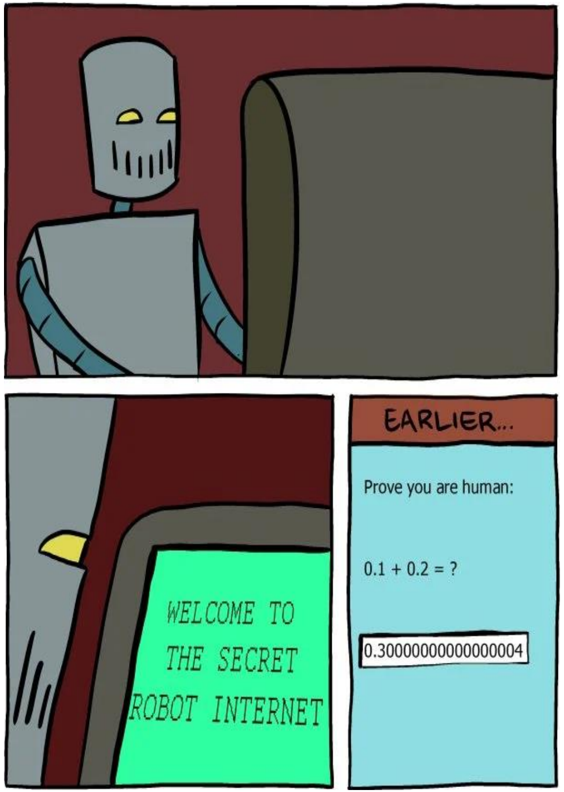
Floating Point



https://0.30000000000000004.com

Number Representation Revisited

- What can we represent so far?
  - Signed and unsigned integers
  - Characters
  - Addresses

- How do we encode the following?
  - Real numbers (ex: 3.14159)
  - Very large numbers (ex: $6.02*10^{23}$)
  - Very small numbers (ex: $6.26*10^{-34}$)
  - Special cases (ex: ∞, NaN)

Floating Point Topics

- **Fractional binary numbers (fixed point)**
- **Floating point**
  - **IEEE standard**
- Floating point operations and rounding

Binary Representation of Fractions

- Let's start by looking at base 10:
  - Each place represents a power of 10. Power decreases as you read left->right
  - Decimal point marks when the negative powers start

Ex: $\begin{matrix} 10^2 & 10^1 & 10^0 & & 10^{-1} & 10^{-2} & 10^{-3} \\ 1 & 2 & 3 & . & 4 & 5 & 6 \end{matrix}$

- Base 2 is similar:
  - Every place to the right of the **binary point** represents a negative power of 2

Ex: $\begin{matrix} 2^3 & 2^2 & 2^0 & & 2^{-1} & 2^{-2} & 2^{-3} \\ 1 & 0 & 1 & . & 0 & 1 & 1 \end{matrix}$

$= 1*2^2 + 0*2^1 + 1*2^0 + 0*2^{-1} + 1*2^{-2} + 1*2^{-3} = 5.375_{10}$

Limits of Representation

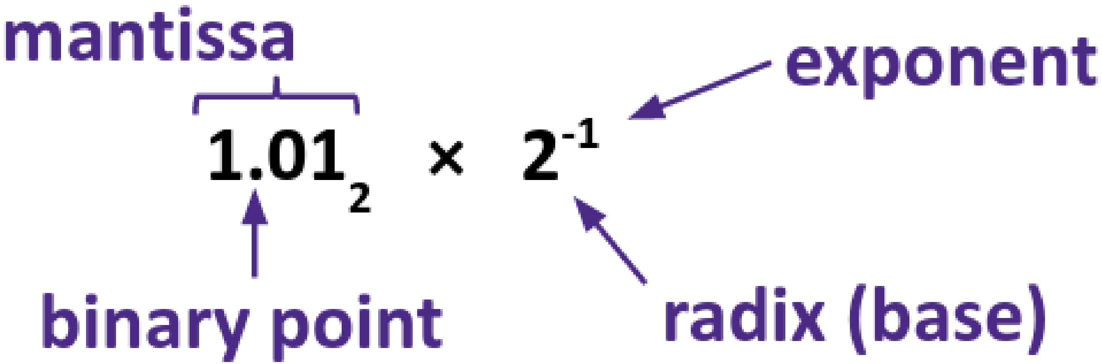
- Even with an arbitrary number of bits, we can only represent numbers of the form **$x*2^y$**
- Other rational numbers have infinite bit representations

Value	Binary Representation
$1/3 = 0.333333\ldots_{10}$	$0.01010101[01]\ldots_2$
$1/5 = 0.2_{10}$	$0.001100110011[0011]\ldots_2$
$1/10 = 0.1_{10}$	$0.0001100110011[0011]\ldots_2$

Floating Point Representation

- Based on **scientific notation**
  - In decimal:
    - 12000000 -> 1.2×10^7
    - 0.0000012 -> 1.2×10^{-6}
  - In binary:
    - 11000.000 -> 1.1×2^4
    - 0.00011 -> 1.1×2^{-4}
- Divvy up the bits in our encoding
  - Sign (+/-)
  - Exponent
  - Mantissa (everything after the binary point)

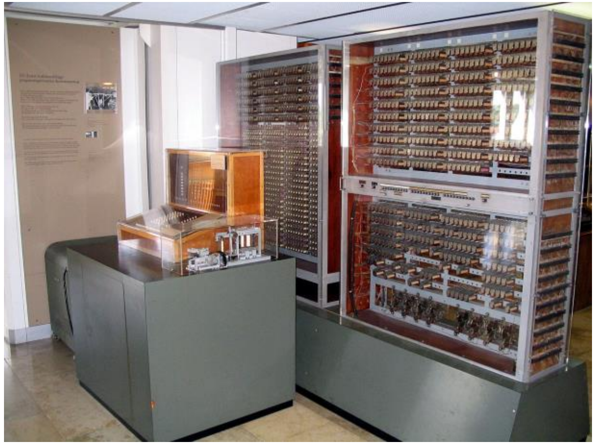
Binary Scientific Notation



- **Normalized form**: exactly one (non-zero) bit to the left of the binary point
- Called “floating point” because the binary point “floats” to different parts of the number (as opposed to fixed)

Floating Point History

- 1914: first design by Leonardo Torres y Quevedo
- 1940: implementations by Konrad Zuse, but not exactly the same as the modern standard
- 1985: IEEE 754 standard
    - Primary architect was William Kahan, who won a Turing Award for this work
    - Standardized bit encoding, well-defined behavior for *all* operations
    - Still what we use today!

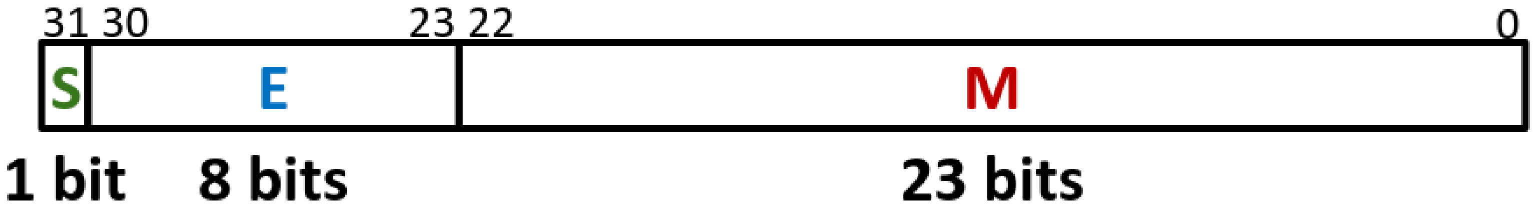


IEEE Floating Point

- IEEE 754 (established 1985)
  - Developed to make numerically-sensitive programs portable
  - Specifies two things: a *representation scheme* and the *result of operations*
  - Supported by all major CPUs
- Two opposing concerns:
  - **Scientists** numerical analysts want them to be as *real* as possible
  - **Engineers** want them to be *easy to implement* and *fast*
  - Who won? Mostly scientists
    - Nice standards for rounding, overflow, underflow, but complex for hardware
    - **Float operations can be an order of magnitude slower than integer ops!**
      - CPU speed commonly measured in **FLOPS** (float ops per second)

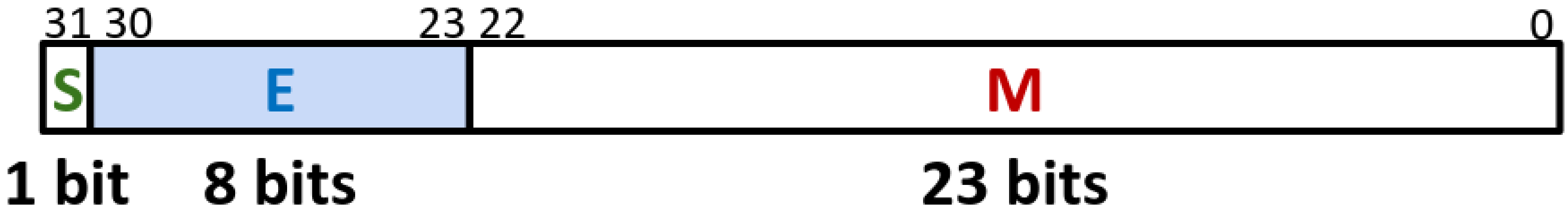
Floating Point Encoding (Review)

- Value = ±1.Mantissa * 2^{Exponent}
- Bit fields: $(-1)^{\mathbf{S}} * 1.\mathbf{M} * 2^{\mathbf{E}}$
- Representation scheme:
  - Sign bit (**S**): 0 is positive, 1 is negative
  - Mantissa (a.k.a. significand): the fractional part of the number in normalized form, encoded in the bit vector **M**
  - Exponent: weighs the number by a (possible negative) power of 2, encoded in the bit vector **E**



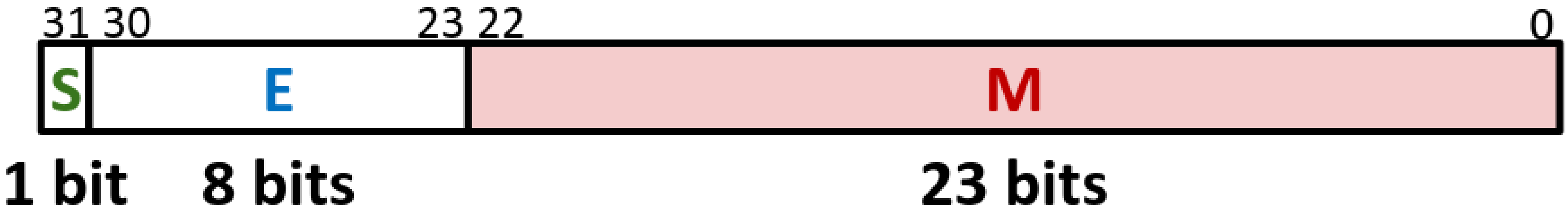
The Exponent Field

- Use **biased notation**
  - Read as unsigned, but with a **bias** of $2^{w-1}-1$ (127, for an 8-bit **E** field)
  - Exponent = **E** - bias ↔ **E** = Exponent + bias
- Why?
  - Makes floating point arithmetic easier
  - Somewhat compatible with two’s complement hardware



The Mantissa Field (Review)

- **Implicit leading 1** before the binary point
  - There's always a 1 there in normalized form, so we don't need to encode it!
  - Example: 0b 0011 1111 1100 0000 0000 0000 0000 0000
    - Read as $1.1_2 = 1.5_{10}$, *not* $0.1_2 = 0.5_{10}$
- Mantissa "limits"
  - Low values (near **M** = 0b00…00) are close to 2^{Exp}
  - High values (near **M** = 0b11…11) are close to 2^{Exp+1}



Normalized Floating Point Conversions

FP -> Decimal

1. Append bits of **M** to leading 1
2. Multiply by $2^{E\text{-bias}}$
3. "Multiply out" exponent by shifting
   a. If exp < 0, shift *right* (logical) by -exp
   a. If exp > 0, shift *left* by exp
4. Multiply by sign (-1^{S})
5. Convert from binary to decimal

Decimal -> FP

1. Convert from decimal to binary
2. Convert to normalized form
   a. Shift left or right (logical) until there's a single 1 before the binary point
   b. Multiply by 2^{exp}, where exp = number of places shifted (negative for left shift, positive for right)
3. **S** = 0 if positive, 1 if negative
4. **E** = exp + bias
5. **M** = bits after the binary point

Practice Question

Convert the decimal number -7.375 into floating point representation.

Challenge Question:

Find the value of the following sum in normalized binary scientific notation:

$$1.01_2 * 2^0 + 1.11_2 * 2^2$$

Floating Point Topics

- Fractional binary numbers (fixed point)
- Floating point
  - IEEE standard
- **Floating point operations and rounding**
- Floating point in C

Precision and Accuracy

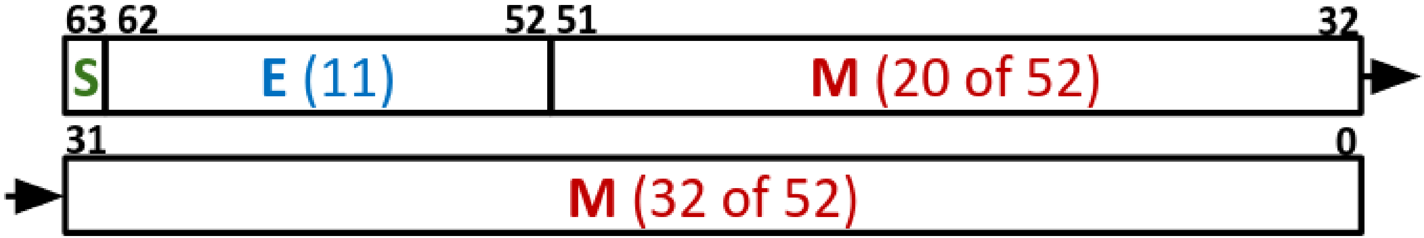
- **Accuracy** is a measure of the difference between the actual value of a number and its computer representation
- **Precision** is a count of the number of bits in a computer word used to represent a value
  - Capacity for accuracy
- **High precision permits high accuracy but doesn't guarantee it. It is possible to have high precision but low accuracy.**

Example: `float pi = 3.14;`

- `pi` will be represented with all 24 bits of mantissa (highly precise), but still an approximation

Need Greater Precision?

- 64 bits = **double precision**
- Exponent bias is now $2^{10}–1 = 1023$
- Advantages
  - Greater precision (larger mantissa), greater range (larger exponent field)
- Disadvantages
  - More space used, slower to manipulate



Representational Errors

- **Overflow** yields $\pm\infty$, **underflow** yields 0
- $\pm\infty$ and NaN can be used in operations
  - Result is usually still the same, but not always intuitive
- **Floating point operations do not work like real math, due to rounding**
  - Not associative
    - Ex: $(3.14 + 10^{100}) - 10^{100} \mathrel{!=} 3.14 + (10^{100} - 10^{100})$
  - Not distributive
    - Ex: $100*(0.1 + 0.2) \mathrel{!=} 100*0.1 + 100*0.2$
  - Not cumulative
    - Repeatedly adding a very small number to a very large one may do nothing

Why does this matter?

- **1982:** Vancouver Stock Exchange 10% error in less than 2 years
- **1991:** Patriot missile targeting error
  - Clock skew due to conversion from int to float
- **1994:** Intel Pentium FDIV (float division) hardware bug ($475 million)
- **1996:** Ariane 5 rocket exploded ($1 billion)
  - Overflow converting 64-bit float to 16-bit int
- **1997:** USS Yorktown “smart” warship stranded
  - Divide by zero

Summary

- Floating point approximates real numbers using **binary scientific notation**
  - Exponent in **biased notation**
- Standard encoding is **IEEE 754**
  - Defines standard bit width for fields, behavior in operations, and special cases
- Floats also suffer from having a fixed bit width
  - Can get **overflow**, but also **underflow** and **rounding**
- **Floating point arithmetic can have unexpected results!**
  - ***Never*** test floats for equality
- Conversion between float and other data types can cause errors
  - Be especially careful when converting between `int` and `float`!