How to represent real numbers

Up to now we have talked only about integers.

Programming languages (and real life) use real numbers.

Real numbers represented in the decimal scientific notation take the form of a fraction multiplied by 10 to some power.
(e.g., $-134.567 \times 10^3, 987.654 \times 10^{-23}$)

In general we let only one single digit to the left of the decimal point (e.g., $-0.134567 \times 10^6, 9.87654 \times 10^{-21}$)

A number is *normalized* if the leading digit is not zero. (e.g., -0.134567×10^6 is NOT normalized, 9.87654×10^{-21} is).

How are real numbers represented inside the computer?

Use a representation akin to scientific notation:

$$sign \times mantissa \times base^{exponent}$$

There have been many variations on the choice of representation of the mantissa, the choice of the base, and representation of the exponent.

Arithmetic supporting this representation of real numbers is called *floating point.*

Floating-point representation: IEEE Standard

Basic choices:

- A single precision number must fit into one word (4 bytes; 32 bits)
- A double precision number must fit into two words
- The base for the exponent is 2
- There should be (approximately) as many positive and negative exponents

Additional criteria

- The mantissa will be represented in sign and magnitude form
- Numbers will be *normalized*

MIPS Floating-point representation
Single precision

A number is represented as:

$$(-1)^s \times F \times 2^E$$

In single precision the representation is:

31	23	22 0
s	exponent	mantissa

s bit 31 sign bit (0 positive, 1 negative)
exponent 8 bits (positive and negative exponents)
mantissa 23 bits: a fraction with implied binary point at left of bit 22.

0 is represented as all zeros.

Note that having the sign bit as most significant bit makes it easier to test for 0, positive and negative.

MIPS Floating-point representation
Single precision

The exponent is *biased*. All positive exponents are larger (in binary) than negative exponents. Having the exponent before the mantissa makes it easier for sorting.

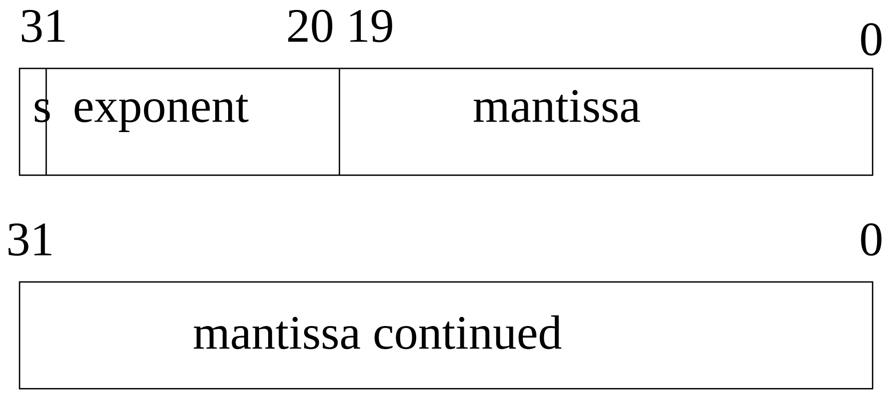
Thus exponent 0 will be: 01111111 (a 0 followed by 7 1's)
Thus exponent 1 will be: 10000000 ("biased 0" + 1)
Thus exponent -1 will be: 01111110 ("biased 0" - 1)
The largest positive exponent will be 11111111 (2^{2^7-1}(about 10^{38})
The smallest negative exponent will be 00000000 (-2^{2^7})

Since the numbers *must* be normalized and are represented in binary, there will always be an implicit 1 at the left of the binary point (no need to store it!)

Thus a floating point number has the value:

$$(-1)^s \times (1 + \ mantissa) \times 2^{exponent-127}$$

MIPS Floating-point representation
Double precision



s bit 31 sign bit (0 positive, 1 negative)
exponent 11bits (positive and negative exponents)
mantissa 52 bits: a fraction with implied binary point at left of bit 19

0 is represented as all zeros.

> Note that we can still have *overflow* (the exponent cannot handle super-big numbers) and *underflow* (the exponent cannot handle super-small numbers)

Floating-point arithmetic
Addition-subtraction

Not as simple as integer addition.

Four step process (can be pipelined!)

1. Compare exponents. Find the “difference”.
2. Shift the mantissa of the smallest exponent (difference) bits to the right (remember the implied 1)
3. Add the (shifted) mantissas (again, remember the implied 1). This is a sign and magnitude addition.
4. Normalize the result (also possibly round it, if need be)

Floating-point arithmetic
Multiplication

Conceptually easier.

1. Add exponents (careful; subtract one “bias”)
2. Multiply mantissas
3. Normalize and round
4. Get the correct sign