

Floating Point



Today: Floating Point

- 🌀 Background: Fractional binary numbers
- 🌀 IEEE floating point standard: Definition
- 🌀 Example and properties
- 🌀 Rounding, addition, multiplication
- 🌀 Floating point in C
- 🌀 Summary

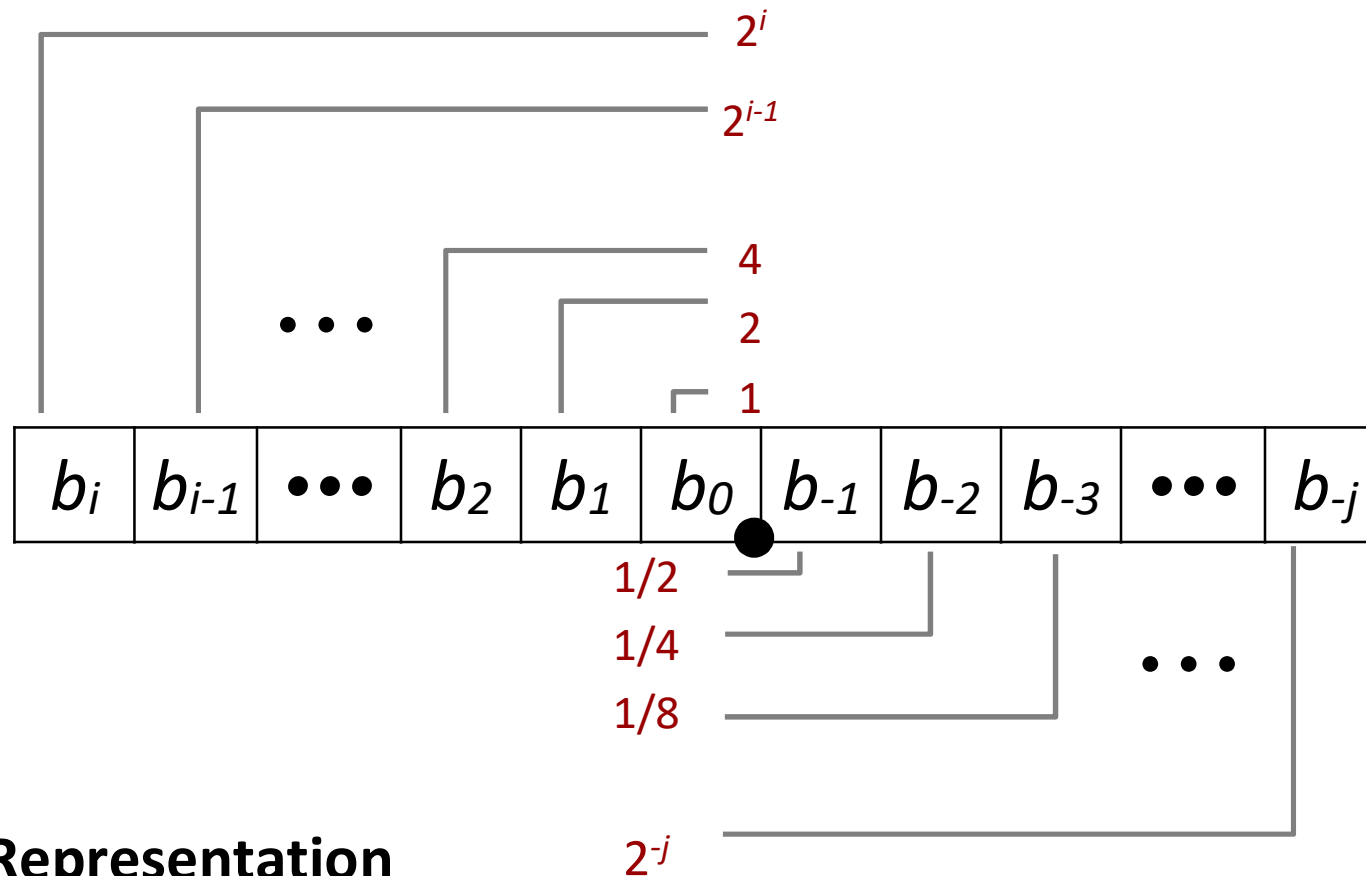


Fractional binary numbers

🔊 What is 1011.101_2 ?



Fractional Binary Numbers



Representation

- Bits to right of “binary point” represent fractional powers of 2
- Represents rational number:

$$\sum_{k=-j}^i b_k \times 2^k$$



Fractional Binary Numbers: Examples

■ Value Representation

$5 \frac{3}{4}$	101.11_2
$2 \frac{7}{8}$	10.111_2
$1 \frac{7}{16}$	1.0111_2

■ Observations

- Divide by 2 by shifting right (unsigned)
- Multiply by 2 by shifting left
- Numbers of form $0.111111..._2$ are just below 1.0
 - $1/2 + 1/4 + 1/8 + \dots + 1/2^i + \dots \rightarrow 1.0$
 - Use notation $1.0 - \epsilon$



Representable Numbers

Limitation #1

- Can only exactly represent numbers of the form $x/2^k$
 - Other rational numbers have repeating bit representations
- | Value | Representation |
|--------|----------------------------------|
| $1/3$ | $0.0101010101 [01] \dots_2$ |
| $1/5$ | $0.001100110011 [0011] \dots_2$ |
| $1/10$ | $0.0001100110011 [0011] \dots_2$ |

Limitation #2

- Just one setting of binary point within the w bits
 - Limited range of numbers (very small values? very large?)



Today: Floating Point

- ⌚ Background: Fractional binary numbers
- ⌚ **IEEE floating point standard: Definition**
- ⌚ Example and properties
- ⌚ Rounding, addition, multiplication
- ⌚ Floating point in C
- ⌚ Summary



IEEE Floating Point

• IEEE Standard 754

- Established in 1985 as uniform standard for floating point arithmetic
 - Before that, many idiosyncratic formats
- Supported by all major CPUs

• Driven by numerical concerns

- Nice standards for rounding, overflow, underflow
- Hard to make fast in hardware
 - Numerical analysts predominated over hardware designers in defining standard



Floating Point Representation

🌀 Numerical Form:

$$(-1)^s M 2^E$$

- 🌀 **Sign bit s** determines whether number is negative or positive
- 🌀 **Significand M** normally a fractional value in range $[1.0, 2.0)$.
- 🌀 **Exponent E** weights value by power of two

🌀 Encoding

- 🌀 MSB s is sign bit s
- 🌀 exp field encodes E (but is not equal to E)
- 🌀 frac field encodes M (but is not equal to M)



Precision options

Single precision: 32 bits



Double precision: 64 bits



Extended precision: 80 bits (Intel only)



“Normalized” Values

$$v = (-1)^s M 2^E$$

- When: $\text{exp} \neq 000\dots 0$ and $\text{exp} \neq 111\dots 1$
- Exponent coded as a *biased* value: $E = \text{Exp} - \text{Bias}$
 - Exp : unsigned value of exp field
 - $\text{Bias} = 2^{k-1} - 1$, where k is number of exponent bits
 - Single precision: 127 (Exp: 1...254, E: -126...127)
 - Double precision: 1023 (Exp: 1...2046, E: -1022...1023)
- Significand coded with implied leading 1: $M = 1.\text{xxx}\dots\text{x}_2$
 - $\text{xxx}\dots\text{x}$: bits of frac field
 - Minimum when $\text{frac} = 000\dots 0$ ($M = 1.0$)
 - Maximum when $\text{frac} = 111\dots 1$ ($M = 2.0 - \epsilon$)
 - Get extra leading bit for “free”



Normalized Encoding Example

$$V = (-1)^S M 2^E$$
$$E = \text{Exp} - \text{Bias}$$

Value: float $F = 15213.0;$

$$15213_{10} = 11101101101101_2$$
$$= 1.1101101101101_2 \times 2^{13}$$

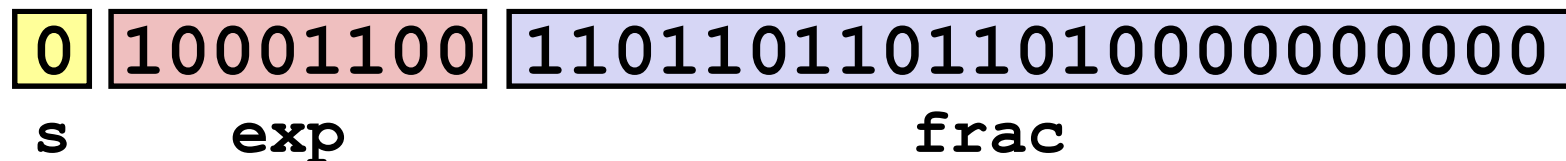
Significand

$$M = 1.\underline{1101101101101}_2$$
$$\text{frac} = \underline{1101101101101}0000000000_2$$

Exponent

$$E = 13$$
$$\text{Bias} = 127$$
$$\text{Exp} = 140 = 10001100_2$$

Result:



Denormalized Values

$$V = (-1)^S M 2^E$$
$$E = 1 - \text{Bias}$$

- Condition: $\text{exp} = 000\dots 0$
- Exponent value: $E = 1 - \text{Bias}$ (instead of $E = 0 - \text{Bias}$)
- Significand coded with implied leading 0: $M = 0.\text{xxx}\dots\text{x}_2$
 - $\text{xxx}\dots\text{x}$: bits of frac
- Cases
 - $\text{exp} = 000\dots 0, \text{frac} = 000\dots 0$
 - Represents zero value
 - Note distinct values: $+0$ and -0
 - $\text{exp} = 000\dots 0, \text{frac} \neq 000\dots 0$
 - Numbers closest to 0.0
 - Equispaced



Special Values

• Condition: **exp** = 111...1

• Case: **exp** = 111...1, **frac** = 000...0

• Represents value ∞ (infinity)

• Operation that overflows

• Both positive and negative

• E.g., $1.0/0.0 = -1.0/-0.0 = +\infty$, $1.0/-0.0 = -\infty$

• Case: **exp** = 111...1, **frac** \neq 000...0

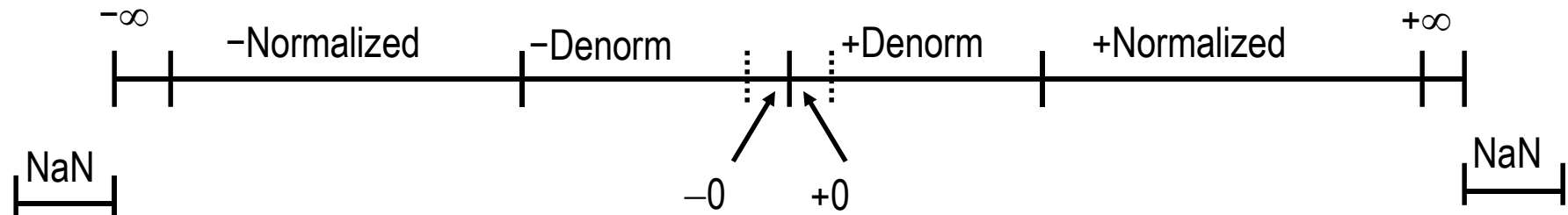
• Not-a-Number (NaN)

• Represents case when no numeric value can be determined

• E.g., $\text{sqrt}(-1)$, $\infty - \infty$, $\infty \times 0$



Visualization: Floating Point Encodings

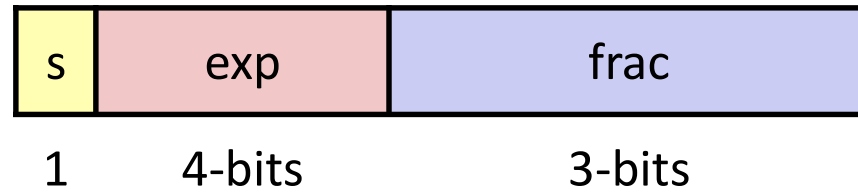


Today: Floating Point

- ⌚ Background: Fractional binary numbers
- ⌚ IEEE floating point standard: Definition
- ⌚ Example and properties**
- ⌚ Rounding, addition, multiplication
- ⌚ Floating point in C
- ⌚ Summary



Tiny Floating Point Example



8-bit Floating Point Representation

- the sign bit is in the most significant bit
- the next four bits are the exponent, with a bias of 7
- the last three bits are the **frac**

Same general form as IEEE Format

- normalized, denormalized
- representation of 0, NaN, infinity



Dynamic Range (Positive Only)

	s	exp	frac	E	Value	
Denormalized numbers	0	0000	000	-6	0	
	0	0000	001	-6	$1/8 * 1/64 = 1/512$	
	0	0000	010	-6	$2/8 * 1/64 = 2/512$	
	...					
	0	0000	110	-6	$6/8 * 1/64 = 6/512$	
	0	0000	111	-6	$7/8 * 1/64 = 7/512$	largest denorm
	0	0001	000	-6	$8/8 * 1/64 = 8/512$	smallest norm
Normalized numbers	0	0001	001	-6	$9/8 * 1/64 = 9/512$	
	...					
	0	0110	110	-1	$14/8 * 1/2 = 14/16$	
	0	0110	111	-1	$15/8 * 1/2 = 15/16$	closest to 1 below
	0	0111	000	0	$8/8 * 1 = 1$	
	0	0111	001	0	$9/8 * 1 = 9/8$	closest to 1 above
	0	0111	010	0	$10/8 * 1 = 10/8$	
	...					
	0	1110	110	7	$14/8 * 128 = 224$	
	0	1110	111	7	$15/8 * 128 = 240$	largest norm
	0	1111	000	n/a	inf	

$$V = (-1)^s M 2^E$$

n: $E = \text{Exp} - \text{Bias}$

d: $E = 1 - \text{Bias}$

closest to zero

largest denorm

smallest norm

closest to 1 below

closest to 1 above

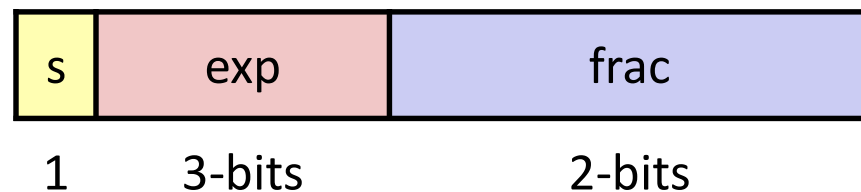
largest norm



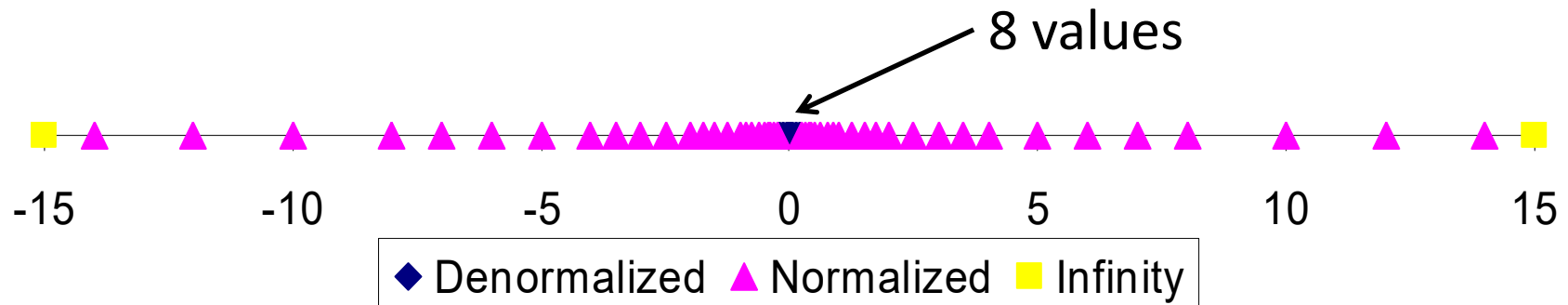
Distribution of Values

6-bit IEEE-like format

- e = 3 exponent bits
- f = 2 fraction bits
- Bias is $2^{3-1}-1 = 3$



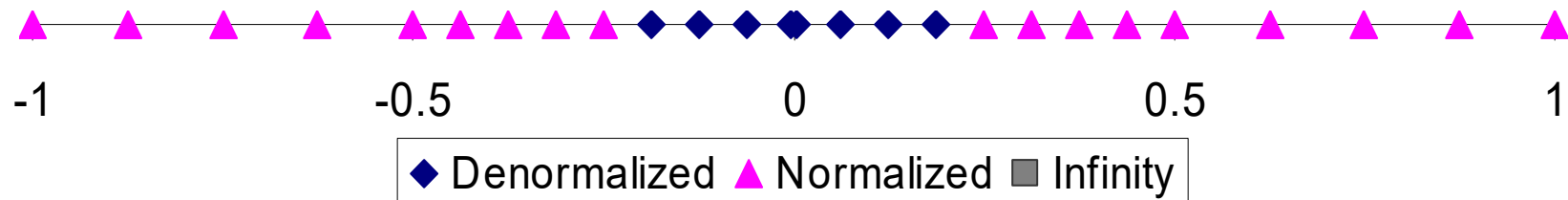
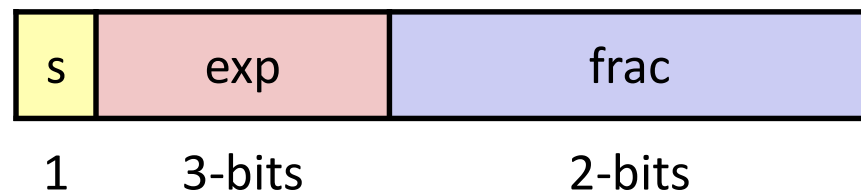
Notice how the distribution gets denser toward zero.



Distribution of Values (close-up view)

6-bit IEEE-like format

- e = 3 exponent bits
- f = 2 fraction bits
- Bias is 3



Special Properties of the IEEE Encoding

• FP Zero Same as Integer Zero

- All bits = 0

• Can (Almost) Use Unsigned Integer Comparison

- Must first compare sign bits
- Must consider $-0 = 0$
- NaNs problematic
 - Will be greater than any other values
 - What should comparison yield?
- Otherwise OK
 - Denorm vs. normalized
 - Normalized vs. infinity



Today: Floating Point

- ⌚ Background: Fractional binary numbers
- ⌚ IEEE floating point standard: Definition
- ⌚ Example and properties
- ⌚ **Rounding, addition, multiplication**
- ⌚ Floating point in C
- ⌚ Summary



Floating Point Operations: Basic Idea

$$\textcircled{a} \quad x +_f y = \text{Round}(x + y)$$

$$\textcircled{a} \quad x \times_f y = \text{Round}(x \times y)$$

\textcircled{a} Basic idea

- \textcircled{a} First **compute exact result**
- \textcircled{a} Make it fit into desired precision
 - \textcircled{a} Possibly overflow if exponent too large
 - \textcircled{a} Possibly **round to fit into frac**



Rounding

🌀 Rounding Modes (illustrate with \$ rounding)

🌀	\$1.40	\$1.60	\$1.50	\$2.50	-\$1.50
🌀 Towards zero	\$1	\$1	\$1	\$2	-\$1
🌀 Round down ($-\infty$)	\$1	\$1	\$1	\$2	-\$2
🌀 Round up ($+\infty$)	\$2	\$2	\$2	\$3	-\$1
🌀 Nearest Even (default)	\$1	\$2	\$2	\$2	-\$2



Closer Look at Round-To-Even

🌀 Default Rounding Mode

- 🌀 Hard to get any other kind without dropping into assembly
- 🌀 All others are statistically biased
 - 🌀 Sum of set of positive numbers will consistently be over- or under-estimated

🌀 Applying to Other Decimal Places / Bit Positions

- 🌀 When exactly halfway between two possible values
 - 🌀 Round so that least significant digit is even
- 🌀 E.g., round to nearest hundredth

7.8949999	7.89	(Less than half way)
7.8950001	7.90	(Greater than half way)
7.8950000	7.90	(Half way—round up)
7.8850000	7.88	(Half way—round down)



Rounding Binary Numbers

Binary Fractional Numbers

- “Even” when least significant bit is 0
- “Half way” when bits to right of rounding position = $100..._2$

Examples

- Round to nearest $1/4$ (2 bits right of binary point)

Value	Binary	Rounded	Action	Rounded
Value				
$2 \frac{3}{32}$	$10.00\textcolor{red}{011}_2$	10.00_2	($<1/2$ —down)	2
$2 \frac{3}{16}$	$10.00\textcolor{red}{110}_2$	10.01_2	($>1/2$ —up)	$2 \frac{1}{4}$
$2 \frac{7}{8}$	$10.11\textcolor{red}{100}_2$	11.00_2	($1/2$ —up)	3
$2 \frac{5}{8}$	$10.10\textcolor{red}{100}_2$	10.10_2	($1/2$ —down)	$2 \frac{1}{2}$



FP Multiplication

• $(-1)^{s1} M1 2^{E1} \times (-1)^{s2} M2 2^{E2}$

• **Exact Result:** $(-1)^s M 2^E$

- Sign s : $s1 \wedge s2$
- Significand M : $M1 \times M2$
- Exponent E : $E1 + E2$

• Fixing

- If $M \geq 2$, shift M right, increment E
- If E out of range, overflow
- Round M to fit **frac** precision

• Implementation

- Biggest chore is multiplying significands



Floating Point Addition

• $(-1)^{s1} M1 2^{E1} + (-1)^{s2} M2 2^{E2}$

• Assume $E1 > E2$

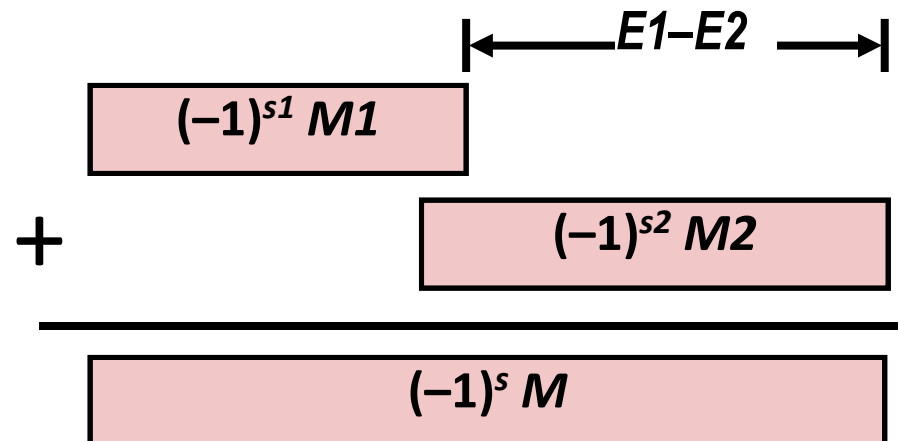
Get binary points lined up

• **Exact Result:** $(-1)^s M 2^E$

• Sign s , significand M :

• Result of signed align & add

• Exponent E : $E1$



Fixing

• If $M \geq 2$, shift M right, increment E

• If $M < 1$, shift M left k positions, decrement E by k

• Overflow if E out of range

• Round M to fit **frac** precision



Today: Floating Point

- ⌚ Background: Fractional binary numbers
- ⌚ IEEE floating point standard: Definition
- ⌚ Example and properties
- ⌚ Rounding, addition, multiplication
- ⌚ Floating point in C**
- ⌚ Summary



Floating Point in C

• C Guarantees Two Levels

• **float** single precision

• **double** double precision

• Conversions/Casting

• Casting between **int**, **float**, and **double** changes bit representation

• **double/float** → **int**

- Truncates fractional part
- Like rounding toward zero
- Not defined when out of range or NaN: Generally sets to TMin

• **int** → **double**

- Exact conversion, as long as **int** has ≤ 53 bit word size

• **int** → **float**

- Will round according to rounding mode



Floating Point Puzzles

🌀 For each of the following C expressions, either:

- 🌀 Argue that it is true for all argument values
- 🌀 Explain why not true

```
int x = ...;  
float f = ...;  
double d = ...;
```

Assume neither
d nor **f** is NaN

- `x == (int)(float) x` 🗨️
- `x == (int)(double) x` 🗨️
- `f == (float)(double) f` 🗨️
- `d == (double)(float) d` 🗨️
- `f == -(-f);` 🗨️
- `2/3 == 2/3.0` 🗨️
- `d < 0.0` \Rightarrow `((d*2) < 0.0)` 🗨️
- `d > f` \Rightarrow `-f > -d` 🗨️
- `d * d >= 0.0` 🗨️
- `(d+f) - d == f` 🗨️



Summary

- IEEE Floating Point has clear mathematical properties
- Represents numbers of form $M \times 2^E$
- One can reason about operations independent of implementation
 - As if computed with perfect precision and then rounded
- Not the same as real arithmetic
 - Violates associativity/distributivity
 - Makes life difficult for compilers & serious numerical applications programmers

