# Combinational Logic Blocks

## Table of contents


What makes a circuit combinational . . . . . 1
Binary subtraction . . . . . 2
Multiplexer (mux) . . . . . 5
Decoder . . . . . 8
See also . . . . . 12


Reference note for the standard combinational building blocks: the adder-subtractor, the multiplexer, and the decoder. See logic-gates[1] for the gate symbols/truth tables and boolean-algebra[2] 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[3], 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).

[1] courses/csse2010/logic-gates.pdf
[2] courses/csse2010/boolean-algebra.pdf
[3] courses/csse2010/sequential-circuits.pdf

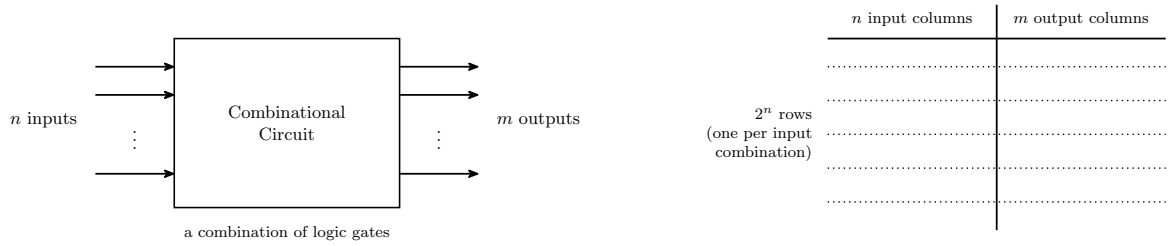



Figure 1: The general shape of a combinational circuit and the truth table it is always equivalent to. The row count $2^n$ depends only on the number of inputs, and the output column count $m$ only on the number of outputs — so a block with six inputs needs 64 rows however few gates are inside it.

## 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[4]).

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 $M$,

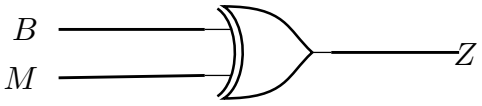
$$Z = \bar{B} \text{ when } M = 1, \qquad Z = B \text{ when } M = 0$$

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**:

$$Z = B \oplus M$$

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

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




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

Figure 2: 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[5]) plus four XOR gates:

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

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

$$C_0 = M$$

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

The cost of subtraction is therefore **four XOR gates and one wire** on top of an adder — a 74HCT86 (quad 2-input XOR) plus a 74HCT283 (4-bit adder), see device-pinouts[6].

[5] courses/csse2010/2026-08-04-binary-arithmetic.pdf
[6] courses/csse2010/device-pinouts.pdf

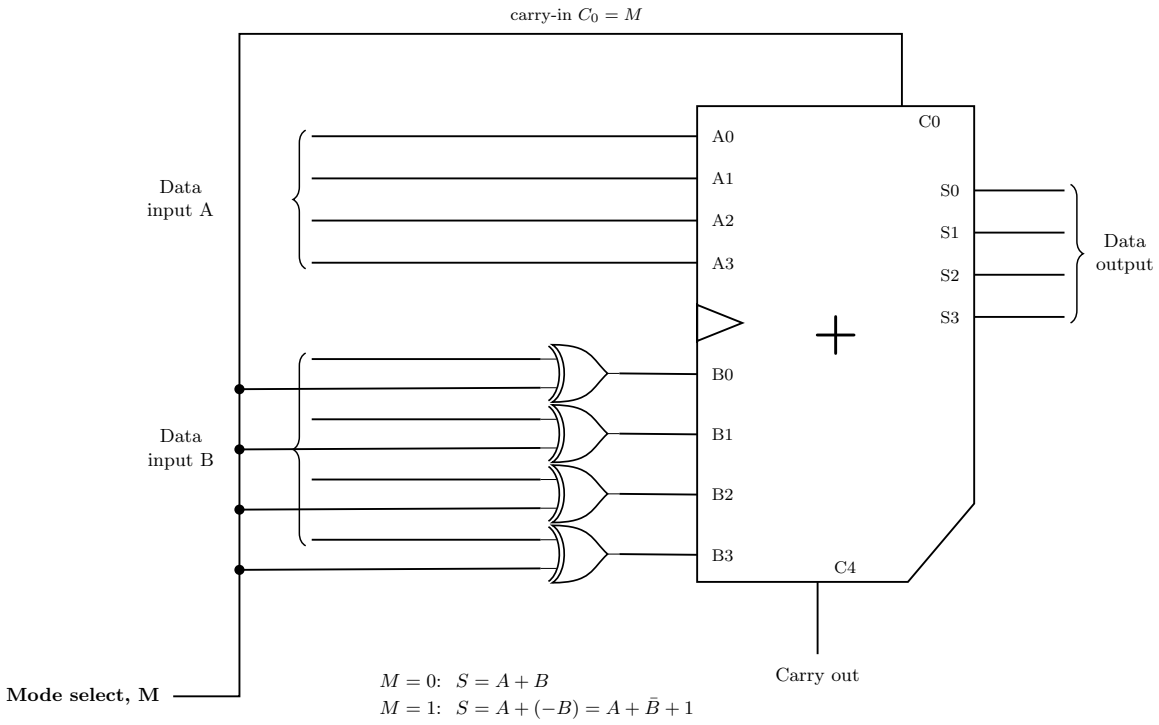



Figure 3: The 4-bit adder-subtractor. One mode line does two jobs: it conditions every bit of B through an XOR, and it is the adder's carry-in. With M = 0 the XORs pass B through and the carry-in is 0, so the adder computes A + B; with M = 1 the XORs deliver B-bar and the carry-in is 1, so it computes A + B-bar + 1, which is A + (-B). No extra hardware is spent on the +1 at all.

## Multiplexer (mux)

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

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

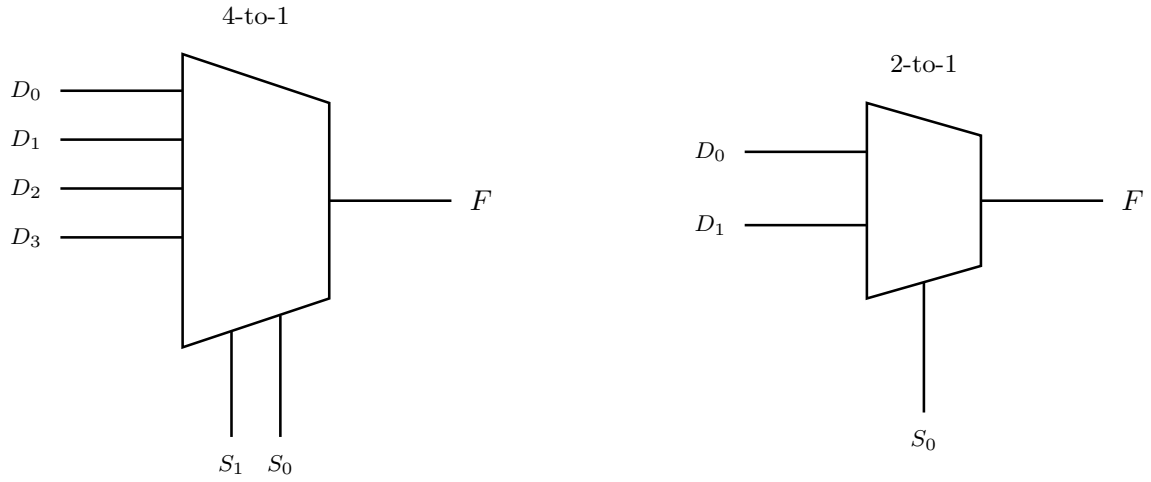



Figure 4: The multiplexer logic symbol at two sizes, 4-to-1 and 2-to-1. Data inputs enter the tall left edge, the single output leaves the short right edge, and the select inputs enter from underneath. The number of select lines is what fixes the size: n selects steer 2^n data inputs.

### 4-to-1 multiplexer

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

Function table:

| $S_1$ | $S_0$ | $F$ |
|---|---|---|
| 0 | 0 | $D_0$ |
| 0 | 1 | $D_1$ |
| 1 | 0 | $D_2$ |
| 1 | 1 | $D_3$ |

| $S_1$ | $S_0$ | $F$ |
|---|---|---|

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 $S_1S_0$.
- The four AND outputs feed a single **4-input OR gate** whose output is $F$.

That is exactly the sum-of-products form:

$$F = D_0\bar{S_1}\bar{S_0} + D_1\bar{S_1}S_0 + D_2S_1\bar{S_0} + D_3S_1S_0$$

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[7]) — building this from the available 2-input chips means cascading them.

#### 2-to-1 multiplexer

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

Function table:

| $S_0$ | $F$ |
|---|---|
| 0 | $D_0$ |
| 1 | $D_1$ |

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

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

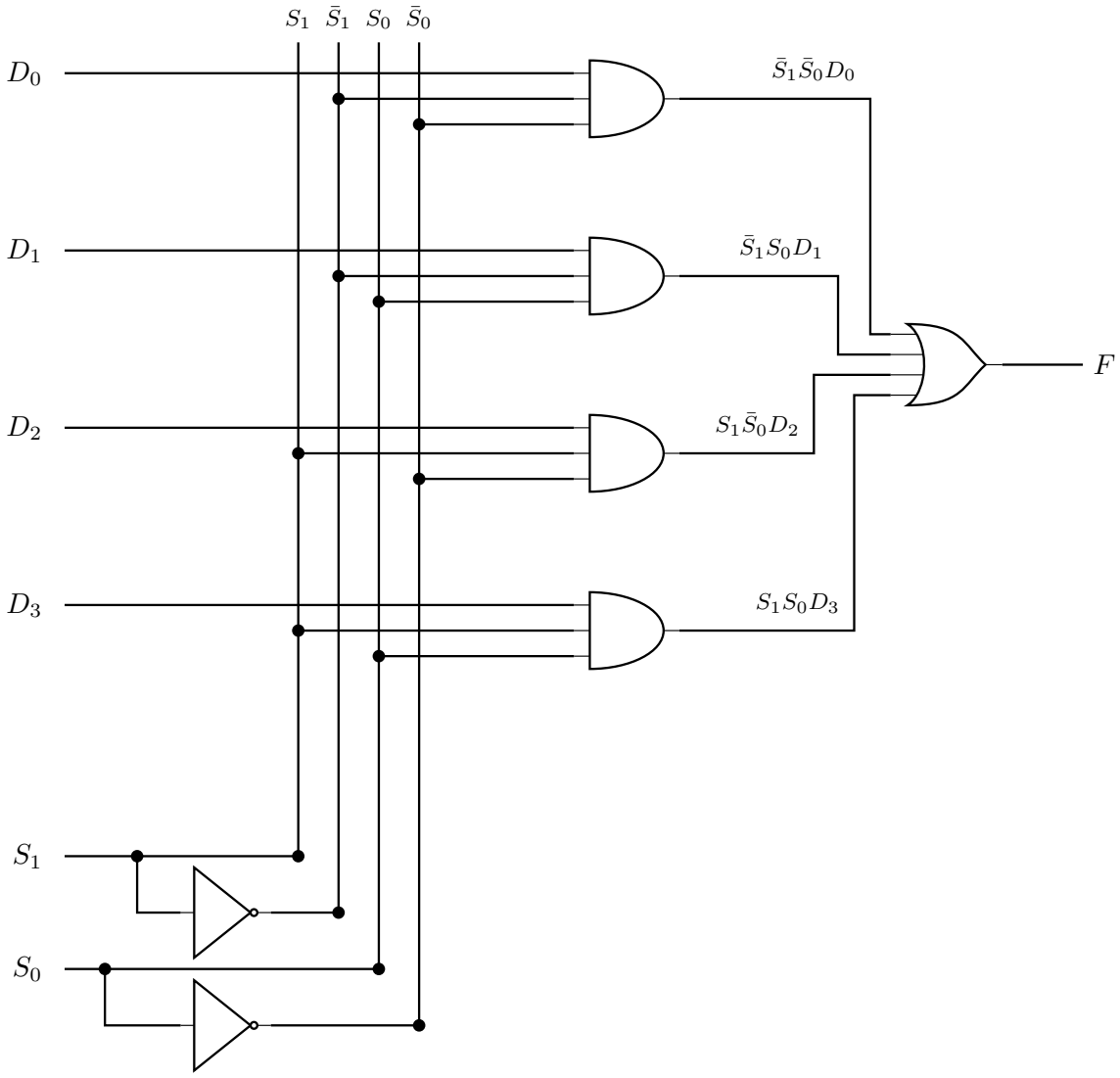



Figure 5: A 4-to-1 multiplexer built from gates. The two inverters generate the complemented selects; the four vertical rails carry S1, S1-bar, S0 and S0-bar past every AND gate; each AND gate is one data input gated by the one select combination that names it, and the OR gate collects them. Only the dots are connections — every other crossing is two wires passing.

| $S_0$ | $D_0$ | $D_1$ | $F$ |
|---|---|---|---|
| 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[8].

### 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[9].

## 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 |
| 0 | 1 | 1 | 0 | 0 | 0 | **1** | 0 | 0 | 0 | 0 |

[8] courses/csse2010/device-pinouts.pdf
[9] courses/csse2010/2026-08-06-combinational-logic.pdf

| A | B | C | $D_0$ | $D_1$ | $D_2$ | $D_3$ | $D_4$ | $D_5$ | $D_6$ | $D_7$ |
|---|---|---|---|---|---|---|---|---|---|---|
| 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.

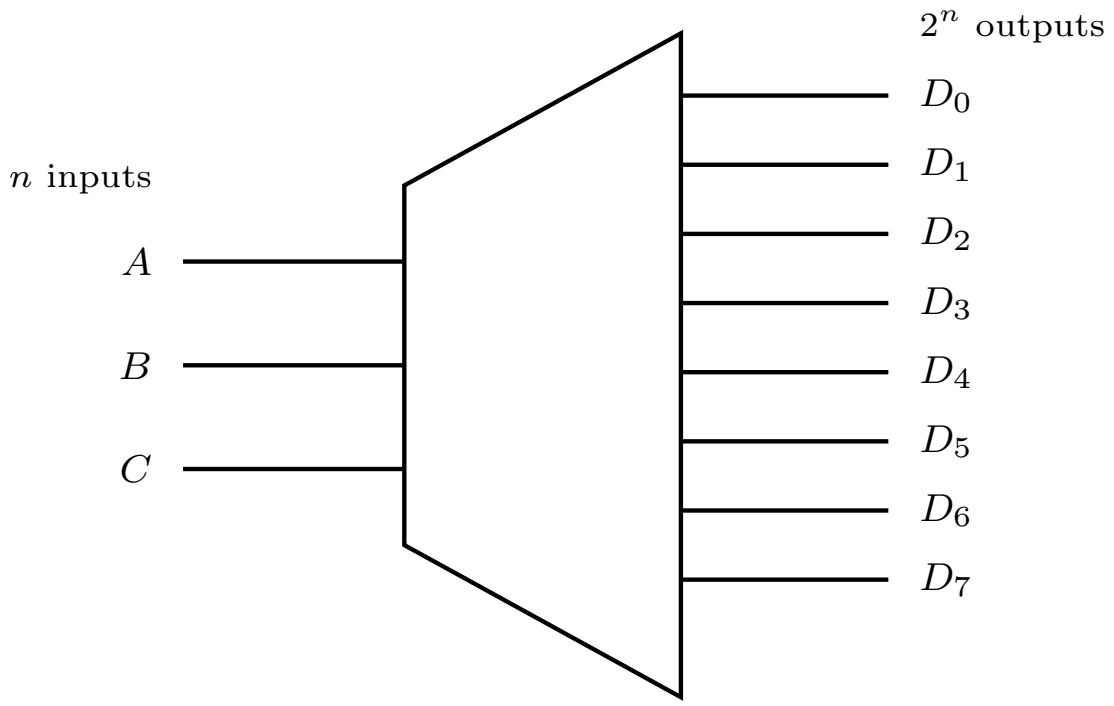



Figure 6: The decoder as a block: n inputs in, 2^n outputs out, and the shape widens accordingly. It is the mux's opposite — a mux funnels 2^n inputs down to one output under n selects, a decoder fans n inputs out to 2^n outputs.

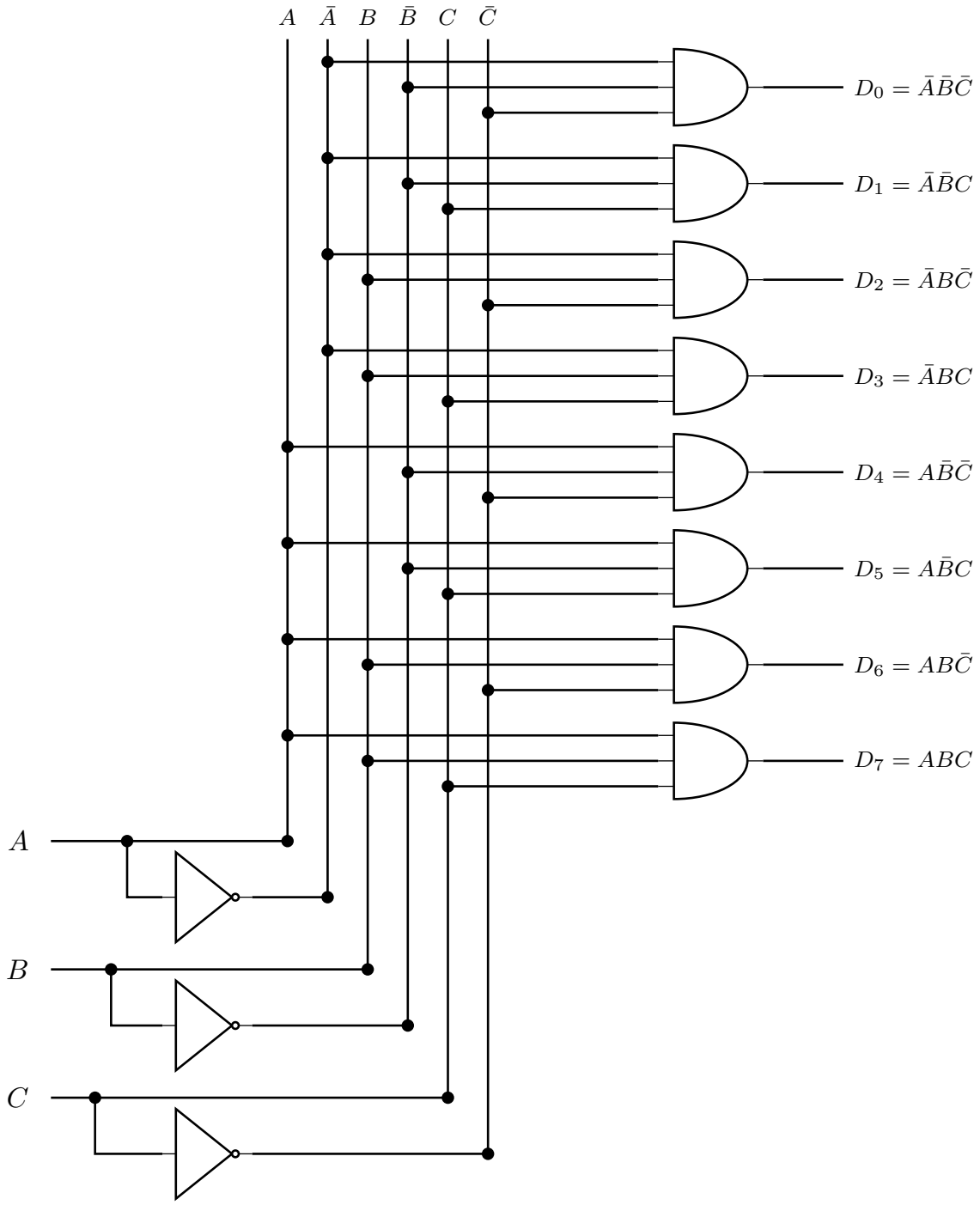



Figure 7: A 3-to-8 decoder built from gates. Three inverters generate the complements onto six vertical rails; each of the eight AND gates taps the one literal triple that matches its index, so it is the minterm for that input combination. Exactly one AND gate can be high at a time, which is the decoder's defining property.

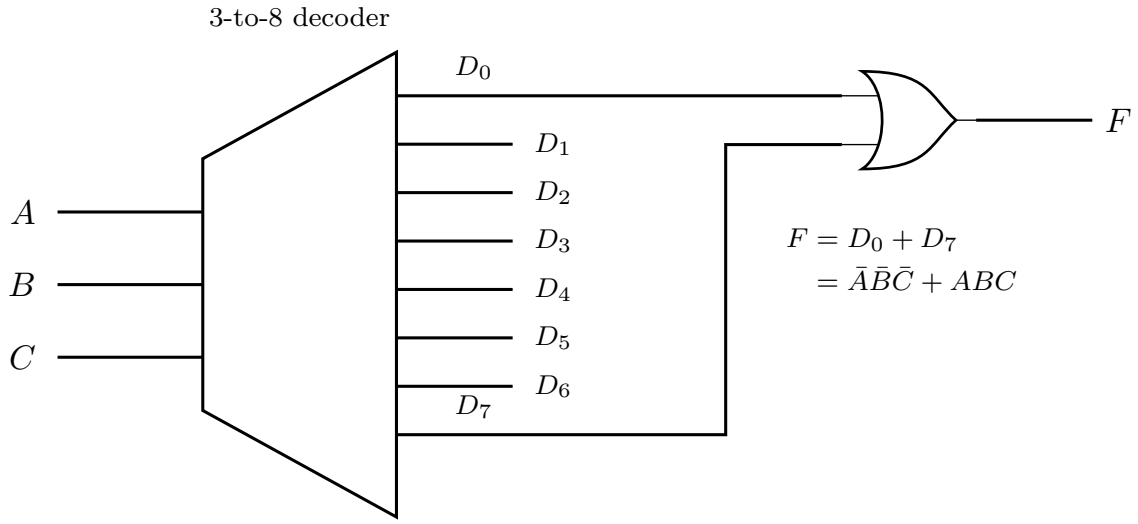



Figure 8: A function built on a decoder: F = A-bar B-bar C-bar + ABC is D0 + D7, so it costs a single OR gate on top of the decoder. Any function of A, B and C can be built this way — pick out the outputs for its true rows and OR them together.

## See also

- 2026-08-06-combinational-logic[10] — Lecture 4, where these blocks were introduced
- device-pinouts[11] — the off-the-shelf parts: 74HCT157 (mux), 74HCT283 (adder), 74HCT86 (XOR)
- arithmetic-logic-unit[12] — the ALU bit slice, which is a mux selecting between several function blocks
- binary-number-representations[13] — two's complement, on which the subtractor depends

[10] courses/csse2010/2026-08-06-combinational-logic.pdf
[11] courses/csse2010/device-pinouts.pdf
[12] courses/csse2010/arithmetic-logic-unit.pdf
[13] courses/csse2010/binary-number-representations.pdf