Binary Arithmetic

Table of contents

Worked: -32 in 8-bit signed magnitude . . . 1
Worked: -32 in 8-bit two's complement . . . 2
Universal (complete) gates . . . 2
Worked: a NOR-only implementation . . . 3
Binary addition . . . 3
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 . . . 9
Admin and reminders . . . 9

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**.

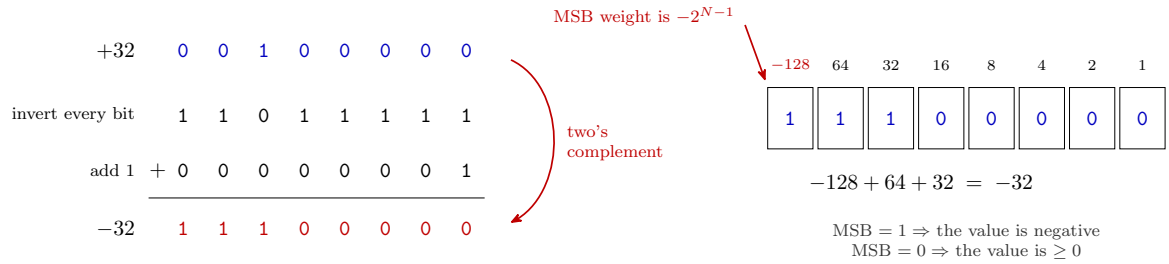

Figure 1: Negating +32 in 8 bits. Left: invert all eight bits of 00100000 and add one. Right: the answer checked against the place values — in two's complement the MSB is the only column with a *negative* weight, -2^{N-1}, which is what makes 11100000 come to -32 rather than 224.

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.

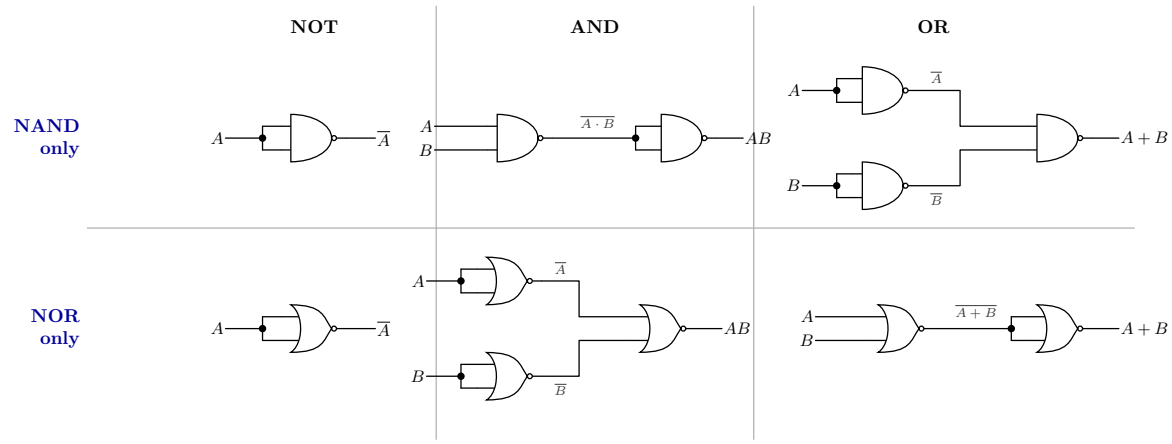

Figure 2: NOT, AND and OR built two ways: from NAND gates only (top row) and from NOR gates only (bottom row). The two building tricks are the ones to remember — a NAND or NOR with its inputs tied together is an inverter, and De Morgan turns an inverted-input NAND into an OR (and an inverted-input NOR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

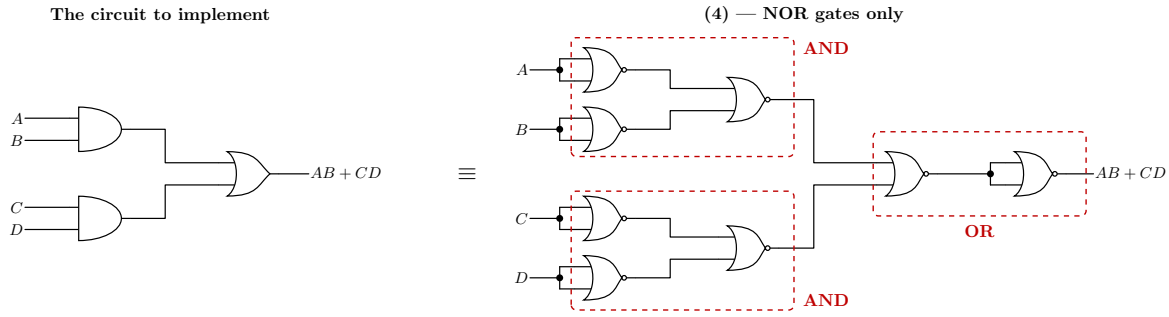

Figure 3: The circuit to implement (left) and its NOR-only version (right). Read it as three blocks: each AND becomes 'invert both inputs, then NOR', and the OR becomes 'NOR, then invert'. The tempting wrong answer is the same circuit with that last inverting NOR left off, which computes the complement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

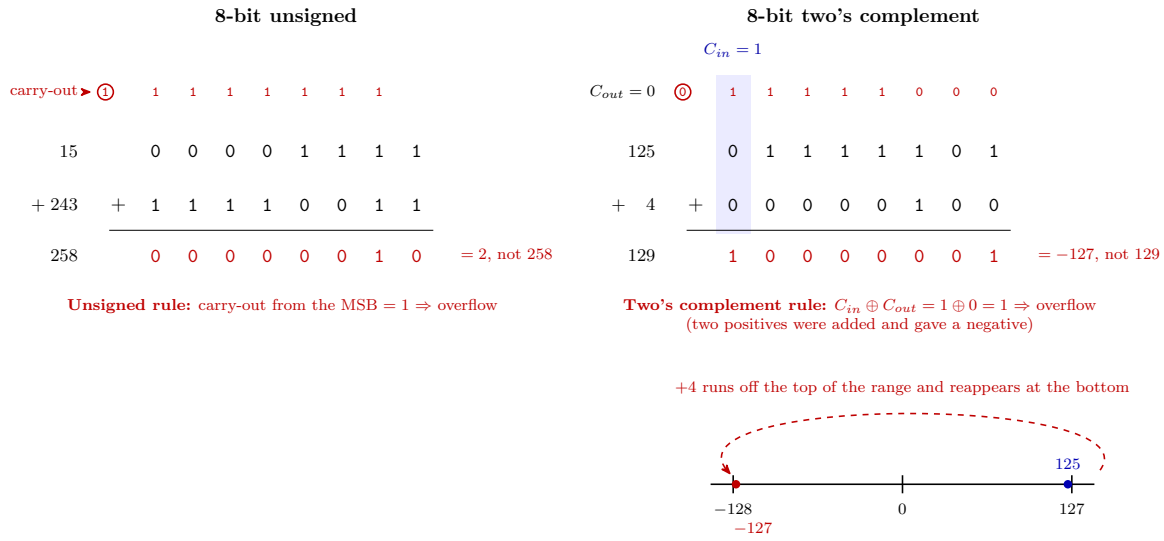

Figure 4: The two overflow cases worked with their carry rows exposed, because the carries are what the detection rules are stated in terms of. Left: unsigned, where a carry-out of the MSB alone means overflow. Right: two's complement, where the carry *into* the MSB and the carry *out of* it disagree — 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**.

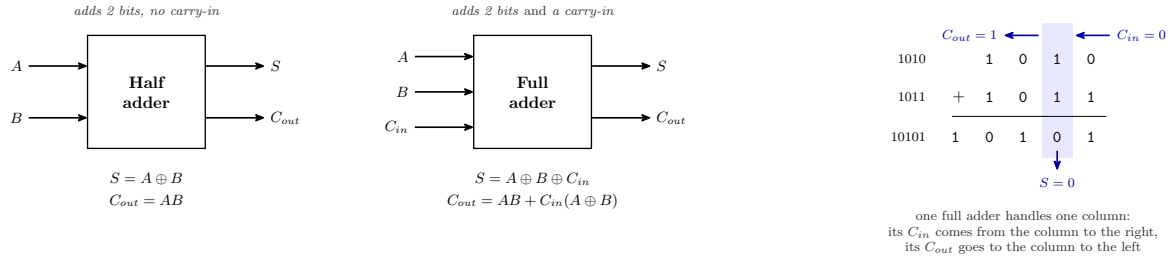

Figure 5: Half-adder versus full adder as blocks, and (right) why the third input exists: in a multi-bit addition each column is one full adder, taking its carry-in from the column on its right and passing its carry-out to the column on its left. Only the least significant column can get away with a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

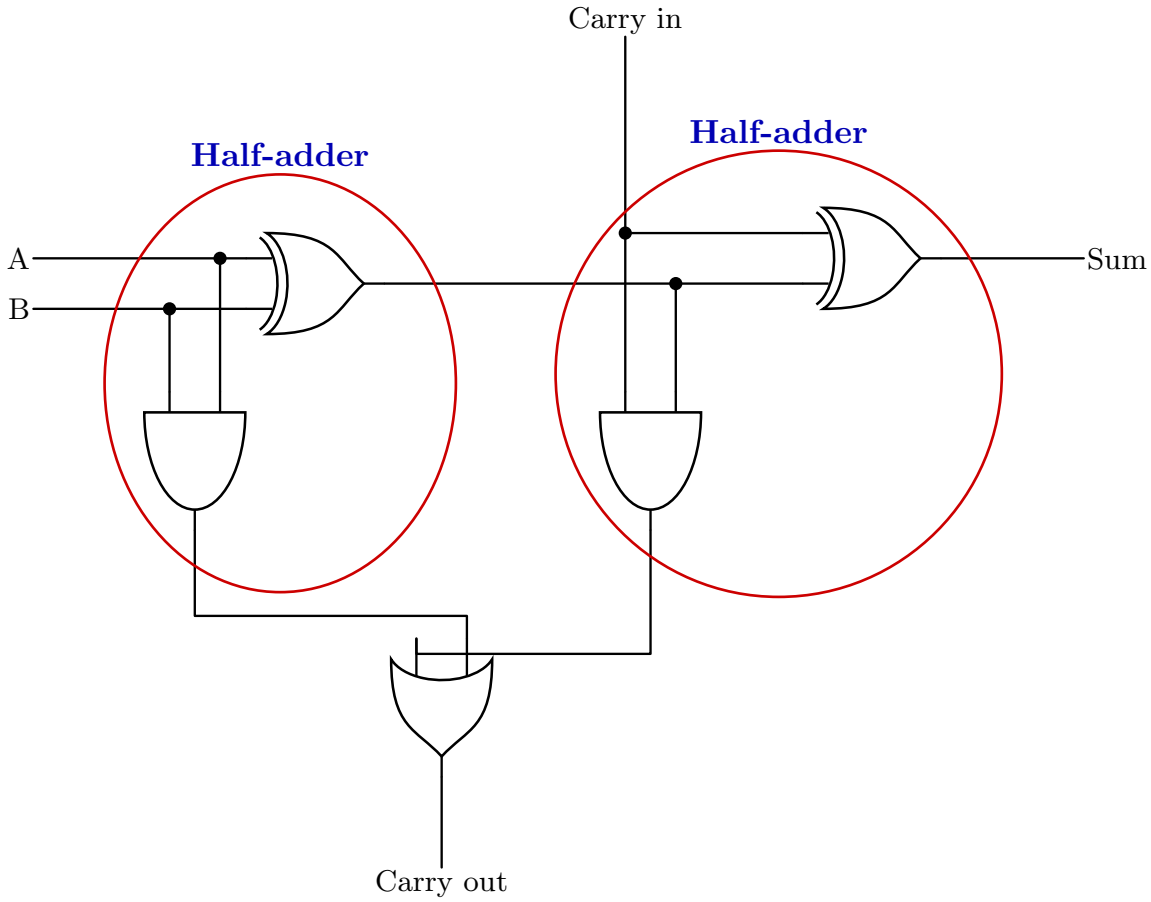

Figure 6: The full adder at gate level, with the two half-adders outlined. The left half-adder adds A and B; the right one adds that sum to the carry in; the OR at the bottom collects the two carries. At the crossing near the top right, the carry-in line passes over the first half-adder's sum line without 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.

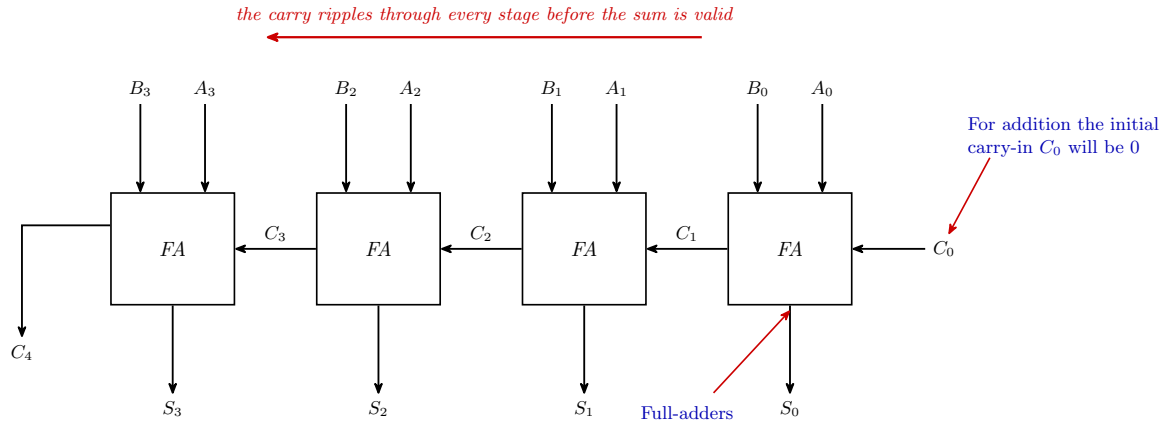

Figure 7: A 4-bit ripple-carry adder: one full adder per bit position, with the carry chained from the least significant stage (right) to the most significant (left). Since stage i cannot produce its final sum until stage $i-1$ has settled, the carry has to propagate through all four stages before any 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.

[7] courses/csse2010/week2-lab-binary-arithmetic.pdf