Number Representations

Table of contents

Worked: two unsigned conversions . . . 1
The bare metal . . . 2
Radices . . . 2
Negative numbers . . . 4
Why "complement" . . . 6
Limitations . . . 6
Format matters . . . 7
Exercise — the five-format table . . . 7
Esoteric formats . . . 8
Admin . . . 8
See also . . . 8

Learning Lab 1. There is no hands-on prac this week; the prac activities start in week 2. See binary-number-representations[1] for the reference version of everything below.

Worked: two unsigned conversions

Two plain unsigned conversions of the kind in 2026-07-27-course-intro-bits-bytes-binary[2]. The format plays no part yet.

Question	Answer	Working
Decimal 37 in 8-bit binary (unsigned)	`00100101`	$37 = 32 + 4 + 1 = 2^5 + 2^2 + 2^0$
Unsigned binary `01000110` in decimal	70	$64 + 4 + 2 = 70$

[1] courses/csse2010/binary-number-representations.pdf
[2] courses/csse2010/2026-07-27-course-intro-bits-bytes-binary.pdf

Wrong answers to recognise: `00110101` is 53, which is what you get by mis-decomposing 37 as $32 + 16 + 4 + 1$. 67, 71 and 72 are all one or two off 70, which is what a slipped bit position costs.

The bare metal

A computer stores everything with high and low voltages, or some other two-state signal: magnetic fields, switches, flash memory. That goes for all of these alike:

- unsigned numbers
- signed numbers
- decimal numbers, characters, images, sounds

You cannot know what data is just by looking at it.

This comes back at the end as "format matters".

Radices

A radix-k number system has k symbols for the digits 0 to $k-1$, and digit values (from the right) $k^0, k^1, k^2, ...$

- **Octal** (radix-8): symbols 0–7; one octal digit = **3 bits**.
- **Hexadecimal** (radix-16): symbols 0–9 A–F; one hex digit = **4 bits** (useful).

Worked: the maximum value in n radix-k digits

Answer: $k^n - 1$.

Two ways to see it:

1. The smallest number you *can't* represent is a one followed by n zeroes, whose value is k^n. The largest you can is one less.
2. The biggest number is n copies of the largest digit $k-1$, i.e. $\sum_{i=0}^{n-1} k^i(k-1) = k^n - 1$ by the geometric series.

Conversion facts

- A hex digit represents 16 values; so do 4 bits ($2^4 = 16$).
- An octal digit represents 8 values; so do 3 bits.
- A base-10 digit represents 10 values; so do 3.321928… bits *[citation needed]*.
- Hex and octal convert to binary **digit-by-digit**; base-10 does not.
- Three hex digits represent 4096 values; so do four octal digits.
- $123 = 0123$, but $456 \neq 4560$ — **pad zeroes on the left**.

The conversion map

Binary is the hub of the conversions. Only two conversions are digit-by-digit, and both of them have binary at one end.

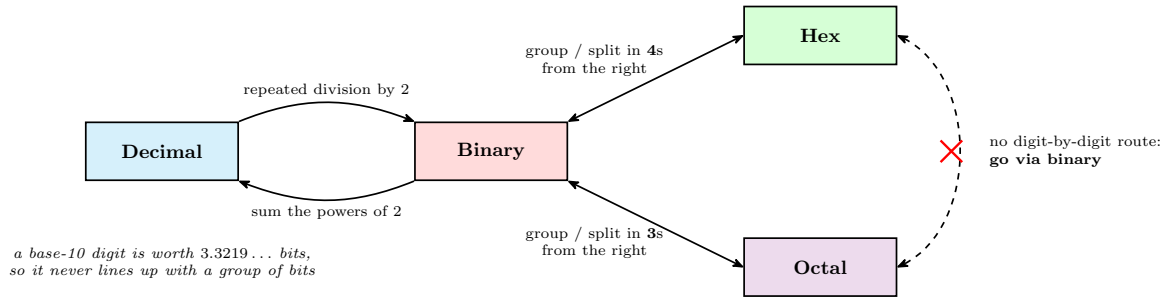


Figure 1: The conversions between decimal, binary, hexadecimal and octal, drawn as one map. Binary sits in the middle because hexadecimal and octal are both powers of two and so convert to it by regrouping bits, four at a time and three at a time respectively. Decimal is not a power of two, so its two arrows are arithmetic (repeated division, summation of powers) rather than regrouping. There is no direct hex-octal arrow: converting 0xC55E7201 to octal goes the long way, through 32 bits of binary. The box colours carry no meaning.

Worked conversions

Group binary into 4s for hex, 3s for octal, padding on the left:

From	To	Working	Result
`001001101001`	hex	`0010 0110 1001`	`0x269`
`01001101001`	octal	`001 001 101 001`	1151_8
`0xA1E`	binary	`1010 0001 1110`	`101000011110`
703_8	binary	`111 000 011`	`111000011`

From	To	Working	Result

`0xC55E7201`	octal	regroup 32 bits into 3s (pad one zero):
		`011 000 101 010`
		`111 100 111 001`
		`000 000 001`
			30527471001_8

Hex → octal has no shortcut: go via binary.

Negative numbers

Computers don't store + and −; four formats have been used. **Two's complement is what is practically used.**

All four — and unsigned — use **the same representation for positive numbers**.

Signed magnitude

Leftmost bit is the sign (0 positive, 1 negative) and **does not contribute to the magnitude**. Negate by inverting the sign bit alone.

Ones' complement

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

Two's complement

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

Excess-2^{N-1}

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

It is the **same as two's complement with the sign bit reversed** — a 0 in the MSB means bit value −128, a 1 means 0. Practical use: comparing two signed numbers becomes simple in hardware.

All of it in one picture

Write each format as a row of bit weights and stack the rows. They differ in one column only.

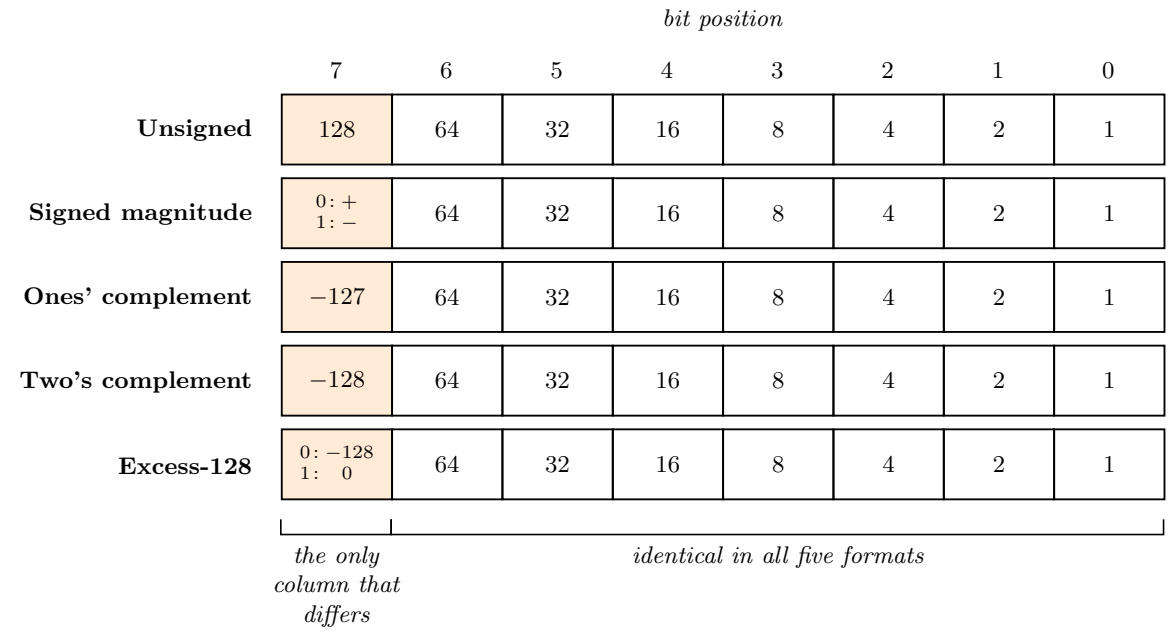


Figure 2: The four signed formats and unsigned, as stacked strips of bit weights. Every format gives bits 6 down to 0 exactly the same weights; the only thing any of them changes is what bit 7 is worth. Signed magnitude and excess-128 do not give bit 7 a weight at all — they give it a meaning, which is why those two boxes say what a 0 and a 1 mean rather than give a number. In signed magnitude the sign bit is not a magnitude position: only bits 6 to 0 carry magnitude.

The shaded column is the only one that differs. The reason two's complement wins is not that it is cleverer at representing negatives; it is that -128 is the only choice of bit-7 weight that leaves the whole row a plain sum of place values, with no special rule attached to the top bit. That is exactly what makes its addition ordinary binary addition, with no case analysis and no end-around carry. Limitations below says the same in words.

The ±47 example, 8-bit

$+47 = \texttt{00101111}$ in all four formats.

Format	−47
Signed magnitude	`10101111`
Ones' complement	`11010000`
Two's complement	`11010001`
Excess-128	`01010001`

And the 4-bit −3 examples: ones' complement `1100`, two's complement `1101`.

Ranges

Format	Range
8-bit signed magnitude	−127 to 127
n-bit signed magnitude	$-(2^{n-1}-1)$ to $2^{n-1}-1$
8-bit ones' complement	−127 to 127
8-bit two's complement	−128 to 127
n-bit two's complement	-2^{n-1} to $2^{n-1}-1$
8-bit excess-128	−128 to 127

Why "complement"

In the loose mathematical sense, a thing and its complement together make a whole. An angle and its complement make a right angle; a graph and its complement a complete graph; a set and its complement a universal set.

- In **ones' complement**, a number plus its negative gives a string of ones (`11111111` in 8 bits).
- In **two's complement**, a number plus its negative gives a power of two, with overflow (`1 00000000`). It would more correctly be called *power-of-two's complement*.
- Base 10 has the equivalent nines' and ten's complements, used on some early mechanical adding machines to do subtraction.

Limitations

Two representations of zero — undesirable, because it 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^{m-1}.

Arithmetic: signed-magnitude addition requires taking cases; ones' complement addition requires an extra addition step; sign extension and comparisons are non-trivial in both.

Two's complement has one zero, and its addition is just binary addition — which is why it wins.

Format matters

"What is 10010010 converted to decimal?" cannot be answered as asked. It depends on the format:

Read as	Value
8-bit unsigned	146
Two's complement	−110
Ones' complement	−109
Signed magnitude	−18
Excess-128	18

In the same way, `01010101` could be 85 (8-bit unsigned), −43 (excess-128), `'U'` (ASCII), or 10.625 (8-bit fixed point, 5 bits before the point and 3 after).

> **Binary data is meaningless unless you know the format.** The format is not encoded in the data — it has to be known separately, by a human, a program, or the hardware.

Exercise — the five-format table

Write the decimal interpretation of every 4-bit pattern under each format:

Pattern	Unsigned	Sign-mag	Ones' comp	Two's comp	Excess-8
`0000`	0	0	0	0	−8
`0001`	1	1	1	1	−7
`0010`	2	2	2	2	−6
`0011`	3	3	3	3	−5
`0100`	4	4	4	4	−4
`0101`	5	5	5	5	−3
`0110`	6	6	6	6	−2
`0111`	7	7	7	7	−1
`1000`	8	−0	−7	−8	0

Pattern	Unsigned	Sign-mag	Ones' comp	Two's comp	Excess-8
`1001`	9	−1	−6	−7	1
`1010`	10	−2	−5	−6	2
`1011`	11	−3	−4	−5	3
`1100`	12	−4	−3	−4	4
`1101`	13	−5	−2	−3	5
`1110`	14	−6	−1	−2	6
`1111`	15	−7	−0	−1	7

The two −0 entries are the wasted patterns. Excess-8 is the two's complement column with the sign bit flipped.

Esoteric formats

Not examinable: binary coded decimal with sign pattern; negabinary (base −2); zig-zag encoding; LEB128; quarter-imaginary base (base $2i$) and the Davis–Knuth "twindragon" base (base $i - 1$) for Gaussian integers; and "integers don't exist" — JavaScript.

Admin

Lab kits are used *during* the lab from week 2 and loaned out later (see week3-lab-logisim-and-flip-flops[3]). If you're on your own laptop you'll need **two USB-A connections from week 11**, and one from week 8 — get a dongle early if you need one. Prac activities start in week 2. Catch up on the lecture content *before* attending a lab.

There is a **summary recording on Blackboard covering the signed binary formats**, to be watched before week 2.

See also

- binary-number-representations[4] — the reference version of every format here
- week-one-exercises[5] — the week 1 problem set, which reuses this lab's `1001101001` conversion and asks for the same four formats at other widths
- 2026-07-27-course-intro-bits-bytes-binary[6] — Lecture 1, the unsigned conversions
- csse2010-sample-midsem-5[7] · csse2010-sample-midsem-6[8] — questions 1 to 4 of each

[3] courses/csse2010/week3-lab-logisim-and-flip-flops.pdf
[4] courses/csse2010/binary-number-representations.pdf
[5] courses/csse2010/week-one-exercises.pdf
[6] courses/csse2010/2026-07-27-course-intro-bits-bytes-binary.pdf
[7] courses/csse2010/csse2010-sample-midsem-5.pdf
[8] courses/csse2010/csse2010-sample-midsem-6.pdf

paper are exactly this material