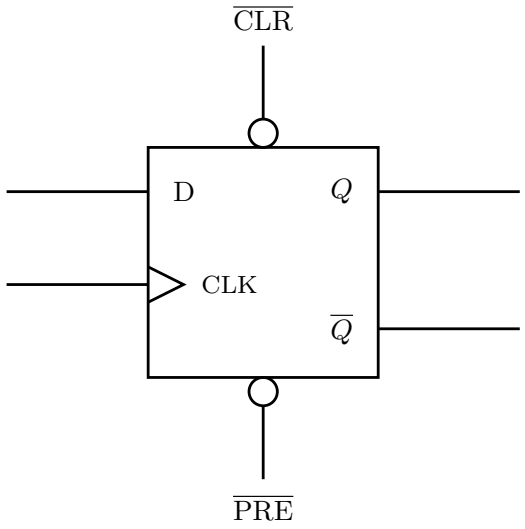
- **$\overline{\text{PRE}}$ (Preset)** — forces Q to 1.
- **$\overline{\text{CLR}}$ (Clear)** — forces Q to 0.

Both are drawn with overbars and with a bubble on the pin, i.e. they are **active low**, and both are **asynchronous**: they act on the latch directly, independent of the clock, rather than waiting for a clock edge. They run straight into the NAND gates of the latches, bypassing

[141] courses/csse2010/csse2010-diagram-notation.pdf
[142] courses/csse2010/device-pinouts.pdf

the D/CLK path entirely — $\overline{\text{PRE}}$ reaches the gate that produces Q and $\overline{\text{CLR}}$ the gate that produces $\bar{Q}$, which is exactly how each forces its output.

The two things to remember about these pins: they are *active low inputs* and *asynchronous inputs*.



Asynchronous, active-low inputs
they act the instant they are driven low — they do *not* wait for a clock edge

$\overline{\text{CLR}} = 0 \Rightarrow Q = 0$ (clear)
$\overline{\text{PRE}} = 0 \Rightarrow Q = 1$ (preset)
$\overline{\text{CLR}} = \overline{\text{PRE}} = 1 \Rightarrow$ normal clocked operation

The bubble is the active-low marking; the overbar on the name says the same thing.

Figure 46: The D flip-flop symbol with its asynchronous pins, CLR on top and PRE underneath. The 74HCT74 in device-pinouts is drawn the other way up (SET on top, CLR below), which is the course's usual order; the course is also inconsistent about the *naming* (PRE/CLR against SET/CLR). The three consequences of the pins being asynchronous and active low are on the right.

Both pins being asserted at once ($\overline{\text{PRE}} = \overline{\text{CLR}} = 0$) is the same kind of contradiction as $S = R = 1$ on the NOR latch — it demands $Q = 1$ and $Q = 0$ simultaneously — and is likewise a combination to avoid. It follows from the SR-latch analysis above.

When neither is needed they can simply be **omitted from a drawing**: the course's *Device Pinouts and Symbols* sheet says so in words, and its own logic-diagram symbol for a D flip-flop leaves them off. See device-pinouts[143] and circuit-schematics[144].

D flip-flop chips

- **74HCT74** — dual D flip-flop (two independent D flip-flops in a 14-pin package), each with its own clock, its own CLR and its own PR (preset), and $Q/\bar{Q}$ outputs. Pinout and

[143] courses/csse2010/device-pinouts.pdf
[144] courses/csse2010/circuit-schematics.pdf

schematic symbol in device-pinouts[145].

- **74HCT273** — eight D flip-flops in a 20-pin package, so it can hold **one byte** (8 bits) of information. It is not in the course's device-symbols document, so there is no pinout for it here.

The full symbol set is in the device symbols PDF on Blackboard.

See also

- 2026-08-10-introduction-to-sequential-circuits[146] — Lecture 5, where all of this is introduced
- sequential-circuits[147] — what you build once you have a storage element
- shift-registers[148] · [counters] · state-machines[149]
- memory-technologies[150] — static RAM, a memory that keeps each bit in a latch
- week3-lab-logisim-and-flip-flops[151] — the lab that simulates these circuits
- logic-gates[152] · boolean-algebra[153] — the NOR/NAND behaviour the latch analysis rests on
- timing-diagrams[154] — how to read and draw the waveforms above
- csse2010-diagram-notation[155] — the drawing conventions these figures follow

The I/O Board

The I/O board in the lab kit carries the input and output devices you wire your breadboard circuits to, for experimenting with simple digital circuits. It plugs directly onto the breadboard, and adds a **logic probe** for debugging and a **UART serial interface** for talking to other devices.

It is powered from a single USB connection and has **current-sense circuitry that detects short circuits and protects your designs**.

Pin numbers for every device are in device-pinouts[156]; the kit itself is loaned out in week3-lab-logisim-and-flip-flops[157]. Two more parts of the kit that are used alongside the board, the

[145] courses/csse2010/device-pinouts.pdf
[146] courses/csse2010/2026-08-10-introduction-to-sequential-circuits.pdf
[147] courses/csse2010/sequential-circuits.pdf
[148] courses/csse2010/shift-registers.pdf
[149] courses/csse2010/state-machines.pdf
[150] courses/csse2010/memory-technologies.pdf
[151] courses/csse2010/week3-lab-logisim-and-flip-flops.pdf
[152] courses/csse2010/logic-gates.pdf
[153] courses/csse2010/boolean-algebra.pdf
[154] courses/csse2010/timing-diagrams.pdf
[155] courses/csse2010/csse2010-diagram-notation.pdf
[156] courses/csse2010/device-pinouts.pdf
[157] courses/csse2010/week3-lab-logisim-and-flip-flops.pdf

joystick and the LED matrix, are described at the end.

What's on it

1. USB connector	2. Power LED	3. Power switch	4. 7-segment display
5. Switches	6. LEDs	7. Buttons	8. UART transmit/receive LEDs
9. Logic probe indicator LEDs	10. Fault LED	11. VCC / GND headers	12. I/O device headers
13. Auxiliary power connector	14. High current mode jumper		

Items 13 and 14 are the auxiliary power connector and the high-current-mode jumper. The one extra power connection the labs use is connector J1, which powers the LED matrix in Lab 16 (below).

The board at a glance

The board is a **C shape**: a left-hand spine carrying the USB connector, the power switch, the two status LEDs and the seven-segment display, plus a top arm and a bottom arm that straddle the breadboard's central channel. All the I/O devices come out on the **two 16-pin headers** on the inner edges of those arms, and *every* device is reached through the column of breadboard holes directly above or below its printed label.

Two things about the numbering are worth fixing in your head before you wire anything:

- **The top header counts down, left to right.** Its leftmost pin is 32 and its rightmost is 17, so the switches run `S7`…`S0` and then the LEDs `L7`…`L0` as you read left to right. That is the opposite of the reading order you would guess from the pin numbers alone.
- **The bottom header counts up, left to right**, pin 1 to pin 16: `TX RX CC DP G F E D C B A B3 B2 B1 B0 PR`. The middle nine of those are the seven-segment display — `CC` its common cathode, `DP` its decimal point, and `G` down to `A` its seven segments — which is why nine consecutive pins carry single-letter names.

`VCC` and `GND` pads sit at the far right of both arms, which is where the breadboard's power rails get fed from.

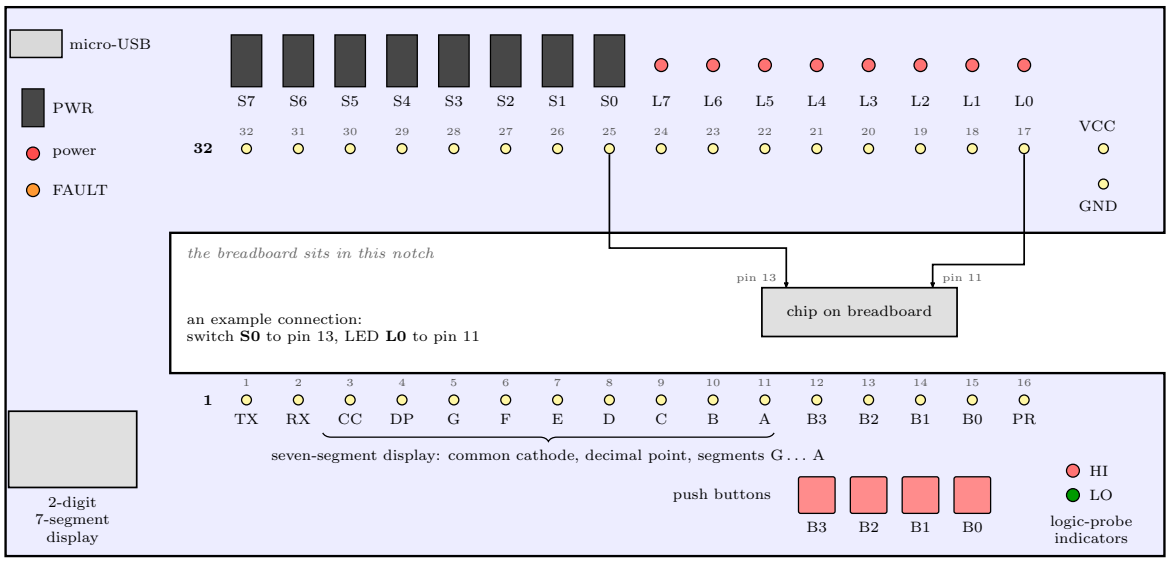


Figure 47: A plan view of the board, not to scale, with the parts on the left spine placed roughly. Each I/O device sits on one pin of one of the two 16-pin headers, and the column of breadboard holes at that pin is how you reach it. The board itself prints only the two end markers, a bold 32 at the left of the top header and a bold 1 at the left of the bottom one. The small numbers at the other pins are their pin numbers, the ones a schematic uses. The connection drawn in the notch is an example: switch S0 to pin 13 of a chip, LED L0 to pin 11.

Setup

Insert the board into the breadboard, aligning the downward-facing headers with the right holes.

> If aligned correctly, the I/O board **should not require excessive force** to insert.

Connect a micro-USB cable to the connector at the **top-left** corner.

> **DO NOT STORE THE BOARD WITH THE USB CABLE ATTACHED — THIS MAY CAUSE THE USB CONNECTOR TO BREAK OFF!!**
>
> The USB connector is the part people break.

Turn the board on by sliding the **PWR** switch up to ON; the red power LED should light. Then **check the orange FAULT LED does not light**. If it does, power off immediately.

The fault LED

The board uses a **current-limited power switch** to protect itself and anything it powers from short circuits. A short lights the orange FAULT LED.

If it's on: push PWR down to OFF **immediately**, and check your design for shorts before restoring power.

> **WARNING! In the case of a short circuit, the current-limiting switch may become extremely hot.**

Switches, LEDs and buttons

All the I/O devices are reached through the breadboard holes **directly above or below the printed label** on the board:

- **Switches** — `S0` … `S7` (pins 25–32)
- **LEDs** — `L0` … `L7` (pins 17–24)
- **Buttons** — `B0` … `B3` (pins 15 down to 12)

Connect one by running a wire from the column of breadboard terminals above/below the label to wherever it's needed. An example, drawn into the notch of Figure 47: `S0` to pin 13 of an external device, `L0` to pin 11.

Switches and buttons are **inputs to your circuit**, LEDs and the seven-segment display are **outputs from it** — which is the distinction the pentagon symbols in circuit-schematics[158] make, and getting it backwards is one of the common errors listed there.

[158] courses/csse2010/circuit-schematics.pdf

The logic probe

Attach a wire to the column of terminals above the `PR` label — pin 16, the rightmost pin of the bottom header. Touch the free end to any point in the circuit and the **HI / LO** LEDs show the logic state there.

> As circuits increase in complexity, so does the necessity of having a clear strategy — **probing random points in a large circuit rarely reveals the location of errors**.

Suggested troubleshooting strategy

1. **Check the FAULT light is not on**, which would indicate a short circuit.
2. **Probe VCC and GND at every device** in your circuit, to confirm each one is actually powered.
3. **Start at the inputs and move logically through to the outputs.** Pen and paper helps: write down the *expected* state at each point first, then verify it with the probe. A schematic and a truth table are great for this.

   > The key to any successful debugging strategy is to **assume nothing, know what to expect before you begin testing, and have a plan.**

Step 2 is the one worth internalising — an unpowered chip looks exactly like a wrong-logic chip until you probe its power pins, and grouped VCC/GND entries on a schematic (see circuit-schematics[159]) make it easy to forget one.

Electrical characteristics

VCC	5 V
GND	0 V
Logic high	> 3.5 V
Logic low	< 1.5 V

The gap between 1.5 V and 3.5 V is *undefined*, which is why a floating pin can read as either (week3-lab-logisim-and-flip-flops[160]). What the logic probe shows for a voltage inside that band is not documented.

[159] courses/csse2010/circuit-schematics.pdf
[160] courses/csse2010/week3-lab-logisim-and-flip-flops.pdf

The joystick

The kit's two-axis joystick is a small board of its own, made to sit in the breadboard next to the I/O board. Each axis is a **potentiometer** between its own supply pin and ground, and the stick moves the wiper. So each axis gives a voltage that follows the stick across its travel, from 0 V at one side to 5 V at the other, for the A/D converter to read (analog-to-digital-conversion[161]).

Pin	What it is	Connect to
L/R+	top of the left-right potentiometer	VCC
U/D+	top of the up-down potentiometer	VCC
GND ×2	bottom of both	GND, either one
L/R ×2	**x**: 0 V with the stick to the left, 5 V to the right	an analog input, either one
U/D ×2	**y**: 0 V at the bottom, 5 V at the top	an analog input, either one

- **The doubled pins are joined on the board.** One wire to each of L/R, U/D and GND is enough.
- **The two + pins are separate.** Both have to go to VCC, or one axis has nothing to divide.
- **Left, right, up and down are the joystick's own,** marked by the X and Y arrows printed on it. Placed at the right-hand end of the breadboard, with room for the microcontroller board and programmer to its left, the X arrow points to the top of the breadboard and the Y arrow to the left.
- **A potentiometer's wiper is a weak source.** Anything that draws current from it, an LED for instance, pulls its voltage away from the stick's position. LEDs make a quick wiring test, but they must come off before the converter is used.
- **Keep the black foam** that protects the pins when the joystick is put away.

The LED matrix

The LED matrix is a display of **16 columns by 8 rows**, each pixel a red LED and a green LED together. It is driven over **SPI**: the AVR is the master and the matrix a slave that takes commands one byte at a time (avr-spi[162]).

[161] courses/csse2010/analog-to-digital-conversion.pdf
[162] courses/csse2010/avr-spi.pdf

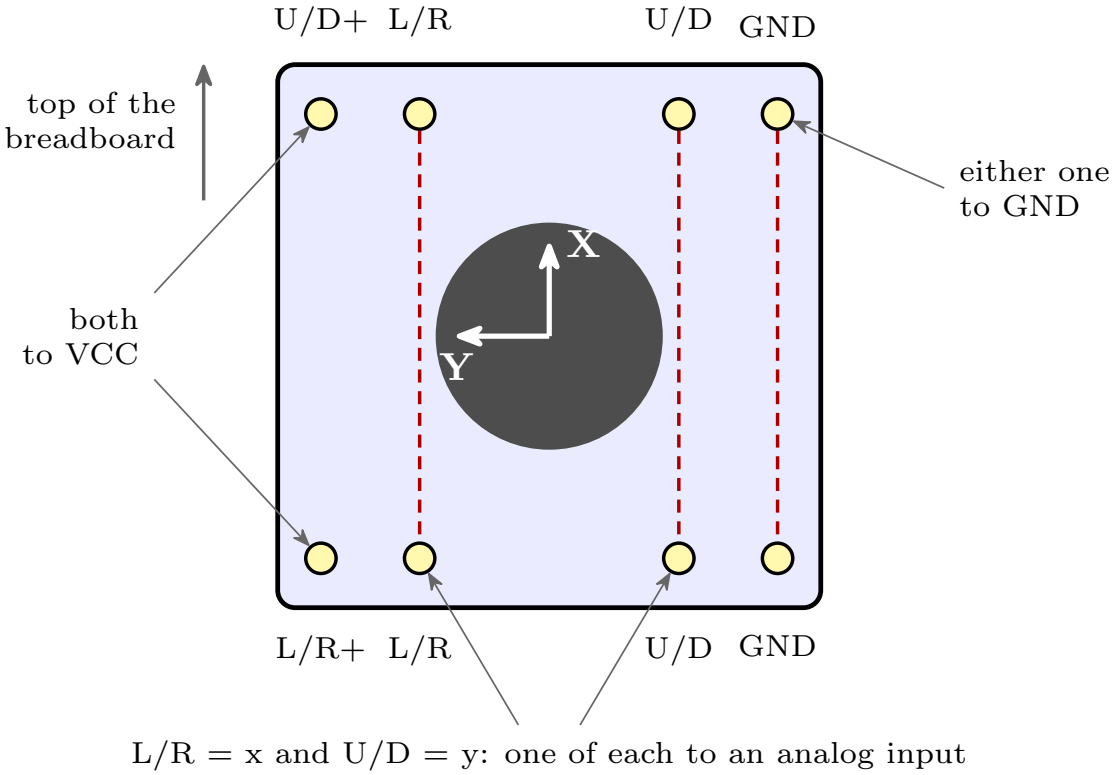


Figure 48: The joystick's eight pins, seen from above with the board placed as it sits in the breadboard, its X arrow pointing to the top. The pins near each corner come in twos. The dashed lines join the pins that are connected on the board itself: the two L/R pins, the two U/D pins and the two GND pins, so one wire to each pair is enough. The two supply pins, U/D+ at the top left and L/R+ at the bottom left, are not joined, and both go to VCC. L/R carries x and U/D carries y to the analog inputs.

Connecting it

It has two headers:

- **K1**, two pins, is its **power**. A two-wire lead joins it to the I/O board's connector **J1**, VCC on J1's left pin and GND on its right.
- **K2**, four pins, is the **SPI connection**, to the AVR's SPI port on PB7 to PB4: SCK on PB7, MISO on PB6, MOSI on PB5 and $\overline{\text{SS}}$ on PB4. Match the order of the pins on K2 against the datasheet before plugging the lead in.

The Err state. PB7 to PB5 are also the pins the programmer uses to load the AVR, so programming sends the matrix a stream of what look to it like garbage commands, and it can show **Err**. Holding the AVR's reset button down for at least half a second after programming clears it before the program starts.

What to send it

Every message starts with a **command byte**. Two commands are enough to draw anything:

Command	Bytes	Does
clear	`0x0F`	every pixel off
set one pixel	`0x01`, *position*, *colour*	sets one pixel to a colour

- **Position byte: `(y << 4) | x`.** The row y, from 0 at the bottom to 7 at the top, is the high nibble. The column x, from 0 at the left to 15 at the right, is the low nibble.
- **Colour byte: `(green << 4) | red`.** Each nibble is a brightness from 0 to 15. Red alone is `0x0F`, green alone `0xF0`, both full is yellow, `0xFF`, and full red with half green is orange, `0x7F`.

The matrix's reference manual is the full description of the commands it accepts.

Lab 16 works through the commands and uses them for a demo, a scrolling marquee across two matrices and an Etch-A-Sketch (week10-lab-adc-and-spi[163]).

See also

- device-pinouts[164] — the pin table behind Figure 47, and the board's schematic symbols
- circuit-schematics[165] — how the board is drawn on a schematic (`IO1` / `IOBOARD`, input and output pentagons)

[163] courses/csse2010/week10-lab-adc-and-spi.pdf
[164] courses/csse2010/device-pinouts.pdf
[165] courses/csse2010/circuit-schematics.pdf

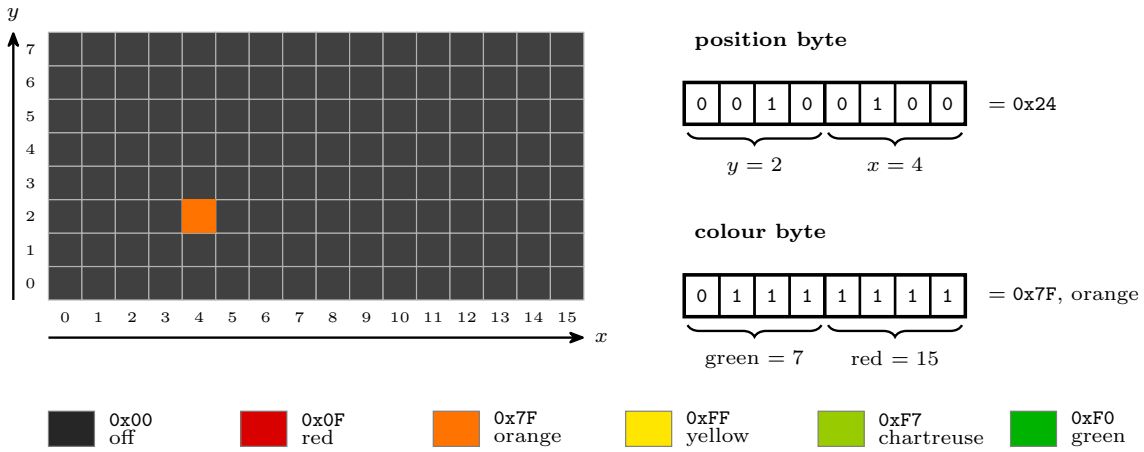


Figure 49: How a pixel is addressed and coloured. The matrix is 16 columns, x from 0 to 15 left to right, by 8 rows, y from 0 to 7 bottom to top. A position byte puts y in its high nibble and x in its low one, so the pixel at x = 4, y = 2 is `0x24`. A colour byte puts the green brightness in its high nibble and the red in its low one, each 0 to 15. The swatches show what some colour bytes look like: the colours here are the light from the red and green LEDs mixed.

- avr-io-ports[166] — the same switches and LEDs driven from an AVR rather than from gates
- week3-lab-logisim-and-flip-flops[167] — where the kit is loaned out
- week10-lab-adc-and-spi[168] — Lab 16: the joystick read by the A/D converter, and the LED matrix driven over SPI
- analog-to-digital-conversion[169] · avr-spi[170] — the two AVR peripherals the joystick and the matrix are used with

Logic Gates

Every logic gate has one or more inputs and exactly one output, and can be described four equivalent ways: **logic symbol, truth table, Boolean expression, timing diagram** (see timing-diagrams[171] for the last of these, and for real-gate propagation delay / rise / fall times).

[166] courses/csse2010/avr-io-ports.pdf
[167] courses/csse2010/week3-lab-logisim-and-flip-flops.pdf
[168] courses/csse2010/week10-lab-adc-and-spi.pdf
[169] courses/csse2010/analog-to-digital-conversion.pdf
[170] courses/csse2010/avr-spi.pdf
[171] courses/csse2010/timing-diagrams.pdf

Gates sit at the bottom of the abstraction stack — the **digital logic level**, level 0 (see 2026-07-27-course-intro-bits-bytes-binary[172]). A digital circuit only ever carries **two** logic levels: logic '0' is usually a small voltage (around 0 V) and logic '1' a larger one (e.g. 0.8 V to 5 V, depending on the **logic family** — the type and size of the transistors the gate is built from). Because inputs and outputs have only these two states, they can equivalently be called *true* and *false*.

The 7 basic gate types

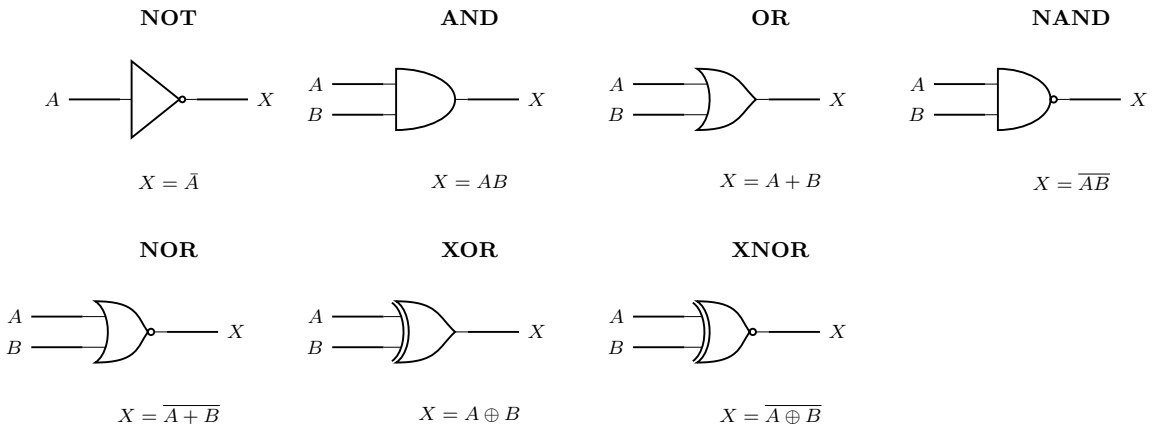


Figure 50: The seven basic gates, with the distinguishing symbol and Boolean expression for each. NOT, AND and OR are the three base shapes; NAND, NOR and XNOR are those same bodies with an inversion bubble on the output, and XOR/XNOR add a second arc across the back of the OR body.

Gate	Behaviour	Truth table ($A, B \to X$)
NOT (inverter)	Inverts the input	`0→1`, `1→0`
AND	Output is HIGH when **all** inputs are HIGH	00→0, 01→0, 10→0, 11→1
OR	Output is HIGH when **at least one** input is HIGH	00→0, 01→1, 10→1, 11→1
NAND	Output is HIGH when **at least one** input is LOW	00→1, 01→1, 10→1, 11→0
NOR	Output is HIGH when **all** inputs are LOW	00→1, 01→0, 10→0, 11→0
XOR	Output is HIGH when **exactly one** input is HIGH	00→0, 01→1, 10→1, 11→0

[172] courses/csse2010/2026-07-27-course-intro-bits-bytes-binary.pdf

Gate	Behaviour	Truth table ($A, B \to X$)
XNOR	Output is HIGH when the inputs are **the same**	00→1, 01→0, 10→0, 11→1

Useful to remember: **XOR is the odd function**, **XNOR is the even function**.

Deriving the inverting gates from the base gates

The quickest way to fill in a NAND/NOR/XNOR truth table is not to memorise it, but to write the underlying AND/OR/XOR column first and then invert it row by row — the "N" prefix is exactly one inversion on the output:

A	B	AND	**NAND**	OR	**NOR**	XOR	**XNOR**
0	0	0	**1**	0	**1**	0	**1**
0	1	0	**1**	1	**0**	1	**0**
1	0	0	**1**	1	**0**	1	**0**
1	1	1	**0**	1	**0**	0	**1**

More than two inputs

A gate of n inputs has a truth table of 2^n rows — two inputs give $2 \times 2 = 4$ rows, three inputs give 8. The "at least one / all" wordings above carry over unchanged to more inputs; the XOR/XNOR wordings do **not**, and must be restated in terms of parity:

- **XOR** output is HIGH when an **odd number** of inputs are HIGH (the *odd function*). For two inputs this collapses to "exactly one", which is why the 2-input wording works.
- **XNOR** output is HIGH when an **even number** of inputs are HIGH (the *even function*).

A	B	C	3-input AND	**3-input NAND**	**3-input XOR**
0	0	0	0	**1**	**0**
0	0	1	0	**1**	**1**
0	1	0	0	**1**	**1**
0	1	1	0	**1**	**0**
1	0	0	0	**1**	**1**
1	0	1	0	**1**	**0**
1	1	0	0	**1**	**0**
1	1	1	1	**0**	**1**

Note that the kit of parts used for practicals has **no** 3-or-more-input gate chips (see device-pinouts[173]) — a 3-input function has to be built from 2-input gates.

Universal (complete) gates

All circuits can be constructed from **NAND-only** or **NOR-only** gates — these are called **complete** (or universal) gates, because NOT, AND, and OR can each be built purely from one gate type:

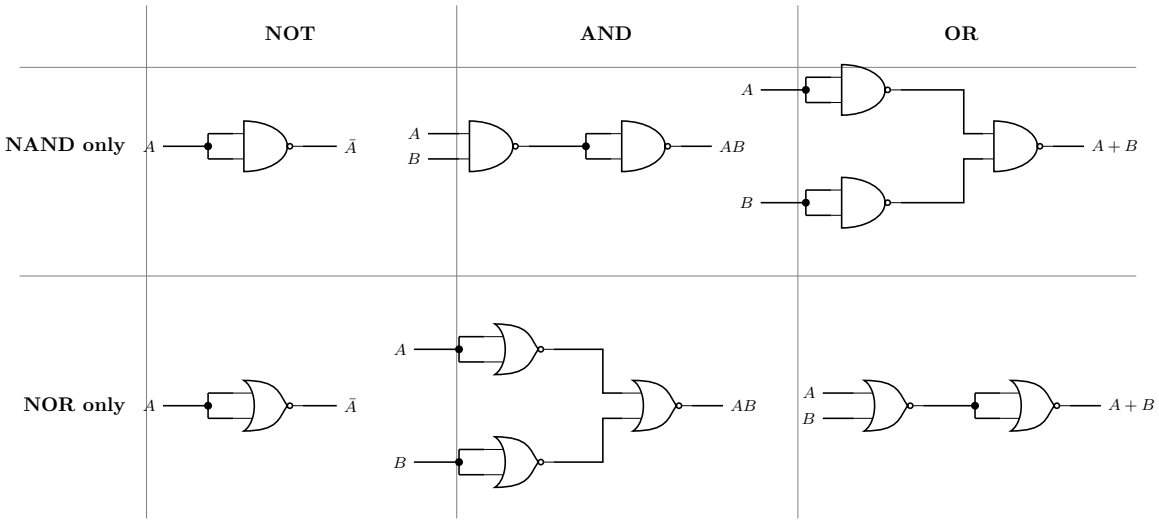


Figure 51: Why NAND and NOR are complete gates: NOT, AND and OR built from each one alone. Tying both inputs of a NAND or NOR together turns it into an inverter, and everything else follows — AND is NAND then invert, OR is invert-both-inputs then NAND (and the NOR row is the mirror image of that).

- **NOT**: a NAND (or NOR) gate with both inputs tied together — $\overline{A \cdot A} = \bar{A}$, and $\overline{A + A} = \bar{A}$.
- **AND from NAND**: a NAND followed by a second NAND wired as an inverter. **AND from NOR**: invert both inputs first, then NOR them ($\overline{\bar{A} + \bar{B}} = AB$, by De Morgan — see boolean-algebra[174]).
- **OR from NOR**: a NOR followed by a NOR-as-inverter. **OR from NAND**: invert both inputs first, then NAND them ($\overline{\bar{A} \cdot \bar{B}} = A + B$).

The two rows are De Morgan's law drawn twice: whenever you push an inversion through a gate, AND and OR swap. NAND and NOR are preferred in practice because they are **easier to build from transistors** than AND/OR directly.

[173] courses/csse2010/device-pinouts.pdf
[174] courses/csse2010/boolean-algebra.pdf

Gates on ICs

Gates are sold packaged several to an **integrated circuit** (IC). The **74HCT00** is four independent 2-input NAND gates in one 14-pin DIP package.

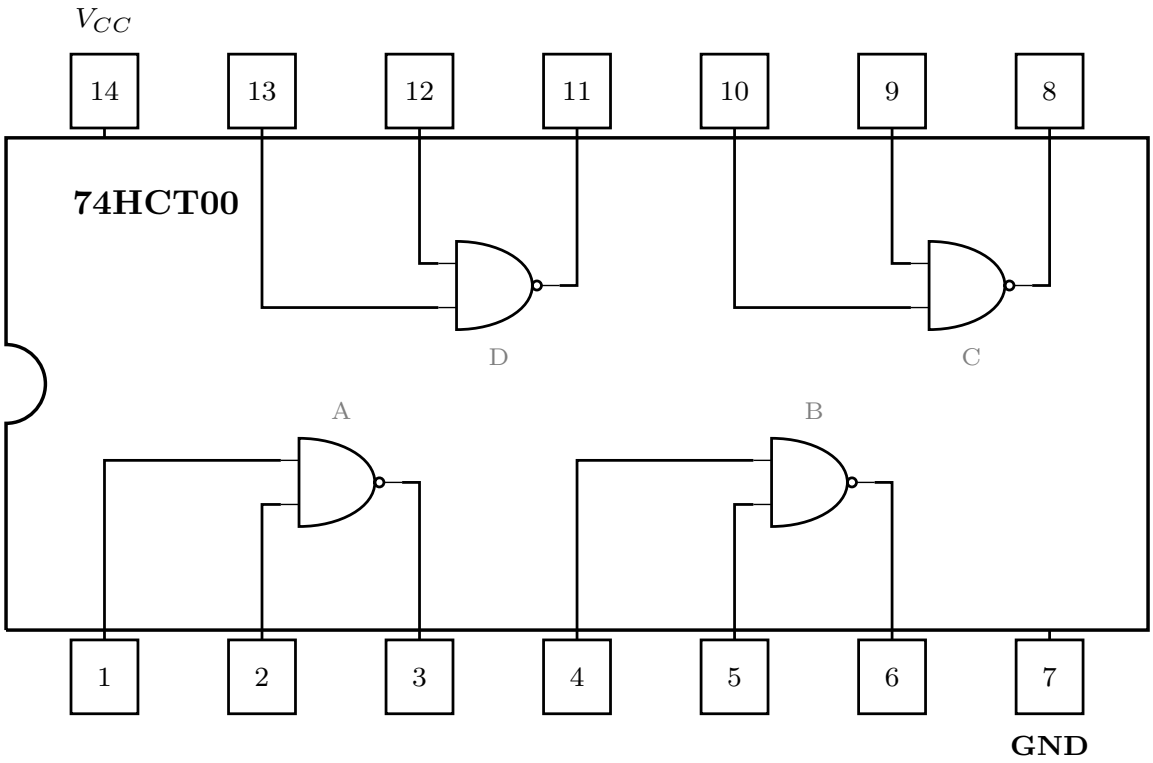


Figure 52: Inside the 74HCT00: four independent 2-input NAND gates in a 14-pin package, with the pin each gate terminal is wired to. Pin 14 is Vcc (power, e.g. 5 V) and pin 7 is GND (0 V) — shared by all four gates, so nothing works if either is left unconnected. Gate letters A-D are the labels a schematic uses as U1:A … U1:D.

- **Vcc** = power (e.g. 5 V), **GND** = ground (0 V). One Vcc/GND pair supplies all four gates.
- Physically: pin spacing is 0.1” along the package and 0.3” across it; the chip is about 15 mm long.
- The drawing above is the chip’s **internal schematic**, not a picture of the part — see circuit-schematics[175] for how a chip like this is drawn (and labelled `U1:A` … `U1:D`) on a schematic you can actually build from, and device-pinouts[176] for the pinouts of every

[175] courses/csse2010/circuit-schematics.pdf
[176] courses/csse2010/device-pinouts.pdf

chip in the course kit.

Sequential Circuits

A **sequential circuit** is a circuit whose output depends not only on the current inputs but also on the values it is currently storing. The storage is done with flip-flops — see flip-flops-and-latches[177] for the storage elements themselves.

The output of any logic gate or combinational circuit depends on the **current value of the inputs only**, so if an input changes, the output can change too and **the previous value is lost forever**. A circuit with **memory** can remember a value even when the input changes. First met in 2026-08-10-introduction-to-sequential-circuits[178].

Combinational vs sequential

	Combinational	Sequential
Contents	Logic gates only (**no** flip-flops)	Includes **flip-flops** as well as logic gates
Output determined by	The inputs alone — uniquely	Current inputs **and** current **state**
Repeatability	Same inputs always give the same output	Same inputs can give different outputs, depending on state
When output changes	Whenever an input changes	Only when the clock 'ticks'
Examples	Adders, multiplexers, decoders, demultiplexers, encoders (see combinational-logic-blocks[179])	Counters, registers (see [counters], shift-registers[180])

The two example lists are worth memorising as the dividing line: the blocks built earlier in the course are all combinational, and the ones that follow — registers, counters, state machines — are all sequential.

A combinational example from earlier in the course: A into one input of an AND gate, B and C into an OR gate whose output feeds the AND gate's other input, giving $X = A(B + C)$. Nothing in that circuit remembers anything — change A, B or C and the output follows immediately (after propagation delay; see timing-diagrams[181]).

[177] courses/csse2010/flip-flops-and-latches.pdf
[178] courses/csse2010/2026-08-10-introduction-to-sequential-circuits.pdf
[179] courses/csse2010/combinational-logic-blocks.pdf
[180] courses/csse2010/shift-registers.pdf
[181] courses/csse2010/timing-diagrams.pdf

The key contrast: if an input to a combinational circuit changes, the output can change too and the **previous value is lost forever**. A sequential circuit can hold onto a value even after the input that produced it has gone away.

State

- **State** = the value stored in the flip-flops.
- **Output** depends on the inputs *and* the state.
- **Next state** depends on the inputs *and* the (present) state.

"Present state" is what the flip-flops hold right now; "next state" is what they will hold after the next clock edge. The combinational logic computes the next state from the present state and the inputs; the flip-flops only adopt it when the clock edge arrives.

General structure of a sequential circuit

The standard block diagram has two blocks and a feedback loop.

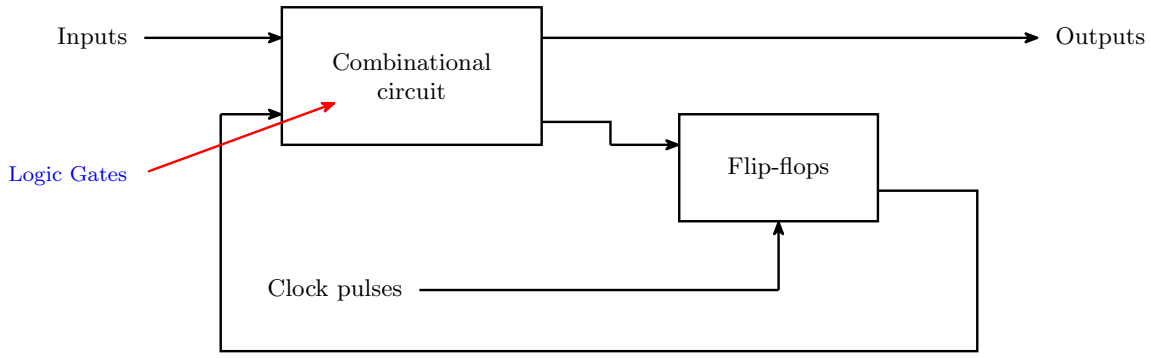


Figure 53: The general sequential circuit (after Mano, Digital Design, 3rd edition). Two things carry the whole idea: the loop running from the flip-flops back into the combinational block, which is what makes the circuit sequential, and the clock stub, which enters the flip-flops and nothing else. The 'Logic Gates' callout marks the combinational block as gates only.

- A **combinational circuit** block (just logic gates). It takes the external **inputs** *plus* the feedback from the flip-flops, and produces the external **outputs** *plus* the values to be loaded into the flip-flops.
- A **flip-flops** block (the storage elements). Its data inputs come from the combinational circuit; it also takes **clock pulses** as a separate control input.

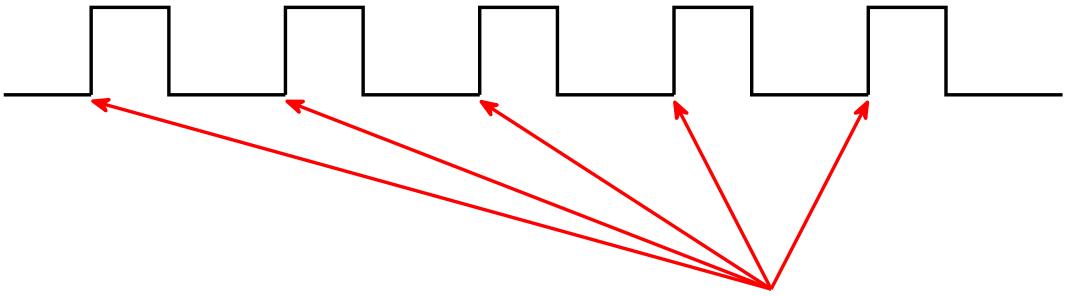
- A **feedback path** runs from the flip-flops' outputs back around into the combinational circuit's inputs. That loop is what makes the circuit sequential — the stored state is fed back in and influences the next output and the next state.

The clock pulses go **only** to the flip-flops, not to the combinational logic. Cut the feedback wire and what is left is a plain combinational circuit with a register bolted on the end; it is the loop that makes the difference.

This same block diagram reappears in shift-registers[182], [counters] and state-machines[183] — those are all just specific choices of what goes in the combinational block.

Synchronous sequential circuits

In a **synchronous** sequential circuit the storage elements can only change at **discrete instants of time**, set by a common clock signal.



the storage elements can change *only* at these instants — the rising edges of the one clock every flip-flop shares

Figure 54: A clock signal. The five arrows all point at rising edges: these are the only instants at which any storage element in a synchronous circuit is allowed to change. Everywhere else on the axis the flip-flops are inert, however much their inputs move — which is exactly the opposite of a logic gate, whose output changes whenever its input does.

- Assume a clock signal — a regularly repeating train of pulses alternating between 0 and 1.
- The outputs of the storage elements change **only on the edges** of that control signal.
- Contrast with logic gates, whose outputs change whenever their inputs change.

Because every flip-flop shares the same clock, the whole circuit's state changes in one step at each clock edge, which is what makes such circuits tractable to design and analyse.

[182] courses/csse2010/shift-registers.pdf
[183] courses/csse2010/state-machines.pdf

It also means the clock period sets the pace of the design: everything the combinational block has to compute must settle within one tick. This follows from the propagation delays of timing-diagrams[184].

See also

- 2026-08-10-introduction-to-sequential-circuits[185] — Lecture 5
- state-machines[186] — the same circuits described by their states and transitions, which is how they are actually designed.
- flip-flops-and-latches[187] · [counters] · shift-registers[188]
- csse2010-diagram-notation[189] — the block-diagram conventions the figures above follow

Shift Registers

Registers and shift registers are the simplest useful sequential circuits[190] — groups of D flip-flops[191] sharing a common clock.

Registers

- A **register** is a group of flip-flops. An **n-bit register** consists of n flip-flops and is capable of storing n bits.
- A register is a sequential circuit **without any combinational logic** — it is just the storage elements and their shared control lines.
- Registers are used to store binary information (data/instructions) **inside a processor**.

A 4-bit register:

Signal	Connection
$I_0 \dots I_3$	the four parallel data inputs, one to each flip-flop's D input
$A_0 \dots A_3$	the four parallel outputs, taken from each flip-flop's Q

[184] courses/csse2010/timing-diagrams.pdf
[185] courses/csse2010/2026-08-10-introduction-to-sequential-circuits.pdf
[186] courses/csse2010/state-machines.pdf
[187] courses/csse2010/flip-flops-and-latches.pdf
[188] courses/csse2010/shift-registers.pdf
[189] courses/csse2010/csse2010-diagram-notation.pdf
[190] courses/csse2010/sequential-circuits.pdf
[191] courses/csse2010/flip-flops-and-latches.pdf

Signal	Connection
`Clock`	a single common line driven to the C (clock) input of **all four** flip-flops
`Clear`	a single common line driven to the R (reset) input of all four flip-flops, drawn **active-low** (bubble on the input)

So on each rising clock edge all four flip-flops simultaneously capture their I inputs; asserting `Clear` forces all four outputs to 0 asynchronously (see flip-flops-and-latches[192] for asynchronous SET/CLR).

Shift registers

> A **shift register** is a register which is capable of shifting its binary information in one or both directions.

The 4-bit shift register example is built as a **chain**: the serial input SI feeds the D input of the first flip-flop; each flip-flop's Q output feeds the next flip-flop's D input; the last flip-flop's Q is the serial output SO. All four flip-flops share a common CLK line.

On each rising clock edge every stored bit moves one place along the chain, the bit currently on SI enters the first stage, and the bit that was in the last stage leaves at SO.

The **common clock is what makes the shift well-defined**. Every flip-flop samples its D input at the same instant, so each stage captures its neighbour's *old* value, not the new one the neighbour is simultaneously taking on; the whole word moves exactly one place per edge rather than racing along the chain. This is the point of the rule that in a synchronous sequential circuit all the sequential elements share a common clock signal — see sequential-circuits[193].

Worked through by hand, with stages labelled Q_0 (nearest SI) through Q_3 ($= SO$), starting from all zeros and shifting in the bit sequence 1, 0, 1, 1:

Clock edge	SI	Q_0	Q_1	Q_2	Q_3 ($= SO$)
(initial)	—	0	0	0	0
1	1	1	0	0	0
2	0	0	1	0	0
3	1	1	0	1	0
4	1	1	1	0	1

[192] courses/csse2010/flip-flops-and-latches.pdf
[193] courses/csse2010/sequential-circuits.pdf

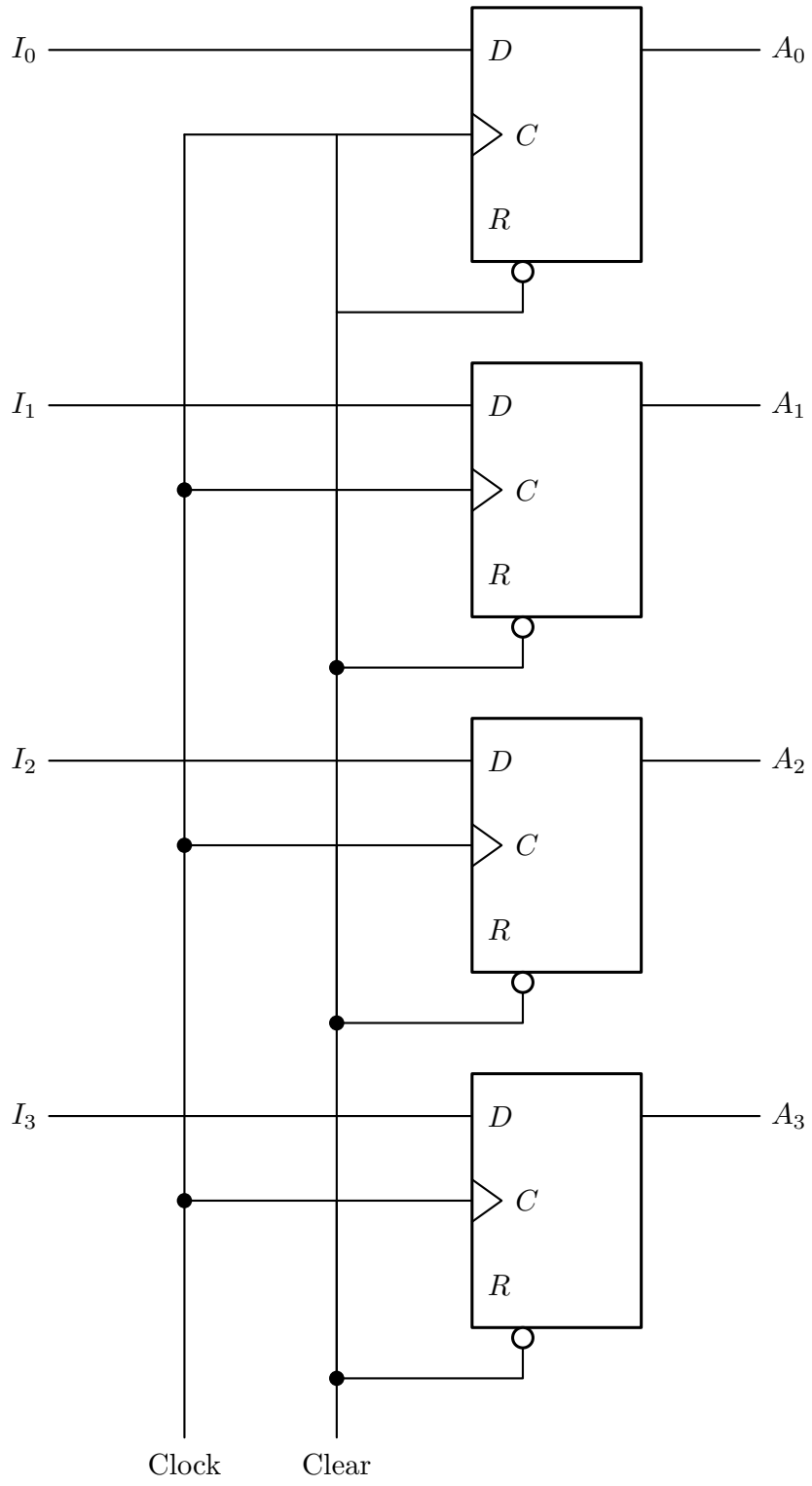


Figure 55: A 4-bit register (Mano, Digital Design, 3rd ed.). Four flip-flops, four parallel inputs, four parallel outputs, and exactly two shared control lines — no gate anywhere, which is what makes a register a sequential circuit without combinational logic. The pin labelling is Mano's, which differs from the rest of the course: the clock input is lettered C rather than CLK and the asynchronous clear is lettered R, marked active-low by the bubble on the pin. The course uses several labellings for both (see the diagram-notation note).

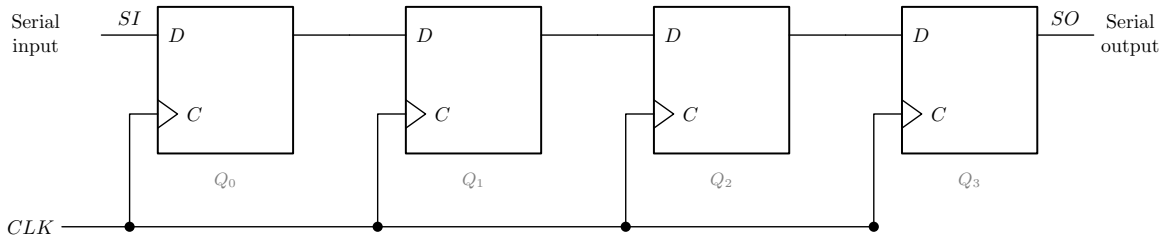


Figure 56: The 4-bit shift register (Mano's Fig. 6-3). The whole device is four D flip-flops and one wire each: the only structural difference from the register above is where each D input comes from. The grey stage names Q0 to Q3 are there so the trace table below can be read against the drawing.

After 4 clock edges the 4 serially-supplied bits sit in the 4 flip-flops, and one further edge would push the first of them out of SO.

Serial parallel conversion

Shift registers can be used to do **serial-to-parallel conversion, and vice versa**. This is the reason they show up everywhere data has to travel over a single wire but be *used* a word at a time.

- **Serial in, parallel out**: drive the bits one per clock edge into SI; after n edges, the n flip-flop Q outputs — read as a group — are the parallel word. This is exactly the table above: clock 4 bits in, then read $Q_0Q_1Q_2Q_3$ in parallel.
- **Parallel in, serial out**: load the n bits into the flip-flops in one clock edge (a parallel load — see Figure 57), then read them off SO one bit per edge — the first bit is sitting at SO as soon as the load edge lands, and $n-1$ further edges bring the rest.

> **Note**
>
> The parallel outputs are simply the Q pins of the flip-flops already present in Figure 56, brought out as a group. Nothing extra is added to the circuit for serial-in/parallel-out at all.

Parallel load vs serial shift

A plain shift register can only be filled one bit per clock. To also allow **parallel load** — all n bits written at once from n parallel inputs — each flip-flop's D input is fed from a **2-to-1 multiplexer** (see combinational-logic-blocks[194]) instead of directly from the previous stage:

[194] courses/csse2010/combinational-logic-blocks.pdf

Mux select	Mux data input chosen	Effect on that stage
"shift"	the previous stage's Q (or SI for the first stage)	serial shift by one place
"load"	that stage's parallel data input I_i	parallel load of the whole word in one clock edge

The select line is common to all the muxes, so on each clock edge the register either shifts by one place or loads the whole parallel word.

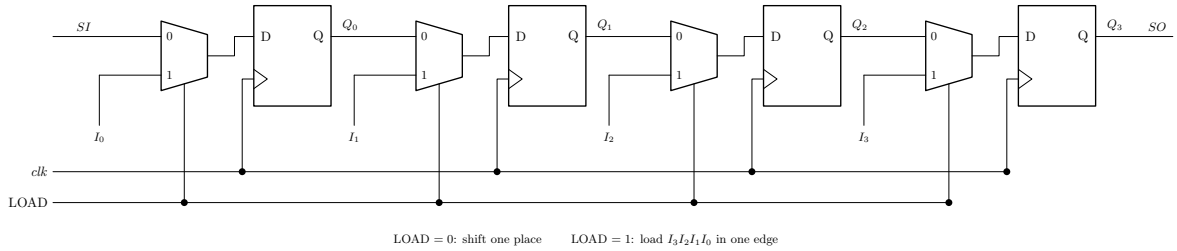


Figure 57: The parallel-load shift register: one 2-to-1 multiplexer in front of every flip-flop's D input, with a single LOAD line steering all four at once. Follow the top mux input across and it is the plain chain of the previous figure; follow the bottom input and every stage is loaded from its own I pin in one edge. The Q pins do double duty as the parallel output. Which mux input is which is a free choice: here LOAD equals 0 for shift and 1 for load.

Adding a hold. The same trick gives a register an *enable*: widen each mux by one input and wire that input to the stage's own Q, so on a "hold" edge every flip-flop reloads the value it already has and the word does not move. This is the standard way to stop a register without gating the clock — gating the clock risks glitching it — and it is what a "recirculating" input means. It follows from the mux construction, and a past-paper question turns on it (csse2010-sample-midsem-1[195] Q18, where each mux's input 0 is its own stage's Q looped back, so the control line chooses shift or hold).

The mux-based bidirectional shift element

A single building block makes a shift register that shifts in **either** direction (drawn in the lecture note[196]):

- a **2-to-1 multiplexer** whose output drives the D input of one D flip-flop;
- the mux's select line is a control signal called **DIRN**;

[195] courses/csse2010/csse2010-sample-midsem-1.pdf
[196] courses/csse2010/2026-08-13-shift-registers.pdf

- **DIRN = 0** selects mux input **0** → **left shift**;
- **DIRN = 1** selects mux input **1** → **right shift**;
- the flip-flop's Q is that stage's output, and the flip-flop is clocked from the common clock.

Chaining n of these elements and wiring, for each stage, mux input 0 to the neighbour on the left-shift side and mux input 1 to the neighbour on the right-shift side gives an n-bit **bidirectional shift register**: one shared DIRN line decides which way the whole register shifts on the next clock edge.

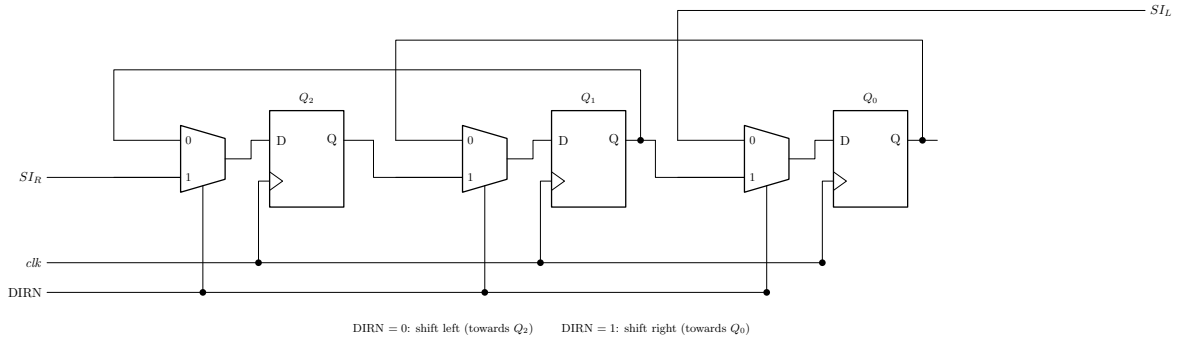


Figure 58: A 3-bit bidirectional shift register: three copies of the DIRN element, one shared DIRN line and one shared clock. Q2 is the most significant bit, on the left. Mux input 0 of every stage comes from the stage on its right (left shift, bits moving towards Q2); mux input 1 comes from the stage on its left (right shift). The two ends need a serial input each, because on each direction one end has no neighbour to take a bit from. Wires that cross without a dot are not connected.

For a 3-bit register with bits $Q_2Q_1Q_0$ (Q_2 most significant), taking *left shift* to mean bits move towards the more-significant end:

Stage	Mux input 0 (DIRN=0, left shift)	Mux input 1 (DIRN=1, right shift)
Q_2	Q_1	SI_R (serial input for right shifts)
Q_1	Q_0	Q_2
Q_0	SI_L (serial input for left shifts)	Q_1

One clock edge applied to a 3-bit register holding $Q_2Q_1Q_0 = 110$, with both serial inputs held at 0:

DIRN	Operation	Before ($Q_2Q_1Q_0$)	After ($Q_2Q_1Q_0$)
0	left shift	110	100
1	right shift	110	011

See 2026-08-13-shift-registers[197] for the exercise that asks for this register.

The configurations a shift register can be wired into

There are **seven named configurations**, parts (a) to (g) of a figure from Floyd, recapped at the start of Lecture 7[198]:

- serial in / shift right / serial out
- serial in / shift left / serial out
- parallel in / serial out
- serial in / parallel out
- parallel in / parallel out
- rotate right
- rotate left

The first six are all the same flip-flop chain differing only in *where each D comes from* and *which pins are brought out*: adding a mux per stage buys the parallel input, tapping the Q pins buys the parallel output, and swapping the direction of the chain buys left instead of right. The last two, the **rotates**, are the chain with its serial output wired back to its serial input, so no bit is lost and the word cycles.

Universal shift register

A **universal shift register** is the combination of all of the above capabilities in one device: it can hold its value, shift left, shift right, and be parallel-loaded, with the operation selected by control inputs.

Structurally it is the two constructions above stacked: four operations need a **4-to-1 mux per flip-flop** and **two select lines**, with the four mux inputs being the stage's own Q (hold), its left neighbour, its right neighbour, and its parallel input I_i.

Wide (multi-bit) shift registers

A shift register does not have to shift one *bit* at a time — it can shift a whole multi-bit word per clock edge. An example is an **8-bit wide, 4-stage queue**:

[197] courses/csse2010/2026-08-13-shift-registers.pdf
[198] courses/csse2010/2026-08-17-counters.pdf

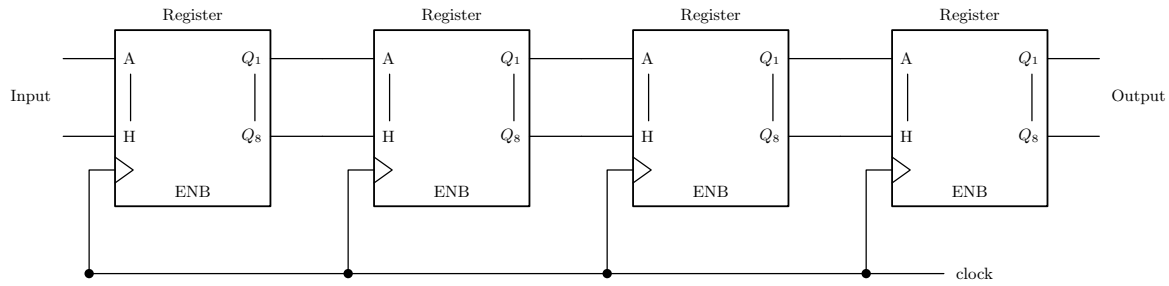


Figure 59: The 4-stage 8-bit queue. Each block is a whole 8-bit register rather than a single flip-flop: the vertical line between A and H, and between Q1 and Q8, is shorthand for the six pins in between. Functionally identical to the 1-bit chain — one stage's outputs are the next stage's inputs, everything on one clock — but each stage now holds a byte, so a byte presented at Input reaches Output four clock edges later.

- four **registers** in a row, each 8 bits wide (inputs labelled A through H, outputs labelled Q_1 through Q_8);
- the 8-bit `Input` bus feeds the first register's 8 data inputs;
- each register's 8 Q outputs feed the next register's 8 data inputs, as a bus;
- the last register's outputs are the 8-bit `Output`;
- all four registers share a **common clock** line, and each has an `ENB` (enable) input.

Functionally this is the same chain as the 1-bit shift register, but each "stage" holds a byte rather than a bit — so it behaves as a 4-deep queue of bytes: a byte presented at `Input` appears at `Output` four clock edges later.

This is also the shape the **Lab 06 combination lock** takes, at a smaller size: a two-stage, 4-bit-wide shift register, so that each press of the "enter" button moves a whole digit from the first stage to the second. See the lecture note[199] and week4-lab-sequential-circuits-shift-registers[200].

Related

- flip-flops-and-latches[201] — the D flip-flop these are built from
- sequential-circuits[202] — state, present/next state, synchronous clocking
- combinational-logic-blocks[203] — the multiplexer used for load/direction selection
- [counters] — the other classic thing built from a row of D flip-flops

[199] courses/csse2010/2026-08-13-shift-registers.pdf
[200] courses/csse2010/week4-lab-sequential-circuits-shift-registers.pdf
[201] courses/csse2010/flip-flops-and-latches.pdf
[202] courses/csse2010/sequential-circuits.pdf
[203] courses/csse2010/combinational-logic-blocks.pdf

- serial-communication[204] · avr-usart[205] — a shift register at each end of a serial link, turning bytes into a stream of bits and back
- avr-spi[206] — two 8-bit shift registers joined in a ring, so that a byte goes each way at once
- device-pinouts[207] — 74HCT175 quad D flip-flop and 74HCT157 quad 2-input multiplexer pinouts
- csse2010-diagram-notation[208] — the drawing conventions the figures above are checked against

Timing Diagrams

A **timing diagram** is the fourth of the four equivalent ways to describe a logic function, alongside the **logic diagram**, the **truth table**, and the **Boolean expression** (see logic-gates[209]).

Timing diagram as a logic representation

It is like a truth table, but in **graphical format**: time runs left to right along the horizontal axis, and each signal gets its own horizontal track that steps between the logic 0 and logic 1 levels (drawn as two dashed reference lines).

Worked example — an inverter. The input waveform starts at logic 0, rises to logic 1 for a period, then falls back to logic 0. The output waveform is the mirror image: it starts at logic 1, falls to logic 0 for exactly that period, then returns to logic 1. Each **change in the input causes the output to change**.

At this idealised level the output changes at exactly the same instant as the input; the transitions are drawn as vertical lines. That is a convenient fiction, corrected at the bottom of this note.

Convention for drawing the input waveforms

The input waveforms must be drawn so that **all possible input combinations are covered** — that is what makes the diagram equivalent to the full truth table.

The standard construction for a 2-input gate divides the time axis into four equal intervals and drives the inputs like a binary count:

[204] courses/csse2010/serial-communication.pdf
[205] courses/csse2010/avr-usart.pdf
[206] courses/csse2010/avr-spi.pdf
[207] courses/csse2010/device-pinouts.pdf
[208] courses/csse2010/csse2010-diagram-notation.pdf
[209] courses/csse2010/logic-gates.pdf

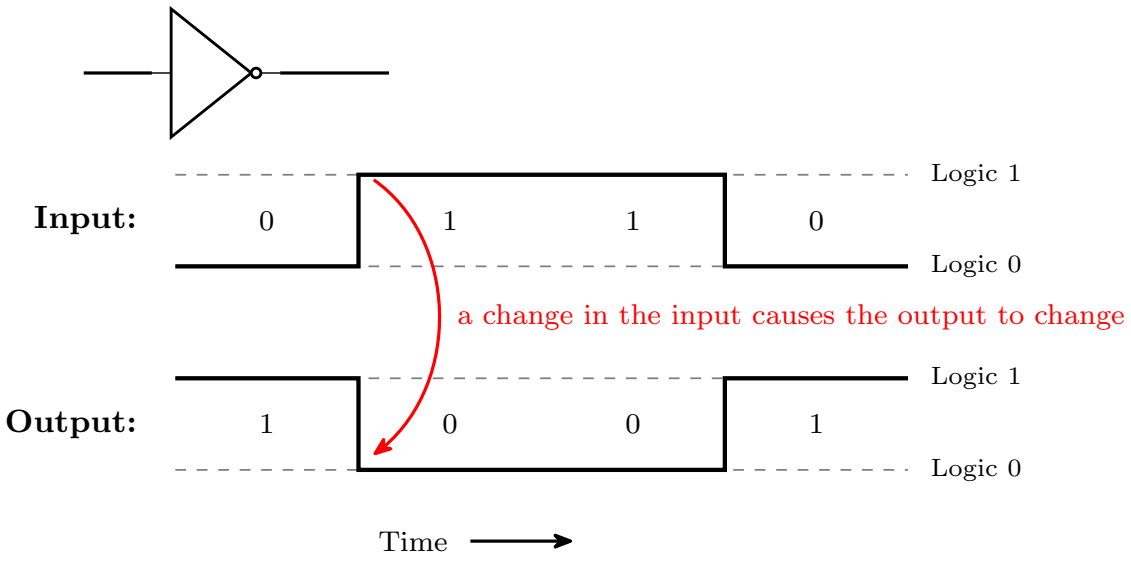


Figure 60: An inverter drawn as a timing diagram. Reading the four intervals off the input gives 0, 1, 1, 0 and off the output gives 1, 0, 0, 1 — which is the inverter's truth table, just laid out along a time axis instead of down a page. The two dashed lines per track are the logic 0 and logic 1 reference levels.

- A is low for the first half of the diagram, then high for the second half.
- B toggles at twice that rate: low, high, low, high.

Reading the four intervals left to right therefore gives $AB = 00, 01, 10, 11$ — every combination, once each, in truth-table order. For n inputs the same scheme generalises: input i toggles at half the rate of input $i + 1$, and the diagram is 2^n intervals wide.

To read a gate's output off the diagram, work interval by interval: identify the AB combination in that interval, look up the gate's truth table row, and draw the output at that level for the width of the interval.

Going the other way — being *given* the output track and asked which gate it is — is the standard exam form of this. Read the output value in each interval, write it beside the corresponding AB row, and compare the resulting column with the gates you know. A worked instance is in 2026-08-06-combinational-logic[210].

[210] courses/csse2010/2026-08-06-combinational-logic.pdf

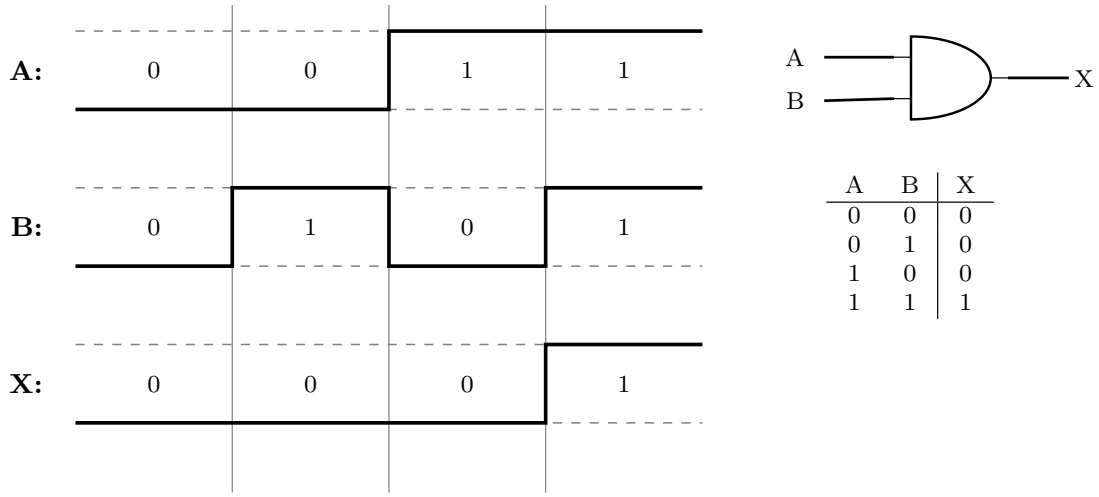


Figure 61: The convention on a 2-input example, with the AND output filled in. The vertical rules divide the four intervals; the digits under each track are the truth-table row for that interval. Because the inputs count 00, 01, 10, 11, the output track is literally the output column of the truth table, drawn left to right.

Real gates aren't perfect: the reality of timing

The idealised diagrams above show instantaneous, perfectly square transitions. Real gates do not behave that way — real waveforms have sloped edges, and the output changes some time *after* the input does. Three quantities describe this:

Term	Definition
Propagation delay	Time for a change in the input to affect the output.
Fall time	Time taken for the output to fall from 1 to 0.
Rise time	Time for the output to rise from 0 to 1.

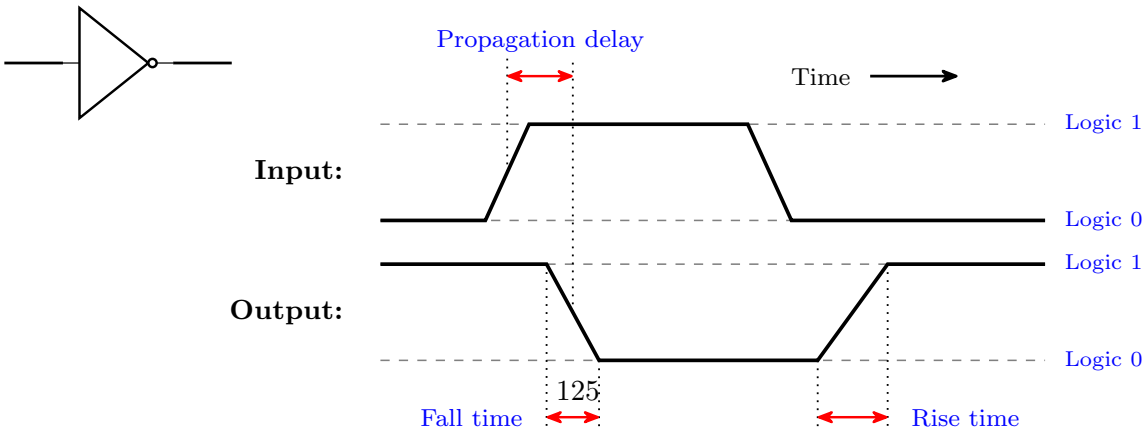


Figure 62: The same inverter drawn honestly. The input's edges slope instead of jumping; the output does not begin to move until the propagation delay has elapsed; and once moving, its downward edge takes the fall time and its later upward edge takes the rise time. Rise and fall times are properties of one edge of one signal — the propagation delay is the gap between two signals.