CSSE2010 — Week 4 Notes

Table of contents

Lecture 7 — Counters **2**
- Outline . . . 3
- Schedule . . . 3
- Worked: shift direction control . . . 3
- Shift register types . . . 5
- Counters . . . 5
- Digital system application . . . 5
- One-bit counter . . . 6
- State . . . 7
- Counter example — the 2-bit binary up counter . . . 8
- Worked: what sequence does this counter count through? . . . 8
- Key points . . . 11
- Worked example — a non-binary sequence with a don't-care . . . 11
- Next: state machines . . . 12
- Reminders . . . 14

Lecture 8 — State Machines **14**
- Outline . . . 14
- Sequential circuits . . . 14
- State machines . . . 15
- Moore vs Mealy . . . 16
- State diagrams . . . 16
- State tables . . . 23
- State encoding . . . 24
- Sequence detector example . . . 25
- Steps involved in FSM design . . . 25
- See also . . . 26

Lab 7 Exercises **26**
- In the lab session . . . 28

Working . . . 28
Circuit Schematic . . . 31
Peer review of the schematic . . . 31
Task 2 — a 4-bit up/down counter built from an adder . . . 34

Sequential Circuits 1 — Shift Registers **37**
Lab plan . . . 37
Ed Lessons — C is starting . . . 37
Exercise — the four-mode shift register . . . 37
Discussing your diagrams . . . 40
Combination lock tasks . . . 41
Next . . . 43
See also . . . 43

Reference material **43**
Binary Number Representations . . . 43
Boolean Algebra . . . 52
Circuit Schematics . . . 54
Combinational Logic Blocks . . . 58
Counters . . . 68
CSSE2010 Diagram Notation . . . 74
Device Pinouts . . . 90
Flip-Flops and Latches . . . 100
The I/O Board . . . 108
Sequential Circuits . . . 116
Shift Registers . . . 120
State Machines . . . 128

Lecture 7 — Counters

See [counters] for the reference material on counters, and shift-registers[1], sequential-circuits[2] and flip-flops-and-latches[3] for the background.

Counter design is a procedure: write the sequence down as a state table, read each D column off as a Boolean function of the Qs, then build it. Three circuits carry that idea — the one-bit toggle, the 2-bit binary up counter, and a 2-bit counter for a non-binary sequence with a don't-care state.

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

Outline

- A worked question on the mux-per-stage shift register
- Shift register types
- Counters, and binary counters
- A digital system application built around a counter
- State: current state, next state, and the D inputs
- Worked counter examples
- Next: state machines

Schedule

From the course teaching outline:

- **Week 4, Mon-Tue 17-18 Aug** — this lecture, *Lecture 7: Counters* (17 Aug).
- **Week 4, Thu-Fri 20-21 Aug** — *Lecture 8: Finite State Machines* (20 Aug).
- Learning labs this week: **Lab 6 Shift Registers** (P2, Mon-Tue) and **Lab 7 Counters** (P1, Thu-Fri).
- Week 5: *Lecture 9 ALU and Control Unit* (24 Aug), *Lecture 10 Introduction to AVR and Assembly Language* (27 Aug), with labs 8 (Finite State Machines) and 9 (AVR Assembly Programming 1).

Later in the schedule: the week 6 Thu-Fri **revision and help sessions for the lab exam**, and the week 7 **Lab Exam (Digital Logic)**, held during the scheduled lab sessions. Everything from this lecture onwards is lab-exam material three weeks later.

Worked: shift direction control

> Which of the following statements about the circuit below is true?
>
> A. When A is 1, flip-flop values are shifted to the right B. When A is 0, flip-flop values are shifted to the right C. When A is 0, flip-flop values stay the same D. When FN is 1, flip-flop values are shifted to the left E. When FN is 0, flip-flop values stay the same F. When FN is 0, flip-flop values are shifted to the left

The circuit is the mux-per-stage construction from last lecture[4], two stages deep.

Reading Figure 1 off, with Q_a = stage 1 and Q_b = stage 2:

[4] courses/csse2010/2026-08-13-shift-registers.pdf
[5] courses/csse2010/csse2010-diagram-notation.pdf

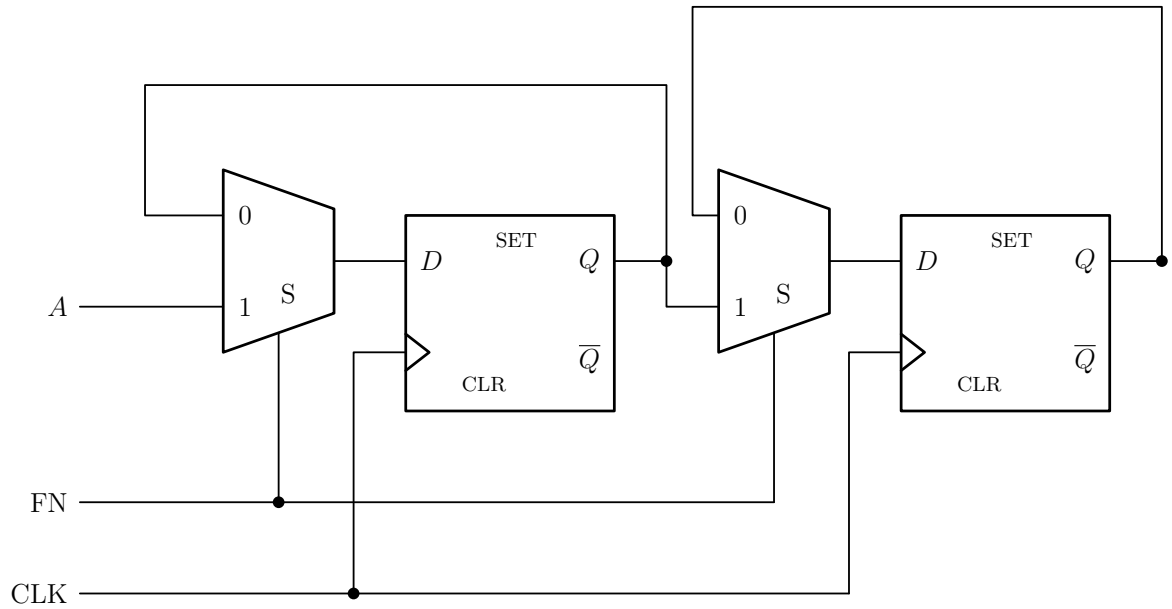


Figure 1: A two-stage shift register with a 2-to-1 mux in front of every flip-flop. Each mux's `0` input is fed back from *its own* flip-flop's Q — that is the hold path — and its `1` input takes the value from the stage before it (A from outside, for stage 1). One line, FN, selects both muxes, so FN enables the shift rather than choosing a direction; there is no left-shift path anywhere in the circuit. Each mux is drawn wide-edge-left with the select entering the bottom sloping edge (see csse2010-diagram-notation[5]). The one crossing, the CLK riser over the FN rail, carries no junction dot and so is not a connection.

$$D_a = \begin{cases} A & FN = 1 \\ Q_a & FN = 0 \end{cases} \qquad D_b = \begin{cases} Q_a & FN = 1 \\ Q_b & FN = 0 \end{cases}$$

- When $FN = 1$, A is loaded into stage 1 and stage 1's value moves into stage 2 — data shifts **left-to-right along the diagram**, one stage per clock.
- When $FN = 0$, each flip-flop's D input is its own Q, so every flip-flop reloads the value it already had — **nothing changes**.

FN is therefore a shift *enable*, not a direction control, so every option mentioning "shifted to the left" is out; and A is plain data rather than a control signal, so A/B/C are out too.

Answer: E — when FN is 0, flip-flop values stay the same.

Shift register types

There are seven shift-register configurations: serial in/shift right/serial out, serial in/shift left/serial out, parallel in/serial out (**PISO**), serial in/parallel out, parallel in/parallel out, rotate right and rotate left. Rotate right is a right shift with a wrap-around path.

Getting `0101` into a 4-bit register takes four clocks when it is shifted in serially (a **serial load**) and a single clock when it is loaded in parallel: n clocks versus one. The configurations and the clock-by-clock trace are in shift-registers[6].

Counters

An n-bit binary counter has n flip-flops and 2^n states, and counts 0 to $2^n - 1$. The definition of a counter and the 4-bit binary counting sequence table are in [counters]. The sequence runs `000…0`, `000…1`, , `111…1` and then back to `000…0`, which is 2^n states. The state count is the *length of the cycle*, not just the width of the register.

Digital system application

Where a counter sits in a real system — an automated tablet-bottling line (the example is from Floyd's *Digital Fundamentals*):

- A **keypad** for the number of tablets per bottle feeds an **encoder**, which feeds **Register A**. Register A drives **Decoder A** and a 7-segment display showing the tablets-per-bottle setting, and also a **code converter** producing the binary code for the preset number.

[6] courses/csse2010/shift-registers.pdf

- On the conveyor, a **sensor** emits one pulse per tablet passing the valve. Those pulses are the clock of a **counter**, so the counter holds the binary code for the actual number of tablets in the bottle. A separate pulse **resets the counter to zero** when the next bottle is in place.
- A **comparator** tests the counter value against the preset value. Its output goes HIGH when $A = B$: HIGH closes the valve and advances the conveyor, LOW keeps the valve open.
- The counter value also feeds an **adder**, whose other input is the running total held in **Register B**; the comparator's HIGH also causes the new sum to be stored back into Register B. Register B drives **Decoder B** (total display) and a **MUX/DEMUX** path that sends the accumulated total on to a computer for storage.

The counter, adder, register, decoder, comparator, encoder, mux and demux blocks covered so far in the course compose into a complete system — see combinational-logic-blocks[7] for the mux/decoder blocks and 2026-08-04-binary-arithmetic[8] for the adder.

One-bit counter

The circuit is a single D flip-flop, clocked on CLK, with a wire from its $\overline{Q}$ output looped back around to its D input.

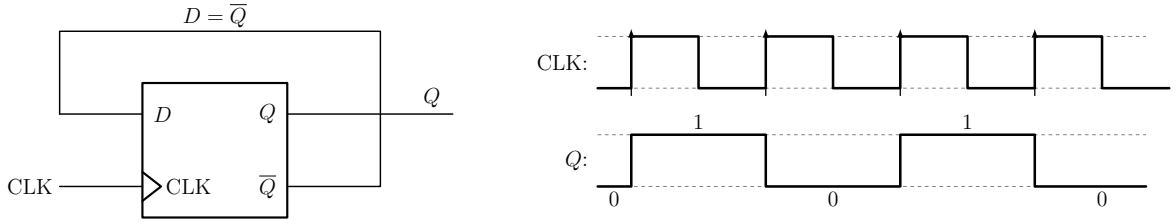


Figure 2: Left: one D flip-flop with $\overline{Q}$ looped back to D, so the next state is always the complement of the present state. The Q output line crosses the feedback wire with no junction dot, i.e. no connection. Right: the waveform, with an arrow on each rising clock edge and the value of Q, `0` or `1`, written on each segment. Q toggles on every rising edge, so the flip-flop divides the clock frequency by two.

The derivation takes three steps. First the equation: $D = \overline{Q}$. Then the roles: Q, the flip-flop's stored value, is the **present state**; D, what is sitting on the input at the clock edge, is the **next state**. Then those two written out as columns give the **state table**:

[7] courses/csse2010/combinational-logic-blocks.pdf
[8] courses/csse2010/2026-08-04-binary-arithmetic.pdf

present state Q	next state D
0	1
1	0

Since D is the next state and $D = \overline{Q}$, the value inverts on every clock edge and the counter counts

$$0 \to 1 \to 0 \to 1 \to \ldots$$

which is exactly the $n = 1$ binary counter ($2^1 = 2$ states, counting 0 to $2^1 - 1 = 1$). This is the standard *toggle* configuration, and the two-row table above is the smallest possible instance of the method the larger counters below use.

> **i** Present state or current state?
>
> **Current state** and **present state** are synonyms. The course uses both and settles neither; see csse2010-diagram-notation[9].

State

Three statements make counter design mechanical. Take two D flip-flops on a common CLK, with inputs D_1 and D_0 still to be worked out:

- the values stored in the flip-flops are the **current state** of the circuit;
- the D inputs are the **next state**;
- the D inputs are some (combinational) **function of the current state and the inputs**.

So each D input is driven by its own combinational circuit, and each of those circuits is fed from the Q outputs. That is the whole architecture — flip-flops hold the state, a block of combinational logic computes the next one, and the clock steps between them. The design method that follows is written up as a recipe in [counters]; the underlying state terminology is in sequential-circuits[10].

[9] courses/csse2010/csse2010-diagram-notation.pdf

[10] courses/csse2010/sequential-circuits.pdf

Counter example — the 2-bit binary up counter

The problem is to **design** a **2-bit** counter for $00 \to 01 \to 10 \to 11 \to 00$. Design means a sequence is given and the circuit has to be found; the worked question after this one is analysis, where a circuit is given and the sequence has to be found.

Start from two D flip-flops on a common CLK with **nothing connected** to their D inputs, and a current-state/next-state table with columns $Q_1, Q_0 \mid D_1, D_0$ and a row for each of the four states. Filling in the next state for each row gives the **state table**:

Q_1	Q_0	D_1	D_0
0	0	0	1
0	1	1	0
1	0	1	1
1	1	0	0

Reading the two next-state columns as Boolean functions of Q_1 and Q_0 gives two equations:

$$D_0 = \overline{Q_0} \qquad\qquad D_1 = Q_1 \oplus Q_0$$

D_0 is 1 exactly when Q_0 is 0, so bit 0 toggles every clock — it is the one-bit counter of Figure 2, unchanged. $D_1 = Q_1 \oplus Q_0$ means bit 1 takes its own value flipped whenever $Q_0 = 1$ and keeps it whenever $Q_0 = 0$ — which is the carry rule for binary counting: bit 1 flips exactly on the clocks where bit 0 rolls over from 1 to 0.

As a check, step the circuit through a clock train and write down the state at each rising edge: 00, 01, 10, 11, 00. The circuit walks the sequence it was designed from. This is the direct model for the 3-bit design in Lab 7.

Worked: what sequence does this counter count through?

> What sequence does the counter below count through? (Assume Q_1Q_0 starts at 00.)
>
> A. $00 \to 11 \to 10 \to 01 \to 00$ B. $00 \to 10 \to 11 \to 01 \to 00$ C. $00 \to 11 \to 01 \to 10 \to 00$ D. $00 \to 01 \to 11 \to 10 \to 00$ E. $00 \to 01 \to 10 \to 11 \to 00$ F. $00 \to 10 \to 01 \to 11 \to 00$

This is the design problem run backwards: the circuit is given, the sequence has to be recovered.

[11] courses/csse2010/csse2010-diagram-notation.pdf

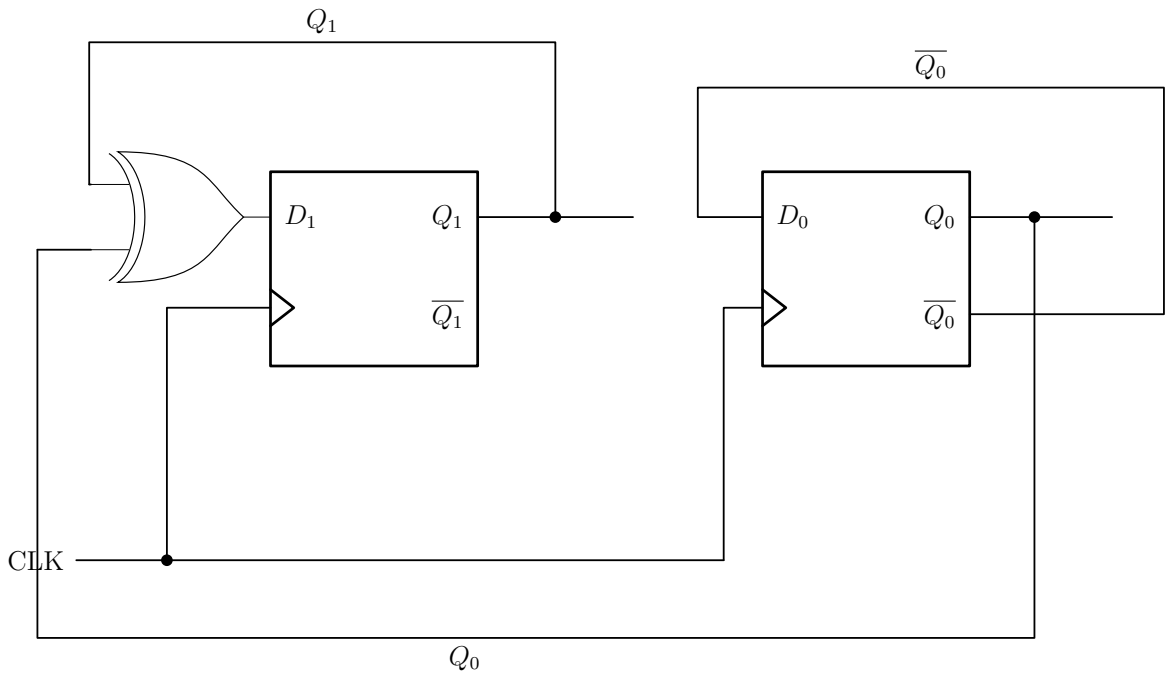


Figure 3: The 2-bit synchronous binary up counter for $00 \to 01 \to 10 \to 11 \to 00$. $\overline{Q_0}$ loops back to D_0 (bit 0 toggles every clock); an XOR of Q_1 and Q_0 drives D_1. Both flip-flops share one clock, which is what makes the counter synchronous. The gate is drawn the conventional way round — inputs on the left, output on the right — a rule the course schematics guide states and that holds without exception (see csse2010-diagram-notation[11]). The feedback to D_0 is taken from the $\overline{Q}$ pin at the lower right of the flip-flop.

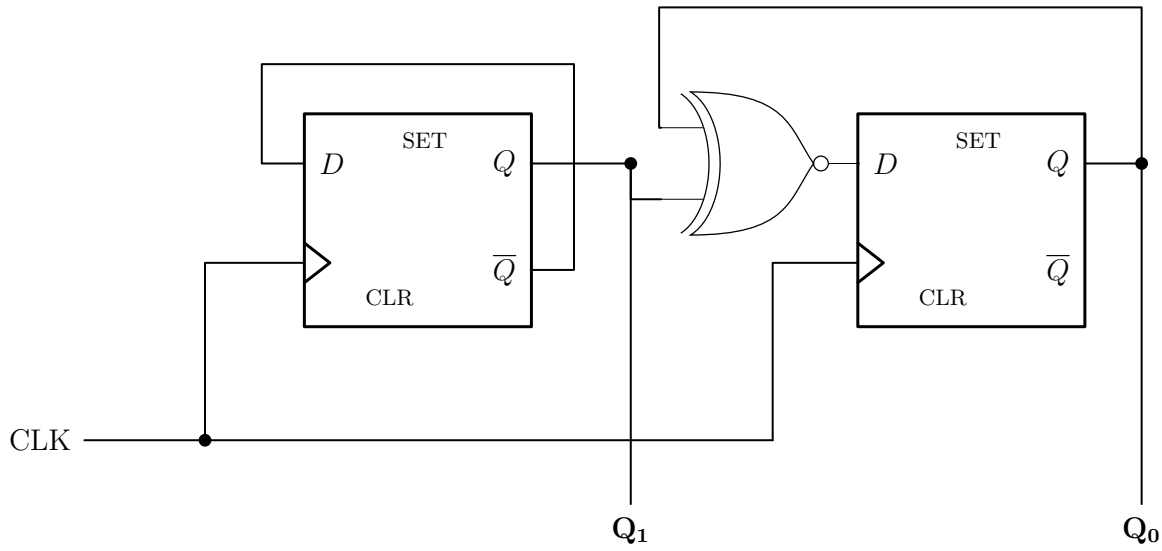


Figure 4: The counter to analyse. The left flip-flop's $\overline{Q}$ is wired back to its own D, so $D_1 = \overline{Q_1}$ and Q_1 toggles every clock. The right flip-flop's D is driven by a two-input **XNOR** — an OR body with the extra input-side arc *and* an inversion bubble on the output — whose inputs are Q_1 and Q_0, so $D_0 = \overline{Q_1 \oplus Q_0}$. Everything in the answer turns on that gate symbol. SET and CLR are lettered inside the boxes and are unused.

Tabulating from Figure 4:

Q_1	Q_0	$D_1 = \overline{Q_1}$	$D_0 = \overline{Q_1 \oplus Q_0}$	next state
0	0	1	1	11
1	1	0	1	01
0	1	1	0	10
1	0	0	0	00

Starting from 00: $00 \to 11 \to 01 \to 10 \to 00 \to ...$

Answer: C.

The gate reading rests entirely on the **symbol** — the extra input-side arc plus the output bubble is an XNOR, not a NOR. Don't try to settle it by elimination: an XOR reading gives $D_0 = Q_1 \oplus Q_0$, $D_1 = \overline{Q_1}$ and the sequence $00 \to 10 \to 01 \to 11 \to 00$, which *is* one of the listed options (F).

This counter is **not** a binary counter. Its sequence is a legitimate four-state cycle that simply isn't the binary count — exactly the case the second key point below covers.

Key points

Four key points, all recorded in [counters]:

- the **next state is a function of the previous 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*;
- these circuits are **synchronous**: all flip-flops share the same clock;
- **asynchronous** counters exist as well, but the course does not discuss them.

Worked example — a non-binary sequence with a don't-care

The problem:

> Design a 2-bit counter that counts $00 \to 10 \to 01 \to 00 \to$

Step 1 — the state table. Present state on the left, next state on the right. The fourth row, present state 11, is an **unused state**. Its next-state entries are written X_1X_0, where an X is a **don't care term**:

Q_1	Q_0	D_1	D_0	
0	0	1	0	$00 \to 10$

Q_1	Q_0	D_1	D_0	
1	0	0	1	$10 \to 01$
0	1	0	0	$01 \to 00$
1	1	X_1	X_0	unused state

Step 2 — spend the don't-cares. Write down *both* choices for each column, then pick:

$$D_0: \quad X_0 = 0 \;\Rightarrow\; D_0 = Q_1\overline{Q_0} \qquad X_0 = 1 \;\Rightarrow\; D_0 = Q_1 \quad \checkmark$$
$$D_1: \quad X_1 = 0 \;\Rightarrow\; D_1 = \overline{Q_1}\,\overline{Q_0} \quad \checkmark \qquad X_1 = 1 \;\Rightarrow\; D_1 = \overline{Q_1 \oplus Q_0} \quad \times$$

$X_0 = 1$ is chosen because it is strictly simpler: adding the 11 row to the single 10 row lets Q_0 drop out entirely and leaves $D_0 = Q_1$, a bare wire. $X_1 = 1$ is **rejected** even though it also looks tidy — an XNOR is one gate, same as an AND. The reason is the next step.

Step 3 — check the unused state. With $X_1X_0 = 01$ the transitions are

$$00 \to 10 \to 01 \to 00, \qquad 11 \to 01$$

This is a **robust design**: the unused state falls straight back into the cycle, so a power-up glitch into 11 costs one clock and nothing more. Taking $X_1 = 1$ instead would have given $11 \to 11$ and the counter would have hung there forever — which is why that option is rejected. The final equations are

$$D_1 = \overline{Q_1}\,\overline{Q_0} \qquad D_0 = Q_1$$

As a check, step the circuit through a clock train again: the state at each rising edge is 00, 10, 01, 00, and from 11 it drops into 01. The full write-up of this example, including the sum-of-products working, is in [counters]; week4-lab-lab-7-exercises[13] is the same procedure at $n = 3$.

Next: state machines

More sequential circuits come next: **state machines**. In the general synchronous sequential circuit, inputs and fed-back state go into a combinational circuit, whose outputs are both the circuit outputs and the flip-flop inputs; the flip-flops are driven by clock pulses and their outputs are fed back (see sequential-circuits[14]).

[12] courses/csse2010/csse2010-diagram-notation.pdf
[13] courses/csse2010/week4-lab-lab-7-exercises.pdf
[14] courses/csse2010/sequential-circuits.pdf

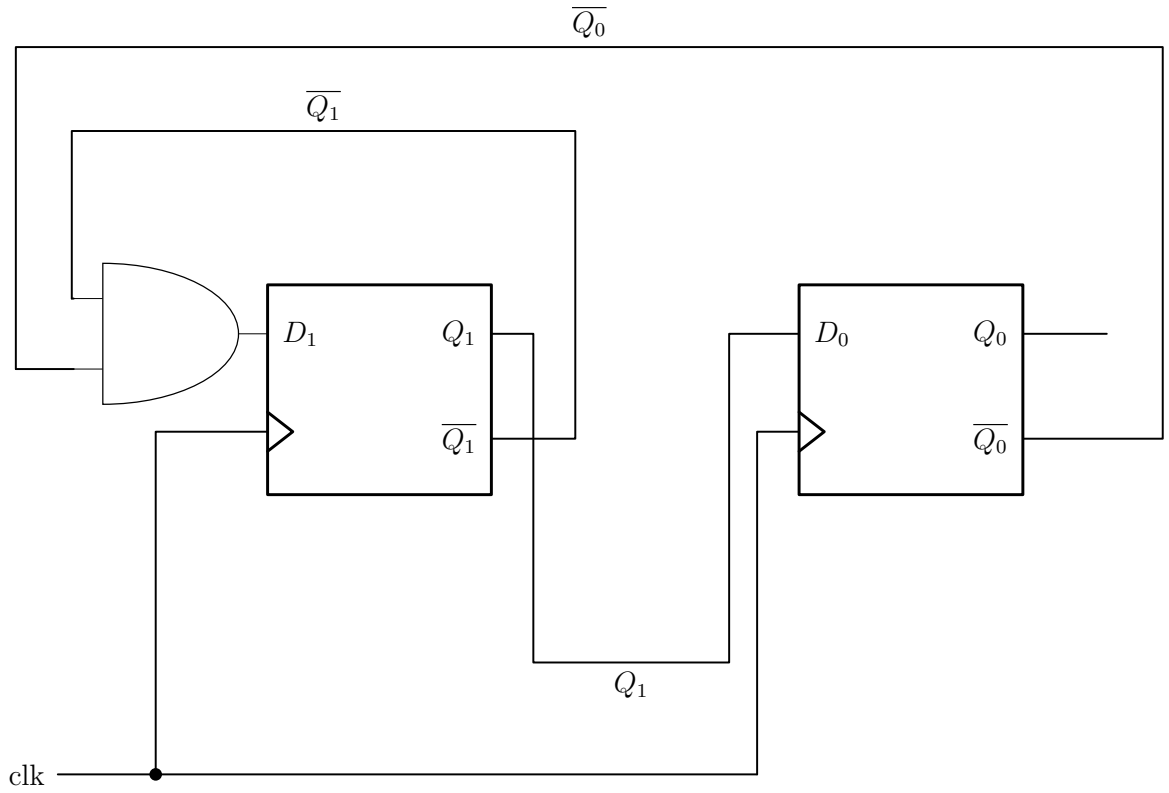


Figure 5: The implementation of $D_1 = \overline{Q_1}, \overline{Q_0}$ and $D_0 = Q_1$ for $00 \to 10 \to 01 \to 00$. One AND gate over the two complement outputs is the whole of the combinational logic; D_0 needs no gate at all, being Q_1 wired straight across. As in Figure 3 the gate is drawn inputs-left/output-right. The clock is lettered `clk` here and `CLK` in the other figures; the course uses both and settles neither (see csse2010-diagram-notation[12]).

A counter may or may not have external inputs. This is the link between the two: a counter is the special case of a state machine whose transition logic reads nothing but the state itself, which is why none of the four counter circuits above has an input other than the clock. See state-machines[15] and 2026-08-20-state-machines[16].

Reminders

- **Lab 7 (Counters)** runs in the P1 sessions this week (Thu-Fri) — the pre-lab schematic for the 3-bit synchronous counter must be drawn beforehand. See week4-lab-lab-7-exercises[17], and device-pinouts[18] for the flip-flop packages.
- Lab 6 (Shift Registers) runs in the P2 sessions (Mon-Tue) this week.
- The centrally scheduled 90-minute in-semester theory exam is on **Saturday 5 September**, and the Digital Logic **lab exam is during the scheduled lab sessions in week 7**.

Lecture 8 — State Machines

See state-machines[19] for the reference material — Moore vs Mealy, the diagram and table notation, encoding schemes and the design procedure — and sequential-circuits[20], flip-flops-and-latches[21] and [counters] for the background. The drawing conventions for state diagrams and state tables are catalogued in csse2010-diagram-notation[22].

Outline

1. State diagrams
2. State tables
3. State encoding

Sequential circuits

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

[15] courses/csse2010/state-machines.pdf
[16] courses/csse2010/2026-08-20-state-machines.pdf
[17] courses/csse2010/week4-lab-lab-7-exercises.pdf
[18] courses/csse2010/device-pinouts.pdf
[19] courses/csse2010/state-machines.pdf
[20] courses/csse2010/sequential-circuits.pdf
[21] courses/csse2010/flip-flops-and-latches.pdf
[22] courses/csse2010/csse2010-diagram-notation.pdf

In the block diagram, the combinational circuit is the *logic gates*:

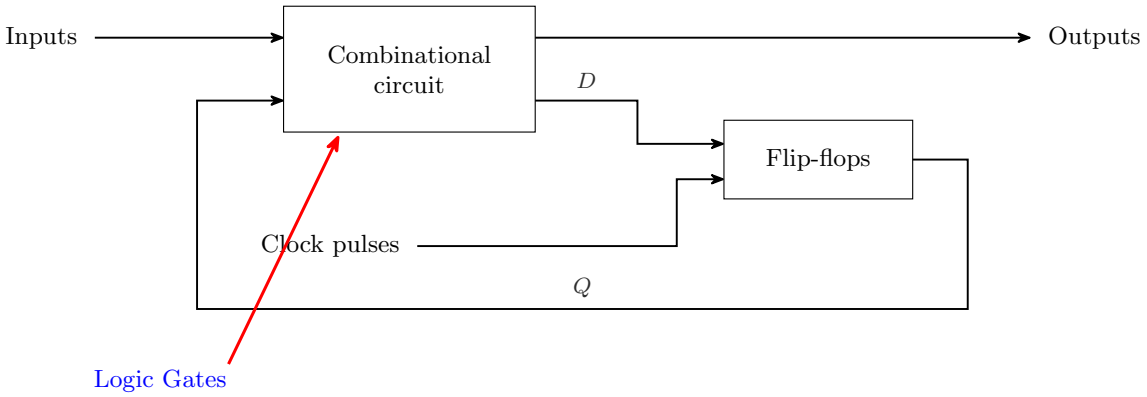


Figure 6: The sequential-circuit block diagram (from Mano, *Digital Design*, 3rd ed.). Flow is left to right with the state fed back around the outside, and the clock enters on its own stub reaching only the flip-flops — never the gates. The red arrow and the blue Logic Gates label mark the combinational circuit as the logic gates. D and Q name the two feedback-side wires, which is what makes the two functions below readable off the picture.

Writing Q for the present state (PS) and D for the next state (NS), the one box marked *Combinational circuit* is really two functions computed side by side:

$$\underbrace{\text{OUT} = g(\text{IN}, Q)}_{\text{output logic}} \qquad \underbrace{D = f(\text{IN}, Q)}_{\text{transition logic}}$$

The clock pulses reach only the flip-flops, never the gates — see sequential-circuits[23].

"Output depends on **input and state**" is the *general* sequential-circuit statement, i.e. the Mealy form. A Moore machine is narrower: the output depends on the state alone. Every state machine drawn in this course from here on obeys the narrower rule.

State machines

Sequential circuits can also be called **state machines**, or **finite state machines (FSMs)**. A state machine has a finite number of possible states, is in **only one current state** at a time, and can **transition** to other states based on the inputs and the current state.

N flip-flops give 2^N states.

[23] courses/csse2010/sequential-circuits.pdf

Defining the states from the problem

The first step in a design is not a circuit at all — it is deciding what the states *are*:

> A vending machine accepts 5 cent and 10 cent coins, and dispenses a candy when it receives 15 cents in total. What can be the different states?

Tabulate the running total:

Coins inserted	0	5	10	15	20	50
Candy?					+ change	—

So the states are the amounts accumulated so far. 15 dispenses; 20 dispenses and owes change (reachable by putting 10c on top of 10c). 50 is ruled out — with only 5c and 10c coins the machine dispenses long before it could accumulate that much, so it is not a reachable state.

The transferable move: ask *what must the circuit remember?*, and the answer is the state set.

Moore vs Mealy

- **Mealy** — outputs depend on the current state **and the inputs**.
- **Moore** — outputs depend **only** on the current state (the flip-flop values), so outputs can only change when the state changes.

A **Moore machine is a special case of a Mealy machine**, and **this course sticks to Moore machines**.

The two structures have the same three-block shape: combinational circuit → flip-flops → combinational circuit, with the state fed back to the first block. The only difference is one branch — in the Mealy machine the input line also runs forward into the *second* (output) combinational block; in the Moore machine it does not. Both are drawn in the "Moore and Mealy machines" section of state-machines[24].

State diagrams

The notation for a state diagram:

Reading it off:

- **A state is a circle divided by a horizontal chord — state name** above, **output value(s)** below. Nothing else goes inside.

[24] courses/csse2010/state-machines.pdf

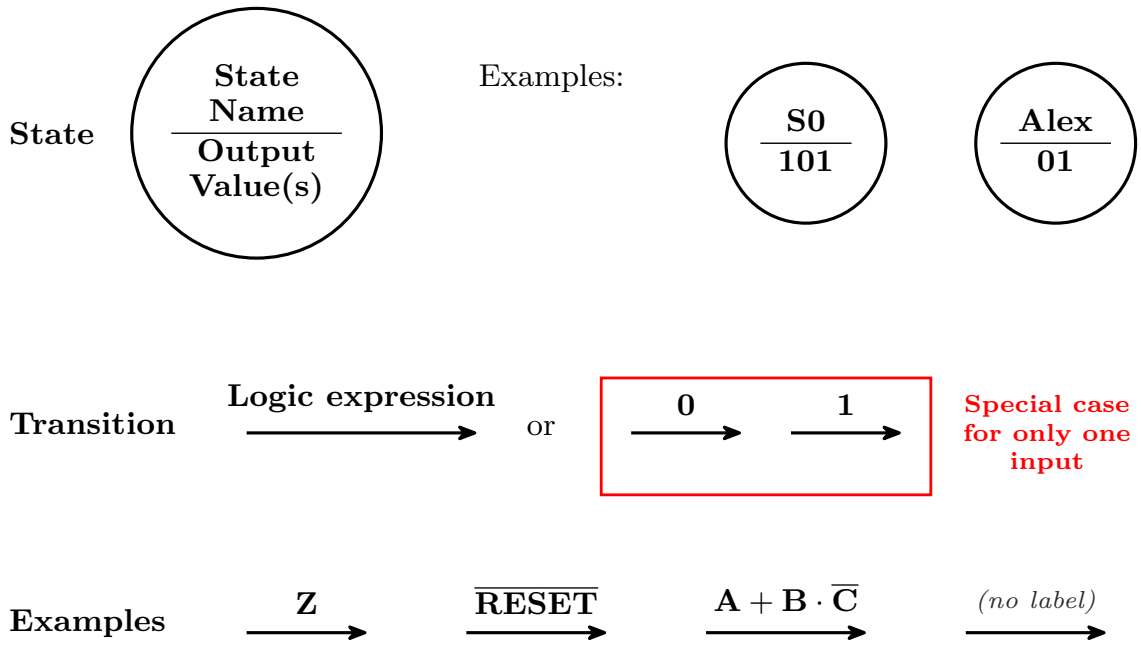


Figure 7: The state-diagram notation. A state is a circle split by a horizontal chord: state name above, output value(s) below. State names are free-form — a systematic S0 and an arbitrary word (Alex) are both allowed — and outputs are a bare bit string with no separators. A transition is a plain single-headed arrow labelled with a Boolean expression written above it; the red box marks the special case where a machine has exactly one input and the label degenerates to a bare 0 or 1. The bottom row gives example labels, including the unlabelled arrow, which means true.

- **Names are free-form.** S_0, S_1, … is a convention, not a rule; `Alex` is just as legal.
- **A transition is an arrow labelled with a logic expression.** If the expression is true, the transition is made. Label algebra is the ordinary Boolean notation of boolean-algebra[25]: overbar for NOT, $\cdot$ or juxtaposition for AND, `+` for OR.
- **No label means "true"** — i.e. always transition.
- **Exactly one input is a special case**: the label may be written as a bare `0` or `1` rather than $\overline{Z}$ or Z. This is the red-boxed case in the figure.

The notation leaves two things unsettled — see csse2010-diagram-notation[26]:

- **how to mark the start / reset state** (the diagrams below mark none: no incoming stub, no double circle, no `reset` label), and
- **how to draw a self-loop** (there is no set shape for one, although the completeness rule below guarantees most machines need them).

Worked: a two-state lock system

A lock with one input `key` and one output `locked`:

IN `key`	OUT `locked`
0	1
1	0

Example — binary counter with 2-bit output

A counter has no inputs at all beyond the clock, so **every arrow is unlabelled** — the machine advances one state per clock edge regardless.

Example — binary up/down counter, one input

One input U: **1 means count up, 0 means count down.** The two output sequences:

$U = 1$	00, 01, 10, 11, 00, 01, …
$U = 0$	11, 10, 01, 00, 11, 10, …

Because there is exactly one input, the one-input special case applies: these labels could equally be written as bare `1` and `0`.

[25] courses/csse2010/boolean-algebra.pdf
[26] courses/csse2010/csse2010-diagram-notation.pdf

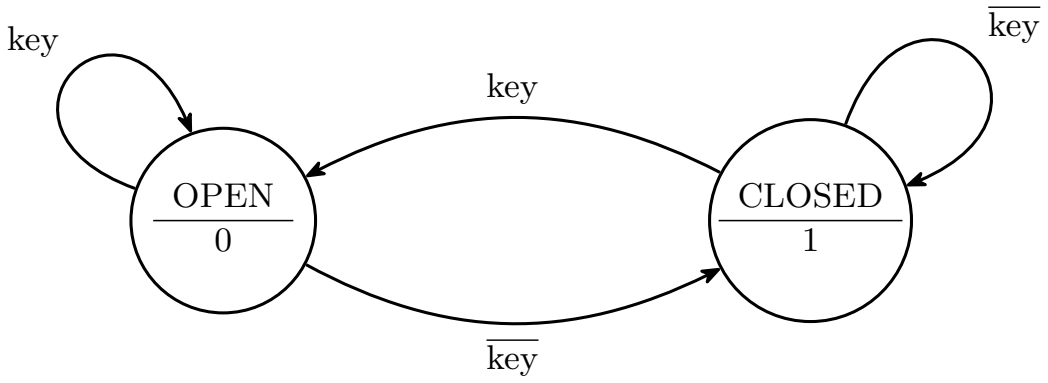


Figure 8: The two-state lock. The state name sits above the chord and the output value below it, so OPEN is the state whose locked output is 0. With one input there are two arrows out of every state, and here they account for both values of key: present the key and the lock opens or stays open, remove it and it closes or stays closed. The self-loops are needed for completeness.

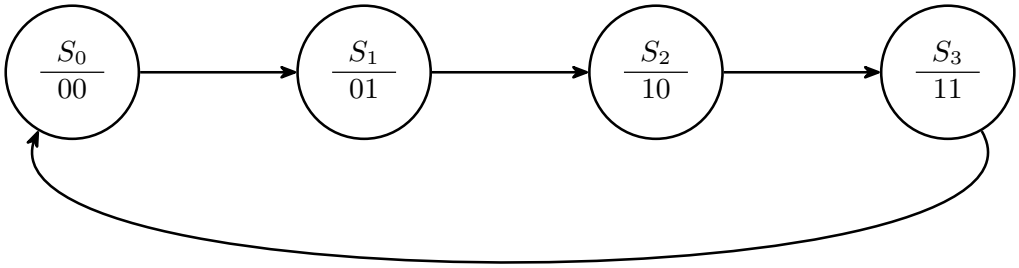


Figure 9: A 2-bit binary counter as a state machine: the four states in a left-to-right row, forward transitions between neighbours, and the wrap-around drawn as a long arc below the row. Outputs 00, 01, 10, 11, and no transition labels because there is no input to test. No state is marked as the start state. Compare the gate-level construction of the same thing in the counters note.

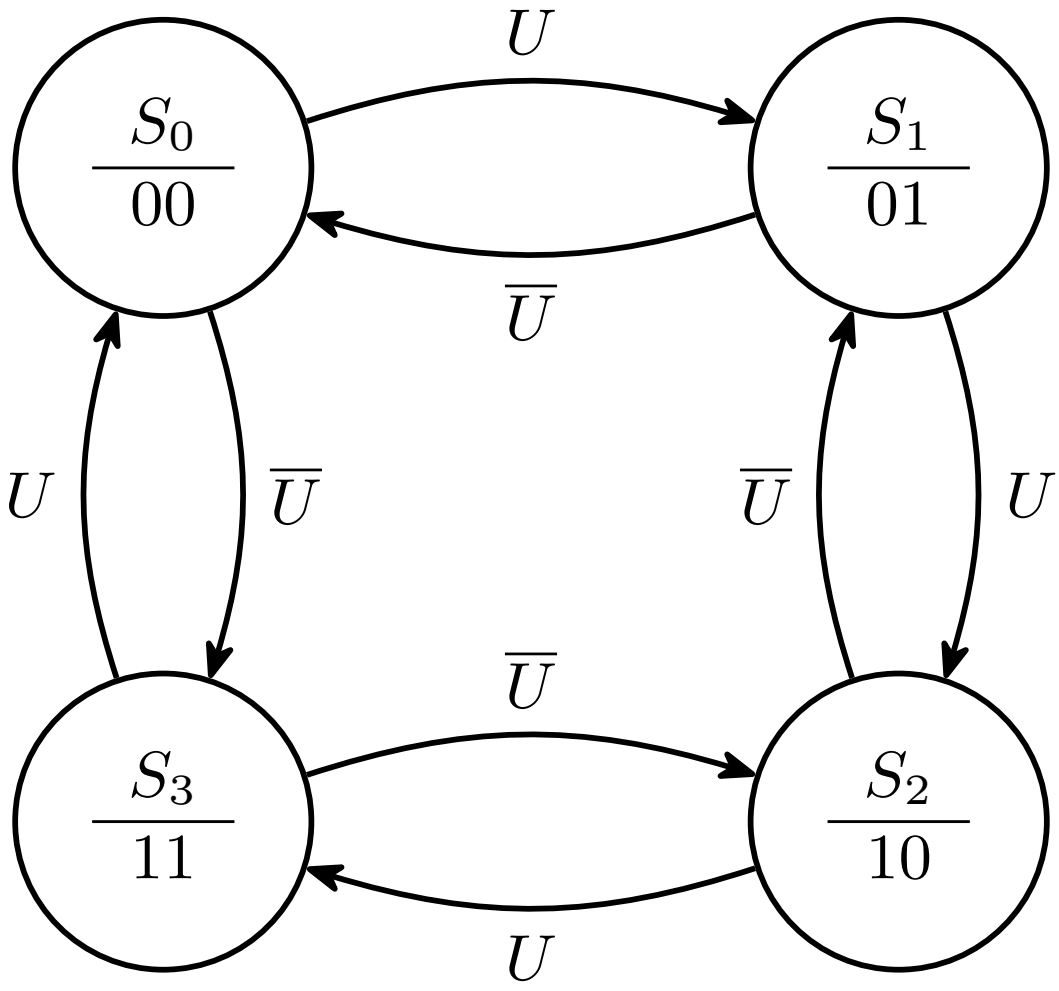


Figure 10: The up/down counter with a single input. Each state has exactly two arrows out — one for U and one for its complement — which is completeness for a one-input machine. The U arrows run clockwise through the ring and the U-bar arrows run anticlockwise. The states sit in a square ring because a straight row does not extend to a machine with arrows running both ways.

Example — binary up/down counter, two inputs

Two inputs U and D, giving four commands:

U	D	Meaning
1	0	count **up**
0	1	count **down**
1	1	**set** to 11
0	0	**reset** to 00

With N inputs there are 2^N transitions to account for out of each present state — here $2^2 = 4$ arrows out of every state. Working the four commands through each state gives:

Current state	$U\overline{D}$ (up)	$\overline{U}D$ (down)	UD (set 11)	$\overline{U}\,\overline{D}$ (reset 00)
S_0 (00)	S_1	S_3	S_3	S_0
S_1 (01)	S_2	S_0	S_3	S_0
S_2 (10)	S_3	S_1	S_3	S_0
S_3 (11)	S_0	S_2	S_3	S_0

Combinations sharing a destination merge onto one arrow, and the merged labels simplify:

Transition	Merged label	Simplifies to
$S_0 \to S_3$	$\overline{U}D + UD = D(\overline{U} + U)$	D
$S_1 \to S_0$	$\overline{U}D + \overline{U}\,\overline{D} = \overline{U}(D + \overline{D})$	$\overline{U}$
$S_2 \to S_3$	$U\overline{D} + UD = U(\overline{D} + D)$	U
$S_3 \to S_0$	$U\overline{D} + \overline{U}\,\overline{D} = \overline{D}(U + \overline{U})$	$\overline{D}$

Both steps are the complement law $X + \overline{X} = 1$ used once each — see boolean-algebra[27].

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).

The two-input counter above is the check in practice: four input combinations, four accounted for out of every state, none twice. Equivalently — the labels leaving a state must OR to 1 (no gap) and no two of them can be true at once (no ambiguity).

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

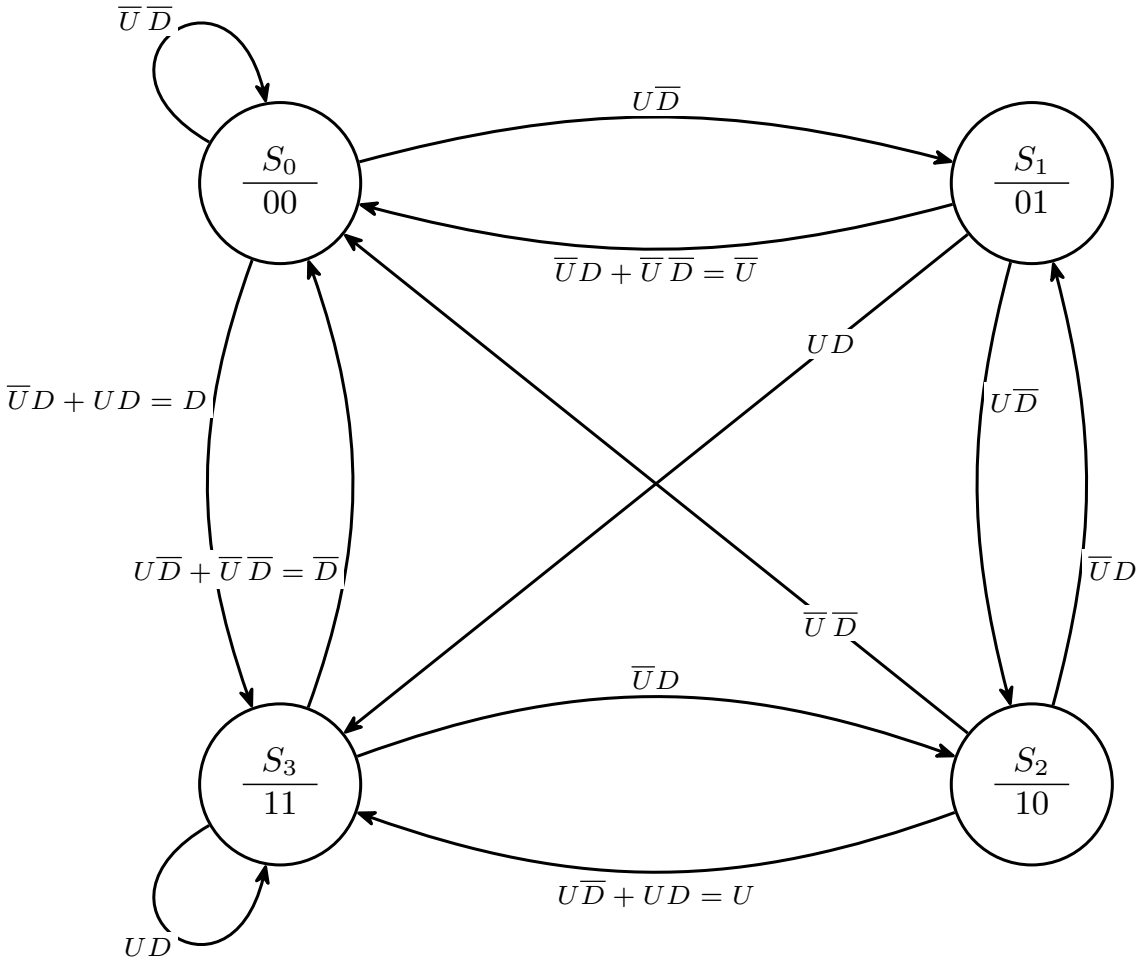


Figure 11: The up/down counter with set and reset. Every state has four input combinations to account for, drawn here as three arrows each once the combinations sharing a destination are merged. S0 keeps its reset combination as a self-loop and S3 keeps its set combination as one; the labels are shown in merged form with the simplification after the equals sign.

State tables

A state diagram can equivalently be written as a **state table**. There are two forms; both here are filled in for the up/down counter with the **single** input U (1 up, 0 down) — the machine of Figure 10.

One-dimensional — a row for **every combination of state and inputs**:

Current State	Input U	Next State	Q_1	Q_0
S_0	0	S_3	0	0
S_0	1	S_1	0	0
S_1	0	S_0	0	1
S_1	1	S_2	0	1
S_2	0	S_1	1	0
S_2	1	S_3	1	0
S_3	0	S_2	1	1
S_3	1	S_0	1	1

Two-dimensional — **one row per state**, with a next-state column for **every combination of inputs**:

Current State	Next State ($\overline{U}$)	Next State (U)	Q_1	Q_0
S_0	S_3	S_1	0	0
S_1	S_0	S_2	0	1
S_2	S_1	S_3	1	0
S_3	S_2	S_0	1	1

In the full layout the last two columns sit under a single spanning `Outputs` header, and the $\overline{U}$ / U columns under a spanning `Next State` header; the tables above repeat the header text instead. Two smaller notational points, both unresolved in csse2010-diagram-notation[28]: the output bits are written both **unsubscripted** (`Q1`, `Q0`) and subscripted (Q_1, Q_0), and the first column is headed both `Current State` and `Present State`. Nothing in the course picks a winner, so the subscripted form is used here for consistency with the rest of these notes.

The output columns repeat in the 1D form (both rows for S_0 output 00) because this is a **Moore** machine — the output is a property of the state alone, which is exactly why the 2D form can give it one column per state. In a Mealy machine the 2D form's output would have to be split by input as well.

[28] courses/csse2010/csse2010-diagram-notation.pdf

State encoding

Each state must be **encoded into flip-flop values**: choose the **number of flip-flops** and the **bit patterns** representing each state. Ideally, pick the encoding that makes the combinational logic simple — for both the output logic and the next-state logic.

Two encodings of the four-state binary counter:

State	Binary	One-hot
S_0	00	0001
S_1	01	0010
S_2	10	0100
S_3	11	1000

One-hot coding uses **one flip-flop per state**, with **only one flip-flop holding 1 at any time** — four flip-flops here instead of two. A third scheme, Gray coding, is in state-machines[29].

Worked: one-hot encoding of the binary counter with 2-bit output

Call the four flip-flops $F_3F_2F_1F_0$, one per state, with state S_k encoded by $F_k = 1$. The counter's cycle $S_0 \to S_1 \to S_2 \to S_3 \to S_0$ becomes

$$D_1 = F_0, \qquad D_2 = F_1, \qquad D_3 = F_2, \qquad D_0 = F_3$$

— i.e. **no next-state gates at all**, just four wires. That circuit is a 4-bit shift register wired end to end, otherwise known as a ring counter (see shift-registers[30] and [counters]).

The outputs are no longer free, though. Q_1 is 1 in S_2 and S_3, and Q_0 is 1 in S_1 and S_3:

$$Q_1 = F_2 + F_3, \qquad Q_0 = F_1 + F_3$$

so two OR gates. Under binary encoding the trade runs the other way: the outputs *are* the flip-flops (Q_1, Q_0 straight out, no gates) but the next-state logic needs $D_1 = Q_1 \oplus Q_0$ and $D_0 = \overline{Q_0}$. That is the point of choosing an encoding, in miniature — the encoding decides where the gates end up, and how many flip-flops you pay for.

[29] courses/csse2010/state-machines.pdf
[30] courses/csse2010/shift-registers.pdf

Sequence detector example

Design a state machine which detects the pattern **101** in an incoming bit stream and outputs a 1 whenever it detects this pattern. The output is 1 for one clock cycle after the third digit is clocked in.

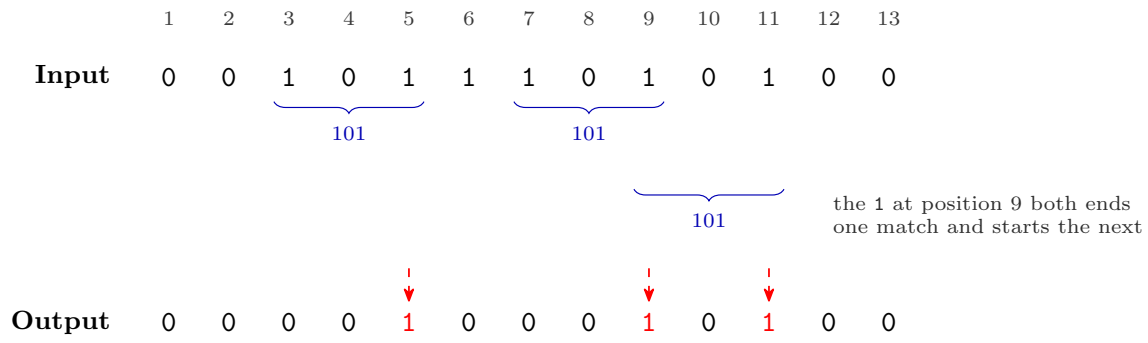


Figure 12: A worked bit stream for the 101 detector. The numbers along the top are bit positions. The braces mark the three occurrences of 101 in the input; the third is drawn on its own level because it overlaps the second. The dashed arrows show which output bit each match raises, and those output bits are red.

Three detections: the `101` finishing at position 5, the one finishing at position 9, and the **overlapping** one finishing at position 11 — the `1` at position 9 serves as both the end of one match and the start of the next. That is the whole difficulty of the problem: after a successful detection the machine must not go back to "seen nothing", because it has already seen the first `1` of a possible next match.

The machine itself is one of the two examples worked through in the Blackboard recordings.

Steps involved in FSM design

1. Defining state based on the problem definition.
2. Drawing the state diagram.
3. Completing the state table (1D or 2D).
4. State encoding and re-doing the state table with encoded states.
5. Obtaining the Boolean expressions for next state logic (i.e. the D's) and for outputs.
6. Drawing the logic diagram.
7. Simulating in Logisim to verify functionality. *the labs*
8. Building and testing the circuit in hardware.

Steps 1–6 are lecture material; **7 and 8 are what the learning labs do** — see week5-lab-state-machines[31]. Steps 5 and 6 are the same derivation used to build a counter by hand — see [counters] and circuit-schematics[32].

The examples above take the up/down counter to step 3 and the plain counter to step 4; steps 5 and 6 are exercised on the sequence detector in the recordings.

Two recordings on Blackboard work through further examples, and should be watched after this lecture.

See also

- state-machines[33] · sequential-circuits[34] · flip-flops-and-latches[35] · [counters] · shift-registers[36]
- csse2010-diagram-notation[37] — the state-diagram and state-table conventions
- week5-lab-state-machines[38] — Lab 8, which is steps 7–8 of the procedure above
- 2026-08-17-counters[39] — Lecture 7

Lab 7 Exercises

Provided task

Pre-Lab Preparation

You should complete a circuit schematic diagram as described below before your **Lab 7 session in week 4 (Thu-Fri)**. You should consult the device pinout information on Blackboard (captured in device-pinouts[40]; the drawing rules it has to satisfy are in circuit-schematics[41]). You will be testing this circuit (or a similar circuit) during the prac session.

Design Requirements

Design and draw a circuit schematic diagram for a 3-bit synchronous counter which counts through your allocated sequence (see table below).

[31] courses/csse2010/week5-lab-state-machines.pdf
[32] courses/csse2010/circuit-schematics.pdf
[33] courses/csse2010/state-machines.pdf
[34] courses/csse2010/sequential-circuits.pdf
[35] courses/csse2010/flip-flops-and-latches.pdf
[36] courses/csse2010/shift-registers.pdf
[37] courses/csse2010/csse2010-diagram-notation.pdf
[38] courses/csse2010/week5-lab-state-machines.pdf
[39] courses/csse2010/2026-08-17-counters.pdf

Hardware Specifications:

- **Clock:** You must use a push button for the clock signal.
- **Preset:** Use a single switch to allow your counter to be preset (i.e., to `111`).
- **Clear:** Use a single switch to allow your counter to be cleared (i.e., to `000`).
- **Outputs:** Your count output should be shown on three LEDs.
  - Q_2**:** Most Significant Bit (MSB)
  - Q_1**:** Middle bit
  - Q_0**:** Least Significant Bit (LSB)

  Note: Your allocated sequence has seven binary numbers; you may choose to do whatever you like with the missing number – i.e., if the counter ever has this value then we don't care what the next count value is.

It is recommended that you simulate your circuit using Logisim to ensure it counts through the expected sequence.

Count Sequences

Find the sequence corresponding to the last digit of your 8-digit UQ student number:

Table 1: Count sequence by last digit of student ID

Last digit of UQ ID	Count Sequence
0	`111 -> 011 -> 101 -> 110 -> 001 -> 000 -> 010 -> 111 -> …`
1	`111 -> 010 -> 101 -> 110 -> 100 -> 000 -> 001 -> 111 -> …`
2	`000 -> 011 -> 010 -> 110 -> 100 -> 001 -> 111 -> 000 -> …`
3	`111 -> 100 -> 101 -> 000 -> 110 -> 001 -> 011 -> 111 -> …`
4	`000 -> 101 -> 011 -> 010 -> 001 -> 110 -> 100 -> 000 -> …`
5	`000 -> 110 -> 001 -> 100 -> 111 -> 011 -> 010 -> 000 -> …`
6	`111 -> 010 -> 000 -> 110 -> 001 -> 101 -> 100 -> 111 -> …`
7	`111 -> 001 -> 110 -> 101 -> 100 -> 000 -> 010 -> 111 -> …`
8	`111 -> 000 -> 001 -> 101 -> 011 -> 110 -> 100 -> 111 -> …`
9	`000 -> 100 -> 001 -> 110 -> 101 -> 011 -> 010 -> 000 -> …`

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

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

In the lab session

The prep task above is only step one. The whole exercise is a seven-step procedure, modelled on the worked examples in lecture 7[42]:

1. Pick the counting sequence based on your student number.
2. Draw the state table — present state and next state.
3. Obtain Boolean expressions for the D's in terms of the Q's.
4. Simplify the expressions if necessary.
5. Obtain a logic diagram — and optionally simulate it.
6. Convert the logic diagram to a schematic diagram.
7. Build and test the circuit for correct operation.

Steps 1–4 are Table 2 through Equation 7 below, step 5 is Figure 13, step 6 is Figure 14, and step 7 is the bench work. The session runs as *review of preparation task → design → build and test*.

Don't forget the preset/clear inputs on the flip-flops. The design requirements need **both** a preset (to `111`) and a clear (to `000`), and only a flip-flop with both asynchronous inputs can give you that — the `74HCT74` has a SET and a CLR on each half, so it can; the `74HCT175` used in task 2 has a single shared CLR and no set at all, so it cannot. See device-pinouts[43].

Working

My student number is ——**6**, so I will be sequencing: `111 -> 010 -> 000 -> 110 -> 001 -> 101 -> 100 -> 111 -> …`

Let's map the current and next states:

Table 2: Current and next state

Current State ($Q_2Q_1Q_0$)	Next State ($D_2D_1D_0$)	Comment
0 0 0	1 1 0	000 → 110
0 0 1	1 0 1	001 → 101
0 1 0	0 0 0	010 → 000
0 1 1	X X X	unused state
1 0 0	1 1 1	100 → 111
1 0 1	1 0 0	101 → 100
1 1 0	0 0 1	110 → 001
1 1 1	0 1 0	111 → 010

[42] courses/csse2010/2026-08-17-counters.pdf
[43] courses/csse2010/device-pinouts.pdf

Deriving D_2

Rows where $D_2 = 1$:

$Q_2 Q_1 Q_0$	D_2
0 0 0	1
0 0 1	1
1 0 0	1
1 0 1	1

Each row gives one product term (1 for that row, 0 elsewhere):

$$
\begin{aligned}
D_2 &= \bar{Q}_2 \cdot \bar{Q}_1 \cdot \bar{Q}_0 + \bar{Q}_2 \cdot \bar{Q}_1 \cdot Q_0 + Q_2 \cdot \bar{Q}_1 \cdot \bar{Q}_0 + Q_2 \cdot \bar{Q}_1 \cdot Q_0 \\
&= \bar{Q}_2 \cdot \bar{Q}_1 \left(\bar{Q}_0 + Q_0\right) + Q_2 \cdot \bar{Q}_1 \left(\bar{Q}_0 + Q_0\right) && \text{(Distributive law)} \\
&= \bar{Q}_2 \cdot \bar{Q}_1 \cdot \mathbf{t} + Q_2 \cdot \bar{Q}_1 \cdot \mathbf{t} && \text{(Negation law)} \\
&= \bar{Q}_2 \cdot \bar{Q}_1 + Q_2 \cdot \bar{Q}_1 && \text{(Identity law)} \\
&= \bar{Q}_1 \left(\bar{Q}_2 + Q_2\right) && \text{(Distributive law)} \\
&= \bar{Q}_1 \cdot \mathbf{t} && \text{(Negation law)} \\
&= \bar{Q}_1 && \text{(Identity law)}
\end{aligned} \tag{1}
$$

D_2 is just $\bar{Q}_1$, so it needs no gate at all. FF1's $\overline{Q}$ output wires straight to FF2's D input.

Deriving D_1

Sum of products where $D_1 = 1$ (`000`, `100`, `111`):

$$
\begin{aligned}
D_1 &= \bar{Q}_2 \cdot \bar{Q}_1 \cdot \bar{Q}_0 + Q_2 \cdot \bar{Q}_1 \cdot \bar{Q}_0 + Q_2 \cdot Q_1 \cdot Q_0 \\
&= \bar{Q}_1 \cdot \bar{Q}_0 \left(\bar{Q}_2 + Q_2\right) + Q_2 \cdot Q_1 \cdot Q_0 && \text{(Distributive law)} \\
&= \bar{Q}_1 \cdot \bar{Q}_0 \cdot \mathbf{t} + Q_2 \cdot Q_1 \cdot Q_0 && \text{(Negation law)} \\
&= \bar{Q}_1 \cdot \bar{Q}_0 + Q_2 \cdot Q_1 \cdot Q_0 && \text{(Identity law)}
\end{aligned} \tag{2}
$$

Deriving D_0

Sum of products where $D_0 = 1$ (`001`, `100`, `110`), plus the unused state `011` — its next state is a don't care, so I am free to take $D_0 = 1$ there, which lets Q_1 cancel:

$$\begin{aligned} D_0 &= \bar{Q}_2 \cdot \bar{Q}_1 \cdot Q_0 + \bar{Q}_2 \cdot Q_1 \cdot Q_0 + Q_2 \cdot \bar{Q}_1 \cdot \bar{Q}_0 + Q_2 \cdot Q_1 \cdot \bar{Q}_0 \\ &= \bar{Q}_2 \cdot Q_0 \left(\bar{Q}_1 + Q_1\right) + Q_2 \cdot \bar{Q}_0 \left(\bar{Q}_1 + Q_1\right) && \text{(Distributive law)} \\ &= \bar{Q}_2 \cdot Q_0 \cdot \mathbf{t} + Q_2 \cdot \bar{Q}_0 \cdot \mathbf{t} && \text{(Negation law)} \\ &= \bar{Q}_2 \cdot Q_0 + Q_2 \cdot \bar{Q}_0 && \text{(Identity law)} \\ &= Q_2 \oplus Q_0 && \text{(definition of XOR)} \end{aligned} \tag{3}$$

$\bar{Q}_2 \cdot Q_0 + Q_2 \cdot \bar{Q}_0$ is exclusive-or, so D_0 is a single XOR gate.

Checking Robustness

`011` was left unused in Table 2, so its next state was treated as a don't care when deriving Equation 1, Equation 2 and Equation 3. The circuit will still compute *something* for this state, so I need to check it doesn't get stuck there. Substituting $Q_2Q_1Q_0 = 011$ into each equation:

$$\begin{aligned} D_2 &= \bar{Q}_1 \\ &= \bar{1} \\ &= 0 \end{aligned} \tag{4}$$

$$\begin{aligned} D_1 &= \bar{Q}_1 \cdot \bar{Q}_0 + Q_2 \cdot Q_1 \cdot Q_0 \\ &= \bar{1} \cdot \bar{1} + 0 \cdot 1 \cdot 1 \\ &= 0 \cdot 0 + 0 \\ &= 0 \end{aligned} \tag{5}$$

$$\begin{aligned} D_0 &= Q_2 \oplus Q_0 \\ &= 0 \oplus 1 \\ &= 1 \end{aligned} \tag{6}$$

So $D_2D_1D_0 = 001$, meaning `011 -> 001`. This is not `011` itself, and `001` is already part of my sequence, so the counter does not lock up in the unused state: it self-corrects back into the main count sequence within one clock cycle.

Taking the don't care as a 1 in D_1 as well would have simplified it to $\overline{Q_1 \oplus Q_0}$, but then `011 -> 011` and the counter would hang, so I only used it for D_0.

Conclusion

As per Equation 1, Equation 2 and Equation 3, the excitation equations for the counter are:

$$\begin{aligned} D_2 &= \bar{Q}_1 \\ D_1 &= \bar{Q}_1 \cdot \bar{Q}_0 + Q_2 \cdot Q_1 \cdot Q_0 \\ D_0 &= Q_2 \oplus Q_0 \end{aligned} \tag{7}$$

and the robustness check in Equation 4, Equation 5 and Equation 6 confirms that the one unused state, `011`, transitions to `001` rather than to itself. The counter is therefore self-correcting, and these three equations fully and safely implement my allocated count sequence.

That comes to five 2-input gates, which fit into three gate chips. No inverters are needed anywhere: the `74HCT74` gives $\overline{Q}$ alongside Q, so $\overline{Q_1}$ and $\overline{Q_0}$ are free.

Circuit Schematic

Figure 13 is a logic diagram. Figure 14 is the same counter drawn as a circuit schematic (see circuit-schematics[44]), with device IDs, chip types and pin numbers from device-pinouts[45].

It takes five chips: `U1` and `U2` are `74HCT74` dual flip-flops, `U3` a `74HCT08` (the 3-input product is `U3:B` and `U3:C` cascaded, since the kit has no 3-input gate), `U4:A` a `74HCT32` and `U5:A` a `74HCT86`. On the IO board, CLOCK is button `B0`, PRESET and CLEAR are switches `S0` and `S1`, and $Q_2Q_1Q_0$ drive LEDs `L2`, `L1` and `L0`.

Peer review of the schematic

Before building, everyone checks a neighbour's schematic against a fixed list. It is worth running over Figure 14 as a self-check, because it is also the marking list:

- **naming** of inputs and outputs;
- **identification of chips** (`U1`, `U2`, …) and of the gate within a chip where it matters (`:A`, `:B`, …);
- **identification of chip types** (`74HCT00` etc.);
- **numbering of pins**, including the IO board pins;
- **power supply connections**;

[44] courses/csse2010/circuit-schematics.pdf
[45] courses/csse2010/device-pinouts.pdf
[46] courses/csse2010/circuit-schematics.pdf

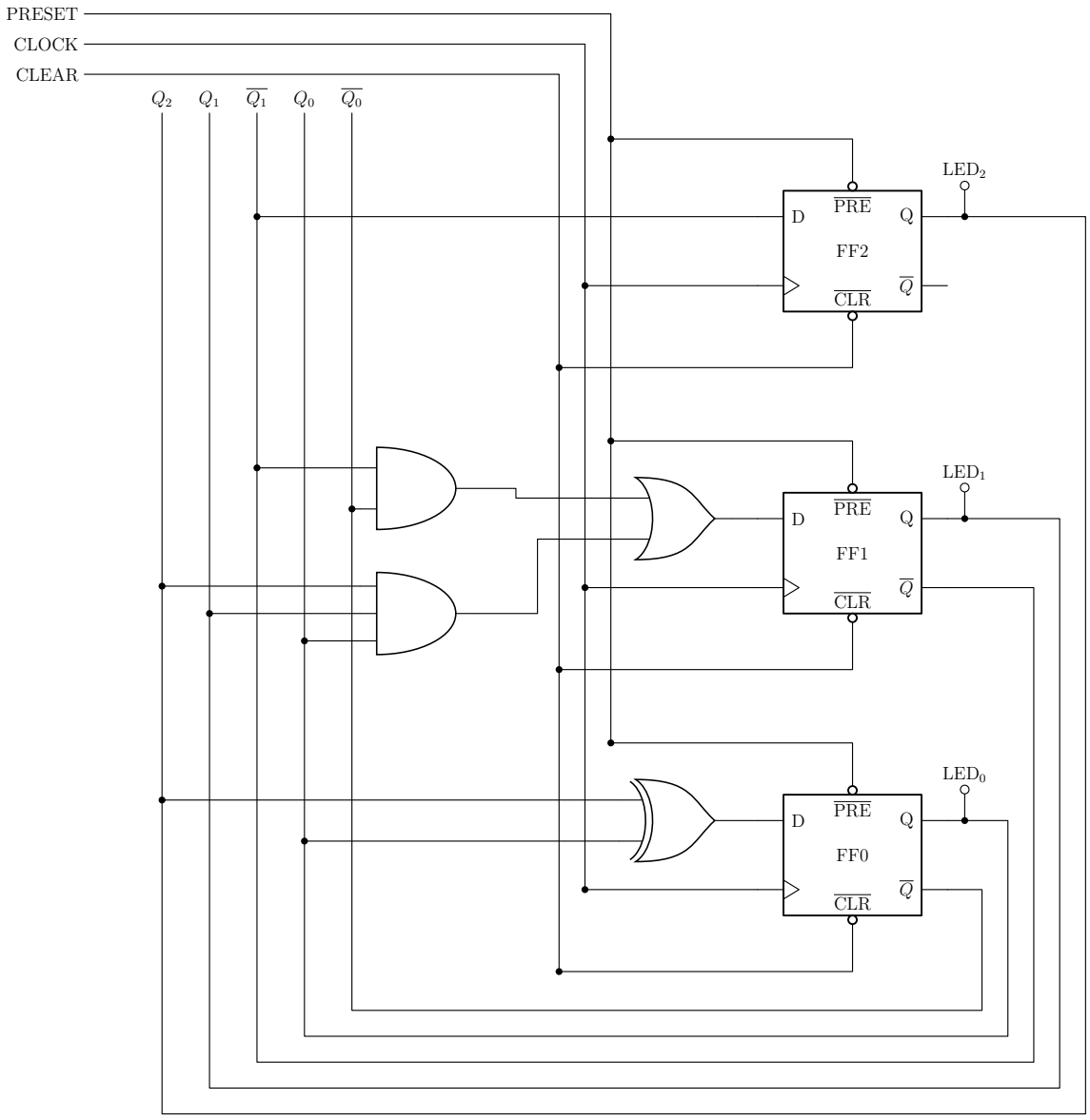


Figure 13: The complete counter: the gates of Equation 7 on the left, the three flip-flops on the right, and $Q/\overline{Q}$ fed back to the input rails. D_2 has no gate: it is FF1's $\overline{Q}$ wired straight across. CLOCK, PRESET and CLEAR come from the push button and the two switches; each Q drives one of the three LEDs.

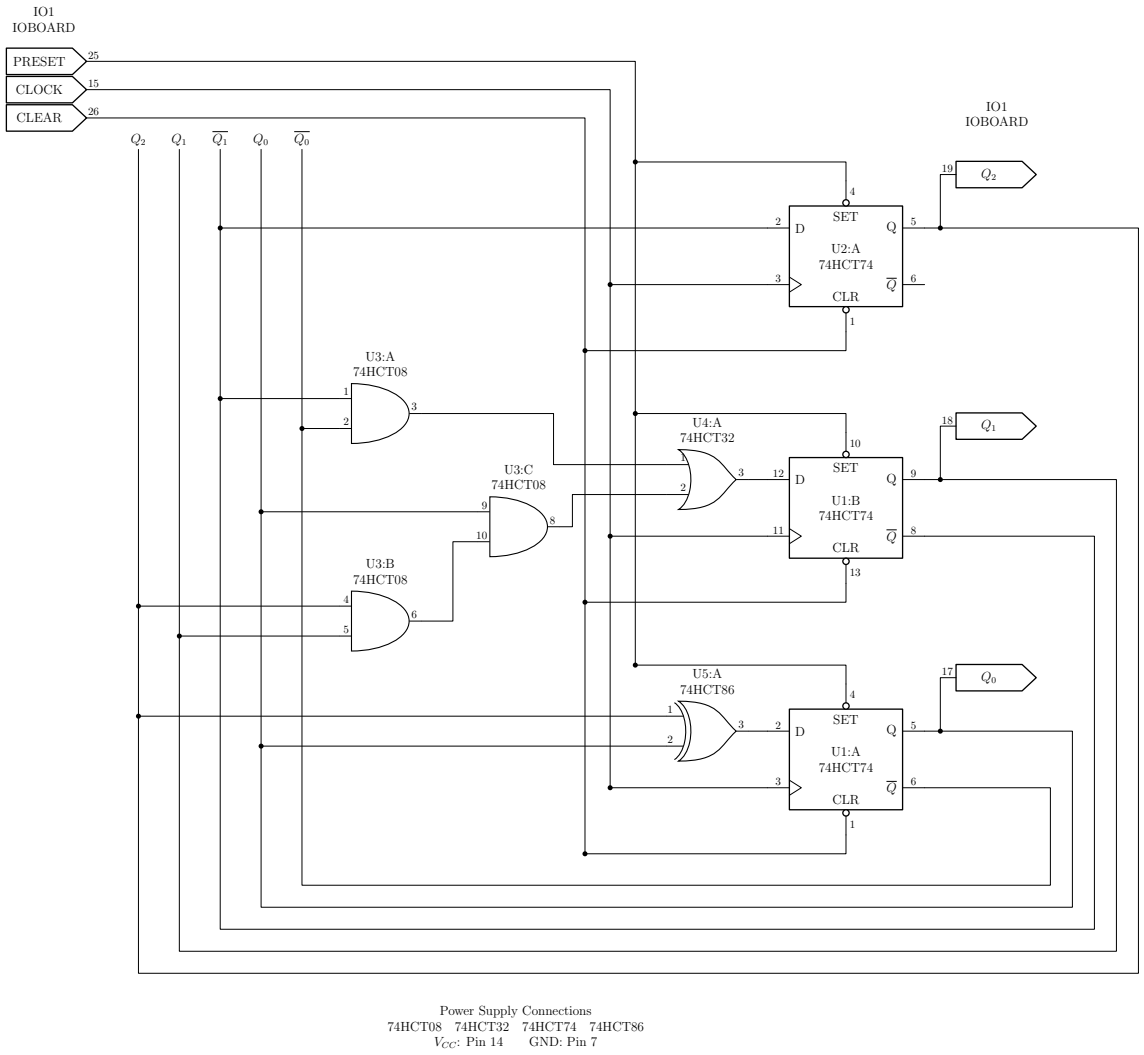


Figure 14: The counter as a circuit schematic, ready to build: device IDs and chip types on every gate, pin numbers on every connection, IO board symbols for the inputs and outputs, and the power connections given once for all four chip types. Inputs and outputs are the same right-pointing pentagon, the input's wire leaving the point and the output's arriving at the flat back edge, per circuit-schematics[46]. A dot marks a joined connection; crossing wires are not connected.

- **circuit functionality — will it do what was asked?**

The schematic guide and device pinouts on Blackboard have the detail (circuit-schematics[47] and device-pinouts[48] here). Ask a tutor if that is not enough. My schematic uses five 74-series chips.

Task 2 — a 4-bit up/down counter built from an adder

The second task drops the derive-the-equations method entirely. Instead of working out $D_3 \dots D_0$ by hand, put a **4-bit adder** (`74HCT283`) in front of a **4-bit register** (`74HCT175`) and let the adder compute the next count:

- the current count — the flip-flop outputs $Q_3Q_2Q_1Q_0$ — goes back to one adder input, A;
- the adder's sum outputs $S_3 \dots S_0$ drive the flip-flop inputs $D_3 \dots D_0$;
- an external input U picks the direction:
  - $U = 0$: $0000 \to 0001 \to \dots \to 1111 \to 0000$ (**up**);
  - $U = 1$: $1111 \to 1110 \to \dots \to 0000 \to 1111$ (**down**).

That leaves the other adder input B and the carry-in C_{in} to work out for each mode:

Table 3: The table to fill in

Mode	Adder B input	C_{in}
Up-count ($U = 0$)		
Down-count ($U = 1$)		

Working out B and Cin

> **Note**
>
> Hint: -1 in four-bit two's complement is `1111`, and that is exactly the thing task 2 needs.

The adder computes $A + B + C_{in}$, and A is already the current count. So the question is only what to add.

Counting up means the next count is one more than this one, so $B + C_{in}$ has to come to 1. The cheapest way is $B = 0000$ with $C_{in} = 1$ — the carry-in is a free `+1`, which is why it is the input being asked about at all.

[47] courses/csse2010/circuit-schematics.pdf
[48] courses/csse2010/device-pinouts.pdf

Counting down means the next count is one *less*, i.e. $Q + (-1)$. In four-bit two's complement -1 is `1111` (see binary-number-representations[49]), so $B = 1111$ with $C_{in} = 0$.

Table 4: The completed table

Mode	Adder B input	C_{in}
Up-count ($U = 0$)	`0000`	`1`
Down-count ($U = 1$)	`1111`	`0`

Read Table 4 down the columns and the combinational logic falls out with no algebra at all:

$$B_3 = B_2 = B_1 = B_0 = U \qquad\qquad C_{in} = \overline{U}$$

Every bit of B is just U wired straight to it, and C_{in} is U through a single inverter. One gate for the whole direction control.

The wrap-around is free, and worth seeing why: the adder's carry-out C_4 goes nowhere, so the arithmetic is mod 16. Counting up from `1111` gives $1111 + 0000 + 1 = 1\,0000$, and the discarded carry leaves `0000`. Counting down from `0000` gives $0000 + 1111 + 0 = 1111$ with no carry generated at all.

Building it

The steps for task 2:

1. Complete the given logic diagram, connecting U to the adder's B input and C_{in} with appropriate combinational logic — that is Figure 15.
2. Simulate the logic diagram in Logisim to check it behaves.
3. Convert the logic diagram to a schematic diagram (the same treatment Figure 13 got in Figure 14).
4. Build and test it: **clock on a push button, U on a slide switch, the four counter outputs on four LEDs** — see io-board[51] for which pins those are.

Two things to watch on the hardware. The `74HCT175` shares one clock (pin 9) and one clear (pin 1) across all four flip-flops, so unlike the `74HCT74` pairs in Figure 14 there is nothing to fan out — but there is also no preset, so this counter can only be cleared, not loaded. And the register has no clock-enable: every push of the button counts, in whichever direction U happens to be selecting at that instant.

[49] courses/csse2010/binary-number-representations.pdf
[50] courses/csse2010/device-pinouts.pdf
[51] courses/csse2010/io-board.pdf

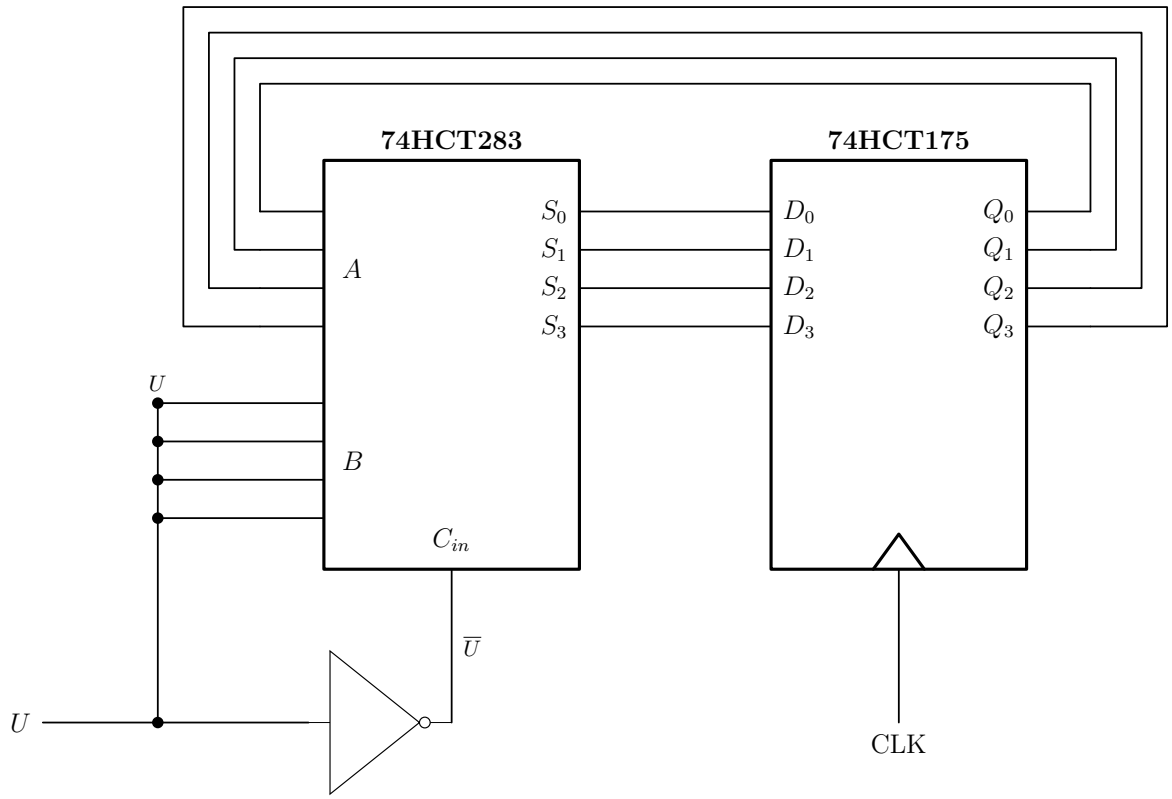


Figure 15: Task 2's logic diagram with the two unknowns filled in. $Q_3 \ldots Q_0$ feed back to the adder's A input, the sum drives the register's D inputs, and the direction input U does two jobs: tied to all four bits of B it supplies `0000` or `1111`, and inverted into C_{in} it supplies the `+1` that up-counting needs. The four nested feedback loops are drawn so no two cross. The task gives the two chips, the clock, the A feedback and the S-to-D connections; the B rail, the inverter and the C_{in} connection complete it, per the derivation above. The chips are plain rectangles rather than the notched arithmetic-block outline device-pinouts[50] gives the `74HCT283`, and the `74HCT175`'s clock wedge sits on the bottom edge here rather than on the left edge as in the pinout sheet's symbol. Either way it is one clock serving all four flip-flops.

Sequential Circuits 1 — Shift Registers

Learning Lab 6 — the four-mode shift-register design exercise, then the combination lock from the preparation task. Pairs with 2026-08-13-shift-registers[52], which carries the full preparation-task spec, and with shift-registers[53] for the mux-per-flip-flop construction the exercise turns on.

Lab plan

- Sequential circuits — the shift register exercise
- The **combination lock** preparation task — discuss with staff and others at your table
- The combination-lock example is set up at the end of Week 3, Lecture 6 (see the lecture note[54]); read it first
- Modifying the combination lock
- Make sure you can draw a circuit schematic and simulate in Logisim. **If in doubt, ask.**

Ed Lessons — C is starting

The Ed discussion board also hosts **Ed Lessons**, teaching the basics of C. The content is no more complex than CSSE1001, and **C programming for the AVR starts in week eight**. Staggered release:

Date	Release
14 Aug	Basics *(out now)*
28 Aug	Next Steps #1
11 Sep	Next Steps #2
25 Sep	Extension Tasks
16 Oct	Floating Point Workshop

Exercise — the four-mode shift register

The exercise is a **half-drawn circuit** to finish. What is already wired matters as much as what is missing.

Complete the design so that it implements this behaviour:

[52] courses/csse2010/2026-08-13-shift-registers.pdf
[53] courses/csse2010/shift-registers.pdf
[54] courses/csse2010/2026-08-13-shift-registers.pdf

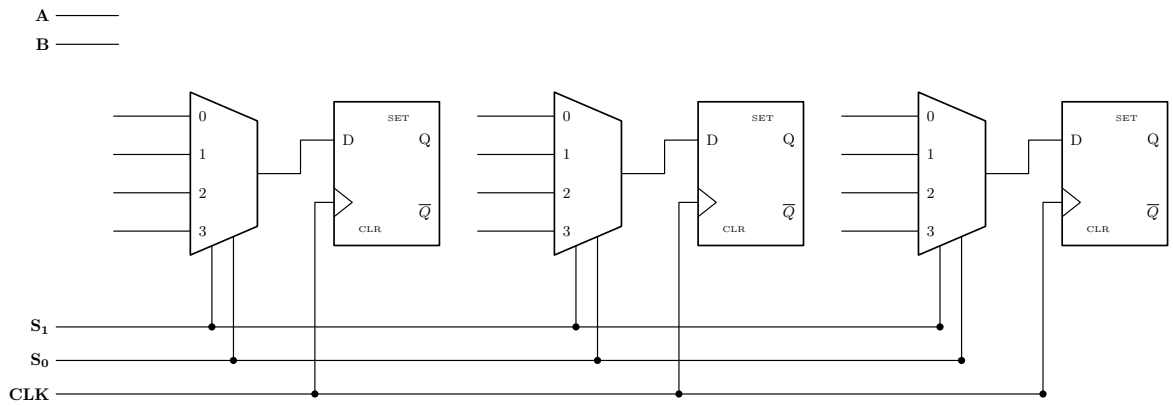


Figure 16: The half-drawn circuit to complete. Three stages, not four: each is a 4-to-1 multiplexer (trapezoid, wide edge left, data inputs numbered 0-3 top to bottom, select inputs entering the bottom sloping edge) feeding one edge-triggered D flip-flop. Already wired for you are the mux outputs into the D inputs and the three shared rails - S1 and S0 into every mux's select, CLK into every flip-flop. Left bare are the twelve mux data inputs and the A and B stubs at the top left: filling those in is the whole exercise. Junction dots mark connections; lines that cross without a dot are not connected.

S_1S_0	Action when the circuit is clocked
`00`	Bits shifted to the **right**; the value on A is stored in the leftmost flip-flop
`01`	**All bit values are toggled**
`10`	If B is 1, the value from A is stored in **all** flip-flops; if B is 0, all flip-flops **reset** (i.e. 0 is stored in all flip-flops)
`11`	All bits are **rotated to the left**. (Bit in leftmost flip-flop is moved to rightmost flip-flop)

Hint: construct the truth table for $f(A, B)$.

What the hint is for

Three of the four modes pick a *source* for each D input — some other flip-flop's output, or that flip-flop's own complement. Mode `10` is the only one that has to *compute* something from the primary inputs, and the hint says to nail it down as a truth table before wiring anything:

A	B	What the row asks for	f
0	0	$B = 0$, so reset — store 0	0
0	1	$B = 1$, so store A — store 0	0
1	0	$B = 0$, so reset — store 0	0
1	1	$B = 1$, so store A — store 1	**1**

$$\boxed{f(A, B) = A \cdot B}$$

One AND gate, its output fanned out to input 2 of all three multiplexers. "Load A" and "reset" are not two behaviours needing two circuits — they are the same wire, because B gates A to zero.

Where each mux input comes from

Name the flip-flops $Q_2Q_1Q_0$, leftmost first, matching the bit ordering used in shift-registers[55]. Input n of every mux is the one selected by $S_1S_0 = n$, so the mode table becomes a wiring table:

Mux input	Mode	Leftmost (D_2)	Middle (D_1)	Rightmost (D_0)
0	`00` shift right	A	Q_2	Q_1
1	`01` toggle	$\overline{Q_2}$	$\overline{Q_1}$	$\overline{Q_0}$
2	`10` load / reset	$f = A \cdot B$	f	f
3	`11` rotate left	Q_1	Q_0	Q_2

Three things fall out of that table, and they are what the exercise is really testing:

- **The toggle mode needs no gates at all.** Every D flip-flop already publishes $\overline{Q}$, so input 1 of each mux is a wire back from the stage's own complementary output. Feeding $\overline{Q}$ into D is the standard way to make a flip-flop toggle — it is the same $D = \overline{Q}$ wiring [counters] gives for a 1-bit counter.
- **The load/reset mode needs exactly one gate**, shared, because f does not depend on the stage.
- **Rotate left is shift right's column read backwards, plus a wrap.** Input 0 always comes from the neighbour on the left, input 3 always from the neighbour on the right; the only asymmetry is at the ends — the leftmost stage takes A for input 0 (there is no neighbour, and the bit shifted out of Q_0 is simply lost), and the rightmost stage takes Q_2 for input 3.

[55] courses/csse2010/shift-registers.pdf
[56] courses/csse2010/shift-registers.pdf

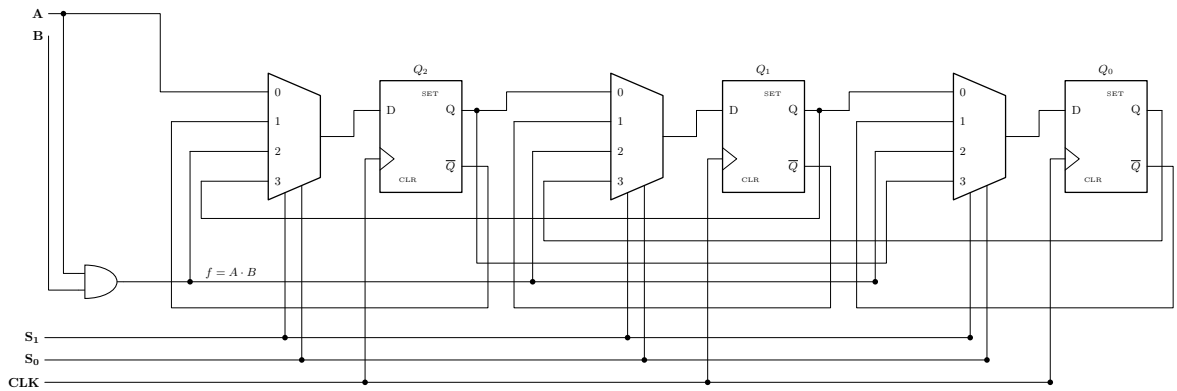


Figure 17: The completed design. Each mux input takes the source its select code names: 0 shifts right (from the left-hand neighbour's Q, or from A at the left end), 1 toggles (the stage's own Qbar, which the flip-flop already provides, so no inverter is needed anywhere), 2 loads f = A.B (one AND gate shared by all three stages), 3 rotates left (from the right-hand neighbour's Q, with Q2 wrapping round to the rightmost stage). The flip-flops are named Q2 Q1 Q0 left to right, matching the bit ordering used in shift-registers[56]. Lines crossing without a junction dot are not connected - that rule is doing a lot of work in the band of return wires below the row.

A trace

A check of the wiring above. Start at $Q_2Q_1Q_0 = 101$ and hold $A = 1$, $B = 1$ except where noted:

S_1S_0	Mode	Before	After
00	shift right, $A = 1$ in at the left	101	**110**
01	toggle	110	**001**
11	rotate left	001	**010**
10	$B = 1$: load $A = 1$ everywhere	010	**111**
10	$B = 0$: reset	111	**000**

The rotate row is the one worth checking twice: $Q_2 \leftarrow Q_1 = 0$, $Q_1 \leftarrow Q_0 = 1$, and the bit that was in the leftmost flip-flop, $Q_2 = 0$, wraps into the rightmost.

Discussing your diagrams

The lab runs a schematic review. The checklist:

Logic diagram — circuit functionality: will it do what was asked?

Circuit schematic —

- naming of inputs and outputs
- identification of chips (`U1`, `U2`, …) and gates within chips where applicable (`:A`, `:B`, …)
- identification of types of chips (`74HCT00` etc.)
- **numbering of pins**
- **power supply connections**

Check the schematic guide and device pinouts on Blackboard for more detail, and ask a staff member if necessary. Full rules in circuit-schematics[57]; pinouts in device-pinouts[58].

Combination lock tasks

The preparation task itself — a two-stage 4-bit shift register, four switches, clock and reset buttons, locked/unlocked LEDs, and a code derived from your student number — is spelled out in 2026-08-13-shift-registers[59], which also carries the block diagram and the ordering trap (you enter the left code digit first, so it is the *older* one and ends up in the second stage).

Task 1 — test your combination lock design.

Task 2 — use fewer flip-flops. On paper, design something that detects the sequence `AC` in hex: switches set to A (`1010`), clock push-button pressed; switches set to C (`1100`), clock push-button pressed.

> Can you do it with **3** flip-flops? Can you do it with **2**? *(Hint: can you move the registers towards the output side?)*

Working — Task 2

The basic lock stores both digits and compares afterwards: two 4-bit stages, **eight flip-flops**, feeding a combinational code check. But the digits themselves are never needed downstream — only *whether each was the expected one*. So compare first, and register the verdict:

$P = W_3\overline{W_2}W_1\overline{W_0}$ (switches show A $= 1010_2$) $\qquad C = W_3W_2\overline{W_1}\,\overline{W_0}$ (switches show C $= 1100_2$)

- **Flip-flop 1**: $D_1 = P$. After a clock, $Q_1 = 1$ means *the digit just entered was A*.
- **Flip-flop 0**: $D_0 = Q_1 \cdot C$. After the next clock, $Q_0 = 1$ means *the last two digits entered were A then C*.
- unlocked LED $= Q_0$; locked LED $= \overline{Q_0}$, taken straight off the flip-flop.

[57] courses/csse2010/circuit-schematics.pdf
[58] courses/csse2010/device-pinouts.pdf
[59] courses/csse2010/2026-08-13-shift-registers.pdf

Two flip-flops, down from eight. The combinational cost barely moves — the original design also has to compare two 4-bit values against the code, so P and C are the same two equality checks moved to the other side of the flip-flops, plus one AND. The whole saving is in storage.

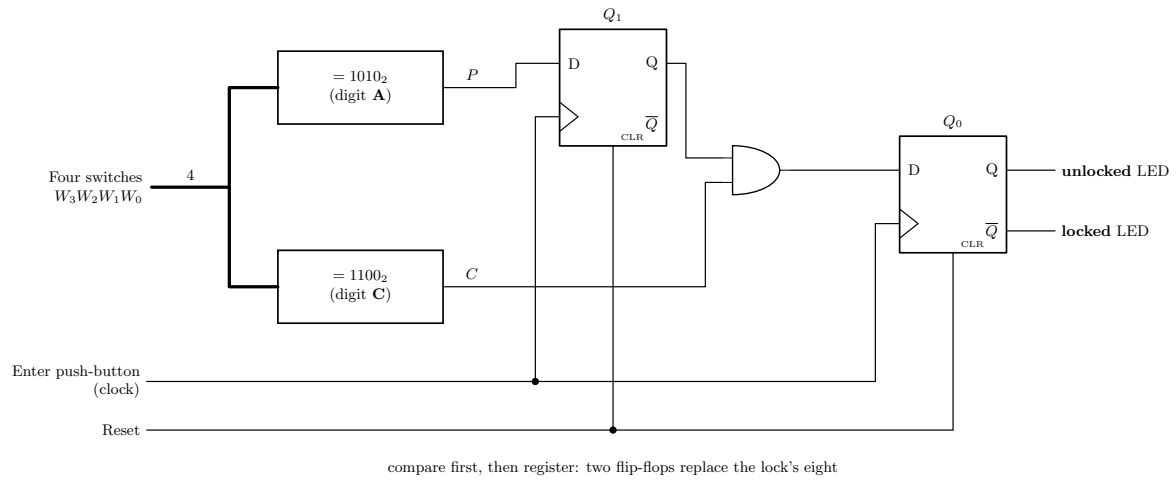


Figure 18: Task 2's two-flip-flop AC detector. This is the hint 'move the registers towards the output side' drawn out: the four switches are compared against the two expected digits combinationally, and only the one-bit verdicts are clocked, so the eight flip-flops of the basic lock (compare after storing, see the block diagram in 2026-08-13-shift-registers[60]) collapse to two. Q1 remembers 'the digit just entered was A'; Q0 is the unlock verdict, and it is a registered output, so the LEDs change only when the enter button is pressed. The locked LED needs no inverter - the flip-flop's Qbar is already the complement. Lines crossing without a junction dot are not connected. The comparators are drawn as plain blocks, and the four-switch bus is drawn heavy with its width lettered beside it, which is neither of the two ways the course marks bus width (a slash across the line with the width beside it, or the width lettered next to a plain line).

Why not one? With a single flip-flop holding "the last digit was A" and the LED driven combinationally by $Q_1 \cdot C$, the lock would open the instant the switches were set to C — before the enter button was pressed. That is a Mealy output, and it is not what the original lock does. Registering the verdict is what keeps the LEDs changing only on a clock edge. Two is also the floor for a Moore machine here: the circuit must distinguish three situations — nothing useful yet, A just entered, AC just entered — and three states need two flip-flops. See state-machines[61].

Three flip-flops. Reading the question as a ladder, three flip-flops is what a **one-hot** encoding of those same three states costs — one flip-flop per state — and going to a binary encoding is

[60] courses/csse2010/2026-08-13-shift-registers.pdf
[61] courses/csse2010/state-machines.pdf

what buys the drop to two. The compare-then-register construction lands on two directly, so three is a waypoint rather than a separate design.

Simulate and test.

Task 3 — challenge. Allow the unlock combination itself to be **set and stored in flip-flops** using switches and buttons, rather than hard-wired. Simulate and test.

The expected trajectory: for each of tasks 1 and 2, *logic/schematic diagrams completed* → *partially working circuit* → *fully working circuit*, with task 3 only at the diagram stage.

Next

Lab 7 has a **preparation task**, and it is to be done **before** you arrive for the practical session — see week4-lab-lab-7-exercises[62].

See also

- shift-registers[63] · combinational-logic-blocks[64] — the 4-to-1 mux and the mux-per-flip-flop stage
- 2026-08-13-shift-registers[65] — Lecture 6, and the full preparation-task specification
- circuit-schematics[66] · device-pinouts[67] — the schematic-review checklist above, in full
- csse2010-diagram-notation[68] — the mux and flip-flop symbol conventions these figures follow

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`.

[62] courses/csse2010/week4-lab-lab-7-exercises.pdf
[63] courses/csse2010/shift-registers.pdf
[64] courses/csse2010/combinational-logic-blocks.pdf
[65] courses/csse2010/2026-08-13-shift-registers.pdf
[66] courses/csse2010/circuit-schematics.pdf
[67] courses/csse2010/device-pinouts.pdf
[68] courses/csse2010/csse2010-diagram-notation.pdf

- 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.

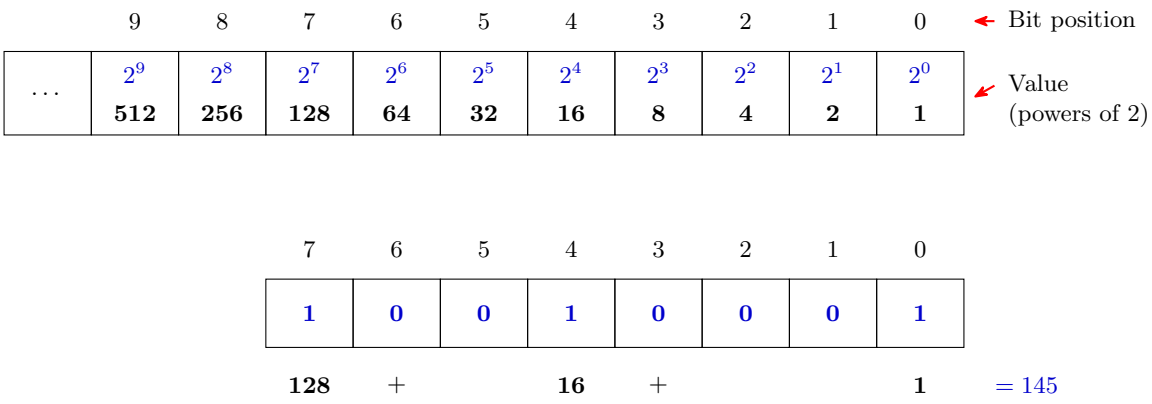


Figure 19: Place values for an unsigned binary number, and a worked conversion. Position i is worth 2^i, so reading a binary number is just adding the place values of the positions holding a 1: $10010001_2 = 128 + 16 + 1 = 145$.

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**.

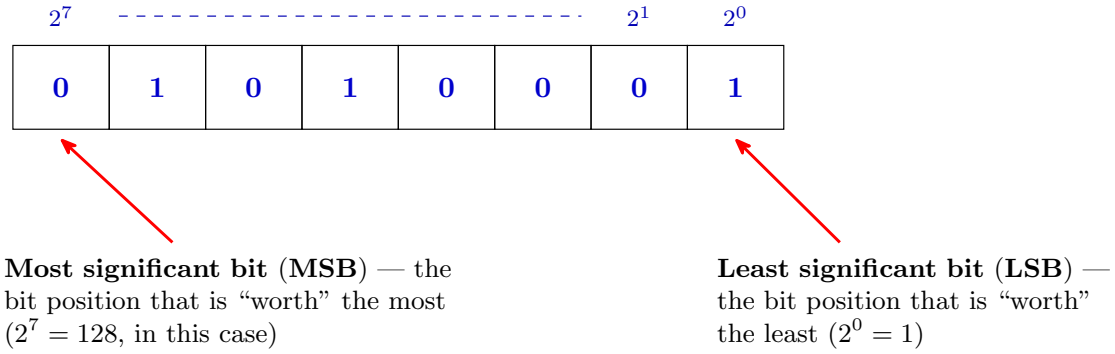


Figure 20: The two ends of a word get names, because almost every later convention (shifting, sign bits, bit numbering in AVR registers) is stated in terms of them. For an n-bit unsigned word the MSB is worth 2^{n-1} — here $2^7 = 128$ — and the LSB is always worth $2^0 = 1$.

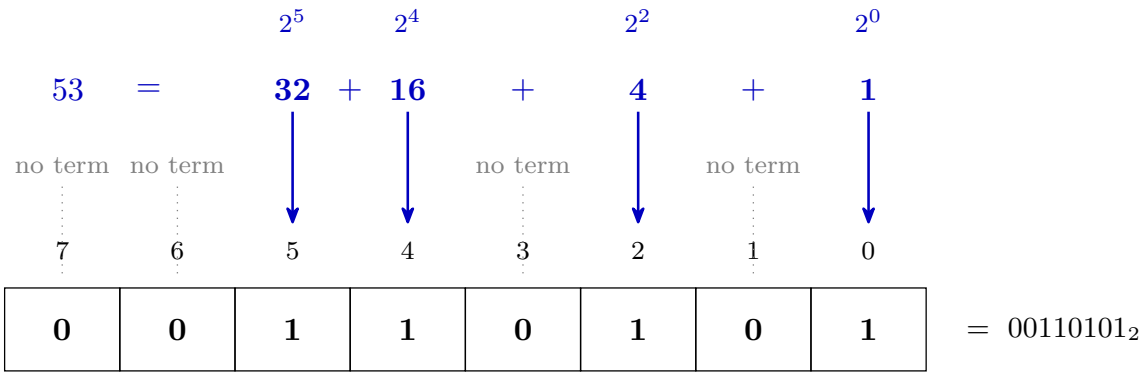


Figure 21: Method 1 on 53. Each power of two in the decomposition names one bit position — 2^5 is position 5, 2^4 is position 4, and so on — and every position with no term in the sum gets a 0. Reading the row off left to right gives 00110101_2.

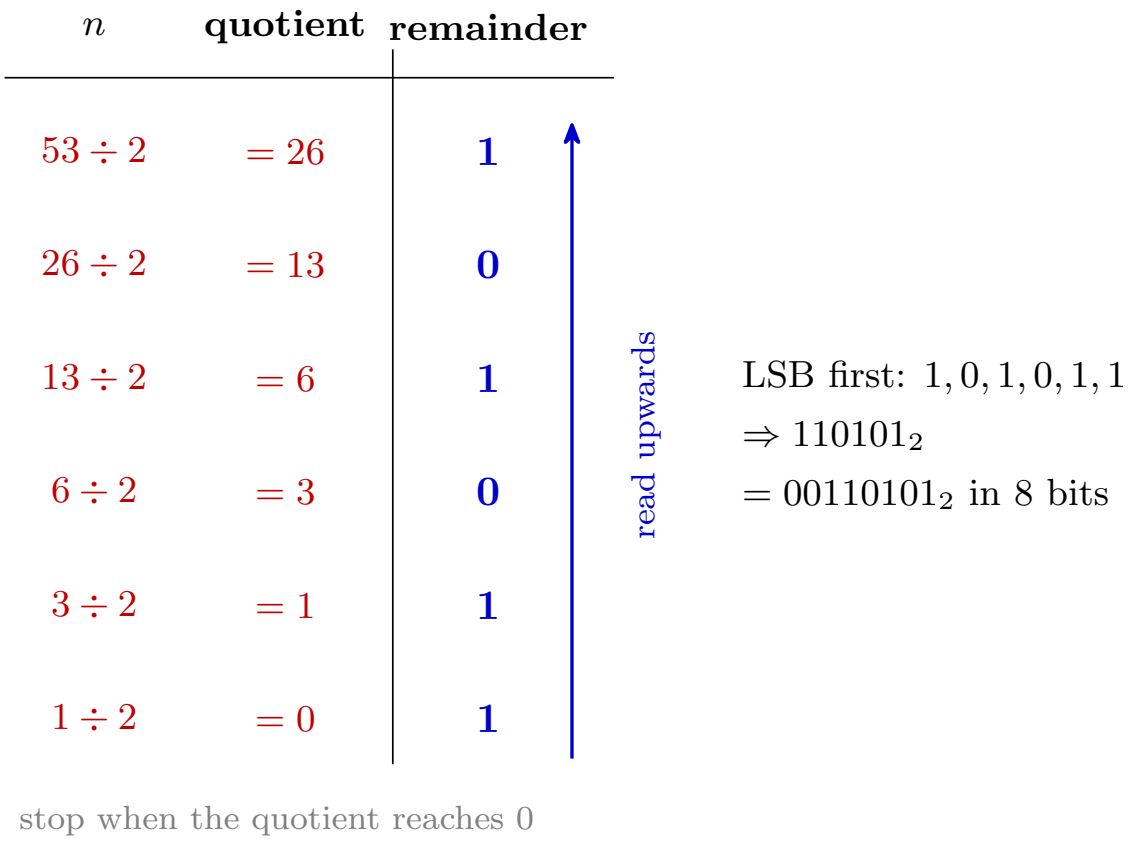


Figure 22: Method 2 on 53. The remainders come out LSB-first, which is why the answer has to be read back up the column: 110101_2, or 00110101_2 padded to a byte. The same procedure with a divisor of 16 or 8 converts to hexadecimal or octal instead.

Number range (unsigned)

- Smallest representable value: all 0's → 0.
- Largest representable value: all 1's → 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.

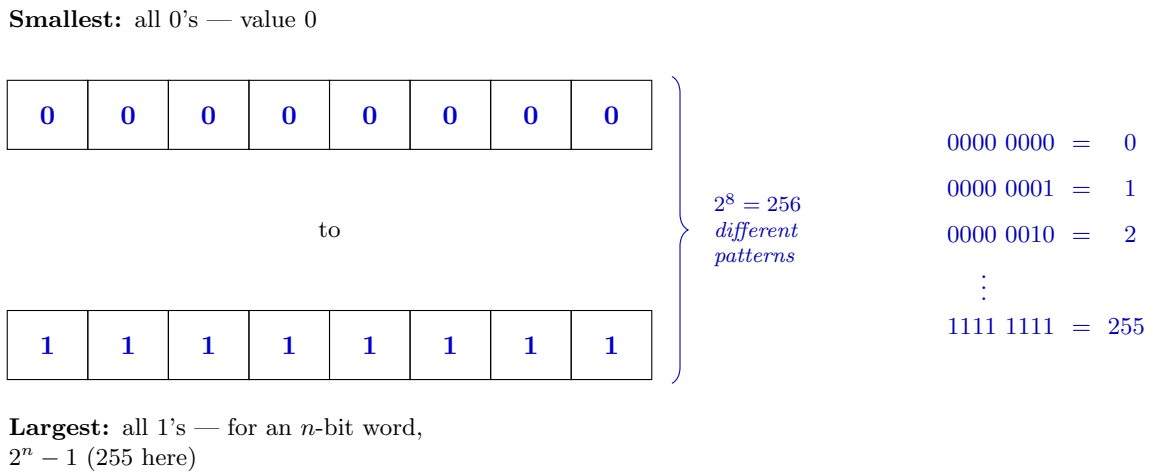


Figure 23: Why the largest unsigned value is $2^n - 1$ and not 2^n. Counting up from all-zeros steps through every pattern in turn — each extra bit doubles how many there are, so 8 bits give $2^8 = 256$ patterns — and because the count starts at 0, the last one is 255.

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. $1\mathrm{A}_{16} = 1 \cdot 16^1 + 10 \cdot 16^0 = 16 + 10 = 26$.

- **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

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.

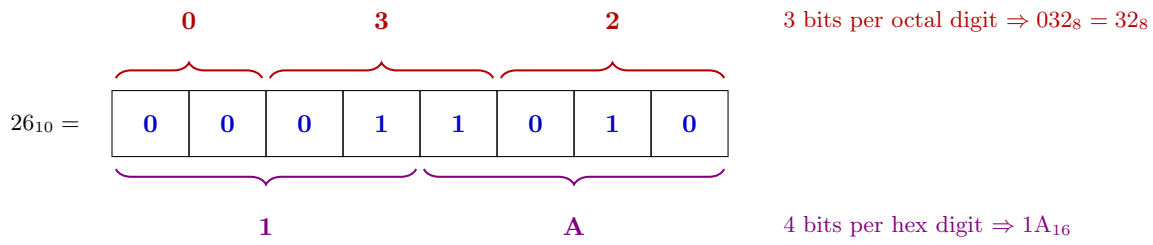


Figure 24: One bit pattern read two ways. Grouping 00011010_2 into nibbles from the right gives 1 and A, i.e. $1A_{16}$; grouping the same bits in threes gives 0, 3 and 2, i.e. 32_8. The grouping must start at the right — starting at the left gives the wrong answer whenever the width is not a multiple 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[69] 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**

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

Radix	Convention	Example	Where used
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.**

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[70]).

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
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).

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

- **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
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[71] for the full 4-bit table across all five formats, and 2026-08-04-binary-arithmetic[72] for what happens when you add these.

[71] courses/csse2010/week1-lab-number-representations.pdf
[72] courses/csse2010/2026-08-04-binary-arithmetic.pdf

Fractions: fixed and floating point

Everything above is whole numbers. Place values carry on past a **binary point** as $\frac{1}{2}, \frac{1}{4}, \frac{1}{8}, ...$, 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[73].
- **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[74].

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

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

Name	AND form	OR form
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:

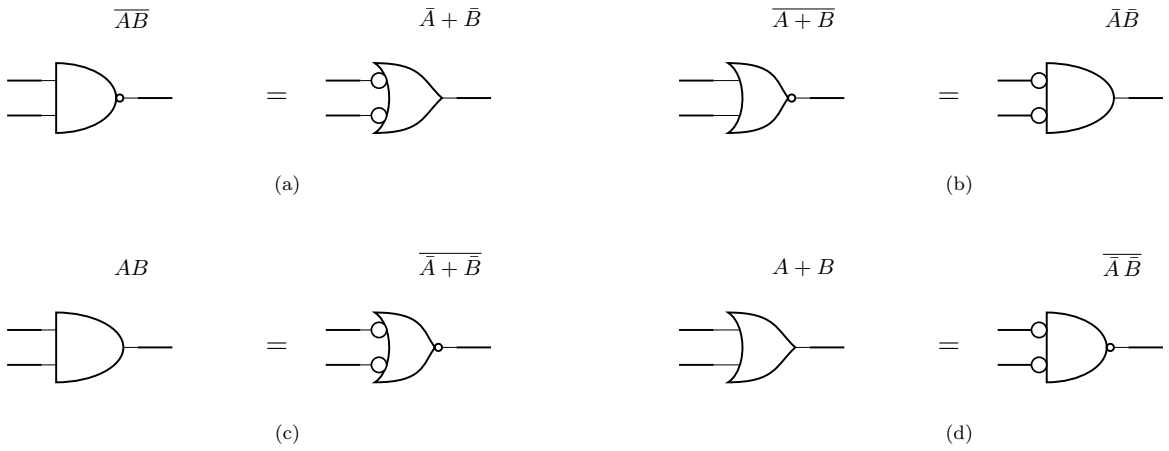


Figure 25: De Morgan's law drawn as gate symbols. (a) a NAND is an OR with inverted inputs; (b) a NOR is an AND with inverted inputs; (c) and (d) are the same two statements read the other way. Each pair is the same function, so either symbol may be used - pick whichever makes the surrounding bubbles cancel.

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[75]) — an AND feeding an OR can always be redrawn as a NAND feeding a NAND.

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

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[76].

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[77]). 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[78] for pin layouts). The rules are set out in the *CSSE2010/CSSE7201 - Guide to Drawing Circuit Schematics* (Blackboard, v2.3).

[76] courses/csse2010/2026-07-30-intro-to-logic-gates.pdf
[77] courses/csse2010/logic-gates.pdf
[78] courses/csse2010/device-pinouts.pdf

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.
- **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[79]).
- **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

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.

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

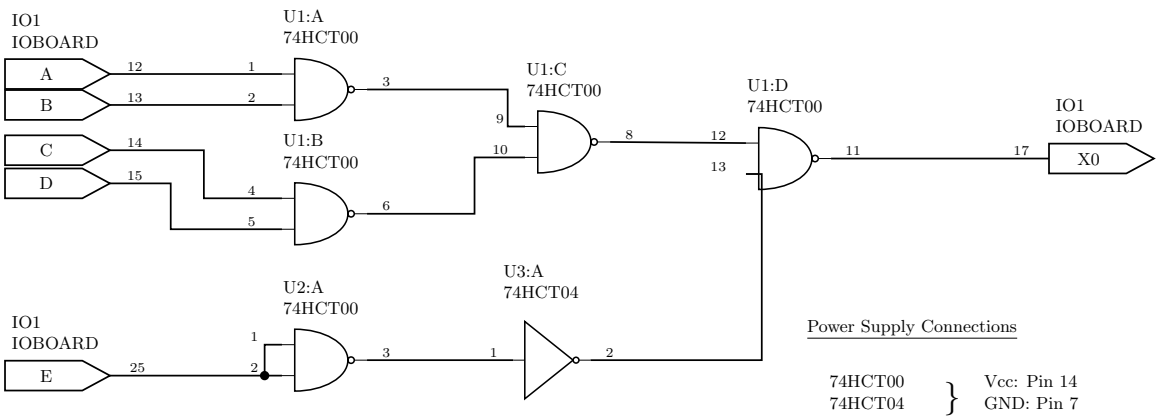


Figure 26: A correct schematic. Every rule above is visible here at once: five named inputs (A-E) and one named output (X0) on the IO board, four gates of one 74HCT00 labelled U1:A to U1:D, a fifth NAND that had to move to a second chip U2, an inverter on a third chip U3 because it is a different gate 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.

- **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 $1 \rightarrow 2$.
- **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

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.)

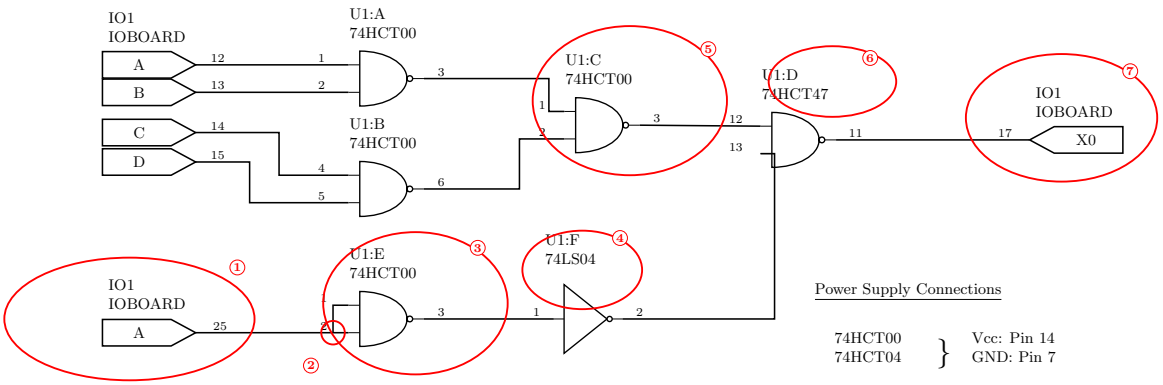


Figure 27: The same circuit with seven deliberate mistakes ringed and numbered left to right, matching the list below. Compare each ring against the corresponding part of the correct schematic above.

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 26.
   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[80] — the pin numbers and symbols every schematic has to match
- logic-gates[81] — the gate symbols themselves, and the logic-diagram level above this one
- week3-lab-lab-4-pre-work[82] — the preparation task that asks for schematics of a 2-to-1 mux, a 2-to-4 decoder and a 4-bit adder/subtractor

[80] courses/csse2010/device-pinouts.pdf
[81] courses/csse2010/logic-gates.pdf
[82] courses/csse2010/week3-lab-lab-4-pre-work.pdf

Combinational Logic Blocks

Reference note for the standard combinational building blocks: the adder-subtractor, the multiplexer, and the decoder. See logic-gates[83] for the gate symbols/truth tables and boolean-algebra[84] 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[85], 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).

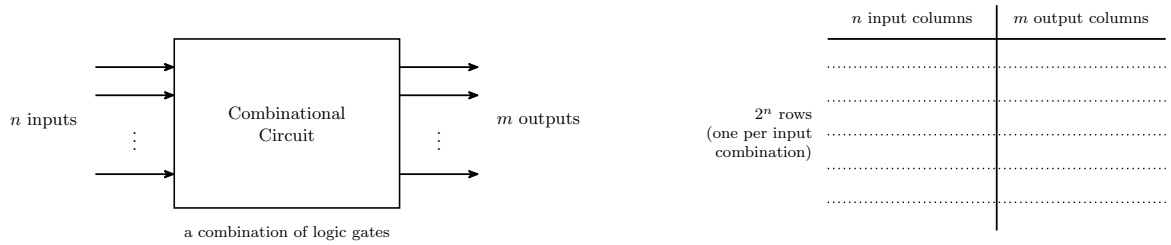


Figure 28: The general shape of a combinational circuit and the truth table it is always equivalent to. The row count 2^n depends only on the number of inputs, and the output column count m only on the number of outputs — so a block with six inputs needs 64 rows however few gates are inside it.

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[86]).

[83] courses/csse2010/logic-gates.pdf
[84] courses/csse2010/boolean-algebra.pdf
[85] courses/csse2010/sequential-circuits.pdf
[86] courses/csse2010/binary-number-representations.pdf

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)

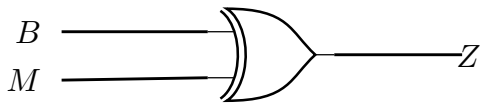
We need a gate that flips a bit *only sometimes*: given a mode bit M,

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

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**:

$$Z = B \oplus M$$

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 29: 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[87]) plus four XOR gates:

[87] courses/csse2010/2026-08-04-binary-arithmetic.pdf

- 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.

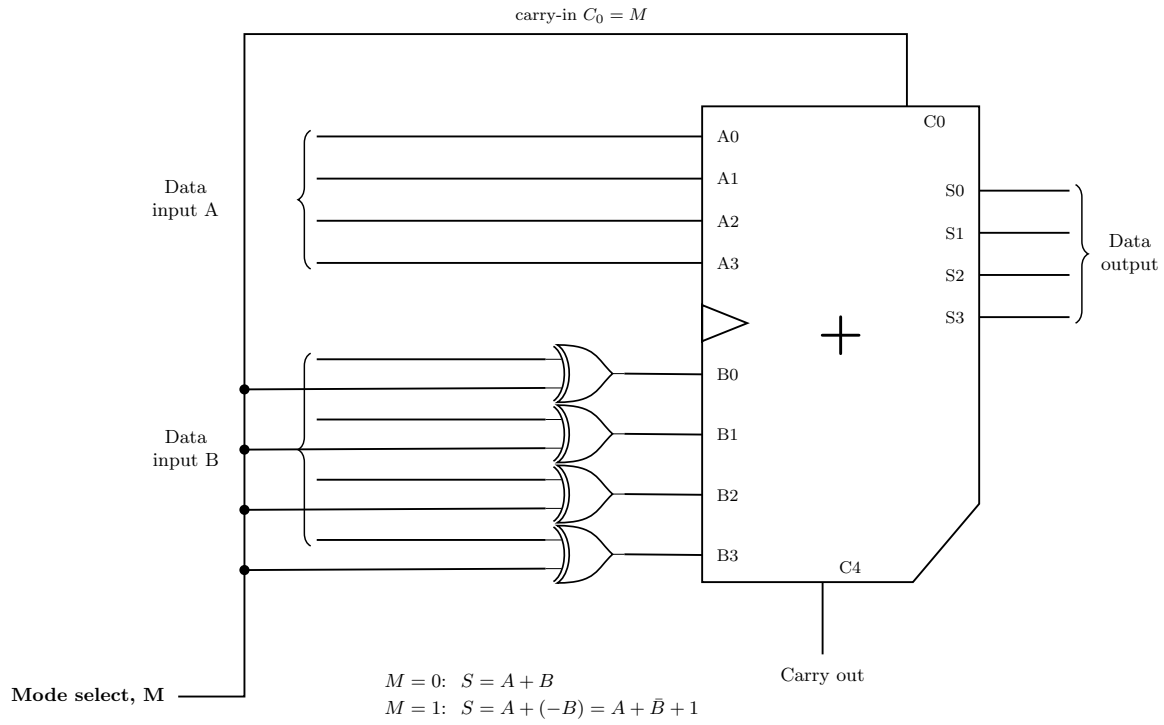


Figure 30: The 4-bit adder-subtractor. One mode line does two jobs: it conditions every bit of B through an XOR, and it is the adder's carry-in. With M = 0 the XORs pass B through and the carry-in is 0, so the adder computes A + B; with M = 1 the XORs deliver B-bar and the carry-in is 1, so it computes A + B-bar + 1, which is A + (-B). No extra hardware is spent on the +1 at all.

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[88].

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.

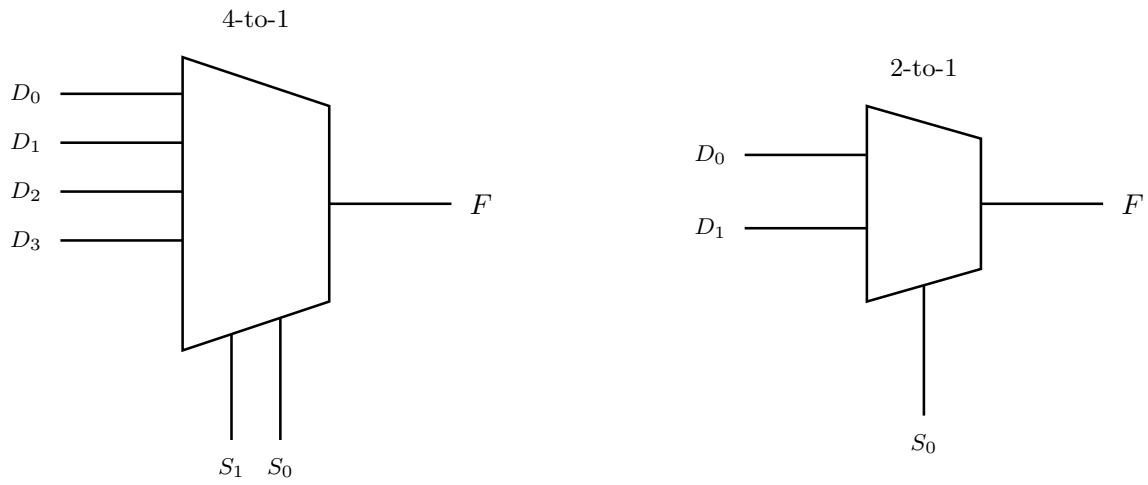


Figure 31: The multiplexer logic symbol at two sizes, 4-to-1 and 2-to-1. Data inputs enter the tall left edge, the single output leaves the short right edge, and the select inputs enter from underneath. The number of select lines is what fixes the size: n selects steer 2^n data inputs.

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:

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

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[89]) — 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

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

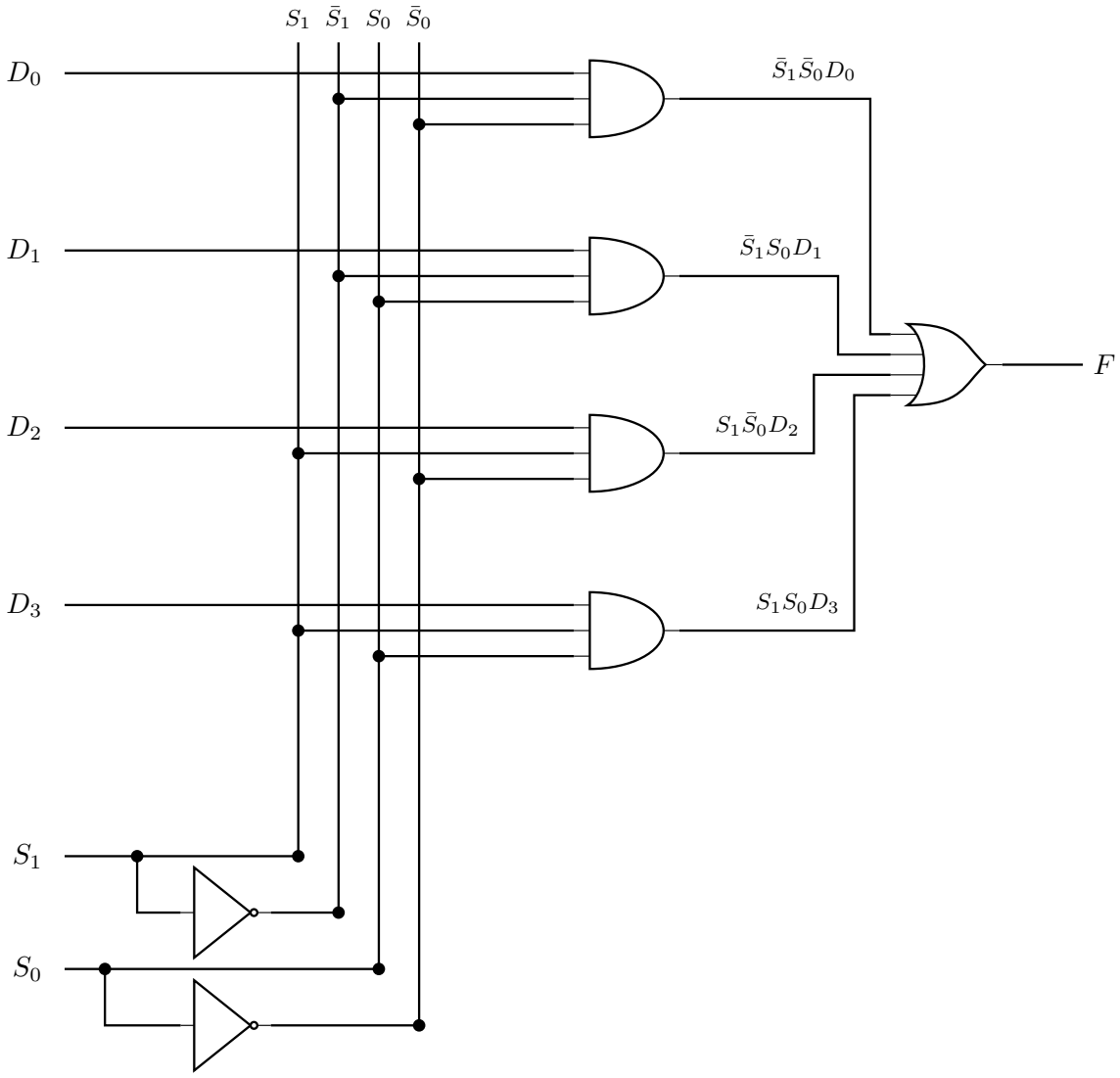


Figure 32: A 4-to-1 multiplexer built from gates. The two inverters generate the complemented selects; the four vertical rails carry S1, S1-bar, S0 and S0-bar past every AND gate; each AND gate is one data input gated by the one select combination that names it, and the OR gate collects them. Only the dots are connections — every other crossing is two wires passing.

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
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[90].

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[91].

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 $D_0 \ldots D_7$.

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

[90] courses/csse2010/device-pinouts.pdf
[91] courses/csse2010/2026-08-06-combinational-logic.pdf

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

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.

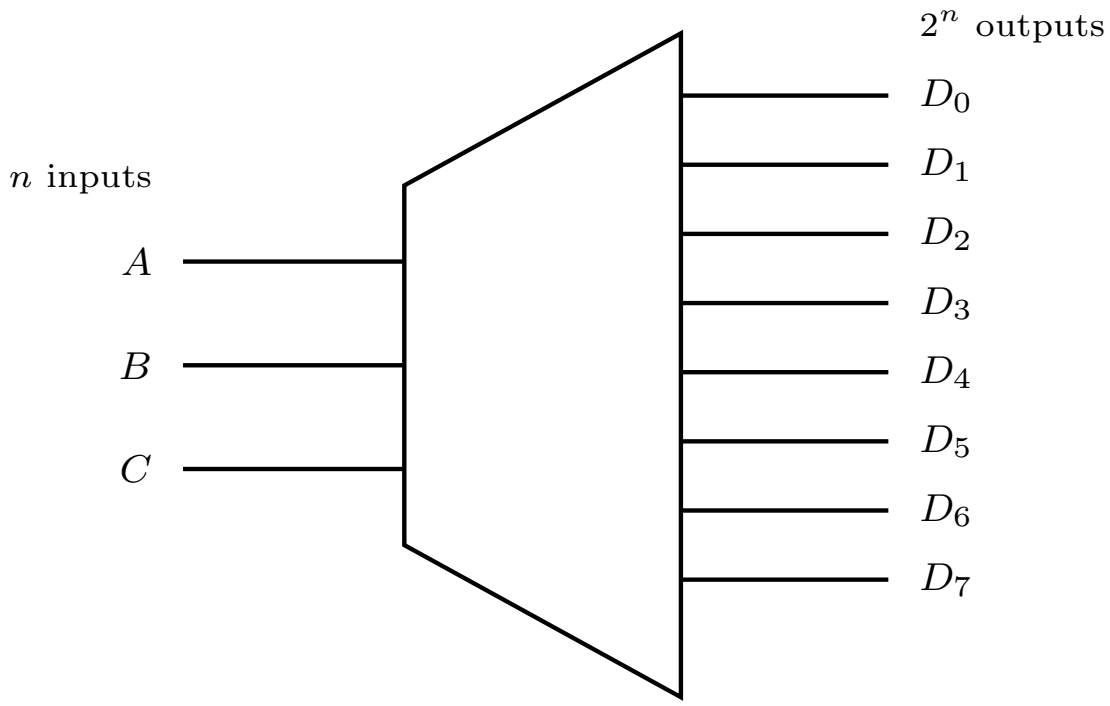


Figure 33: The decoder as a block: n inputs in, 2^n outputs out, and the shape widens accordingly. It is the mux's opposite — a mux funnels 2^n inputs down to one output under n selects, a decoder fans n inputs out to 2^n outputs.

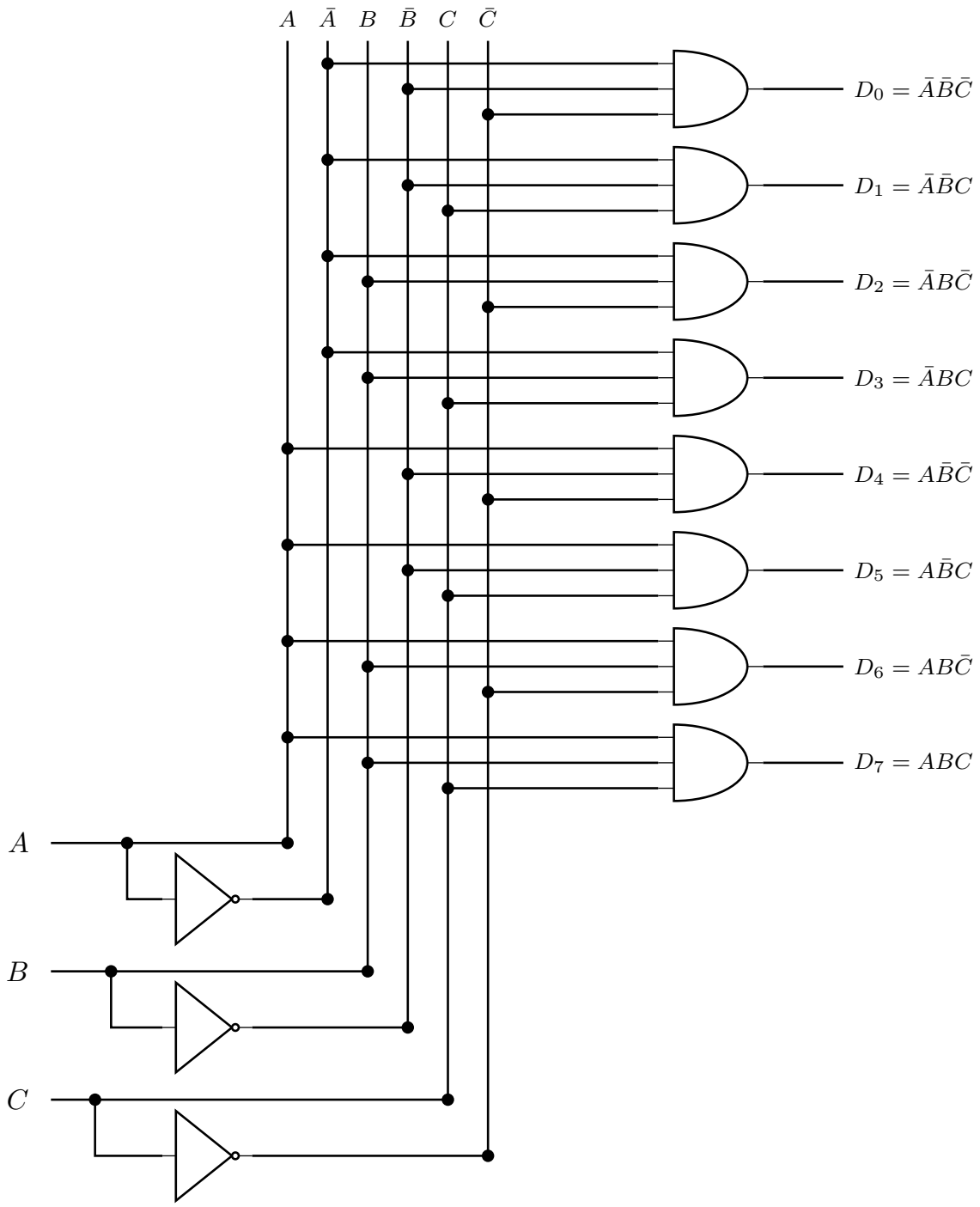


Figure 34: A 3-to-8 decoder built from gates. Three inverters generate the complements onto six vertical rails; each of the eight AND gates taps the one literal triple that matches its index, so it is the minterm for that input combination. Exactly one AND gate can be high at a time, which is the decoder's defining property.

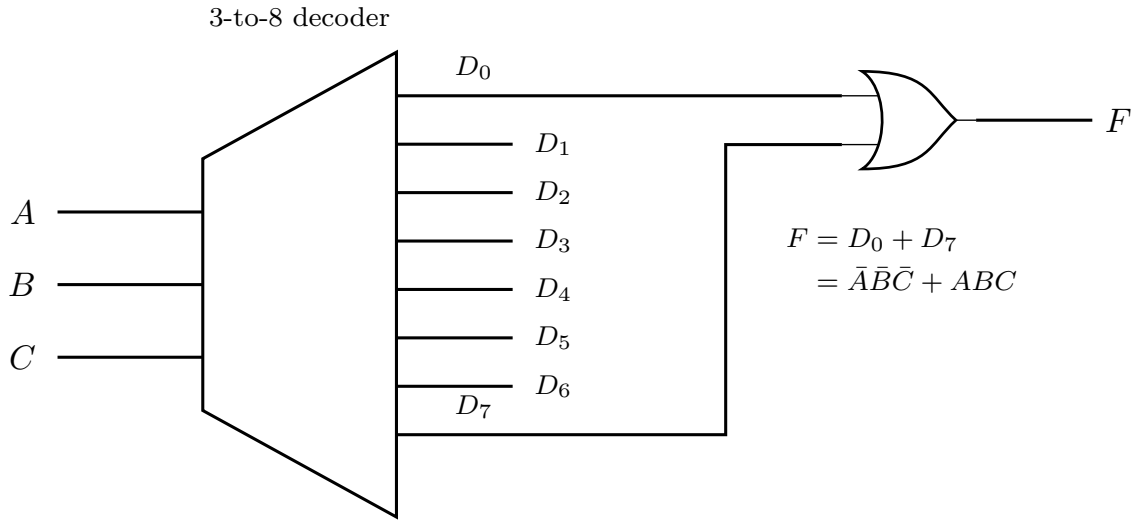


Figure 35: A function built on a decoder: F = A-bar B-bar C-bar + ABC is D0 + D7, so it costs a single OR gate on top of the decoder. Any function of A, B and C can be built this way — pick out the outputs for its true rows and OR them together.

See also

- 2026-08-06-combinational-logic[92] — Lecture 4, where these blocks were introduced
- device-pinouts[93] — the off-the-shelf parts: 74HCT157 (mux), 74HCT283 (adder), 74HCT86 (XOR)
- arithmetic-logic-unit[94] — the ALU bit slice, which is a mux selecting between several function blocks
- binary-number-representations[95] — two's complement, on which the subtractor depends

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[96] built from D flip-flops[97] plus some combinational logic in the feedback path.

[92] courses/csse2010/2026-08-06-combinational-logic.pdf
[93] courses/csse2010/device-pinouts.pdf
[94] courses/csse2010/arithmetic-logic-unit.pdf
[95] courses/csse2010/binary-number-representations.pdf
[96] courses/csse2010/sequential-circuits.pdf
[97] courses/csse2010/flip-flops-and-latches.pdf

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

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

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[98].

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 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[99].

[98] courses/csse2010/logic-gates.pdf
[99] courses/csse2010/boolean-algebra.pdf

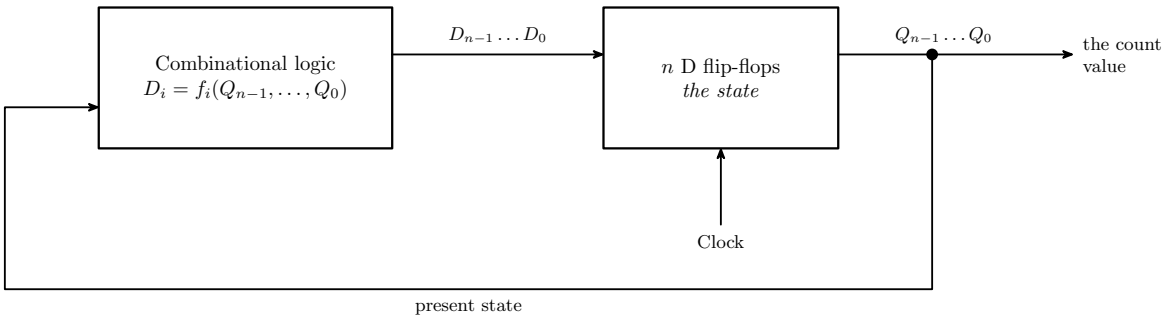


Figure 36: A counter is the general sequential-circuit block diagram with the external inputs deleted. That diagram, as Mano draws it, is a combinational circuit feeding flip-flops, the flip-flop outputs routed back round to the combinational circuit, and the clock entering the flip-flops on its own 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.

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[100] for schematic conventions and device-pinouts[101] 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

[100] courses/csse2010/circuit-schematics.pdf
[101] courses/csse2010/device-pinouts.pdf

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

$$D_0 = Q_1$$

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_1D_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[102] 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.

[102] courses/csse2010/week4-lab-lab-7-exercises.pdf

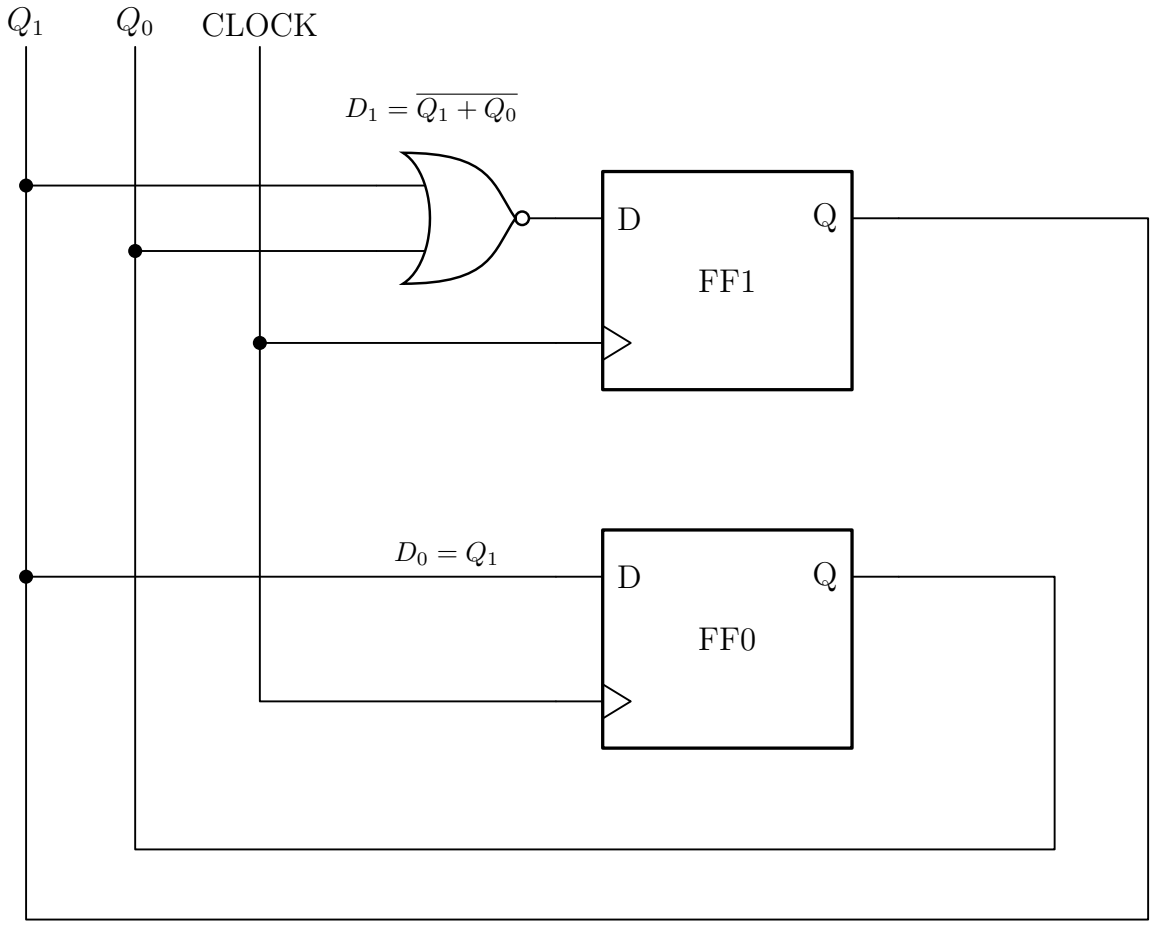


Figure 37: The whole counter, drawn from the derivation above. Each Q runs back to a vertical rail that the next-state logic taps, which is the physical form of the feedback loop in the block diagram earlier: D0 is FF1's Q wired straight across with no gate at all, and D1 is one NOR gate. Dots are connections; 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.

Related

- flip-flops-and-latches[103] — the D flip-flop and its characteristic table.
- sequential-circuits[104] — state, present state / next state, synchronous sequential circuits.
- shift-registers[105] — the other family of registers built from a flip-flop chain.
- boolean-algebra[106] — sum of products and the simplification laws used in step 4–5.
- binary-number-representations[107] — the binary counting sequence itself.
- state-machines[108] — the same design recipe generalised: a counter is a state machine whose transitions carry no input condition.
- 2026-08-17-counters[109] — Lecture 7.
- csse2010-diagram-notation[110] — 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[111]; for the 74-series pinouts and their schematic symbols see device-pinouts[112]. 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

[103] courses/csse2010/flip-flops-and-latches.pdf
[104] courses/csse2010/sequential-circuits.pdf
[105] courses/csse2010/shift-registers.pdf
[106] courses/csse2010/boolean-algebra.pdf
[107] courses/csse2010/binary-number-representations.pdf
[108] courses/csse2010/state-machines.pdf
[109] courses/csse2010/2026-08-17-counters.pdf
[110] courses/csse2010/csse2010-diagram-notation.pdf
[111] courses/csse2010/circuit-schematics.pdf
[112] courses/csse2010/device-pinouts.pdf

- *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
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
`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 ($\bar{A}$).
- **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 ($\bar{A}BC$, $AB\bar{C}$, $A\bar{B}C$, $\bar{A}\bar{B}C$ — 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` $\bar{A}$ `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[113] 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).

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

- **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.
- **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`, `RĒ SĒT`, 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	
	Ū	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[114] and cpu-organisation[115].

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).

[114] courses/csse2010/avr-architecture.pdf
[115] 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), `PR`$\bar{\texttt{E}}$ (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[116]). 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.

[116] 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[117] — 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[118] — 74-series pinouts, the logic-diagram vs circuit-schematic symbol columns, the IO board
- logic-gates[119] — the seven gate types and their truth tables
- timing-diagrams[120] — what a timing diagram means, as opposed to how it is drawn
- state-machines[121] — state diagrams and state tables as content
- boolean-algebra[122] — the identities the transition-label algebra uses
- avr-architecture[123] — 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[124]. 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:

[117] courses/csse2010/circuit-schematics.pdf
[118] courses/csse2010/device-pinouts.pdf
[119] courses/csse2010/logic-gates.pdf
[120] courses/csse2010/timing-diagrams.pdf
[121] courses/csse2010/state-machines.pdf
[122] courses/csse2010/boolean-algebra.pdf
[123] courses/csse2010/avr-architecture.pdf
[124] 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[125] and logic diagrams
Circuit schematic symbol	the shape *plus* device ID, device type and pin numbers	circuit-schematics[126]

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.

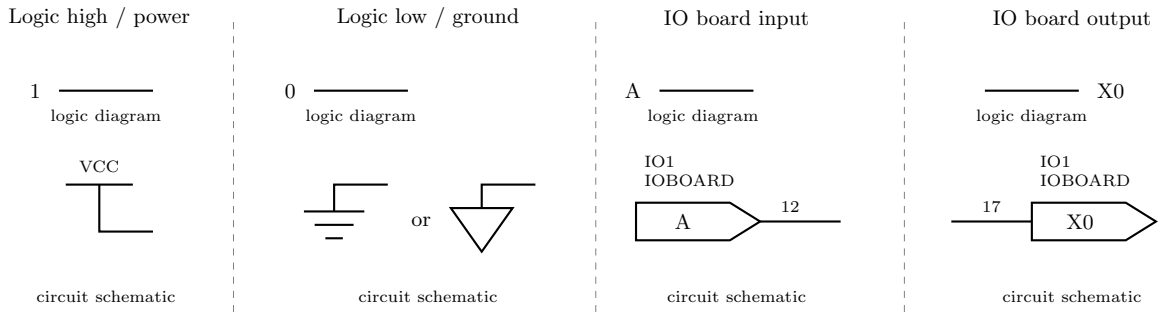


Figure 38: The four non-gate symbols. Logic high and logic low are a bare 1 or 0 on a logic diagram, and a VCC bar or a ground symbol (either of the two accepted forms) on a circuit schematic. The IO board input and output symbols are the same pentagon; the difference is that an input's wire leaves the point and an output's wire arrives at the flat back.

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

[125] courses/csse2010/logic-gates.pdf
[126] courses/csse2010/circuit-schematics.pdf

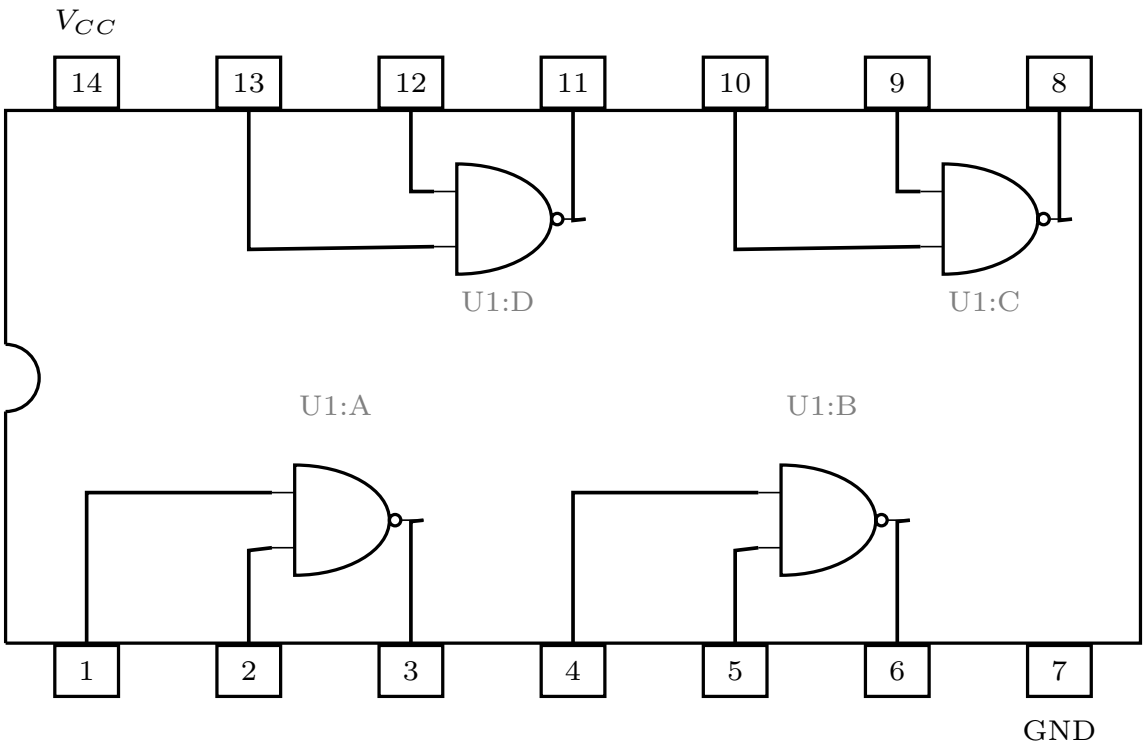


Figure 39: The 14-pin DIP pinout, drawn for the 74HCT00 but identical for the 08, 32 and 86 — only the gate shape inside changes. The notch marks the pin-1 end; pins run 1-7 along the bottom and 14-8 along the top. Reading the internal wiring is what tells you the gates pair pins 1,2 to 3, 4,5 to 6, 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.

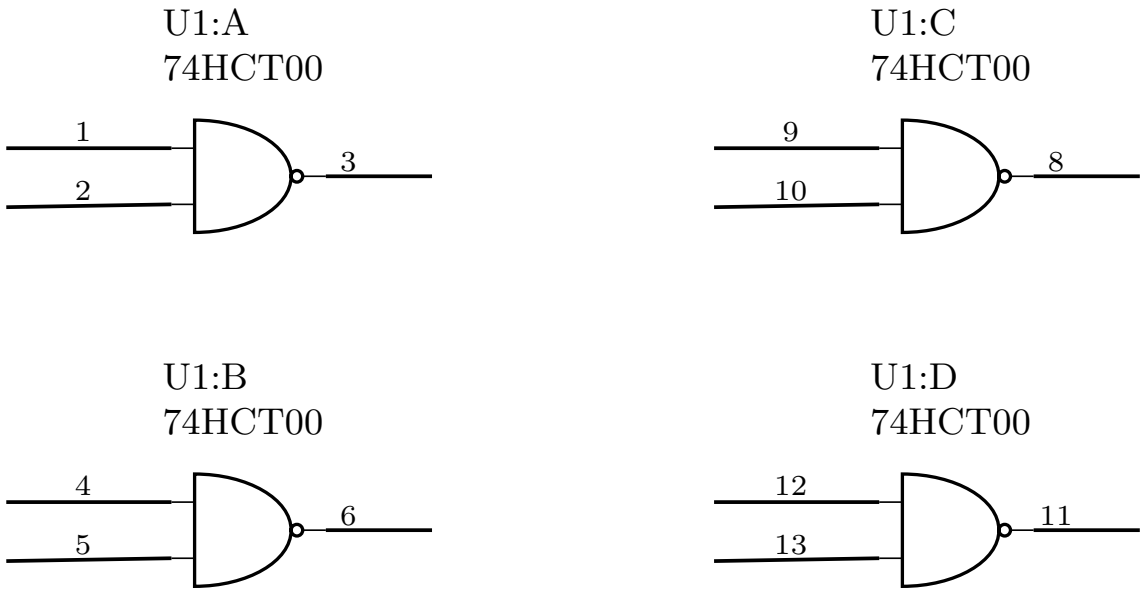


Figure 40: The four gates of a 74HCT00 as circuit schematic symbols: device ID, device type, and a pin number on every terminal. For a 74HCT08, 74HCT32 or 74HCT86 the pin numbers are exactly these; only the gate body changes to AND, OR or XOR.

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

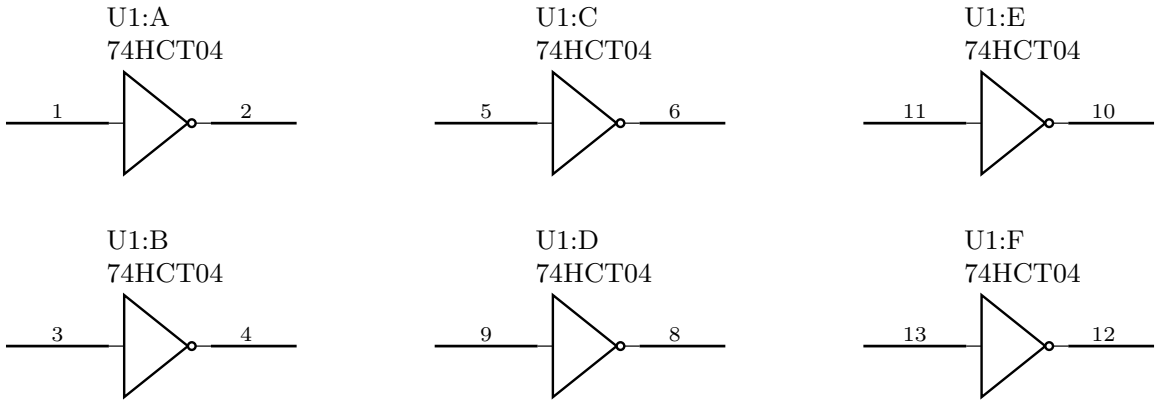


Figure 41: All six inverters of a 74HCT04. The numbering is not a simple run: the first three gates go low-to-high (1 to 2, 3 to 4, 5 to 6) but the last three run the other way (9 to 8, 11 to 10, 13 to 12), because the package is symmetric about its centre.

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.

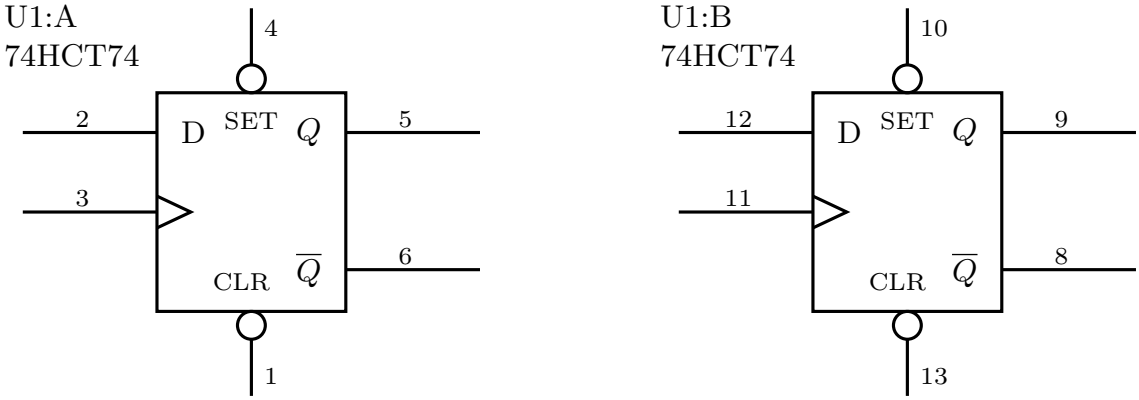


Figure 42: The two flip-flops of a 74HCT74 as circuit schematic symbols. Unlike the 74HCT175 below, each flip-flop has its own clock, its own set and its own clear — nothing is shared. SET and CLR carry bubbles, and the wedge on the left-hand lower input marks the clock.

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[127] 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.

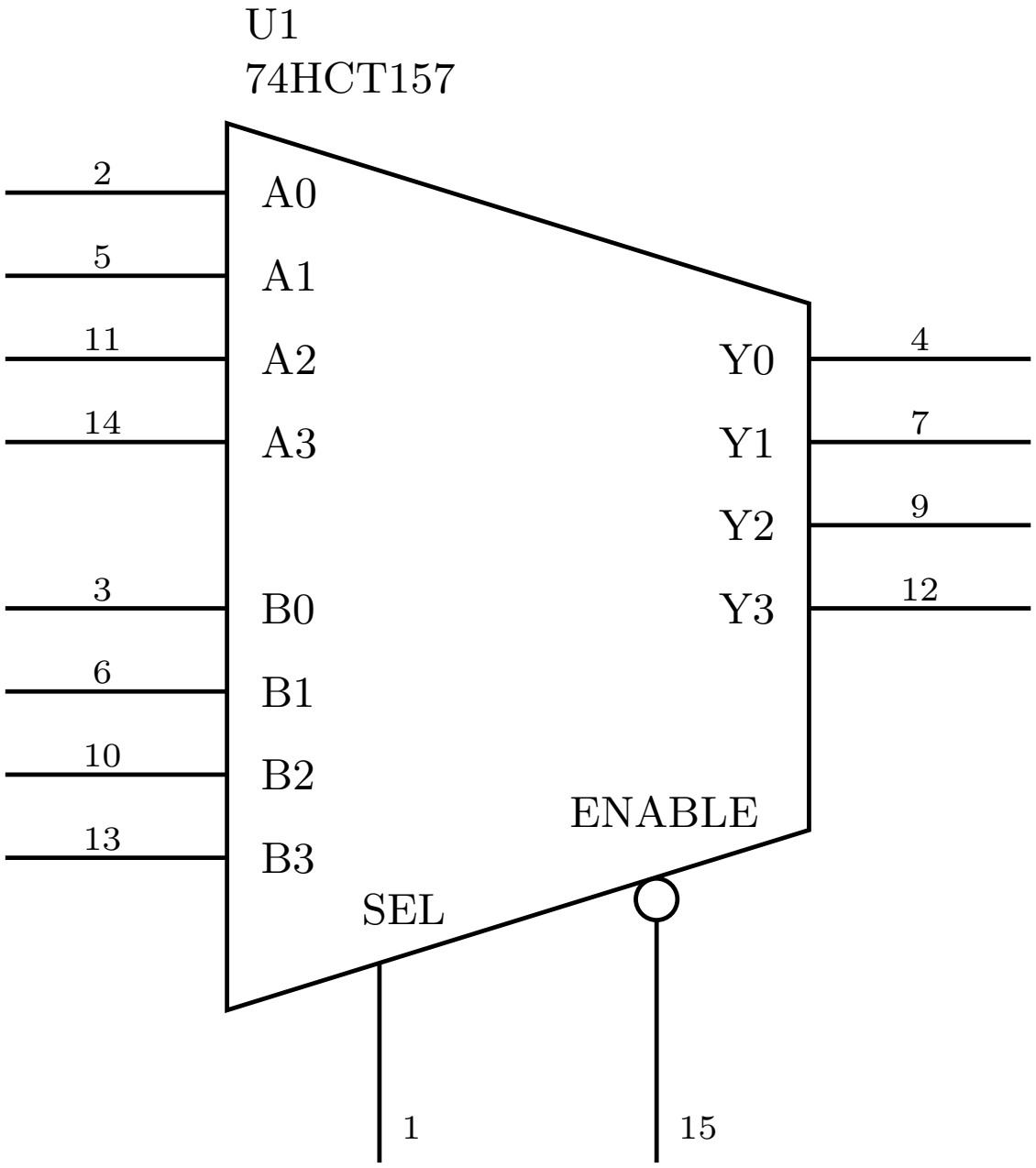


Figure 43: The 74HCT157 as one circuit schematic symbol — the multiplexer trapezoid, tall edge for the eight data inputs and short edge for the four outputs. It is four 2-to-1 multiplexers sharing a single SEL line and a single active-low ENABLE, so it selects between two whole 4-bit words rather than between two individual bits: one value of SEL puts A0-A3 on Y0-Y3 and the other puts B0-B3 there.

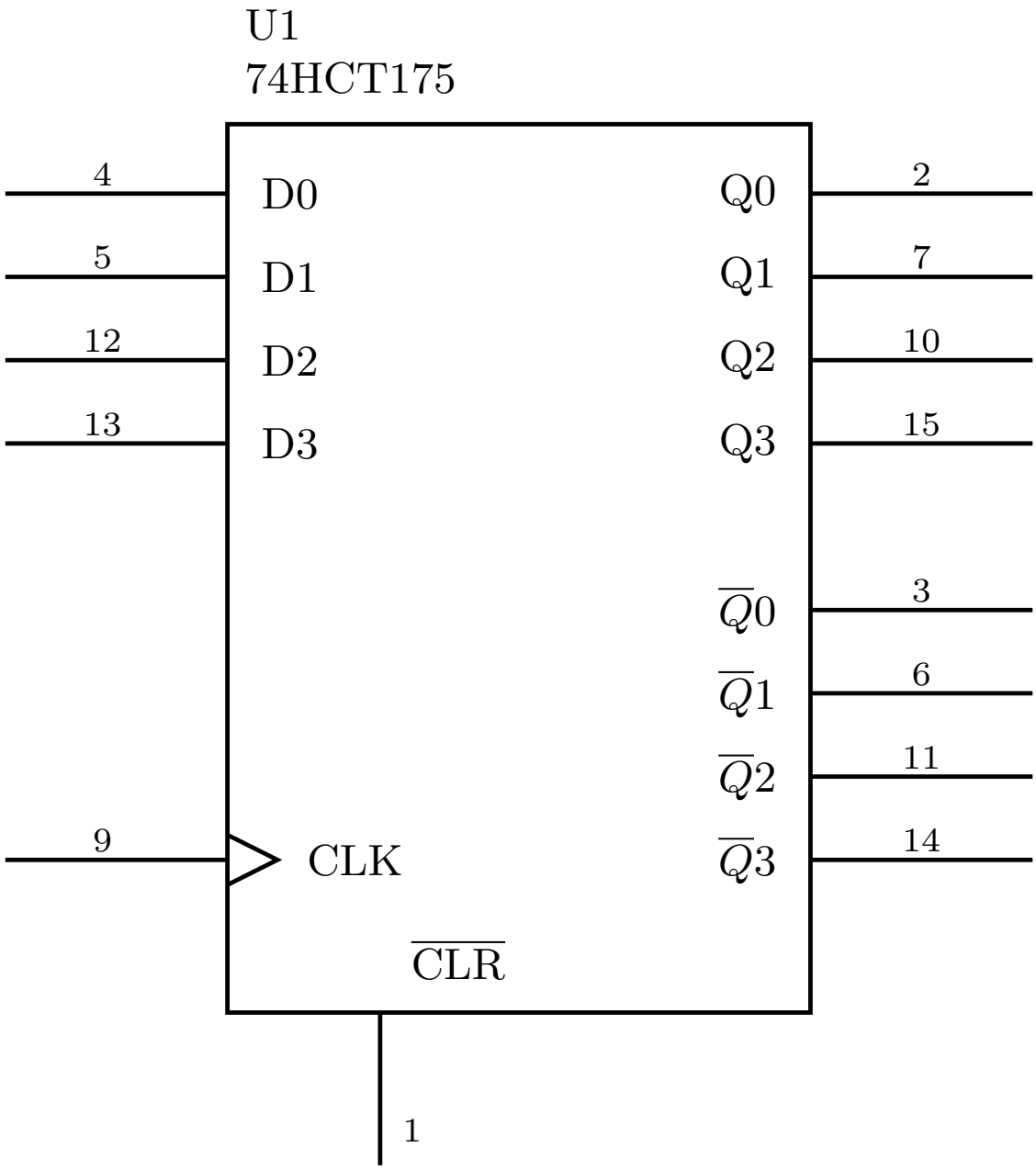


Figure 44: The 74HCT175 as one circuit schematic symbol. CLK (pin 9) and CLR (pin 1) enter once and serve all four flip-flops, which is exactly what a flat pin table hides: this is a 4-bit register, not four independent flip-flops.

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[128].

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 38 and circuit-schematics[129]. 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[130] for what the board does electrically.

[128] courses/csse2010/combinational-logic-blocks.pdf
[129] courses/csse2010/circuit-schematics.pdf
[130] courses/csse2010/io-board.pdf

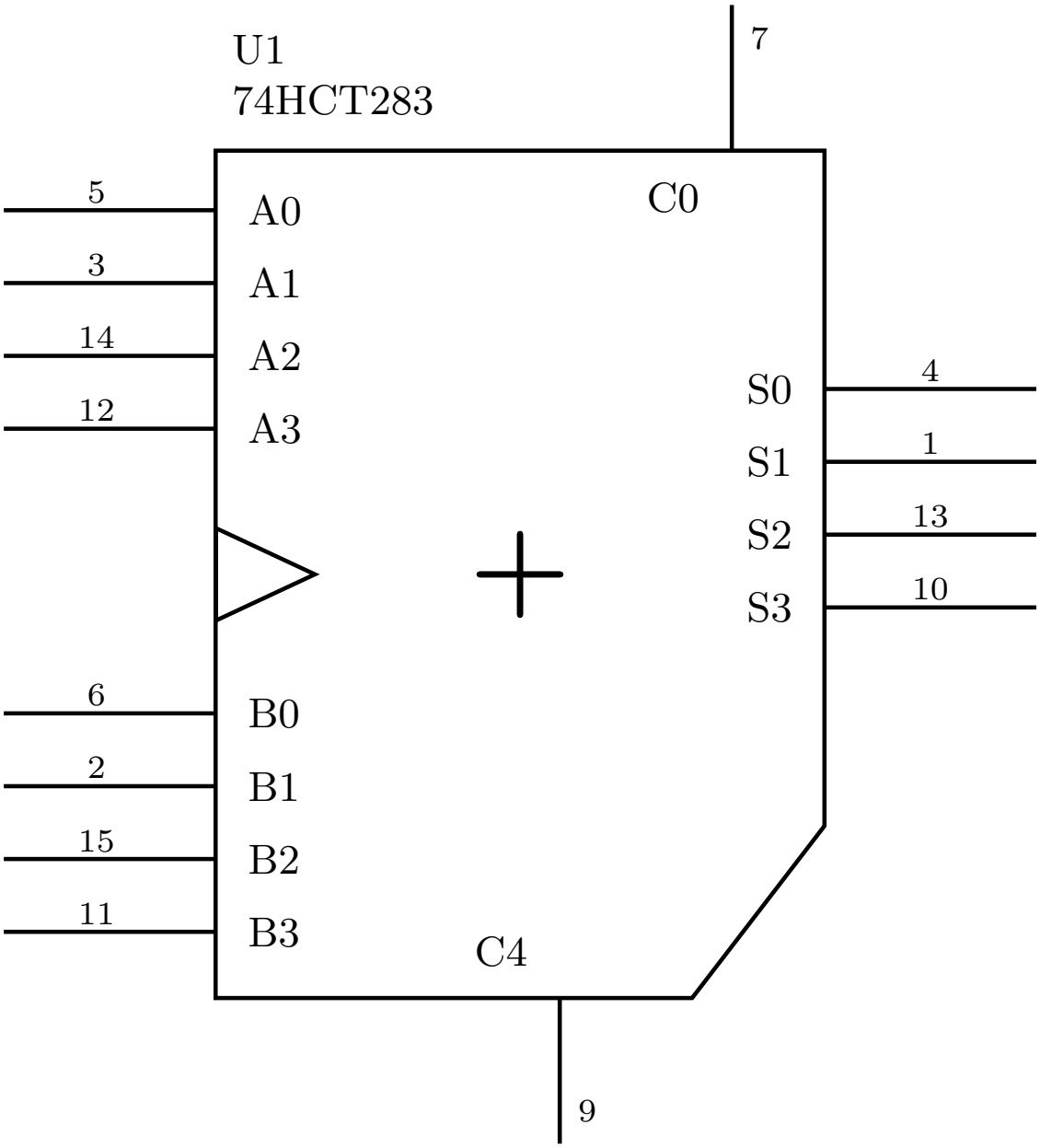


Figure 45: The 74HCT283 as one circuit schematic symbol. Internally it is four full adders with C1, C2 and C3 wired stage to stage inside the package, so only C0 and C4 are brought out to pins.

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[131].

Introduced in Lecture 5, 2026-08-10-introduction-to-sequential-circuits[132].

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[133].
- 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

[131] courses/csse2010/sequential-circuits.pdf
[132] courses/csse2010/2026-08-10-introduction-to-sequential-circuits.pdf
[133] 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[134] and [counters] call the same thing *Next State*. The course uses both names (see csse2010-diagram-notation[135]).

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.

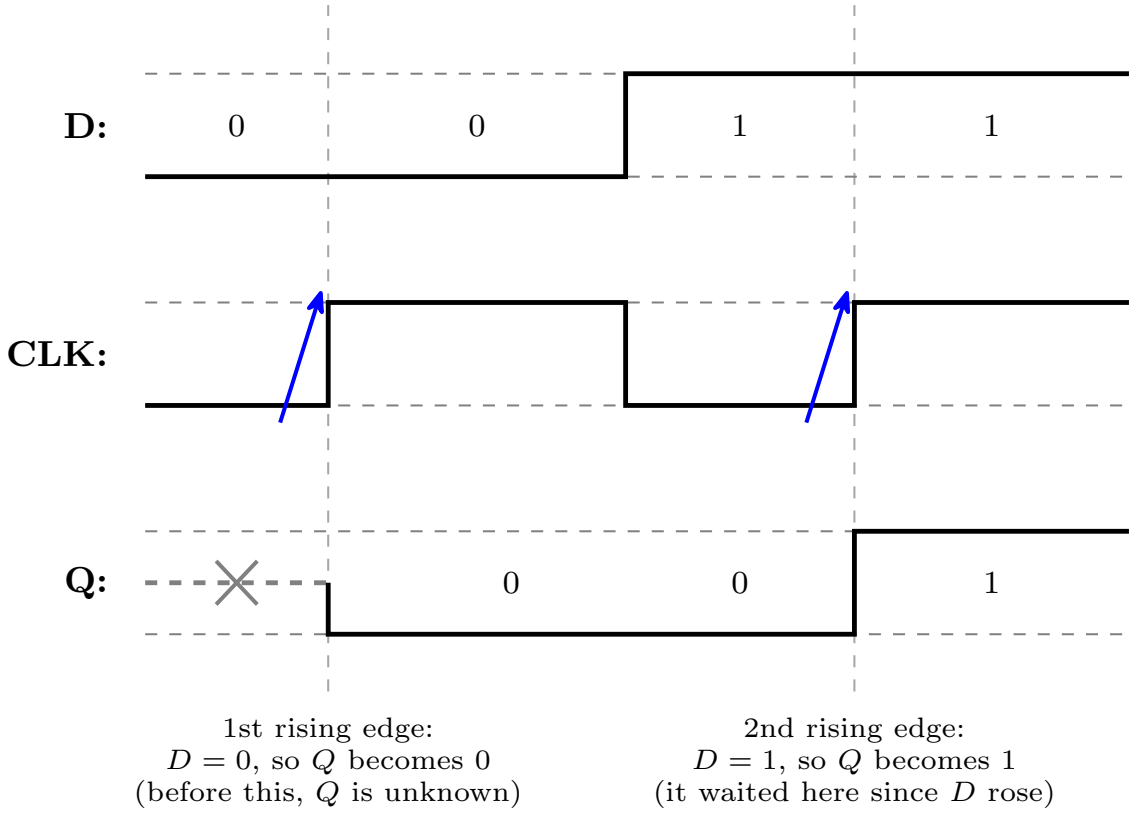


Figure 46: A D flip-flop waveform. The dashed verticals are the two rising clock edges, and the arrows mark them. Q is undefined (drawn dashed, with a cross) until the first edge. The point of the figure is the middle: D rises well before the second edge, and Q does not follow until that edge arrives.

[134] courses/csse2010/state-machines.pdf
[135] 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[136].

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.

[136] 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.

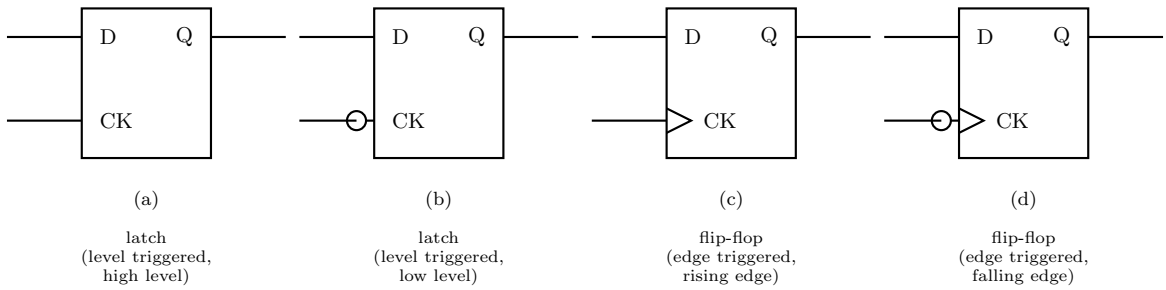


Figure 47: The four latch and flip-flop symbols, with the clock pin labelled CK (the course writes this pin CLK elsewhere). The wedge and the bubble read independently: the wedge says edge-triggered, therefore flip-flop; the bubble says inverted, therefore falling edge or low level. The reading under 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:

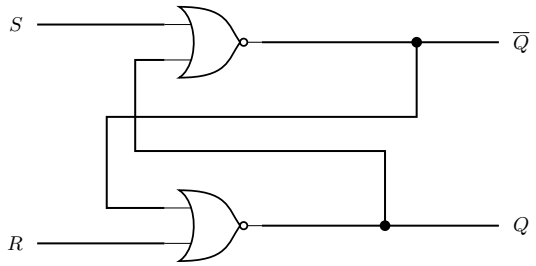
- The **triangle** indicates **edge-triggered**, and therefore that the device is a flip-flop rather than a latch. No triangle means level-triggered, i.e. a latch.
- The **bubble** indicates inversion, i.e. the *falling* edge (or the low level) rather than the rising edge (or the high level).

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 48: 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[137]): $\overline{A+B} = \bar{A}\cdot\bar{B}$.

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:

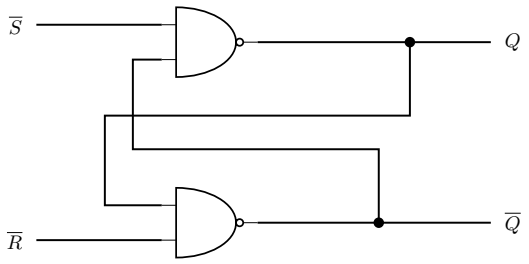
[137] courses/csse2010/boolean-algebra.pdf

- $S = 0, R = 0$: 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 49: 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[138].)

- $\bar{S} = \bar{R} = 1$: 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[139]). 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:

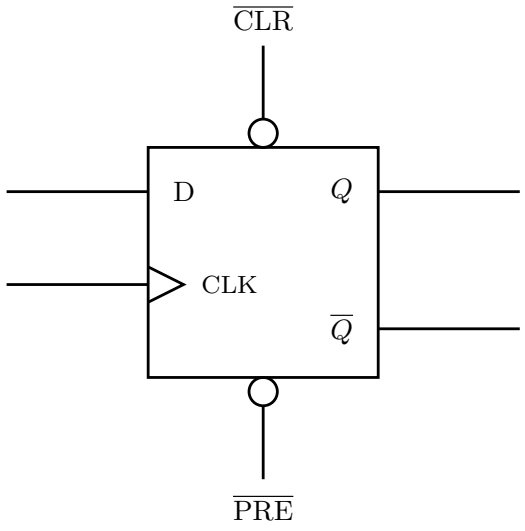
- $\overline{\textbf{PRE}}$ **(Preset)** — forces Q to 1.
- $\overline{\textbf{CLR}}$ **(Clear)** — forces Q to 0.

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

[138] courses/csse2010/csse2010-diagram-notation.pdf
[139] 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 50: 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[140] and circuit-schematics[141].

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

[140] courses/csse2010/device-pinouts.pdf
[141] courses/csse2010/circuit-schematics.pdf

schematic symbol in device-pinouts[142].

- **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[143] — Lecture 5, where all of this is introduced
- sequential-circuits[144] — what you build once you have a storage element
- shift-registers[145] · [counters] · state-machines[146]
- memory-technologies[147] — static RAM, a memory that keeps each bit in a latch
- week3-lab-logisim-and-flip-flops[148] — the lab that simulates these circuits
- logic-gates[149] · boolean-algebra[150] — the NOR/NAND behaviour the latch analysis rests on
- timing-diagrams[151] — how to read and draw the waveforms above
- csse2010-diagram-notation[152] — 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[153]; the kit itself is loaned out in week3-lab-logisim-and-flip-flops[154]. Two more parts of the kit that are used alongside the board, the

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

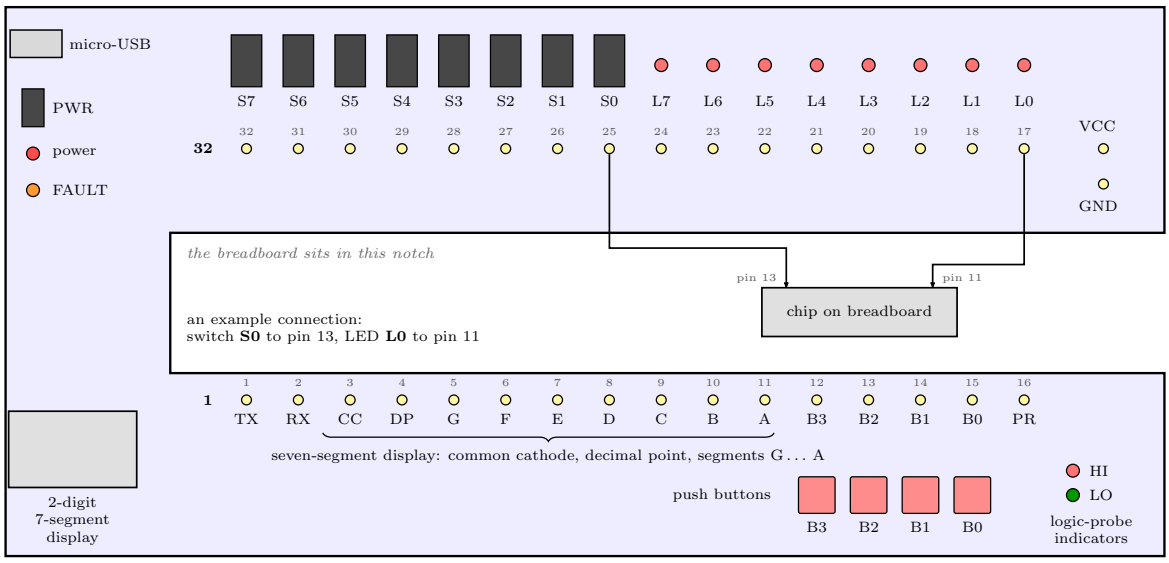


Figure 51: A plan view of the board, not to scale, with the parts on the left spine placed roughly. Each I/O device sits on one pin of one of the two 16-pin headers, and the column of breadboard holes at that pin is how you reach it. The board itself prints only the two end markers, a bold 32 at the 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 51: `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[155] make, and getting it backwards is one of the common errors listed there.

[155] 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[156]) 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[157]). What the logic probe shows for a voltage inside that band is not documented.

[156] courses/csse2010/circuit-schematics.pdf
[157] 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[158]).

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[159]).

[158] courses/csse2010/analog-to-digital-conversion.pdf
[159] courses/csse2010/avr-spi.pdf

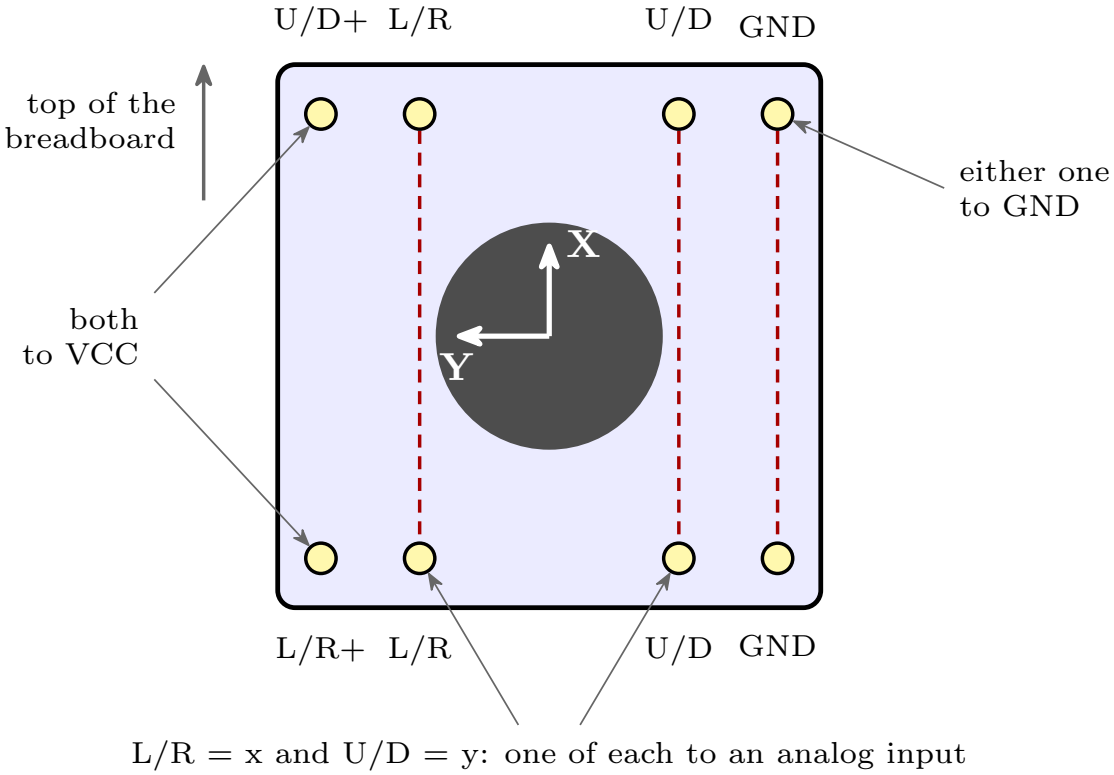


Figure 52: The joystick's eight pins, seen from above with the board placed as it sits in the breadboard, its X arrow pointing to the top. The pins near each corner come in twos. The dashed lines join the pins that are connected on the board itself: the two L/R pins, the two U/D pins and the two GND 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[160]).

See also

- device-pinouts[161] — the pin table behind Figure 51, and the board's schematic symbols
- circuit-schematics[162] — how the board is drawn on a schematic (`IO1` / `IOBOARD`, input and output pentagons)

[160] courses/csse2010/week10-lab-adc-and-spi.pdf
[161] courses/csse2010/device-pinouts.pdf
[162] courses/csse2010/circuit-schematics.pdf

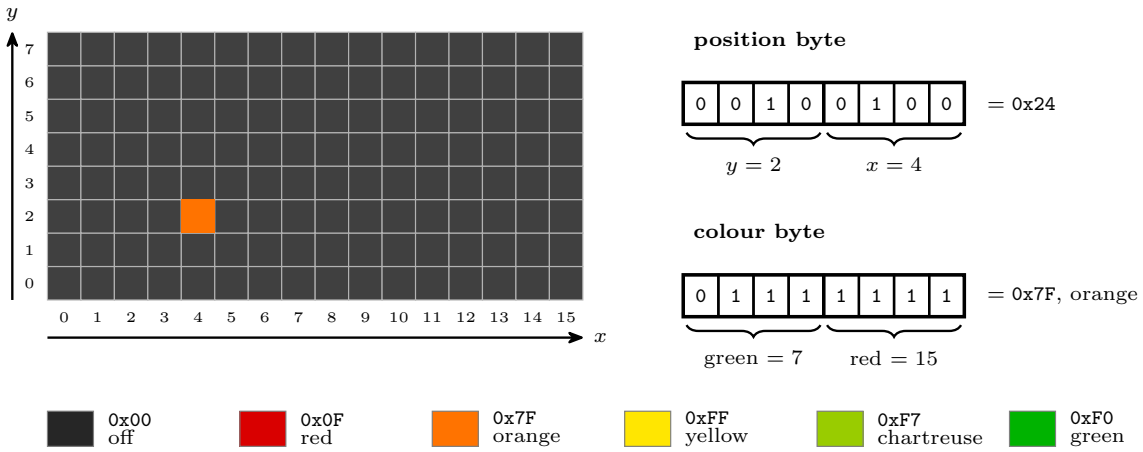


Figure 53: How a pixel is addressed and coloured. The matrix is 16 columns, x from 0 to 15 left to right, by 8 rows, y from 0 to 7 bottom to top. A position byte puts y in its high nibble and x in its low one, so the pixel at x = 4, y = 2 is `0x24`. A colour byte puts the green brightness in its high 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[163] — the same switches and LEDs driven from an AVR rather than from gates
- week3-lab-logisim-and-flip-flops[164] — where the kit is loaned out
- week10-lab-adc-and-spi[165] — Lab 16: the joystick read by the A/D converter, and the LED matrix driven over SPI
- analog-to-digital-conversion[166] · avr-spi[167] — the two AVR peripherals the joystick and the matrix are used with

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[168] for the storage elements themselves.

[163] courses/csse2010/avr-io-ports.pdf
[164] courses/csse2010/week3-lab-logisim-and-flip-flops.pdf
[165] courses/csse2010/week10-lab-adc-and-spi.pdf
[166] courses/csse2010/analog-to-digital-conversion.pdf
[167] courses/csse2010/avr-spi.pdf
[168] courses/csse2010/flip-flops-and-latches.pdf

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[169].

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[170])	Counters, registers (see [counters], shift-registers[171])

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[172]).

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.

[169] courses/csse2010/2026-08-10-introduction-to-sequential-circuits.pdf
[170] courses/csse2010/combinational-logic-blocks.pdf
[171] courses/csse2010/shift-registers.pdf
[172] courses/csse2010/timing-diagrams.pdf

- **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.

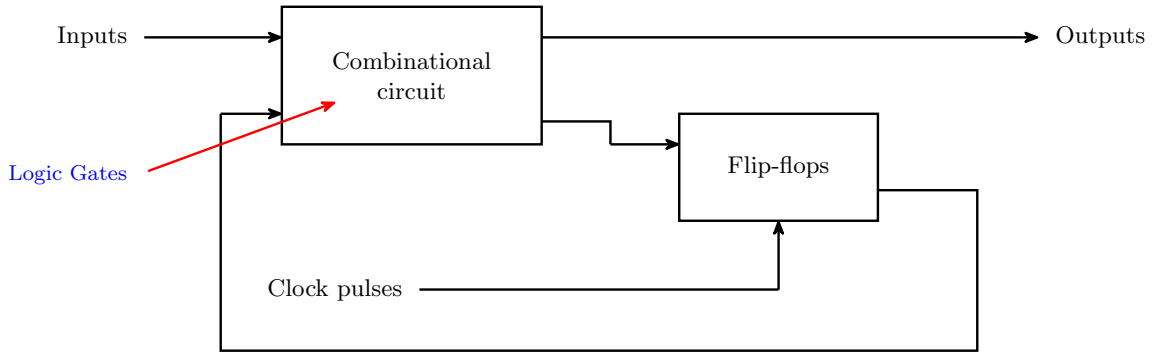


Figure 54: The general sequential circuit (after Mano, Digital Design, 3rd edition). Two things carry the whole idea: the loop running from the flip-flops back into the combinational block, which is what makes the circuit sequential, and the clock stub, which enters the flip-flops and nothing else. The 'Logic Gates' callout marks the combinational block as gates only.

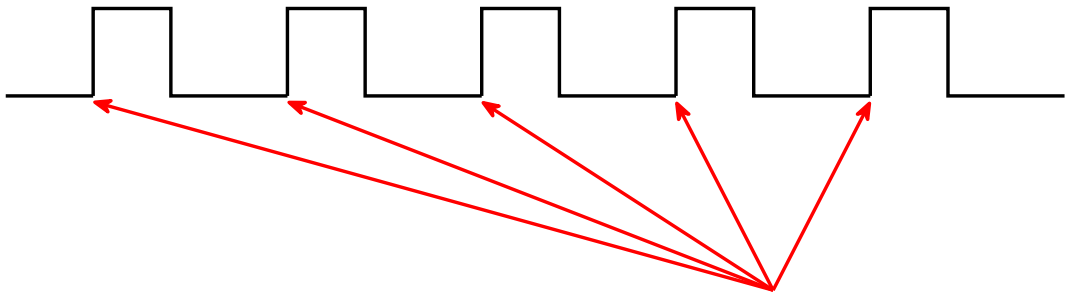
- A **combinational circuit** block (just logic gates). It takes the external **inputs** *plus* the feedback from the flip-flops, and produces the external **outputs** *plus* the values to be loaded into the flip-flops.
- A **flip-flops** block (the storage elements). Its data inputs come from the combinational 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[173], [counters] and state-machines[174] — 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 55: 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.

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[175].

[173] courses/csse2010/shift-registers.pdf
[174] courses/csse2010/state-machines.pdf
[175] courses/csse2010/timing-diagrams.pdf

See also

- 2026-08-10-introduction-to-sequential-circuits[176] — Lecture 5
- state-machines[177] — the same circuits described by their states and transitions, which is how they are actually designed.
- flip-flops-and-latches[178] · [counters] · shift-registers[179]
- csse2010-diagram-notation[180] — the block-diagram conventions the figures above follow

Shift Registers

Registers and shift registers are the simplest useful sequential circuits[181] — groups of D flip-flops[182] 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
`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)

[176] courses/csse2010/2026-08-10-introduction-to-sequential-circuits.pdf
[177] courses/csse2010/state-machines.pdf
[178] courses/csse2010/flip-flops-and-latches.pdf
[179] courses/csse2010/shift-registers.pdf
[180] courses/csse2010/csse2010-diagram-notation.pdf
[181] courses/csse2010/sequential-circuits.pdf
[182] courses/csse2010/flip-flops-and-latches.pdf

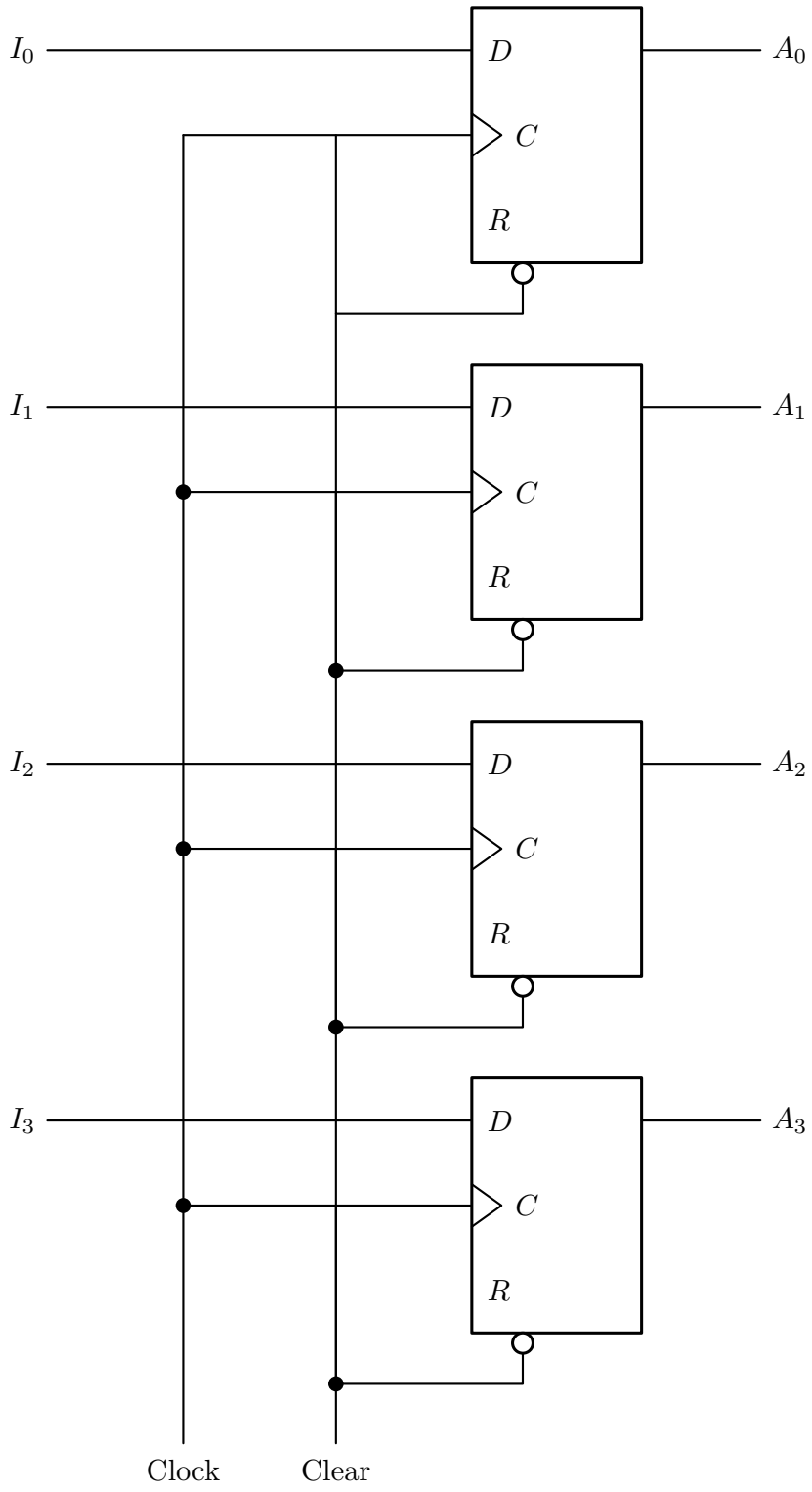


Figure 56: A 4-bit register (Mano, Digital Design, 3rd ed.). Four flip-flops, four parallel inputs, four parallel outputs, and exactly two shared control lines — no gate anywhere, which is what makes a register a sequential circuit without combinational logic. The pin labelling is Mano's, which differs 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).

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[183] 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.

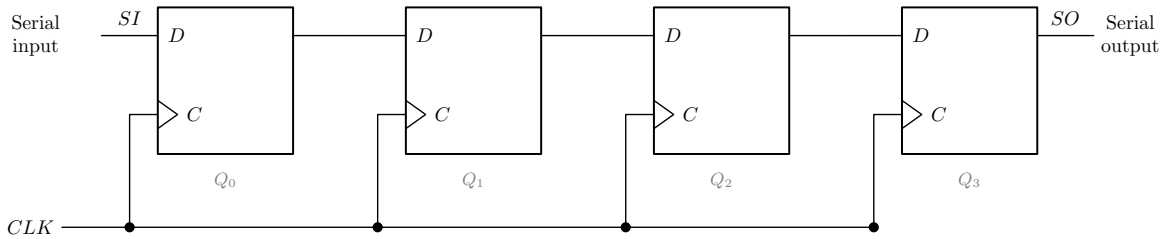


Figure 57: The 4-bit shift register (Mano's Fig. 6-3). The whole device is four D flip-flops and one wire each: the only structural difference from the register above is where each D input comes from. The grey stage names Q0 to Q3 are there so the trace table below can be read against the drawing.

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[184].

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

[183] courses/csse2010/flip-flops-and-latches.pdf
[184] courses/csse2010/sequential-circuits.pdf

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

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 58), 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 57, 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[185]) instead of directly from the previous stage:

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

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

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.

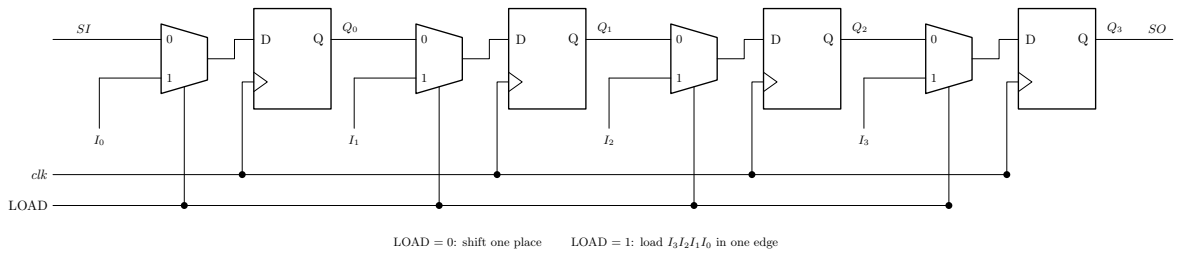


Figure 58: The parallel-load shift register: one 2-to-1 multiplexer in front of every flip-flop's D input, with a single LOAD line steering all four at once. Follow the top mux input across and it is the plain chain of the previous figure; follow the bottom input and every stage is loaded from its own 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[186] 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[187]):

- 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**;
- **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.

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

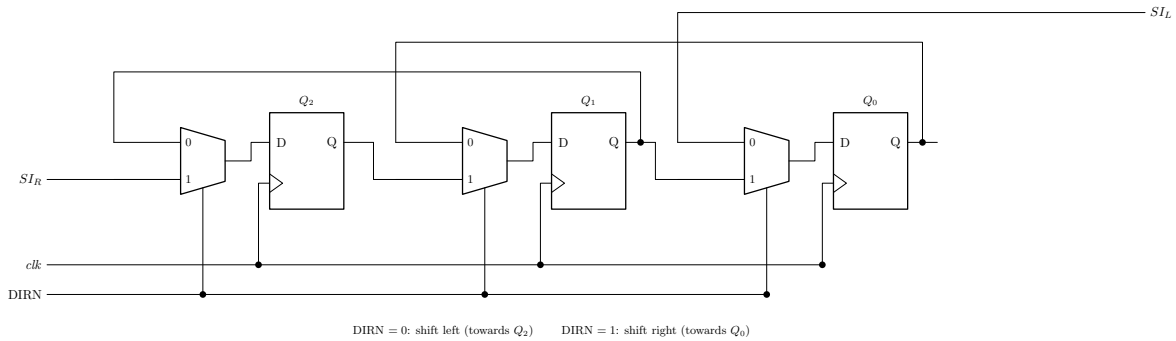


Figure 59: A 3-bit bidirectional shift register: three copies of the DIRN element, one shared DIRN line and one shared clock. Q2 is the most significant bit, on the left. Mux input 0 of every stage comes from the stage on its right (left shift, bits moving towards Q2); mux input 1 comes from the stage 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[188] for the exercise that asks for this register.

[188] courses/csse2010/2026-08-13-shift-registers.pdf

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[189]:

- 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**:

- 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.

[189] courses/csse2010/2026-08-17-counters.pdf

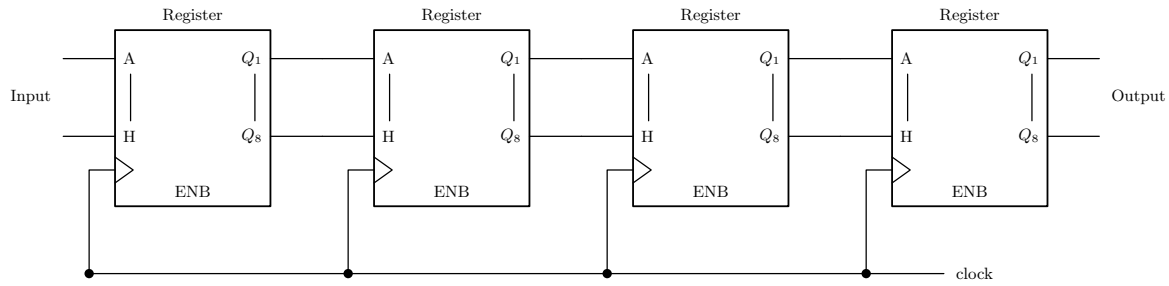


Figure 60: The 4-stage 8-bit queue. Each block is a whole 8-bit register rather than a single flip-flop: the vertical line between A and H, and between Q1 and Q8, is shorthand for the six pins in between. Functionally identical to the 1-bit chain — one stage's outputs are the next stage's inputs, everything on one clock — but each stage now holds a byte, so a byte presented at Input reaches Output four clock edges later.

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[190] and week4-lab-sequential-circuits-shift-registers[191].

Related

- flip-flops-and-latches[192] — the D flip-flop these are built from
- sequential-circuits[193] — state, present/next state, synchronous clocking
- combinational-logic-blocks[194] — the multiplexer used for load/direction selection
- [counters] — the other classic thing built from a row of D flip-flops
- serial-communication[195] · avr-usart[196] — a shift register at each end of a serial link, turning bytes into a stream of bits and back
- avr-spi[197] — two 8-bit shift registers joined in a ring, so that a byte goes each way at once

[190] courses/csse2010/2026-08-13-shift-registers.pdf
[191] courses/csse2010/week4-lab-sequential-circuits-shift-registers.pdf
[192] courses/csse2010/flip-flops-and-latches.pdf
[193] courses/csse2010/sequential-circuits.pdf
[194] courses/csse2010/combinational-logic-blocks.pdf
[195] courses/csse2010/serial-communication.pdf
[196] courses/csse2010/avr-usart.pdf
[197] courses/csse2010/avr-spi.pdf

- device-pinouts[198] — 74HCT175 quad D flip-flop and 74HCT157 quad 2-input multiplexer pinouts
- csse2010-diagram-notation[199] — the drawing conventions the figures above are checked against

State Machines

A **state machine** — equivalently a **finite state machine (FSM)** — is another name for a sequential circuit, viewed in terms of the states it can be in rather than the gates and flip-flops that implement it. See sequential-circuits[200] for the circuit-level view and flip-flops-and-latches[201] for the storage elements the state lives in.

What a state machine is

A state machine has:

- a **finite number of possible states** — N flip-flops give at most 2^N of them;
- **only one current state** at any moment;
- the ability to **transition** to other states, based on the inputs and the current state.

The state is exactly the value held in the flip-flops (see sequential-circuits[202]), so the flip-flop count bounds the state count. It does not fix it: how many flip-flops a given number of states actually costs is an encoding choice, made at the state-encoding step below.

Naming the two blocks of logic

The standard sequential-circuit structure has one combinational block computing two different things, and it is worth naming them separately:

$$\text{OUT} = g(\text{IN}, Q) \qquad\qquad D = f(\text{IN}, Q)$$

- Q is the **present state** (PS) — what the flip-flops hold right now.
- D is the **next state** (NS) — what they will hold after the next clock edge.
- g is the **output logic**; f is the **transition logic** (also called next-state logic).

Designing an FSM comes down to finding Boolean expressions for f and g.

[198] courses/csse2010/device-pinouts.pdf
[199] courses/csse2010/csse2010-diagram-notation.pdf
[200] courses/csse2010/sequential-circuits.pdf
[201] courses/csse2010/flip-flops-and-latches.pdf
[202] courses/csse2010/sequential-circuits.pdf

Defining the states from the problem

States are not handed to you — they come from the problem, and the useful question is *what does the circuit need to remember?*

Vending machine. A machine accepts 5c and 10c coins and dispenses a candy when it has received 15 cents in total. What are the states?

What the machine must remember is the running total inserted so far:

Total inserted	0	5	10	15	20	50
Dispense candy?	no	no	no	**yes**	**yes**, plus change	—

So the states are the totals 0, 5 and 10, with 15 and 20 being the ones that dispense (20 arises from inserting 10c on top of 10c, and needs change). A total of 50 is not a state at all: with only 5c and 10c coins the machine dispenses and resets long before it could get there, so it is unreachable.

Moore and Mealy machines

There are two types, differing only in what the output logic is allowed to see:

- **Mealy machine** — outputs depend on the current state **and the inputs**.
- **Moore machine** — outputs depend **only** on the current state (the flip-flop values). Consequently, outputs can only change when the state changes.

A Moore machine is a special case of a Mealy machine — the one where the output logic simply ignores its input connections. **CSSE2010 sticks to Moore machines.**

State diagram notation

A **state diagram** draws the states as circles and the transitions as labelled arrows.

States are drawn as a circle split in two: the **state name** on top, the **output value(s)** underneath. So S_0 over 101 is a state called S_0 whose outputs are 101; the name is arbitrary and need not be S_n (a state called `Alex` over 01 is just as legal).

Transitions are arrows labelled with a **logic expression** over the inputs — if the expression is true, that transition is taken. Examples of labels: Z, $\overline{\text{RESET}}$, $A + B \cdot \overline{C}$.

Two conventions worth knowing:

- **No label means "true"** — the transition is always taken on the next clock edge. This is what the arrows in a plain counter's diagram look like.

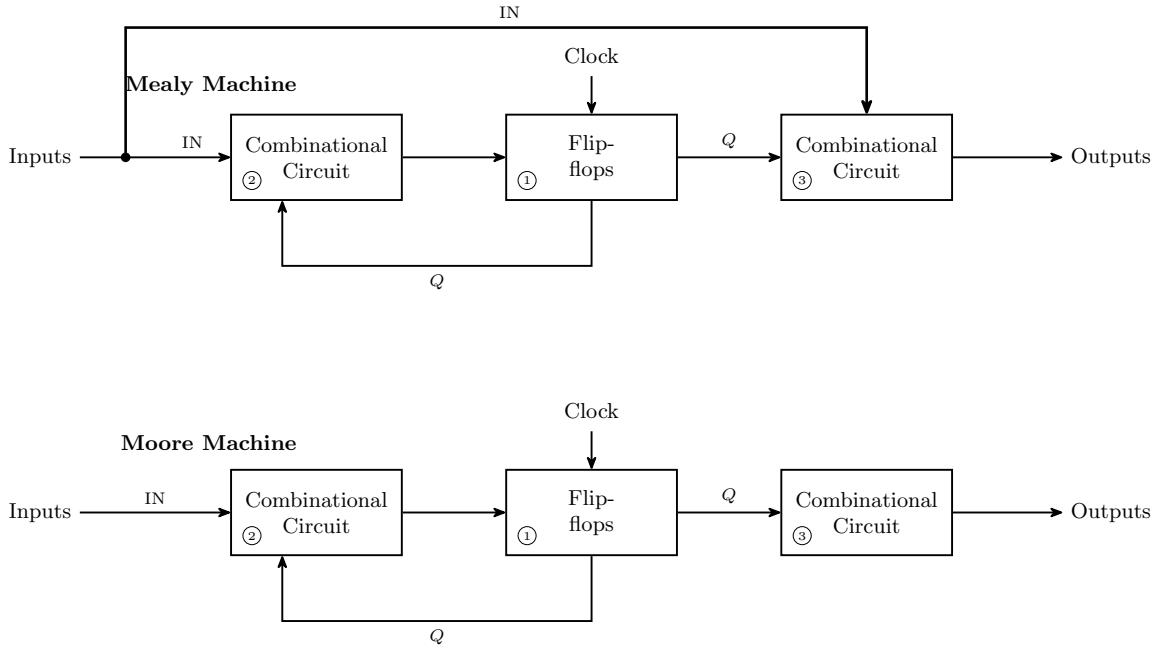


Figure 61: Mealy (top) and Moore (bottom). is the state, held in the flip-flops; is the transition logic; is the output logic. The only structural difference is the branch drawn in bold: in a Mealy machine the inputs reach the output logic directly, so the outputs are a function of IN and Q; in a Moore machine the output logic sees Q alone.

- **With only one input**, the label may be written as just `0` or `1` instead of a Boolean expression. This is a special case of the general form; 1 means the same as Z and 0 the same as $\overline{Z}$.

Completeness

> Each possible combination of inputs should be addressed **exactly once** for each state.

That is, the transition arrows leaving each state must encompass all possibilities, without overlap and without gaps. With N inputs there are 2^N input combinations, so **every state needs 2^N transitions accounted for** — though several combinations leading to the same next state can be merged onto one arrow with an OR'd label.

This is a checkable property, and checking it is the cheapest way to catch a half-finished design: if the labels on the arrows out of a state do not OR together to 1, some input combination has no defined behaviour; if any two of them can be true at once, the machine is ambiguous.

A counter is a state machine with no inputs

The design method for counters — write the sequence as a present-state/next-state table, read each D column off as a Boolean function of the Q's, draw the gates — is steps 3, 5 and 6 of the pipeline below, applied to the degenerate case where the machine has **no external inputs at all**. A counter may or may not have external inputs.

Three things collapse when IN is empty:

- **Transition logic.** $D = f(\text{IN}, Q)$ becomes $D = f(Q)$, so the state table needs no input column and the excitation equations are functions of the Q's alone.
- **Output logic.** A counter's output *is* its state, so g is the identity and there is no second block of gates. That also makes every counter trivially a Moore machine.
- **Completeness.** With $N = 0$ inputs there are $2^0 = 1$ input combinations, so the completeness rule reduces to *exactly one arrow out of each state* — and by the "no label means true" convention, that arrow is unlabelled.

The state encoding step also disappears, or rather it is made for you: the states of a counter are already bit patterns, and the sequence you were given *is* the encoding.

Figure 62 read as a state table, with the unused row carrying the don't-cares:

Current state	Next state	Q_1	Q_0
S_0	S_1	0	0
S_1	S_2	1	0

Current state	Next state	Q_1	Q_0
S_2	S_0	0	1
S_3	*(don't care)*	1	1

The unused state is where the FSM framing pays for itself. A don't-care is a **free choice of transition**, not an absence of one: the circuit will do *something* in state S_3, and the only question is whether the arrow you have implicitly drawn leads back into the cycle or loops onto itself. An arrow from S_3 to itself is a **trap state** — the machine leaves the intended cycle and never returns. There are two candidate simplifications for D_1. The tidier one, $D_1 = \overline{Q_1 \oplus Q_0}$, puts a self-loop on S_3 and is rejected. The other sends $11 \to 01$, back into the cycle, and is the robust design. See [counters] and 2026-08-17-counters[203] for that working.

State tables

The same information as a state diagram, in tabular form. Two layouts, both in use.

One-dimensional — one row for **every combination of state and inputs**:

Current State	Input U	Next State	Q_1	Q_0
S_0	0	S_3	0	0
S_0	1	S_1	0	0
S_1	0	S_0	0	1
S_1	1	S_2	0	1
S_2	0	S_1	1	0
S_2	1	S_3	1	0
S_3	0	S_2	1	1
S_3	1	S_0	1	1

Two-dimensional — one row **per state**, with a next-state column for **every combination of inputs**:

Current State	Next State ($\overline{U}$)	Next State (U)	Q_1	Q_0
S_0	S_3	S_1	0	0
S_1	S_0	S_2	0	1
S_2	S_1	S_3	1	0
S_3	S_2	S_0	1	1

[203] courses/csse2010/2026-08-17-counters.pdf

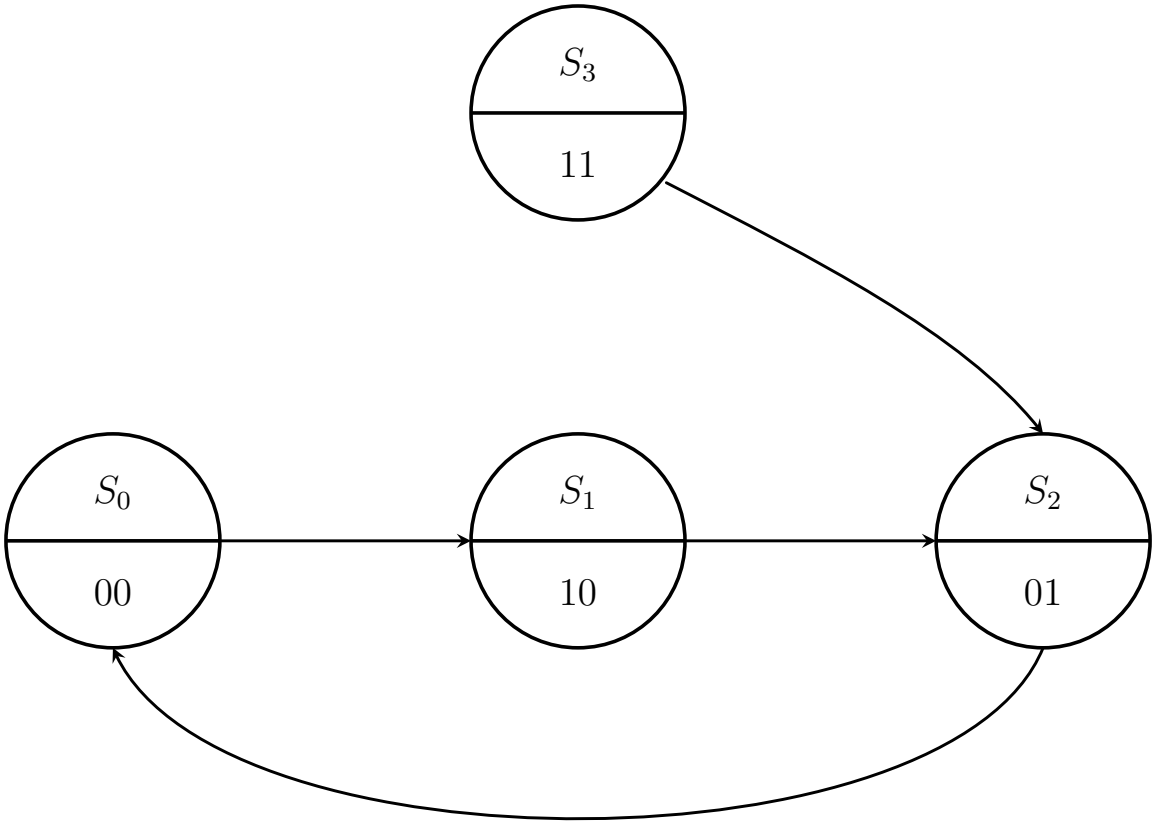


Figure 62: The 2-bit counter for $00 \rightarrow 10 \rightarrow 01 \rightarrow 00$ drawn as a state diagram. Every arrow is unlabelled because the machine has no inputs, so each transition is taken unconditionally on the next clock edge; each state's output half carries the count value, since a counter's output is its state. S_3 is the **unused** state: three states cannot fill a two-flip-flop machine, so its next state is a don't-care, and choosing $11 \rightarrow 01$ for it is what makes the design self-correcting.

Both describe the same machine (a 2-bit up/down counter, $U = 1$ counts up). The 2D form is more compact and makes completeness visually obvious — every input combination is a column, so a missing case is a missing column. The 1D form grows as $2^N\times$ (number of states) but maps directly onto a truth table, which is what the next design step needs. In a **Moore** machine the output columns depend only on the current state, which is why they can sit in a per-state row at all.

State encoding

Each state must be **encoded into flip-flop values** before the design can become a circuit. Two choices to make:

- the **number of flip-flops**, and
- the **bit patterns** representing each state.

Ideally, choose the encoding that makes the combinational logic simple — for both the output logic and the next-state logic. Different encodings of the same machine give genuinely different gate counts.

Three common schemes, on a four-state machine:

State	Binary	Gray	One-hot
S_0	00	00	0001
S_1	01	01	0010
S_2	10	11	0100
S_3	11	10	1000

- **Binary** — the obvious count; $\lceil \log_2 n \rceil$ flip-flops for n states.
- **Gray** — same width, but consecutive states differ in exactly one bit (see binary-number-representations[204]).
- **One-hot** — **one flip-flop per state**, with **only one flip-flop holding 1 at any time**. Far more flip-flops, but the next-state and output logic often collapses to almost nothing, since "am I in state S_k" is just a single flip-flop's output rather than a decoded combination.

Steps involved in FSM design

1. Defining state based on the problem definition.
2. Drawing the state diagram.
3. Completing the state table (1D or 2D).

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

4. State encoding, and re-doing the state table with encoded states.
5. Obtaining the Boolean expressions for the next-state logic (i.e. the D's) and for the outputs.
6. Drawing the logic diagram.
7. Simulating in Logisim to verify functionality.
8. Building and testing the circuit in hardware.

Steps 1–6 are the pen-and-paper design; steps 7–8 are what the learning labs do with it. Steps 5 and 6 are the same techniques used to build a counter by hand — see [counters] and circuit-schematics[205].

See also

- sequential-circuits[206] · flip-flops-and-latches[207] · [counters] · shift-registers[208]
- 2026-08-17-counters[209] — Lecture 7, where this design method is first used (on counters)
- 2026-08-20-state-machines[210] — Lecture 8
- week9-lab-avr-interrupts[211] — Lab 14: a stopwatch controller designed as a Moore machine and written in C

[205] courses/csse2010/circuit-schematics.pdf
[206] courses/csse2010/sequential-circuits.pdf
[207] courses/csse2010/flip-flops-and-latches.pdf
[208] courses/csse2010/shift-registers.pdf
[209] courses/csse2010/2026-08-17-counters.pdf
[210] courses/csse2010/2026-08-20-state-machines.pdf
[211] courses/csse2010/week9-lab-avr-interrupts.pdf