CSSE2010 — Week 2 Notes

Table of contents

Binary Arithmetic **1**
- Worked: -32 in 8-bit signed magnitude . . . 1
- Worked: -32 in 8-bit two's complement . . . 1
- Universal (complete) gates . . . 2
- Worked: a NOR-only implementation . . . 3
- Binary addition . . . 4
- Adding whole words . . . 4
- Overflow . . . 5
- Worked: adding two 6-bit two's complement numbers . . . 6
- Half-adder . . . 6
- Dealing with carry-in: the full adder . . . 7
- Ripple-carry adder . . . 7
- Admin and reminders . . . 9

Lecture 4 — Combinational Logic **10**
- Outline . . . 10
- Recap — binary adder . . . 10
- Binary subtractors . . . 11
- Combinational circuits in general . . . 12
- Multiplexers . . . 12
- Worked: 2-to-1 multiplexer truth table . . . 12
- Worked: choosing mux inputs to realise $X = \bar{S_0} + S_1$ . . . 13
- Decoders . . . 14
- Timing diagram representations . . . 14
- Worked: timing diagram for an XOR gate . . . 15
- Reminders . . . 16

Binary Arithmetic Lab **16**
- Worked: -16_{10} in 8-bit two's complement . . . 16
- Worked: `001110` + `110100` in 6-bit two's complement . . . 16

Worked: which operation overflows in 4-bit two's complement? . . . 17
Task 1 — 1-bit full adder . . . 17
Task 2 — 4-bit adder with the 74HCT283 . . . 18
Next . . . 22

Majority Function Circuit Design **22**
Question 1 — 3-input majority function . . . 22
See also . . . 24

Reference material **25**
Arithmetic Logic Unit . . . 25
Binary Number Representations . . . 32
Boolean Algebra . . . 40
Circuit Schematics . . . 43
Combinational Logic Blocks . . . 46
Sample Mid-semester Exam #2 . . . 57
Sample Mid-semester Exam #3 . . . 95
Sample Mid-semester Exam #4 . . . 138
Device Pinouts . . . 176
Fetch-Decode-Execute Cycle . . . 186
Logic Gates . . . 192
Timing Diagrams . . . 195

Binary Arithmetic

See binary-number-representations[1] for signed-format background (signed magnitude, two's complement) and logic-gates[2]/boolean-algebra[3] for the gate-level building blocks used below. The matching lab is week2-lab-binary-arithmetic[4].

Worked: -32 in 8-bit signed magnitude

Signed magnitude splits the word into a **sign bit** (the MSB, 1 = negative) and a **magnitude** in the remaining $N-1$ bits. Here $32 = 0100000$ in the seven magnitude bits, and the sign bit is 1:

$$\underbrace{1}_{\text{sign}}\ \underbrace{0100000}_{\text{magnitude}} = \texttt{10100000}$$

[1] courses/csse2010/binary-number-representations.pdf
[2] courses/csse2010/logic-gates.pdf
[3] courses/csse2010/boolean-algebra.pdf
[4] courses/csse2010/week2-lab-binary-arithmetic.pdf

Answer: `10100000`.

Worked: -32 in 8-bit two's complement

To negate a two's complement number: **invert every bit, then add 1**.

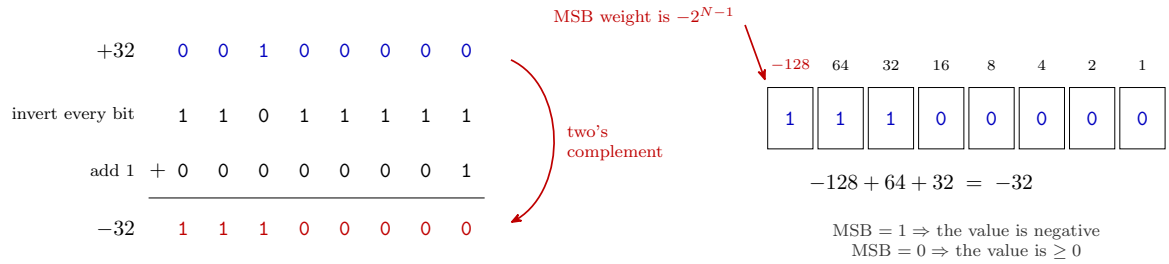


Figure 1: Negating +32 in 8 bits. Left: invert all eight bits of 00100000 and add one. Right: the answer checked against the place values — in two's complement the MSB is the only column with a *negative* weight, -2^{N-1}, which is what makes 11100000 come to -32 rather than 224.

Answer: `11100000`.

Two rules to carry forward:

- The MSB alone tells you the sign. $x \geq 0$ looks like $0xx \ldots x$; $x < 0$ looks like $1xx \ldots x$.
- The MSB is not just a flag — it is a **place value of** -2^{N-1}. Every other column keeps its usual positive weight, so $11100000 = -128 + 64 + 32 = -32$.

The number -32 is `10100000` in signed magnitude and `11100000` in two's complement. **The same number has different bit patterns in different formats**, so a bit pattern on its own means nothing until you know which format it is in.

Universal (complete) gates

Everything below is built out of gates. **All circuits can be constructed from NAND gates only, or from NOR gates only** — these are called **complete** (or universal) gates. The reason is practical: NAND and NOR are easier to build from transistors than AND and OR are.

The two moves that generate the whole table:

- **Tie the inputs together to get an inverter.** $\overline{A \cdot A} = \overline{A}$ for a NAND, and $\overline{A + A} = \overline{A}$ for a NOR.

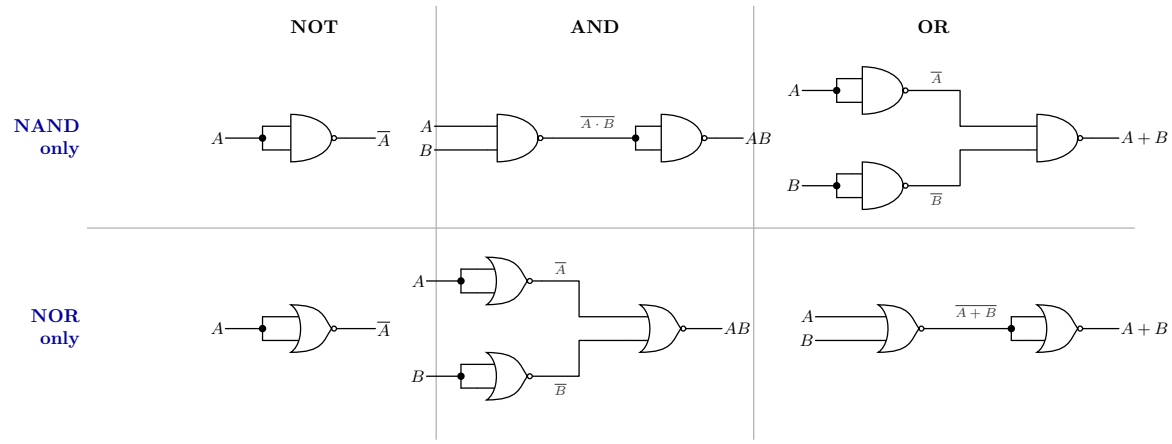


Figure 2: NOT, AND and OR built two ways: from NAND gates only (top row) and from NOR gates only (bottom row). The two building tricks are the ones to remember — a NAND or NOR with its inputs tied together is an inverter, and De Morgan turns an inverted-input NAND into an OR (and an inverted-input NOR into an AND).

- **Invert the inputs and apply De Morgan.** For the NAND-OR cell:

$$\overline{\overline{A} \cdot \overline{B}} \;=\; \overline{\overline{A}} + \overline{\overline{B}} \;=\; A + B$$

  and the mirror image for the NOR-AND cell, $\overline{\overline{A} + \overline{B}} = A \cdot B$.

Everything else is one of these two composed with a plain NAND/NOR: AND from NAND is a NAND followed by an inverter, and OR from NOR is a NOR followed by an inverter.

Worked: a NOR-only implementation

Implement $AB + CD$ using NOR gates only.

Answer: **the right-hand circuit**. The quick way to read a circuit like this is to chunk it — mark off which gates form the two ANDs and which form the OR, rather than tracing signals gate by gate. The same circuit with the final inverting NOR missing produces $\overline{AB + CD}$. Other wrong answers use the wrong gate type or too few gates.

Binary addition

Addition is simple in binary. Adding a single **addend** bit A to a single **augend** bit B gives a **sum** bit S and a **carry**:

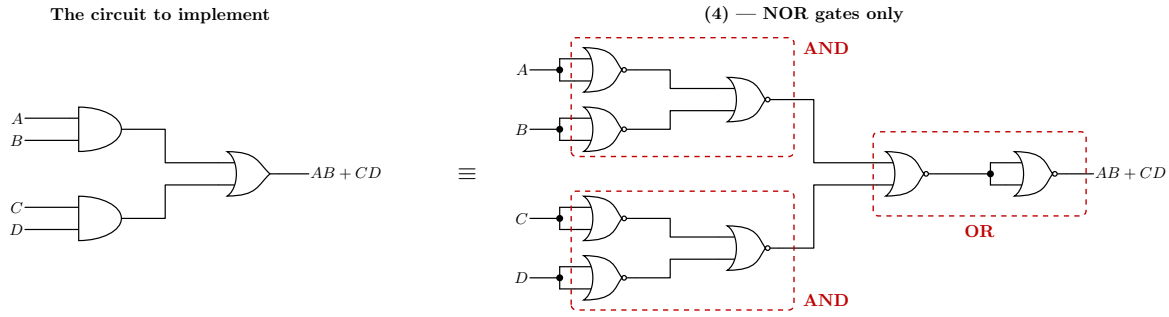


Figure 3: The circuit to implement (left) and its NOR-only version (right). Read it as three blocks: each AND becomes 'invert both inputs, then NOR', and the OR becomes 'NOR, then invert'. The tempting wrong answer is the same circuit with that last inverting NOR left off, which computes the complement of the right answer.

Addend	0	0	1	1
Augend	+0	+1	+0	+1
Sum	0	1	1	0
Carry	0	0	0	1

This table ignores carry-in — it is only correct for the least significant column. The full adder, below, handles carry-in.

Adding whole words

```
  Decimal   8-bit Unsigned      Decimal   8-bit 2's complement
  10        00001010            10        00001010
+ 243     + 11110011          +(-13)    + 11110011
--------- ------------        -------   --------------------
  253       11111101            -3        11111101
```

The two sums use **exactly the same bit pattern and exactly the same procedure** — only the reading of the operands and the result differs.

- **Format matters when you interpret the numbers**, not when you add them. Whatever the format, the bit-wise addition is the same, which is why one adder circuit serves both.
- **Two's complement**: you do not need to do anything with the carry-out from the MSB to get the correct result — just drop it. A 4-bit check:

```
   111          <- carries
   1111   (-1)
 + 1111   (-1)
--------
 1 1110          carry-out discarded; the 4-bit result 1110 = -2
```

- **One's complement**: you *do* have to take the carry-out from the MSB and add it back into the result (an "end-around carry") to get the right answer. That extra step is one of the practical drawbacks of one's complement compared to two's complement.

To check the two's complement result above, negate it: `11111101` → `00000010` → $+1$ → `00000011` $= 3$, so the pattern is -3.

Overflow

Overflow: there are not enough bits to represent the answer, so the result goes out of range and you get a wrong answer.

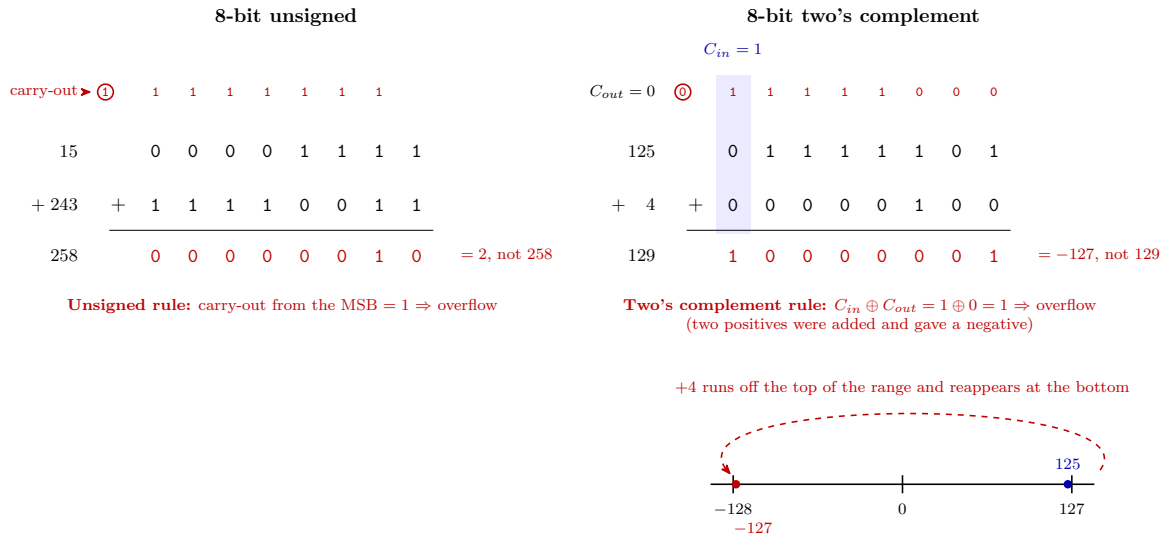


Figure 4: The two overflow cases worked with their carry rows exposed, because the carries are what the detection rules are stated in terms of. Left: unsigned, where a carry-out of the MSB alone means overflow. Right: two's complement, where the carry *into* the MSB and the carry *out of* it disagree — and, below, what that looks like on the number line: $125 + 4$ walks off the top of the range and wraps round to the bottom.

How overflow is **detected**:

- **Unsigned**: a carry-out from the MSB means overflow.

- **Two's complement**: overflow if the carry-*in* to the MSB and the carry-*out* from the MSB differ, i.e.

$$\text{Overflow} = C_{in} \oplus C_{out}$$

- Equivalently in two's complement: overflow happens when **two negatives added together give a positive**, or **two positives added together give a negative**. (Adding a positive to a negative can never overflow — the result is between the two operands.)

The two rules differ because in two's complement the carry-out is meaningless on its own and is simply discarded, as above; it only says something about overflow when compared against the carry-in.

Worked: adding two 6-bit two's complement numbers

$110101 + 001111$, with the answer constrained to 6 bits:

```
  11111     <- carries
  110101
+ 001111
---------
1 000100    the 7th bit is outside the 6-bit word, so drop it
```

Answer: `000100`.

Check in decimal: $110101 = -32 + 16 + 4 + 1 = -11$ and $001111 = 15$, so the answer should be $4 = 000100$. And there is no overflow: the carry into the MSB is 1 and the carry out is 1, so $C_{in} \oplus C_{out} = 0$.

Half-adder

A device that adds two bits has inputs A, B and outputs S (sum) and C (carry). Its truth table is the single-bit table from earlier:

A	B	C	S
0	0	0	0
0	1	0	1
1	0	0	1
1	1	1	0

$$S = A \oplus B \qquad C = AB$$

A device which adds 2 bits **with no carry-in** is called a **half-adder**. The sum column is exactly XOR and the carry column is exactly AND, so a half-adder is one XOR gate and one AND gate.

Dealing with carry-in: the full adder

A half-adder is not enough for a real column of an addition, because there may be a **carry-in from the previous stage**. A device with three inputs A, B, C_{in} and two outputs S, C_{out} is a **full adder**.

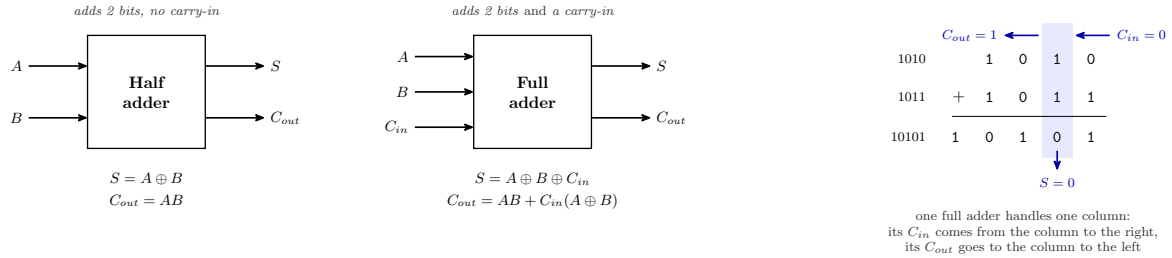


Figure 5: Half-adder versus full adder as blocks, and (right) why the third input exists: in a multi-bit addition each column is one full adder, taking its carry-in from the column on its right and passing its carry-out to the column on its left. Only the least significant column can get away with a half-adder.

The full adder's truth table:

A	B	C_{in}	C_{out}	Sum
0	0	0	0	0
0	0	1	0	1
0	1	0	0	1
0	1	1	1	0
1	0	0	0	1
1	0	1	1	0
1	1	0	1	0
1	1	1	1	1

$$S = A \oplus B \oplus C_{in}$$

$$C_{out} = AB + C_{in}(A \oplus B)$$

S is 1 whenever an odd number of the three inputs is 1, which is the three-input XOR. C_{out} is 1 when A and B both carry, *or* when exactly one of A, B is 1 and there was a carry-in to push the column over.

Those two expressions say the circuit is **two half-adders and an OR gate**: the first half-adder adds A and B; the second adds its sum to C_{in}; and the two half-adders' carry outputs are OR'd to give C_{out} (they can never both be 1, so an OR is enough).

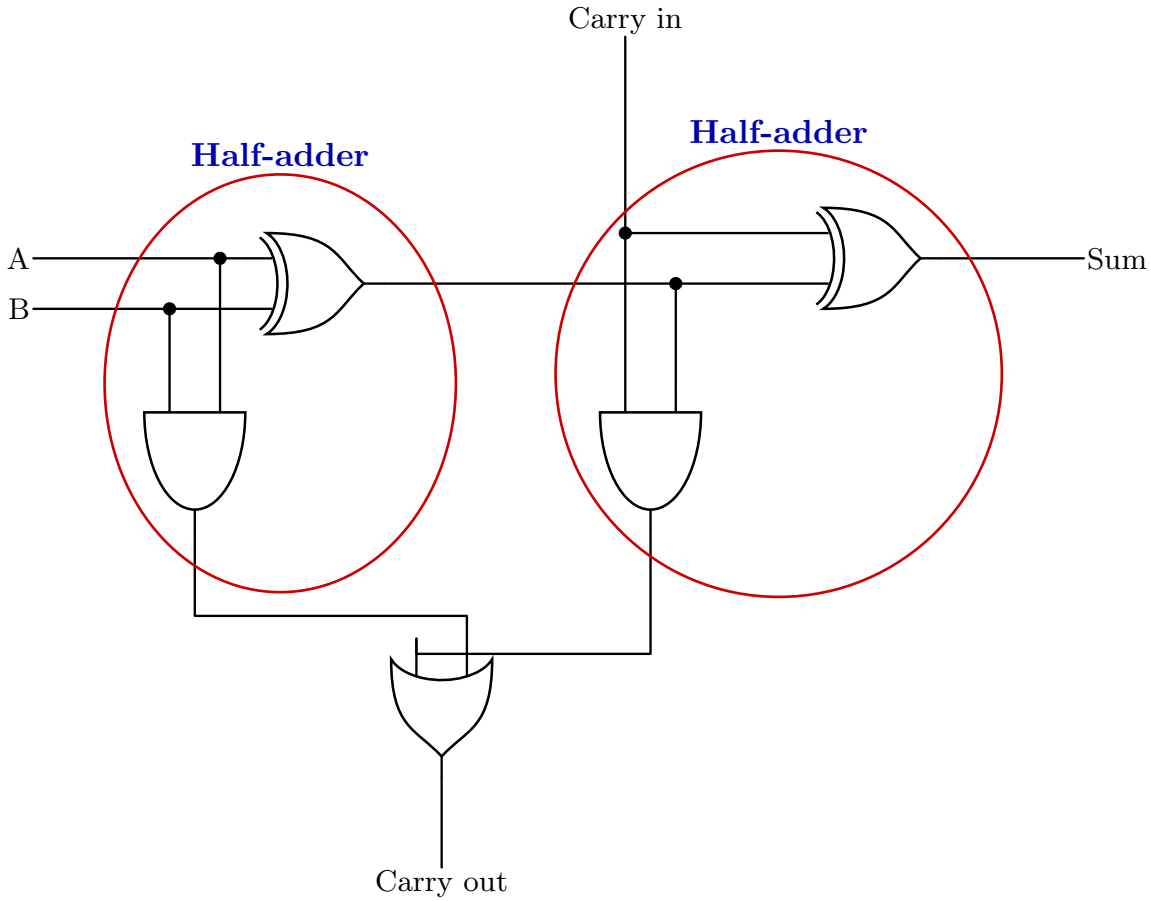


Figure 6: The full adder at gate level, with the two half-adders outlined. The left half-adder adds A and B; the right one adds that sum to the carry in; the OR at the bottom collects the two carries. At the crossing near the top right, the carry-in line passes over the first half-adder's sum line without a junction dot, so they are not connected.

Ripple-carry adder

Cascading full adders — each stage's C_{out} wired into the next stage's C_{in} — builds a multi-bit binary adder. Four of them give a 4-bit adder.

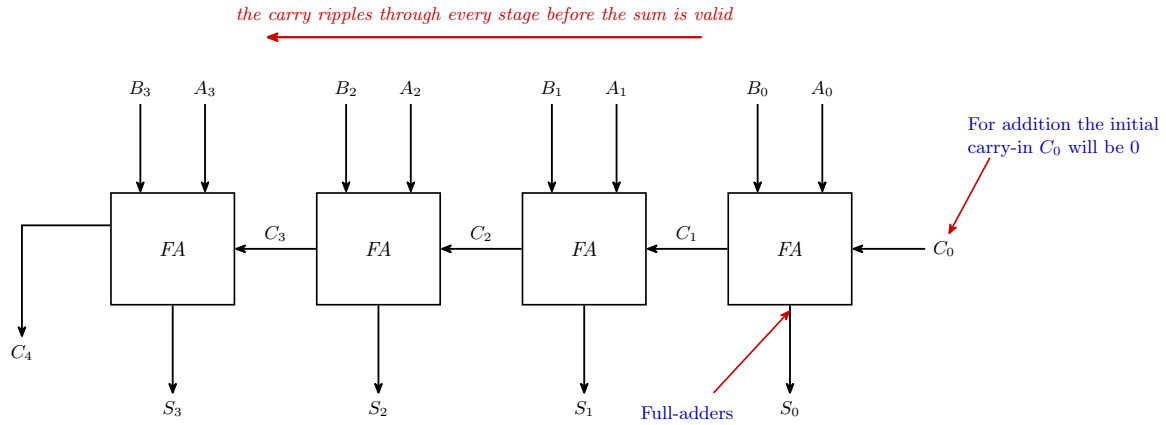


Figure 7: A 4-bit ripple-carry adder: one full adder per bit position, with the carry chained from the least significant stage (right) to the most significant (left). Since stage i cannot produce its final sum until stage $i-1$ has settled, the carry has to propagate through all four stages before any of the outputs can be trusted — which is what makes this structure simple but slow.

- **For plain addition the initial carry-in C_0 is 0.** (It is there because it is useful for other things — chaining adders, and subtraction, which comes later.)
- This structure is called a **ripple-carry adder**, because the carry has to propagate ("ripple") through every stage before the final sum is valid.
- C_4 and C_3 are exactly the C_{out} and C_{in} of the MSB stage, so the overflow test above is available straight off this diagram.

See combinational-logic-blocks[5] for the adder as a standard block, and arithmetic-logic-unit[6] for the same ripple structure reused inside an ALU.

Admin and reminders

- Contact `eait.mytimetable@uq.edu.au` if not yet signed up for a lab, or unable to attend the labs you are signed on to.
- Two lab sessions per week from this week; refer to the week-by-week teaching outline on Blackboard.

[5] courses/csse2010/combinational-logic-blocks.pdf
[6] courses/csse2010/arithmetic-logic-unit.pdf

- Weekly exercises and quizzes (not assessed) are on Blackboard.
- Week 2 labs: P2 sessions (Mon-Tue) run Lab 2 (Logic Gates); P1 sessions (Thu-Fri) run Lab 3 (Binary Arithmetic) — see week2-lab-binary-arithmetic[7].
- Complete the week 1 exercises folder and ask if unclear.
- Watch the summary video on Signed Binary Formats.

Lecture 4 — Combinational Logic

See combinational-logic-blocks[8] for the reference definitions/tables of the adder-subtractor, multiplexer and decoder introduced here, and timing-diagrams[9] for the timing-diagram representation and real-gate timing parameters.

Outline

- Admin
- Binary subtractors
- More combinational logic circuits
  - Multiplexers
  - Decoders
  - Timing diagram representations

Recap — binary adder

Picking up from 2026-08-04-binary-arithmetic[10]: full adders can be **cascaded** to make a multi-bit binary adder. For 4 bits, four full adders take $A_3B_3 \ldots A_0B_0$ and produce $S_3 \ldots S_0$ plus C_4, with each stage's carry-out C_{i+1} wired to the next stage's carry-in. For addition the initial carry-in C_0 is **0**. This arrangement is a **ripple-carry adder**.

Binary subtractors

$A - B$ is implemented as $A + (-B)$, with $-B$ the two's complement of B. Three things to keep straight:

- A and B are **multi-bit** quantities.
- The "+" means **addition**, not OR.

[7] courses/csse2010/week2-lab-binary-arithmetic.pdf
[8] courses/csse2010/combinational-logic-blocks.pdf
[9] courses/csse2010/timing-diagrams.pdf
[10] courses/csse2010/2026-08-04-binary-arithmetic.pdf

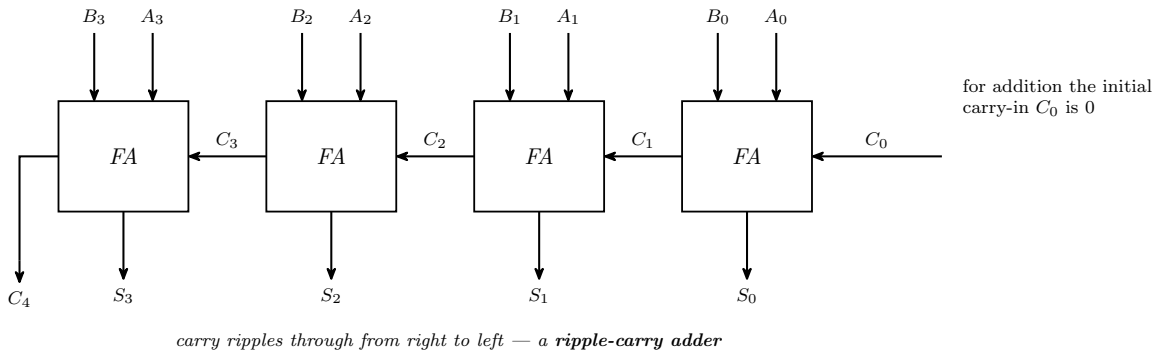


Figure 8: The 4-bit ripple-carry adder. Each full adder handles one bit position: it takes A and B for that position plus the carry coming in from the position to its right, and produces that position's sum bit plus a carry out to the left. Nothing but the carry crosses between stages.

- $-B$ is the two's complement of B: **flip the bits and add 1**, i.e. $-B = \bar{B} + 1$.

Take $5 - 3$, first as an ordinary subtraction and then as the addition the hardware actually performs.

The two pieces needed on top of a plain adder are a **controlled inverter** (an XOR gate per bit, with the mode select M as the second input) and the extra $+1$, which is supplied via the adder's carry-in. Full derivation and the adder-subtractor circuit are in combinational-logic-blocks[11].

Two questions settle the design:

- **Which gate flips a bit, but only sometimes?** The gate needs $Z = \bar{B}$ when $M = 1$ and $Z = B$ when $M = 0$. The three-column M, B, Z truth table gives a 2-input XOR, $Z = B \oplus M$. The general identities behind it are $X \oplus 0 = X$ and $X \oplus 1 = \bar{X}$ **for any signal** X.
- **What should carry-in be?** $C_0 = M$ — the same mode-select line that drives the XOR gates, so subtract mode automatically contributes the $+1$ that two's complement needs. $M = 0$ gives $S = A + B$, and $M = 1$ gives $S = A + (-B) = A + \bar{B} + 1$.

Combinational circuits in general

A combinational circuit is a combination of logic gates with n inputs and m outputs; each output is a function of the n inputs, the **output depends on current inputs only**, and it can be written as a truth table with n input columns, m output columns and 2^n rows. See combinational-logic-blocks[12].

[11] courses/csse2010/combinational-logic-blocks.pdf
[12] courses/csse2010/combinational-logic-blocks.pdf

$$-3 \longrightarrow 3 = 0011 \xrightarrow{\text{invert}} 1100 \xrightarrow{+1} 1101 = -3$$

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

what we want			**what the hardware does**	
5	0101		5	0101
$-$ 3	0011	$\xrightarrow{A-B}$ $= A + (-B)$	$+$ (-3)	1101
2	0010		2	0010

the carry out of the top bit is discarded — with four bits the answer $0010_2 = 2$ is read straight off, exactly as the subtraction gives

Figure 9: 5 - 3 worked as a subtraction and as the addition the hardware performs. Negating 3 takes two steps — invert 0011 to 1100, then add 1 to get 1101 — and 1101 is the four-bit two's complement pattern for -3. Adding it to 0101 gives the same 0010 the subtraction gives, so a machine that can only add can subtract.

Multiplexers

2^n data inputs, 1 output, n select inputs that steer one data input to the output — a **data selector**. The two sizes used here are the 4-to-1 mux (logic symbol, function table, and the AND-OR sum-of-products implementation with inverted select literals) and the 2-to-1 mux. Tables and circuit descriptions: combinational-logic-blocks[13]. Off-the-shelf part: the 74HCT157 quad 2-input mux, see device-pinouts[14].

The 4-to-1 function table has four rows, one per select combination, with F holding a *data input name* rather than a 0 or 1. That is deliberate: the full truth table over all six inputs would need $2^6 = 64$ rows.

Worked: 2-to-1 multiplexer truth table

A 2-to-1 multiplexer has data inputs D_0 and D_1, control input S_0, and output F, with function table $F = D_0$ when $S_0 = 0$ and $F = D_1$ when $S_0 = 1$. Which of three candidate truth tables (columns S_0, D_0, D_1, F) goes with this circuit?

Three inputs means $2^3 = 8$ rows, which all three candidates have. Reasoning from the function table, row by row in the order $S_0 D_0 D_1 = 000, 001, 010, 011, 100, 101, 110, 111$:

- The first four rows have $S_0 = 0$, so F must copy D_0 and ignore D_1: $F = 0, 0, 1, 1$.
- The last four rows have $S_0 = 1$, so F must copy D_1 and ignore D_0: $F = 0, 1, 0, 1$.

$$F = 0,\ 0,\ 1,\ 1,\ 0,\ 1,\ 0,\ 1$$

That is **option 2**. Option 1 has F following D_1 when $S_0 = 0$ and D_0 when $S_0 = 1$ — the two data inputs swapped. Option 3 is $F = 0, 0, 0, 0, 1, 1, 1, 1$, i.e. $F = S_0$: it ignores the data inputs completely.

Equivalently: $F = D_0 \bar{S}_0 + D_1 S_0$.

Worked: choosing mux inputs to realise $X = \bar{S}_0 + S_1$

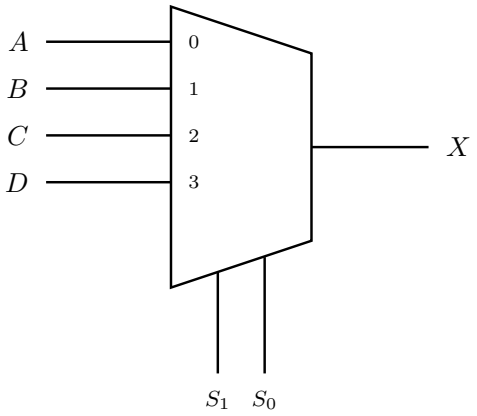
A 4-to-1 multiplexer has A on data input 0, B on 1, C on 2, D on 3, selects $S_1 S_0$, and output X. What must A, B, C, D be so that

$$X = \bar{S}_0 + S_1$$

[13] courses/csse2010/combinational-logic-blocks.pdf
[14] courses/csse2010/device-pinouts.pdf

Options: (1) A=0, B=1, C=0, D=0; (2) A=0, B=1, C=1, D=1; (3) A=1, B=0, C=1, D=1; (4) A=1, B=1, C=0, D=1.

Reasoning: the mux steers data input i to the output when S_1S_0 equals i in binary, so each data input must simply be **the value the desired function takes for that select combination**. Evaluate $\bar{S_0} + S_1$ for all four:



$X = \bar{S_0} + S_1$

S_1	S_0	$X = \bar{S_0} + S_1$	data input selected
0	0	$1 + 0 = 1$	$A = 1$
0	1	$0 + 0 = 0$	$B = 0$
1	0	$1 + 1 = 1$	$C = 1$
1	1	$0 + 1 = 1$	$D = 1$

Figure 10: The 4-to-1 mux and the function it has to realise. Each select combination picks one data input, so the value that input must carry is just the function evaluated at that combination — the four evaluations are 1+0, 0+0, 1+1, 0+1.

S_1	S_0	$\bar{S_0}$	$X = \bar{S_0} + S_1$	Selected input
0	0	1	1	A (input 0)
0	1	0	0	B (input 1)
1	0	1	1	C (input 2)
1	1	0	1	D (input 3)

So $A = 1, B = 0, C = 1, D = 1$ — **option 3**.

This is the general trick: tying a mux's data inputs to constants turns it into a lookup table for any function of the select variables.

Decoders

A decoder converts an n-bit input to a logic 1 on exactly one of 2^n outputs. The 3-to-8 decoder (inputs A, B, C; outputs $D_0 \ldots D_7$) is implemented with eight AND gates. Each row of its truth table corresponds to one minterm — $\bar{A}\bar{B}\bar{C}$ at the top down to ABC at the bottom.

Because a decoder puts every minterm on its own wire, an arbitrary function is just an OR of the right outputs. For example

$$F = \bar{A}\bar{B}\bar{C} + ABC = D_0 + D_7$$

one OR gate on two decoder outputs. Circuit and diagram: combinational-logic-blocks[15].

Timing diagram representations

The **timing diagram** is another logic representation — like a truth table but graphical. timing-diagrams[16] has the inverter as the worked example, the convention for drawing 2-input waveforms so all input combinations are covered, and the reality that gates aren't perfect (propagation delay, fall time, rise time).

Worked: timing diagram for an XOR gate

Given input waveforms where A is low for the first half of the diagram then high for the second half, and B toggles at twice that rate (low, high, low, high) — so the four intervals are $AB = 00, 01, 10, 11$ — which of three candidate output waveforms represents an **XOR** gate?

Reasoning from the XOR truth table ($X = 1$ iff exactly one input is 1):

Interval	A	B	$A \oplus B$
1	0	0	0
2	0	1	1
3	1	0	1
4	1	1	0

So the output must be **low, high, high, low** — a single pulse spanning the middle two intervals.

That is **option 2** — the only candidate with a pulse that goes high and comes back low again. Option 1 has a single rising edge and then stays high to the end, so it can never be XOR. Option 3 is the exact complement of option 2 (high, low, low, high), i.e. XNOR.

[15] courses/csse2010/combinational-logic-blocks.pdf
[16] courses/csse2010/timing-diagrams.pdf

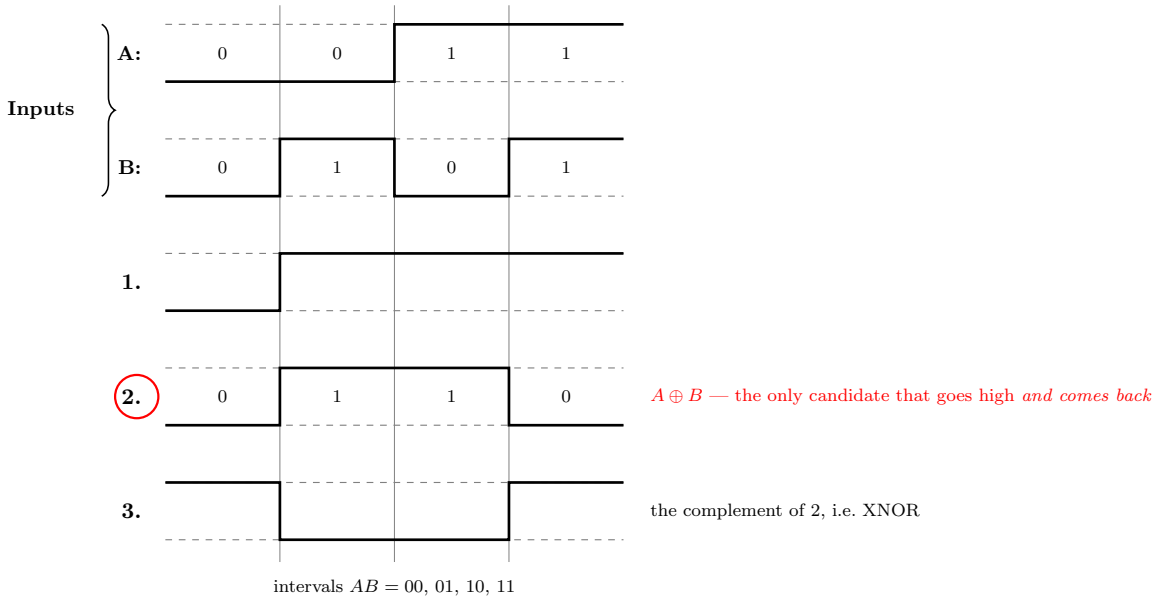


Figure 11: The inputs A and B and the three candidate outputs. Candidate 2 is the only one that goes high and comes back down, which is what a pulse over the middle two intervals requires. Candidate 1 rises once and stays high, so it can never be XOR; candidate 3 is candidate 2 inverted, i.e. XNOR.

Reminders

- **Lab 4 preparation task** — next week (week 3). It will be available on Blackboard under *Learning Resources*; it involves drawing some circuit schematic diagrams (see circuit-schematics[17]), and you must bring it to Lab 4 (P2 Mon-Tue sessions in week 3). Complete it before Lab 4. Worked answers: week3-lab-lab-4-pre-work[18].
- Supporting material for the labs is now on Blackboard.
- Attempt the posted exercises and quiz (not assessed).
- Any doubts: use course consultation, ask after the lecture, or use the Ed forum.

Binary Arithmetic Lab

Learning Lab 3 — binary arithmetic revision, then circuits that do arithmetic, simulated in Logisim and built with ICs. Pairs with 2026-08-04-binary-arithmetic[19].

Worked: -16_{10} in 8-bit two's complement

`11110000`.

$16 =$ `00010000`; invert to `11101111`; add 1 to get `11110000`. Check: $-128+64+32+16 = -16$.

The wrong answers to watch for: `11101111` is the *ones'* complement — the answer if you forget the +1. `10010000` and `10010001` are the signed-magnitude and signed-magnitude-off-by-one answers, and `00010000` is +16, i.e. the magnitude with the negation left out altogether.

Worked: `001110` + `110100` in 6-bit two's complement

`000010`, with no overflow.

$001110_2 = 14$ and $110100_2 = -32 + 16 + 4 = -12$, so the true answer is $2 =$ `000010`.

Overflow check, with the carry row written out:

	c_5	c_4	c_3	c_2	c_1	c_0	
carry in	1	1	1	0	0	0	← carry *into* each column
	0	0	1	1	1	0	$= 14$
	1	1	0	1	0	0	$= -12$

[17] courses/csse2010/circuit-schematics.pdf
[18] courses/csse2010/week3-lab-lab-4-pre-work.pdf
[19] courses/csse2010/2026-08-04-binary-arithmetic.pdf

	c_5	c_4	c_3	c_2	c_1	c_0	
sum	0	0	0	0	1	0	$= 2$

The carry *into* the MSB is 1 and the carry *out* is also 1, so $C_{in} \oplus C_{out} = 0$ — **no overflow**. The carry-out of 1 would signal overflow if these were *unsigned*. Getting the sum right is not enough: the answer has to say whether it overflows as well; see 2026-08-04-binary-arithmetic[20].

Worked: which operation overflows in 4-bit two's complement?

4-bit two's complement covers -8 to 7.

	Operation	Result	Overflow?
A	`0100 − 0101`	$4 - 5 = -1$	no
B	`1001 − 0001`	$-7 - 1 = -8$	no (just fits)
C	`0100 + 0101`	$4 + 5 = 9$	**yes** — $9 > 7$
D	`1001 + 0001`	$-7 + 1 = -6$	no

Answer: C. B is the near-miss: -8 is exactly representable, so it does not overflow.

Task 1 — 1-bit full adder

Turn the full-adder logic diagram (two XORs, two ANDs and an OR) into a **circuit schematic**, using **switches for the three inputs** (A, B, carry-in) and **LEDs for the two outputs** (sum, carry-out). Have it checked by a tutor, then simulate, build and test it — systematically, against the full-adder truth table. The gate-level diagram itself is in 2026-08-04-binary-arithmetic[21]; the equations it implements are

$$S = A \oplus B \oplus C_{in} \qquad C_{out} = AB + C_{in}(A \oplus B)$$

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

The two-chip challenge

The challenge: build the full adder using only two chips.

The literal reading of the diagram needs **three** chips — a `74HCT86` for the two XORs, a `74HCT08` for the two ANDs and a `74HCT32` for the OR. Going all-NAND does *not* help: a full adder in NAND alone takes nine 2-input NAND gates, and a `74HCT00` holds four, so that route needs three chips too.

The two-chip answer is to convert **only the carry expression** to NAND and leave the sum as XORs. De Morgan on C_{out}:

$$C_{out} = AB + C_{in}(A \oplus B) = \overline{\overline{AB} \cdot \overline{C_{in}(A \oplus B)}} = \text{NAND}(\text{NAND}(A, B),\ \text{NAND}(C_{in}, A \oplus B))$$

That is **two XOR gates and three NAND gates** — two of the four gates on a `74HCT86` and three of the four on a `74HCT00`. Two chips, with one spare gate in each.

The stages of completion, in order: schematic done / simulated / partially working / fully working with 3 ICs / fully working with **2 ICs**. The two-chip build is the top rung, not an optional extra. (Completion is self-reported, not marked.)

See combinational-logic-blocks[22] for the adder structure and circuit-schematics[23] for the drawing rules.

Task 2 — 4-bit adder with the `74HCT283`

The `74HCT283` is a **single-chip 4-bit adder**. Draw a schematic that adds two 4-bit numbers set on switches, showing the 4-bit result and the carry-out on LEDs, with the **carry-in on a push button**.

- Use the *Device Pinouts and Symbols* symbol: a trapezoid with A_0–A_3 on the upper left (pins 5, 3, 14, 12), B_0–B_3 on the lower left (pins 6, 2, 15, 11), C_0 entering the top (pin 7), S_0–S_3 on the right (pins 4, 1, 13, 10) and C_4 leaving the bottom (pin 9). See device-pinouts[24].
- **Don't forget the power connections.** It's a **16-pin** chip: Vcc is pin 16 and GND pin 8, *not* 14/7 as on every gate chip you have used so far.
- Have the schematic checked, then wire it up, test it and simulate it. Try both unsigned and two's complement test cases and check the results are what you expect. (They are the same four bits either way; only your *reading* of them changes, and only C_4 versus the overflow rule tells you whether that reading is valid.)

[22] courses/csse2010/combinational-logic-blocks.pdf
[23] courses/csse2010/circuit-schematics.pdf
[24] courses/csse2010/device-pinouts.pdf

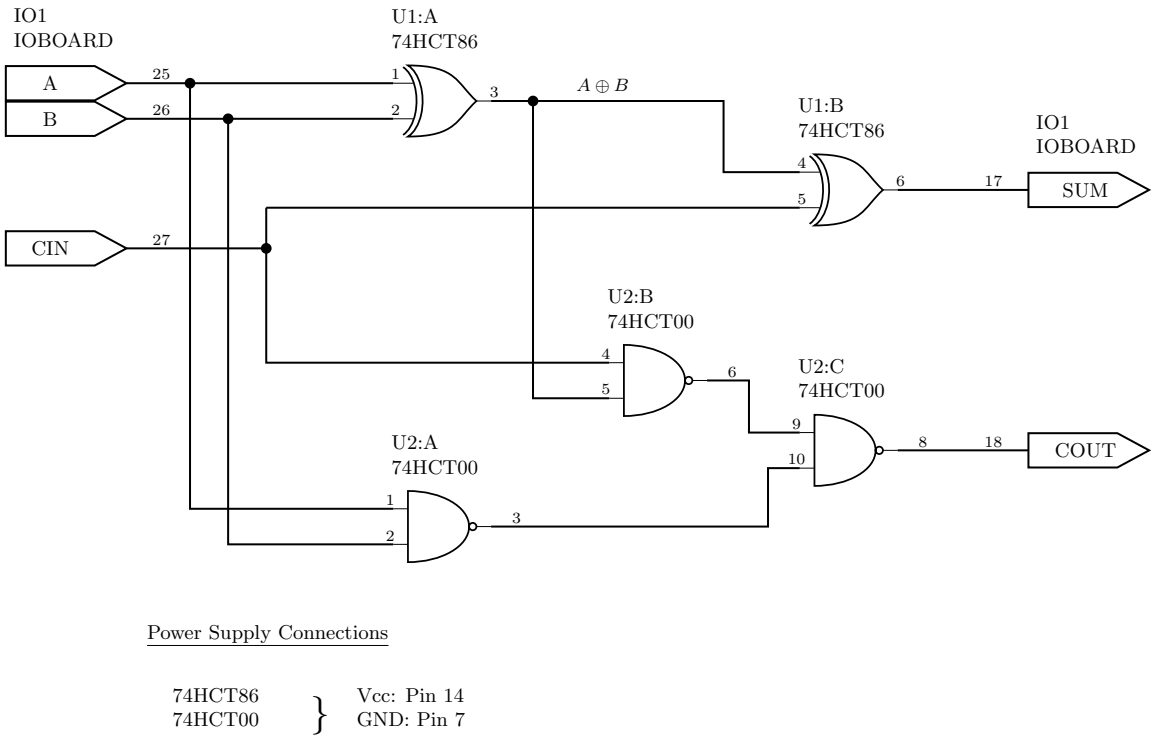
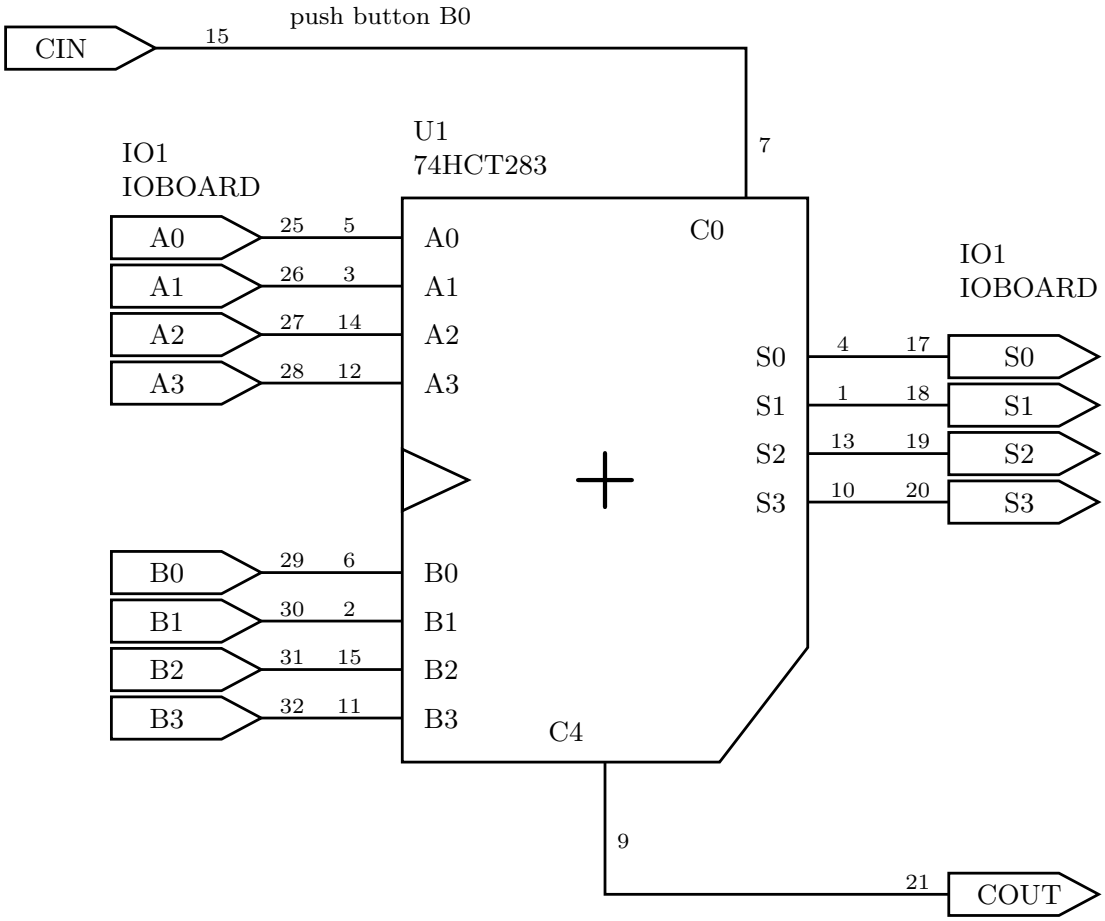


Figure 12: The Task 1 circuit schematic in its two-chip form: a 74HCT86 supplies the two XOR gates of the sum path and a 74HCT00 the three NAND gates of the carry path. Drawn to the rules in circuit-schematics — device ID and type on every gate, a pin number on every terminal, IO board inputs attached at the point of the pentagon and outputs at its flat back, and one power block because both chips are 14-pin parts with Vcc on 14 and GND on 7. The four junction dots — one at each fan-out of A, B, CIN and A-xor-B — are the only connections; where CIN's branch to U1:B crosses the A-xor-B drop, and where the switch wires cross CIN's run, the wires pass without a dot.



Power Supply Connections

74HCT283 } Vcc: Pin 16
GND: Pin 8

Figure 13: The Task 2 schematic. Eight switches supply the two operands, a push button supplies the carry-in, and five LEDs show the sum and the carry-out. The one thing this figure exists to make unmissable is the power block: the 74HCT283 is a 16-pin package, so its supply pins are 16 and 8, and a schematic that reuses 14/7 out of habit is wrong. The switch, button and LED pin assignments are chosen from the IO board pin list in device-pinouts.

Extension — overflow detector

Add a two's-complement overflow detector driving an LED. The catch:

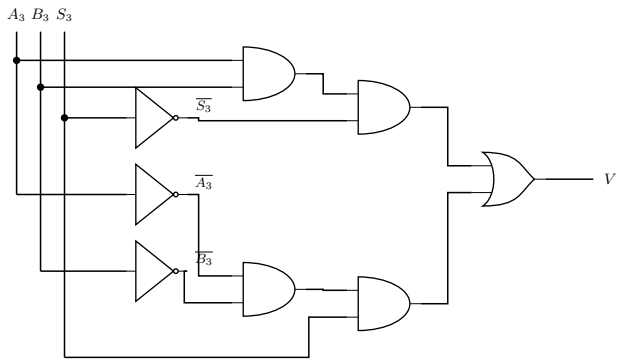
> There is no access to C_3, which is an internal signal. The overflow output has to be expressed in terms of available information — i.e. A_3, B_3, S_3, C_4.

The usual formula $V = C_3 \oplus C_4$ is therefore unavailable: the `74HCT283` chains C_1, C_2 and C_3 inside the package and only brings out C_0 and C_4.

The rule that uses only the pins you can reach is: **overflow happens exactly when both operands share a sign and the result's sign differs from it.** Adding two negatives can't give a positive; adding two positives can't give a negative. As an expression,

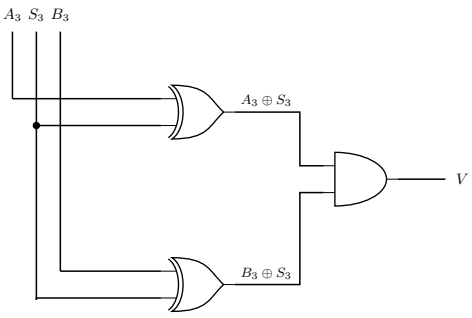
$$V = A_3 B_3 \overline{S_3} + \overline{A_3}\,\overline{B_3} S_3$$

C_4 is available but not needed — the sign bits alone settle it.



(a) $V = A_3 B_3 \overline{S_3} + \overline{A_3}\,\overline{B_3} S_3$

3 inverters, 4 ANDs, 1 OR: 74HCT04 + 74HCT08 + 74HCT32



(b) $V = (A_3 \oplus S_3)(B_3 \oplus S_3)$

2 XORs, 1 AND: 74HCT86 + 74HCT08

Figure 14: Two ways to build the overflow LED, both using only the pins the 74HCT283 brings out. (a) is the sum-of-products form read straight off the rule, and costs three chips because the kit has no gate with more than two inputs, so each three-literal product needs two ANDs. (b) is the same function factored as a product of two XORs: A3 differs from S3 and B3 differs from S3 exactly when the two operands agree in sign and the sum does not. Its two XORs are exactly the pair left spare on Task 1's 74HCT86, though the AND still needs a chip of its own: Task 1's 74HCT00 has only one gate free, and an AND built from NANDs takes two.

The stages of completion for Task 2: schematic done / partially working / fully working 4-bit adder / fully working **with overflow detection**. Again, the extension is the top rung.

Next

Lab 4 has a **preparation task** to be done before the session — see week3-lab-lab-4-pre-work[25].

Majority Function Circuit Design

Learning Lab 2's task, following on from revision of gate truth tables and Boolean identities (see logic-gates[26] and boolean-algebra[27]) and the introduction of circuit-schematics[28]/device-pinouts[29] conventions. Main lab objective: verify the functionality of a logic circuit either as a hardware circuit built from logic ICs, or as a Logisim simulation.

Question 1 — 3-input majority function

The 3-input **majority function** Z is true if at least two of the inputs A, B, C are true. Already simplified in 2026-07-30-intro-to-logic-gates[30] (the M truth table/sum-of-products example there is this same function):

$$Z = AB + AC + BC$$

(a) Convert Z to a NAND-only circuit (using Boolean algebra or a logic diagram).

(b) Draw a circuit schematic for the NAND-only circuit — inputs are switches, output is an LED.

(c) Wire up the circuit (or simulate in Logisim) and determine the truth table.

i Working

(a) NAND-only conversion. The standard **sum-of-products → NAND-only** technique (double negation + De Morgan's law, see boolean-algebra[31]): double-invert the whole expression, then push one inversion inside with De Morgan's law, turning each AND term into a NAND and the outer OR into a NAND of those NAND outputs:

$$Z = AB + AC + BC = \overline{\overline{AB + AC + BC}} = \overline{\overline{AB} \cdot \overline{AC} \cdot \overline{BC}}$$

[25] courses/csse2010/week3-lab-lab-4-pre-work.pdf
[26] courses/csse2010/logic-gates.pdf
[27] courses/csse2010/boolean-algebra.pdf
[28] courses/csse2010/circuit-schematics.pdf
[29] courses/csse2010/device-pinouts.pdf
[30] courses/csse2010/2026-07-30-intro-to-logic-gates.pdf

i.e. $Z = \text{NAND}(\text{NAND}(A,B),\ \text{NAND}(A,C),\ \text{NAND}(B,C))$ — each first-level AND gate becomes a NAND gate, and the second-level OR gate becomes a NAND gate combining the (already-inverted) first-level outputs.

The catch: that final combining gate is a 3-input NAND, and the kit has none. Every gate chip in *Device Pinouts and Symbols* is 2-input (plus the hex inverter) — see device-pinouts[32] — so the 3-input NAND has to be built from 2-input ones. Writing $N_1 = \overline{AB}$, $N_2 = \overline{AC}$, $N_3 = \overline{BC}$:

$$Z = \overline{N_1 N_2 N_3} = \text{NAND}(\overline{\overline{N_1 N_2}},\ N_3)$$

which is three more 2-input NANDs: one for $\overline{N_1 N_2}$, one wired as an inverter (both inputs tied together) to undo that inversion, and one to combine the result with N_3.

Six 2-input NAND gates in total, so **two 74HCT00 packages** — four gates on the first and two on the second.

(b) Circuit schematic.

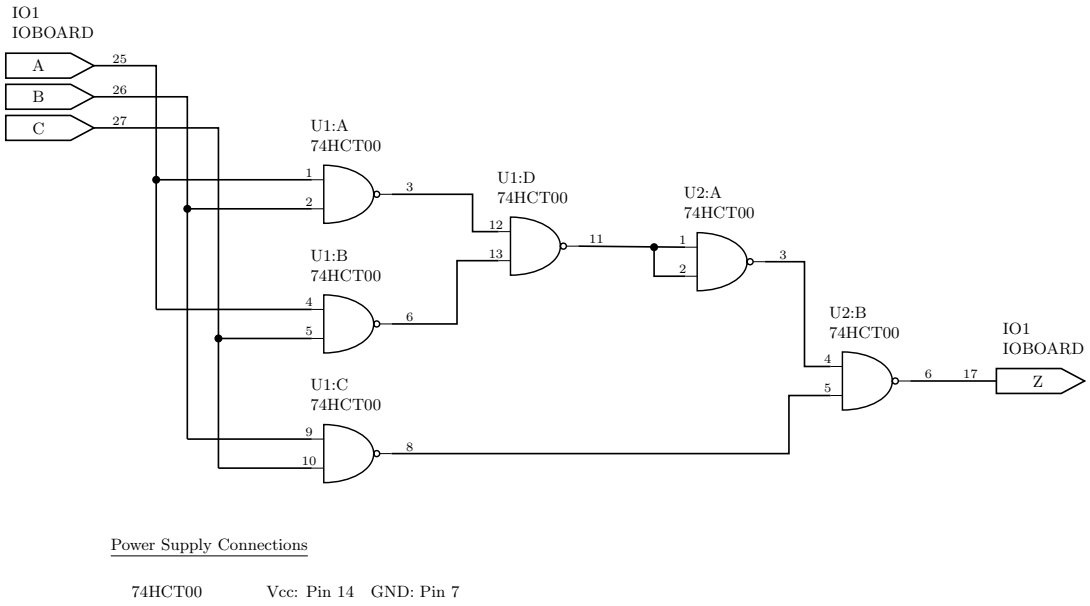


Figure 15: The NAND-only majority circuit as a circuit schematic. U1:A to U1:C are the three product terms; U1:D, U2:A and U2:B are the 3-input NAND built out of 2-input ones — combine, invert, combine again. U2:A is the NAND-as-inverter trick: both its inputs are tied to the same signal, which is what the junction dot on its input pair means. Six gates do not fit in one quad package, so the last two need a second chip, which is why U2 exists at all. The partitioning into packages, the pin assignment and the IO board pin choices follow the device pinouts.

(c) Truth table. Reading it off the definition (at least two inputs true) — and this is what the built circuit should reproduce:

A	B	C	Z
0	0	0	0
0	0	1	0
0	1	0	0
0	1	1	1
1	0	0	0
1	0	1	1
1	1	0	1
1	1	1	1

Test the built circuit against all eight rows in this order, not at random — the systematic sweep is what catches a swapped pin. Lab 4 asks for the same habit: decide the test procedure and the truth table you are expecting (see week3-lab-lab-4-pre-work[33]).

See also

- circuit-schematics[34] — the labelling rules the schematic above follows, and the seven common errors it is checked against
- device-pinouts[35] — the 74HCT00 gate/pin pairs and the IO board switch and LED pin numbers
- week2-lab-binary-arithmetic[36] — the next lab, which reuses the same NAND-conversion move on a full adder's carry expression

[33] courses/csse2010/boolean-algebra.pdf
[33] courses/csse2010/device-pinouts.pdf
[33] courses/csse2010/week3-lab-lab-4-pre-work.pdf
[34] courses/csse2010/circuit-schematics.pdf
[35] courses/csse2010/device-pinouts.pdf
[36] courses/csse2010/week2-lab-binary-arithmetic.pdf

Reference material

Arithmetic Logic Unit

The **ALU** is versatile hardware which can perform various operations based on some control inputs — it does considerably more than adding. See cpu-organisation[37] for where it sits in the CPU, and combinational-logic-blocks[38] and logic-gates[39] for the pieces it is built from.

The ALU as a block

A **function/control** input indicates the operation the ALU is to perform: addition, increment (+1), subtraction, bitwise AND, bitwise OR, and so on.

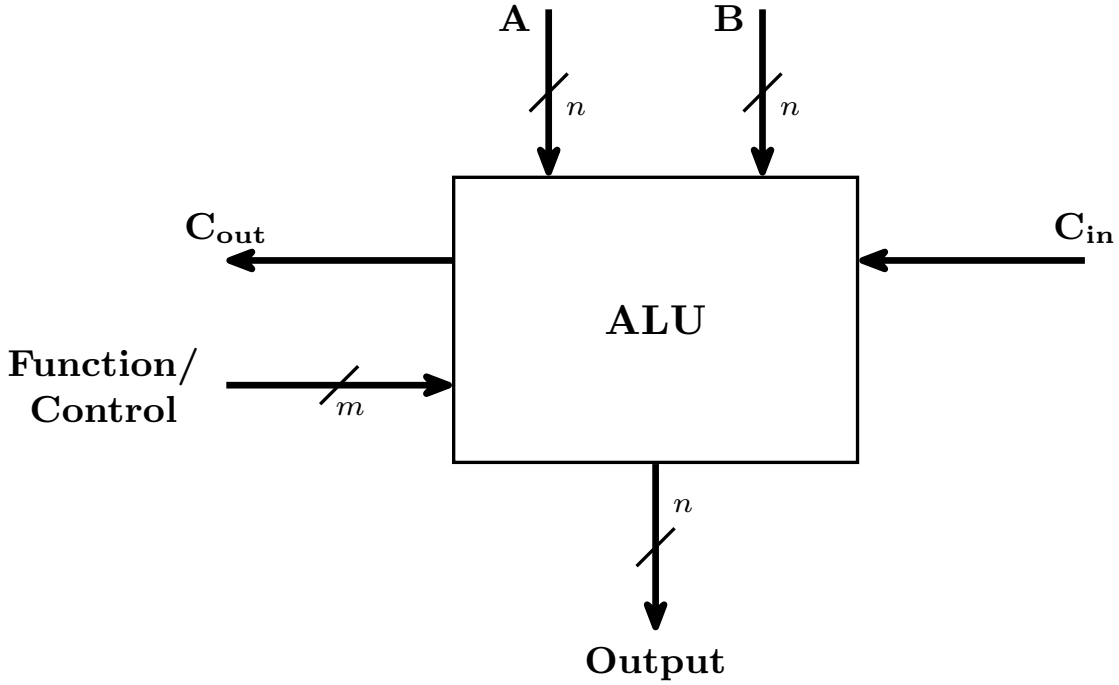


Figure 16: The ALU as a block. A and B are n-bit data inputs and Output is the n-bit result; the m-bit Function/Control input selects which operation is performed, and the carry in and out let several ALUs be chained. Slashes mark bus widths.

[37] courses/csse2010/cpu-organisation.pdf
[38] courses/csse2010/combinational-logic-blocks.pdf
[39] courses/csse2010/logic-gates.pdf

Like adders, ALUs can be made up from 1-bit slices — an n-bit ALU is n identical circuits, one per bit position.

The 1-bit ALU (bit slice)

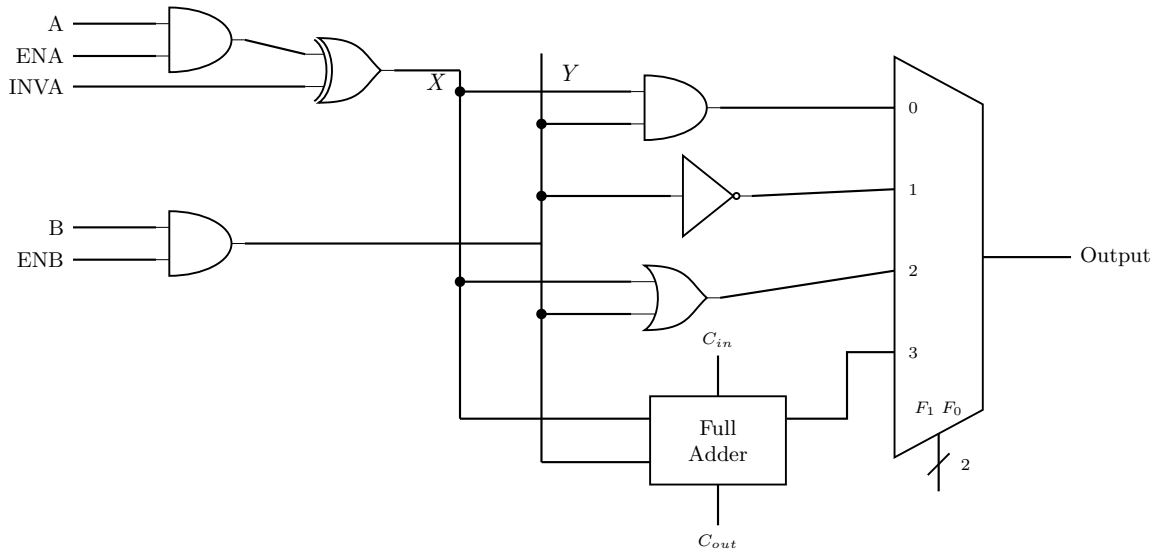


Figure 17: One bit slice. The ENA/INVA/ENB gates on the left condition the two data inputs into internal signals X and Y; the four function blocks in the middle compute AND, NOT, OR and sum from them simultaneously; the multiplexer selects which one reaches the output, under the F1 F0 select. Cin and Cout belong to the full adder, and are what chain one slice to the next.

F_1 and F_0 select which of the four function blocks reaches the output:

F_1F_0	Function	
00	A and B	*logical*
01	invert B	*logical*
10	A or B	*logical*
11	A plus B	*arithmetic*

Two levels of control

The bit slice has control inputs at **two distinct levels**, and keeping them apart is what makes the combinations table below readable:

1. **Permutations of the data inputs** — `ENA`, `ENB` and `INVA` condition A and B *before* any function sees them.
2. **Function select** — F_1F_0 picks which operation's result is actually output.

The conditioning gates are worth working out explicitly, because most of the useful ALU tricks come from them rather than from the function select. Writing X for the conditioned A line and Y for the conditioned B line:

$$X = (A \cdot \text{ENA}) \oplus \text{INVA} \qquad\qquad Y = B \cdot \text{ENB}$$

ENA	INVA	X	ENB	Y
0	0	0	0	0
0	1	1	1	B
1	0	A		
1	1	$\overline{A}$		

So the A line can be forced to a **constant 0**, a **constant 1** (all ones across the whole word, i.e. -1 in two's complement), A itself, or $\overline{A}$ — and that constant-generating ability is what lets one ALU compute -1, $B-1$ and $B-A$ without any dedicated hardware for them.

`INC` is the third conditioning input: it is the carry into the least significant slice, **0 for addition and 1 for increment**.

Cascading into an n-bit ALU

A machine word is n bits (often 8, 16, 32, …), and an **n-bit ALU is built from n identical bit slices**.

F_1, F_0, `ENA`, `ENB` and `INVA` are **shared between all bit slices** — there is one function select for the whole ALU, not one per bit. Only the carry is per-slice, and it **ripples through from right to left**, which is exactly the ripple-carry adder behaviour from earlier in the course.

Useful combinations of ALU inputs

Everything a single ALU can be made to do, by choosing the six control inputs. This table is worth understanding rather than memorising — each row follows from X, Y and the function select above.

F_1	F_0	ENA	ENB	INVA	INC	Function
1	0	1	0	0	0	A

F_1	F_0	ENA	ENB	INVA	INC	Function
1	0	0	1	0	0	B
1	0	1	0	1	0	$\overline{A}$
0	1	1	1	0	0	$\overline{B}$
1	1	1	1	0	0	$A + B$
1	1	1	1	0	1	$A + B + 1$
1	1	1	0	0	1	$A + 1$
1	1	0	1	0	1	$B + 1$
1	1	1	1	1	1	$B - A$
1	1	0	1	1	0	$B - 1$
1	1	1	0	1	1	$-A$
0	0	1	1	0	0	A AND B
1	0	1	1	0	0	A OR B
1	0	0	0	0	0	0
1	1	0	0	0	1	1
1	0	0	0	1	0	-1

Reading a few rows to see the pattern:

- A — select OR ($F_1F_0 = 10$), pass A through ($X = A$), force the other side to zero ($Y = 0$ via ENB $= 0$). $A + 0 = A$ in the OR sense.
- -1 — select OR, force X to all ones (ENA $= 0$, INVA $= 1$) and Y to zero. All ones **is** -1 in two's complement.
- $B - 1$ — select the adder, $Y = B$, \$X = \$ all ones $= -1$, and no extra carry. So $B + (-1)$.
- 1 — select the adder with both inputs forced to zero and INC $= 1$, so the result is the carry alone.

Worked example — computing $B - A$

What are the values of F_1, F_0, ENA, ENB, INVA and INC to calculate $B - A$?

The route is **two's complement negation**:

$$B - A = B + (-A) = B + (\overline{A} + 1) = B + \overline{A} + 1$$

So the ALU needs $\overline{A}$ on one adder input, B on the other, and a carry of 1 into the bottom slice:

Control	Value	Why
F_1F_0	1, 1	select the **adder**
ENA	1	let A through

Control	Value	Why
INVA	1	invert it, giving $\overline{A}$
ENB	1	let B through
INC	1	the +1 of the negation

Laid out as a column addition across the word, this is:

$$\begin{array}{cccccc} & \overline{B_7} & \overline{B_6} & \cdots & \overline{B_1} & \overline{B_0} \\ + & \overline{A_7} & \overline{A_6} & \cdots & \overline{A_1} & \overline{A_0} \\ + & 0 & 0 & \cdots & 0 & 1 \end{array}$$

The final row is the `INC` carry entering at the least significant slice and rippling left — which is why the +1 costs no extra hardware at all.

Overflow: the flag the carry-out does not give you

Every arithmetic row of the table above can go wrong in the same way: the true answer needs more bits than the word has. The ALU still produces *a* result, and it is a wrong one — so something has to say so.

The carry-out is not that something. C_{out} in Figure 16 is the correct overflow test for *unsigned* arithmetic and the wrong one for two's complement. In two's complement a carry out of the top slice is routine and is simply discarded (`1111 + 1111` gives a carry out and the perfectly correct answer -2), while $0111 + 0001$ produces **no** carry out at all and is nonetheless an overflow. The signed test is whether the top slice's carry *in* and carry *out* disagree:

$$V = C_n \oplus C_{n-1}$$

and equivalently, in terms of signs: **overflow happens exactly when the two operands share a sign and the result does not.** Adding two positives cannot legitimately give a negative; adding two negatives cannot legitimately give a positive.

When C_{n-1} is not available

That formula assumes you can reach the carry into the most significant slice, which inside your own cascade of bit slices you can. On a **single-package adder you cannot**: a 74HCT283 wires C_1, C_2 and C_3 stage-to-stage inside the chip and brings out only C_0 and C_4 (see device-pinouts[40]). Learning Lab 3 makes exactly this its extension task — build an overflow LED using only A_3, B_3, S_3 and C_4 — so the sign form is the one you actually wire:

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

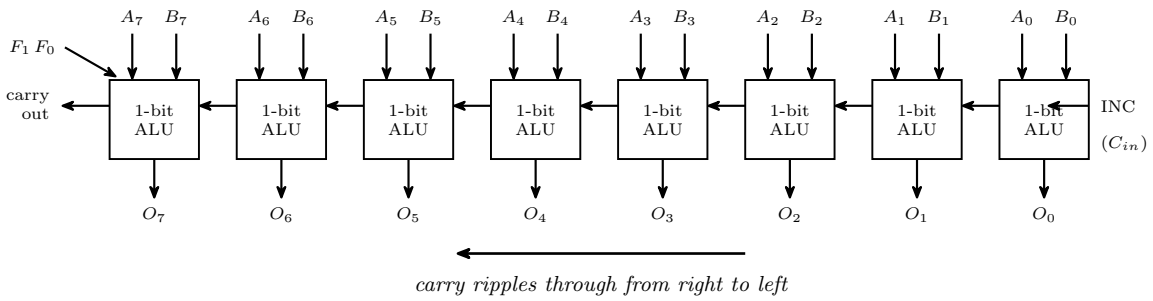


Figure 18: An 8-bit ALU as eight cascaded slices. Each slice takes its own bit of A and B and produces its own bit of the output; the carry out of one slice is the carry in of the next, rippling from right (least significant) to left. F1, F0, ENA, ENB and INVA are shared by every slice, so all eight always perform the same operation.

$$V = A_3 B_3 \overline{S_3} + \overline{A_3}\,\overline{B_3} S_3$$

Read literally: *(both operands negative and the sum positive) or (both operands positive and the sum negative)*. With only the two-input gates the course's parts list stocks, the same function factors into three gates — "the signs agree" AND "the sum disagrees with them":

$$V = (A_3 \odot B_3) \cdot (A_3 \oplus S_3)$$

There is a cheaper form still. A full adder gives $S_3 = A_3 \oplus B_3 \oplus C_3$, so the missing internal carry can be recovered as $C_3 = A_3 \oplus B_3 \oplus S_3$, and substituting into $V = C_4 \oplus C_3$ leaves

$$V = C_4 \oplus A_3 \oplus B_3 \oplus S_3$$

— three two-input XOR gates, all four of which fit in a single 74HCT86.

Back to $B - A$

The worked example above computes $B - A$ as $B + \overline{A} + 1$, and it is an ordinary addition as far as the top slice is concerned — so it is the *conditioned* operand that the overflow test must look at. The sign bit entering the adder is $\overline{A_{n-1}}$, not A_{n-1}, which is why a subtraction of two numbers whose signs *differ* is the case that can overflow, while an addition of two numbers whose signs differ never can. See week2-lab-binary-arithmetic[41] for the lab build and 2026-08-04-binary-arithmetic[42] for the arithmetic behind the rule.

[41] courses/csse2010/week2-lab-binary-arithmetic.pdf
[42] courses/csse2010/2026-08-04-binary-arithmetic.pdf

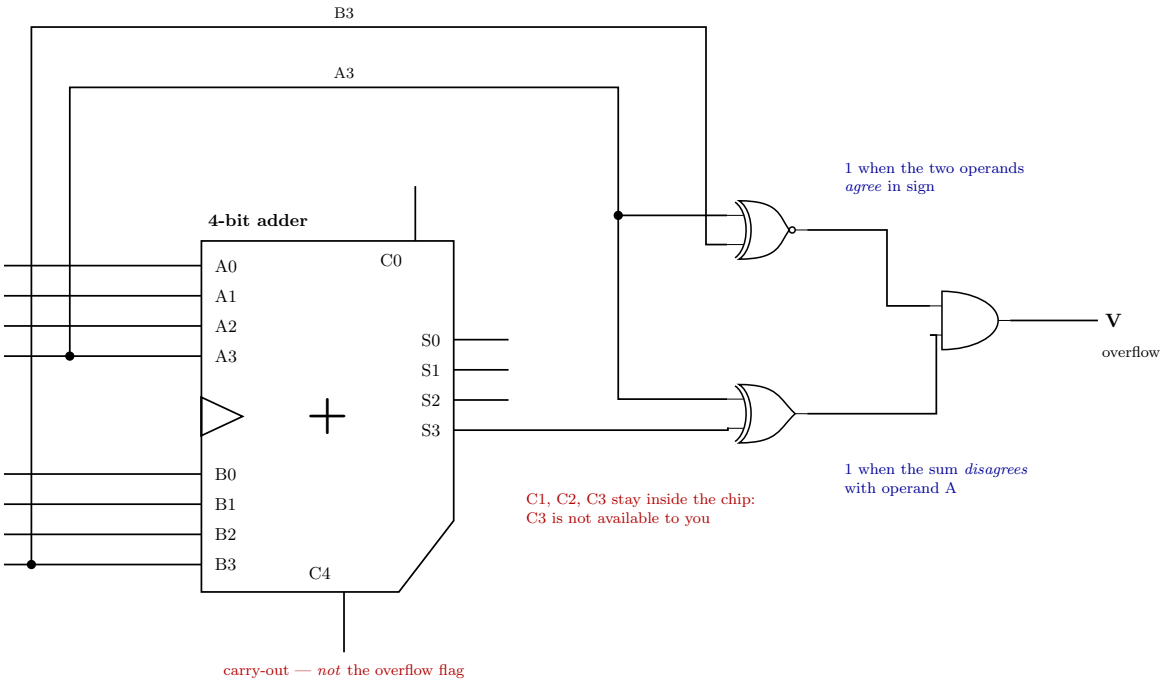


Figure 19: Two's complement overflow detected from the pins a packaged adder actually brings out. A3 and B3 are tapped from the operand inputs and S3 from the sum output; the XNOR asks whether the two operands agree in sign and the XOR asks whether the sum disagrees with them, and overflow is both at once. C4 is deliberately left going nowhere, because the carry-out is the unsigned overflow test, not this one. The adder outline, the '+' glyph and the input/output placement follow the course's 74HCT283 schematic symbol.

See also

- cpu-organisation[43] · fetch-decode-execute-cycle[44]
- binary-number-representations[45] — two's complement, which every subtraction row above relies on
- combinational-logic-blocks[46] — the multiplexer and decoder used inside the slice
- 2026-08-24-alu-and-control-unit[47] — Lecture 9

Binary Number Representations

Bits, bytes, and words

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

Representing whole (unsigned) numbers

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

Most and least significant bits

Converting decimal to binary

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

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

[43] courses/csse2010/cpu-organisation.pdf
[44] courses/csse2010/fetch-decode-execute-cycle.pdf
[45] courses/csse2010/binary-number-representations.pdf
[46] courses/csse2010/combinational-logic-blocks.pdf
[47] courses/csse2010/2026-08-24-alu-and-control-unit.pdf

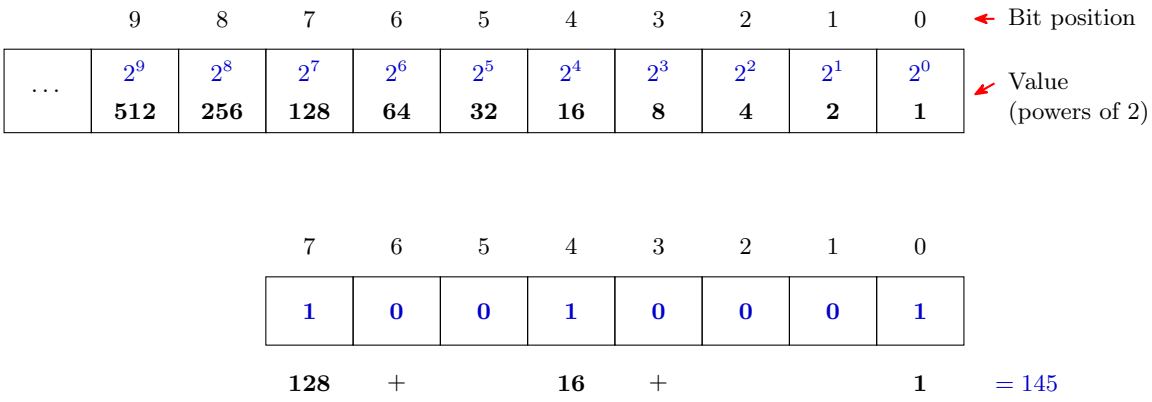


Figure 20: Place values for an unsigned binary number, and a worked conversion. Position i is worth 2^i, so reading a binary number is just adding the place values of the positions holding a 1: $10010001_2 = 128 + 16 + 1 = 145$.

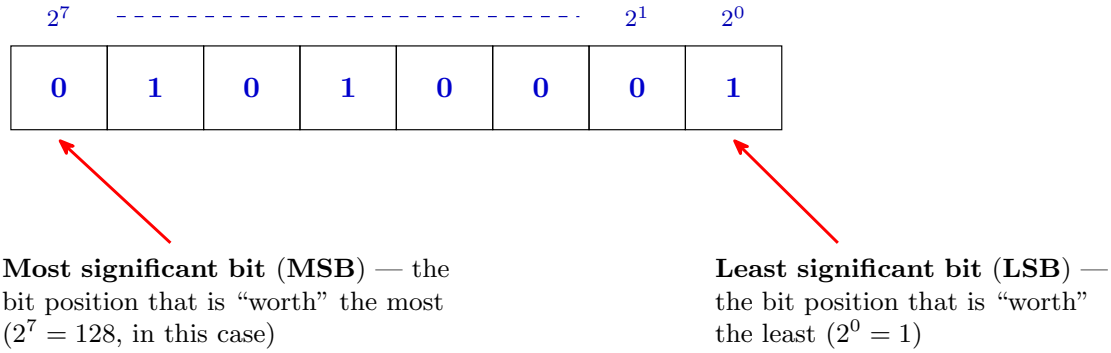


Figure 21: The two ends of a word get names, because almost every later convention (shifting, sign bits, bit numbering in AVR registers) is stated in terms of them. For an n-bit unsigned word the MSB is worth 2^{n-1} — here $2^7 = 128$ — and the LSB is always worth $2^0 = 1$.

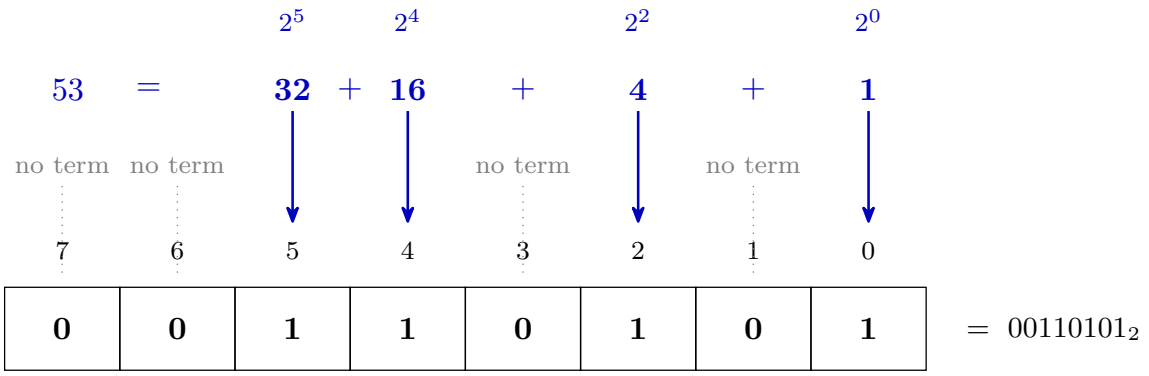


Figure 22: Method 1 on 53. Each power of two in the decomposition names one bit position — 2^5 is position 5, 2^4 is position 4, and so on — and every position with no term in the sum gets a 0. Reading the row off left to right gives 00110101_2.

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**.

Number range (unsigned)

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

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

Other radices

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

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

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

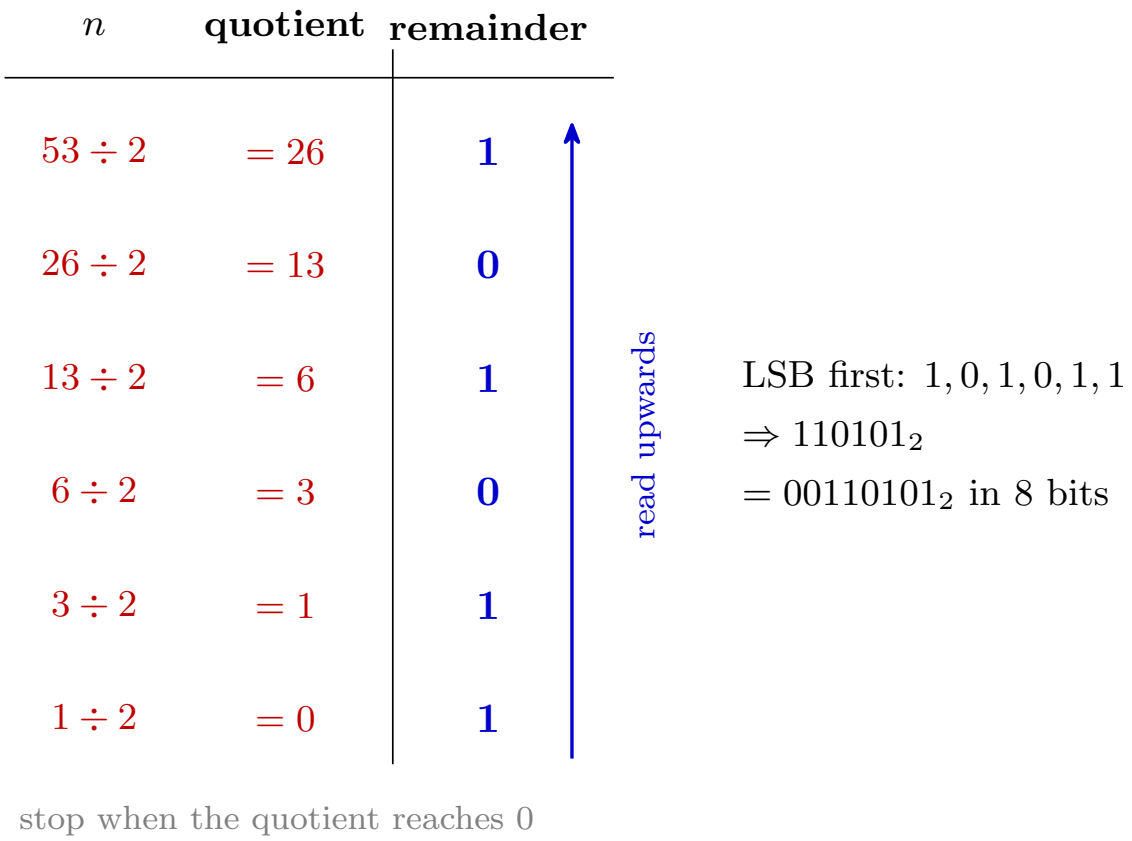


Figure 23: Method 2 on 53. The remainders come out LSB-first, which is why the answer has to be read back up the column: 110101_2, or 00110101_2 padded to a byte. The same procedure with a divisor of 16 or 8 converts to hexadecimal or octal instead.

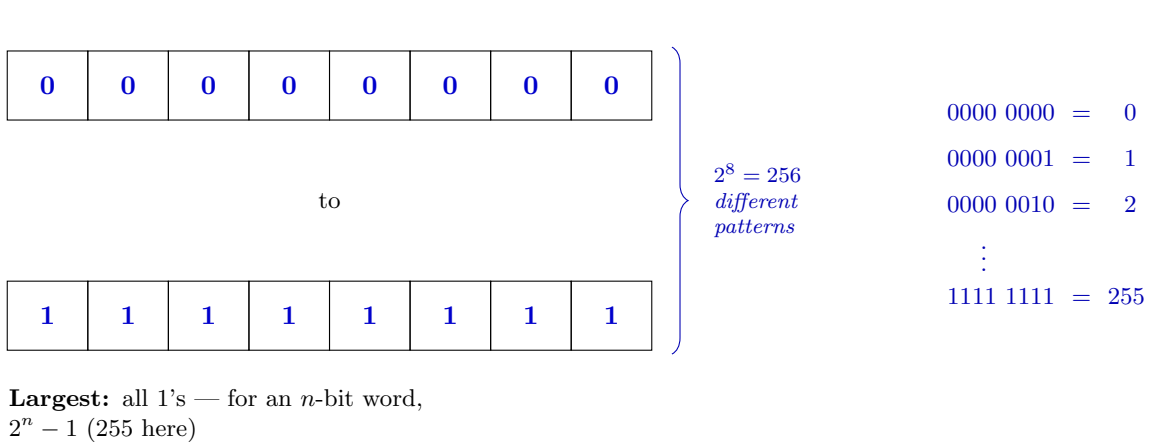


Figure 24: Why the largest unsigned value is $2^n - 1$ and not 2^n. Counting up from all-zeros steps through every pattern in turn — each extra bit doubles how many there are, so 8 bits give $2^8 = 256$ patterns — and because the count starts at 0, the last one is 255.

Converting binary ⇄ hex/octal by grouping

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

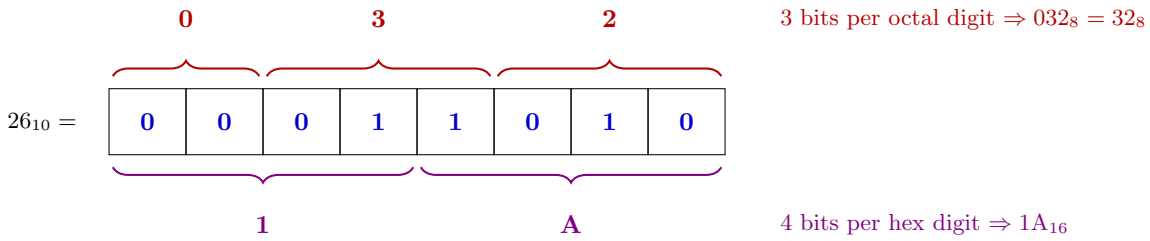


Figure 25: One bit pattern read two ways. Grouping 00011010_2 into nibbles from the right gives 1 and A, i.e. $1A_{16}$; grouping the same bits in threes gives 0, 3 and 2, i.e. 32_8. The grouping must start at the right — starting at the left gives the wrong answer whenever the width is not a multiple 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[48] for that worked exercise.

Radix notation conventions

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

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

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

Representing signed numbers

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

Signed magnitude

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

Ones' complement

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

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

Two's complement

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

Compare the position values, 8-bit:

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

Excess-2^{N-1}

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

It is exactly **two's complement with the sign bit reversed** — a 0 in the MSB means bit value −128, a 1 means 0. Its practical use is that comparing two signed numbers becomes simple in hardware. The exponent of an IEEE floating-point number is stored in excess notation for exactly that reason, with a bias one smaller: excess-127 for 8 bits (floating-point-numbers[49]).

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

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

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

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

Why "complement"?

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

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

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

Why two's complement wins

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

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

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

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

Format matters

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

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

Read as	Value
Signed magnitude	-18
Excess-128	18

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

The format is not encoded in the data. It has to be known separately — by a human, by a program, or by the hardware. See week1-lab-number-representations[50] for the full 4-bit table across all five formats, and 2026-08-04-binary-arithmetic[51] for what happens when you add these.

Fractions: fixed and floating point

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

- **Fixed point**: agree where the point sits, as in the 5.3 reading of `01010101` above, and read the bits as before. Ordinary integer hardware does the arithmetic, but the format fixes the range and the precision once and for all. See fixed-point-numbers[52].
- **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[53].

Boolean Algebra

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

Notation conventions

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

[50] courses/csse2010/week1-lab-number-representations.pdf
[51] courses/csse2010/2026-08-04-binary-arithmetic.pdf
[52] courses/csse2010/fixed-point-numbers.pdf
[53] courses/csse2010/floating-point-numbers.pdf

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

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

Boolean identities

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

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

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

De Morgan's law and gate equivalents

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

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

This is also why NAND-only and NOR-only circuits can implement any function (see logic-gates[54]) — an AND feeding an OR can always be redrawn as a NAND feeding a NAND.

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

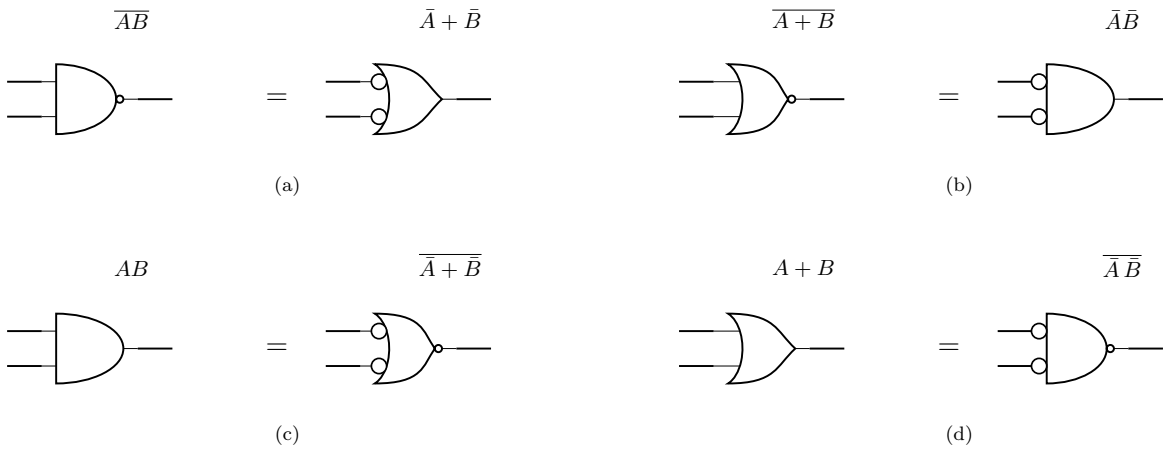


Figure 26: De Morgan's law drawn as gate symbols. (a) a NAND is an OR with inverted inputs; (b) a NOR is an AND with inverted inputs; (c) and (d) are the same two statements read the other way. Each pair is the same function, so either symbol may be used - pick whichever makes the surrounding bubbles cancel.

Simplifying an expression

Worked example:

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

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

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

Sum of products

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

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

Because every 1-row gets its own term, and each term is true for exactly that one row, the result is guaranteed to reproduce the table. Worked through in full, with the resulting circuit drawn out, in 2026-07-30-intro-to-logic-gates[55].

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[56]). 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[57] for pin layouts). The rules are set out in the *CSSE2010/CSSE7201 - Guide to Drawing Circuit Schematics* (Blackboard, v2.3).

Required elements

Every circuit schematic must show:

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

[55] courses/csse2010/2026-07-30-intro-to-logic-gates.pdf
[56] courses/csse2010/logic-gates.pdf
[57] courses/csse2010/device-pinouts.pdf

- **Gate labels within a chip** — where a chip contains multiple gates, each gate is labelled `A`, `B`, `C`, … after the chip ID, separated by a colon (e.g. `U1:A`, `U1:B`). Labels can be assigned to gates in any order. A chip with only one gate/device doesn't need a gate letter (just `U3`).
- **Device type** — written on the line below the device ID (e.g. `74HCT00`, `74HCT04`, `IOBOARD`).
- **Pin numbers** — every pin used must be numbered per the actual device pinout (see device-pinouts[58]).
- **Power supply connections** — shown once per device *type*, not per individual chip. Where two+ chip types share the same power pins, they can be grouped together. The IO board doesn't need power connections shown.

Worked example

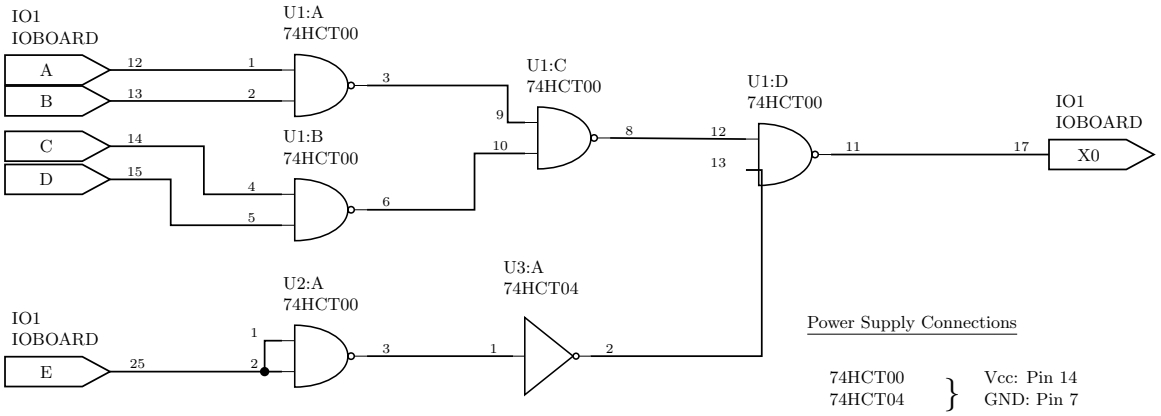


Figure 27: A correct schematic. Every rule above is visible here at once: five named inputs (A-E) and one named output (X0) on the IO board, four gates of one 74HCT00 labelled U1:A to U1:D, a fifth NAND that had to move to a second chip U2, an inverter on a third chip U3 because it is a different gate type, a pin number on every terminal, a junction dot where E splits to both inputs of U2:A, and one power-supply block covering both chip types.

Reading it against the rules:

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

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

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

Common errors

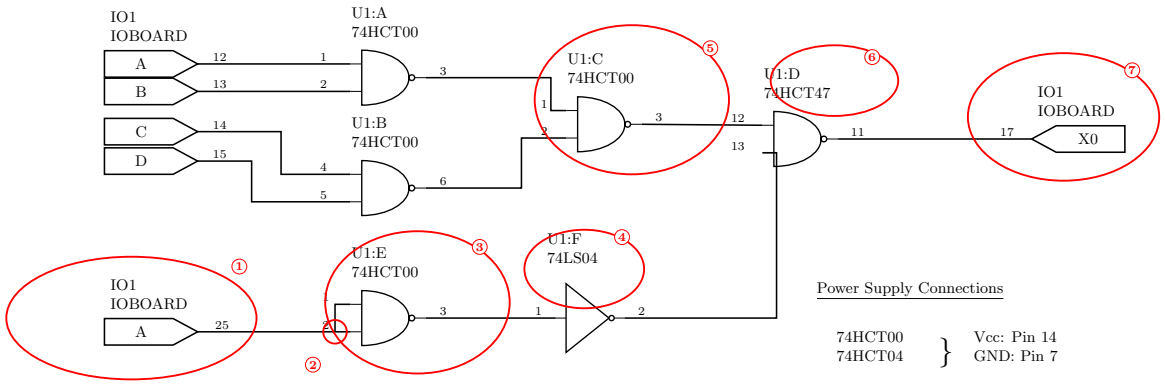


Figure 28: The same circuit with seven deliberate mistakes ringed and numbered left to right, matching the list below. Compare each ring against the corresponding part of the correct schematic above.

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

a. The *input* symbol is used (wire attached to the point) where the output symbol is needed (wire attached to the flat back) — see Figure 27.
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[59] — the pin numbers and symbols every schematic has to match
- logic-gates[60] — the gate symbols themselves, and the logic-diagram level above this one
- week3-lab-lab-4-pre-work[61] — the preparation task that asks for schematics of a 2-to-1 mux, a 2-to-4 decoder and a 4-bit adder/subtractor

Combinational Logic Blocks

Reference note for the standard combinational building blocks: the adder-subtractor, the multiplexer, and the decoder. See logic-gates[62] for the gate symbols/truth tables and boolean-algebra[63] 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[64], 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).

[59] courses/csse2010/device-pinouts.pdf
[60] courses/csse2010/logic-gates.pdf
[61] courses/csse2010/week3-lab-lab-4-pre-work.pdf
[62] courses/csse2010/logic-gates.pdf
[63] courses/csse2010/boolean-algebra.pdf
[64] courses/csse2010/sequential-circuits.pdf

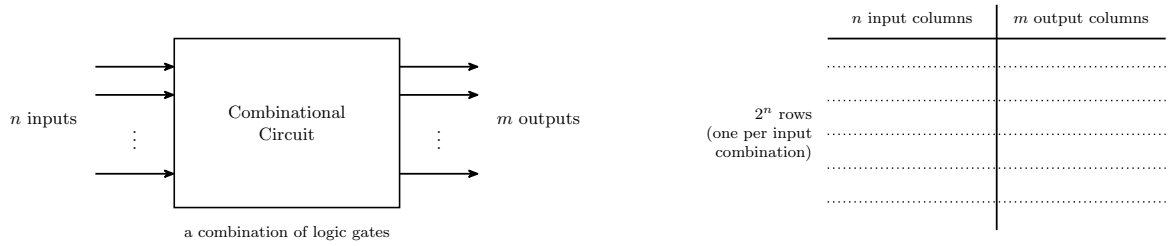


Figure 29: The general shape of a combinational circuit and the truth table it is always equivalent to. The row count 2^n depends only on the number of inputs, and the output column count m only on the number of outputs — so a block with six inputs needs 64 rows however few gates are inside it.

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[65]).

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,

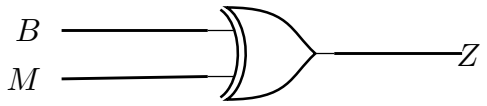
$$Z = \bar{B} \text{ when } M = 1, \qquad Z = B \text{ when } M = 0$$

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**:

$$Z = B \oplus M$$

[65] 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 30: 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[66]) 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[67].

[66] courses/csse2010/2026-08-04-binary-arithmetic.pdf
[67] courses/csse2010/device-pinouts.pdf

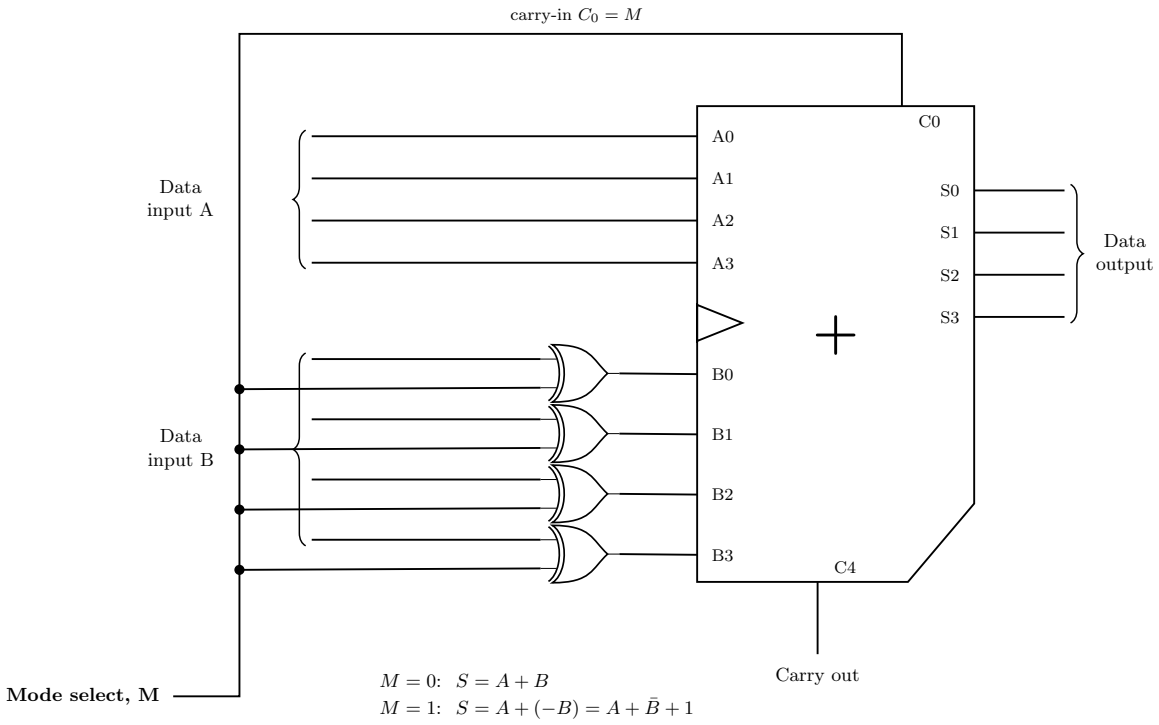


Figure 31: The 4-bit adder-subtractor. One mode line does two jobs: it conditions every bit of B through an XOR, and it is the adder's carry-in. With M = 0 the XORs pass B through and the carry-in is 0, so the adder computes A + B; with M = 1 the XORs deliver B-bar and the carry-in is 1, so it computes 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.

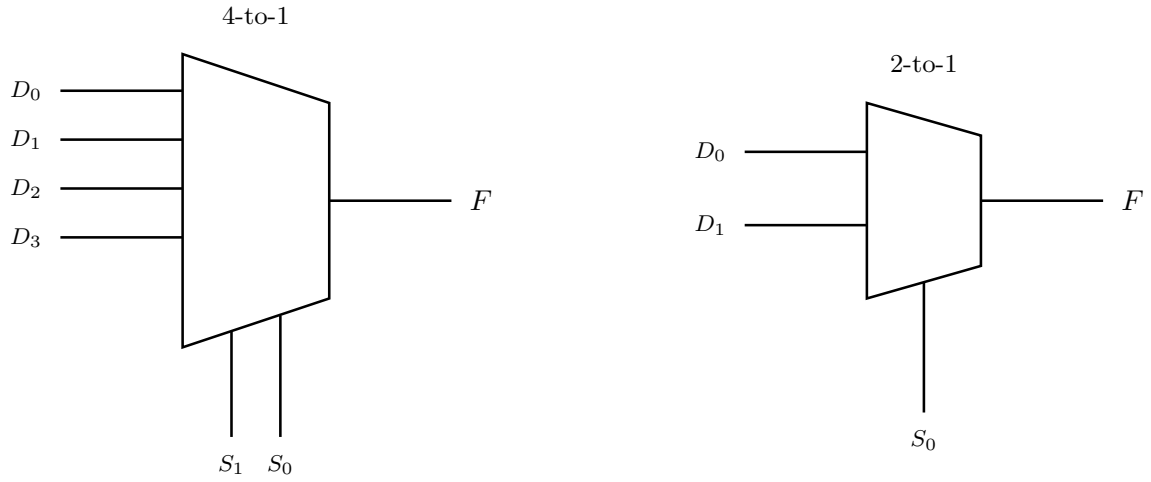


Figure 32: The multiplexer logic symbol at two sizes, 4-to-1 and 2-to-1. Data inputs enter the tall left edge, the single output leaves the short right edge, and the select inputs enter from underneath. The number of select lines is what fixes the size: n selects steer 2^n data inputs.

4-to-1 multiplexer

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

Function table:

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

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

4-to-1 mux logic circuit implementation

- S_1 and S_0 each feed an inverter, giving $\bar{S_1}$ and $\bar{S_0}$ as well as the true forms.
- Four **3-input AND gates**, one per data input. Each AND gate takes its data input plus the select-literal pair that identifies it: D_0 with $\bar{S_1}\bar{S_0}$, D_1 with $\bar{S_1}S_0$, D_2 with $S_1\bar{S_0}$, D_3 with S_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[68]) — building this from the available 2-input chips means cascading them.

2-to-1 multiplexer

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

Function table:

S_0	F
0	D_0
1	D_1

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

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

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

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

i.e. $F = D_0\bar{S}_0 + D_1S_0$. 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[69].

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[70].

Decoder

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

3-to-8 decoder

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

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

[69] courses/csse2010/device-pinouts.pdf
[70] courses/csse2010/2026-08-06-combinational-logic.pdf

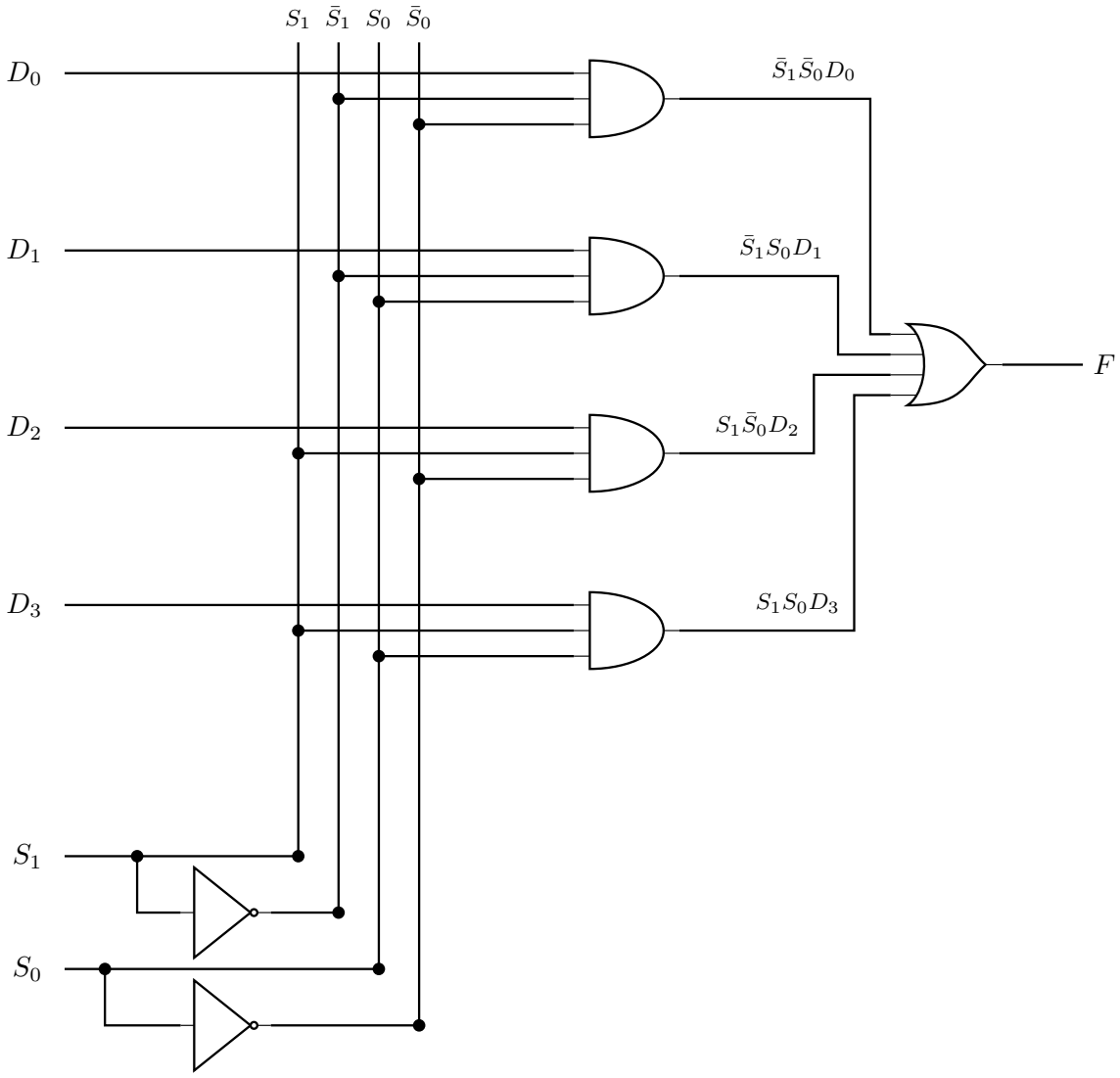


Figure 33: A 4-to-1 multiplexer built from gates. The two inverters generate the complemented selects; the four vertical rails carry S1, S1-bar, S0 and S0-bar past every AND gate; each AND gate is one data input gated by the one select combination that names it, and the OR gate collects them. Only the dots are connections — every other crossing is two wires passing.

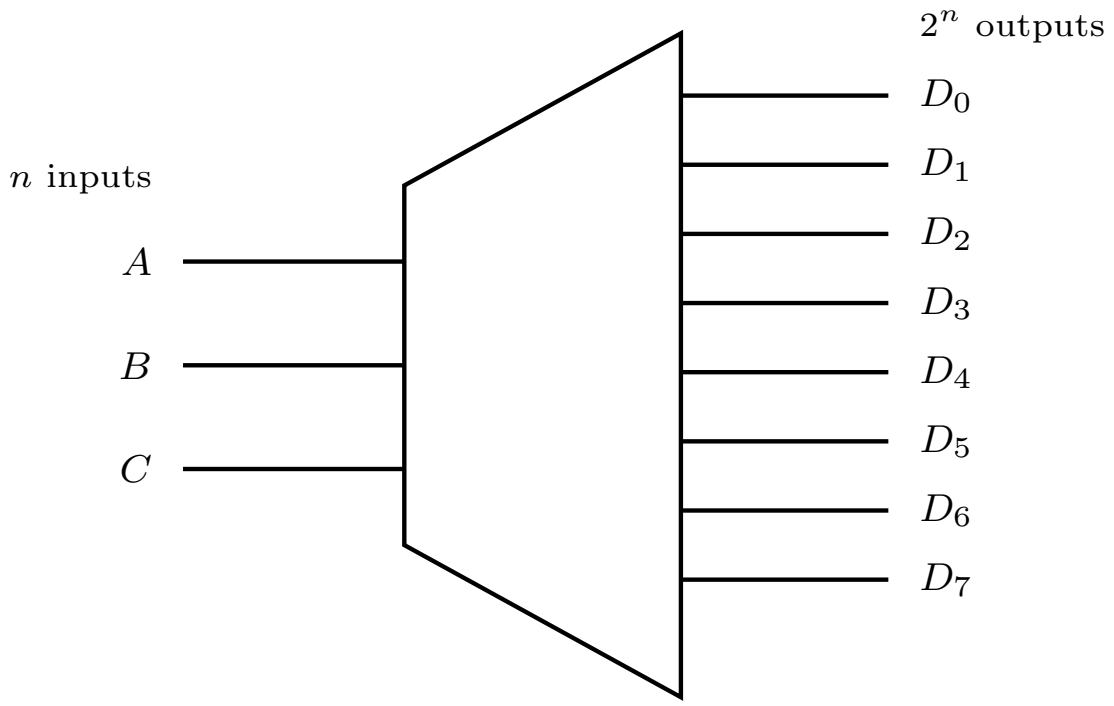


Figure 34: The decoder as a block: n inputs in, 2^n outputs out, and the shape widens accordingly. It is the mux's opposite — a mux funnels 2^n inputs down to one output under n selects, a decoder fans n inputs out to 2^n outputs.

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

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

3-to-8 decoder logic circuit implementation

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

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

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

Using a decoder to implement a logic function

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

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

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

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

See also

- 2026-08-06-combinational-logic[71] — Lecture 4, where these blocks were introduced
- device-pinouts[72] — the off-the-shelf parts: 74HCT157 (mux), 74HCT283 (adder), 74HCT86 (XOR)
- arithmetic-logic-unit[73] — the ALU bit slice, which is a mux selecting between several function blocks
- binary-number-representations[74] — two's complement, on which the subtractor depends

[71] courses/csse2010/2026-08-06-combinational-logic.pdf
[72] courses/csse2010/device-pinouts.pdf
[73] courses/csse2010/arithmetic-logic-unit.pdf
[74] courses/csse2010/binary-number-representations.pdf

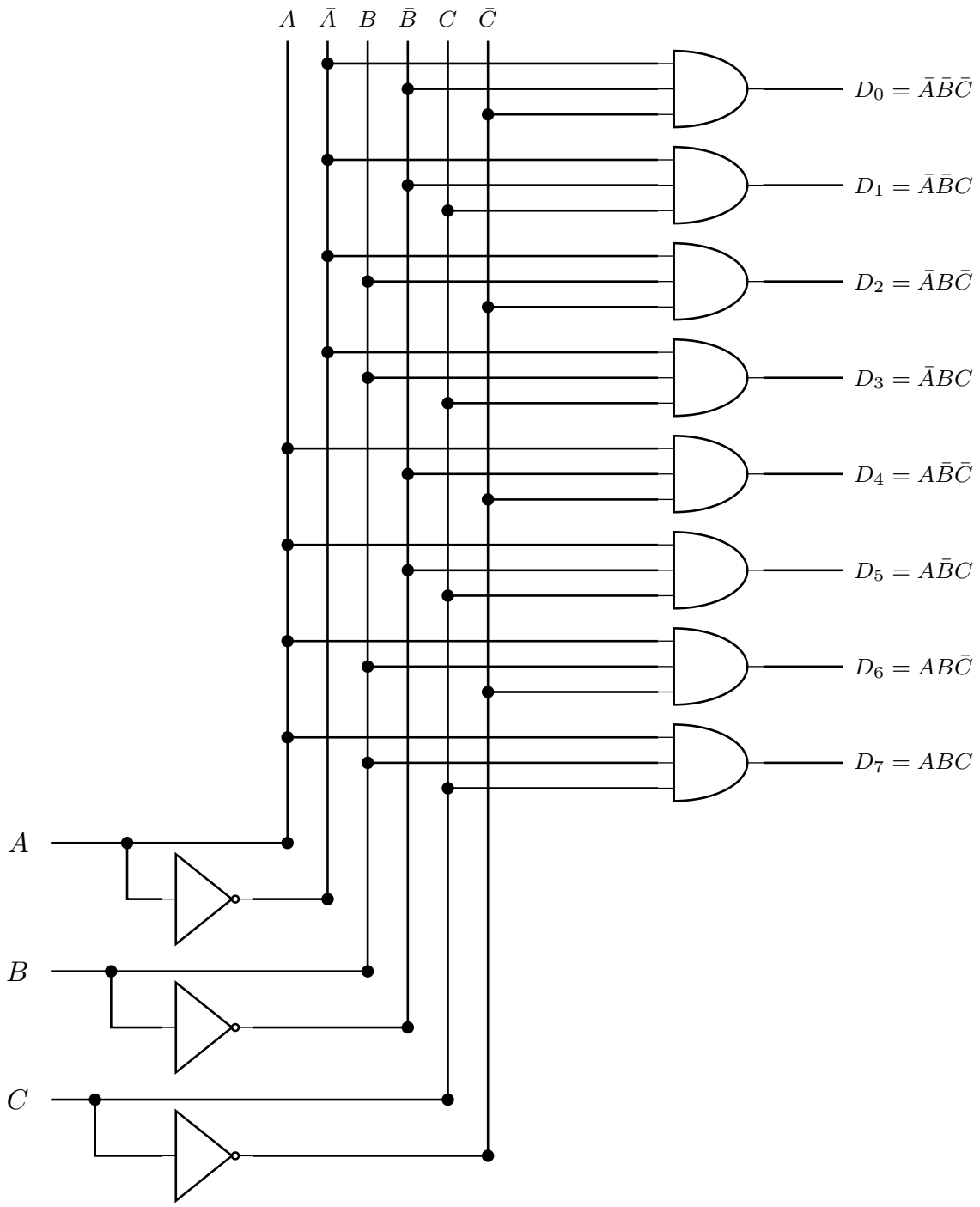


Figure 35: A 3-to-8 decoder built from gates. Three inverters generate the complements onto six vertical rails; each of the eight AND gates taps the one literal triple that matches its index, so it is the minterm for that input combination. Exactly one AND gate can be high at a time, which is the decoder's defining property.

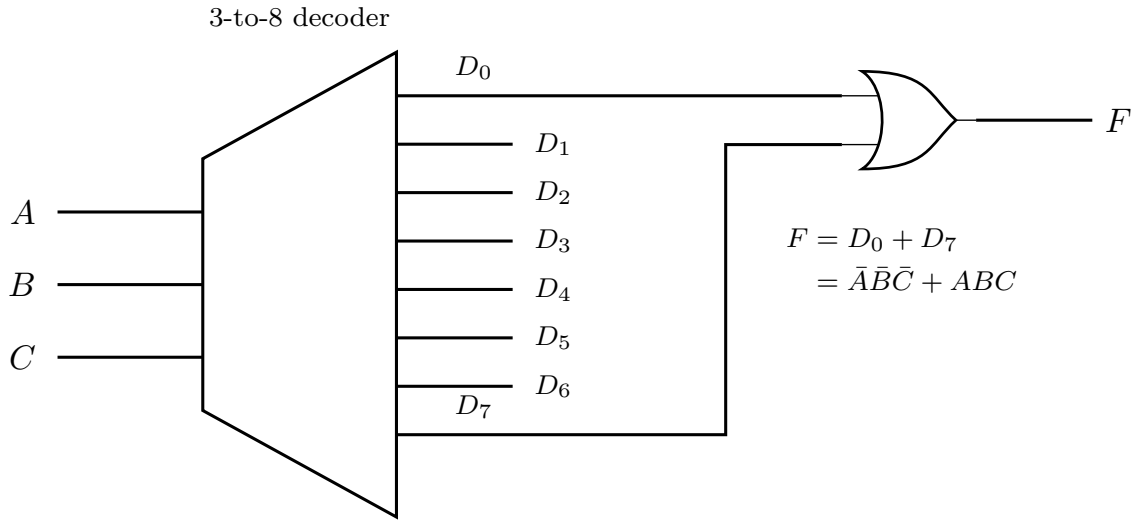


Figure 36: A function built on a decoder: F = A-bar B-bar C-bar + ABC is D0 + D7, so it costs a single OR gate on top of the decoder. Any function of A, B and C can be built this way — pick out the outputs for its true rows and OR them together.

Sample Mid-semester Exam #2

The second of UQ's sample mid-semester papers for **CSSE2010/CSSE7201 Introduction to Computer Systems** — see csse2010[75] for the course note. The cover page is from the Semester One mid-semester examinations, 2020, for St Lucia campus students. It allows 60 minutes of working time plus 10 minutes of reading time, is an **open book** examination with no electronic aids other than a Casio FX82-series or UQ-approved calculator, and is answered on a supplied multiple choice answer sheet. There are **25 questions**, each worth one mark, and the marking is "choose the answer which best answers the question or completes the statement" — a correct answer scores one mark, and an incorrect, missing or multiple answer scores zero. There is no penalty for guessing.

> ⚠ Answers are derived, not official
>
> The source paper ships without an answer key, so every answer below was worked out during transcription rather than taken from UQ. Each one shows its reasoning so it can be checked rather than trusted; anything uncertain is flagged. Treat a disagreement between your working and the answer here as a reason to check both.

[75] courses/csse2010/csse2010.pdf

How to use this

Two ways. **Timed run** — sit all 25 questions in 60 minutes as if it were the real thing; on the website the answers are collapsed, so they stay hidden until you click them. **Targeted practice** — use the index table below to pick out only the questions for one week or one concept, and follow the wikilink to the note that covers the method.

Note that the PDF build of this vault renders every callout expanded, so the answers are visible there. Practise from the site version.

Question index

Q	Topic	Week	Concept notes
1	Two's complement representation	1	binary-number-representations[76]
2	Octal, hex and decimal	1	binary-number-representations[77]
3	Excess-N range	1	binary-number-representations[78]
4	One's complement to excess-N	1	binary-number-representations[79]
5	Reading a gate circuit	1	logic-gates[80] · boolean-algebra[81]
6	Circuit for an XOR function	1	logic-gates[82]
7	Boolean simplification	1	boolean-algebra[83]
8	Truth table for an XOR chain	1	logic-gates[84]
9	Sum of products from a table	1	boolean-algebra[85]
10	XOR identities	1	boolean-algebra[86] · logic-gates[87]

[76] courses/csse2010/binary-number-representations.pdf
[77] courses/csse2010/binary-number-representations.pdf
[78] courses/csse2010/binary-number-representations.pdf
[79] courses/csse2010/binary-number-representations.pdf
[80] courses/csse2010/logic-gates.pdf
[81] courses/csse2010/boolean-algebra.pdf
[82] courses/csse2010/logic-gates.pdf
[83] courses/csse2010/boolean-algebra.pdf
[84] courses/csse2010/logic-gates.pdf
[85] courses/csse2010/boolean-algebra.pdf
[86] courses/csse2010/boolean-algebra.pdf
[87] courses/csse2010/logic-gates.pdf

Q	Topic	Week	Concept notes
11	NAND as a universal gate	1	logic-gates[88]
12	Multiplexer as function generator	2	combinational-logic-blocks[89]
13	Two's complement addition	2	binary-number-representations[90]
14	D flip-flop timing	3	flip-flops-and-latches[91] · timing-diagrams[92]
15	NOR SR latch behaviour	3	flip-flops-and-latches[93]
16	Shift register with function select	3	shift-registers[94]
17	Synchronous counter sequence	4	counters[95] · sequential-circuits[96]
18	Overflow detection	2	binary-number-representations[97]
19	Overflow in a signed addition	2	binary-number-representations[98]
20	State table to state diagram	4	state-machines[99]
21	Flip-flops needed for a machine	4	state-machines[100]
22	Tracing a state machine	4	state-machines[101]
23	ALU control for NOR	5	arithmetic-logic-unit[102]
24	ALU control for NOT B	5	arithmetic-logic-unit[103]

[88] courses/csse2010/logic-gates.pdf
[89] courses/csse2010/combinational-logic-blocks.pdf
[90] courses/csse2010/binary-number-representations.pdf
[91] courses/csse2010/flip-flops-and-latches.pdf
[92] courses/csse2010/timing-diagrams.pdf
[93] courses/csse2010/flip-flops-and-latches.pdf
[94] courses/csse2010/shift-registers.pdf
[95] courses/csse2010/counters.pdf
[96] courses/csse2010/sequential-circuits.pdf
[97] courses/csse2010/binary-number-representations.pdf
[98] courses/csse2010/binary-number-representations.pdf
[99] courses/csse2010/state-machines.pdf
[100] courses/csse2010/state-machines.pdf
[101] courses/csse2010/state-machines.pdf
[102] courses/csse2010/arithmetic-logic-unit.pdf
[103] courses/csse2010/arithmetic-logic-unit.pdf

Q	Topic	Week	Concept notes
25	Half adder from multiplexers	2	combinational-logic-blocks[104]

Q1 — Two's complement representation

What is -27_{10} expressed in 7-bit two's complement format?

(a) 0011011

(b) 1011011

(c) 1100100

(d) 1100101

(e) 1100110

Answer

(d)
Two's complement negation is "invert every bit, then add one". Start from +27 in seven bits:

$$27 = 16 + 8 + 2 + 1 = 0011011_2$$

Invert: 1100100. Add one: $1100100 + 1 = 1100101$.
The distractors are the intermediate steps, which is exactly the trap: (a) is +27, (c) is the one's complement (the inverted bits, before the +1), and (b) is +27 with only the sign bit flipped, i.e. sign-and-magnitude. See binary-number-representations[105].

Q2 — Octal, hex and decimal

110_8 is equal to

(a) 10001000_2

(b) 68_{10}

(c) 0x48

[104] courses/csse2010/combinational-logic-blocks.pdf
[105] courses/csse2010/binary-number-representations.pdf

(d) 6_{10}

(e) None of the above

Answer

(c)
Octal digits are worth powers of 8, so

$$110_8 = 1 \times 8^2 + 1 \times 8^1 + 0 \times 8^0 = 64 + 8 = 72_{10}$$

Checking each option: (a) $10001000_2 = 128 + 8 = 136$; (b) 68; (c) $0x48 = 4 \times 16 + 8 = 72$; (d) 6.
The quick route avoids decimal altogether — each octal digit is three bits, so $110_8 = 001\,001\,000_2 = 0100\,1000_2$, and regrouping those bits into nibbles reads straight off as `0x48`. See binary-number-representations[106].

Q3 — Excess-N range

A 10-bit binary number in excess-512 representation can represent integers from

(a) -512 to 511

(b) -512 to 512

(c) -511 to 511

(d) -511 to 512

(e) None of the above

Answer

(a)
In excess-N, the stored bit pattern is the value plus the bias: pattern = value + N. Ten bits hold the unsigned patterns 0 through 1023, so with $N = 512$:

$$\text{value} = \text{pattern} - 512 \quad \Rightarrow \quad 0 - 512 = -512 \;\text{ to }\; 1023 - 512 = 511$$

That is the same range as 10-bit two's complement, which is the point of choosing the bias to be 2^{n-1} — the two representations differ only in the most significant bit. See binary-number-representations[107].

106 courses/csse2010/binary-number-representations.pdf

107

Q4 — One's complement to excess-N

How can any n-bit binary number in one's complement representation be converted to excess-2^{n-1} representation?

(a) By flipping the most-significant bit and then adding 1

(b) By flipping the most-significant bit and then adding the value of the most-significant bit

(c) By adding the value of the most-significant bit and then flipping the most-significant bit

(d) By adding the value of the most-significant bit and flipping all the bits

(e) By flipping all the bits except the most-significant bit

Answer

(c)
Go through two's complement, because each leg is a one-line rule:

1. **One's complement → two's complement.** The two representations agree on non-negative values; for negative values the one's complement pattern is one less than the two's complement one. So add 1 when the number is negative and 0 when it is not — and "is it negative" is exactly the most significant bit. Adding *the value of the MSB* does both cases at once.
2. **Two's complement → excess-2^{n-1}.** Adding the bias 2^{n-1} modulo 2^n is the same thing as flipping the most significant bit, because 2^{n-1} is the MSB's weight.

Doing those in order gives (c). Worked on $n = 4$, with the pattern 1000 (one's complement -7): add the MSB, $1000 + 1 = 1001$ (two's complement -7); flip the MSB, $0001 = 1$, and $-7 + 8 = 1$.
Option (b) has the same two steps in the other order, and this is where the word *any* in the question earns its underline. Read literally, (b) flips the MSB first, so "the value of the most-significant bit" it then adds is the *flipped* bit — the opposite of what is wanted. Only (c) reads the MSB while it still means "is this number negative".
Check this one. If you read the "value of the most-significant bit" in (b) as the value it had *before* the flip, then (b) and (c) describe the same operation and both are correct — flipping the MSB is addition of 2^{n-1} modulo 2^n, which commutes with adding 1. The paper can only have one answer, so the intended distinction must be the reading above, but it rests on wording rather than on arithmetic.

courses/csse2010/binary-number-representations.pdf

Q5 — Reading a gate circuit

What function is implemented by the following circuit?

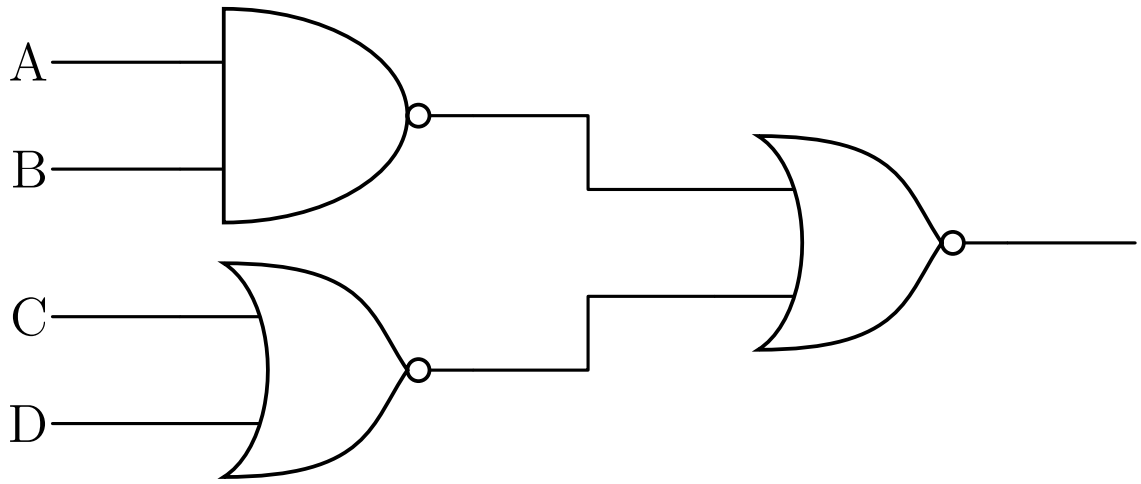


Figure 37: Question 5's circuit: a NAND on A and B, a NOR on C and D, and a NOR combining the two.

(a) $(\overline{A.B}) + (\overline{C+D})$

(b) $\overline{(\overline{A.B}) + (\overline{C+D})}$

(c) $\overline{(\bar{A}.\bar{B}) + (\overline{C+D})}$

(d) $(\overline{A.B}) \oplus (\overline{C+D})$

(e) $(\overline{A.B}) \oplus (C \oplus D)$

Answer

(b)
Name the two first-stage outputs and then wrap them in the output gate. The top gate is an AND with a bubble, so it is a NAND: $P = \overline{A \cdot B}$. The bottom gate is an OR with a bubble, so it is a NOR: $Q = \overline{C + D}$. The output gate is an OR with a bubble, so it is a NOR of those two:

$$X = \overline{P + Q} = \overline{(\overline{A.B}) + (\overline{C+D})}$$

which is (b) verbatim. Option (a) is the same expression without the final inversion — the trap for anyone who reads the last gate as a plain OR.

Simplifying confirms it is not any of the others: De Morgan turns $X = \overline{P + Q}$ into $\overline{P} \cdot \overline{Q}$, and $\overline{P}$ and $\overline{Q}$ undo the two first-stage bubbles, so $X = (A \cdot B) \cdot (C + D)$. The circuit is really "A and B, and at least one of C or D". See logic-gates[108] for the bubble-means-invert convention and boolean-algebra[109] for De Morgan.

Q6 — Circuit for an XOR function

Which of the following circuits implements the function $A \oplus (B + C) \oplus D$

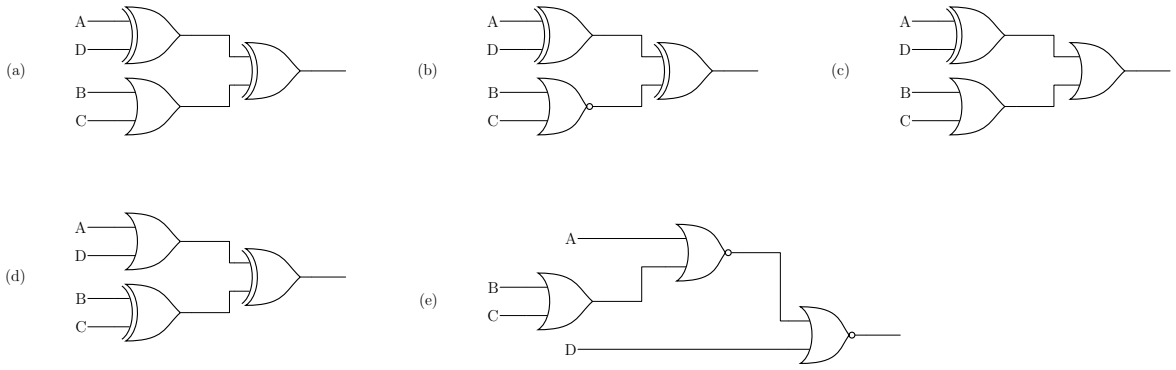


Figure 38: Question 6's five candidate circuits, laid out (a) to (e) as they appear on the paper.

Answer

(a)
XOR is both commutative and associative, so the target function can be regrouped freely:

$$A \oplus (B + C) \oplus D = (A \oplus D) \oplus (B + C)$$

That is exactly circuit (a): an XOR on A and D, an OR on B and C, and an XOR joining them.
The others each break one gate. (b) replaces the OR with a NOR, giving $(A \oplus D) \oplus \overline{B + C}$ — the complement of the right answer, since inverting one input of an XOR inverts its output. (c) uses an OR as the final gate rather than an XOR. (d) swaps the two first-stage gates, giving $(A + D) \oplus (B \oplus C)$. (e) is built from NORs and computes $\overline{\overline{A + (B + C)} + D}$, which is not an XOR of anything. See logic-gates[110].

109 courses/csse2010/logic-gates.pdf
109 courses/csse2010/boolean-algebra.pdf
110

Q7 — Boolean simplification

Which of the following logic functions is equivalent to $\overline{(A + B.\bar{C}).(\overline{\bar{A} + B})}$

(a) $\bar{A} + B$

(b) $\bar{A} + C + B$

(c) $A.\bar{B}$

(d) $\bar{A}.C + B$

(e) None of the above

Answer

(a)
Write $P = A + B\bar{C}$ and $Q = \bar{A} + B$, so the expression is $\overline{P \cdot Q}$. De Morgan turns that into a sum:

$$\overline{P \cdot Q} = \bar{P} + \bar{\bar{Q}} = \bar{P} + Q$$

Now expand $\bar{P}$ with De Morgan again:

$$\bar{P} = \overline{A + B\bar{C}} = \bar{A} \cdot \overline{B\bar{C}} = \bar{A}(\bar{B} + C)$$

so

$$\bar{P} + Q = \bar{A}\bar{B} + \bar{A}C + \bar{A} + B$$

Every one of the first two terms contains $\bar{A}$, so absorption collapses them into the standalone $\bar{A}$:

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

leaving $\bar{A} + B$, which is (a).
Sanity check without any algebra: the whole expression is $\bar{P} + Q$, and $Q = \bar{A} + B$ is false only when $A = 1, B = 0$. In that case $P = 1 + 0 = 1$, so $\bar{P} = 0$ and the expression is 0 too. Whenever Q is true the expression is true. So the expression equals $Q = \bar{A} + B$ exactly. See boolean-algebra[111].

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

courses/csse2010/boolean-algebra.pdf

Q8 — Truth table for an XOR chain

What is the truth table corresponding to $= \bar{A} \oplus \bar{B} \oplus \bar{C}$?

Table 1: Question 8's five candidate truth tables, transcribed one under the other. Each option is its own eight-row table on the paper.

	A	B	C	X
(a)	0	0	0	0
	0	0	1	1
	0	1	0	1
	0	1	1	0
	1	0	0	1
	1	0	1	0
	1	1	0	0
	1	1	1	1
(b)	0	0	0	0
	0	0	1	1
	0	1	0	1
	0	1	1	0
	1	0	0	1
	1	0	1	0
	1	1	0	0
	1	1	1	0
(c)	0	0	0	0
	0	0	1	0
	0	1	0	0
	0	1	1	1
	1	0	0	0
	1	0	1	1
	1	1	0	1
	1	1	1	0
(d)	0	0	0	1
	0	0	1	0
	0	1	0	0
	0	1	1	0
	1	0	0	0
	1	0	1	0
	1	1	0	0
	1	1	1	1
(e)	0	0	0	1
	0	0	1	0

	A	B	C	X
	0	1	0	0
	0	1	1	1
	1	0	0	0
	1	0	1	1
	1	1	0	1
	1	1	1	0

Answer

(e)
XOR has a useful property: inverting *two* of its inputs leaves the result unchanged, because the two inversions cancel. Inverting an odd number of inputs inverts the output. Here all three inputs are inverted, so

$$\bar{A} \oplus \bar{B} \oplus \bar{C} = \overline{A \oplus B \oplus C}$$

and $A \oplus B \oplus C$ is the odd-parity function — 1 when an odd number of the inputs is 1. Its complement is even parity, so $X = 1$ exactly when the row contains an even number of 1s:

A B C	number of 1s	X
0 0 0	0	1
0 0 1	1	0
0 1 0	1	0
0 1 1	2	1
1 0 0	1	0
1 0 1	2	1
1 1 0	2	1
1 1 1	3	0

That column is `1 0 0 1 0 1 1 0`, which is option (e). Option (a) is the exact complement of it — odd parity, i.e. the answer you get if you forget that three inversions do not cancel. The paper writes the question as "corresponding to $= \bar{A} \oplus \bar{B} \oplus \bar{C}$", with a stray equals sign; the function is clearly meant to be $X = \bar{A} \oplus \bar{B} \oplus \bar{C}$. See logic-gates[112].

Q9 — Sum of products from a table

Consider the logic function represented by the following truth table:

112 courses/csse2010/logic-gates.pdf

A	B	C	X
0	0	0	0
0	0	1	1
0	1	0	1
0	1	1	1
1	0	0	0
1	0	1	1
1	1	0	0
1	1	1	1

Which of the following expressions is equivalent to the function represented by this table?

(a) $\bar{A}.B + A.C$

(b) $A.C + (B + C)$

(c) $\bar{A}.B + C$

(d) $\bar{A}.B + B.C + \bar{A}.C$

(e) None of the above

Answer

(c)
Read off the rows where $X = 1$: `001`, `010`, `011`, `101`, `111`. As a sum of products,

$$X = \bar{A}\bar{B}C + \bar{A}B\bar{C} + \bar{A}BC + A\bar{B}C + ABC$$

Four of those five minterms — `001`, `011`, `101`, `111` — are every row with $C = 1$, and together they simplify to just C. That leaves `010`, i.e. $\bar{A}B\bar{C}$:

$$X = C + \bar{A}B\bar{C}$$

and $C + \bar{A}B\bar{C} = C + \bar{A}B$ by absorption (adding $\bar{A}B$ where C is already 1 changes nothing). So $X = \bar{A}.B + C$, which is (c).
Checking (c) against all eight rows: it is 1 whenever $C = 1$ (four rows) and, among the $C = 0$ rows, only for $A = 0, B = 1$ — exactly the five rows above. See boolean-algebra[113].

113 courses/csse2010/boolean-algebra.pdf

Q10 — XOR identities

Which of the following logic functions is identical to $\bar{A} \oplus \bar{B}$?

(a) $A.\bar{B} + \bar{A}.B$

(b) $\bar{A}.\bar{B} + \bar{B}.\bar{A}$

(c) $\bar{A} + \bar{B}$

(d) $\overline{A \oplus B}$

(e) None of the above

Answer

(a)
Inverting both inputs of a two-input XOR leaves it unchanged — the two inversions cancel, as in Q8:

$$\bar{A} \oplus \bar{B} = A \oplus B$$

and the definition of XOR is $A \oplus B = A\bar{B} + \bar{A}B$, which is (a) as written.
The others: (b) is $\bar{A}\bar{B} + \bar{A}\bar{B} = \bar{A}\bar{B}$, i.e. NOR, since the two product terms are the same term written in the opposite order. (c) is NAND, by De Morgan. (d) is XNOR — the complement of the right answer, and the trap for treating the two bars as if they inverted the result once rather than twice. See boolean-algebra[114].

Q11 — NAND as a universal gate

Which of the following implements a 3-input NAND gate ($\overline{A.B.C}$) using only 2-input NAND gates?

Answer

(b)
A NAND with both inputs tied together is an inverter, since $\overline{X \cdot X} = \bar{X}$. That is the only extra building block needed, because the target $\overline{ABC}$ is "AND the three, then invert", and a NAND followed by an inverter is an AND.
Working through (b): the first NAND gives $\overline{AB}$; the second has its inputs tied, so it inverts that back to AB; the third NANDs AB with C:

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

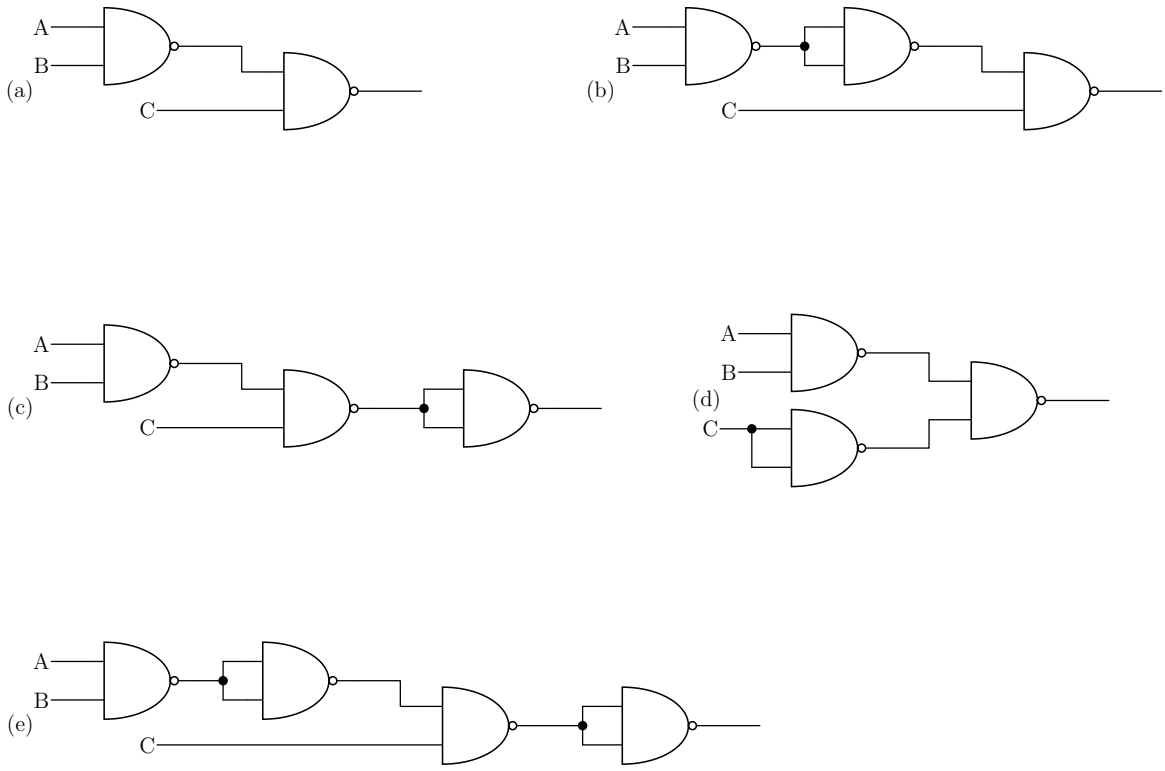


Figure 39: Question 11's five candidate NAND-only circuits. Where a NAND's two inputs are tied together it is acting as an inverter.

$$\overline{(A \cdot B) \cdot C} = \overline{A.B.C} \checkmark$$

The others:

- **(a)** $\overline{\overline{AB} \cdot C}$ — no inverter between the stages, so the first NAND's inversion is still in the expression.
- **(c)** inverts at the *end* instead: $\overline{\overline{\overline{AB} \cdot C}} = \overline{AB} \cdot C$.
- **(d)** inverts C rather than the first NAND's output: $\overline{\overline{AB} \cdot \bar{C}}$.
- **(e)** is (b) with one more inverter tacked on, so it produces $A.B.C$ — the AND, not the NAND.

See logic-gates[115] for why NAND is a universal (complete) gate.

Q12 — Multiplexer as function generator

Consider the following multiplexer. What must the inputs A,B,C,D be so that the multiplexer implements the function $X = S_0 + (S_1 \oplus G)$?

(a) A=1 B=G C=1 D=$\bar{G}$

(b) A=G B=$\bar{G}$ C=1 D=1

(c) A=$\bar{G}$ B=1 C=G D=1

(d) A=G B=1 C=$\bar{G}$ D=1

(e) None of the above

Answer

(d)

A 4-to-1 multiplexer passes input S_1S_0 through to the output, so each data input only has to be right for its own select combination. Substitute each combination into the target function and see what is left, treating G as an ordinary variable rather than a select line:

S_1	S_0	selected input	$X = S_0 + (S_1 \oplus G)$	so the input must be
0	0	A	$0 + (0 \oplus G) = G$	G
0	1	B	$1 + \ldots = 1$	1
1	0	C	$0 + (1 \oplus G) = \bar{G}$	$\bar{G}$

115 courses/csse2010/logic-gates.pdf

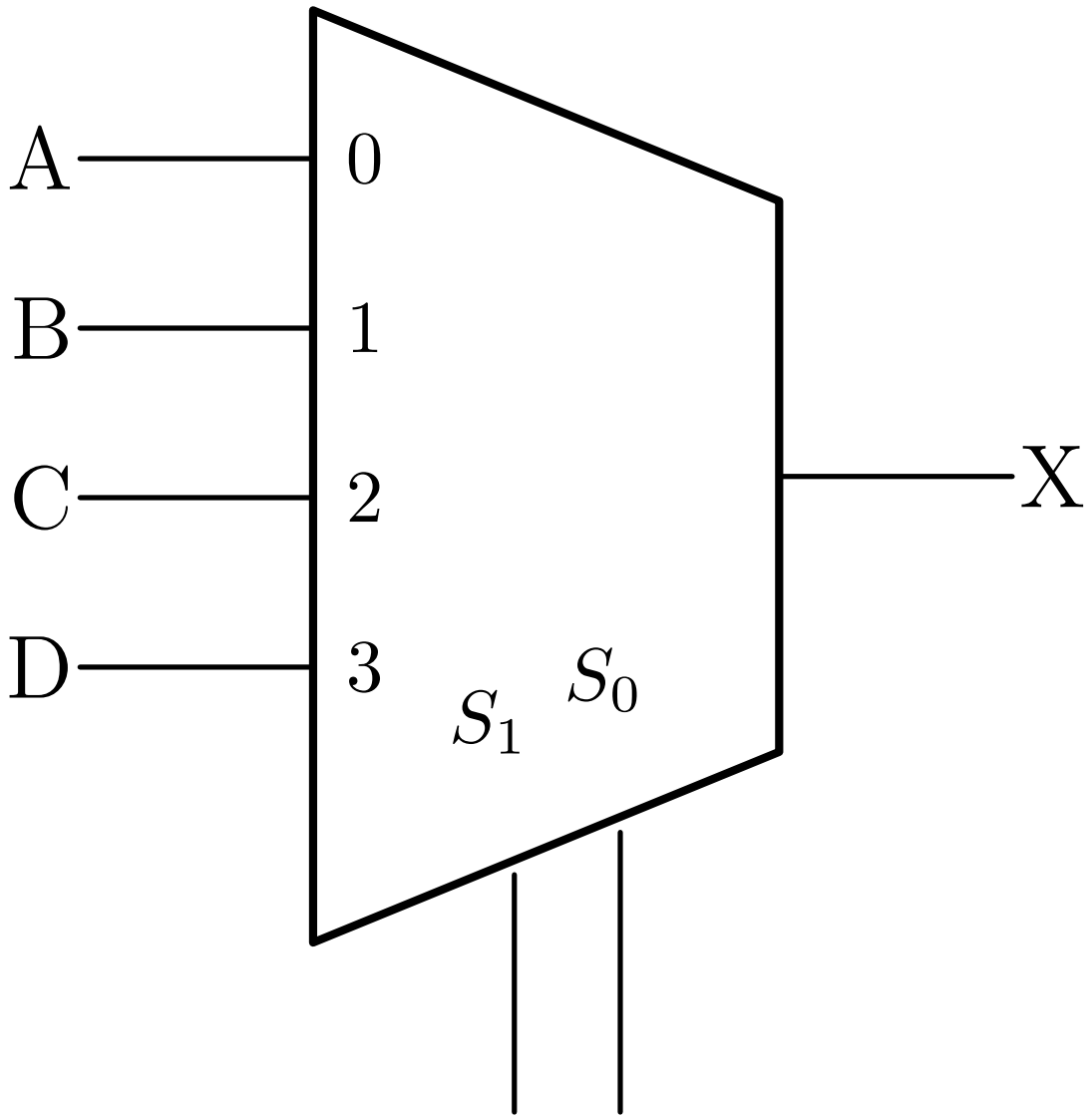


Figure 40: Question 12's 4-to-1 multiplexer: data inputs A, B, C, D on lines 0 to 3, select lines S_1 and S_0, and output X.

1	1	D	$1 + \ldots = 1$	1

So A=G, B=1, C=$\bar{G}$, D=1 — option (d). The two rows with $S_0 = 1$ are constants because the OR is already satisfied, and the two with $S_0 = 0$ reduce to $S_1 \oplus G$ evaluated at that value of S_1. See combinational-logic-blocks[116].

Q13 — Two's complement addition

What is the result of adding the 6-bit two's complement numbers 101111 and 101001?

(a) 010000

(b) 010111

(c) 011000

(d) 111000

(e) None of the above

Answer

(c)
Addition in two's complement is plain binary addition with the carry out of the top bit discarded:

$$\begin{array}{r} 101111 \\ +\ 101001 \\ \hline 1\,011000 \end{array}$$

Column by column from the right: $1+1=0$ carry 1; $1+0+1=0$ carry 1; $1+0+1=0$ carry 1; $1+1+1=1$ carry 1; $0+0+1=1$; $1+1=0$ carry 1 out of the word. Keeping six bits gives **011000**.
It is worth noticing that this result is wrong as arithmetic: 101111 is -17 and 101001 is -23, so the true sum is -40, which does not fit in the range -32 to 31. Two negatives have produced a positive result ($011000 = +24$), which is the overflow signature. The question asks only what the adder produces, and that is (c). See binary-number-representations[117].

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

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

Q14 — D flip-flop timing

Consider the flip-flop circuit shown below and the associated CLK and output (Q) signals. Which timing diagram shows input (A) values over time which would have resulted in the given output (Q).

Answer

(b)
The flip-flop is positive-edge triggered (the wedge on the clock input, with no bubble), so Q takes the value A had just before each **rising** edge of CLK, and holds it until the next one. There are four rising edges. Reading Q:

rising edge	Q before	Q after	so A must have been
1st	1	0	0
2nd	0	1	1
3rd	1	1	1
4th	1	0	0

So the required pattern of A, sampled at the four rising edges, is **0, 1, 1, 0**. Checking each candidate at those four instants:

- **(a)** 0, 1, 0, — : fails at the third edge, where Q stays high but A has just gone low. Note also that (a) changes exactly *on* the clock edges, which is what a real design goes out of its way to avoid.
- **(b)** 0, 1, 1, 0 — and every one of its transitions sits comfortably away from an edge, so there is no ambiguity about which value is sampled.
- **(c)** 0, 1, 1, 1 : fails at the fourth edge, where Q falls but A is still high.
- **(d)** 0, 1, 0, 1 : fails at the third and fourth edges.
- **(e)** 0, 1, 1, 1 : the single wide pulse is still high at the fourth edge, so Q would not fall.

The third edge is the interesting one — Q does not change there, which does not mean A was ignored; it means A happened to equal the value already stored. See flip-flops-and-latches[118] and timing-diagrams[119].

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

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

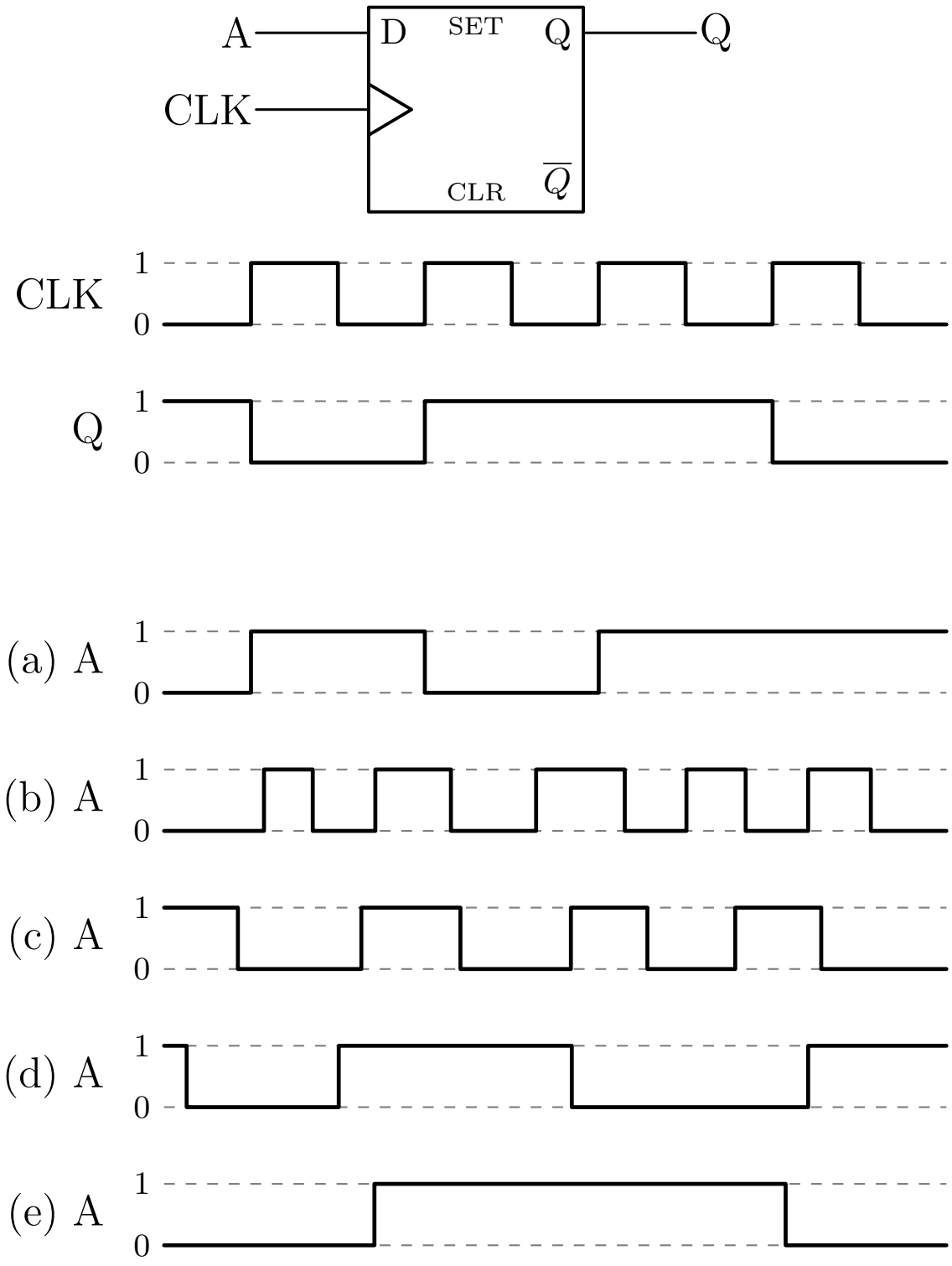


Figure 41: Question 14's positive-edge-triggered D flip-flop, the given CLK and Q waveforms, and the five candidate A waveforms.

Q15 — NOR SR latch behaviour

Which truth table correctly captures the behaviour of the following latch circuit?

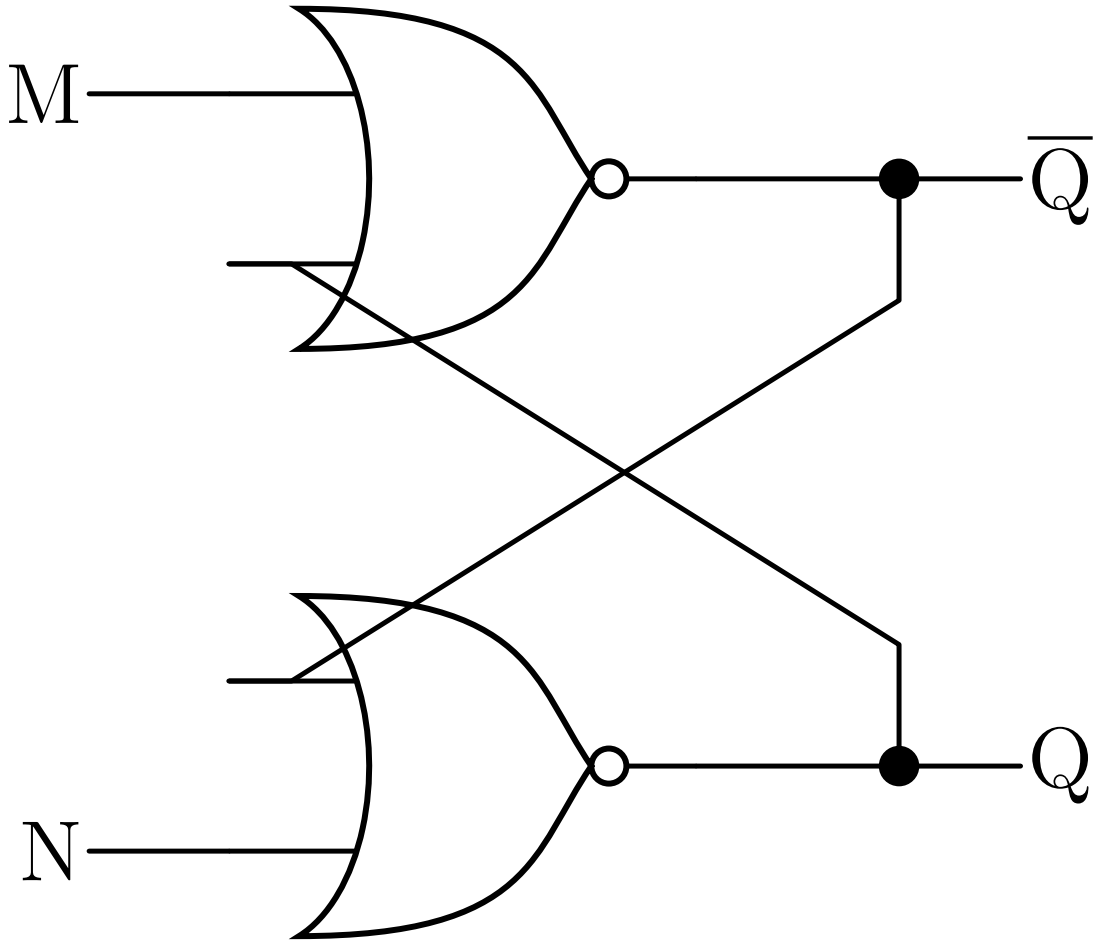


Figure 42: Question 15's cross-coupled NOR latch. The top gate takes M and the Q feedback and drives $\overline{Q}$; the bottom gate takes N and the $\overline{Q}$ feedback and drives Q.

Table 2: Question 15's four candidate truth tables, transcribed one under the other. Each option is its own four-row table on the paper.

	M	N	Q	$\overline{Q}$
(a)	0	0	Q	$\overline{Q}$
	0	1	1	0

	M	N	Q	$\overline{Q}$
	1	0	0	1
	1	1	0	0
(b)	0	0	Q	$\overline{Q}$
	0	1	0	1
	1	0	1	0
	1	1	1	1
(c)	0	0	Q	$\overline{Q}$
	0	1	0	1
	1	0	1	0
	1	1	0	0
(d)	0	0	0	0
	0	1	1	0
	1	0	0	1
	1	1	Q	$\overline{Q}$

(e) None of the above

Answer

(c)
This is the standard cross-coupled NOR SR latch, with M in the set position and N in the reset position. A NOR outputs 1 only when *both* its inputs are 0, so any input driven to 1 forces that gate's output to 0 regardless of the feedback — which is what makes the analysis easy.

- **M=1, N=0.** The top NOR sees a 1, so $\overline{Q} = 0$. The bottom NOR then sees 0 and 0, so $Q = 1$. **Set.**
- **M=0, N=1.** The bottom NOR sees a 1, so $Q = 0$. The top NOR then sees 0 and 0, so $\overline{Q} = 1$. **Reset.**
- **M=0, N=0.** Neither gate is forced. Each output feeds the other's input, so whichever state the latch was already in satisfies both gates and persists. **Hold.**
- **M=1, N=1.** Both gates are forced, so $Q = 0$ *and* $\overline{Q} = 0$. The two outputs are no longer complementary — this is the forbidden combination, and it is why the pair of names Q and $\overline{Q}$ stops being honest here.

That gives hold, reset, set, 0 0 — option (c). Option (a) has set and reset swapped; (b) puts 1 1 in the forbidden row, which is what a NAND latch would do; (d) puts the hold row in the wrong place entirely. See flip-flops-and-latches[120].

120 courses/csse2010/flip-flops-and-latches.pdf

Q16 — Shift register with function select

Which of the following statements about this shift register circuit is false?

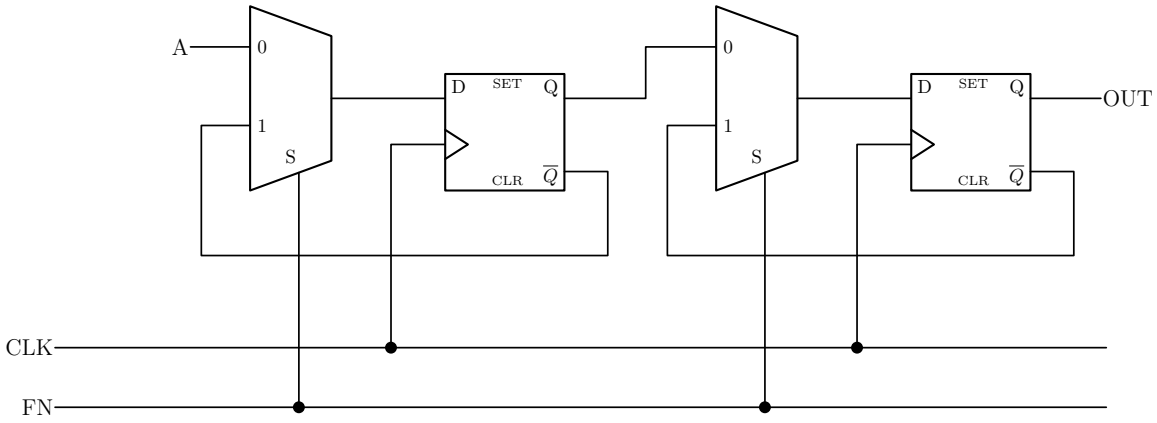


Figure 43: Question 16's two-stage register. Each stage is a 2-to-1 multiplexer feeding a D flip-flop; input 1 of each multiplexer is that stage's own $\overline{Q}$, so FN=1 makes both flip-flops toggle.

(a) If FN is 0, the value of A will be transferred to OUT after two rising clock edges

(b) If FN is 1, OUT will toggle on each rising clock edge

(c) If FN is 0, data is shifted to the right on each rising clock edge

(d) If FN is 1, the value of A is irrelevant

(e) All of the above are true

Answer

(e)
Trace what FN selects. Each multiplexer's input 0 is the data coming from the left (A for the first stage, the first flip-flop's Q for the second), and each input 1 is a feedback wire from that same stage's own $\overline{Q}$. FN drives both select lines together, so:

- **FN = 0** — every flip-flop's D is the value to its left: $D_1 = A$ and $D_2 = Q_1$. That is a two-bit shift register.
- **FN = 1** — every flip-flop's D is its own $\overline{Q}$, so each one inverts itself on every rising edge. That is a toggle, and the A input is disconnected from the datapath entirely.

Now each statement:

- **(a)** With FN=0, A is clocked into the first flip-flop on one edge and out to OUT on

the next — two rising edges. **True.**

- **(b)** With FN=1, $D_2 = \overline{Q_2}$, so OUT toggles every rising edge. **True.**
- **(c)** Data moves from A through the first stage to the second, which is left-to-right as drawn. **True.**
- **(d)** With FN=1 both multiplexers select input 1, so A reaches nothing. **True.**

Nothing among (a) to (d) is false, so the answer is the option that says so: (e). See shift-registers[121].

The wording is a little awkward — the question asks which statement is false, and (e) is not itself a candidate false statement but a summary of the other four. Selecting it is the paper's way of saying "none of them is false".

Q17 — Synchronous counter sequence

What sequence will the following synchronous counter count through, if it starts at $Q_1Q_0 = 00$?

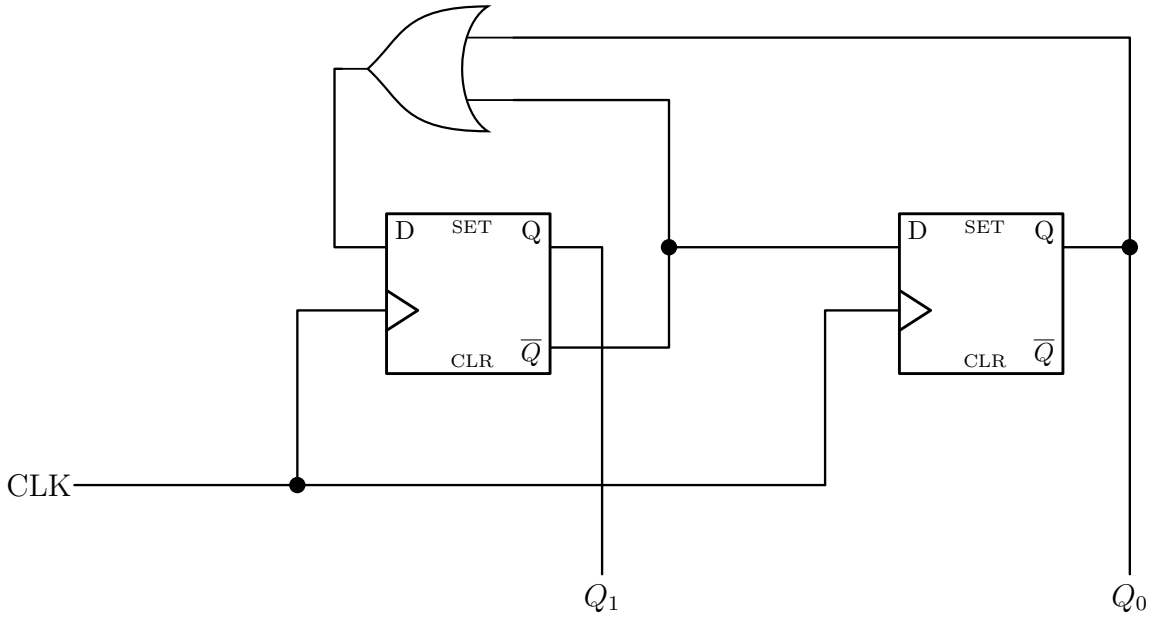


Figure 44: Question 17's two-flip-flop synchronous counter. The OR gate is drawn mirrored, as on the paper: its output faces left into the first flip-flop's D input, and its two inputs are fed from Q_0 and $\overline{Q_1}$.

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

(a) Q_1Q_0: $00 \to 01 \to 11 \to 10 \to 10 \to \ldots$

(b) Q_1Q_0: $00 \to 01 \to 11 \to 00 \to \ldots$

(c) Q_1Q_0: $00 \to 11 \to 01 \to 00 \to \ldots$

(d) Q_1Q_0: $00 \to 11 \to 10 \to 00 \to \ldots$

(e) None of the above

Answer

(d)

First read the excitation equations off the schematic. The left flip-flop holds Q_1 (its Q output runs down to the Q_1 label) and the right one holds Q_0. The left flip-flop's $\overline{Q}$ output is the wire that goes right and then splits: it feeds the second flip-flop's D input and, going up, one input of the OR gate. The OR gate's other input comes from the long wire across the top, which runs to the Q_0 node. The OR gate's output goes back left into the first flip-flop's D. So:

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

Both flip-flops share one clock, so both D values are computed from the *present* state and land together on the edge. Tabulating:

Q_1Q_0	$\overline{Q_1}$	$D_1 = Q_0 + \overline{Q_1}$	$D_0 = \overline{Q_1}$	next Q_1Q_0
0 0	1	1	1	**1 1**
1 1	0	1	0	**1 0**
1 0	0	0	0	**0 0**
0 1	1	1	1	1 1

Starting from 00 the sequence is $00 \to 11 \to 10 \to 00 \to \ldots$, a three-state cycle — option (d). The fourth row shows the unused state 01 drops into the cycle on the next edge, so the counter cannot lock up outside it.

The mirrored OR gate is the thing to get right when reading the paper's drawing: the single wire on its pointed left-hand side is the output, and the two wires on its curved right-hand side are the inputs. See counters[122] and sequential-circuits[123].

[123]

courses/csse2010/counters.pdf

[123]

courses/csse2010/sequential-circuits.pdf

Q18 — Overflow detection

Which of the following operations will result in an overflow in 4-bit two's complement arithmetic?

(a) 0111+0111

(b) 1111+1111

(c) 1010+0110

(d) All of the above result in overflow

(e) None of the above

Answer

(a)
The rule for two's complement is that overflow happens only when both operands have the **same** sign and the result has the other one. Adding numbers of opposite signs can never overflow, because the result lies between them. Four bits cover -8 to $+7$.

- **(a)** $0111 + 0111 = 7 + 7 = 14$. In binary $0111 + 0111 = 1110$, which reads as -2. Two positives gave a negative: **overflow**.
- **(b)** $1111 + 1111 = (-1) + (-1) = -2$. In binary $1111 + 1111 = 1\,1110$; discarding the carry out leaves $1110 = -2$. Two negatives gave a negative, and -2 is in range: **no overflow**. The carry out of the top bit is not itself overflow.
- **(c)** $1010 + 0110 = (-6) + 6 = 0$. In binary $1010 + 0110 = 1\,0000 \to 0000$. Opposite signs, so it cannot overflow: **no overflow**.

Only (a) overflows, so (d) and (e) are both out. See binary-number-representations[124], and Beyond the paper below for the gate circuit that decides this same question from the sign bits alone.

Q19 — Overflow in a signed addition

Consider the binary addition operation 1011 + 1111. Which of the following statements is true if the numbers are interpreted as two's complement?

(a) The result is negative and no overflow occurs

(b) The result should be negative but overflow occurs

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

(c) The result is positive and no overflow occurs

(d) The result should be positive but overflow occurs

(e) The result is zero

Answer

(a)
Decode both operands. 1011: invert and add one gives $0100+1 = 0101 = 5$, so $1011 = -5$. 1111: invert and add one gives $0000 + 1 = 0001$, so $1111 = -1$. The true sum is -6, which is comfortably inside the 4-bit range -8 to $+7$, so no overflow is possible.
Doing the addition:

$$\begin{array}{r} 1011 \\ +\ 1111 \\ \hline 1\,1010 \end{array}$$

Discarding the carry out leaves 1010, which decodes as $-(0101 + 1) = -6$ — negative, correct, and matching the sign of both operands, which is the overflow test passing.
The carry out of the most significant bit is 1 here, and that is the trap: in two's complement a carry out is normal and is simply discarded. Overflow is a *sign* condition, not a carry condition. See binary-number-representations[125] — and Beyond the paper below, where the carry-based rule $V = C_3 \oplus C_4$ has to be rewritten because a real 4-bit adder chip never brings C_3 out to a pin.

Q20 — State table to state diagram

Consider the following state table:

Current State	Next State (A=0)	Next State (A=1)	Outputs (X,Y)
SA	SA	SB	0 0
SB	SA	SC	0 0
SC	SA	SD	0 1
SD	SD	SE	1 0
SE	SA	SE	1 1

Which of the following state diagrams is an equivalent representation of this state table? (Input: A, Outputs: XY)

125 courses/csse2010/binary-number-representations.pdf

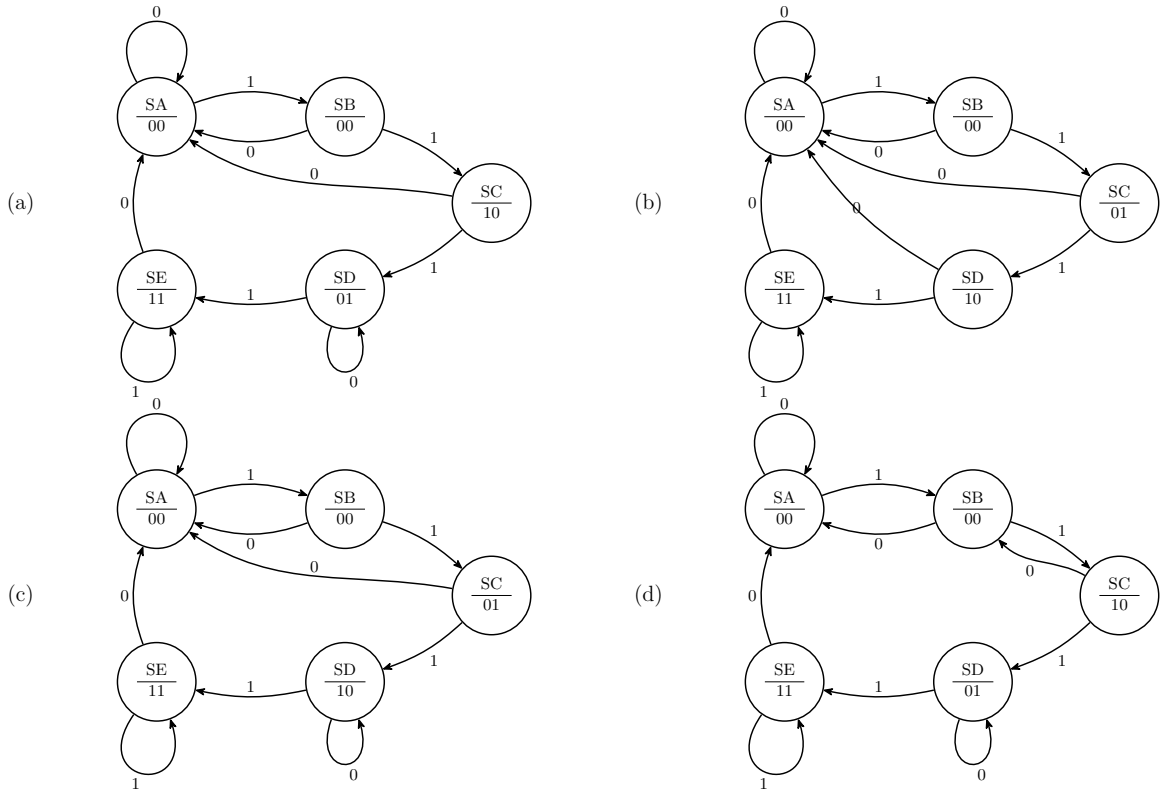


Figure 45: Question 20's four candidate state diagrams, laid out (a) to (d) as on the paper. Option (e) is None of the above.

(e) None of the above

Answer

(c)
Check the two things the table specifies — the output written under each state name, and where each of the two arrows out of each state goes.
Outputs. The table gives SA=00, SB=00, SC=**01**, SD=**10**, SE=11. Options (a) and (d) both label SC as 10 and SD as 01, i.e. the two are swapped, so both are out immediately. That leaves (b) and (c), which have the outputs right.
Transitions. The table's five rows are: SA self-loops on 0 and goes to SB on 1; SB returns to SA on 0 and goes to SC on 1; SC returns to SA on 0 and goes to SD on 1; SD **self-loops on 0** and goes to SE on 1; SE returns to SA on 0 and self-loops on 1.
Diagram (b) has SD's 0-transition going back to SA rather than staying in SD, which contradicts the SD row. Diagram (c) has the self-loop on SD, and matches every other row as well, so (c) is the answer.
Worth checking completeness while you are there: each state in (c) has exactly two arrows out, one labelled 0 and one labelled 1, which is what a one-input machine needs — no missing case and no ambiguity. See state-machines[126].

Q21 — Flip-flops needed for a machine

Consider the state machine represented by the state table in question 20 above (see Q20). What is the minimum number of flip-flops needed to implement this state machine?

(a) 2

(b) 3

(c) 4

(d) 5

(e) 7

Answer

(b)
N flip-flops hold N bits, so they can encode at most 2^N distinct states. The machine has five states (SA to SE), so the requirement is

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

$$2^N \geq 5 \quad \Rightarrow \quad N \geq \log_2 5 = 2.32\ldots \quad \Rightarrow \quad N = \lceil \log_2 5 \rceil = 3$$

Two flip-flops give only four codes, one short. Three give eight, so five states fit with three codes left unused.

The distractors are the two other plausible counting rules: (d) 5 is one flip-flop per state, which is the **one-hot** encoding — legal, and often the one that makes the combinational logic simplest, but not the *minimum*. (c) 4 and (e) 7 do not correspond to anything. See state-machines[127] for the encoding choices.

Q22 — Tracing a state machine

Consider the state machine represented by the state diagram below:

Inputs: A
Outputs: X,Y

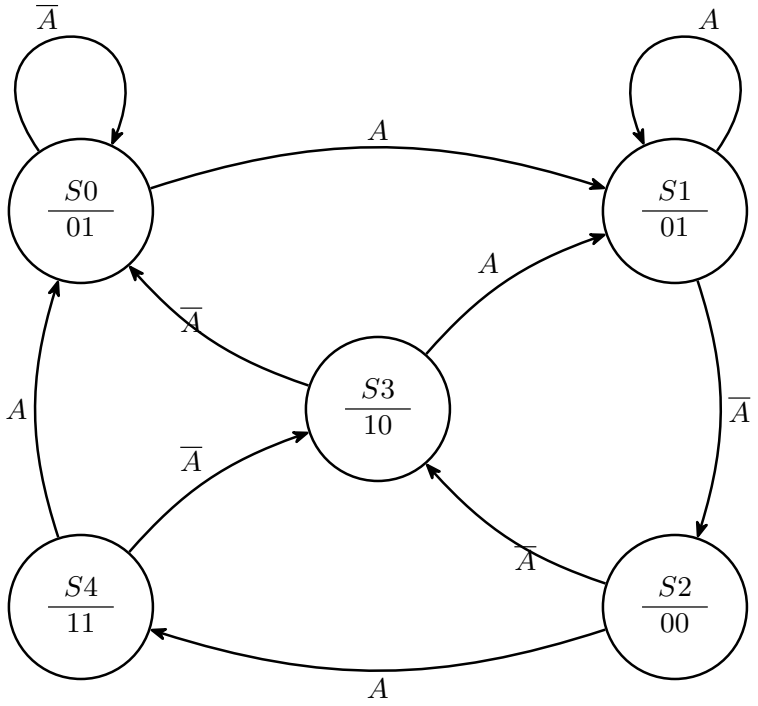


Figure 46: Question 22's state machine: one input A, outputs X and Y, and five states each with an A and an $\overline{A}$ transition.

What sequence of states is traversed if the machine starts in state S0 and A goes through the sequence 11110000 (i.e. four 1's followed by four 0's)?

127 courses/csse2010/state-machines.pdf

(a) S0→S1→S1→S1→S1→S2→S3→S0→S0

(b) S0→S1→S3→S1→S3→S0→S0→S0→S0

(c) S0→S1→S1→S1→S2→S3→S2→S0→S0

(d) S0→S1→S1→S1→S1→S2→S4→S3→S0

(e) None of the above

Answer

(a)
Read the transitions off the diagram first:

State	on $\overline{A}$ (0)	on A (1)
S0	S0 (self-loop)	S1
S1	S2	S1 (self-loop)
S2	S3	S4
S3	S0	S1
S4	S3	S0

Now walk the eight inputs from S0, one clock edge per input:

step	input	from	to
1	1	S0	S1
2	1	S1	S1
3	1	S1	S1
4	1	S1	S1
5	0	S1	S2
6	0	S2	S3
7	0	S3	S0
8	0	S0	S0

Including the starting state, that is S0→S1→S1→S1→S1→S2→S3→S0→S0 — nine entries for a start state plus eight inputs, which is option (a). Note that each of the listed options has nine entries, so the count itself is not a discriminator here; the run of self-loops on S1 while A stays at 1, and the fact that S0 self-loops once the input has gone to 0 and the machine has come home, are what pick (a) out. See state-machines[128].

128 courses/csse2010/state-machines.pdf

Q23 — ALU control for NOR

Consider the ALU bit slice below (not the same as seen previously).

For an ALU made up of 8 of the bit slices above, which of the following control input settings will result in the ALU output being $\overline{A+B}$ (bitwise)? (The "Right Shift In" control input refers to that for the most significant bit; the "Carry in" control input refers to that for the least significant bit.)

(a) ENA=**1** ENB=**1** INVA=**0** F_2=**0** F_1=**0** F_0=**1** Right Shift In=**1** Carry in=**1**

(b) ENA=**1** ENB=**1** INVA=**0** F_2=**0** F_1=**1** F_0=**0** Right Shift In=**1** Carry in=**1**

(c) ENA=**1** ENB=**1** INVA=**1** F_2=**0** F_1=**0** F_0=**1** Right Shift In=**1** Carry in=**1**

(d) ENA=**1** ENB=**1** INVA=**1** F_2=**0** F_1=**1** F_0=**0** Right Shift In=**1** Carry in=**1**

(e) None of the above

Answer

(b)
Read the slice in two halves, exactly as in arithmetic-logic-unit[129].
Input conditioning. The left-hand gates produce two internal signals:

$$X = (A \cdot \text{ENA}) \oplus \text{INVA} \qquad Y = B \cdot \text{ENB}$$

These are what every function block actually sees. X is also what leaves the slice as Right Shift Out.
Function select. $F_2F_1F_0$ is the select code of the 8-to-1 multiplexer, so reading the blocks down the middle of the drawing from the top:

$F_2F_1F_0$	mux input	Output
000	0	$X \cdot Y$
001	1	$X + Y$
010	2	$\overline{X+Y}$
011	3	$X \oplus Y$
100	4	Carry In
101	5	Right Shift In
110	6	$\overline{Y}$
111	7	$X \oplus Y \oplus C_{in}$ (the sum)

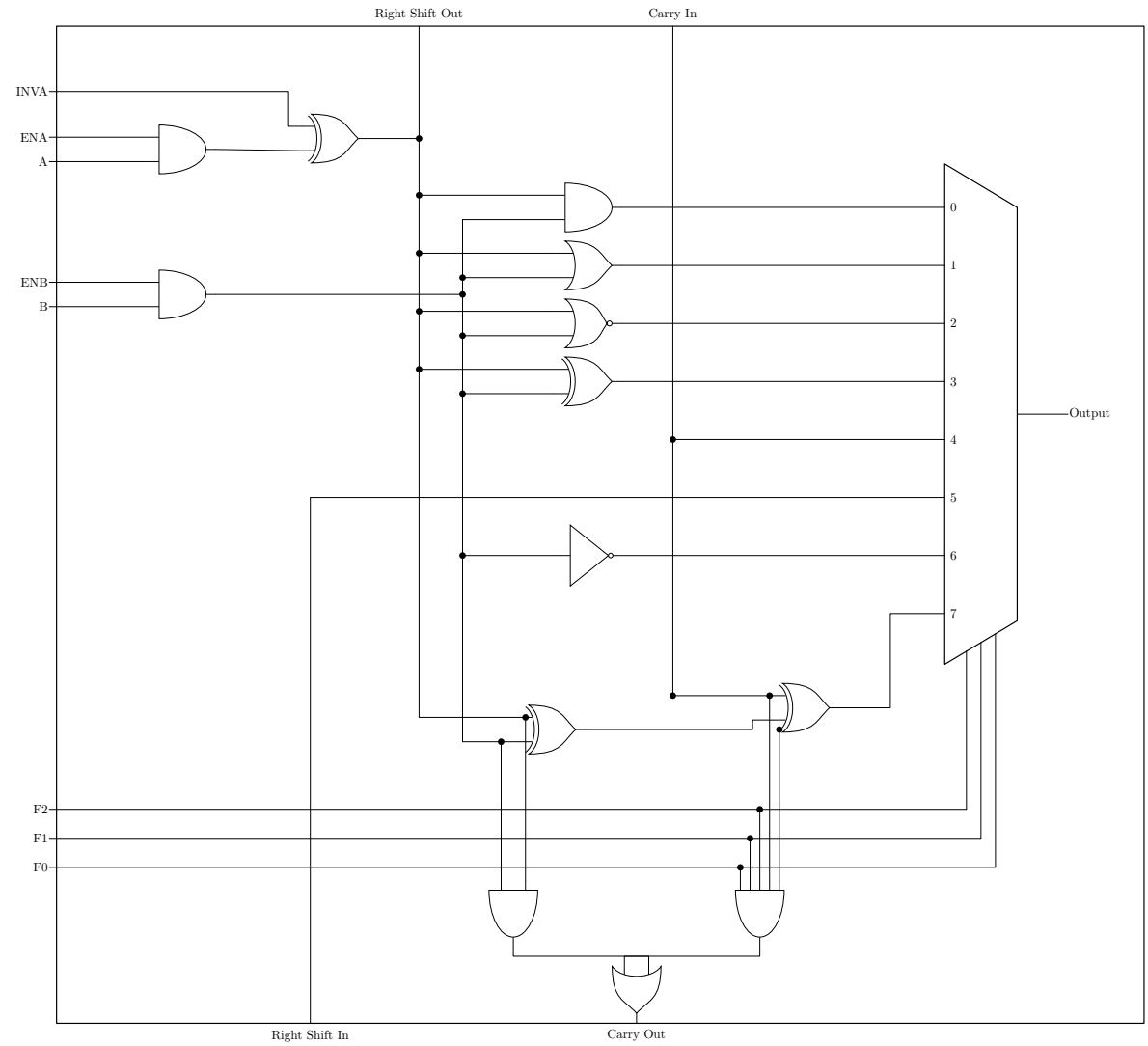


Figure 47: Question 23's ALU bit slice. The gates at the left condition A and B into the internal signals routed on the two vertical rails; the eight function blocks compute in parallel; the 8-to-1 multiplexer picks one under F2 F1 F0. Carry out is the AND of the two conditioned inputs, ORed with a propagate term gated by the function select.

The target is $\overline{A+B}$, a **bitwise** NOR. So the conditioning must pass both operands through untouched — ENA=1, INVA=0 gives $X = A$, and ENB=1 gives $Y = B$ — and the select must pick the NOR block, which is $F_2F_1F_0 = 010$.
That is option (b). Checking the others: (a) selects 001, the plain OR, giving $A + B$ rather than its complement; (c) selects 001 with INVA=1, giving $\overline{A} + B$; (d) has the right select but INVA=1, so it computes $\overline{\overline{A} + B}$. The Right Shift In and Carry in values are irrelevant once a purely logical function is selected, which is why all four options can share them.

Q24 — ALU control for NOT B

For an ALU made up of 8 of the bit slices shown in question 23 above (see Q23), which of the following control input settings will **NOT** result in the ALU output being $\overline{B}$ (bitwise)?

(a) ENA=**0** ENB=**1** INVA=**0** F_2=**0** F_1=**1** F_0=**0** Right Shift In=**0** Carry in=**0**

(b) ENA=**0** ENB=**1** INVA=**1** F_2=**0** F_1=**1** F_0=**1** Right Shift In=**0** Carry in=**0**

(c) ENA=**0** ENB=**1** INVA=**1** F_2=**1** F_1=**1** F_0=**0** Right Shift In=**0** Carry in=**0**

(d) ENA=**0** ENB=**1** INVA=**1** F_2=**1** F_1=**1** F_0=**1** Right Shift In=**0** Carry in=**0**

(e) All of the above result in the ALU output being $\overline{B}$ (bitwise)

Answer

(d)
Every option has ENA=0 and ENB=1, so from the conditioning equations in Q23:

$$X = (A \cdot 0) \oplus \text{INVA} = \text{INVA} \qquad Y = B$$

That is, the A side is forced to a constant — all zeros when INVA=0, all ones when INVA=1 — and the B side passes through. Now take each option's select code:

- **(a)** 010 selects $\overline{X+Y}$. With INVA=0, X is all zeros, and $\overline{0+B} = \overline{B}$.
- **(b)** 011 selects $X \oplus Y$. With INVA=1, X is all ones, and $1 \oplus B = \overline{B}$. This is the "XOR with all ones inverts" trick.
- **(c)** 110 selects $\overline{Y} = \overline{B}$. INVA does not matter here, because the inverter block never looks at X.
- **(d)** 111 selects the **adder**: $X \oplus Y \oplus C_{in}$, with the carry rippling from slice to slice. With X all ones and the external carry in 0, the machine computes $(-1) + B$, i.e. $B - 1$ — not $\overline{B}$.

129 courses/csse2010/arithmetic-logic-unit.pdf

So (d) is the odd one out. The distinction between (b) and (d) is the whole point of the question: mux input 3 is a *bitwise* XOR with no carry involved, so it really does produce $\overline{B}$, whereas mux input 7 is the same XOR plus the carry chain, and once carries propagate the result is arithmetic rather than bitwise. Adding all-ones to B decrements it; only $B = 0$ (and, in eight bits, $B = -128$) happens to satisfy $B - 1 = \overline{B}$.
The question does not depend on which way round the select bits go, which is a small mercy. The four codes offered are 010, 011, 110 and 111; reversing the bit order maps that set onto itself, swapping (b) and (c) with each other and leaving (a) and (d) where they are. So (d) is still the only failure if F_0 is read as the most significant bit — the answer is the same either way. See arithmetic-logic-unit[130].

Q25 — Half adder from multiplexers

Consider the half-adder shown below:

In the multiplexer circuit below, what values do the multiplexer inputs X0,X1,X2,X3 and Y0,Y1,Y2,Y3 need to take on in order for the circuit to implement a half-adder?

(Both drawings the question refers to are combined into Figure 48: the half-adder block on top, the two multiplexers underneath.)

Table 3: Question 25's four candidate settings, transcribed one per row. On the paper each option is drawn as a pair of four-cell boxes.

	X0	X1	X2	X3	Y0	Y1	Y2	Y3
(a)	0	0	0	1	0	1	1	0
(b)	0	1	1	0	0	0	1	1
(c)	0	1	1	0	0	0	0	1
(d)	0	1	1	0	1	0	0	0

(e) None of the above

Answer

(c)
A half adder is just the truth table of two bits added:

130 courses/csse2010/arithmetic-logic-unit.pdf

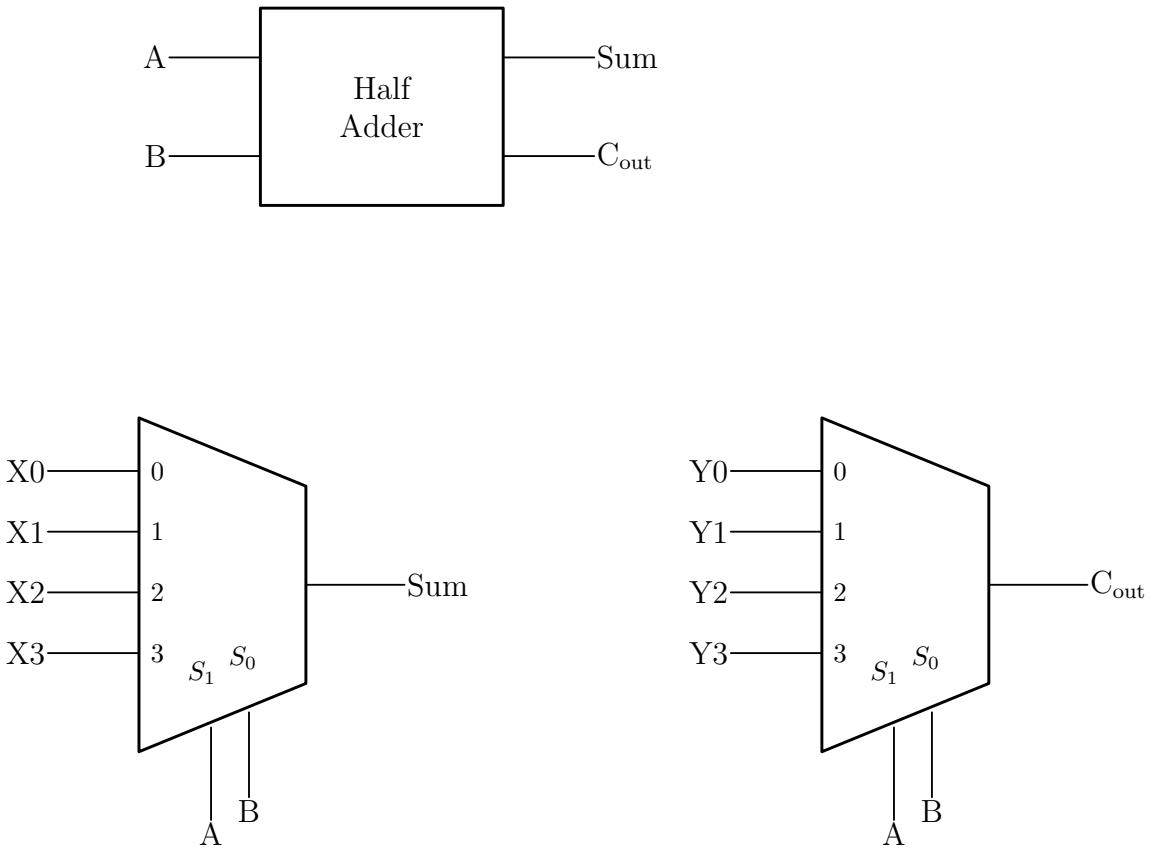


Figure 48: Question 25's half adder block and the pair of 4-to-1 multiplexers meant to implement it, one for Sum and one for C_{out}, both selected by A and B.

A	B	Sum $= A \oplus B$	$C_{out} = A \cdot B$
0	0	0	0
0	1	1	0
1	0	1	0
1	1	0	1

Both multiplexers are selected by the same pair of lines, with A on S_1 and B on S_0, so input n of each multiplexer is the one reached when AB is the two-bit number n: input 0 for $AB = 00$, input 1 for 01, input 2 for 10, input 3 for 11. Using a multiplexer this way turns it into a direct lookup of a truth table — the data inputs *are* the output column, in row order.

Reading the Sum column down gives $X_0X_1X_2X_3 = 0, 1, 1, 0$, and reading the C_{out} column gives $Y_0Y_1Y_2Y_3 = 0, 0, 0, 1$. That is option (c).

The distractors are the same two columns misassigned: (a) has them swapped between the two multiplexers, (b) has C_{out} set for $AB = 10$ as well as 11, and (d) has the C_{out} column reversed. See combinational-logic-blocks[131].

Beyond the paper — building the overflow detector

Q18 and Q19 are both answered on paper the same way: decode the operands into decimal, look at the signs, decide. **Learning Lab 3 (*Binary Arithmetic*)** asks the same question the way hardware has to answer it — with gates, on a real adder chip, where nothing is in decimal and not every carry is even visible.

> This is the extension to Task 2 of Learning Lab 3 (Binary Arithmetic). Both circuits below are derived from the constraint and checked against the truth table. week2-lab-binary-arithmetic[132] covers the rest of that lab.

The constraint the chip imposes

Task 2 of the lab builds a 4-bit adder from a **74HCT283** — a single 16-pin part holding four full adders with the intermediate carries C_1, C_2 and C_3 wired stage to stage *inside* the package, so only C_0 (carry in) and C_4 (carry out) come out on pins. See device-pinouts[133] for the pinout and symbol.

The rule Q19's answer appeals to is usually written in terms of the carries around the sign bit:

---

131 courses/csse2010/combinational-logic-blocks.pdf

132 courses/csse2010/week2-lab-binary-arithmetic.pdf

133 courses/csse2010/device-pinouts.pdf

$$V = C_3 \oplus C_4$$

— overflow exactly when the carry *into* the sign column differs from the carry *out of* it. The lab's extension task kills that formula in one line: *"you don't have access to C3 which is an internal signal. Therefore, you need the overflow output expressed in terms of available information (i.e. A3, B3, S3, C4)."*

So the detector has to be built from the sign bit of each operand, the sign bit of the result, and the carry out.

Two detectors that use only pins

(a) compares signs directly. Overflow means the two operands agreed in sign and the answer came out with the other one, which as a sum of products is

$$V = A_3 B_3 \overline{S_3} + \overline{A_3}\,\overline{B_3}\,S_3 = \overline{(A_3 \oplus B_3)} \cdot (A_3 \oplus S_3)$$

(b) rebuilds the carry the chip hid. Stage 3 is a full adder, so $S_3 = A_3 \oplus B_3 \oplus C_3$; XOR is its own inverse, so $C_3 = A_3 \oplus B_3 \oplus S_3$. Substituting into $V = C_3 \oplus C_4$:

$$V = A_3 \oplus B_3 \oplus S_3 \oplus C_4$$

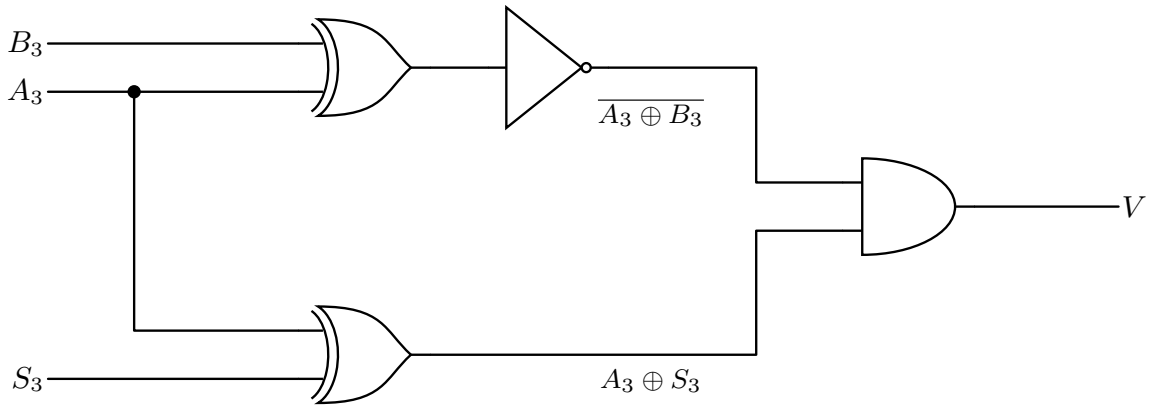
That is why C_4 is listed among the available signals: it is the only one of the four that (a) does not use, and its presence in the list is the hint that a carry-based answer exists.

Form (b) is the one to build. Its three gates are all the same type, so they come out of **one 74HCT86** with a gate spare; form (a) as drawn mixes three types and so needs three packages. (Form (a) can be squeezed onto fewer, since an XOR with one input tied to logic high is an inverter and the pinout sheet gives a symbol for logic high — but it is still four gates against three, and two chips against one.) See device-pinouts[134] for the parts and circuit-schematics[135] for what turns either of these logic diagrams into a buildable schematic.

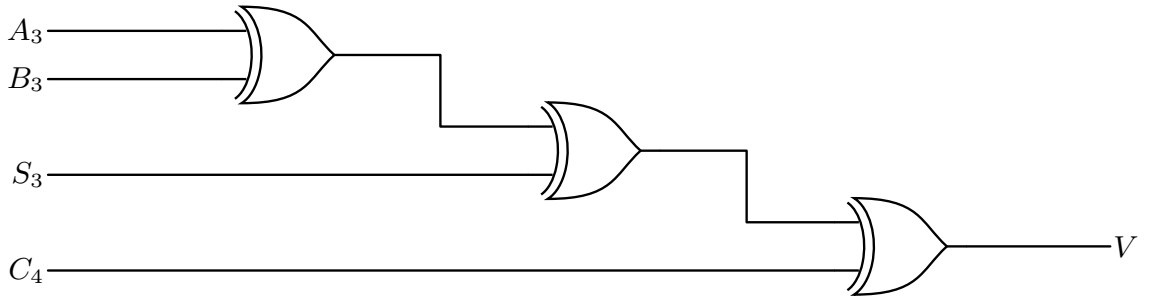
Checking both against the truth table

C_3 and C_4 below are what the top stage of the adder must have been doing for that combination of A_3, B_3 and S_3 ($C_3 = A_3 \oplus B_3 \oplus S_3$, and C_4 is the majority of A_3, B_3 and C_3):

[134] courses/csse2010/device-pinouts.pdf
[135] courses/csse2010/circuit-schematics.pdf



(a) $\quad V = \overline{(A_3 \oplus B_3)} \,.\, (A_3 \oplus S_3)$



(b) $\quad V = A_3 \oplus B_3 \oplus S_3 \oplus C_4$

Figure 49: Two gate-level answers to Lab 3's extension task, both using only signals the 74HCT283 brings out to pins. (a) compares signs directly: the inverted XOR is true when the two operands agree in sign, the second XOR is true when the result disagrees with them, and overflow is the AND of the two. (b) rebuilds the hidden carry: since stage 3 is a full adder, $C_3 = A_3 \oplus B_3 \oplus S_3$, so the textbook $V = C_3 \oplus C_4$ becomes a chain of three XORs. Both are derived here — the lab poses the task and gives no solution.

A_3	B_3	S_3	C_3	C_4	V
0	0	0	0	0	0
0	0	1	1	0	**1**
0	1	0	1	1	0
0	1	1	0	0	0
1	0	0	1	1	0
1	0	1	0	0	0
1	1	0	0	1	**1**
1	1	1	1	1	0

V is 1 in exactly two rows — two positives giving a negative, and two negatives giving a positive — which is the rule Q18 applies by hand. Both expressions reproduce that column.

The detector reproduces the paper answers

Run the four sign patterns from Q18 and Q19 through the table. Only the top bits matter:

From	Sum	$A_3B_3S_3$	V	Matches
Q18 (a) $0111 + 0111$	1110	0 0 1	**1**	overflow
Q18 (b) $1111 + 1111$	1110	1 1 1	0	no overflow, despite $C_4 = 1$
Q18 (c) $1010 + 0110$	0000	1 0 0	0	opposite signs cannot overflow
Q19 $1011 + 1111$	1010	1 1 1	0	negative, no overflow

Row two is the whole point of Q19's last paragraph, now visible as wiring: C_4 is 1 there, but C_4 is not an input to the sign comparison at all in form (a), and in form (b) it is cancelled by C_3, which is also 1. **A carry out is not an overflow** — the detector is built so that it cannot confuse the two, which is exactly why the LED for it is a separate output from the carry LED. See binary-number-representations[136] and combinational-logic-blocks[137].

Sample Mid-semester Exam #3

This is the third of UQ's published sample mid-semester papers for CSSE2010/CSSE7201 Introduction to Computer Systems (see csse2010[138] for the course note). The cover sheet is

[136] courses/csse2010/binary-number-representations.pdf
[137] courses/csse2010/combinational-logic-blocks.pdf
[138] courses/csse2010/csse2010.pdf

dated Semester One Mid-semester Examinations, 2020, for St Lucia campus students. It allows 60 minutes of working time plus 10 minutes of reading time, and it is an open book examination — no electronic aids, but a Casio FX82-series or other UQ-approved calculator is permitted. There are 25 multiple-choice questions answered on a supplied answer sheet; a correct answer scores one mark, and an incorrect, missing or multiple answer scores zero. There is no penalty for guessing, so leaving a question blank is never the right move.

> ⚠ **Answers are derived, not official**
>
> The source paper ships without an answer key, so every answer below was worked out during transcription rather than taken from UQ. Each one shows its reasoning so it can be checked rather than trusted; anything uncertain is flagged. Treat a disagreement between your working and the answer here as a reason to check both.

How to use this

Two ways. The first is a timed run: 25 questions in 60 minutes is a little over two minutes each, and on the website every answer is collapsed behind a disclosure triangle, so you can work the whole paper without seeing anything. The second is targeted practice — the index table below gives the week and the concept note for each question, so you can pick out just the questions covering one topic.

Note that the PDF build of this note has every answer expanded, because a PDF cannot hide them. Practise from the site version.

Question index

Q	Topic	Week	Concept notes
1	Excess-256 representation	1	binary-number-representations[139]
2	Hexadecimal, binary and octal	1	binary-number-representations[140]
3	Signed magnitude range	1	binary-number-representations[141]
4	Two's to one's complement	1	binary-number-representations[142]

[139] courses/csse2010/binary-number-representations.pdf
[140] courses/csse2010/binary-number-representations.pdf
[141] courses/csse2010/binary-number-representations.pdf
[142] courses/csse2010/binary-number-representations.pdf

Q	Topic	Week	Concept notes
5	Reading a gate circuit	1	logic-gates[143]
6	Boolean simplification	2	boolean-algebra[144]
7	Building a sum of products	2	logic-gates[145] · boolean-algebra[146]
8	Expression from a truth table	2	boolean-algebra[147]
9	NAND-only implementation	2	logic-gates[148] · boolean-algebra[149]
10	NAND latch behaviour	3	flip-flops-and-latches[150]
11	Which circuit is a latch	3	flip-flops-and-latches[151]
12	Timing diagram of a function	2	timing-diagrams[152] · logic-gates[153]
13	Multiplexer as a function	2	combinational-logic-blocks[154]
14	Two's complement addition	2	binary-number-representations[155]
15	Detecting overflow	2	binary-number-representations[156]
16	Interpreting an overflow	2	binary-number-representations[157]
17	D flip-flop timing	3	flip-flops-and-latches[158] · timing-diagrams[159]

[143] courses/csse2010/logic-gates.pdf
[144] courses/csse2010/boolean-algebra.pdf
[145] courses/csse2010/logic-gates.pdf
[146] courses/csse2010/boolean-algebra.pdf
[147] courses/csse2010/boolean-algebra.pdf
[148] courses/csse2010/logic-gates.pdf
[149] courses/csse2010/boolean-algebra.pdf
[150] courses/csse2010/flip-flops-and-latches.pdf
[151] courses/csse2010/flip-flops-and-latches.pdf
[152] courses/csse2010/timing-diagrams.pdf
[153] courses/csse2010/logic-gates.pdf
[154] courses/csse2010/combinational-logic-blocks.pdf
[155] courses/csse2010/binary-number-representations.pdf
[156] courses/csse2010/binary-number-representations.pdf
[157] courses/csse2010/binary-number-representations.pdf
[158] courses/csse2010/flip-flops-and-latches.pdf
[159] courses/csse2010/timing-diagrams.pdf

Q	Topic	Week	Concept notes
18	Load-or-swap register pair	3	sequential-circuits[160] · shift-registers[161]
19	Flip-flops for a sequence	4	counters[162] · state-machines[163]
20	Tracing a synchronous counter	4	counters[164]
21	Counter width	4	counters[165]
22	State diagram to state table	4	state-machines[166]
23	Tracing a state machine	4	state-machines[167]
24	ALU control settings	5	arithmetic-logic-unit[168]
25	What the ALU cannot do	5	arithmetic-logic-unit[169]

Q1 — Excess-256 representation

What is -57_{10} expressed in 9-bit excess-256 format?

(a) 000111001

(b) 011000111

(c) 100111001

(d) 111000111

(e) None of the above

[160] courses/csse2010/sequential-circuits.pdf
[161] courses/csse2010/shift-registers.pdf
[162] courses/csse2010/counters.pdf
[163] courses/csse2010/state-machines.pdf
[164] courses/csse2010/counters.pdf
[165] courses/csse2010/counters.pdf
[166] courses/csse2010/state-machines.pdf
[167] courses/csse2010/state-machines.pdf
[168] courses/csse2010/arithmetic-logic-unit.pdf
[169] courses/csse2010/arithmetic-logic-unit.pdf

Answer

(b)
Excess-K stores a value v as the plain unsigned binary of $v + K$. With $K = 256$:

$$-57 + 256 = 199$$

and $199 = 128 + 64 + 4 + 2 + 1 = 11000111_2$. Padded to nine bits that is 011000111.
The distractors are the recognisable wrong turns: (d) is the same magnitude bits with a leading 1 (as if the bias had been applied and then a sign bit added), and (a) and (c) are $+57 = 00111001$ dressed up. See binary-number-representations[170].

Q2 — Hexadecimal, binary and octal

0xB9 is equal to

(a) 11001001_2

(b) 10101001_2

(c) 169_{10}

(d) 201_{10}

(e) 271_8

Answer

(e)
Hex digits map straight onto nibbles: B = 1011 and 9 = 1001, so 0xB9 = 10111001_2 = 185_{10}.
None of (a)–(d) is 185: $11001001_2 = 201$, $10101001_2 = 169$, and (c) and (d) name those same two decimal values. So the paper has cross-paired its distractors — each wrong binary answer has its own decimal value sitting next to it as a separate option.
Octal groups the bits in threes from the right: $10\,111\,001 \to 2, 7, 1$, so 0xB9 = 271_8. Checking, $2 \times 64 + 7 \times 8 + 1 = 128 + 56 + 1 = 185$. See binary-number-representations[171].

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

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

Q3 — Signed magnitude range

A 6-bit binary number in signed magnitude representation can represent integers from

(a) -32 to 32

(b) -32 to 31

(c) -31 to 31

(d) 0 to 31

(e) None of the above

Answer

(c)
Signed magnitude spends the most significant bit on the sign and leaves the remaining five bits as an ordinary unsigned magnitude, so the magnitude runs 0 to $2^5 - 1 = 31$ and each magnitude gets both signs. The range is therefore -31 to $+31$.
Option (b) is the two's complement range for six bits, which is the trap: two's complement gets one extra negative value because it has no separate -0. Signed magnitude wastes a code on $100000 = -0$, which is why its range is symmetric and one value shorter. See binary-number-representations[172].

Q4 — Two's to one's complement

How can any n-bit binary number in two's complement representation (other than -2^{n-1}) be converted to n-bit one's complement representation?

(a) By subtracting 1

(b) By subtracting 1 and flipping all the bits

(c) By subtracting the value of the most-significant bit

(d) By adding the value of the most-significant bit

(e) By flipping all the bits and subtracting the value of the most-significant bit

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

Answer

(c)
Write the bit pattern as $a_{n-1}a_{n-2}\ldots a_0$. The two representations differ only in what the top bit is worth:

$$V_{2\text{'s}} = -a_{n-1}2^{n-1} + \sum_{i<n-1} a_i 2^i \qquad V_{1\text{'s}} = -a_{n-1}\left(2^{n-1}-1\right) + \sum_{i<n-1} a_i 2^i$$

So for a given bit pattern, $V_{1\text{'s}} = V_{2\text{'s}} + a_{n-1}$. To keep the *value* the same while switching representation you therefore have to move the pattern down by a_{n-1} — that is, subtract the value of the most significant bit: 1 when the number is negative, 0 when it is positive. Checking on 8 bits with -57: two's complement is 11000111, and subtracting 1 gives 11000110, which is indeed $\overline{00111001}$, the one's complement of -57. And for $+57 = 00111001$ the top bit is 0, so nothing is subtracted and the pattern is unchanged — which is right, because the two representations agree on positive numbers.
That last point is what rules out (a): "subtract 1" is correct for negatives but would corrupt every positive. The exclusion of -2^{n-1} in the question is because one's complement cannot represent it at all. See binary-number-representations[173].

Q5 — Reading a gate circuit

What function does the following circuit implement?

(a) $A.B.C + \overline{D}$

(b) $\overline{A.B.C} \oplus D$

(c) $\overline{\overline{A.B.C} \oplus D}$

(d) $\overline{\overline{A}.\overline{B}.\overline{C} + D}$

(e) $\overline{\overline{A.B.C} + D}$

Answer

(e)
Read it inside out. The first gate is a three-input AND with a bubble, i.e. a NAND, so its output is $\overline{A.B.C}$. That feeds one input of the second gate, which is an OR with a bubble — a NOR — with D on its other input. So the output is

173 courses/csse2010/binary-number-representations.pdf

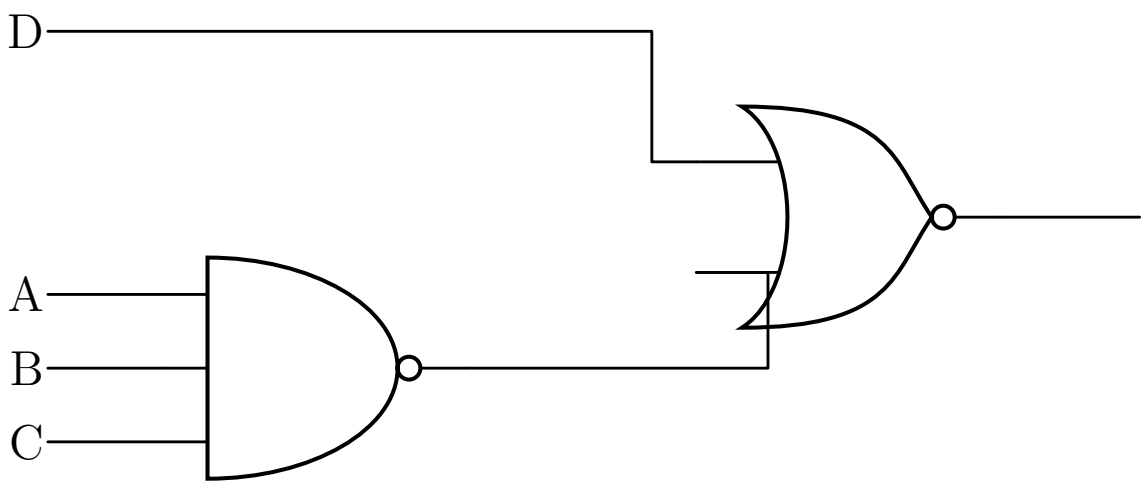


Figure 50: Question 5's circuit: a three-input NAND on A, B and C, whose output is NORed with D.

$$\overline{\overline{A.B.C} + D}$$

which is exactly option (e), written with no simplification at all.
It is worth pushing it one step further to see what the circuit actually does, because that also disposes of (a) and (d). By De Morgan,

$$\overline{\overline{A.B.C} + D} = \overline{\overline{A.B.C}} \cdot \overline{D} = A.B.C.\overline{D}$$

so the circuit is "all three of A, B, C are 1 and D is 0". Option (a) is $A.B.C + \overline{D}$, an OR where the circuit has an AND, and option (d) simplifies to $(A + B + C).\overline{D}$, which is true whenever *any* of A, B, C is 1. Neither matches. See logic-gates[174].

Q6 — Boolean simplification

Which of the following logic functions is equivalent to $(A.\overline{B} + C).(\overline{\overline{A.C} + B})$?

(a) $A.\overline{B}$

(b) $A.\overline{B}.C$

(c) $A.\overline{B} + \overline{B}.C$

174 courses/csse2010/logic-gates.pdf

(d) $A.\overline{B}.\overline{C} + \overline{A}.\overline{B}.C$

(e) None of the above

Answer

(a)
Disputed by review. The question stem was mis-transcribed and has now been corrected: the source prints an inner bar over the whole product $A.C$, so the second factor is $\overline{\overline{A.C} + B}$, not $\overline{\overline{A}.C + B}$. With the corrected stem I make the answer **(b)**, not (a):

$$\overline{\overline{A.C} + B} = A.C.\overline{B} \qquad \text{so} \qquad \left(A.\overline{B} + C\right).A.C.\overline{B} = A.\overline{B}.C$$

since both products of the expansion collapse to $A.\overline{B}.C$. The reasoning below is internally sound but solves the mis-transcribed expression, which is why it lands on (a). Work it yourself before trusting either, and check the bar placement in the PDF.
Deal with the complemented bracket first, using De Morgan twice:

$$\overline{\overline{A}.C + B} = \overline{\overline{A}.C} \cdot \overline{B} = \left(A + \overline{C}\right).\overline{B}$$

So the whole expression is

$$\left(A.\overline{B} + C\right)\left(A + \overline{C}\right)\overline{B}$$

Multiply out the first two brackets:

$$\left(A.\overline{B} + C\right)\left(A + \overline{C}\right) = A.\overline{B}.A + A.\overline{B}.\overline{C} + C.A + C.\overline{C}$$

$C.\overline{C} = 0$ and $A.A = A$, so this is $A.\overline{B} + A.\overline{B}.\overline{C} + A.C$. The middle term is absorbed by the first ($X + X.Y = X$), leaving $A.\overline{B} + A.C$.
Now bring back the trailing $\overline{B}$:

$$\left(A.\overline{B} + A.C\right)\overline{B} = A.\overline{B} + A.\overline{B}.C = A.\overline{B}$$

again by absorption. See boolean-algebra[175].

Q7 — Building a sum of products

Which of the following circuits implements the function $A.B + C.D$?

(e) None of the above

175 courses/csse2010/boolean-algebra.pdf

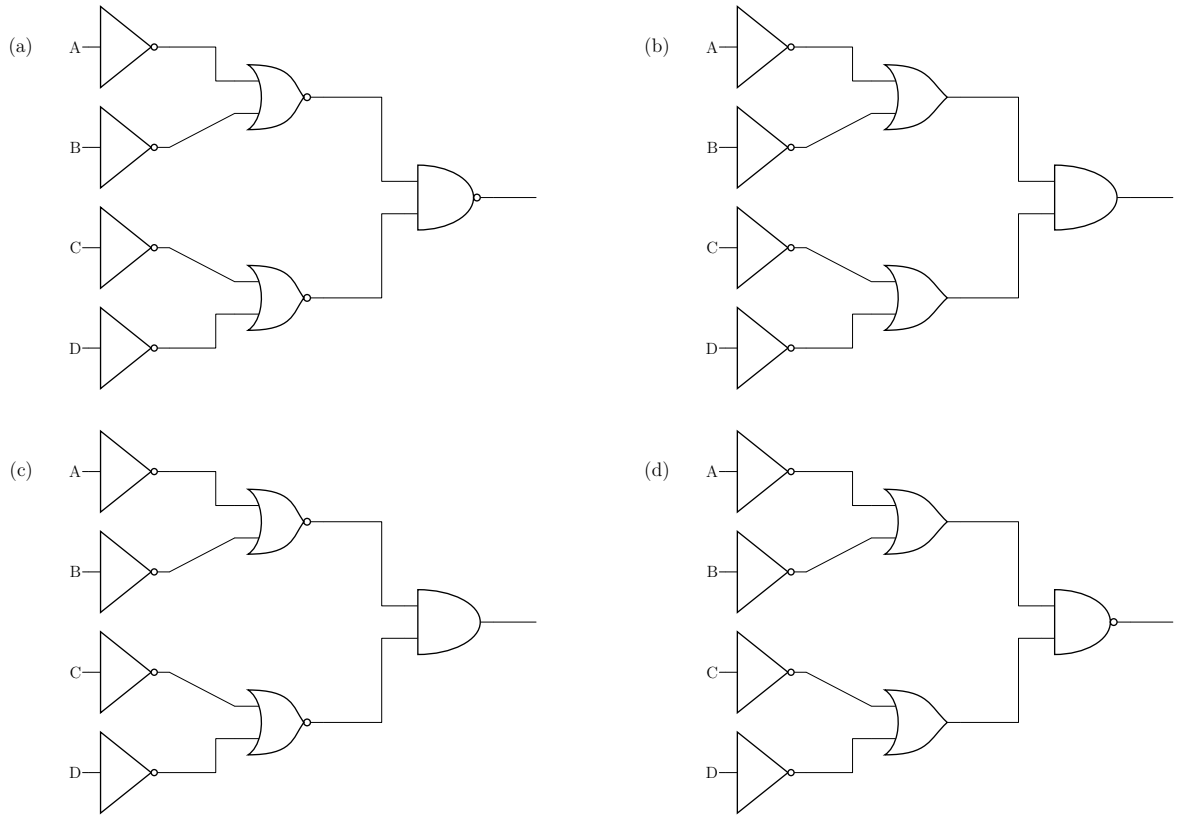


Figure 51: Question 7's four candidate circuits, laid out as in the paper: (a) top left, (b) top right, (c) bottom left, (d) bottom right. All four invert every input first; they differ only in the pair of second-stage gates (NOR or OR) and in the output gate (NAND or AND).

Answer

(d)

Every option starts by inverting all four inputs, so the second-stage gates always see $\overline{A}, \overline{B}, \overline{C}, \overline{D}$. That makes the two useful De Morgan identities the whole question:

$$\overline{\overline{A} + \overline{B}} = A.B \qquad \overline{A} + \overline{B} = \overline{A.B}$$

The first says a NOR fed with inverted inputs is an AND; the second says a plain OR fed with inverted inputs is a NAND. Working the four options through:

	second stage	output gate	function
(a)	NOR, so $A.B$ and $C.D$	NAND	$\overline{A.B \cdot C.D}$
(b)	OR, so $\overline{A.B}$ and $\overline{C.D}$	AND	$\overline{A.B} \cdot \overline{C.D} = \overline{A.B + C.D}$
(c)	NOR, so $A.B$ and $C.D$	AND	$A.B.C.D$
(d)	OR, so $\overline{A.B}$ and $\overline{C.D}$	NAND	$\overline{\overline{A.B} \cdot \overline{C.D}} = A.B + C.D$

Only (d) gives the required $A.B + C.D$, and it does it the standard way: a sum of products built entirely from inverting gates, with the final NAND undoing the double complement. Note that (b) is the exact complement of the answer — an easy option to pick if you forget the final bubble. See logic-gates[176] and boolean-algebra[177].

Q8 — Expression from a truth table

Consider the logic function represented by the following truth table:

A	B	C	X
0	0	0	0
0	0	1	1
0	1	0	1
0	1	1	0
1	0	0	1

177 courses/csse2010/logic-gates.pdf

177 courses/csse2010/boolean-algebra.pdf

A	B	C	X
1	0	1	0
1	1	0	0
1	1	1	0

Which of the following expressions is equivalent to the function represented by this table?

(a) $(A \oplus B).\overline{C} + (\overline{A + B}).C$

(b) $(A \oplus B) + C$

(c) $(A \oplus B) + \overline{A}.\overline{B}.C$

(d) $\overline{A}.\overline{B}.C + \overline{A}.B.\overline{C} + A.B.\overline{C}$

(e) None of the above

Answer

(a)
The function is 1 for exactly three input combinations — 001, 010 and 100 — so it is "exactly one of A, B, C is 1". Its canonical sum of products is

$$X = \overline{A}.\overline{B}.C + \overline{A}.B.\overline{C} + A.\overline{B}.\overline{C}$$

Option (a) splits that on the value of C. When $C = 0$ the second term vanishes and the first leaves $A \oplus B$, which is 1 for exactly 01 and 10 — the minterms 010 and 100. When $C = 1$ the first term vanishes and $\overline{A + B}.C = \overline{A}.\overline{B}.C$ picks out 001 alone. Together that is exactly the three rows, so (a) matches.
Checking the others against the table is quick. (b) $(A \oplus B) + C$ is 1 at 011, where the table says 0. (c) is 1 at 011 for the same reason. (d) is the canonical form with its third term mistyped: it has $A.B.\overline{C}$ (row 110, where the table says 0) instead of $A.\overline{B}.\overline{C}$ (row 100, where the table says 1). See boolean-algebra[178].

Q9 — NAND-only implementation

Which of the following circuits is a NAND-gate only implementation of the following function:

(e) None of the above

178 courses/csse2010/boolean-algebra.pdf

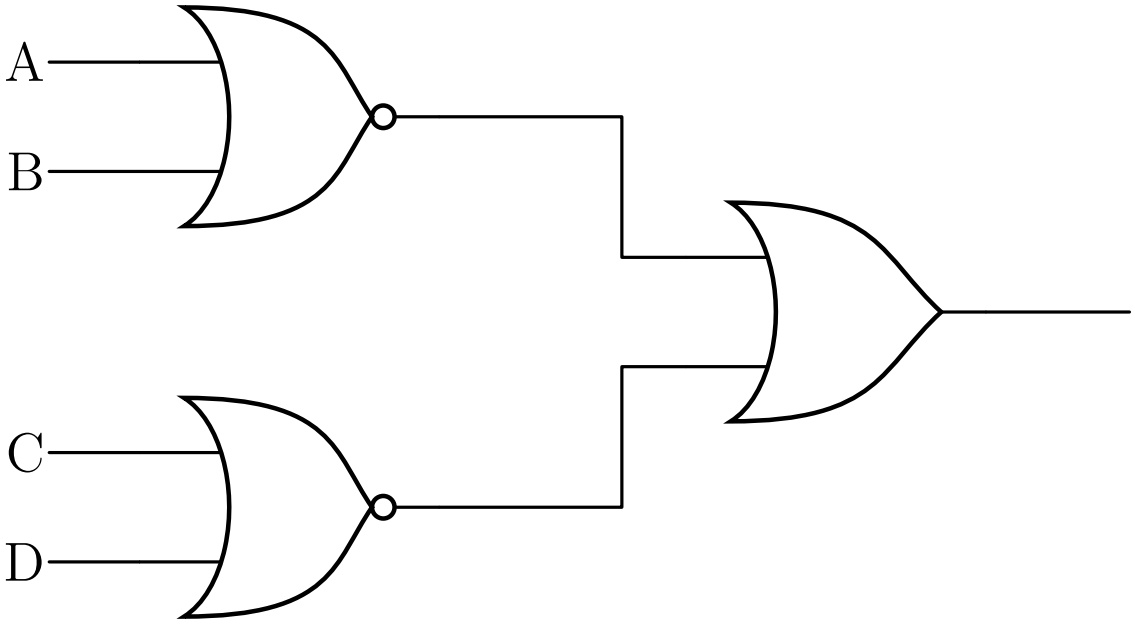


Figure 52: The function question 9 asks you to rebuild: NOR(A,B) ORed with NOR(C,D).

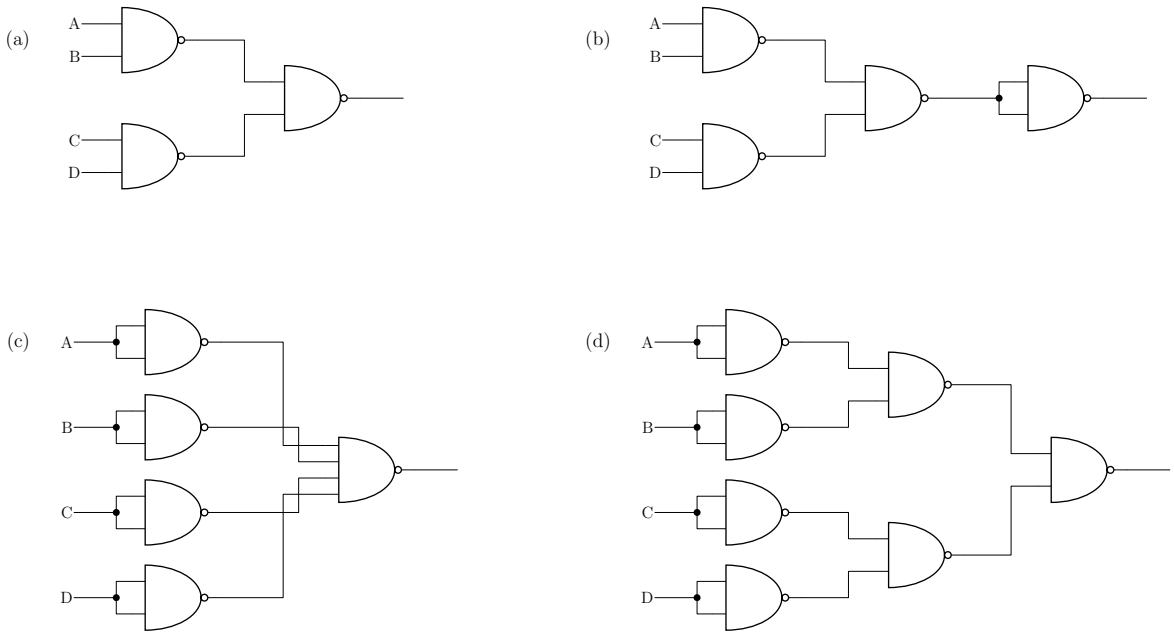


Figure 53: Question 9's four candidate NAND-only circuits: (a) top left, (b) top right, (c) bottom left, (d) bottom right. A NAND with both inputs tied to the same signal — drawn here as a short fork with a junction dot — is a NAND used as an inverter.

Answer

(d)
First write down what the target circuit computes. The two first-stage gates are NORs and the output gate is a plain OR, so

$$F = \overline{A + B} + \overline{C + D} = \overline{A}.\overline{B} + \overline{C}.\overline{D}$$

Now convert. A sum becomes a NAND of the two complemented terms, $P + Q = \overline{\overline{P} \cdot \overline{Q}}$, so with $P = \overline{A}.\overline{B}$ and $Q = \overline{C}.\overline{D}$ the output gate must be a NAND fed with $\overline{P}$ and $\overline{Q}$. And $\overline{P} = \overline{\overline{A}.\overline{B}}$ is itself just a NAND on $\overline{A}$ and $\overline{B}$. So:

$$F = \text{NAND}(\text{NAND}(\overline{A}, \overline{B}),\ \text{NAND}(\overline{C}, \overline{D}))$$

The four inverters are NANDs with their inputs tied together, giving seven NAND gates in total — which is exactly circuit (d).
The others are all recognisable near-misses:

- **(a)** is $\overline{\overline{A.B} \cdot \overline{C.D}} = A.B + C.D$ — the inputs are not inverted, so it computes the sum of the *un*-complemented products.
- **(b)** is (a) with an inverter, i.e. $\overline{A.B + C.D}$. Try $A = 1, B = 0, C = 1, D = 0$: the target gives 0, (b) gives 1.
- **(c)** inverts all four inputs correctly but then feeds them into a single four-input NAND, giving $\overline{\overline{A}.\overline{B}.\overline{C}.\overline{D}}$ — an AND of all four where the target wants an OR of two pairs.

See logic-gates[179] and boolean-algebra[180].

Q10 — NAND latch behaviour

Which truth table correctly captures the behaviour of the following latch circuit?

(a)

M	N	Q	$\overline{\text{Q}}$
0	0	Q	$\overline{\text{Q}}$
0	1	1	0
1	0	0	1

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

M	N	Q	$\overline{\text{Q}}$
1	1	0	0

(b)

M	N	Q	$\overline{\text{Q}}$
0	0	1	1
0	1	0	1
1	0	1	0
1	1	Q	$\overline{\text{Q}}$

(c)

M	N	Q	$\overline{\text{Q}}$
0	0	1	1
0	1	1	0
1	0	0	1
1	1	Q	$\overline{\text{Q}}$

(d)

M	N	Q	$\overline{\text{Q}}$
0	0	0	0
0	1	1	0
1	0	0	1
1	1	Q	$\overline{\text{Q}}$

(e) None of the above

Answer

(b)
This is the standard cross-coupled NAND latch, so both inputs are active low. Call the top gate's output P (the one the paper labels $\overline{\text{Q}}$) and the bottom gate's output Q. Then

$$P = \overline{M \cdot Q} \qquad Q = \overline{P \cdot N}$$

Work the four input combinations:

- **M = 0, N = 0.** A 0 on a NAND input forces its output to 1, so both gates output

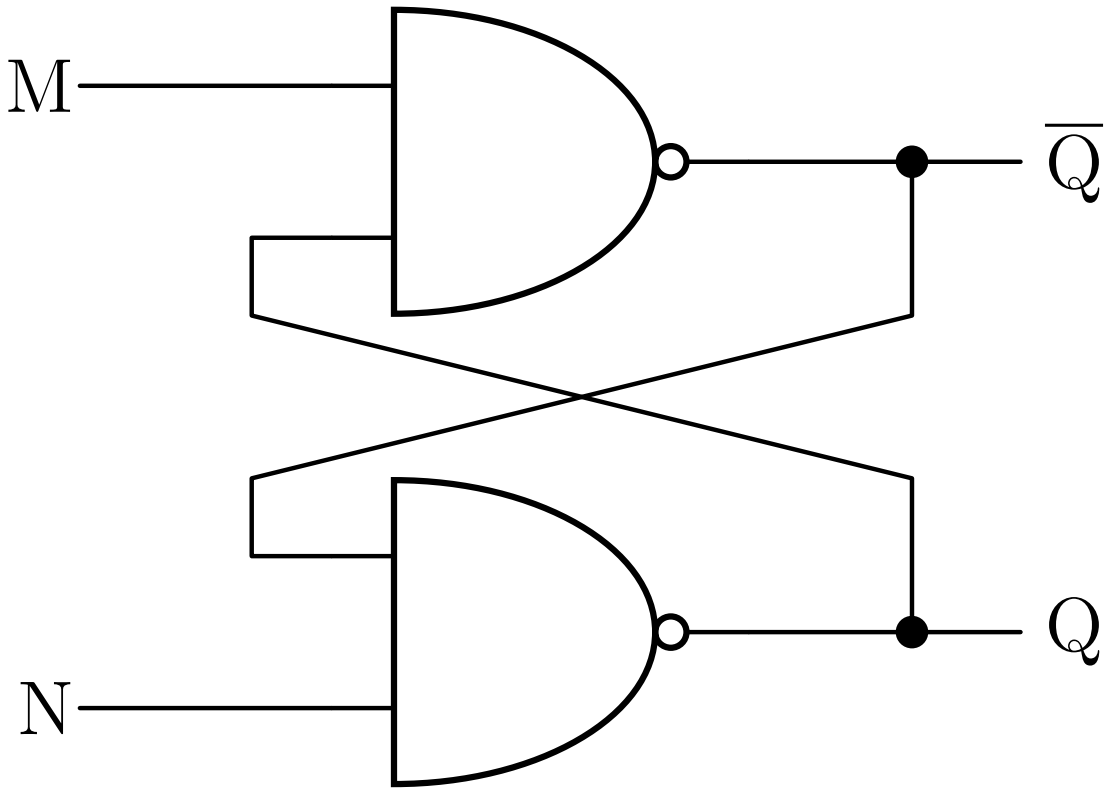


Figure 54: Question 10's latch: two cross-coupled NAND gates, with M into the top gate (whose output is labelled Q-bar) and N into the bottom gate (whose output is labelled Q).

1: $Q = 1$ and $\overline{Q} = 1$. This is the forbidden combination — the two outputs are not complements.

- **M = 0, N = 1.** The top gate is forced, $P = 1$. Then $Q = \overline{1 \cdot 1} = 0$. So $Q = 0$, $\overline{Q} = 1$ — a clear.
- **M = 1, N = 0.** The bottom gate is forced, $Q = 1$. Then $P = \overline{1 \cdot 1} = 0$. So $Q = 1$, $\overline{Q} = 0$ — a set.
- **M = 1, N = 1.** Neither gate is forced, so $P = \overline{Q}$ and $Q = \overline{P}$, which is satisfied by either stored value. This is the hold state.

That is exactly table (b): $1, 1$ then $0, 1$ then $1, 0$ then hold.
The traps are worth naming. Table (a) puts the hold row at 00 and the forbidden row at 11 — that is the cross-coupled *NOR* latch, which has active-high inputs. Tables (c) and (d) get the hold row right but flip the set and clear rows, or write the forbidden state as $0, 0$ instead of $1, 1$ (NANDs cannot both output 0). See flip-flops-and-latches[181].

Q11 — Which circuit is a latch

Which of the following circuits is capable of acting as a latch? i.e. it has combinations of inputs (M,N) which allow a bit (0 or 1) to be stored, the latch to be set (Q=1) with a single input transition, and the latch to be cleared (Q=0) with a single input transition?

(e) None of the above

Answer

(e)
All four circuits are wired the same way. Writing P for the bottom gate's output, each is $Q = g_{\text{top}}(M, P)$ and $P = g_{\text{bot}}(Q, N)$, so the whole question is which pair of gate types gives a working memory element. For each circuit, substitute both equations into each other for all four (M, N) and see whether the result is consistent (a stable state), forced (set or clear), or self-contradictory (the circuit oscillates).
(a) NOR over XOR — $Q = \overline{M + P}$, $P = Q \oplus N$.

M N	working	result
0 0	$P = Q$, so $Q = \overline{Q}$	oscillates
0 1	$P = \overline{Q}$, so $Q = Q$	**hold**
1 0	$Q = 0$, $P = 0$	clear
1 1	$Q = 0$, $P = 1$	clear

181 courses/csse2010/flip-flops-and-latches.pdf

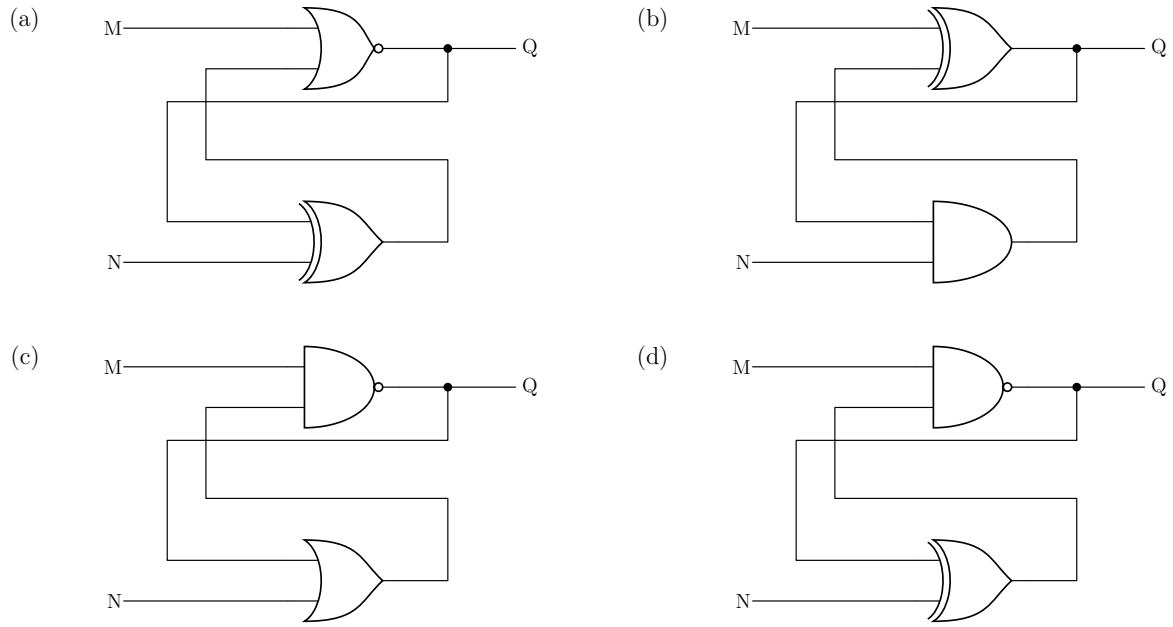


Figure 55: Question 11's four candidate circuits, wired identically and differing only in the two gate types: (a) NOR over XOR, (b) XOR over AND, (c) NAND over OR, (d) NAND over XOR. In each, M feeds the top gate whose output is Q, N feeds the bottom gate, and the two are cross-coupled.

From the hold state 01, changing M gives 11 (a clear) but changing N gives 00 (oscillation). It can be cleared but never set.
(b) XOR over AND — $Q = M \oplus P$, $P = Q.N$. States: 00 clear, 01 hold, 10 set, 11 oscillates. The hold state 01 can reach the clear at 00 in one transition, but reaching the set at 10 needs two. Fails.
(c) NAND over OR — $Q = \overline{M.P}$, $P = Q + N$. With $M = 0$ the NAND is forced, so $Q = 1$ regardless of N; with $M = 1, N = 1$ we get $Q = 0$; with $M = 1, N = 0$ it oscillates. There is no hold state at all, so this circuit has no memory.
(d) NAND over XOR — $Q = \overline{M.P}$, $P = Q \oplus N$. States: 00 and 01 both set ($Q = 1$), 10 oscillates, 11 holds. From the hold state 11, changing M gives a set, but changing N gives oscillation. It can be set but never cleared.
So none of the four does all three jobs, and the answer is (e). This is the point of the question: the classic latches are cross-coupled *NOR* pairs (hold at 00, set and clear on either input alone) or cross-coupled *NAND* pairs (hold at 11, as in Q10), and none of these four mixed pairs reproduces that. See flip-flops-and-latches[182].
Check this one. All eight gate symbols have now been read twice, independently, at high magnification against the source PDF, and both readings agree with the identifications above. The whole answer still turns on them, though: read the bottom gate of (a) as a NOR rather than an XOR and (a) becomes the textbook NOR latch, making it the answer instead.

Q12 — Timing diagram of a function

Which of the following timing diagrams represents the function $A.B.C + \overline{B}.C$?

(e) None of the above

Answer

(a)
The three input traces are a binary counter: C toggles every slot, B every two, A every four, so over the nine slots ABC runs $000, 001, 010, 011, 100, 101, 110, 111$ and then back to 000. That turns the timing diagram into a truth table read left to right.
Evaluating $F = A.B.C + \overline{B}.C$ slot by slot:

slot	A B C	$A.B.C$	$\overline{B}.C$	F
1	000	0	0	0
2	001	0	1	**1**

182 courses/csse2010/flip-flops-and-latches.pdf

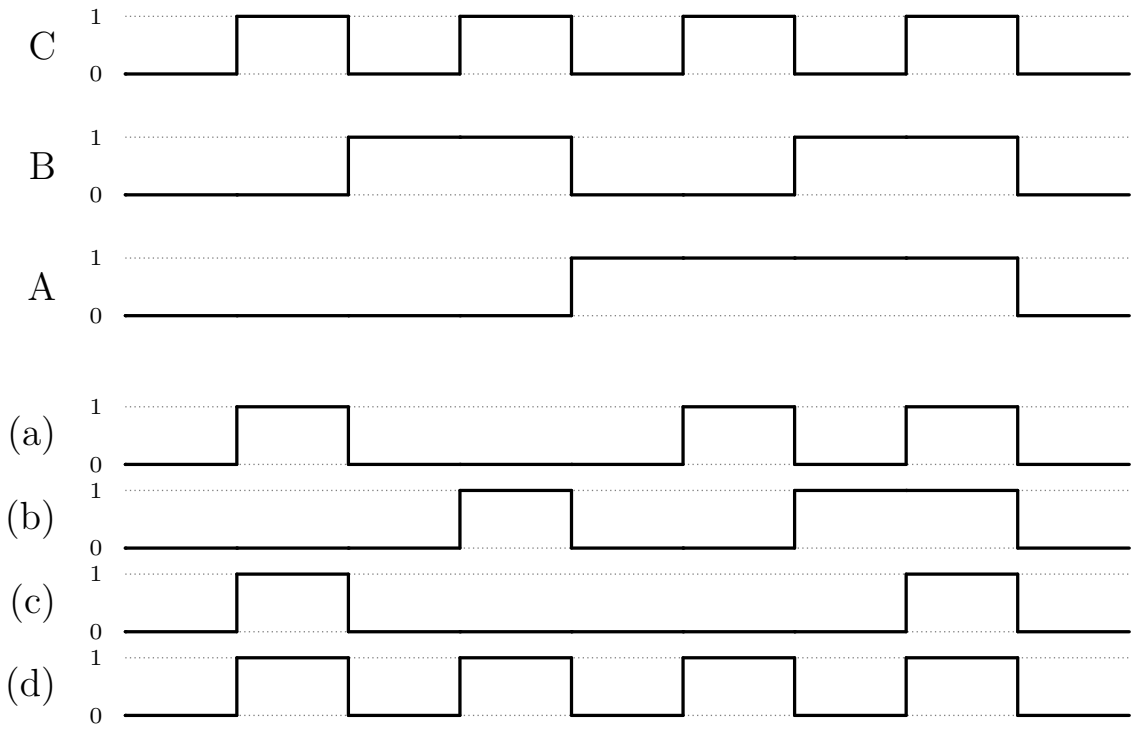


Figure 56: Question 12's timing diagrams. The top three traces are the inputs C, B and A, which between them count 000 to 111 over the first eight slots and return to 000 in the ninth; the bottom four are the candidate outputs.

3	010	0	0	0
4	011	0	0	0
5	100	0	0	0
6	101	0	1	**1**
7	110	0	0	0
8	111	1	0	**1**
9	000	0	0	0

So the output is high in slots 2, 6 and 8 and low everywhere else, which is trace (a). The other traces are each a different function of the same inputs: (d) is simply C, which would be the answer if the B terms were missing; (c) is high only in slots 2 and 8, i.e. it drops the 101 case; and (b) is high in slots 4, 7 and 8, which is a different function entirely. See timing-diagrams[183] and logic-gates[184].

Q13 — Multiplexer as a function

Consider the following multiplexer. What must the inputs A,B,C,D be so that the multiplexer implements the function $X = S_1 + S_0.G$?

(a) A=**0** B=***G*** C=**1** D=**1**

(b) A=**1** B=**1** C=**0** D=***G***

(c) A=**1** B=**1** C=***G*** D=**0**

(d) A=***G*** B=***G*** C=**1** D=**1**

(e) A=***G*** B=**0** C=**1** D=**1**

Answer

(a)
A multiplexer whose select lines are the function's own variables is just a lookup table: input n is passed through when S_1S_0 is the binary encoding of n, so input n must carry whatever the function evaluates to for that combination of selects. Any variable *not* used as a select — here G — is allowed to appear on the data inputs.
Evaluating $X = S_1 + S_0.G$ for each select value:

184 courses/csse2010/timing-diagrams.pdf

184 courses/csse2010/logic-gates.pdf

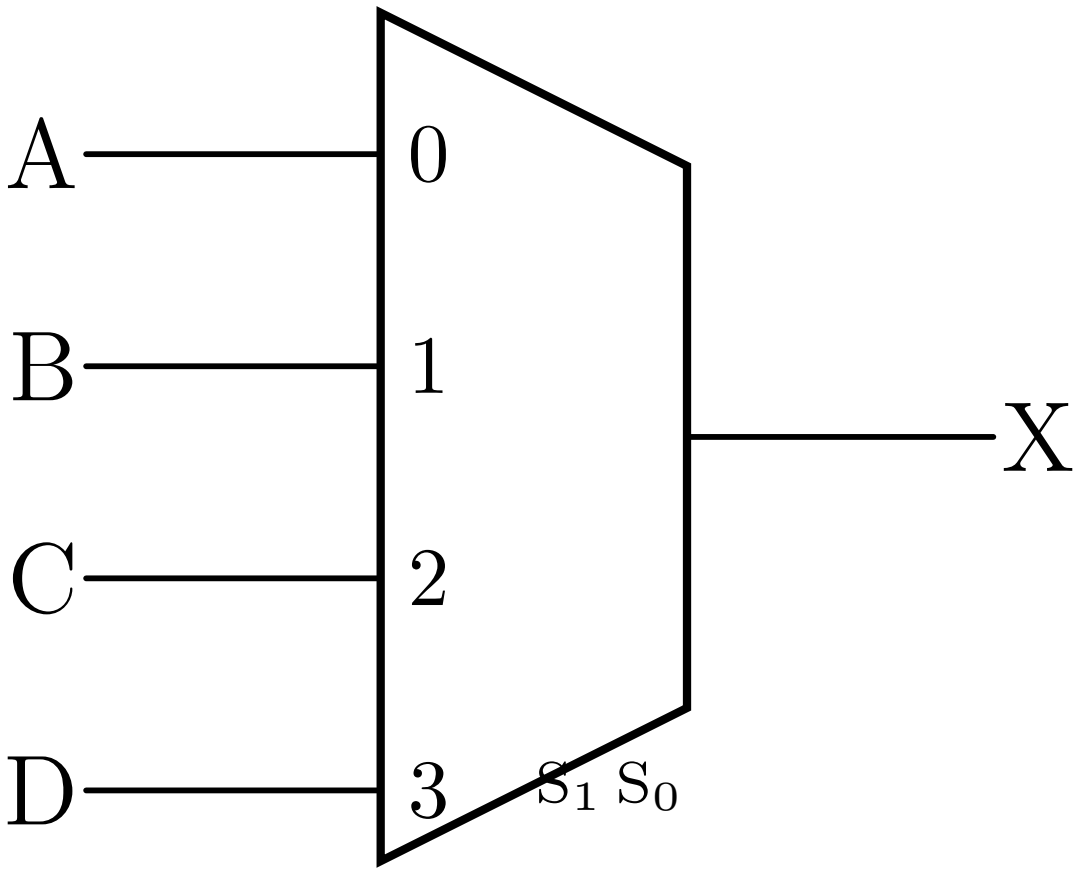


Figure 57: Question 13's four-to-one multiplexer: inputs A, B, C and D on lines 0 to 3, selected by S1 S0, with output X.

S_1S_0	line	$X = S_1 + S_0.G$	required input
00	0	$0 + 0 \cdot G = 0$	A = 0
01	1	$0 + 1 \cdot G = G$	B = G
10	2	$1 + 0 \cdot G = 1$	C = 1
11	3	$1 + 1 \cdot G = 1$	D = 1

So A=0, B=G, C=1, D=1, which is option (a). Note how $S_1 = 1$ makes the function true whatever S_0 and G are, which is why both of the bottom two lines are hard-wired to 1. See combinational-logic-blocks[185].

Q14 — Two's complement addition

What is the result of adding the 6-bit two's complement numbers 010110 and 110011?

(a) 000001

(b) 001001

(c) 100101

(d) 101001

(e) 110111

Answer

(b)
Add the bit patterns as if they were unsigned, then throw away anything past the sixth bit — that is the whole trick of two's complement, and it is why the same adder works for signed and unsigned:

$$\begin{array}{r} 010110 \\ +\ 110011 \\ \hline 1\,001001 \end{array}$$

Column by column from the right: $0+1=1$; $1+1=0$ carry 1; $1+0+1=0$ carry 1; $0+0+1=1$; $1+1=0$ carry 1; $0+1+1=0$ carry 1. Discarding the carry out of the top bit leaves 001001.
Sanity check in decimal: 010110 = 22, and 110011 is negative, with magnitude $\overline{110011}+1 = 001100 + 1 = 001101 = 13$, so it is -13. And $22 - 13 = 9 = 001001$. The result is positive and within range, so the discarded carry is not an overflow. See binary-number-

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

representations[186].

Q15 — Detecting overflow

Which of the following operations will result in an overflow in 4-bit two's complement arithmetic?

(a) 0100+0101

(b) 0100–0101

(c) 1001–0001

(d) 1001+0001

(e) None of the above

Answer

(a)
Four-bit two's complement covers -8 to $+7$. Overflow means the true mathematical result falls outside that range; equivalently, it happens only when the two operands have the same sign and the result has the opposite sign.

	decimal	result	in range?
(a) $0100 + 0101$	$4 + 5$	9	no — **overflow**
(b) $0100 - 0101$	$4 - 5$	-1	yes
(c) $1001 - 0001$	$-7 - 1$	-8	yes, just
(d) $1001 + 0001$	$-7 + 1$	-6	yes

186 courses/csse2010/binary-number-representations.pdf

Only (a) escapes. Its bit sum is $0100 + 0101 = 1001$, which reads as -7 — two positives adding to a negative, the classic signature of overflow.
Option (c) is the near miss worth checking carefully: -8 is representable in four bits (it is the one asymmetry of two's complement), so $1001 - 0001 = 1000 = -8$ is a legitimate result. Note also that (b) and (d) can never overflow at all, since adding numbers of opposite signs always produces a result no larger in magnitude than the larger operand. See binary-number-representations[187].
These same four operations also come up in Learning Lab 3 (Binary Arithmetic), in a different order and with the same answer, so this is a question the course drills rather than a one-off. See week2-lab-binary-arithmetic[188].

Q16 — Interpreting an overflow

Consider the binary addition operation 1001 + 1110. Which of the following statements is true if the numbers are interpreted as two's complement?

(a) The result is negative and no overflow occurs

(b) The result should be negative but overflow occurs

(c) The result is positive and no overflow occurs

(d) The result should be positive but overflow occurs

(e) The result is zero

Answer

(b)
As two's complement values, $1001 = -7$ and $1110 = -2$, so the true answer is -9 — outside the -8 to $+7$ range, hence an overflow. The correct answer "should be negative", which rules out (d).
Doing the addition confirms it:

$$\begin{array}{r} 1001 \\ +\ 1110 \\ \hline 1\,0111 \end{array}$$

Discarding the carry leaves $0111 = +7$: two negatives have produced a positive, which is precisely the overflow signature and is why (a) and (c) are wrong. The carry out of the

188 courses/csse2010/binary-number-representations.pdf

188 courses/csse2010/week2-lab-binary-arithmetic.pdf

top bit is not itself the overflow flag here — a two's complement addition can produce a carry out with no overflow at all (as in Q14). See binary-number-representations[189].

Q17 — D flip-flop timing

Consider the flip-flop circuit shown below and the associated CLK and output (Q) signals. Which timing diagram shows input (A) values over time which would have resulted in the given output (Q).

Answer

(d)
The clock triangle on the symbol has no bubble, so this is a positive-edge-triggered D flip-flop: at each rising clock edge, Q takes the value D had just before that edge, and between edges Q holds. There are four rising edges in the diagram. Reading the Q trace against them:

edge	Q before	Q after	so A just before the edge must be
1	0	1	**1**
2	1	1	**1**
3	1	0	**0**
4	0	1	**1**

So the requirement is $A = 1, 1, 0, 1$ at the four edges. Checking each candidate:

- **(a)** goes low right at the first edge and only comes back high at the second, so it reads $1, 0, \ldots$ — wrong at edge 2. (Its transitions also land exactly on the clock edges, which would be a setup-time violation in real hardware.)
- **(b)** is high at edges 1, 2 and 3, so Q would stay high at edge 3 instead of falling.
- **(c)** gets edges 1, 2 and 3 right but is low at edge 4, so Q would not rise again.
- **(d)** is high until part-way through the third clock cycle and low until part-way through the fifth, giving $1, 1, 0, 1$ — a match, and with every transition comfortably away from a clock edge.
- **(e)** has already gone low before edge 2, so Q would fall there.

Only (d) works. The useful habit here is to ignore the shape of A between edges entirely: a D flip-flop samples, so only the four instants matter. See flip-flops-and-latches[190] and timing-diagrams[191].

189 courses/csse2010/binary-number-representations.pdf

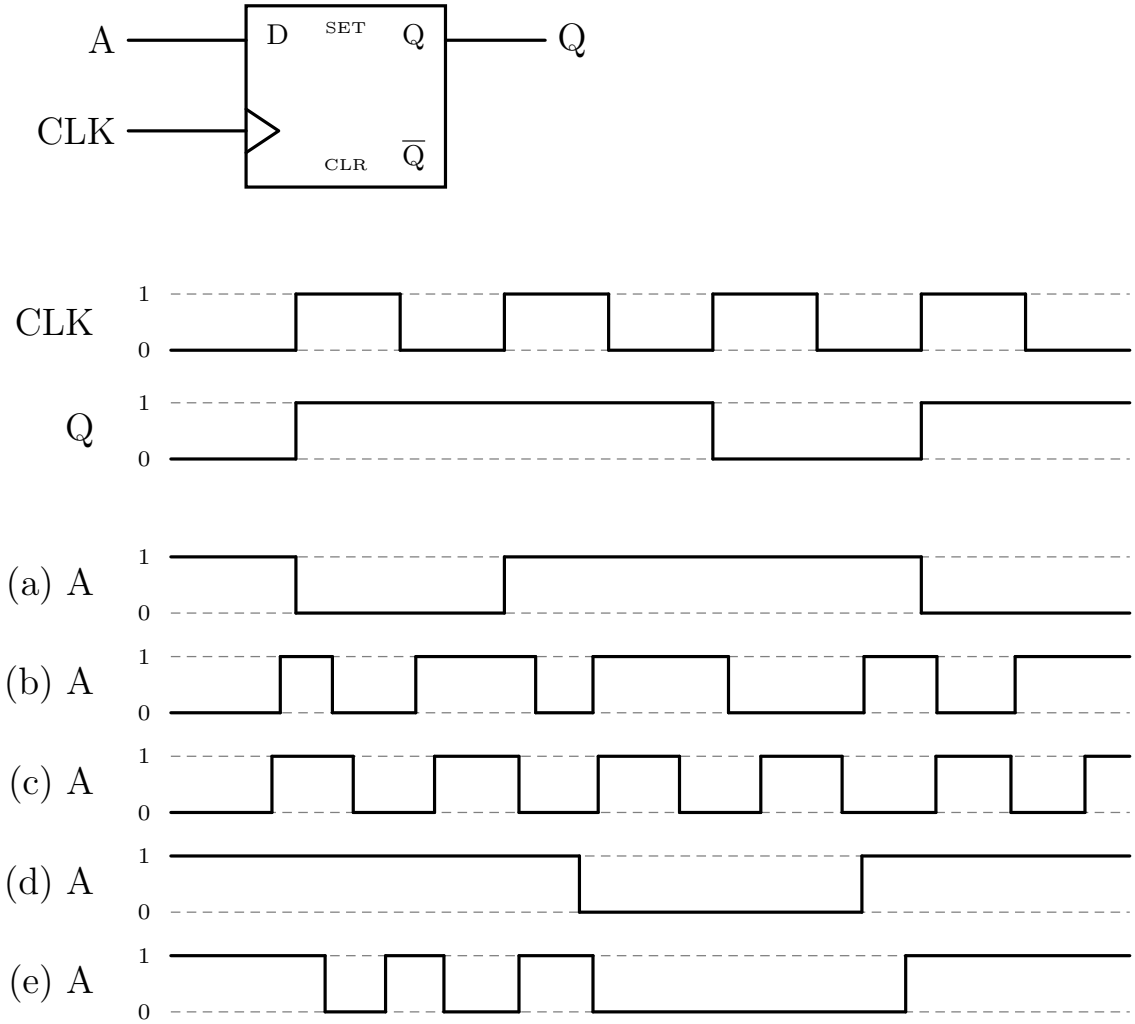


Figure 58: Question 17's positive-edge-triggered D flip-flop, the given CLK and Q traces, and the five candidate A inputs.

Q18 — Load-or-swap register pair

Consider the following circuit. If A_1 is 1 and A_0 is 0, what will be the values of Q_1 and Q_0 after three rising clock edges if FN is 0 for one rising clock edge then 1 for two rising clock edges?

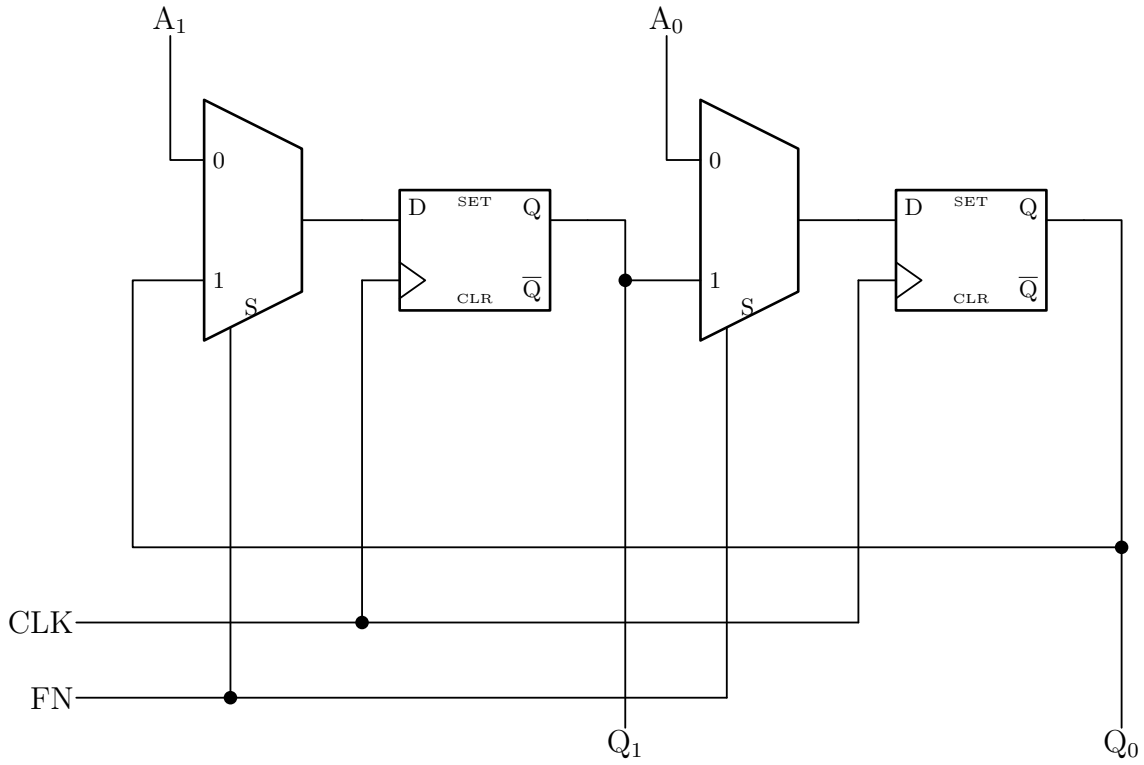


Figure 59: Question 18's circuit: two D flip-flops, each fed by a multiplexer selected by FN. With FN=0 each flip-flop loads its own A input; with FN=1 each loads the other flip-flop's output, so the pair swaps.

(a) $Q_1 = 0$ $Q_0 = 0$

(b) $Q_1 = 0$ $Q_0 = 1$

(c) $Q_1 = 1$ $Q_0 = 0$

(d) $Q_1 = 1$ $Q_0 = 1$

---

191 courses/csse2010/flip-flops-and-latches.pdf

191 courses/csse2010/timing-diagrams.pdf

(e) Insufficient information is provided to determine the values

Answer

(c)
Read the two multiplexers first. Both take their select from FN, so FN picks one of two behaviours for the whole circuit:

- **FN = 0** selects line 0 on each multiplexer, which is A_1 for the left flip-flop and A_0 for the right one. This is a parallel load.
- **FN = 1** selects line 1. The left multiplexer's line 1 comes from the long feedback rail running back from Q_0, and the right multiplexer's line 1 comes from Q_1. So $D_1 = Q_0$ and $D_0 = Q_1$ — the two bits swap.

Both flip-flops share the clock, so the swap happens simultaneously and there is no race: each captures the other's *old* value.
Now trace the three edges with $A_1A_0 = 10$:

edge	FN	action	Q_1Q_0 after
1	0	load A_1A_0	1 0
2	1	swap	0 1
3	1	swap	1 0

After three edges $Q_1 = 1$ and $Q_0 = 0$, which is option (c). Two swaps cancel, so the pair is back to what the load put there — which is the real point of the question.
Option (e) is the trap for anyone who worries about the initial contents of the flip-flops, but the first edge loads both of them from A_1 and A_0, so whatever they held beforehand is irrelevant. See sequential-circuits[192] and shift-registers[193].

Q19 — Flip-flops for a sequence

A sequential circuit has four outputs (X_3, X_2, X_1, X_0) which advance through the following repeating sequence:

$$X_3X_2X_1X_0 : \quad 0000 \to 0001 \to 0011 \to 0111 \to 1111 \to 1100 \to 0000 \to \ldots$$

What is the minimum number of flip-flops needed to implement this circuit?

193 courses/csse2010/sequential-circuits.pdf

193 courses/csse2010/shift-registers.pdf

(a) 2

(b) 3

(c) 4

(d) 6

(e) 8

Answer

(b)
Count the distinct values in the cycle: 0000, 0001, 0011, 0111, 1111, 1100 — six of them, and then it repeats. The flip-flops have to hold the *state*, not the outputs, and n flip-flops give 2^n states, so we need the smallest n with $2^n \geq 6$:

$$2^2 = 4 < 6 \leq 8 = 2^3$$

so three flip-flops, which is (b).
The obvious answer is (c), four — one flip-flop per output bit. That works, and it is what you would build if you wanted the outputs straight off the flip-flops with no extra logic. But the question asks for the *minimum*, and with only six states in the cycle you can encode the state in three bits and derive the four outputs combinationally from that encoding. The cost is a block of output logic; the saving is a flip-flop.
Note that two would not do: four states cannot distinguish six points in the cycle, and each state must have a unique successor, so distinct steps need distinct encodings. See counters[194] and state-machines[195].

Q20 — Tracing a synchronous counter

What sequence will the following synchronous counter count through, if it starts at $Q_1Q_0 = 00$?

(a) Q_1Q_0: $00 \to 01 \to 10 \to 11 \to 00 \to \ldots$

(b) Q_1Q_0: $00 \to 01 \to 11 \to 00 \to \ldots$

(c) Q_1Q_0: $00 \to 01 \to 11 \to 10 \to 00 \to \ldots$

(d) Q_1Q_0: $00 \to 10 \to 11 \to 00 \to \ldots$

(e) Q_1Q_0: $00 \to 11 \to 00 \to \ldots$

195 courses/csse2010/counters.pdf

195 courses/csse2010/state-machines.pdf

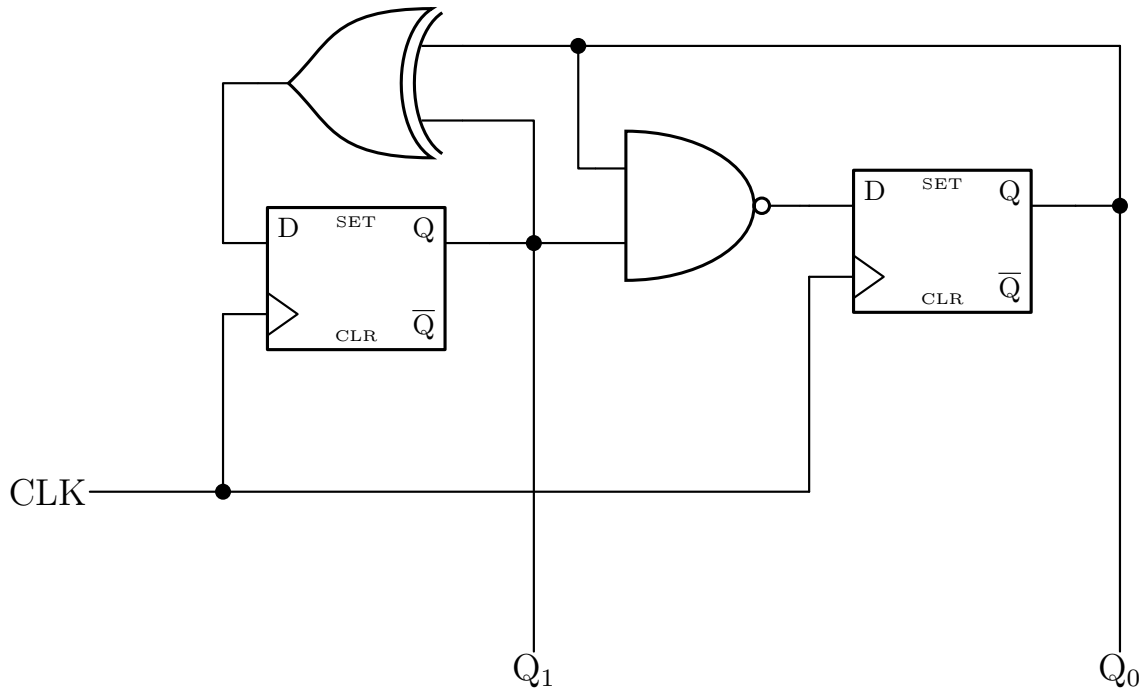


Figure 60: Question 20's counter. The XOR is drawn mirrored, as in the paper: its output faces left into the first flip-flop's D input and its two inputs face right onto the Q1 and Q0 lines. The NAND takes the same two signals and drives the second flip-flop.

Answer

(b)
Both gates see the same two signals — the top rail carries Q_0 back from the second flip-flop, and the vertical line carries Q_1 from the first — so the excitation equations are

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

Both flip-flops share the clock, so on each edge they update together from the *current* state. Building the state table:

Q_1Q_0	$D_1 = Q_1 \oplus Q_0$	$D_0 = \overline{Q_1.Q_0}$	next
00	0	1	**01**
01	1	1	**11**
11	0	0	**00**
10	1	1	11

Starting at 00 the counter goes $00 \to 01 \to 11 \to 00$, a three-state cycle, which is option (b).
The fourth row is the unused state. It is worth checking because a counter that can lock up in an unused state is broken: $10 \to 11$, which is inside the main cycle, so the circuit self-corrects within one clock. See counters[196].

Q21 — Counter width

What's the minimum number of flip-flops needed to build a synchronous binary counter capable of counting from 0 to 16 (inclusive)?

(a) 3

(b) 4

(c) 5

(d) 16

(e) 17

196 courses/csse2010/counters.pdf

Answer

(c)
"0 to 16 inclusive" is seventeen values, not sixteen — that is the whole question. With n flip-flops a binary counter has 2^n distinct states, so we need $2^n \geq 17$:

$$2^4 = 16 < 17 \leq 32 = 2^5$$

so five flip-flops. Four would get to 15 and no further.
Option (b) is what you get by reading "16" as the count of values rather than the last value, and options (d) and (e) confuse flip-flops with states — a one-hot ring counter would need one flip-flop per state, but a *binary* counter encodes the count in binary, which is exactly why it only needs $\lceil \log_2 17 \rceil = 5$. See counters[197].

Q22 — State diagram to state table

Consider the state diagram below.

Which of the following is an equivalent state table?

(a)

Current State	AB=00	AB=01	AB=10	AB=11	Outputs (X,Y)
S0	S0	S0	S1	S1	11
S1	S0	S2	S1	S2	10
S2	S0	S2	S1	S2	01

(b)

Current State	AB=00	AB=01	AB=10	AB=11	Outputs (X,Y)
S0	S0	S1	S0	S1	11
S1	S0	S2	S1	S1	10
S2	S0	S1	S2	S2	01

(c)

197 courses/csse2010/counters.pdf

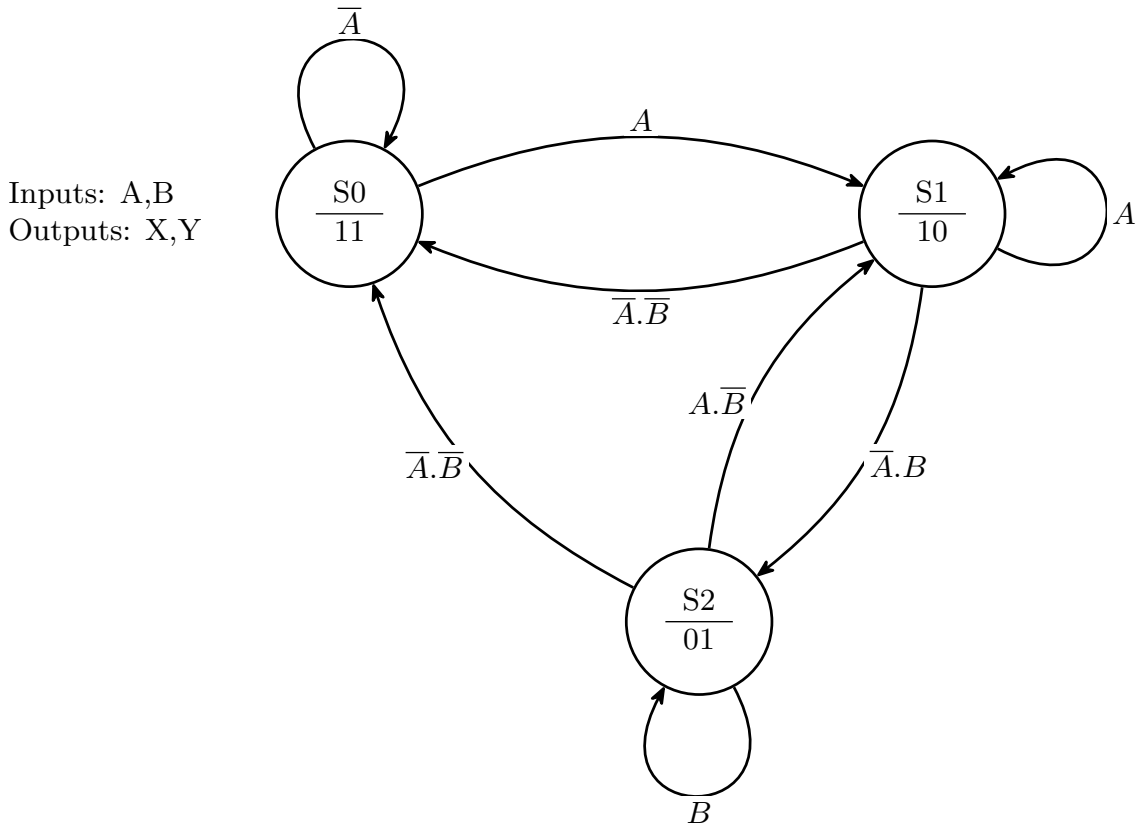


Figure 61: Question 22's state diagram. Three states, each labelled with its name over its X,Y output pair; every arc carries the condition on the inputs A and B under which it is taken.

Current State	AB=00	AB=01	AB=10	AB=11	Outputs (X,Y)
S0	S0	S1	S0	S1	00
S1	S0	S2	S1	S1	01
S2	S0	S2	S1	S2	10

(d)

Current State	AB=00	AB=01	AB=10	AB=11	Outputs (X,Y)
S0	S0	S0	S1	S1	11
S1	S0	S2	S1	S1	10
S2	S0	S2	S1	S2	01

(e) None of the above

Answer

(d)

Take each state's outgoing arcs and expand their conditions into the four values of AB. Note that the diagram's labels do not all name both inputs — A on its own covers $AB = 10$ and $AB = 11$, and B on its own covers 01 and 11 — which is where most of the difference between the options lives.

S0 has two arcs: a self-loop on $\overline{A}$ and an arc to S1 on A. So $AB = 00, 01 \rightarrow$ S0 and $AB = 10, 11 \rightarrow$ S1.

S1 has three arcs: a self-loop on A, an arc to S0 on $\overline{A}.\overline{B}$, and an arc to S2 on $\overline{A}.B$. So $AB = 00 \rightarrow$ S0, $01 \rightarrow$ S2, and $10, 11 \rightarrow$ S1 (both covered by the bare A).

S2 has three arcs: a self-loop on B, an arc to S1 on $A.\overline{B}$, and an arc to S0 on $\overline{A}.\overline{B}$. So $AB = 01, 11 \rightarrow$ S2 (both covered by the bare B), $10 \rightarrow$ S1 and $00 \rightarrow$ S0.

Each state's arcs cover all four combinations exactly once, which is the completeness check worth doing before comparing anything. The outputs are read straight off the bottom half of each circle: S0 is 11, S1 is 10, S2 is 01.

That gives table (d). The near-miss is (a), which is identical except that it sends S1 to S2 on $AB = 11$ — that would be right if S1's self-loop were labelled $A.\overline{B}$ rather than plain A. Table (b) mangles S0's row (sending it to S1 on 01 instead of 10), and (c) has the outputs shifted onto the wrong states. See state-machines[198].

The table you have just built is the **2-dimensional state table** of the lab's design pipeline — one row per state, one next-state column per input combination. It is step 2 of six; Beyond the paper below walks the remaining four on a machine of the same size, ending in the waveforms that circuit produces.

198 courses/csse2010/state-machines.pdf

Q23 — Tracing a state machine

Consider the state machine represented by the state diagram shown in question 22 (see Q22). What sequence of states is traversed if the machine starts in state S0 and A goes through the sequence 1010101 and B is always the inverse of A?

(a) S0→S0→S1→S2→S1→S2→S1→S2

(b) S0→S1→S1→S2→S1→S2→S1→S2

(c) S0→S1→S2→S2→S1→S2→S1→S2

(d) S0→S1→S2→S1→S2→S1→S2→S1

(e) None of the above

Answer

(d)
B is the inverse of A, so the input pairs alternate between $AB = 10$ and $AB = 01$: seven values of A give the sequence $10, 01, 10, 01, 10, 01, 10$.
From the state table derived in Q22, only two columns are ever used:

- $AB = 10$: S0 → S1, S1 → S1, S2 → S1
- $AB = 01$: S0 → S0, S1 → S2, S2 → S2

Starting in S0:

step	AB	from	to
1	10	S0	S1
2	01	S1	S2
3	10	S2	S1
4	01	S1	S2
5	10	S2	S1
6	01	S1	S2
7	10	S2	S1

So the machine goes S0→S1→S2→S1→S2→S1→S2→S1, which is (d) — eight states listed, being the start state plus one per input.
The trap is (a), which begins S0→S0 as if the first input left the machine where it was; but the first input has $A = 1$, and S0 only stays put on $\overline{A}$. Options (b) and (c) each stutter once in the middle, which cannot happen because the input never repeats. See state-machines[199]. For what a traced machine looks like once it is built out of flip-flops and gates, see Beyond the paper below.

Q24 — ALU control settings

Consider the ALU bit slice below (not the same as seen previously).

For an ALU made up of 8 of the bit slices above, which of the following control input settings will result in the ALU output being $\overline{A}.B$ (bitwise)? (The "Right Shift In" control input refers to that for the most significant bit; the "Carry in" control input refers to that for the least significant bit.)

(a) ENA=**1** ENB=**1** INVB=**0** F_2=**0** F_1=**0** F_0=**0** Right Shift In=**1** Carry in=**1**

(b) ENA=**1** ENB=**1** INVB=**0** F_2=**0** F_1=**0** F_0=**1** Right Shift In=**1** Carry in=**1**

(c) ENA=**1** ENB=**1** INVB=**1** F_2=**0** F_1=**0** F_0=**1** Right Shift In=**1** Carry in=**1**

(d) ENA=**1** ENB=**1** INVB=**0** F_2=**0** F_1=**1** F_0=**1** Right Shift In=**1** Carry in=**1**

(e) ENA=**1** ENB=**1** INVB=**1** F_2=**1** F_1=**0** F_0=**1** Right Shift In=**1** Carry in=**1**

Answer

(e)
Two things have to be read off the figure: what the conditioning gates do to A and B, and which function each value of $F_2F_1F_0$ selects.
Conditioning. A goes through one AND gate with ENA and nothing else — note there is no INVA on this slice — so the internal signal is

$$X = A \cdot \text{ENA}$$

B goes through an AND with ENB and then an XOR with INVB, and XORing with 1 is inversion, so

$$Y = (B \cdot \text{ENB}) \oplus \text{INVB}$$

Every option sets ENA = ENB = 1, so $X = A$ and $Y = B \oplus \text{INVB}$; INVB is the only thing left that can change the data.
Function select. Following each mux input back to its source in the figure gives:

$F_2F_1F_0$	input	function
000	0	$X \oplus Y$
001	1	$X \cdot Y$
010	2	$X + Y$
011	3	$\overline{X \cdot Y}$

199

courses/csse2010/state-machines.pdf

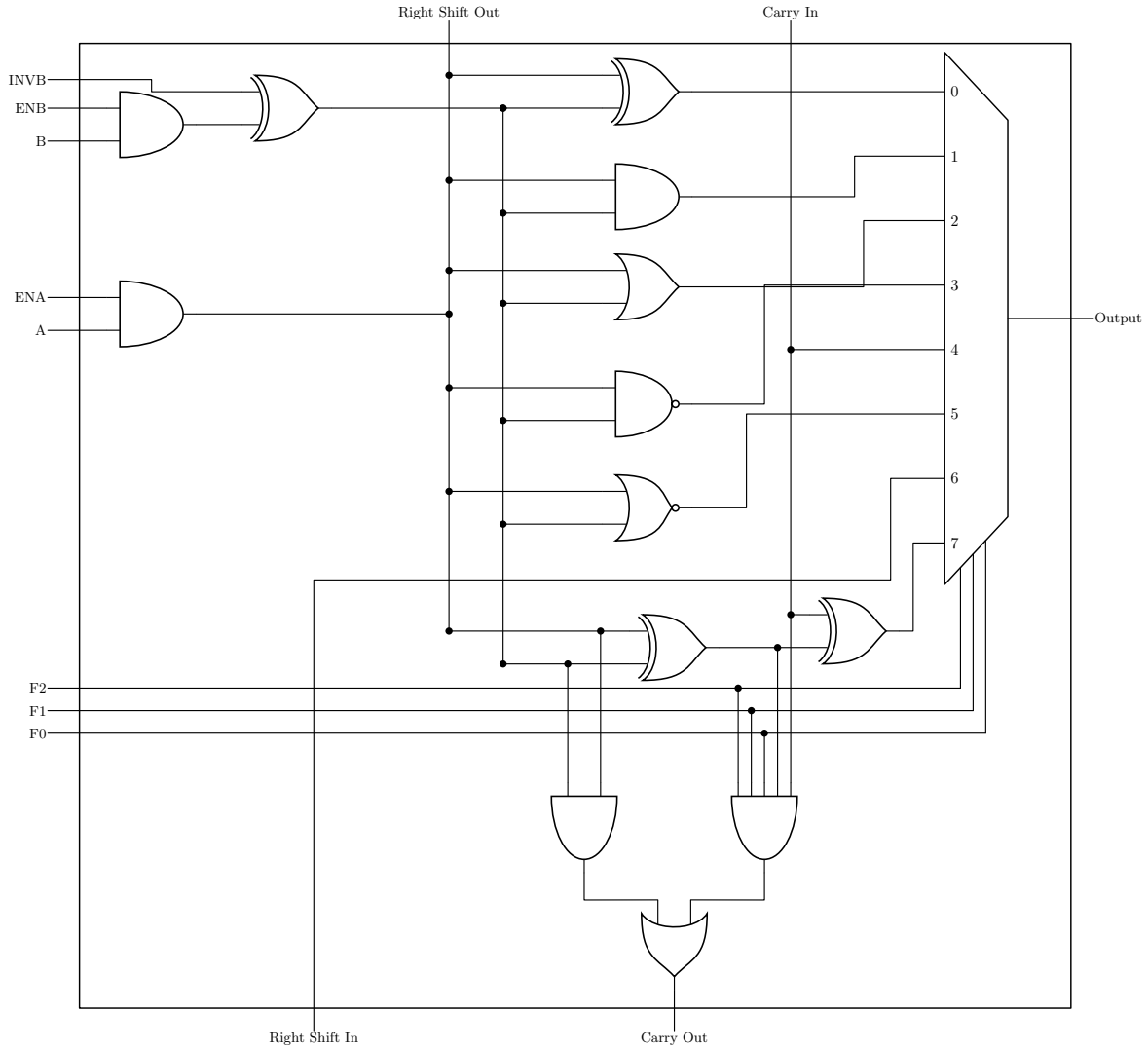


Figure 62: Question 24's ALU bit slice. ENA/ENB/INVB condition A and B into the internal X and Y rails; five gates and the adder compute their functions in parallel, and the eight-way multiplexer picks one under F2 F1 F0. Right Shift Out is the X rail itself, so selecting input 6 (Right Shift In) hands each slice the bit from the slice to its left.

100	4	Carry In
101	5	$\overline{X+Y}$
110	6	Right Shift In
111	7	$X \oplus Y \oplus C_{in}$ (the sum)

Now, how do you get $\overline{A}.B$ from a slice that cannot invert A? Push the complement onto the other side with De Morgan:

$$\overline{A}.B = \overline{A + \overline{B}}$$

So take $X = A$ and $Y = \overline{B}$ (which needs INVB = 1) and select the **NOR**, which is $F_2F_1F_0 = 101$. That is option (e).
Checking the others: (a) is INVB = 0 with the XOR, giving $A \oplus B$; (b) is INVB = 0 with the AND, giving $A.B$; (c) is INVB = 1 with the AND, giving $A.\overline{B}$ — the mirror image of what was asked, and the most tempting wrong answer; (d) is INVB = 0 with the NAND ($F_2F_1F_0 = 011$), giving $\overline{A.B}$, which is the complement of (b) rather than anything with a single complemented variable.
Right Shift In and Carry in are set to 1 in every option, so they cannot distinguish the answers — and neither reaches the output anyway once $F_2F_1F_0$ selects a logic function.
See arithmetic-logic-unit[200].

Q25 — What the ALU cannot do

For an ALU made up of 8 of the bit slices shown in question 24 above (see Q24), which of the following control input function settings can **NOT** be used to output $\overline{B}$ (bitwise) — assuming other control inputs are chosen appropriately?

(a) F_2=**0** F_1=**0** F_0=**0** XOR

(b) F_2=**0** F_1=**1** F_0=**0** OR

(c) F_2=**0** F_1=**1** F_0=**1** NAND

(d) F_2=**1** F_1=**0** F_0=**1** NOR

(e) None of the above

200 courses/csse2010/arithmetic-logic-unit.pdf

Answer

(c)
The question hands you the function each select code picks, so the work is in what the conditioning gates can produce. From Q24, the two internal signals are $X = A \cdot \text{ENA}$ and $Y = (B \cdot \text{ENB}) \oplus \text{INVB}$, so the reachable values are:

ENA	X	ENB	INVB	Y
0	0	0	0	0
1	A	0	1	1
		1	0	B
		1	1	$\overline{B}$

The critical asymmetry is that **X can be forced to 0 but never to 1**, because there is no INVA on this slice. Y, having the INVB stage, can be forced to either constant.
With that in hand:

- **(a) XOR**, $X \oplus Y$. Set ENA = 0 so $X = 0$, and ENB = INVB = 1 so $Y = \overline{B}$. Then $0 \oplus \overline{B} = \overline{B}$, so this one works.
- **(b) OR**, $X + Y$. Same settings: $0 + \overline{B} = \overline{B}$, so this one works too.
- **(d) NOR**, $\overline{X + Y}$. Set $X = 0$ and $Y = B$ (ENB = 1, INVB = 0). Then $\overline{0 + B} = \overline{B}$, which works.
- **(c) NAND**, $\overline{X \cdot Y}$. To get $\overline{B}$ out of a NAND you need $X \cdot Y = B$, which requires $X = 1$ with $Y = B$ (or $X = B$, which is not available either — A and B cannot be swapped). But $X \in \{0, A\}$, so X can never be a constant 1. Every reachable combination fails: $X = 0$ gives a constant 1; $X = A$ gives $\overline{A.B}$, $\overline{A.\overline{B}}$, $\overline{A}$ or 1, none of which is $\overline{B}$. This is the one that cannot be done.

So (c) is the setting that cannot produce $\overline{B}$. The general lesson is the one from arithmetic-logic-unit[201]: forcing a data input to a constant is what lets one ALU compute unary functions, and here that trick works on the B side but only half-works on the A side.

Beyond the paper — from state diagram to wired circuit

Q22 asks for one step of a longer road — state diagram to state table — and Q23 asks you to walk the machine along it. **Learning Lab 8 (*State Machines*)** names the whole road and makes you travel all of it: *"do hand design first — state diagram → state table → next state logic and output logic"*, and then *"enter the circuit into Logisim and simulate it"*.

201 courses/csse2010/arithmetic-logic-unit.pdf

This follows the design task of Learning Lab 8 (State Machines). Its Design Task 1, a detector for the sequence `001`, is worked through state diagram, state tables, encoding, Boolean expressions and logic diagram in week5-lab-state-machines[202]. This section maps the lab's design steps onto this paper's questions and adds a timing diagram of the finished design, using the encoding and expressions from that note.

The pipeline, and where this paper cuts into it

Step	What Lab 8 asks for	Tested here by
1	Draw the **state diagram** — one circle per state, name over output, an arrow per input condition	Q22 reads one; Q23 walks one
2	Write the **2-dimensional state table** — one row per state, one next-state column per input combination	Q22, which is exactly this conversion
3	**Choose a state encoding** — the number of flip-flops and their values in each state. *"Examine different options for state encoding!"*	Q19, Q21 — both are the "how many flip-flops" half of this step
4	Write the **1-dimensional state table** — one row per (state, input) combination, with the D inputs spelled out	not examined directly; it is the working for step 5
5	Derive **Boolean expressions** for the outputs and the flip-flop D inputs, in terms of the Q outputs and the inputs	Q20 runs this backwards, reading a counter's next-state logic off its circuit
6	**Enter the circuit and simulate**	the timing diagram below is what that simulation shows

Step 3 is the one the lab shouts about, and it is worth seeing why. For the `001` detector, the obvious binary assignment $S_0S_1S_2S_3 = 00, 01, 10, 11$ leaves several product terms in each next-state expression and nothing collapses; the assignment week5-lab-state-machines[203] settles on instead —

[202] courses/csse2010/week5-lab-state-machines.pdf
[203] courses/csse2010/week5-lab-state-machines.pdf

$$S_0 = 01 \qquad S_1 = 00 \qquad S_2 = 10 \qquad S_3 = 11$$

— makes one flip-flop input collapse to a bare wire. **Nothing about the state diagram changed; only the numbers written on the states did.**

What the machine does on a clock

$$D_0 = S \qquad D_1 = \overline{Q_0}\,(Q_1 + \overline{S}) = \overline{Q_0} \cdot \overline{\overline{Q_1} \,.\, S} \qquad X = Q_1.Q_0$$

Three chips build that: a `74HCT74` for the two flip-flops, one NAND of a `74HCT00`, and two ANDs of a `74HCT08` — which is the lab's *"are you a bad enough dude to implement a sequence detector with just three chips?"* challenge, answered. The logic diagram is drawn in week5-lab-state-machines[204]; what is worth adding here is the other view of the same machine, because it is the one Q23 and Q17 are really asking about — where each value sits **in time**.

Three things come off that picture, and each is an exam trap somewhere:

- **The input is sampled, not watched.** Nothing happens to Q_1 or Q_0 between edges, however S wriggles. This is the whole content of Q17, where the question is which candidate A waveform could have produced the given Q — and the answer only ever depends on what A was doing at the edges.
- **A Moore output lags the input that caused it.** Bit 3 completes the first `001`, but the machine is only *in* S_3 after edge 3 has happened, so X rises after that edge and stays up for one clock period. The lab's two bit strings are aligned by index, not by time.
- Q_0 **is** S **delayed by one clock**, which is exactly what $D_0 = S$ means once the flip-flop is in the path. Encoding choices that make a D input equal an existing signal turn a gate into a wire, and that is why step 3 of the pipeline is worth the effort.

Two more questions the same design answers

- **"How many flip-flops does this machine need?"** — Q19 and Q21. Count the states, then count the flip-flops: N flip-flops hold 2^N distinct patterns, so $N = \lceil \log_2(\text{number of states}) \rceil$. Four states here, hence the two tracks Q_1 and Q_0 in the waveform.
- **"Moore or Mealy?"** — $X = Q_1.Q_0$ mentions the state and nothing else, which is what makes this Moore. A Mealy output would have S in its expression too, and would then be free to change part-way through a clock period rather than only just after an edge — the flat-topped one-period-wide pulses on the X track are the visible signature of Moore. CSSE2010 sticks to Moore; see state-machines[205].

[204] courses/csse2010/week5-lab-state-machines.pdf
[205] courses/csse2010/state-machines.pdf

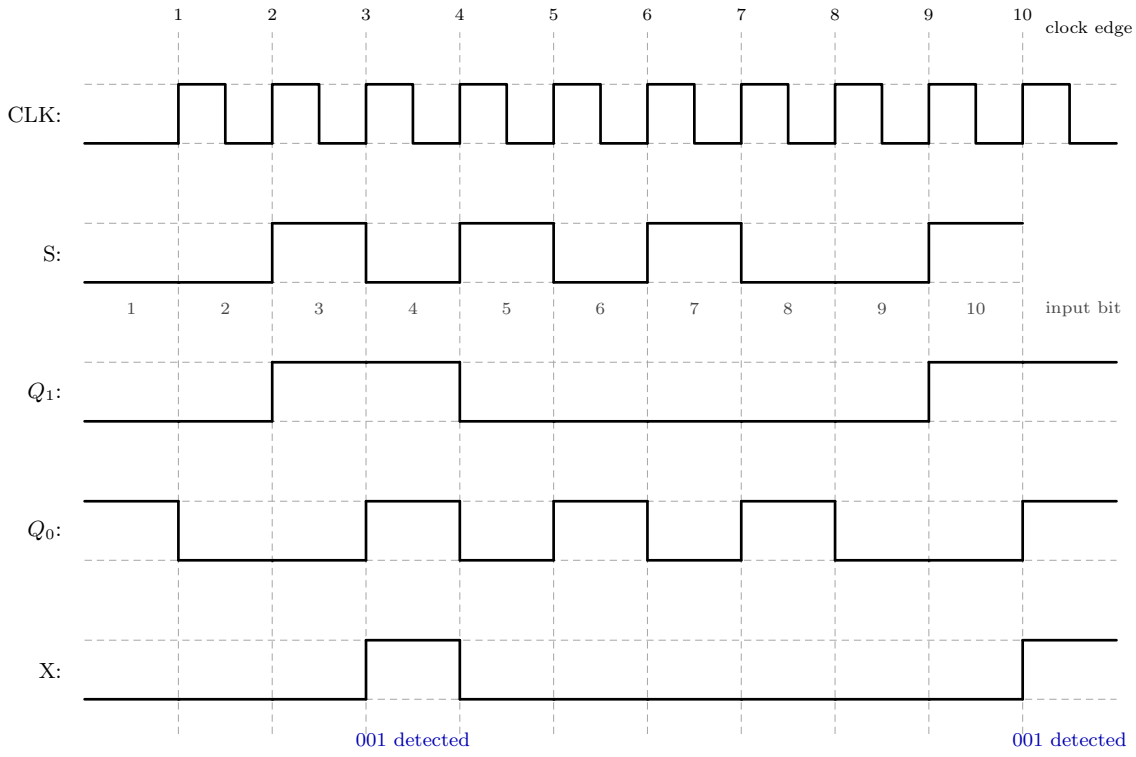


Figure 63: The 001 detector running on the first ten bits of the input sequence from Lab 8's Design Task 1. Read it in columns: the dashed verticals are the ten rising clock edges, the number under the S track is which input bit is being presented, and each of Q1, Q0 and X holds the value it took at the previous edge. The offset is the whole point. Bit 3 completes the first 001 while it sits in the interval before edge 3, but X only goes high after edge 3 has been taken, so the pulse on X appears one column to the right of the bit that caused it - even though the lab's S and X strings are written aligned by index. Q0 is a delayed copy of S, which is what D0 = S means in the time domain, and X is high exactly where Q1 and Q0 are both high. The two blue notes are added here, not on the slide.

Sample Mid-semester Exam #4

This is the fourth of UQ's published sample mid-semester papers for CSSE2010/CSSE7201 Introduction to Computer Systems — the Semester One 2020 St Lucia paper, run as a 60-minute open-book examination with 10 minutes of reading time. The only aids permitted are a Casio FX82-series or UQ-approved calculator; no laptops or phones. There are 25 multiple-choice questions, each worth one mark, with zero marks for an incorrect, missing or multiple answer, so there is no penalty for guessing. The course note is csse2010[206].

> ⚠ Answers are derived, not official
>
> The source paper ships without an answer key, so every answer below was worked out during transcription rather than taken from UQ. Each one shows its reasoning so it can be checked rather than trusted; anything uncertain is flagged. Treat a disagreement between your working and the answer here as a reason to check both.

How to use this

Two ways. For a **timed run**, work through all 25 questions in 60 minutes without opening any of the answer callouts — on the website they stay collapsed until clicked, so the paper reads as a clean exam. For **targeted practice**, use the question index to pick out only the questions for one week or one concept, and read the working straight away.

Note that the PDF build renders every callout expanded, so the answers are always visible there. The site version is the one to practise from.

Question index

Q	Topic	Week	Concept notes
1	Excess-64 representation	1	binary-number-representations[207]
2	Radix conversion	1	binary-number-representations[208]
3	Ones' complement range	1	binary-number-representations[209]
4	Properties of representations	1	binary-number-representations[210]

[206] courses/csse2010/csse2010.pdf
[207] courses/csse2010/binary-number-representations.pdf
[208] courses/csse2010/binary-number-representations.pdf
[209] courses/csse2010/binary-number-representations.pdf
[210] courses/csse2010/binary-number-representations.pdf

Q	Topic	Week	Concept notes
5	Reading a gate circuit	1	logic-gates[211]
6	Boolean simplification	2	boolean-algebra[212]
7	Circuit for an expression	1	logic-gates[213]
8	Truth table for an expression	2	boolean-algebra[214]
9	Function from a timing diagram	2	timing-diagrams[215]
10	NAND built from NOR gates	2	boolean-algebra[216] · logic-gates[217]
11	Equivalent expressions	2	boolean-algebra[218]
12	D flip-flop input from output	3	flip-flops-and-latches[219] · timing-diagrams[220]
13	Two's complement addition	2	binary-number-representations[221]
14	Overflow detection	2	binary-number-representations[222]
15	Overflow and a zero result	2	binary-number-representations[223]
16	Full adder from multiplexers	2	combinational-logic-blocks[224]
17	Synchronous counter sequence	4	counters[225] · flip-flops-and-latches[226]

[211] courses/csse2010/logic-gates.pdf
[212] courses/csse2010/boolean-algebra.pdf
[213] courses/csse2010/logic-gates.pdf
[214] courses/csse2010/boolean-algebra.pdf
[215] courses/csse2010/timing-diagrams.pdf
[216] courses/csse2010/boolean-algebra.pdf
[217] courses/csse2010/logic-gates.pdf
[218] courses/csse2010/boolean-algebra.pdf
[219] courses/csse2010/flip-flops-and-latches.pdf
[220] courses/csse2010/timing-diagrams.pdf
[221] courses/csse2010/binary-number-representations.pdf
[222] courses/csse2010/binary-number-representations.pdf
[223] courses/csse2010/binary-number-representations.pdf
[224] courses/csse2010/combinational-logic-blocks.pdf
[225] courses/csse2010/counters.pdf
[226] courses/csse2010/flip-flops-and-latches.pdf

Q	Topic	Week	Concept notes
18	Multiplexer-fed register pair	3	sequential-circuits[227] · combinational-logic-blocks[228]
19	Tracing a state machine	4	state-machines[229]
20	State diagram to state table	4	state-machines[230]
21	Flip-flops for a state machine	4	state-machines[231]
22	Multiplexer as a logic function	2	combinational-logic-blocks[232]
23	Which circuit is a latch	3	flip-flops-and-latches[233]
24	ALU control settings	5	arithmetic-logic-unit[234]
25	ALU operation coverage	5	arithmetic-logic-unit[235]

Q1 — Excess-64 representation

What is -34_{10} expressed in 7-bit excess-64 format?

(a) 0011110

(b) 0100010

(c) 1011101

(d) 1011110

(e) 1100010

[227] courses/csse2010/sequential-circuits.pdf
[228] courses/csse2010/combinational-logic-blocks.pdf
[229] courses/csse2010/state-machines.pdf
[230] courses/csse2010/state-machines.pdf
[231] courses/csse2010/state-machines.pdf
[232] courses/csse2010/combinational-logic-blocks.pdf
[233] courses/csse2010/flip-flops-and-latches.pdf
[234] courses/csse2010/arithmetic-logic-unit.pdf
[235] courses/csse2010/arithmetic-logic-unit.pdf

Answer

(a)
In excess-k (biased) representation the stored bit pattern is the unsigned binary of value$+k$. Here $k = 64$, so

$$-34 + 64 = 30$$

and $30_{10} = 16 + 8 + 4 + 2 = 11110_2$, which padded to seven bits is 0011110.
A quick sanity check on the other options: $0100010_2 = 34$, which is the *unsigned* 34 rather than the biased -34; $1011110_2 = 94 = 30 + 64$, i.e. the pattern you get if you add the bias twice. See binary-number-representations[236].

Q2 — Radix conversion

95_{16} is equal to

(a) 10100101_2

(b) 137_8

(c) 141_{10}

(d) 225_8

(e) None of the above

Answer

(d)
$95_{16} = 9 \times 16 + 5 = 149_{10}$. In binary that is $1001\,0101_2$ (each hex digit expands to four bits: $9 \to 1001$, $5 \to 0101$).
Regrouping those bits in threes from the right for octal: $010\ 010\ 101 \to 225_8$. Checking: $2 \times 64 + 2 \times 8 + 5 = 128 + 16 + 5 = 149$.
The distractors are all near misses: $10100101_2 = 165$, $137_8 = 95$ (the *digits* of the hex number read as a decimal value), and 141_{10} is simply wrong.

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

Q3 — Ones' complement range

A 4-bit number in ones' complement format can represent integers from

(a) -15 to 15

(b) -8 to 7

(c) -7 to 7

(d) -7 to 8

(e) 0 to 15

Answer

(c)
In ones' complement the negative of a value is its bit-by-bit complement, so the positive patterns $0000 \ldots 0111$ cover $0 \ldots 7$ and their complements $1111 \ldots 1000$ cover $-0 \ldots -7$. The range is therefore -7 to $+7$, with two representations of zero (0000 and 1111) — which is exactly why ones' complement has one fewer distinct value than two's complement's -8 to $+7$ (option (b)). See binary-number-representations[237].

Q4 — Properties of representations

Which of the following statements about number representations is false?

(a) The 8-bit number 01100110 is even when considered to be in unsigned, signed-magnitude, ones' complement, two's complement or excess-128 formats

(b) An n-bit excess 2^{n-1} number can be converted to two's complement representation by inverting the most significant bit

(c) An 8-bit ones' complement number can be converted to 9-bit ones' complement representation by prepending a copy of the most significant bit

(d) An 8-bit two's complement number can be converted to 9-bit two's complement representation by prepending a copy of the most significant bit

(e) None of the above

237 courses/csse2010/binary-number-representations.pdf

Answer

(e)
All four statements are true, so the answer is "none of the above".
(a) 01100110 has a 0 in the least significant bit. Unsigned it is 102; as signed-magnitude the sign bit is 0 so it is +102; in ones' complement and two's complement the leading 0 again means +102; in excess-128 it is $102 - 128 = -26$. All five values are even. (Parity of the interpreted value is really just the LSB in the first four formats, and biasing by the even number 128 cannot change parity either.)
(b) For an n-bit excess-2^{n-1} number the stored pattern is $v+2^{n-1}$, and in two's complement it is $v \bmod 2^n$. Adding 2^{n-1} modulo 2^n flips exactly the most significant bit and leaves the rest alone, so inverting the MSB converts between the two.
(c) and **(d)** are both sign extension. Copying the sign bit preserves the value in ones' complement ($-3 = 11111100 \to 111111100$, still the complement of 000000011) and in two's complement ($-3 = 11111101 \to 111111101 = 512 - 3$). Only unsigned and signed-magnitude would break under this rule.

Q5 — Reading a gate circuit

What function is implemented by the following circuit?

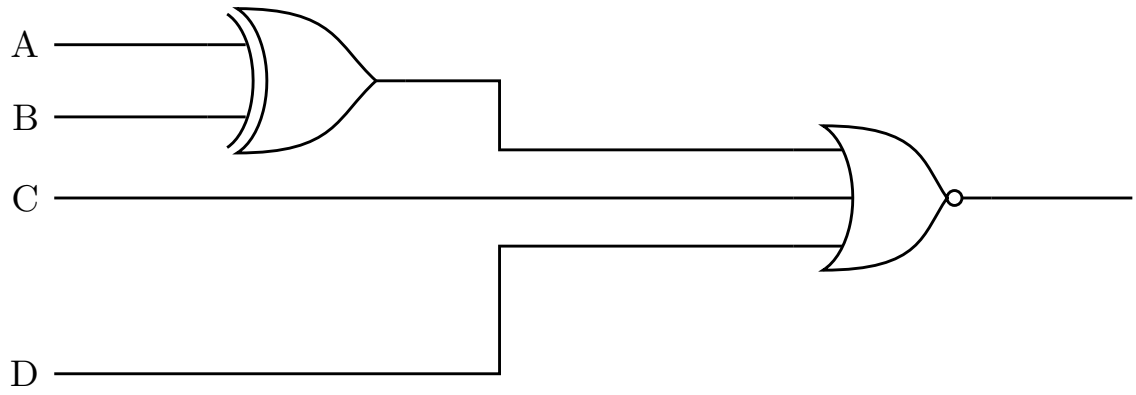


Figure 64: Question 5's circuit: an XOR on A and B feeding a three-input NOR together with C and D.

(a) $(A \oplus B) + C + D$
(b) $\overline{A \oplus B} + \overline{C} + \overline{D}$
(c) $\overline{(A \oplus B) + (C + D)}$

(d) $(A \oplus B).\overline{C}.\overline{D}$

(e) $\overline{\overline{A \oplus B}.(C + D)}$

Answer

(c)
Read it left to right. The first gate has the extra curved bar across its inputs, so it is an XOR: its output is $A \oplus B$. The second gate has the same curved back but a bubble on its output and three inputs, so it is a three-input NOR fed by $A \oplus B$, C and D:

$$X = \overline{(A \oplus B) + C + D}$$

Option (c) is that expression with $C + D$ merely bracketed, and OR is associative, so (c) is the same function.
The near misses are worth checking. Option (a) drops the inversion. Option (b), $\overline{A \oplus B} + \overline{C} + \overline{D}$, is by De Morgan $\overline{(A \oplus B).C.D}$ — the three-input NAND, not the NOR. Option (d) has the right form for De Morgan applied to (c) except that the first term is not inverted; the correct expansion is $\overline{A \oplus B}.\overline{C}.\overline{D}$. Option (e) inverts the wrong subexpression.

Q6 — Boolean simplification

Which of the following logic functions is identical to $\overline{\overline{\overline{A + B} + C} + \overline{B.C}}$?

(a) 0

(b) $B.C$

(c) $A.B.C$

(d) $\overline{B} + \overline{C}$

(e) None of the above

Answer

(b)
The bars in the printed expression nest three deep on the left-hand term: innermost $\overline{A + B}$, then $\overline{\overline{A + B} + C}$, and the whole thing is NORed with $\overline{B.C}$.
Name the two operands of the outer bar $P = \overline{\overline{A + B} + C}$ and $R = \overline{B.C}$, so the function is $\overline{P + R}$. By De Morgan that is $\overline{P}.\overline{R}$, and both complements collapse a level of bars:

$$\overline{P} = \overline{A+B} + C$$
$$\overline{R} = B.C$$

so

$$\begin{aligned} X &= \left(\overline{A+B} + C\right).B.C \\ &= \left(\overline{A}.\overline{B} + C\right).B.C && \text{(De Morgan)} \\ &= \overline{A}.\overline{B}.B.C + C.B.C && \text{(distribute)} \\ &= 0 + B.C && (\overline{B}.B = 0,\ C.C = C) \\ &= B.C \end{aligned}$$

The $\overline{B}.B$ term is what kills the A dependence, which is why the answer has no A in it at all. See boolean-algebra[238].

Q7 — Circuit for an expression

Which of the following logic circuits is an implementation of the function $(A \oplus B) + C + \overline{D}$?

Answer

(b)
The target is an OR of three terms: $A \oplus B$, C, and $\overline{D}$. Every candidate starts with $A \oplus B$, so the question is what the rest of each circuit does with C and D.
In **(b)** the lower gate is an AND with a bubble on its output (a NAND) and a bubble on its C input, so it computes $\overline{\overline{C}.D}$. De Morgan turns that into $C + \overline{D}$, and the final OR adds $A \oplus B$:

$$(A \oplus B) + \left(C + \overline{D}\right) = (A \oplus B) + C + \overline{D} \quad \checkmark$$

The others fail:

- **(a)** the lower gate is an XOR, giving $(A \oplus B) + (C \oplus \overline{D})$. Try $C = 1, D = 0$: the target is 1 but $C \oplus \overline{D} = 1 \oplus 1 = 0$, so the two differ whenever $A \oplus B = 0$.
- **(c)** ends in a NOR, so the whole result is inverted: $\overline{(A \oplus B) + C + D}$.
- **(d)** is $\overline{(A \oplus B) + C} + \overline{D}$ — the NOR is in the middle, so the first two terms come out inverted.
- **(e)** cascades XORs: $((A \oplus B) \oplus C) + \overline{D}$, which is not the same as ORing C in.

238 courses/csse2010/boolean-algebra.pdf

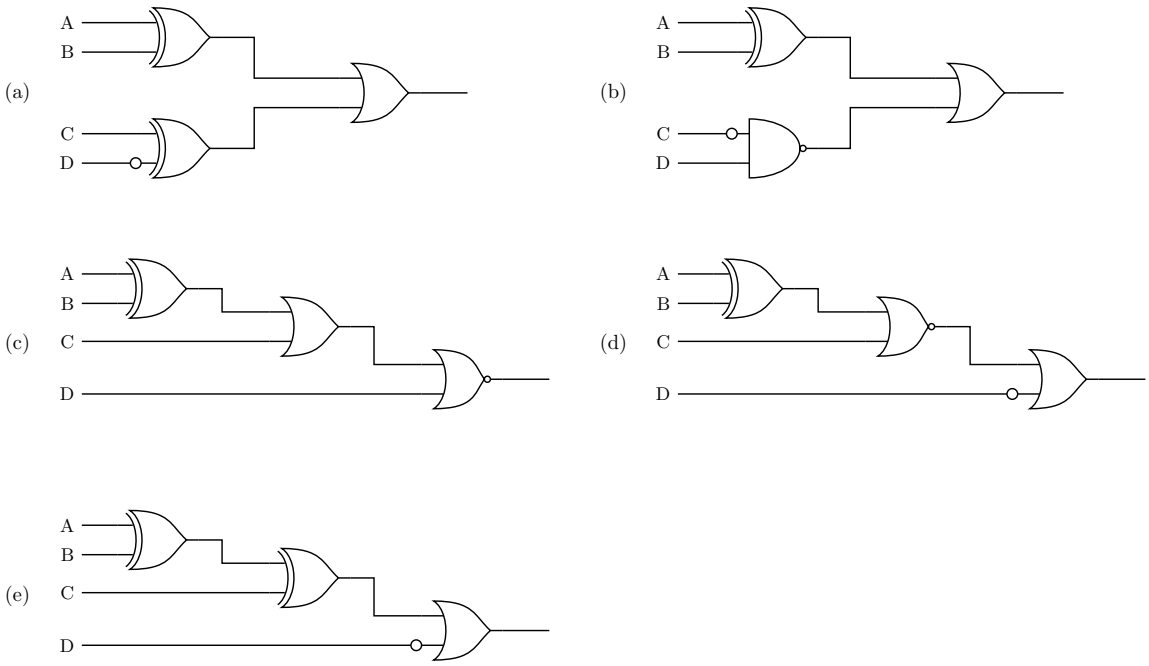


Figure 65: Question 7's five candidate circuits, drawn side by side as in the paper. Small open circles on an input or output lead are inversion bubbles.

Q8 — Truth table for an expression

Which of the following truth tables is equivalent to the function $X = (A.\overline{B} + C).\overline{(B + A.\overline{C})}$

(a)

A	B	C	X
0	0	0	1
0	0	1	1
0	1	0	0
0	1	1	0
1	0	0	0
1	0	1	0
1	1	0	0
1	1	1	0

(b)

A	B	C	X
0	0	0	1
0	0	1	1
0	1	0	0
0	1	1	0
1	0	0	0
1	0	1	1
1	1	0	0
1	1	1	0

(c)

A	B	C	X
0	0	0	0
0	0	1	1
0	1	0	0
0	1	1	0
1	0	0	0
1	0	1	0
1	1	0	0
1	1	1	0

(d)

A	B	C	X
0	0	0	0
0	0	1	1
0	1	0	0
0	1	1	0
1	0	0	0
1	0	1	1
1	1	0	0
1	1	1	0

(e) None of the above

Answer

(d)

Evaluate the two factors separately. Write $P = A.\overline{B} + C$ and $Q = \overline{B + A.\overline{C}}$, so $X = P.Q$:

A	B	C	$A.\overline{B}$	P	$A.\overline{C}$	$B + A.\overline{C}$	Q	X
0	0	0	0	0	0	0	1	0
0	0	1	0	1	0	0	1	**1**
0	1	0	0	0	0	1	0	0
0	1	1	0	1	0	1	0	0
1	0	0	1	1	1	1	0	0
1	0	1	1	1	0	0	1	**1**
1	1	0	0	0	1	1	0	0
1	1	1	0	1	0	1	0	0

X is 1 for $ABC = 001$ and 101 only, which is table **(d)**. Note that Q forces $X = 0$ whenever $B = 1$, so only the four $B = 0$ rows can ever be 1, and of those the two with $C = 0$ fail — at 000 because $P = 0$, and at 100 because $A.\overline{C} = 1$ makes $Q = 0$. Simplified, $X = \overline{B}.C$, which is a neater way to check the table.

Q9 — Function from a timing diagram

What logic function does the following timing diagram represent?

(a) $A \oplus B \oplus C$

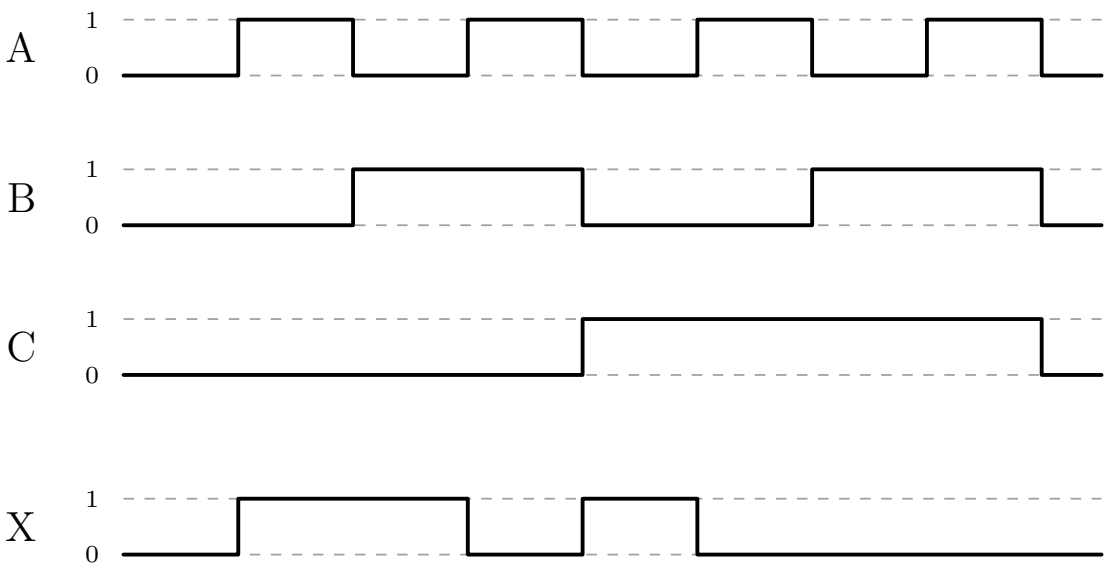


Figure 66: Question 9's timing diagram. A, B and C step through all eight input combinations with A as the fastest signal, and X is the output to be identified.

(b) $A.\overline{B}.\overline{C} + \overline{A}.B.\overline{C} + \overline{A}.\overline{B}.C$

(c) $A.B.\overline{C} + A.\overline{B}.C + \overline{A}.B.C$

(d) $A.\overline{B}.\overline{C} + \overline{A}.B.C + \overline{A}.\overline{B}.C$

(e) None of the above

Answer

(b)
A timing diagram is a truth table drawn sideways (see timing-diagrams[239]). Reading the eight time slots left to right, A toggles every slot, B every two and C every four, so the slots walk through all eight combinations with A as the least significant bit:

Slot	A	B	C	X
0	0	0	0	0
1	1	0	0	**1**
2	0	1	0	**1**
3	1	1	0	0
4	0	0	1	**1**
5	1	0	1	0

6	0	1	1	0
7	1	1	1	0

X is 1 in exactly the three slots where precisely one input is 1, so as a sum of minterms

$$X = A.\overline{B}.\overline{C} + \overline{A}.B.\overline{C} + \overline{A}.\overline{B}.C$$

which is option (b) — an "exactly one input is high" detector.
Option (a) is tempting because $A \oplus B \oplus C$ is also 1 for those three combinations, but three-input XOR is odd parity, so it is 1 for $ABC = 111$ as well, and the diagram shows $X = 0$ there. Options (c) and (d) each pick the wrong three minterms.

Q10 — NAND built from NOR gates

Which of the following circuits implements a 3 input NAND gate ($\overline{A.B.C}$) using only 2-input NOR gates?

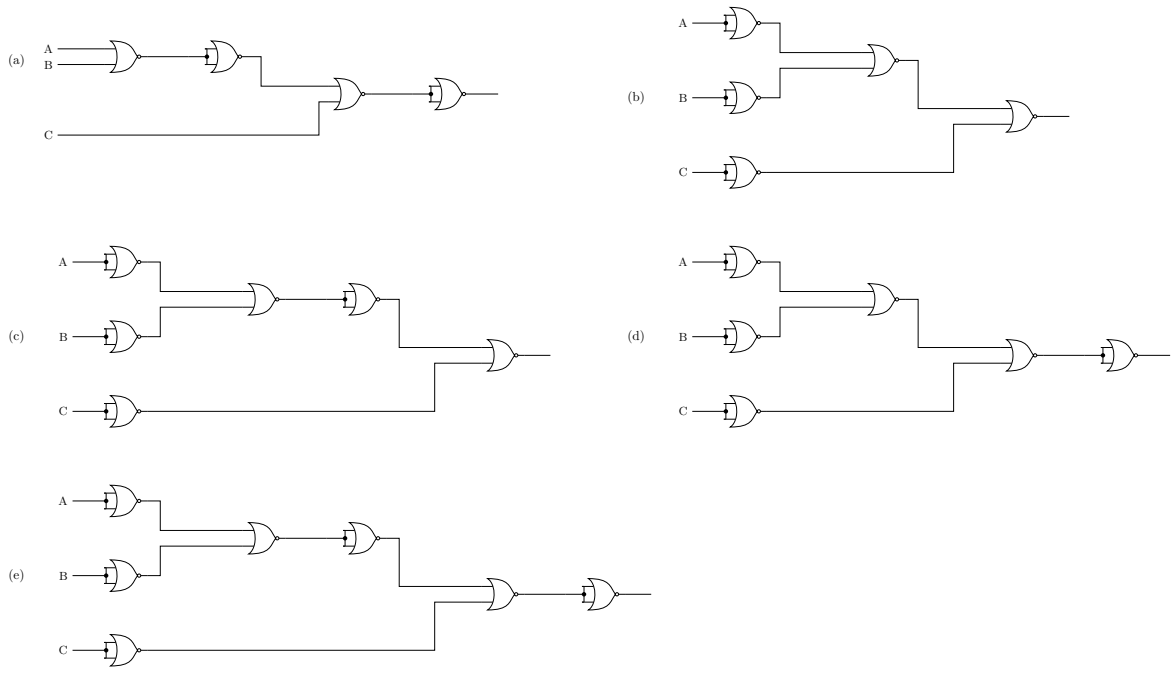


Figure 67: Question 10's five candidate NOR-only circuits. A NOR gate with both inputs tied together (marked by a junction dot) is being used as an inverter.

239 courses/csse2010/timing-diagrams.pdf

Answer

(e)
The target is $\overline{A.B.C} = \overline{A} + \overline{B} + \overline{C}$. A NOR with its inputs tied together is an inverter, since $\overline{x + x} = \overline{x}$, and a NOR of two inverted signals is an AND, since $\overline{\overline{x} + \overline{y}} = x.y$. Those two facts decide every option.
Working each circuit through:

- **(a)** $\overline{A + B}$, inverted back to $A + B$, NORed with C to give $\overline{A + B + C}$, then inverted again: the output is $A + B + C$.
- **(b)** $\overline{A}$, $\overline{B}$, $\overline{C}$; then $\overline{\overline{A} + \overline{B}} = A.B$; then $\overline{A.B + \overline{C}} = \overline{A.B}.C$. At $A = B = C = 0$ this gives 0, but the NAND should give 1.
- **(c)** the same up to $A.B$, then an inverter making $\overline{A.B}$, then $\overline{\overline{A.B} + \overline{C}} = A.B.C$ — the AND, not the NAND.
- **(d)** $A.B$, then $\overline{A.B + \overline{C}} = \overline{A.B}.C$, then an inverter giving $A.B + \overline{C}$. At $A = B = C = 1$ that is 1, but the NAND should give 0.
- **(e)** $A.B$, invert to $\overline{A.B}$, NOR with $\overline{C}$ to get $\overline{\overline{A.B} + \overline{C}} = A.B.C$, then invert once more:

$$\overline{A.B.C} \quad \checkmark$$

Circuit (e) is (c) with one extra inverter on the end, which is exactly the difference between AND and NAND — worth noticing, because (c) is the obvious trap.

Q11 — Equivalent expressions

Consider the logic function represented by the following truth table:

A	B	C	X
0	0	0	1
0	0	1	1
0	1	0	1
0	1	1	0
1	0	0	1
1	0	1	1
1	1	0	0
1	1	1	0

Which of the following expressions is NOT equivalent to this function?

(a) $\overline{B} + \overline{A}.\overline{C}$

(b) $\overline{A}.\overline{B} + \overline{A}.\overline{C} + A.\overline{B}$

(c) $\overline{A}.\overline{B}.\overline{C} + \overline{B}$

(d) $\overline{B}.C + \overline{A}.\overline{C} + \overline{B}.\overline{C}$

(e) $\overline{A}.\overline{B}.\overline{C} + \overline{A}.B.\overline{C} + A.\overline{B}.\overline{C} + \overline{A}.\overline{B}.C + A.\overline{B}.C$

Answer

(c)
First get a canonical form. $X = 1$ on the rows 000, 001, 010, 100, 101. Grouping the four rows with $B = 0$ collapses them completely, leaving one extra term for row 010:

$$\begin{aligned} X &= \overline{A}.\overline{B}.\overline{C} + \overline{A}.\overline{B}.C + A.\overline{B}.\overline{C} + A.\overline{B}.C + \overline{A}.B.\overline{C} \\ &= \overline{B} + \overline{A}.B.\overline{C} && \text{(the four } B = 0 \text{ rows are all of } \overline{B}) \\ &= \overline{B} + \overline{A}.\overline{C} && (x + \overline{x}y = x + y \text{ with } x = \overline{B}) \end{aligned}$$

So the function is $\overline{B} + \overline{A}.\overline{C}$, which is option (a) verbatim. Checking the rest:

- **(b)** $\overline{A}.\overline{B} + A.\overline{B} = \overline{B}$, leaving $\overline{B} + \overline{A}.\overline{C}$. Equivalent.
- **(c)** $\overline{A}.\overline{B}.\overline{C}$ is contained in $\overline{B}$, so by absorption the whole expression is just $\overline{B}$. That loses row 010 ($A = 0$, $B = 1$, $C = 0$), where the table says $X = 1$ but $\overline{B} = 0$. **Not equivalent** — this is the answer.
- **(d)** $\overline{B}.C + \overline{B}.\overline{C} = \overline{B}(C + \overline{C}) = \overline{B}$, leaving $\overline{B} + \overline{A}.\overline{C}$. Equivalent.
- **(e)** is the sum of exactly the five minterms where $X = 1$. Equivalent by construction.

Q12 — D flip-flop input from output

Consider the flip-flop circuit shown below and the associated CLK and output (Q) signals. Which timing diagram shows input (A) values over time which would have resulted in the given output (Q).

Answer

(e)
The triangle on the clock input makes this a positive-edge-triggered D flip-flop, so the only thing that matters is the value of A at the instant of each *rising* clock edge; everything A

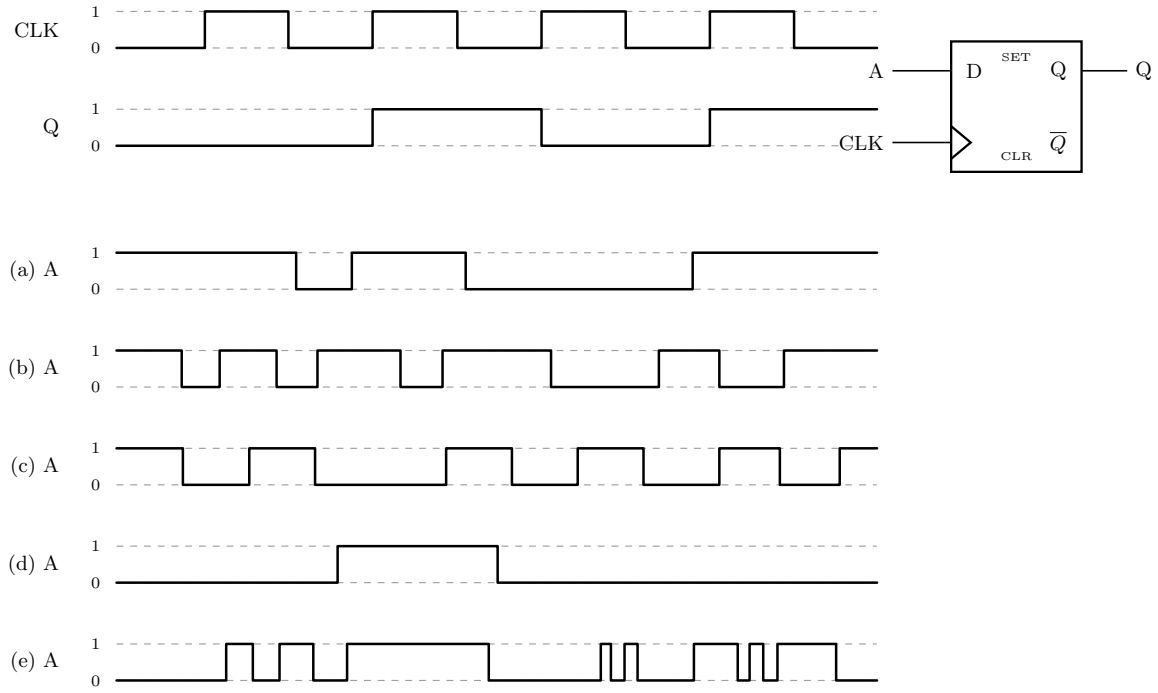


Figure 68: Question 12's given CLK and Q waveforms and the positive-edge-triggered D flip-flop that produced them, with the five candidate A waveforms below.

does between edges is invisible. There are four rising edges in the diagram, and Q tells us what A must have been at each:

Rising edge	Q after the edge	so A at the edge
1st	0 (unchanged)	0
2nd	1	1
3rd	0	0
4th	1	1

So A must read 0, 1, 0, 1 at the four rising edges. Checking each candidate at those four instants:

- **(a)** is high at the first rising edge — needs 0.
- **(b)** gives 0, 1, **1**, … — wrong at the third edge, where its pulse has not yet fallen.
- **(c)** gives 0, **0**, … — it is low across the second rising edge.
- **(d)** gives 0, 1, 0, **0** — its single pulse has long finished by the fourth edge.
- **(e)** gives 0, 1, 0, 1 at the four edges.

Waveform (e) is deliberately full of narrow glitches, and that is the point of the question: none of them land on a rising clock edge, so the flip-flop never sees them. See flip-flops-and-latches[240].

Q13 — Two's complement addition

What's the result of adding the 6-bit two's complement numbers 101101 and 110011?

(a) 000000

(b) 010000

(c) 100000

(d) 111110

(e) 111111

Answer

(c)
Add the bit patterns as plain binary and discard the carry out of the top bit, which is

240 courses/csse2010/flip-flops-and-latches.pdf

what two's complement addition amounts to:

```
  101101
+ 110011
--------
1 100000
```

Every column from the least significant up produces a carry, and the carry out of the top bit is dropped, so the result in six bits is 100000.
Checking in decimal: 101101 has its MSB set, so its magnitude is the two's complement $\overline{101101} + 1 = 010010 + 1 = 010011 = 19$, i.e. the value is -19. Likewise $110011 \to 001100 + 1 = 001101 = 13$, i.e. -13. The sum is -32, and -32 in 6-bit two's complement is 100000.
Two negatives adding to a negative means no overflow, so the six-bit answer is valid. See binary-number-representations[241], and Beyond the paper below for the single chip that does this addition in hardware — the same chip whether you read the operands as signed or unsigned.

Q14 — Overflow detection

Which of the following operations will result in an overflow in 6-bit two's complement arithmetic?

(a) 011111 – 010010

(b) 101011 – 111000

(c) 110100 – 000111

(d) 010101 + 111111

(e) None of the above

Answer

(e)
Six-bit two's complement covers -32 to $+31$. Overflow happens exactly when the true mathematical result falls outside that window, so convert each operand and check:

241 courses/csse2010/binary-number-representations.pdf

	operands (decimal)	result	in range?
(a)	$011111 = 31$, $010010 = 18$	$31 - 18 = 13$	yes
(b)	$101011 = -21$, $111000 = -8$	$-21 - (-8) = -13$	yes
(c)	$110100 = -12$, $000111 = 7$	$-12 - 7 = -19$	yes
(d)	$010101 = 21$, $111111 = -1$	$21 + (-1) = 20$	yes

None of them overflows, so the answer is (e).
The structural shortcut gets there faster in three of the four cases: subtracting two values of the *same* sign, or adding two values of *opposite* sign, can never overflow, because the result lies between the two operands. That immediately clears (a) (positive minus positive), (b) (negative minus negative) and (d) (positive plus negative). Only (c) — negative minus positive — can overflow in principle, and -19 comfortably fits.
The same decision made in gates, from the sign bits of the operands and of the result, is the overflow LED of Beyond the paper below.

Q15 — Overflow and a zero result

Consider the binary addition operation 1010 + 0110. Which of the following statements is true if the numbers are interpreted as two's complement?

(a) The result is negative and no overflow occurs

(b) The result should be negative but overflow occurs

(c) The result is positive and no overflow occurs

(d) The result should be positive but overflow occurs

(e) The result is zero

Answer

(e)
As 4-bit two's complement, 1010 has its sign bit set so its magnitude is $\overline{1010} + 1 = 0101 + 1 = 0110 = 6$, i.e. the value is -6; and $0110 = +6$. The sum is $-6 + 6 = 0$.
The bit-level addition agrees:

```
  1010
+ 0110
------
 10000
```

The carry out of the top bit is discarded, leaving 0000. Note that the carry out being 1

does *not* mean overflow here — for two's complement, overflow is signalled by the carry *into* the sign bit differing from the carry *out* of it, and here both are 1. Adding a positive and a negative number can never overflow anyway, since the result always lies between them.
So the result is exactly zero, which is neither the "negative" of (a) nor the "positive" of (c), and no overflow occurs, ruling out (b) and (d).
On the board this is two lit indicators that mean opposite things: the carry-out LED on and the overflow LED off. Beyond the paper below shows why they are separate outputs.

Q16 — Full adder from multiplexers

Consider the full-adder shown below:

In the multiplexer circuit below, what values do the multiplexer inputs X0,X1,X2,X3 and Y0,Y1,Y2,Y3 need to take on in order for the circuit to implement a full-adder?

(a) X0 = **0**, X1 = **1**, X2 = **1**, X3 = **0**; Y0 = **0**, Y1 = **0**, Y2 = **0**, Y3 = **1**

(b) X0 = C_{in}, X1 = $\overline{C_{in}}$, X2 = $\overline{C_{in}}$, X3 = C_{in}; Y0 = **0**, Y1 = C_{in}, Y2 = C_{in}, Y3 = **1**

(c) X0 = **0**, X1 = $\overline{C_{in}}$, X2 = $\overline{C_{in}}$, X3 = **1**; Y0 = **0**, Y1 = C_{in}, Y2 = C_{in}, Y3 = **1**

(d) X0 = C_{in}, X1 = **1**, X2 = **1**, X3 = C_{in}; Y0 = **0**, Y1 = $\overline{C_{in}}$, Y2 = $\overline{C_{in}}$, Y3 = C_{in}

(e) None of the above

Answer

(b)
Both multiplexers are steered by the same select pair, $S_1 = A$ and $S_0 = B$, so input n of each multiplexer is the one selected when AB is the two-bit number n. Each data input therefore has to be the full-adder output *as a function of* C_{in} *alone*, for that one combination of A and B. This is the standard "multiplexer as a function generator" trick — see combinational-logic-blocks[242].
The full adder is Sum = $A \oplus B \oplus C_{in}$ and $C_{out} = A.B + C_{in}.(A \oplus B)$, so:

$S_1S_0 = AB$	input	$A \oplus B$	Sum $= (A \oplus B) \oplus C_{in}$	C_{out}
00	X0 / Y0	0	C_{in}	0
01	X1 / Y1	1	$\overline{C_{in}}$	C_{in}
10	X2 / Y2	1	$\overline{C_{in}}$	C_{in}
11	X3 / Y3	0	C_{in}	1

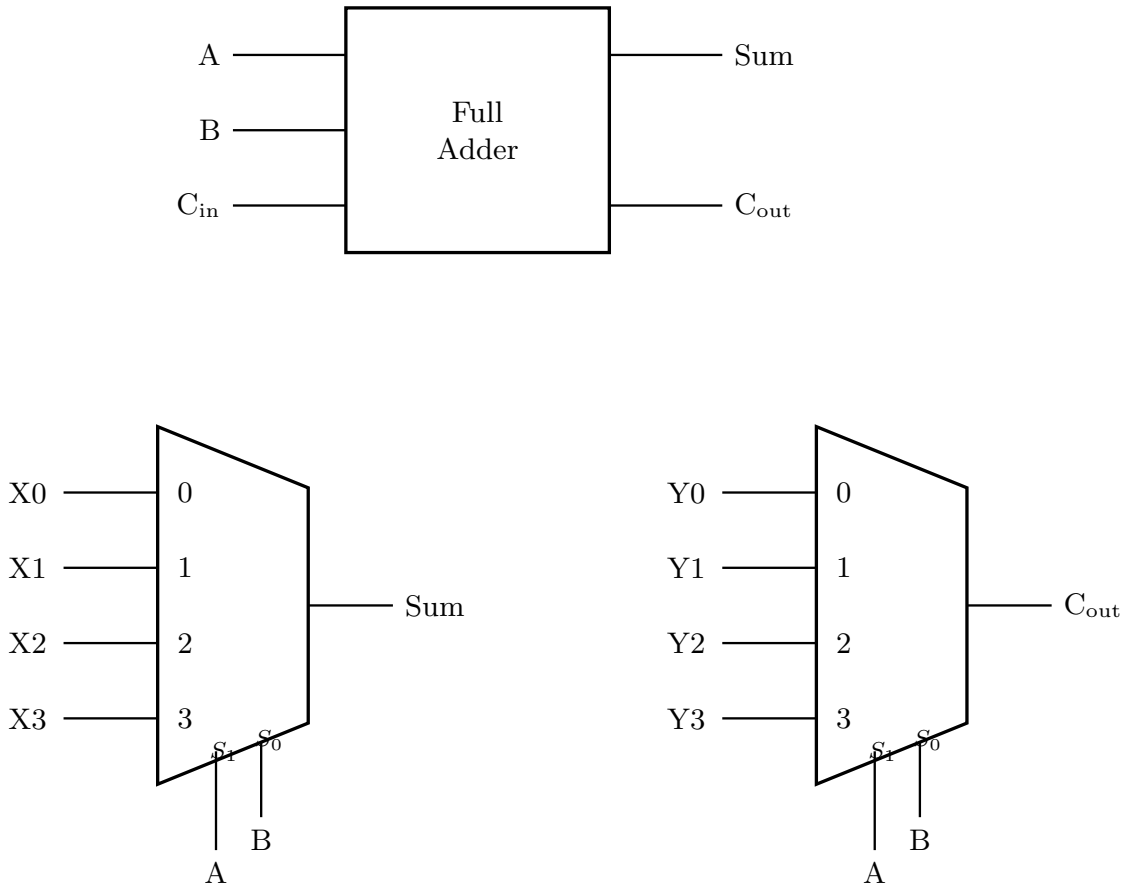


Figure 69: Question 16's full-adder block and the two 4-to-1 multiplexers that are to implement it, with A on the S_1 select and B on S_0.

That is exactly option (b). The pattern is easy to remember: the Sum multiplexer gets C_{in} where $A \oplus B = 0$ and $\overline{C_{in}}$ where $A \oplus B = 1$; the carry multiplexer gets the constants 0 and 1 where A and B agree, and C_{in} where they differ.
This is one bit slice. Four of them chained carry-to-carry are what a `74HCT283` holds in a single package — see Beyond the paper below.

Q17 — Synchronous counter sequence

What sequence will the following synchronous counter count through, if it starts at $Q_1Q_0 = 00$?

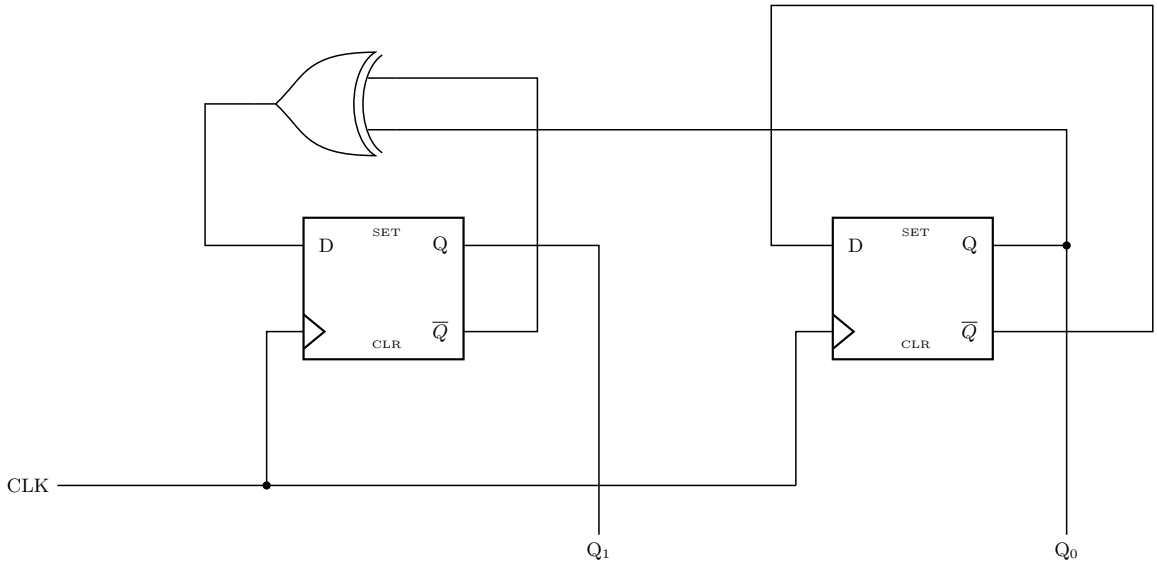


Figure 70: Question 17's counter. Note which flip-flop output feeds the XOR: the wire from FF1's $\overline{Q}$ turns up to the gate, while FF1's Q crosses it and carries on to the Q_1 label.

(a) Q_1Q_0: $00 \to 11 \to 10 \to 01 \to 00 \to \ldots$

(b) Q_1Q_0: $00 \to 11 \to 10 \to 11 \to 10 \to \ldots$

(c) Q_1Q_0: $00 \to 01 \to 10 \to 11 \to 00 \to \ldots$

(d) Q_1Q_0: $00 \to 01 \to 00 \to \ldots$

(e) Q_1Q_0: $00 \to 11 \to 00 \to \ldots$

242 courses/csse2010/combinational-logic-blocks.pdf

Answer

(a)
Read the excitation equations off the wiring. The right-hand flip-flop has its own $\overline{Q}$ routed back round to its D input, so it toggles every clock:

$$D_0 = \overline{Q_0}$$

The XOR feeding the left-hand flip-flop takes Q_0 along the long top wire and — this is the detail the question turns on — $\overline{Q_1}$, not Q_1: the wire that turns up into the gate leaves FF1's $\overline{Q}$ pin, and FF1's Q wire merely crosses it on its way to the Q_1 output. So

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

Both flip-flops share CLK, so the counter is synchronous and both update from the *current* state. Tracing from $Q_1Q_0 = 00$:

Q_1Q_0	$D_1 = Q_0 \oplus \overline{Q_1}$	$D_0 = \overline{Q_0}$	next
00	$0 \oplus 1 = 1$	1	11
11	$1 \oplus 0 = 1$	0	10
10	$0 \oplus 0 = 0$	1	01
01	$1 \oplus 1 = 0$	0	00

So $00 \to 11 \to 10 \to 01 \to 00 \to \ldots$, which is option (a) — a down-counter.
Had the XOR been fed Q_1 instead, D_1 would be $Q_1 \oplus Q_0$ and the sequence would be the ordinary up-count $00 \to 01 \to 10 \to 11$ of option (c), so the two answers differ by exactly which flip-flop pin that wire leaves from. See counters[243].

Q18 — Multiplexer-fed register pair

Consider the circuit below. If A_1 is 0 and A_0 is 1, what are the values of Q_1 and Q_0 after four rising clock edges if FN is 0 for one rising clock edge then 1 for three rising clock edges?

(a) $Q_1 = 0$, $Q_0 = 0$

(b) $Q_1 = 0$, $Q_0 = 1$

(c) $Q_1 = 1$, $Q_0 = 0$

(d) $Q_1 = 1$, $Q_0 = 1$

(e) Insufficient information is provided to determine the values

243 courses/csse2010/counters.pdf

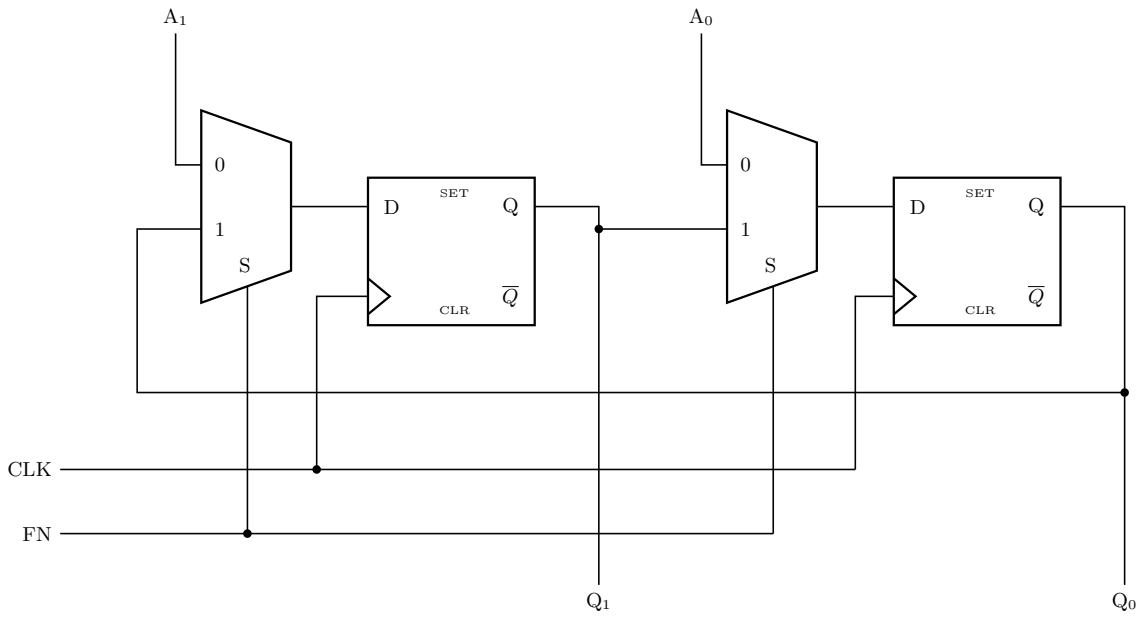


Figure 71: Question 18's circuit: each flip-flop is fed by a 2-to-1 multiplexer that selects between an external input and the *other* flip-flop's output, under the common FN control.

Answer

(c)

Both multiplexers share the select line FN, and each takes the external input on its "0" leg and the *other* flip-flop's output on its "1" leg — Q_0 comes back along the bottom rail into the left multiplexer, and Q_1 crosses over into the right one. So

$$D_1 = \begin{cases} A_1 & \text{FN} = 0 \\ Q_0 & \text{FN} = 1 \end{cases} \qquad D_0 = \begin{cases} A_0 & \text{FN} = 0 \\ Q_1 & \text{FN} = 1 \end{cases}$$

FN $= 0$ is a parallel load of A_1A_0; FN $= 1$ swaps the two bits. With $A_1 = 0$, $A_0 = 1$:

Edge	FN	Q_1Q_0 after
1	0	01 (load A_1A_0)
2	1	10 (swap)
3	1	01 (swap)
4	1	10 (swap)

After the fourth edge $Q_1 = 1$ and $Q_0 = 0$, which is option (c). Note that the initial contents of the flip-flops never matter, because the first edge loads them both — so (e) is wrong.

Q19 — Tracing a state machine

Consider the state machine represented by the state diagram below:

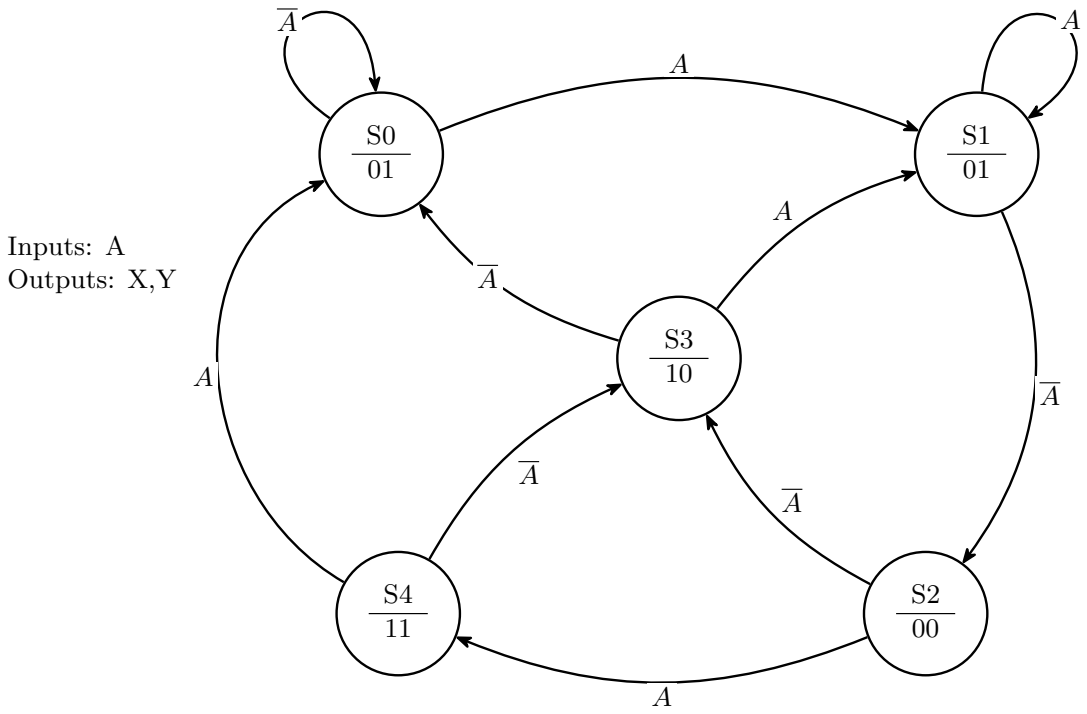


Figure 72: Question 19's five-state machine. Each circle carries the state name above the rule and its X,Y output below.

What sequence of states is traversed if the machine starts in state S1 and A goes through the sequence 101010101?

(a) S1→S2→S4→S3→S1→S2→S4→S3→S1→S2

(b) S1→S1→S2→S3→S1→S2→S4→S3→S1→S2

(c) S1→S1→S2→S4→S3→S1→S2→S4→S3→S1

(d) S1→S1→S2→S4→S0→S0→S1→S2→S4→S3

(e) None of the above

Answer

(c)
First read the transitions off the diagram:

State	on $\overline{A}$	on A
S0	S0	S1
S1	S2	S1
S2	S3	S4
S3	S0	S1
S4	S3	S0

Now walk the input $1, 0, 1, 0, 1, 0, 1, 0, 1$ from S1, listing the *starting* state and then the state after each input:

$$\text{S1} \xrightarrow{1} \text{S1} \xrightarrow{0} \text{S2} \xrightarrow{1} \text{S4} \xrightarrow{0} \text{S3} \xrightarrow{1} \text{S1} \xrightarrow{0} \text{S2} \xrightarrow{1} \text{S4} \xrightarrow{0} \text{S3} \xrightarrow{1} \text{S1}$$

Ten states for nine inputs, which is option (c). The machine falls into the four-state loop S1 → S2 → S4 → S3 → S1 and never visits S0 for this input, because S0 is only reachable from S3 on $\overline{A}$ or S4 on A, and neither happens here.
Option (a) is the same walk with the first (self-loop) step omitted, so it is the trap for forgetting that the initial 1 leaves the machine in S1. See state-machines[244].

Q20 — State diagram to state table

Consider the state diagram below.

Which of the following is an equivalent state table?

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

Inputs: A
Outputs: X,Y,Z

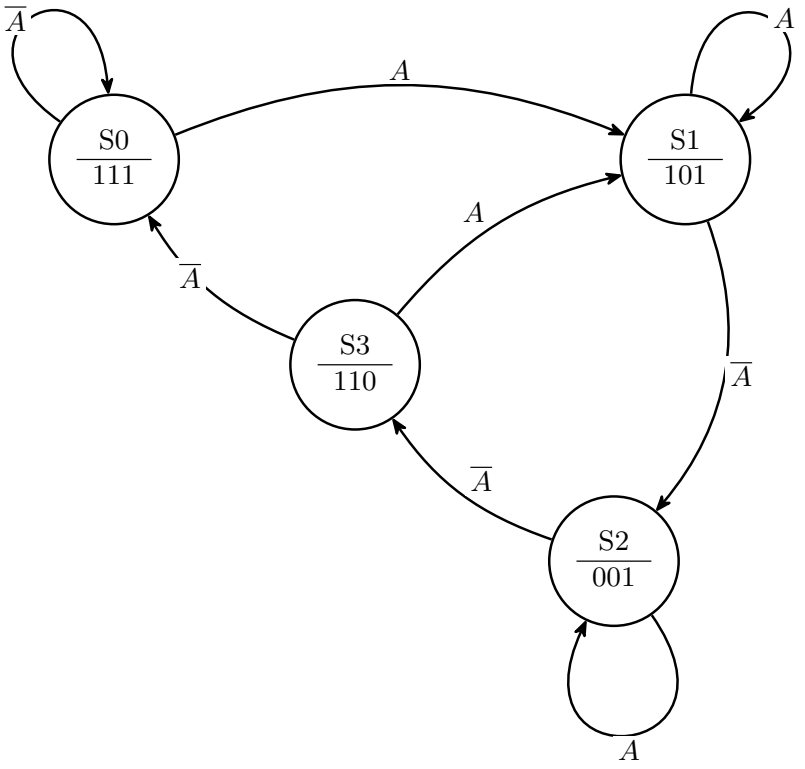


Figure 73: Question 20's four-state machine, with each state's X,Y,Z output printed under its name.

(a)

Current State	Input (A)	Next State	Outputs (X,Y,Z)
S0	0	S0	111
S0	1	S1	111
S1	0	S1	101
S1	1	S2	101
S2	0	S3	001
S2	1	S2	001
S3	0	S0	110
S3	1	S1	110

(b)

Current State	Input (A)	Next State	Outputs (X,Y,Z)
S0	0	S0	111
S0	1	S1	111
S1	0	S2	101
S1	1	S1	101
S2	0	S3	001
S2	1	S2	001
S3	0	S1	110
S3	1	S0	110

(c)

Current State	Next State A=0	Next State A=1	Outputs (X,Y,Z)
S0	S0	S1	111
S1	S2	S1	101
S2	S2	S3	001
S3	S0	S1	110

(d)

Current State	Next State A=0	Next State A=1	Outputs (X,Y,Z)
S0	S0	S1	111
S1	S2	S1	101
S2	S3	S2	001
S3	S0	S1	110

Current State	Next State A=0	Next State A=1	Outputs (X,Y,Z)

(e) None of the above

Answer

(d)
Read the diagram into a table. Every state has one arrow for A and one for $\overline{A}$, and the outputs are printed inside the circles, so this is a Moore machine:

Current State	$A = 0$	$A = 1$	Outputs
S0	S0 (self-loop $\overline{A}$)	S1	111
S1	S2	S1 (self-loop A)	101
S2	S3	S2 (self-loop A)	001
S3	S0	S1	110

That is exactly table (d). The three wrong tables each get one row backwards:

- **(a)** sends S1 to itself on $A = 0$ and to S2 on $A = 1$, i.e. S1's two arrows swapped.
- **(b)** swaps S3's two arrows, sending it to S1 on $\overline{A}$ and S0 on A.
- **(c)** swaps S2's, so the self-loop lands on $A = 0$ instead of $A = 1$.

Q21 — Flip-flops for a state machine

What's the minimum number of flip-flops needed to build the state machine shown in question 20?

(a) 1

(b) 2

(c) 3

(d) 4

(e) 5

Answer

(b)
The flip-flops hold the *state*, not the outputs. The machine of Q20 has four states, and n flip-flops give 2^n distinct state encodings, so the minimum is

$$n = \lceil \log_2 4 \rceil = 2$$

The three outputs X, Y and Z are a combinational function of the two state bits — that is what makes it a Moore machine — so they cost gates, not flip-flops. Answer (c) is the trap for counting the three output bits, and (d) for using one flip-flop per state (a valid one-hot encoding, but not the minimum). See state-machines[245].

Q22 — Multiplexer as a logic function

Consider the following multiplexer. What must the inputs A,B,C,D be so that the multiplexer implements the function $X = \overline{S_0} + S_1$?

(a) A=0, B=1, C=0, D=0

(b) A=0, B=1, C=1, D=1

(c) A=1, B=0, C=1, D=1

(d) A=1, B=1, C=0, D=1

(e) None of the above

Answer

(c)
The select pair S_1S_0 read as a two-bit number picks the data input of that number, so A is chosen for $S_1S_0 = 00$, B for 01, C for 10 and D for 11. Each data input therefore just has to be the value the function takes for that one select combination — with no variables left over, the inputs are all constants:

S_1	S_0	input chosen	$X = \overline{S_0} + S_1$
0	0	A	$1 + 0 = 1$
0	1	B	$0 + 0 = 0$
1	0	C	$1 + 1 = 1$
1	1	D	$0 + 1 = 1$

So A = 1, B = 0, C = 1, D = 1, which is option (c). Reading the data inputs top to bottom gives the function's truth-table column directly — that is the whole trick to using a multiplexer as a function generator, see combinational-logic-blocks[246].

245 courses/csse2010/state-machines.pdf

246

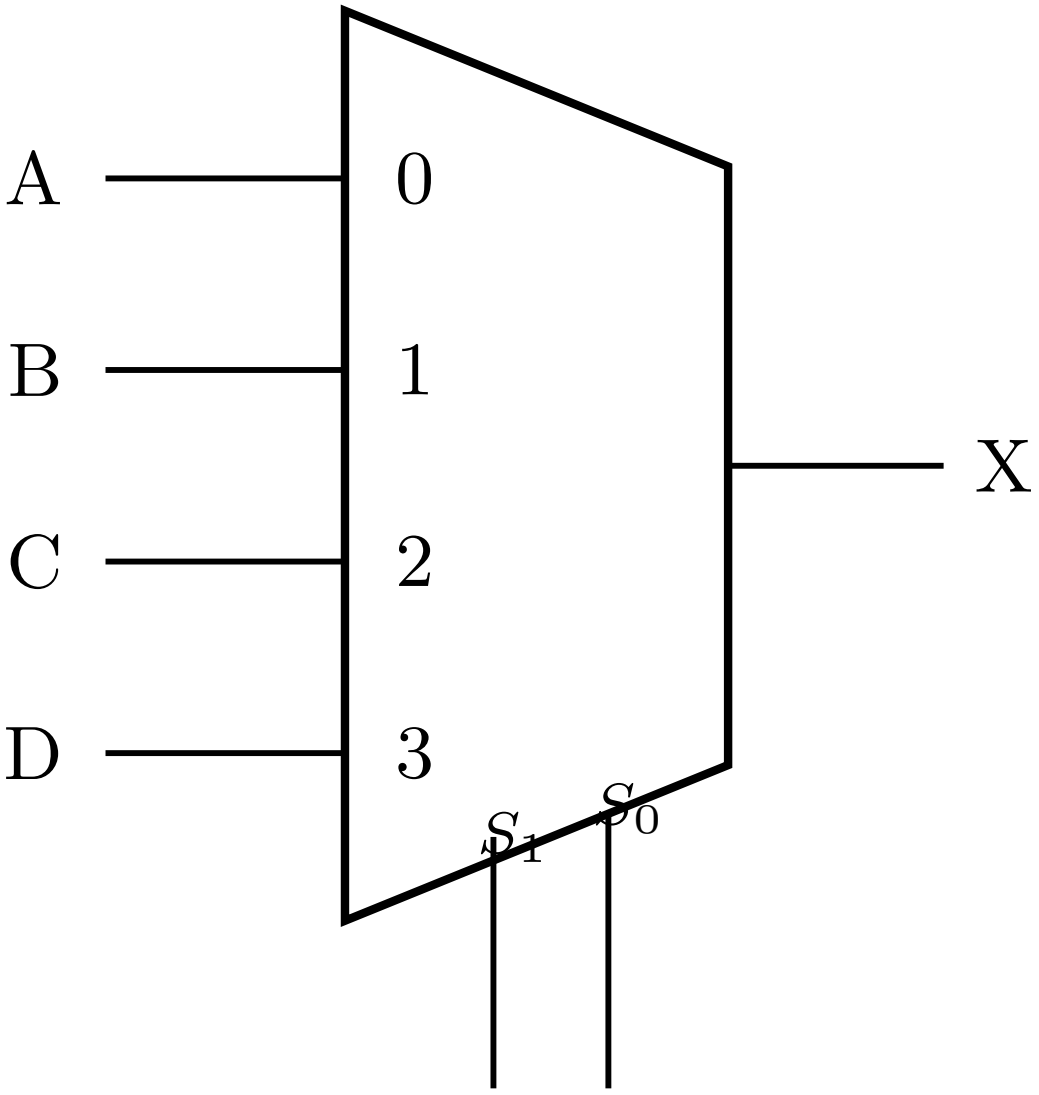


Figure 74: Question 22's 4-to-1 multiplexer, with its four data inputs A, B, C, D and the select pair S_1S_0.

Q23 — Which circuit is a latch

Which of the following circuits is capable of acting as a latch? i.e. it has combinations of inputs (M,N) which allow a bit (0 or 1) to be stored, the latch to be set (Q=1) with a single input transition, and the latch to be cleared (Q=0) with a single input transition?

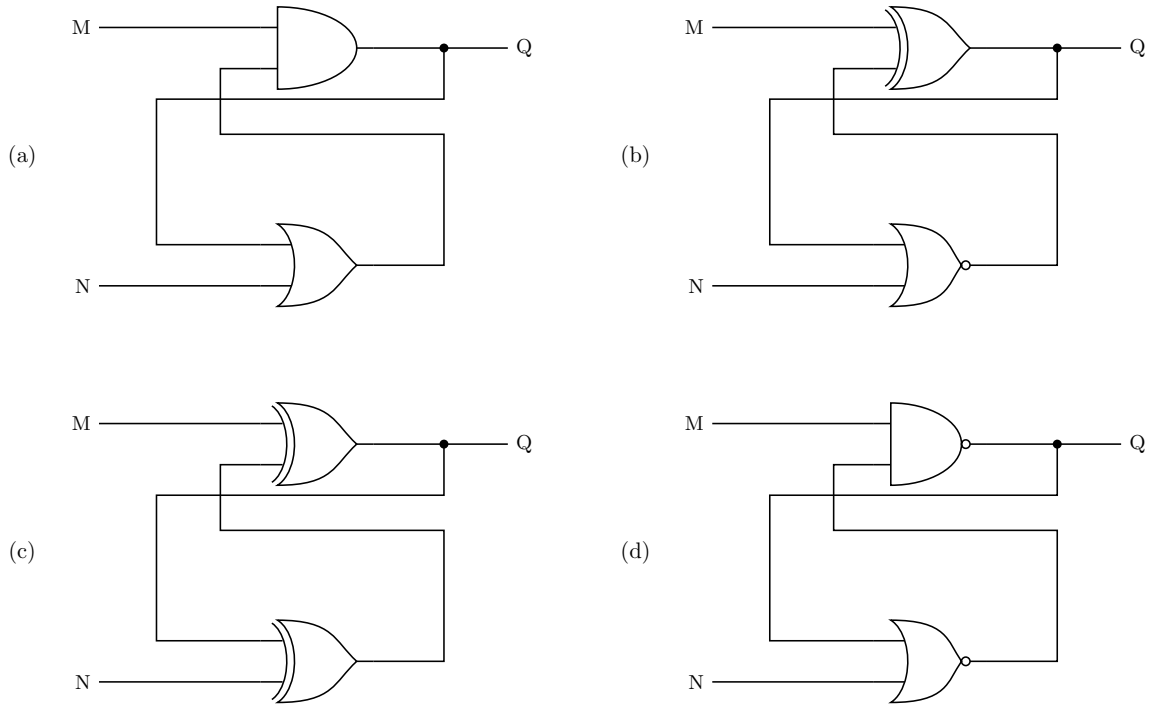


Figure 75: Question 23's four candidate cross-coupled circuits. In each one Q is the top gate's output, and it is fed back to the bottom gate whose output returns to the top gate.

(e) None of the above

Answer

(a)
Each circuit is the same feedback loop with different gates, so write the loop equation and look for a hold state.
(a) — AND on top, OR underneath: $Q = M.(Q + N)$.

courses/csse2010/combinational-logic-blocks.pdf

M	N	$Q = M.(Q + N)$	behaviour
0	0	0	clear
0	1	0	clear
1	0	Q	**hold**
1	1	1	set

Sitting in the hold state $(M, N) = (1, 0)$, taking N to 1 sets it and taking M to 0 clears it — one input transition each, and both return to holding when the input goes back. This is the classic AND–OR latch (N is set, M is an active-low reset), so (a) is the answer.
(b) — XOR on top, NOR underneath: $Q = M \oplus \overline{Q + N}$. At $(1, 0)$ this is $1 \oplus \overline{Q} = Q$, so it holds; at $(1, 1)$ it forces $Q = 1$; but the only clearing combination is $(0, 1)$, which is two transitions away from the hold state, and $(0, 0)$ gives $Q = \overline{Q}$, which oscillates rather than settling.
(c) — XOR on top and bottom: $Q = M \oplus Q \oplus N$. That is stable only when $M \oplus N = 0$, and then $Q = Q$ for *both* $(0, 0)$ and $(1, 1)$ — it can hold but can never be forced to a value, so it can be neither set nor cleared. When $M \neq N$ it oscillates.
(d) — NAND on top, NOR underneath: $Q = \overline{M.\overline{Q + N}}$. With $M = 0$ the output is forced to 1; with $M = 1$ it reduces to $Q + N$, which holds at $N = 0$ and sets at $N = 1$. There is no combination that forces $Q = 0$, so it can never be cleared.
See flip-flops-and-latches[247] for the standard NOR and NAND cross-coupled latches this question is testing against.

Q24 — ALU control settings

Consider the ALU bit slice below.

For an ALU made up of 8 of the bit slices above, which of the following control input settings will result in the ALU output being $\overline{A}.B$? (The "Right Shift In" control input refers to that for the most significant bit; the "Carry in" control input refers to that for the least significant bit.)

(a) ENA = 1, ENB = 1, INVB = 0, $F_2 = 0$, $F_1 = 0$, $F_0 = 0$, Right Shift In = 0, Carry in = 0

(b) ENA = 1, ENB = 1, INVB = 1, $F_2 = 0$, $F_1 = 1$, $F_0 = 1$, Right Shift In = 0, Carry in = 1

(c) ENA = 1, ENB = 1, INVB = 0, $F_2 = 0$, $F_1 = 1$, $F_0 = 1$, Right Shift In = 0, Carry in = 1

(d) ENA = 1, ENB = 1, INVB = 1, $F_2 = 1$, $F_1 = 0$, $F_0 = 1$, Right Shift In = 0, Carry in = 0

(e) ENA = 1, ENB = 1, INVB = 1, $F_2 = 1$, $F_1 = 1$, $F_0 = 1$, Right Shift In = 0, Carry in = 1

247 courses/csse2010/flip-flops-and-latches.pdf

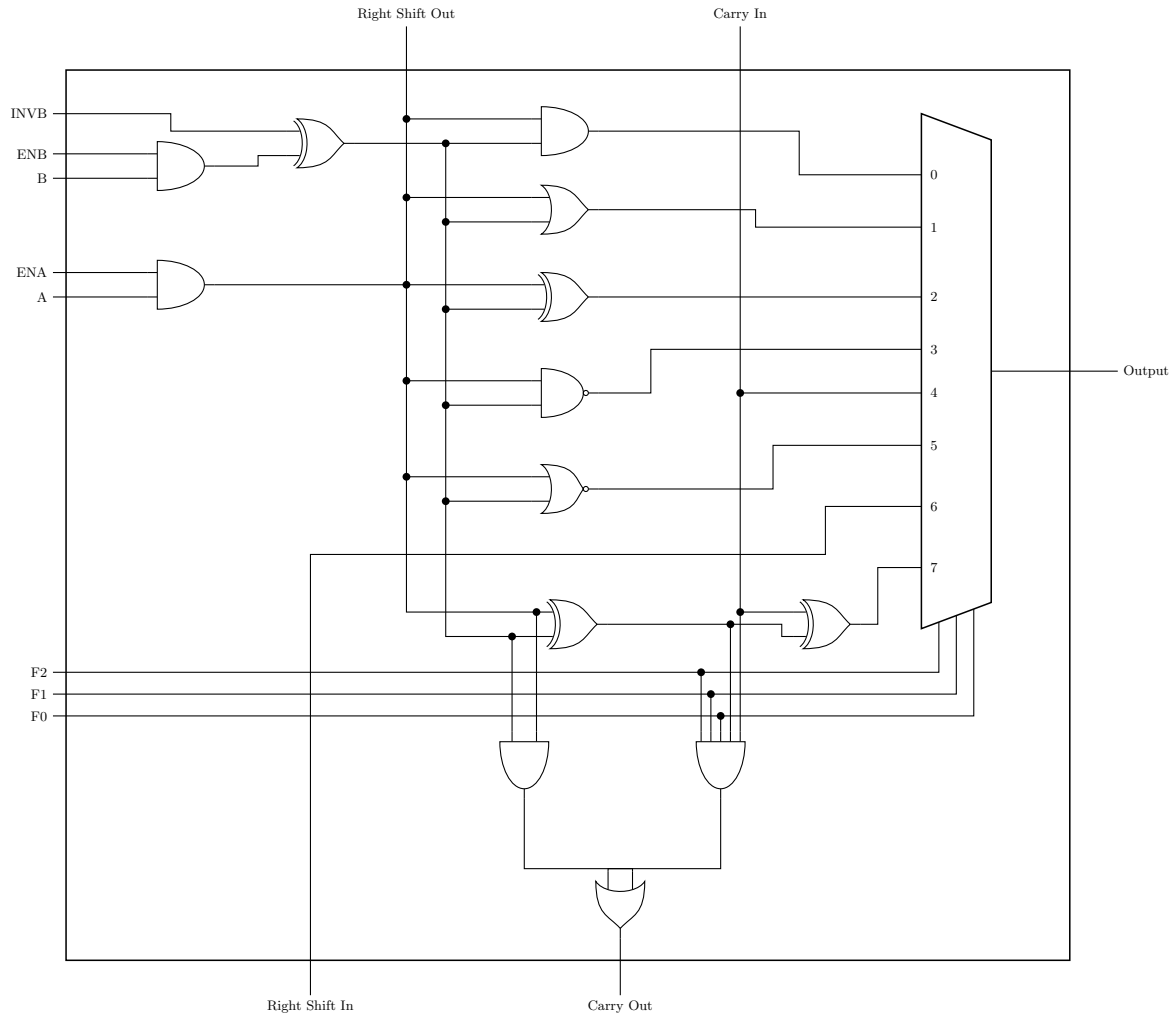


Figure 76: Question 24's ALU bit slice. The ENA/ENB/INVB gates on the left condition A and B into internal lines X and Y; five logic gates and the full adder compute candidate results from them, and the 8-way multiplexer picks one under $F_2F_1F_0$. Note that the conditioned A line is also what leaves as Right Shift Out.

Answer

(d)

Start from the conditioning gates. Writing X for the conditioned A line and Y for the conditioned B line,

$$X = A.\text{ENA} \qquad Y = (B.\text{ENB}) \oplus \text{INVB}$$

so with ENA = ENB = 1, $X = A$ and Y is B when INVB = 0 or $\overline{B}$ when INVB = 1. Note there is no INVA on this slice, so $\overline{A}$ is never available directly — that is the crux of the question.

Tracing each multiplexer input back to the gate that drives it gives the function table:

$F_2F_1F_0$	multiplexer input	Output
000	0	$X.Y$
001	1	$X + Y$
010	2	$X \oplus Y$
011	3	$\overline{X.Y}$
100	4	Carry In
101	5	$\overline{X + Y}$
110	6	Right Shift In
111	7	$X \oplus Y \oplus C_{in}$ (the adder sum)

Since $\overline{A}$ cannot be produced by the input conditioning, the inversion has to come from a gate. De Morgan supplies it:

$$\overline{A}.B = \overline{A + \overline{B}}$$

which is the NOR of A and $\overline{B}$ — i.e. set INVB = 1 so $Y = \overline{B}$, and select the NOR at input 5, $F_2F_1F_0 = 101$. That is option (d), and its Right Shift In and Carry in are irrelevant to a purely logical operation.

Checking the others: (a) selects the AND with $Y = B$, giving $A.B$; (b) and (c) both select the NAND ($F_2F_1F_0 = 011$) and differ only in INVB, giving $\overline{A.\overline{B}} = \overline{A} + B$ and $\overline{A.B}$ respectively; (e) selects the adder with $Y = \overline{B}$ and carry in 1, giving $A \oplus \overline{B} \oplus 1 = A \oplus B$. See arithmetic-logic-unit[248].

248 courses/csse2010/arithmetic-logic-unit.pdf

Q25 — ALU operation coverage

For the ALU in question 24, which of the following bitwise logical operations can NOT be performed?

(a) $\overline{A} + B$

(b) $A \oplus \overline{B}$

(c) $\overline{A} \oplus B$

(d) $\overline{A.B}$

(e) All of the above operations can be performed

Answer

(e)
Using the same reading of the slice as Q24: X can be A or 0, Y can be B, $\overline{B}$, 0 or 1, and the five logic functions available are AND, OR, XOR, NAND and NOR of X and Y. Setting ENA = ENB = 1 and choosing INVB and $F_2F_1F_0$:

Wanted	Rewrite	Settings
(a) $\overline{A} + B$	$\overline{A.\overline{B}}$	INVB = 1, NAND ($F = 011$)
(b) $A \oplus \overline{B}$	as written	INVB = 1, XOR ($F = 010$)
(c) $\overline{A} \oplus B$	$= A \oplus \overline{B}$	INVB = 1, XOR ($F = 010$)
(d) $\overline{A.B}$	as written	INVB = 0, NAND ($F = 011$)

All four are reachable, so the answer is (e).
The one worth pausing on is (c). Complementing either operand of an XOR complements the result, so $\overline{A} \oplus B$ and $A \oplus \overline{B}$ are the same function — both are $\overline{A \oplus B}$, i.e. XNOR — and the slice can produce it even though it has no INVA. The absence of INVA restricts less than it looks like it should: De Morgan and the XOR identity between them let the missing complement be pushed onto B.

Beyond the paper — the adder as a real chip

Q13, Q14 and Q15 all run the same adder by hand, and Q16 builds one out of multiplexers. **Learning Lab 3 (*Binary Arithmetic*)** builds it out of a single part instead, and the schematic is worth having in front of you, because three of this paper's traps are visible in it as separate wires.

This is Task 2 of Learning Lab 3 (Binary Arithmetic), drawn to the rules in circuit-schematics[249] with the symbol and pin numbers from the *Device Pinouts and Symbols* sheet. **The particular signal-to-pin assignments on the IO board are a free choice.** The sheet fixes the ranges (switches on pins 25–32, buttons on 12–15, LEDs on 17–24) but not which switch carries which bit. See week2-lab-binary-arithmetic[250] for the rest of the lab.

The task

Add two 4-bit numbers set on switches, show the 4-bit result and the carry out on LEDs, and put the carry in on a push button. One chip does all the arithmetic: the **74HCT283**, four full adders in a 16-pin package with the intermediate carries C_1, C_2, C_3 joined stage to stage inside it, so only C_0 and C_4 appear on pins.

One adder, two interpretations

The lab's instruction after building it is *"try some unsigned and two's complement test cases — are the results as you expect?"*, and the answer is the point of the whole exercise: **the chip does not know which you meant.** It adds bit patterns. Nothing inside it changes between Q13's two's complement reading and an unsigned one; the same eight switches and the same five LEDs serve both. What changes is only which condition means *the answer on the LEDs is wrong*:

Reading	Range in 4 bits	The answer is wrong when
Unsigned	0 to 15	$C_4 = 1$ — the true sum needed a fifth bit
Two's complement	-8 to $+7$	the operands' sign bits agreed and S_3 disagrees with them

That is why C_4 gets its own LED separate from the sum. It is also precisely the trap in Q15: $1010 + 0110$ produces $C_4 = 1$, and the answer is still exactly right, because the reading is two's complement and the carry out is discarded rather than consulted.

The extension, and what it needs

The lab then asks for an overflow LED as well, with a constraint that changes the problem: *"you don't have access to C3 which is an internal signal. Therefore, you need the overflow output expressed in terms of available information (i.e. A3, B3, S3, C4)."* Reading the schematic

[249] courses/csse2010/circuit-schematics.pdf
[250] courses/csse2010/week2-lab-binary-arithmetic.pdf

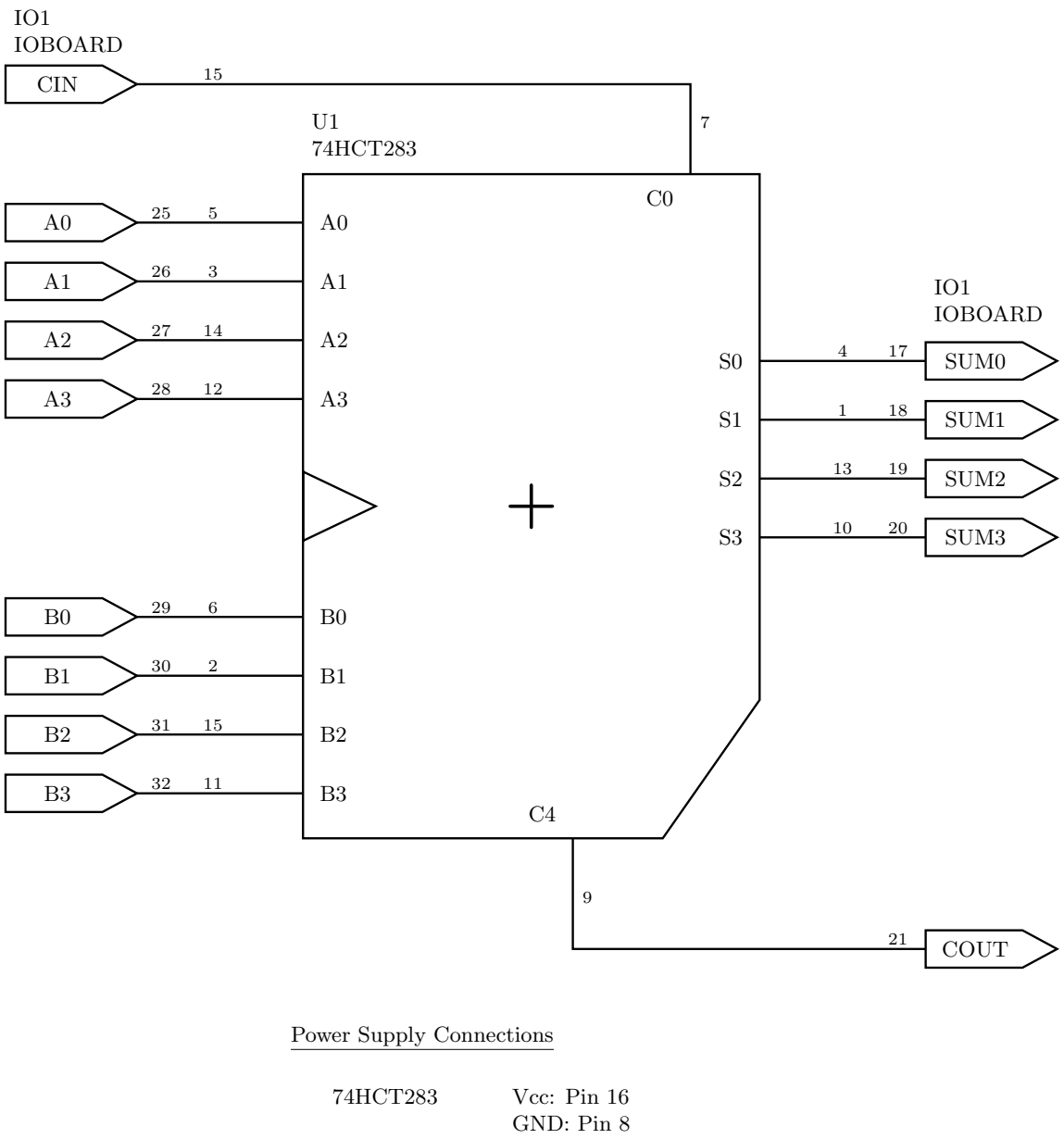


Figure 77: Lab 3 Task 2 as a circuit schematic: a 74HCT283 adding two 4-bit numbers from IO-board switches, with the carry in on a push button and the sum and carry out on LEDs. Everything a schematic is required to carry is here — a unique name on every input and output, the device identifier U1 with its type underneath, a pin number at both ends of every wire, the IO board labelled once per group rather than once per signal, and a power block giving Vcc and GND for the device type. The 74HCT283 is a 16-pin part, so its supply pins are 16 and 8 — not the 14 and 7 of the quad gate packages, which is the mistake this schematic exists to prevent. The input pentagons attach their wire at the point and the output pentagons at the flat back; getting that round the wrong way is one of the errors the drawing guide lists.

explains why — C_3 is joined to the next full adder inside the package and never reaches a pin, so the familiar $V = C_3 \oplus C_4$ cannot be wired.

The two expressions that can be wired from pins are

$$V = A_3 B_3 \overline{S_3} + \overline{A_3}\,\overline{B_3}\,S_3 \qquad \text{and} \qquad V = A_3 \oplus B_3 \oplus S_3 \oplus C_4$$

both of which are derived and drawn in csse2010-sample-midsem-2[251]. The second is the cheaper build — three XOR gates, one `74HCT86` — and it is the reason C_4 is among the signals you are allowed to use. Either one, driven onto an LED, turns Q14 from a decimal calculation into something the board tells you.

Where Q16 fits

Q16's pair of multiplexers implements **one** of the four full adders inside this package. Cascading four of them, carry to carry, is what the 74HCT283 has done for you — see combinational-logic-blocks[252] for the ripple-carry structure and device-pinouts[253] for the chip's internals as the source sheet draws them.

Device Pinouts

Pin numbers for the 74-series logic chips and IO board used in circuit-schematics[254]. 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:

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

[251] courses/csse2010/csse2010-sample-midsem-2.pdf
[252] courses/csse2010/combinational-logic-blocks.pdf
[253] courses/csse2010/device-pinouts.pdf
[254] courses/csse2010/circuit-schematics.pdf

Column	What it is	Where it is used
Logic diagram symbol	the hardware-independent shape	logic-gates[255] and logic diagrams
Circuit schematic symbol	the shape *plus* device ID, device type and pin numbers	circuit-schematics[256]

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.

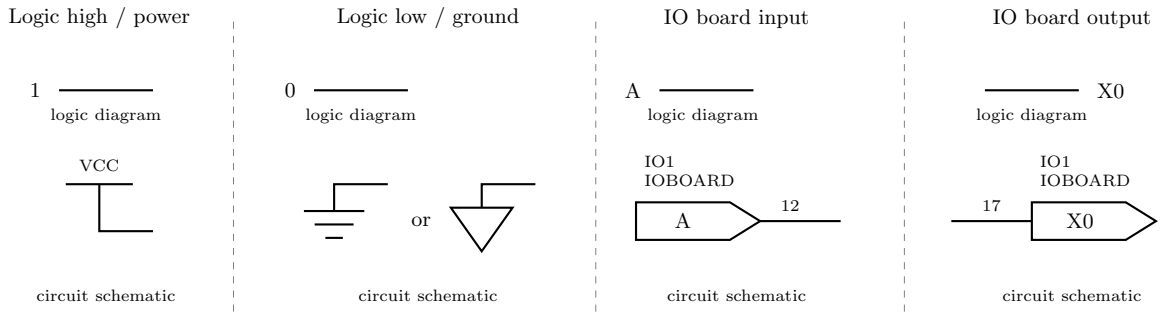


Figure 78: The four non-gate symbols. Logic high and logic low are a bare 1 or 0 on a logic diagram, and a VCC bar or a ground symbol (either of the two accepted forms) on a circuit schematic. The IO board input and output symbols are the same pentagon; the difference is that an input's wire leaves the point and an output's wire arrives at the flat back.

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

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

[255] courses/csse2010/logic-gates.pdf
[256] courses/csse2010/circuit-schematics.pdf

Chip	Function	Gate A	Gate B	Gate C	Gate D	VCC	GND
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.

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

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.

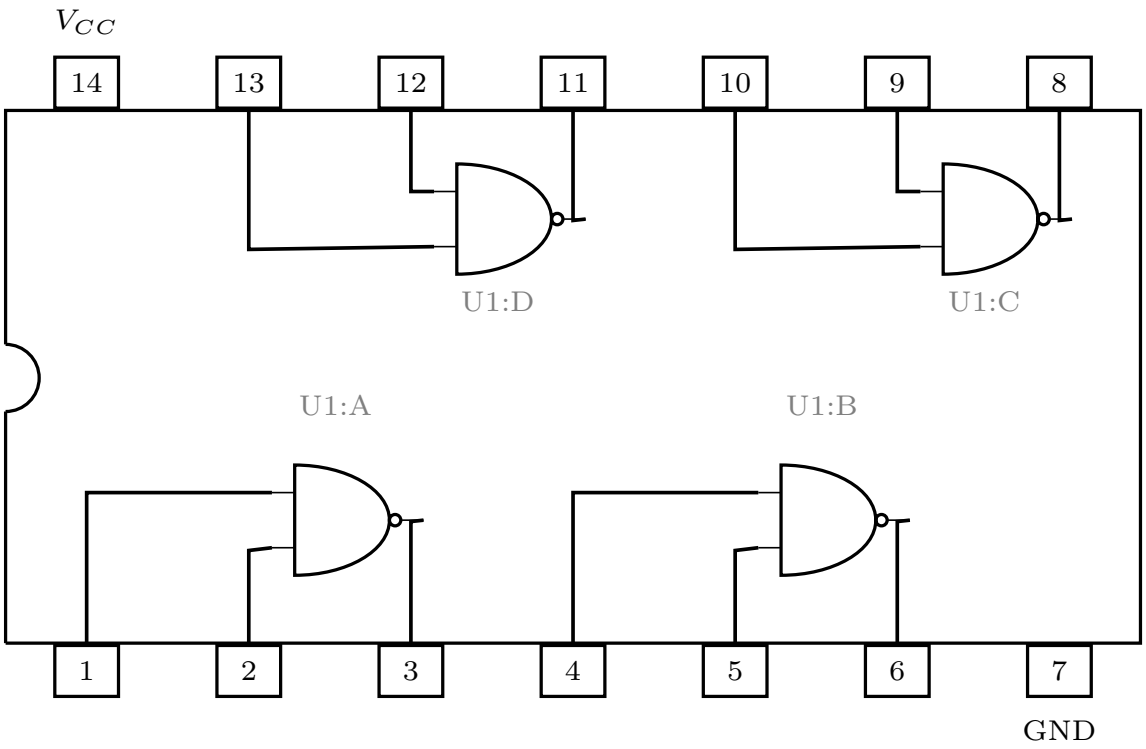


Figure 79: The 14-pin DIP pinout, drawn for the 74HCT00 but identical for the 08, 32 and 86 — only the gate shape inside changes. The notch marks the pin-1 end; pins run 1-7 along the bottom and 14-8 along the top. Reading the internal wiring is what tells you the gates pair pins 1,2 to 3, 4,5 to 6, 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.

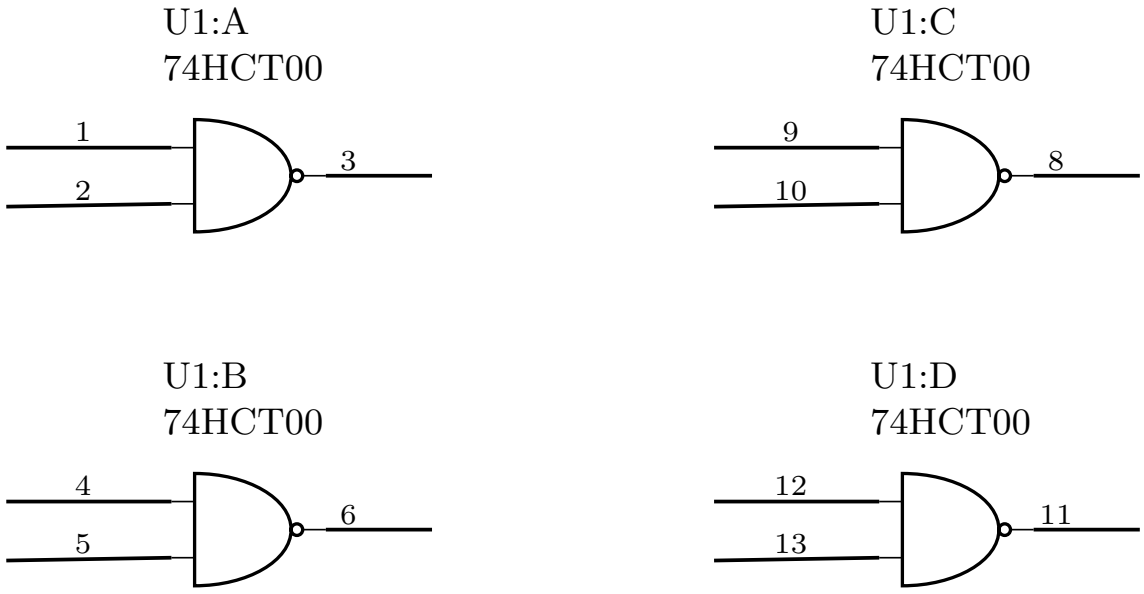


Figure 80: The four gates of a 74HCT00 as circuit schematic symbols: device ID, device type, and a pin number on every terminal. For a 74HCT08, 74HCT32 or 74HCT86 the pin numbers are exactly these; only the gate body changes to AND, OR or XOR.

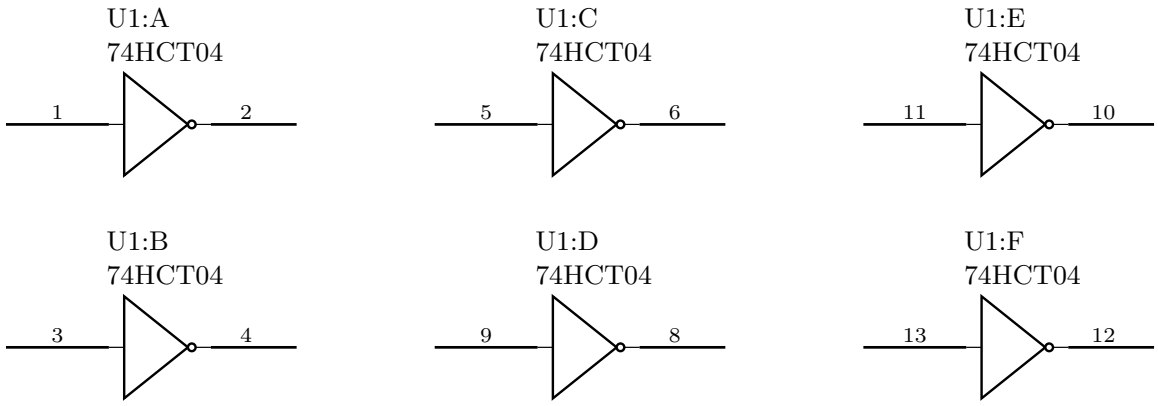


Figure 81: All six inverters of a 74HCT04. The numbering is not a simple run: the first three gates go low-to-high (1 to 2, 3 to 4, 5 to 6) but the last three run the other way (9 to 8, 11 to 10, 13 to 12), because the package is symmetric about its centre.

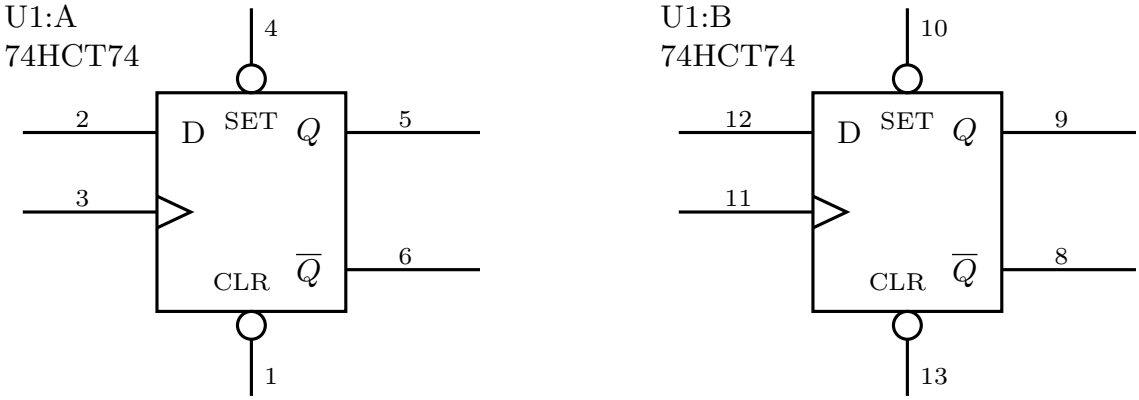


Figure 82: The two flip-flops of a 74HCT74 as circuit schematic symbols. Unlike the 74HCT175 below, each flip-flop has its own clock, its own set and its own clear — nothing is shared. SET and CLR carry bubbles, and the wedge on the left-hand lower input marks the clock.

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[257] 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.

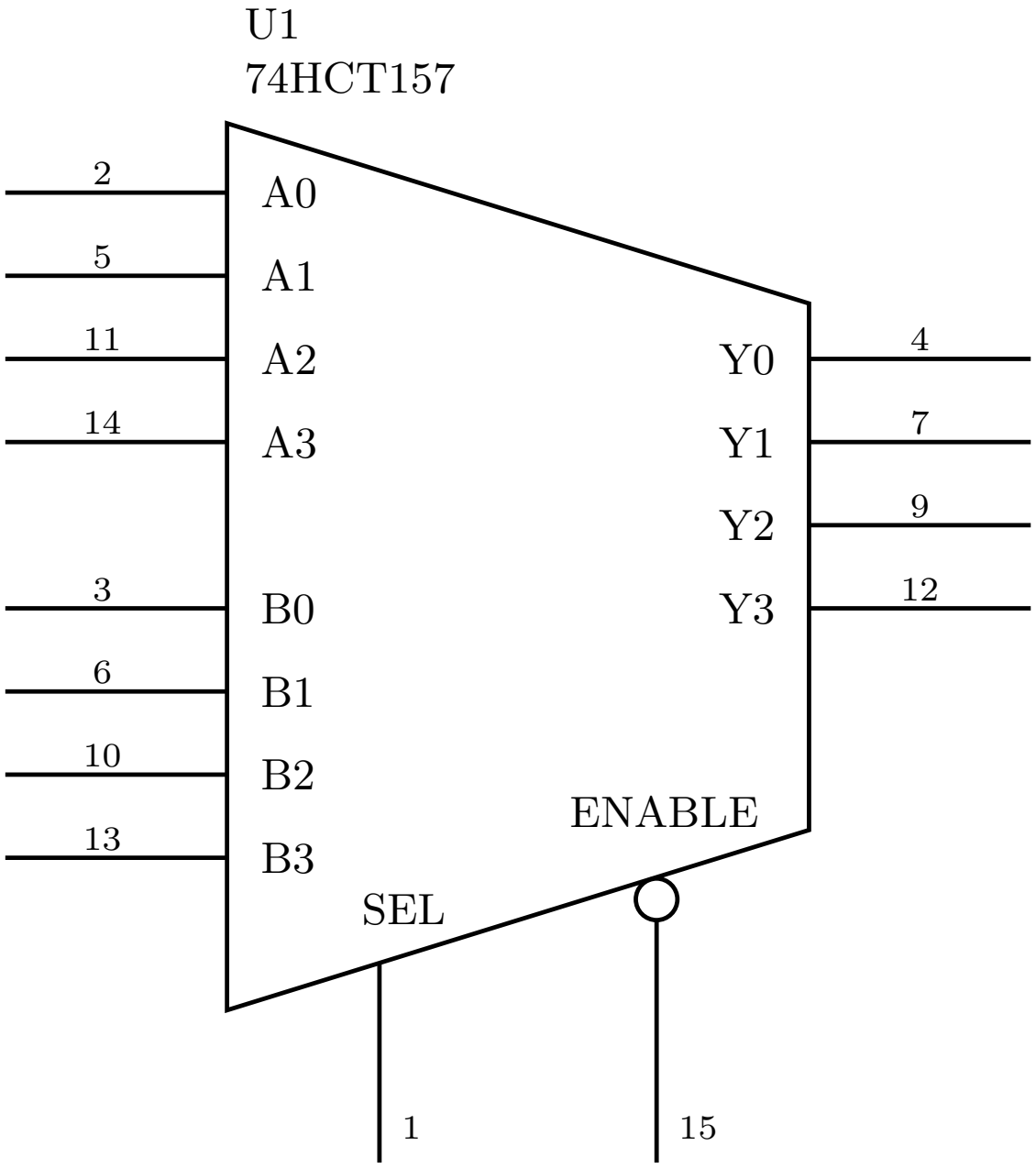


Figure 83: The 74HCT157 as one circuit schematic symbol — the multiplexer trapezoid, tall edge for the eight data inputs and short edge for the four outputs. It is four 2-to-1 multiplexers sharing a single SEL line and a single active-low ENABLE, so it selects between two whole 4-bit words rather than between two individual bits: one value of SEL puts A0-A3 on Y0-Y3 and the other puts B0-B3 there.

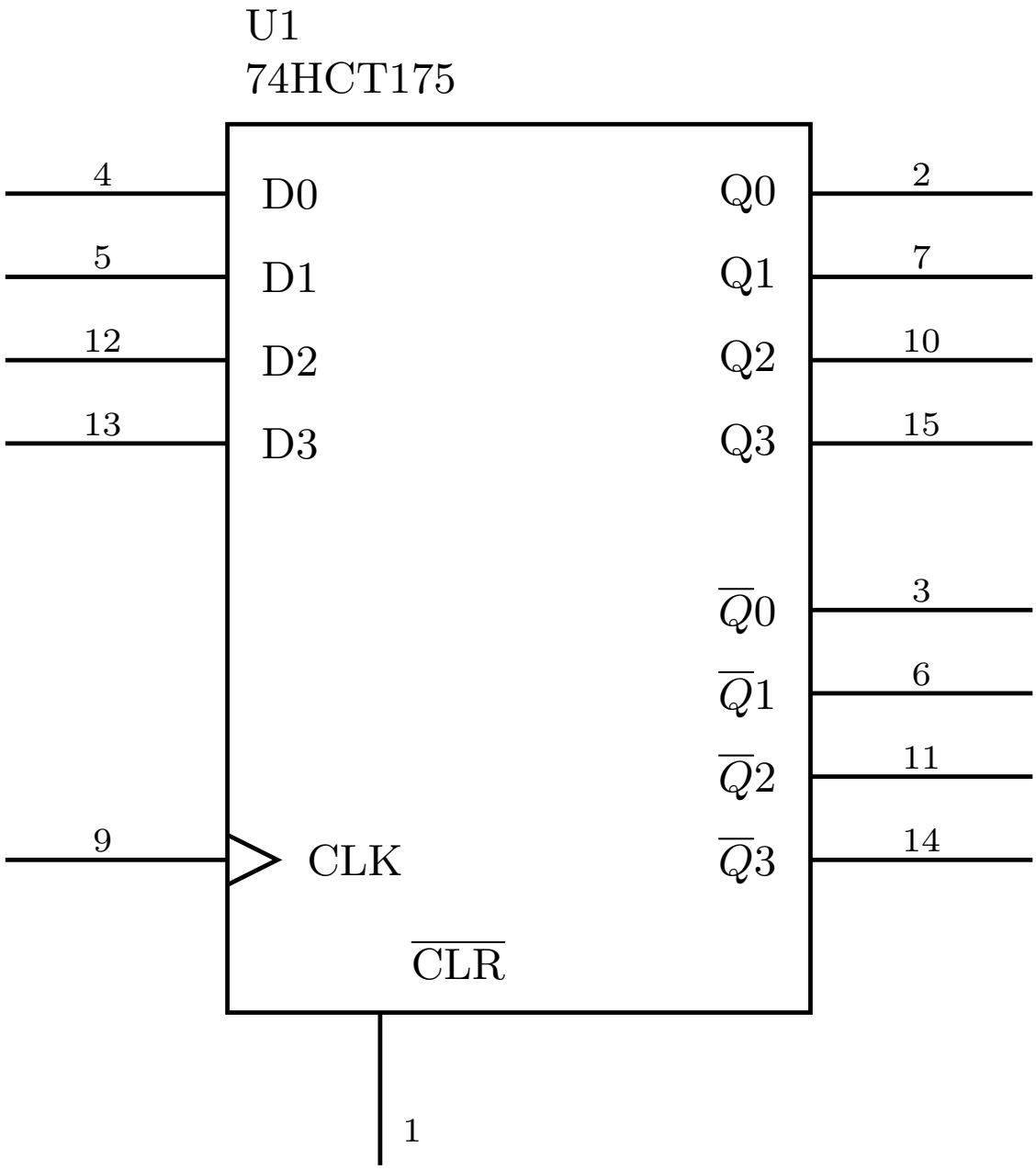


Figure 84: The 74HCT175 as one circuit schematic symbol. CLK (pin 9) and CLR (pin 1) enter once and serve all four flip-flops, which is exactly what a flat pin table hides: this is a 4-bit register, not four independent flip-flops.

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[258].

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 78 and circuit-schematics[259]. 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[260] for what the board does electrically.

[258] courses/csse2010/combinational-logic-blocks.pdf
[259] courses/csse2010/circuit-schematics.pdf
[260] courses/csse2010/io-board.pdf

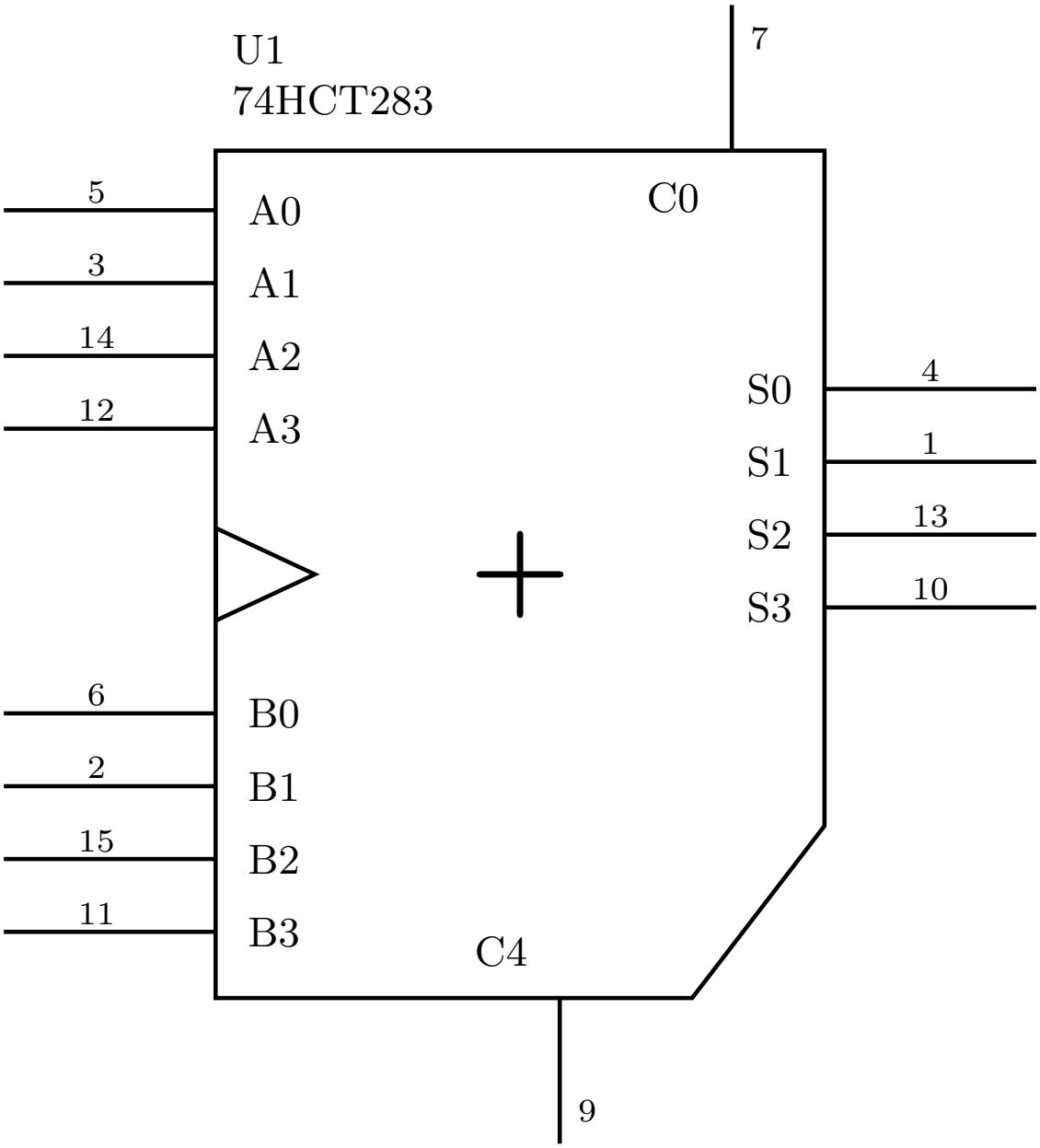


Figure 85: The 74HCT283 as one circuit schematic symbol. Internally it is four full adders with C1, C2 and C3 wired stage to stage inside the package, so only C0 and C4 are brought out to pins.

Fetch-Decode-Execute Cycle

What the control unit actually does: a continuous cycle over the datapath, issuing control signals in the right order for whatever instruction it just read. See cpu-organisation[261] for the blocks involved and arithmetic-logic-unit[262] for the one being driven.

The cycle

The control unit operates in a **continuous cycle** on the datapath:

Stage	What happens
Fetch	Fetch an instruction from memory and bring it inside the CPU. Two special purpose registers involved — **PC** and **IR**
Decode	Resolve the operation (**opcode**) and operands (variables)
Execute	Control signals are issued and the operation takes place in the datapath

Stated as steps:

1. **Fetch** the next instruction from memory into the Instruction Register (**IR**).
2. Change the Program Counter (**PC**) to point to the next instruction.
3. Determine the type of (**decode**) the instruction just fetched — i.e. look at the opcode.
4. **Execute** the instruction.
5. Return to step 1.

Step 2 advances the PC *before* the instruction executes, which is why a jump instruction works by overwriting the PC rather than by suppressing the increment.

The datapath

The question this raises — and the one the control unit exists to answer — is **how do we control this?** Specifically:

- How do we tell the ALU what to do?
- How do we choose what register values are placed on the A and B buses?
- How do we choose which register receives the result from the destination bus?
- How do we know when to read or write memory?

[261] courses/csse2010/cpu-organisation.pdf
[262] courses/csse2010/arithmetic-logic-unit.pdf

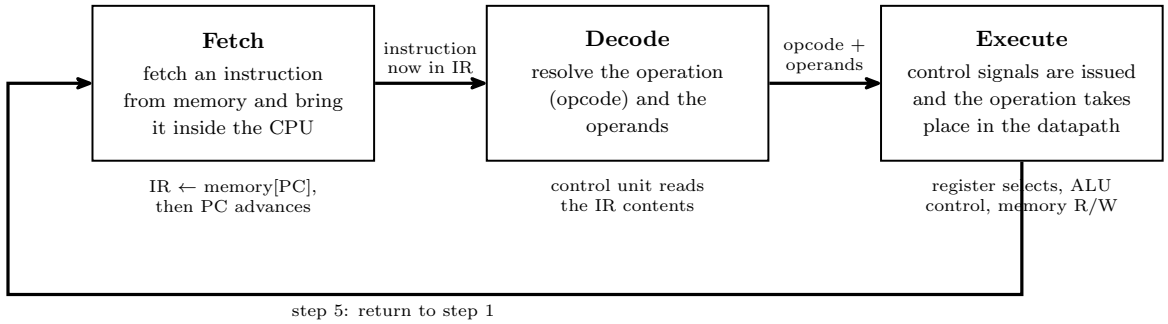


Figure 86: The cycle as a loop: a three-stage pipeline. Each box holds a one-line description of its stage, each arrow carries what the stage leaves behind for the next one, and the notes under the boxes summarise steps 1–4 above.

Control signals

Each of those questions is answered by a bundle of wires out of the control unit. For a machine with 8 registers and the ALU of arithmetic-logic-unit[263]:

Signal	Width	Answers
A register select	3	which register drives the A bus ($2^3 = 8$ registers)
B register select	3	which register drives the B bus
Destination select	3	which register the result is written back into
ALU control	6	what the ALU does — F_1, F_0, ENA, INVA, ENB, INC
Memory read/write	1	whether this is a read or a write

Plus the address and data connections to memory itself. **The widths are not arbitrary**: three bits because there are eight registers, and six bits because the ALU has exactly six control inputs.

Control signals come from the control unit, which must generate them **in the right order** for a given instruction — and which instruction that is, is determined by the contents of the **Instruction Register**.

So a line of high-level code like `r5 = r0 + r2` becomes: A select = 0, B select = 2, destination select = 5, ALU control = the $A + B$ row of the combinations table, memory read/write idle.

Data influencing control

The IR is not enough. **The control unit can't operate without knowing something about the data** — for example, if a for-loop variable is supposed to count from 10 down to 0, how does the control unit know when it gets there?

The answer is the **status register**: determined by ALU operations, it stores the status of the **last** ALU operation and is available to the control unit. This closes the loop — the ALU's results feed back into the control unit and change what it does next, which is what makes conditional branching possible at all. It is the return path drawn in Figure 88, and it is what a source-level `if (x < 0)` compiles down to.

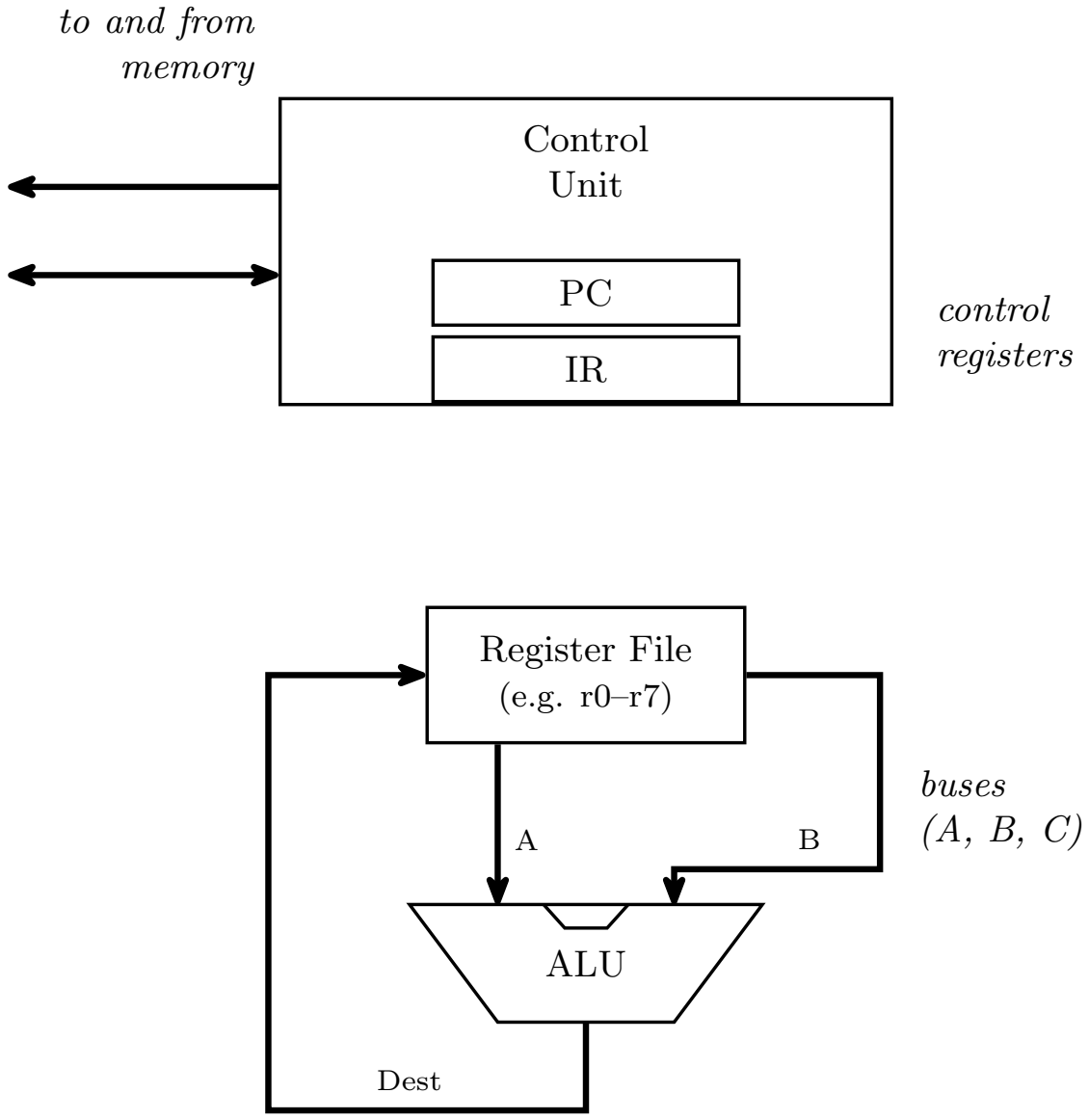


Figure 87: The datapath the cycle runs on. The control unit holds PC and IR and talks to memory; below it, the register file puts two values onto the A and B buses, the ALU combines them, and the result returns on the destination bus to be written back into a register.

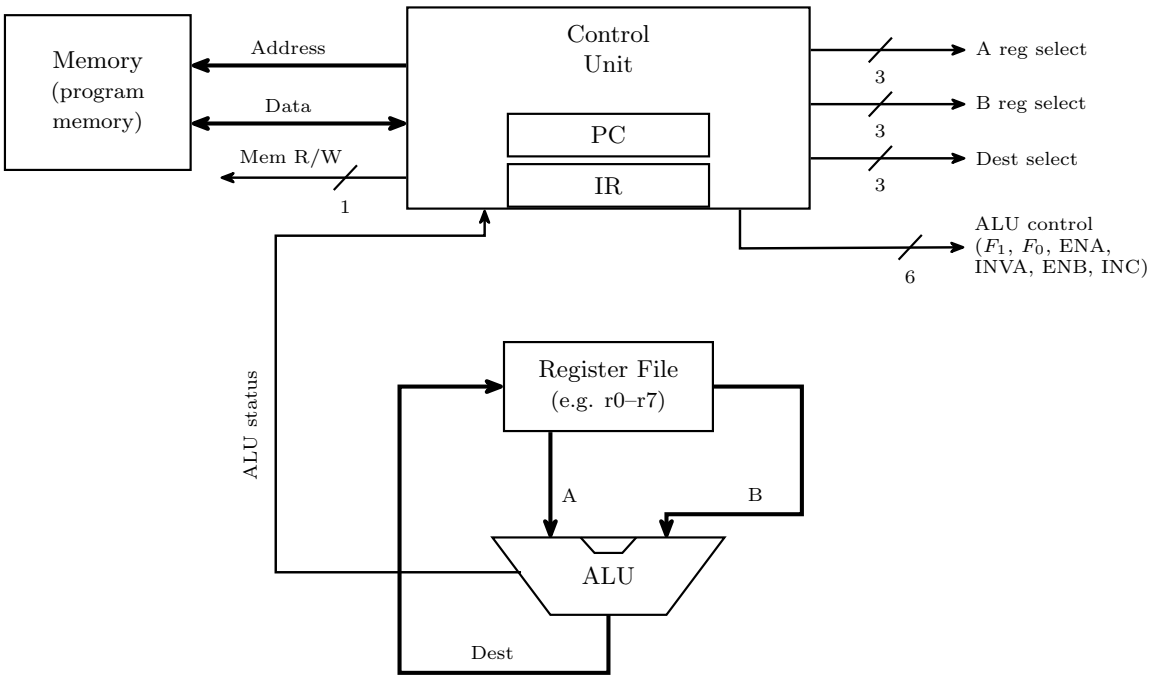


Figure 88: The same datapath with every control signal named. Added to the previous figure: the memory block, the *Address*, *Data* and *Mem R/W* labels, the five bundles with their widths, and the ALU status path back into the control unit. The bundles are drawn as labelled stubs: A/B/Dest select drive the register file, ALU control drives the ALU. Widths are marked with a slash and a number; the course also letters the width beside a plain arrow, and neither is fixed as the standard. Everything except the memory block sits inside the CPU.

ALU status flags

The ALU emits, alongside its result, a summary of *what happened during the last operation.* Typically stored in a special purpose register called the **STATUS register**:

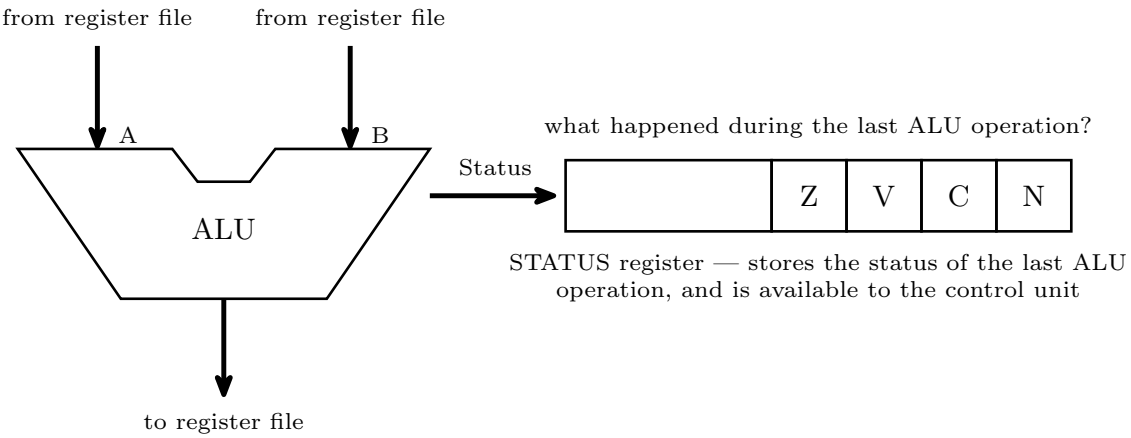


Figure 89: The ALU's second output. The result goes back to the register file; alongside it the ALU reports what happened, and those flags land in the STATUS register where the control unit can read them. The ALU is drawn as a notched trapezoid here; the block diagram in the ALU note draws the same block as a plain rectangle, and the course uses both shapes. The register is drawn as four named cells at the right-hand end of a wider register. That is shorthand, not a bit-position claim — the AVR's own SREG orders and numbers its bits differently, see the AVR architecture note.

Flag	Name	Asks
Z	zero	was the last result 0?
V	overflow	did the last addition/subtraction overflow, assuming it was a two's complement operation?
C	carry	was a carry out generated?
N	negative	was the result negative if considered as two's complement — i.e. what is the sign bit?

V and N both presuppose a two's complement interpretation while **C does not** — the hardware computes all four regardless, and it is the program's business to read the ones

that mean something for the operation it just did. See binary-number-representations[264], and 2026-08-04-binary-arithmetic[265] for the overflow rules V implements.

Worked example

What values are in the status register flags after the operation $8_{10} - 9_{10}$ on a 5-bit ALU?

First, -9 in 5-bit two's complement: $9 = 01001$, invert to 10110, add one to get 10111. Then add, tracking the carry into and out of the sign column:

$$\begin{array}{rr} \text{carry into MSB} \rightarrow & 0 \\ & 01000 \\ + & 10111 \\ \hline \text{carry out} = 0 \rightarrow & 11111 \end{array}$$

The result is 11111, with no carry out of the top bit. So:

Flag	Value	Why
Z	0	the result 11111 is not zero
C	0	no carry out of the most significant bit
V	0	the carry into the sign column and the carry out of it agree, both 0 — and indeed $8 + (-9) = -1$ is comfortably in range for 5-bit two's complement
N	1	the sign bit is 1 — the result is negative

And indeed 11111 read as two's complement is -1, which is $8 - 9$.

See also

- cpu-organisation[266] · arithmetic-logic-unit[267]

[264] courses/csse2010/binary-number-representations.pdf
[265] courses/csse2010/2026-08-04-binary-arithmetic.pdf
[266] courses/csse2010/cpu-organisation.pdf
[267] courses/csse2010/arithmetic-logic-unit.pdf

- binary-number-representations[268] — two's complement, overflow and the sign bit
- 2026-08-04-binary-arithmetic[269] — the $C_{in} \oplus C_{out}$ overflow test the V flag implements
- avr-architecture[270] — the AVR's own status register, bit by bit
- state-machines[271] — the control unit is itself a state machine, sequencing signals for each instruction
- 2026-08-24-alu-and-control-unit[272] — Lecture 9

Logic Gates

Every logic gate has one or more inputs and exactly one output, and can be described four equivalent ways: **logic symbol, truth table, Boolean expression, timing diagram** (see timing-diagrams[273] for the last of these, and for real-gate propagation delay / rise / fall times).

Gates sit at the bottom of the abstraction stack — the **digital logic level**, level 0 (see 2026-07-27-course-intro-bits-bytes-binary[274]). A digital circuit only ever carries **two** logic levels: logic '0' is usually a small voltage (around 0 V) and logic '1' a larger one (e.g. 0.8 V to 5 V, depending on the **logic family** — the type and size of the transistors the gate is built from). Because inputs and outputs have only these two states, they can equivalently be called *true* and *false*.

The 7 basic gate types

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

[268] courses/csse2010/binary-number-representations.pdf
[269] courses/csse2010/2026-08-04-binary-arithmetic.pdf
[270] courses/csse2010/avr-architecture.pdf
[271] courses/csse2010/state-machines.pdf
[272] courses/csse2010/2026-08-24-alu-and-control-unit.pdf
[273] courses/csse2010/timing-diagrams.pdf
[274] courses/csse2010/2026-07-27-course-intro-bits-bytes-binary.pdf

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

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

Deriving the inverting gates from the base gates

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

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

More than two inputs

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

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

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

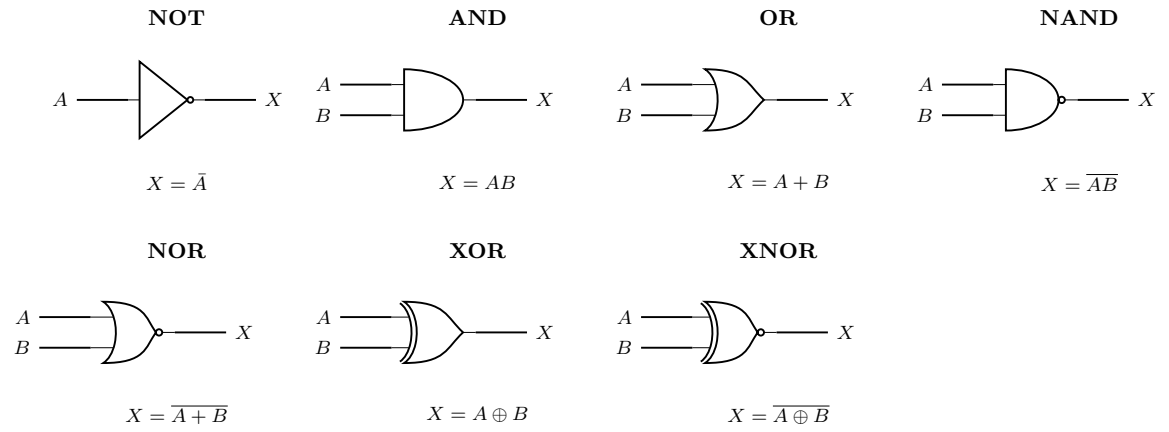


Figure 90: The seven basic gates, with the distinguishing symbol and Boolean expression for each. NOT, AND and OR are the three base shapes; NAND, NOR and XNOR are those same bodies with an inversion bubble on the output, and XOR/XNOR add a second arc across the back of the OR body.

Note that the kit of parts used for practicals has **no** 3-or-more-input gate chips (see device-pinouts[275]) — a 3-input function has to be built from 2-input gates.

Universal (complete) gates

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

- **NOT**: a NAND (or NOR) gate with both inputs tied together — $\overline{A \cdot A} = \bar{A}$, and $\overline{A + A} = \bar{A}$.
- **AND from NAND**: a NAND followed by a second NAND wired as an inverter. **AND from NOR**: invert both inputs first, then NOR them ($\overline{\bar{A} + \bar{B}} = AB$, by De Morgan — see boolean-algebra[276]).
- **OR from NOR**: a NOR followed by a NOR-as-inverter. **OR from NAND**: invert both inputs first, then NAND them ($\overline{\bar{A} \cdot \bar{B}} = A + B$).

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

[275] courses/csse2010/device-pinouts.pdf
[276] courses/csse2010/boolean-algebra.pdf

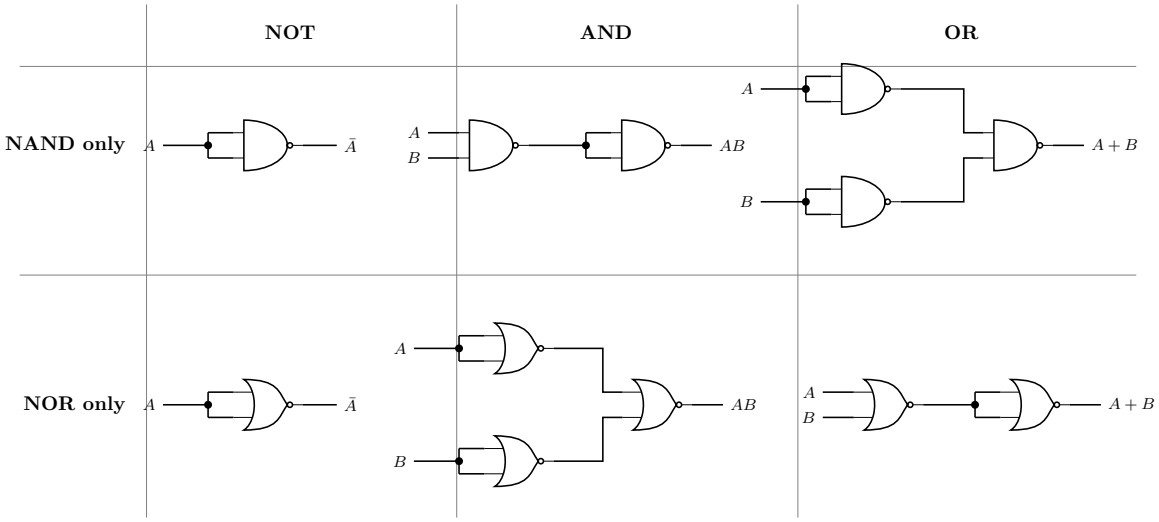


Figure 91: Why NAND and NOR are complete gates: NOT, AND and OR built from each one alone. Tying both inputs of a NAND or NOR together turns it into an inverter, and everything else follows — AND is NAND then invert, OR is invert-both-inputs then NAND (and the NOR row is the mirror image of that).

Gates on ICs

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

- **Vcc** = power (e.g. 5 V), **GND** = ground (0 V). One Vcc/GND pair supplies all four gates.
- Physically: pin spacing is 0.1” along the package and 0.3” across it; the chip is about 15 mm long.
- The drawing above is the chip’s **internal schematic**, not a picture of the part — see circuit-schematics[277] for how a chip like this is drawn (and labelled `U1:A` … `U1:D`) on a schematic you can actually build from, and device-pinouts[278] for the pinouts of every chip in the course kit.

Timing Diagrams

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

[277] courses/csse2010/circuit-schematics.pdf
[278] courses/csse2010/device-pinouts.pdf

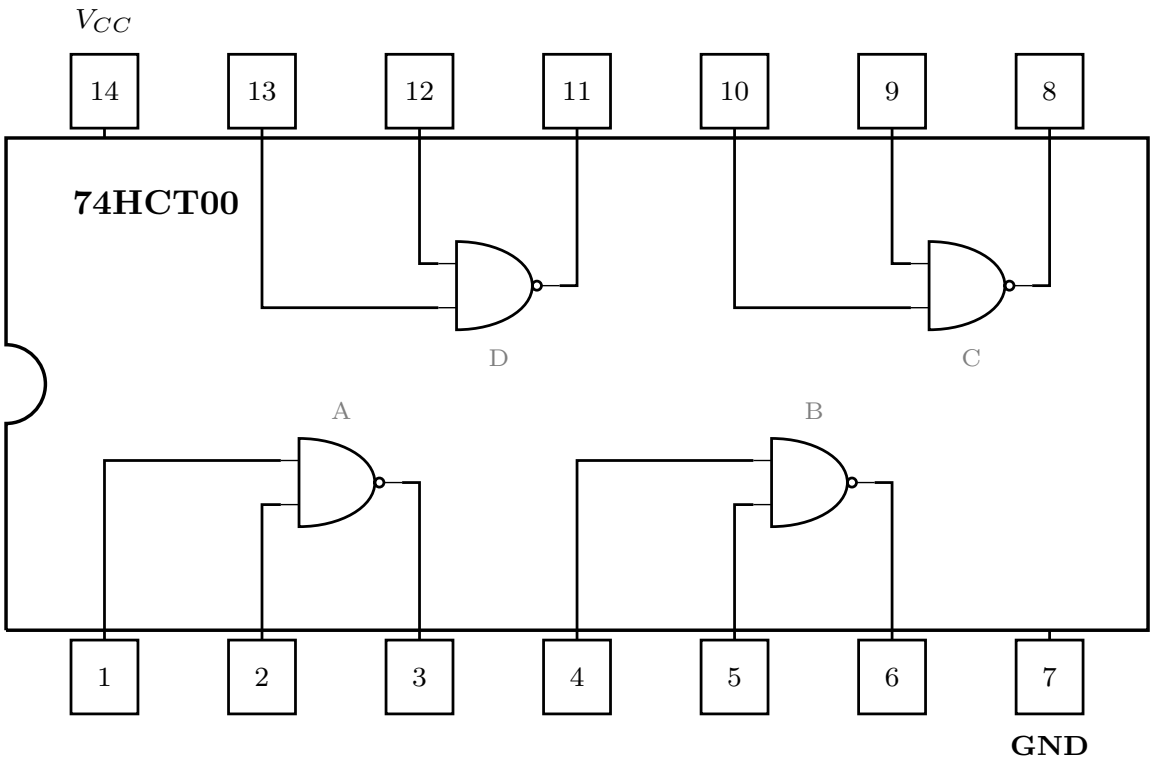


Figure 92: Inside the 74HCT00: four independent 2-input NAND gates in a 14-pin package, with the pin each gate terminal is wired to. Pin 14 is Vcc (power, e.g. 5 V) and pin 7 is GND (0 V) — shared by all four gates, so nothing works if either is left unconnected. Gate letters A-D are the labels a schematic uses as U1:A … U1:D.

gates[279]).

Timing diagram as a logic representation

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

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

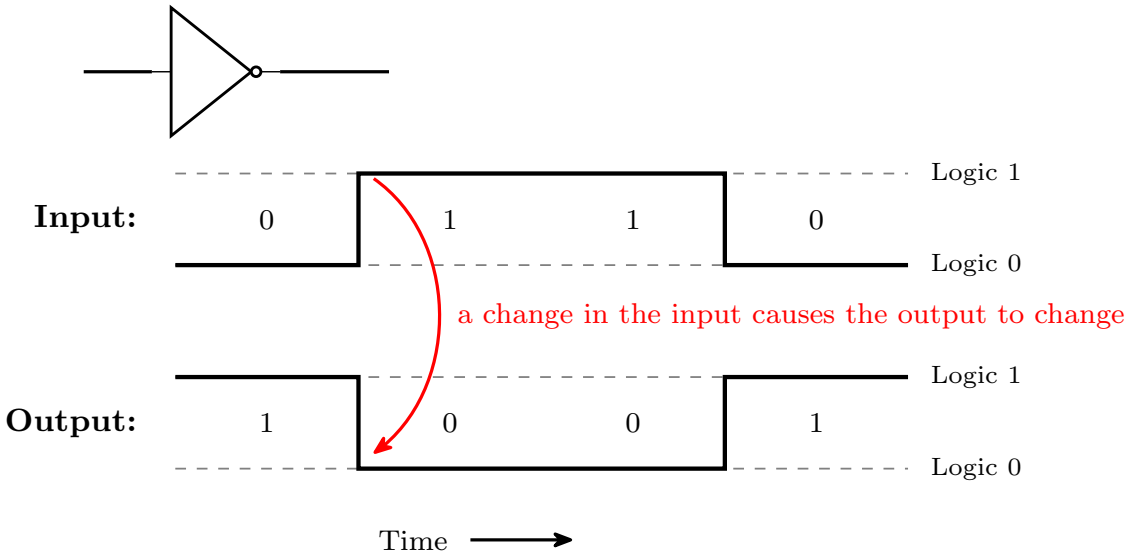


Figure 93: An inverter drawn as a timing diagram. Reading the four intervals off the input gives 0, 1, 1, 0 and off the output gives 1, 0, 0, 1 — which is the inverter's truth table, just laid out along a time axis instead of down a page. The two dashed lines per track are the logic 0 and logic 1 reference levels.

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

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

Convention for drawing the input waveforms

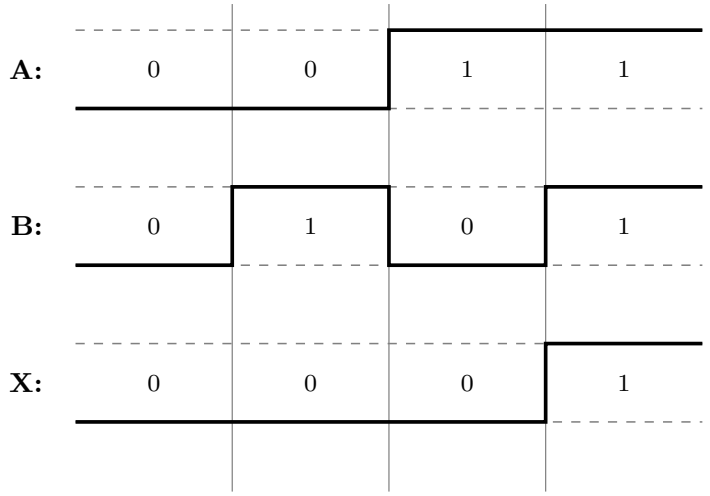
The input waveforms must be drawn so that **all possible input combinations are covered** — that is what makes the diagram equivalent to the full truth table.

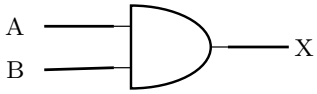
The standard construction for a 2-input gate divides the time axis into four equal intervals and drives the inputs like a binary count:

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

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

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





A	B	X
0	0	0
0	1	0
1	0	0
1	1	1

Figure 94: The convention on a 2-input example, with the AND output filled in. The vertical rules divide the four intervals; the digits under each track are the truth-table row for that interval. Because the inputs count 00, 01, 10, 11, the output track is literally the output column of the truth table, drawn left to right.

Going the other way — being *given* the output track and asked which gate it is — is the standard exam form of this. Read the output value in each interval, write it beside the

corresponding AB row, and compare the resulting column with the gates you know. A worked instance is in 2026-08-06-combinational-logic[280].

Real gates aren't perfect: the reality of timing

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

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

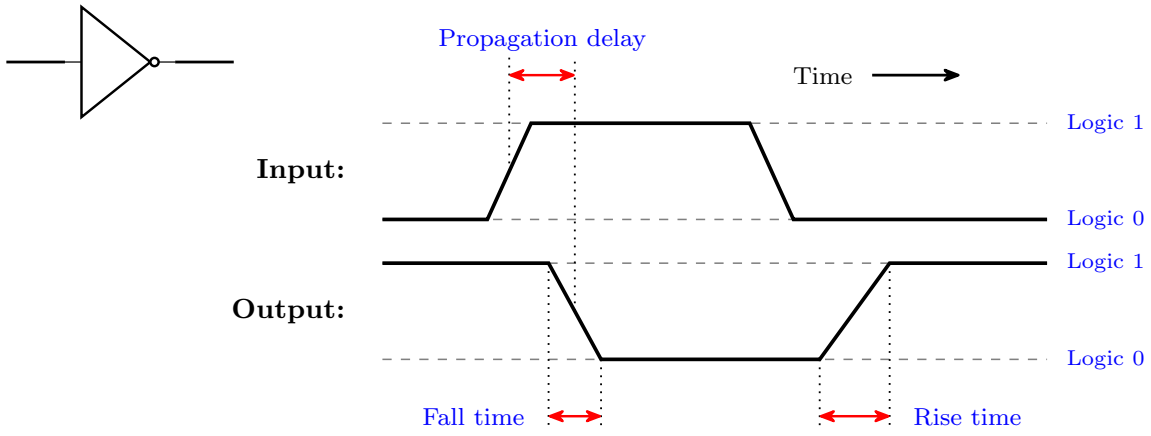


Figure 95: The same inverter drawn honestly. The input's edges slope instead of jumping; the output does not begin to move until the propagation delay has elapsed; and once moving, its downward edge takes the fall time and its later upward edge takes the rise time. Rise and fall times are properties of one edge of one signal — the propagation delay is the gap between two signals.

Measured on an inverter: the input ramps up, and only after the propagation delay does the output begin to move; the output's downward edge then takes the fall time to complete, and its later upward edge takes the rise time.

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

The three quantities are measured differently, and it is worth keeping them apart. **Rise time and fall time** are properties of a single edge of a single waveform — how long that edge takes to get from one level to the other. **Propagation delay** is the gap *between* two waveforms — how long the output takes to react at all. All three are marked on the same pair of inverter waveforms above.