FLOATING-POINT ARITHMETIC

How to compute when all you have is a computer

FIRST, PHYSICISTS…

Floating-Point Numbers

- We want to represent numerical values on a computer
- Unfortunately, most numerical values cannot be represented on a computer for fundamental reasons
- Instead, only approximations can be represented on a computer
- **New problem: how accurate are machine computations?**

Floating-Point Numbers

Ask a physicist: what is the numerical value of the speed of light when measured in m/s?

Answer: $c = 2.99792458 \times 10^8$

That's scientific notation.

Somewhat terse, we can write down 2.99792458e8,

where 'e' reminds us that 8 is the exponent of the base 10.

Floating-Point Numbers

Let us try that again with

$$\sqrt{777777} = 881.916662729534779709025956 2387 \ldots$$

We use scientific notation again but only keep 12 digits after the point:

$$\sqrt{777777} \approx 8.819166627295 \times 10^2$$

Terser notation: 8.819166627295e2

Floating-Point Numbers

8.81916662729 5e2 is an example of a floating-point representation of a number in the decimal system.

The digits 881916662729 5 are called ***mantissa*** or **significant**.

The number 2 is called the **exponent**.

The number 10 is the **base**.

To get the actual number, we place the point after the first digit of the mantissa and multiply that with the base raised to the exponent. The number of digits in the mantissa is called **precision**.

Floating-Point Numbers

Some examples with precision 6:

- 3.14159e0 = 3.14159×10^{0}
- 7.15671e8 = 7.15671×10^{8}
- -8.14670e2 = -8.1467×10^{2}
- 2.29804e-4 = 2.29804×10^{-4}
- -2.00114e-15 = -2.00114×10^{-15}

Floating-Point Numbers

When computing with a lot of numbers (e.g., obtained from measurements), it often is a good idea to agree on a floating-point format: we *fix some precision* and stick with it through the calculations:

- All input numbers are rounded to that precision
- All intermediate results are rounded to that precision

In most applications, the exponents are confined to an exponent range $[e_{min}, e_{max}]$, and thus require at most some fix number of digits.

As a result, computations are more manageable, all numbers have the same number of digits, and all computations take roughly the same effort.

However, we need to make sure that the rounding errors in intermediate computations do not "pollute" the final result too much. If rounding errors are relatively large, the calculation results are useless.

Floating-Point Numbers

Let's work with precision 6, so 5 fractional digits.

5.11243e6 x 2.11001e3 = 1.07872784243e10

We only keep 5 fractional digits: **1.07872e10**

The relative error is:

$$\frac{|1.07872842423 \times 10^{10} - 1.07872 \times 10^{10}|}{|1.07872842423 \times 10^{10}|} = \frac{0.00000842423}{1.07872842423} \approx 7 \times 10^{-6}$$

The relative error measures how large the rounding error is in comparison to the true result. We hope that the rounding errors remain sufficiently close to zero.

Floating-Point Numbers

This often works fine we must be in for some surprises

1.00000e5 + 1.00000e-6 = 1.00000000001e5

Result after rounding: 1.00000e5

That's fine, the relative error is low:

$$\frac{|10000.000001 - 10000|}{|10000.000001|} \approx 10^{-11}$$

Floating-Point Numbers

Generally, we cannot count on the associative or distributive laws when computing in floating-point arithmetic in some fixed precision:

- $(a + b) + c \neq a + (b + c)$
- $(a \times b) \times c \neq a \times (b \times c)$
- $a \times (b + c) \neq a \times b + a \times c$

Typically, these laws remain only true up to some (hopefully small) relative error.

NOW, COMPUTERS…

Floating-Point Numbers

The examples up to now have used the decimal system.

Non-human computers use the binary system. The only possible values of each digit are 0 and 1.

Many technical details in practical implementations (representations of infinity, representations of 0, precise rounding policy, flags for invalid computations, ….)

Here only a simplified outline.

Floating-Point Numbers

A brief reminder of numbers in the binary system:

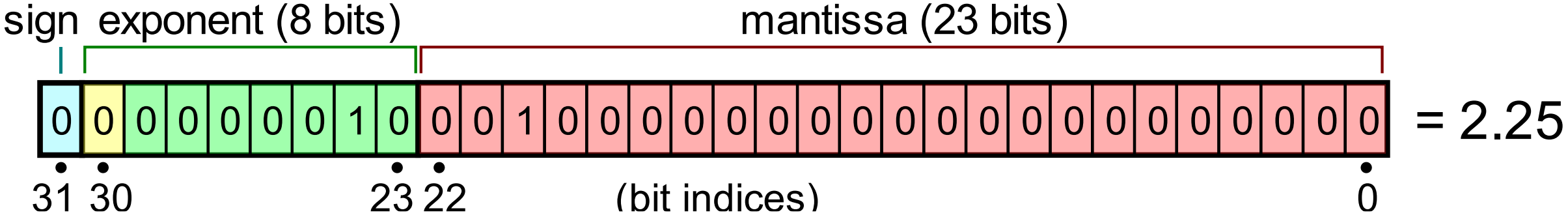
$$1101.011$$

$$1 \times 2^3 + 1 \times 2^2 + 0 \times 2^1 + 1 \times 2^0 + 0 \times 2^{-1} + 1 \times 2^{-2} + 1 \times 2^{-3}$$

In other words: 13.375

$$1 \times 10^1 + 3 \times 10^0 + 3 \times 10^{-1} + 7 \times 10^{-2} + 5 \times 10^{-3}$$

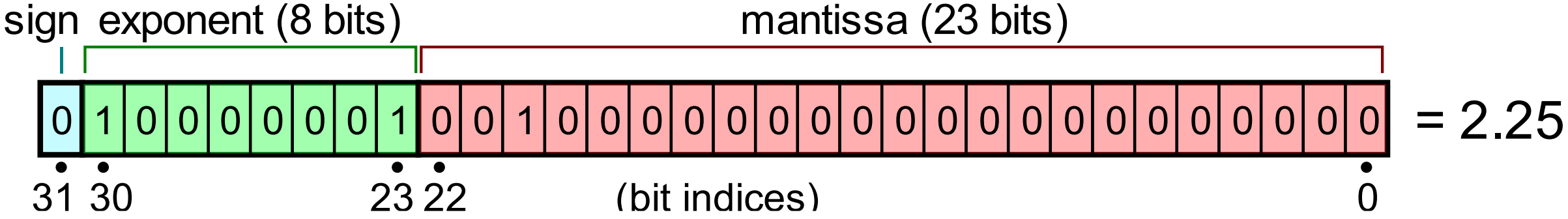
Floating-Point Numbers (simplified)



An example for a 32-bit floating-point number (4 bytes). The mantissa has 23 bits, the exponent has 8 bits, and there is a sign bit to store the sign of the number (positive or negative).

Only the fractional digits of the mantissa are stored because the leading digit is known to be 1.

Floating-Point Numbers



In practice, the 8-bit exponent $e \in [-127,128]$ is stored as $e + 127$.

In practice, the exponent bit patterns 00000000 and 11111111 are reserved for special values (infinities, zeroes, not-a-numbers, …), so the true exponent range is $e \in [-126,127]$.

Floating-Point Numbers

Popular formats:

Single precision	32 bits, 23+1 bit mantissa, 8 bit exponent
Double precision	64 bits, 53+1 bit mantissa, 10 bit exponent

Also present in most implementations

Half precision	16 bits, 10+1 bit mantissa, 5 bit exponent
Quadruple precision	128 bits, 112+1 bit mantissa, 15 bit exponent

Floating-Point Arithmetic on Computers

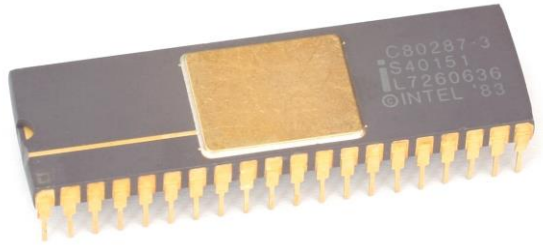
Floating-Point arithmetic is implemented on specialized chips on mainboards and graphics cards

Floating-point computations appear in applications such as

- Avionics, GPS
- Computer graphics
- Industrial computer simulations
- Climate simulations

Once several competing industrial standards, since 1985 we have the IEEE 754 as technical standard (2019, 84 pages).

Standardized formats for floating-point numbers with 4, 8 and 16 bytes.



FINAL POINT

What's the floating point?

- Computers perform inexact computations with floating-point arithmetic, not exact computations with real numbers
- Floating-point numbers are a crucial technology for modern computing with many intricate details
- Rounding errors are manageable in practice, but we need to be careful. Some problems are more sensitive to rounding errors than others.
- **In numerical analysis:**
  Most algorithms are stated with exact arithmetic. Behavior under inexact computation is added at a later stage.

A few practical pointers

- In coding, make a distinction between quantities that you represent exactly (integers) and quantities that potentially inexact (floating-point numbers)
- Assume that any floating-point number is subject to rounding errors unless you know the technical details
- Read about the technical details if you need to approach this topic professionally
- Don’t compare floating-point numbers for exactness.