Binary Number Representations

Table of contents

Bits, bytes, and words 1
Representing whole (unsigned) numbers 1
Other radices 3
Representing signed numbers 6
Fractions: fixed and floating point 9

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.

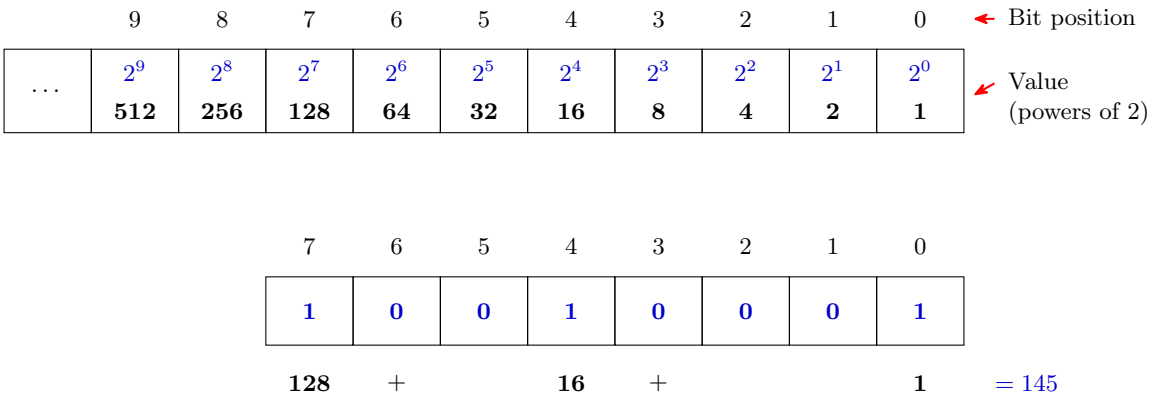

Figure 1: Place values for an unsigned binary number, and a worked conversion. Position i is worth 2^i, so reading a binary number is just adding the place values of the positions holding a 1: $10010001_2 = 128 + 16 + 1 = 145$.

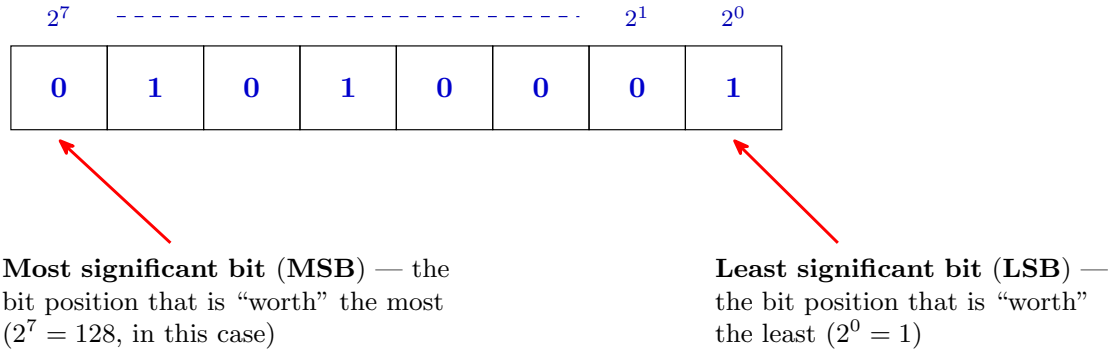

Figure 2: The two ends of a word get names, because almost every later convention (shifting, sign bits, bit numbering in AVR registers) is stated in terms of them. For an n-bit unsigned word the MSB is worth 2^{n-1} — here $2^7 = 128$ — and the LSB is always worth $2^0 = 1$.

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.

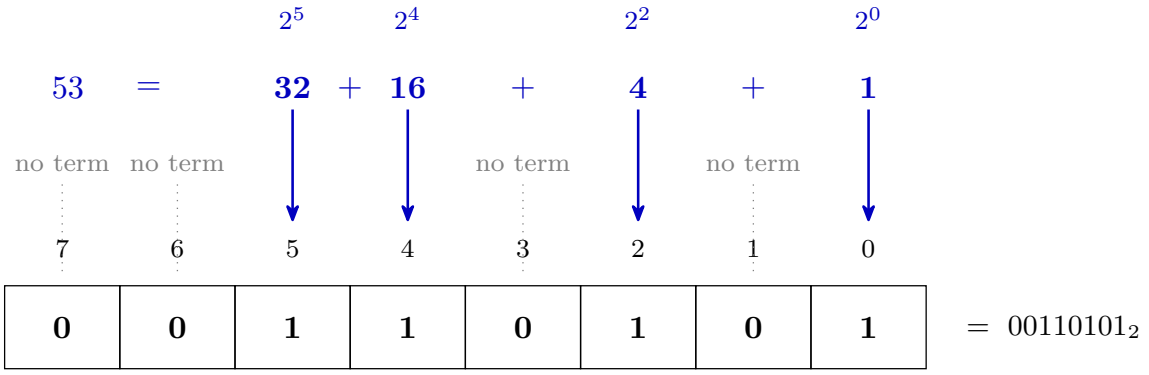

Figure 3: Method 1 on 53. Each power of two in the decomposition names one bit position — 2^5 is position 5, 2^4 is position 4, and so on — and every position with no term in the sum gets a 0. Reading the row off left to right gives 00110101_2.

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

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

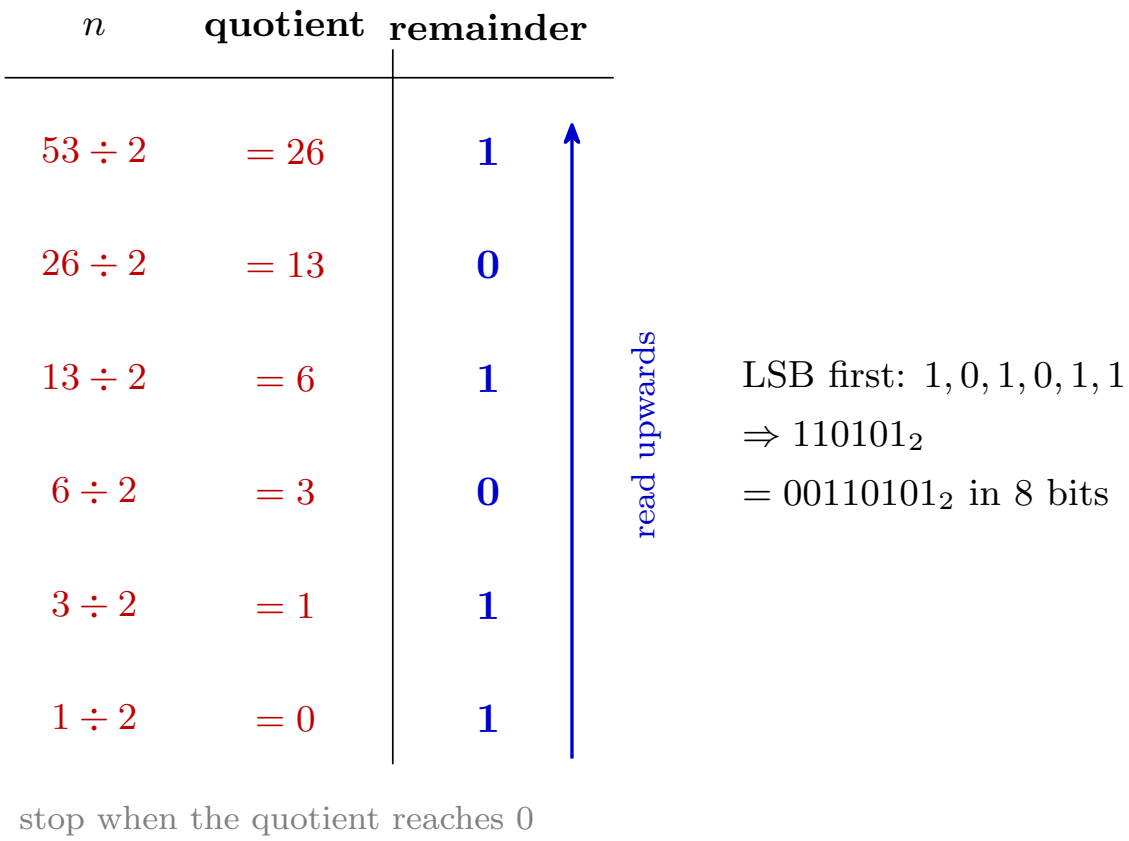

Figure 4: Method 2 on 53. The remainders come out LSB-first, which is why the answer has to be read back up the column: 110101_2, or 00110101_2 padded to a byte. The same procedure with a divisor of 16 or 8 converts to hexadecimal or octal instead.

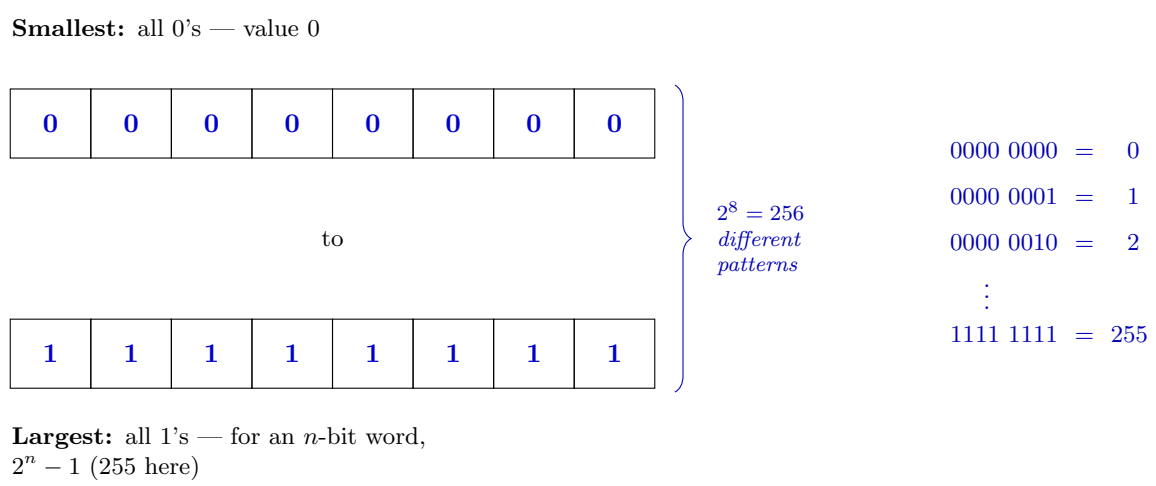

Figure 5: Why the largest unsigned value is $2^n - 1$ and not 2^n. Counting up from all-zeros steps through every pattern in turn — each extra bit doubles how many there are, so 8 bits give $2^8 = 256$ patterns — and because the count starts at 0, the last one is 255.

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

Converting binary ⇄ hex/octal by grouping

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

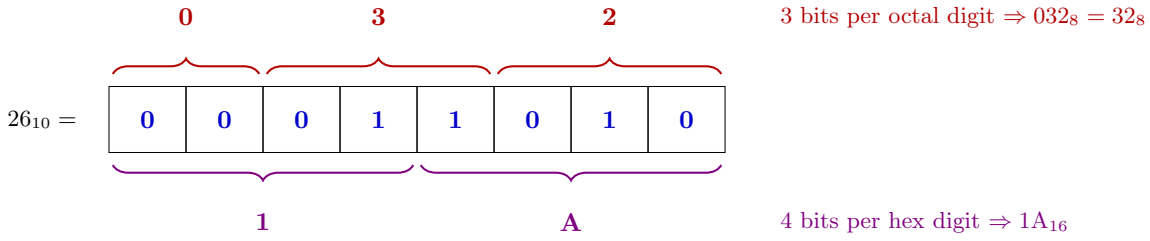

Figure 6: One bit pattern read two ways. Grouping 00011010_2 into nibbles from the right gives 1 and A, i.e. $1A_{16}$; grouping the same bits in threes gives 0, 3 and 2, i.e. 32_8. The grouping must start at the right — starting at the left gives the wrong answer whenever the width is not a multiple 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[1] for that worked exercise.

Radix notation conventions

Since a bare number like `101` or `747` is ambiguous, a radix indicator is needed. A subscript

Radix	Convention	Example	Where used
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.**

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

	bit 7	6	5	4	3	2	1	0
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[2]).

Ranges

Format	8-bit	n-bit
Unsigned	0 to 255	0 to $2^n - 1$
Signed magnitude	−127 to 127	$-(2^{n-1} - 1)$ to $2^{n-1} - 1$
Ones' complement	−127 to 127	$-(2^{n-1} - 1)$ to $2^{n-1} - 1$
Two's complement	−128 to 127	-2^{n-1} to $2^{n-1} - 1$
Excess-2^{n-1}	−128 to 127	-2^{n-1} to $2^{n-1} - 1$

Worked example — ±47 in 8 bits

+47 = `00101111` in every format.

Format	−47	How
Signed magnitude	`10101111`	flip the sign bit
Ones' complement	`11010000`	invert all bits
Two's complement	`11010001`	invert all bits, add 1
Excess-128	`01010001`	$-47 + 128 = 81$

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

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

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
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[3] for the full 4-bit table across all five formats, and 2026-08-04-binary-arithmetic[4] 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[5].
- **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[6].

[3] courses/csse2010/week1-lab-number-representations.pdf
[4] courses/csse2010/2026-08-04-binary-arithmetic.pdf
[5] courses/csse2010/fixed-point-numbers.pdf
[6] courses/csse2010/floating-point-numbers.pdf