

Computer Arithmetic

David Marchant
Based on slides by Troels Henriksen

2022-09-13

Inspired by slides by Randal E. Bryant and David R. O'Hallaron.
Some material by Michael Kirkedal Tomsen.

Agenda

Floating point arithmetic

- Biased numbers

- Background: Fractional binary numbers

- IEEE floating point standard

- Examples and properties

- Rounding, addition, and multiplication

- Floating point in C

Summary

Floating point arithmetic

Biased numbers

Background: Fractional binary numbers

IEEE floating point standard

Examples and properties

Rounding, addition, and multiplication

Floating point in C

Summary

Biased number representation

For *biased numbers*, the raw bits are interpreted as unsigned, and then a constant *bias* is subtracted.

Unsigned

$$B2U(X) = \sum_{i=0}^{w-1} x_i \cdot 2^i$$

Two's complement

$$B2S(X) = -x_{w-1} \cdot 2^{w-1} + \sum_{i=0}^{w-2} x_i \cdot 2^i$$

Biased

$$B2I(X) = \left(\sum_{i=0}^{w-1} x_i \cdot 2^i \right) - \text{Bias}$$

- Typically

$$\text{Bias} = 2^{w-1} - 1$$

- Examples for $w = 8$, Bias = 127

	B2U	B2S	B2I
<hr/>			
00000000 ₂			

Biased number representation

For *biased numbers*, the raw bits are interpreted as unsigned, and then a constant *bias* is subtracted.

Unsigned

$$B2U(X) = \sum_{i=0}^{w-1} x_i \cdot 2^i$$

Two's complement

$$B2S(X) = -x_{w-1} \cdot 2^{w-1} + \sum_{i=0}^{w-2} x_i \cdot 2^i$$

Biased

$$B2I(X) = \left(\sum_{i=0}^{w-1} x_i \cdot 2^i \right) - \text{Bias}$$

- Typically

$$\text{Bias} = 2^{w-1} - 1$$

- Examples for $w = 8$, Bias = 127

	B2U	B2S	B2I
00000000 ₂	0 ₁₀		

Biased number representation

For *biased numbers*, the raw bits are interpreted as unsigned, and then a constant *bias* is subtracted.

Unsigned

$$B2U(X) = \sum_{i=0}^{w-1} x_i \cdot 2^i$$

Two's complement

$$B2S(X) = -x_{w-1} \cdot 2^{w-1} + \sum_{i=0}^{w-2} x_i \cdot 2^i$$

Biased

$$B2I(X) = \left(\sum_{i=0}^{w-1} x_i \cdot 2^i \right) - \text{Bias}$$

- Typically

$$\text{Bias} = 2^{w-1} - 1$$

- Examples for $w = 8$, Bias = 127

	B2U	B2S	B2I
00000000 ₂	0 ₁₀	0 ₁₀	

Biased number representation

For *biased numbers*, the raw bits are interpreted as unsigned, and then a constant *bias* is subtracted.

Unsigned

$$B2U(X) = \sum_{i=0}^{w-1} x_i \cdot 2^i$$

Two's complement

$$B2S(X) = -x_{w-1} \cdot 2^{w-1} + \sum_{i=0}^{w-2} x_i \cdot 2^i$$

Biased

$$B2I(X) = \left(\sum_{i=0}^{w-1} x_i \cdot 2^i \right) - \text{Bias}$$

- Typically

$$\text{Bias} = 2^{w-1} - 1$$

- Examples for $w = 8$, Bias = 127

	B2U	B2S	B2I
00000000 ₂	0 ₁₀	0 ₁₀	-127 ₁₀
01111111 ₂			

Biased number representation

For *biased numbers*, the raw bits are interpreted as unsigned, and then a constant *bias* is subtracted.

Unsigned

$$B2U(X) = \sum_{i=0}^{w-1} x_i \cdot 2^i$$

Two's complement

$$B2S(X) = -x_{w-1} \cdot 2^{w-1} + \sum_{i=0}^{w-2} x_i \cdot 2^i$$

Biased

$$B2I(X) = \left(\sum_{i=0}^{w-1} x_i \cdot 2^i \right) - \text{Bias}$$

- Typically

$$\text{Bias} = 2^{w-1} - 1$$

- Examples for $w = 8$, Bias = 127

	B2U	B2S	B2I
00000000 ₂	0 ₁₀	0 ₁₀	-127 ₁₀
01111111 ₂	127 ₁₀		

Biased number representation

For *biased numbers*, the raw bits are interpreted as unsigned, and then a constant *bias* is subtracted.

Unsigned

$$B2U(X) = \sum_{i=0}^{w-1} x_i \cdot 2^i$$

Two's complement

$$B2S(X) = -x_{w-1} \cdot 2^{w-1} + \sum_{i=0}^{w-2} x_i \cdot 2^i$$

Biased

$$B2I(X) = \left(\sum_{i=0}^{w-1} x_i \cdot 2^i \right) - \text{Bias}$$

- Typically

$$\text{Bias} = 2^{w-1} - 1$$

- Examples for $w = 8$, Bias = 127

	B2U	B2S	B2I
00000000 ₂	0 ₁₀	0 ₁₀	-127 ₁₀
01111111 ₂	127 ₁₀	127 ₁₀	

Biased number representation

For *biased numbers*, the raw bits are interpreted as unsigned, and then a constant *bias* is subtracted.

Unsigned

Two's complement

Biased

$$B2U(X) = \sum_{i=0}^{w-1} x_i \cdot 2^i$$

$$B2S(X) = -x_{w-1} \cdot 2^{w-1} + \sum_{i=0}^{w-2} x_i \cdot 2^i$$

$$B2I(X) = \left(\sum_{i=0}^{w-1} x_i \cdot 2^i \right) - \text{Bias}$$

- Typically

$$\text{Bias} = 2^{w-1} - 1$$

- Examples for $w = 8$, Bias = 127

	B2U	B2S	B2I
00000000 ₂	0 ₁₀	0 ₁₀	-127 ₁₀
01111111 ₂	127 ₁₀	127 ₁₀	0 ₁₀
11111111 ₂			

Biased number representation

For *biased numbers*, the raw bits are interpreted as unsigned, and then a constant *bias* is subtracted.

Unsigned

Two's complement

Biased

$$B2U(X) = \sum_{i=0}^{w-1} x_i \cdot 2^i$$

$$B2S(X) = -x_{w-1} \cdot 2^{w-1} + \sum_{i=0}^{w-2} x_i \cdot 2^i$$

$$B2I(X) = \left(\sum_{i=0}^{w-1} x_i \cdot 2^i \right) - \text{Bias}$$

- Typically

$$\text{Bias} = 2^{w-1} - 1$$

- Examples for $w = 8$, Bias = 127

	B2U	B2S	B2I
00000000 ₂	0 ₁₀	0 ₁₀	-127 ₁₀
01111111 ₂	127 ₁₀	127 ₁₀	0 ₁₀
11111111 ₂	255 ₁₀		

Biased number representation

For *biased numbers*, the raw bits are interpreted as unsigned, and then a constant *bias* is subtracted.

Unsigned

Two's complement

Biased

$$B2U(X) = \sum_{i=0}^{w-1} x_i \cdot 2^i \quad B2S(X) = -x_{w-1} \cdot 2^{w-1} + \sum_{i=0}^{w-2} x_i \cdot 2^i \quad B2I(X) = \left(\sum_{i=0}^{w-1} x_i \cdot 2^i \right) - \text{Bias}$$

- Typically

$$\text{Bias} = 2^{w-1} - 1$$

- Examples for $w = 8$, Bias = 127

	B2U	B2S	B2I
00000000 ₂	0 ₁₀	0 ₁₀	-127 ₁₀
01111111 ₂	127 ₁₀	127 ₁₀	0 ₁₀
11111111 ₂	255 ₁₀	-1 ₁₀	

Biased number representation

For *biased numbers*, the raw bits are interpreted as unsigned, and then a constant *bias* is subtracted.

Unsigned

Two's complement

Biased

$$B2U(X) = \sum_{i=0}^{w-1} x_i \cdot 2^i$$

$$B2S(X) = -x_{w-1} \cdot 2^{w-1} + \sum_{i=0}^{w-2} x_i \cdot 2^i$$

$$B2I(X) = \left(\sum_{i=0}^{w-1} x_i \cdot 2^i \right) - \text{Bias}$$

- Typically

$$\text{Bias} = 2^{w-1} - 1$$

- Examples for $w = 8$, Bias = 127

	B2U	B2S	B2I
00000000 ₂	0 ₁₀	0 ₁₀	-127 ₁₀
01111111 ₂	127 ₁₀	127 ₁₀	0 ₁₀
11111111 ₂	255 ₁₀	-1 ₁₀	128 ₁₀

Floating point arithmetic

Biased numbers

Background: Fractional binary numbers

IEEE floating point standard

Examples and properties

Rounding, addition, and multiplication

Floating point in C

Summary

Integral binary numbers

We have seen that

$$10010101_2$$

is basically interpreted like

$$149_{10}$$

“Structure” is the same, just with 2 instead of 10.

Integral binary numbers

We have seen that

$$10010101_2$$

is basically interpreted like

$$149_{10}$$

“Structure” is the same, just with 2 instead of 10.

Can we do the same thing for fractional numbers?

$$1011.101_2$$

Fractional numbers

$$123.456 = 1 \cdot 10^2 + 2 \cdot 10^1 + 3 \cdot 10^0 + 4 \cdot 10^{-1} + 5 \cdot 10^{-2} + 6 \cdot 10^{-3}$$

Fractional numbers

$$123.456 = 1 \cdot 10^2 + 2 \cdot 10^1 + 3 \cdot 10^0 + 4 \cdot 10^{-1} + 5 \cdot 10^{-2} + 6 \cdot 10^{-3}$$

Generally

$$a_{m-1} \cdots a_0 . a_{-1} \cdots a_{-n} = \sum_{i=-n}^{m-1} a_i \cdot 10^i$$

Fractional numbers

$$123.456 = 1 \cdot 10^2 + 2 \cdot 10^1 + 3 \cdot 10^0 + 4 \cdot 10^{-1} + 5 \cdot 10^{-2} + 6 \cdot 10^{-3}$$

Generally

$$a_{m-1} \cdots a_0.a_{-1} \cdots a_{-n} = \sum_{i=-n}^{m-1} a_i \cdot 10^i$$

Even more generally, for radix r

$$a_{m-1} \cdots a_0.a_{-1} \cdots a_{-n} = \sum_{i=-n}^{m-1} a_i \cdot r^i$$

Fractional binary numbers

Weight	2^{m-1}	2^{m-2}	\dots	4	2	1	$1/2$	$1/4$	$1/8$	\dots	2^{-n}
Digits	b_{m-1}	b_{m-2}	\dots	b_2	b_1	b_0	b_{-1}	b_{-2}	b_{-3}	\dots	b_{-n}

Representation

- Bits to the right of “binary point” represents fractional powers of 2.
- Represents rational number.

$$b_{m-1} \dots b_0 . b_{-1} \dots b_{-n} = \sum_{i=-n}^{m-1} b_i \cdot 2^i$$

Examples of fractional binary numbers

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 logical shifting right.
- Multiply by 2 by shifting left.
- Numbers of form $0.111\dots$ are just below 1.0.
 - ▶ $1/2 + 1/4 + 1/8 + \dots + 1/2^n + \dots \sim 1.0$.
 - ▶ Use notation $1.0 - \epsilon$.

Representable numbers

Limitation #1

- Can only represent fractional part of form $x/2^k$
- Other rational numbers have repeating bit representation

Value	Representation
$\frac{1}{3}$	$0.0101010101[01] \cdots_2$
$\frac{1}{5}$	$0.001100110011[0011] \cdots_2$
$\frac{1}{10}$	$0.0001100110011[0011] \cdots_2$

Limitation #2

- Just one setting of binary point within the w bits.
 - ▶ Limited range of numbers—very small values? Very large?

The fixed-point dilemma

Consider $w = 8$

1 bit for fraction

- Largest number: $1111111.1_2 = 127.5_{10}$
- Increment: $0000000.1_2 = 0.5_{10}$

7 bits for fraction

- Largest number: $1.1111111_2 = 1.9921875_{10}$
- Increment: $0.0000001_2 = 0.0078125_{10}$

4 bits for fraction

- Largest number: $1111.1111_2 = 15.9375_{10}$
- Increment: $0000.0001_2 = 0.0625_{10}$

Fixed-point has same absolute precision everywhere, but this means relative precision is worse for numbers close to 0!

Floating point arithmetic

Biased numbers

Background: Fractional binary numbers

IEEE floating point standard

Examples and properties

Rounding, addition, and multiplication

Floating point in C

Summary

IEEE Floating Point

IEEE Standard 754

- Established in 1985 as uniform standard for floating point.
 - ▶ Many idiosyncratic formats before then.
- Supported by all major CPUs, GPUs, and most other processors.

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.
 - ▶ ... but (later) Turing Award winner William Kahan secretly knew that Intel had figured out how to make accurate computation *fast*.
 - ▶ **Beware the wrath of Kahan!**
 - ▶ <http://people.eecs.berkeley.edu/~wkahan/>

Essentially scientific notation

$$3.5 \times 10^2 = 350$$

- **Significand** is 3.5
 - ▶ Conventionally a number in range $[1, 10)$, with sign.
- **Exponent** is 2.
 - ▶ Can also be negative.

Essentially scientific notation

$$3.5 \times 10^2 = 350$$

- **Significand** is 3.5
 - ▶ Conventionally a number in range $[1, 10)$, with sign.
- **Exponent** is 2.
 - ▶ Can also be negative.

To keep significand in range, *adjust exponent*:

$$\begin{aligned} 35 \times 10^1 &\Rightarrow 3.5 \times 10^2 \\ 0.35 \times 10^3 &\Rightarrow 3.5 \times 10^2 \end{aligned}$$

IEEE 754 uses bits instead of digits, and specifies a fixed-size encoding, but idea is the same.

Floating Point Representation

Numerical form

$$(-1)^S \cdot M \cdot 2^E$$

- **Sign bit** S determines whether number negative or positive.
- **Significand** M normally a fractional value in range $[1, 2)$.
- **Exponent** E weights value by power of two.

Encoding

- Most significant bit is sign bit.
- Exp field encodes E (but is not equal to E).
- Frac field encodes M (but is not equal to M).

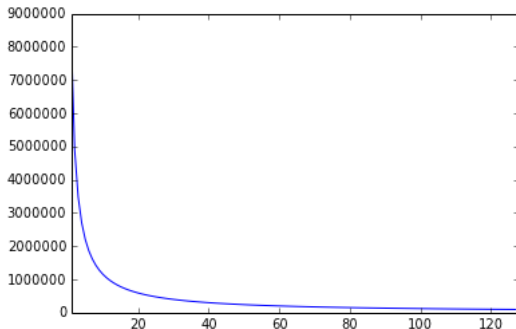
yellow!30S	red!30Exp	blue!30Frac
------------	-----------	-------------

Why such a weird format?

The point is floating

- No fixed number of bits allocated to “fraction”.
- More bits close to 0, fewer bits for numbers with large magnitude.
- Symmetric around 0.

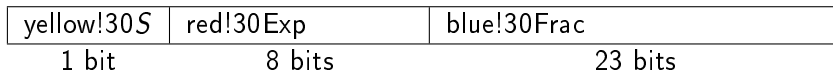
Density of floats



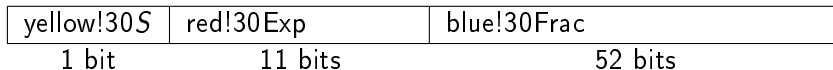
<https://stackoverflow.com/a/24179424/6131552>

Precision options

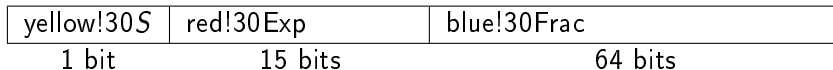
32-bit single precision: **float**



64-bit double precision: **double**



80-bit Extended precision (Intel only, never use): **long double**



Newer standards contain more variants (16 bits, decimal floats) that we will not cover.

Main problem: not all numbers are representable

Format trouble

For example the number

$$0.1_{10}$$

cannot be represented on the form

$$(-1)^S \cdot M \cdot 2^E$$

Precision trouble

- A fixed number of bits cannot represent all numbers.
- Integer types represent an interval of natural numbers.
- *Any nontrivial interval* of rational numbers contains infinity elements.
- “Neighbouring” floats are separated by a “step size” 2^E (we’ll see).

Consequence

- **Rounding.**

Rounding of floating point numbers

- Floating point arithmetic returns the floating point number *closest* to the mathematically correct result.

Example

Mathematically,

$$1/10 = 0.1$$

But since 0.1 cannot be represented in binary floating point, we instead get the number that is closest:

$$1/10 = 0.10000000000000000555111512312578270211815834045410156250$$

(With 64-bit floats).

We will return to this, but writing algorithms that are robust to roundoff errors is a *big topic* that is outside the scope of this course.

Normalised values when $\text{Exp} \neq 0 \dots 0$ and $\text{Exp} \neq 1 \dots 1$

$$v = (-1)^S \cdot M \cdot 2^E$$

yellow!30S

red!30Exp

blue!30Frac

- Exponent encoded as *biased* value

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

- ▶ Exp: unsigned value of Exp field.
- ▶ Bias = $2^{k-1} - 1$, where k is number of Exp bits.
 - ▶ Single precision: 127 ($\text{Exp} \in [1, 254]$, $E \in [-126, 127]$).
 - ▶ Double precision: 1023 ($\text{Exp} \in [1, 2046]$, $E \in [-1022, 1023]$).

- Significand coded with implied leading 1:

$$M = 1.xxx \dots x_2$$

- ▶ $xxx \dots x$: bits of Frac field.
- ▶ Minimum when Frac = 0000 \dots 0 ($M = 1$).
- ▶ Maximum when Frac = 1111 \dots 1 ($M = 2 - \epsilon$).
- ▶ Get extra leading bit for free.

Normalised encoding example

$$v = (-1)^S \cdot M \cdot 2^E$$

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

Value: float $F = 15213.0$

$$\begin{aligned} 15213_{10} &= 11101101101101_2 \\ &= 1.1101101101101_2 \cdot 2^{13} \end{aligned}$$

Significand

$$\begin{aligned} M &= 1.1101101101101_2 \\ \text{Frac} &= 11011011011010000000000_2 \end{aligned}$$

Exponent

$$\begin{aligned} E &= 13_{10} \\ \text{Bias} &= 127_{10} \\ \text{Exp} &= E + \text{Bias} = 140_{10} = 10001100_2 \end{aligned}$$

Result

yellow!300	red!3010001100	blue!3011011011011010000000
------------	----------------	-----------------------------

Denormal values

$$v = (-1)^S \cdot M \cdot 2^E$$

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

Occur when $\text{Exp} = 000 \cdots 0_2$.

- Exponent encoded as

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

- Significand coded with implied leading 0:

$$M = 0.\text{xxx} \cdots \text{x}_2$$

- Cases

- ▶ $\text{Exp} = 000 \cdots 0_2, \text{Frac} = 000 \cdots 0_2$
 - ▶ Represents zero value.
 - ▶ Note distinct values $-0, +0$ — when might that be useful?
- ▶ $\text{Exp} = 000 \cdots 0_2, \text{Frac} \neq 000 \cdots 0_2$
 - ▶ Numbers closest to 0.0.
 - ▶ Called **subnormal numbers**.
 - ▶ Ensure that $x \neq y \Rightarrow x - y \neq 0$, i.e. avoid underflow.

Special values

Occur when $\text{Exp} = 111 \cdots 1_2$.

When $\text{Exp} = 111 \cdots 1_2, \text{Frac} = 000 \cdots 0_2$

- Represents $\pm\infty$.
- Typically the result of *overflow*.
 - ▶ Overflow can be negative!
 - ▶ *Underflow* is when the result becomes zero due to rounding.
- Both positive and negative.
- Examples:

$$\frac{1}{0} = \frac{-1}{-0} = \infty \qquad \frac{1}{-0} = -\infty$$

When $\text{Exp} = 111 \cdots 1_2, \text{Frac} \neq 000 \cdots 0_2$

- Not A Number (NaN).
- Represents case when no numeric value can be determined.
- Examples:

$$\text{sqrt}(-1) \qquad \infty - \infty \qquad \infty \cdot 0$$

The floating point number line

← very positive E very negative E → ← very negative E very positive E →

$-\infty$	-Normal	-Denorm	-0	+0	+Denorm	+Normal	$+\infty$

NaN

NaN

Note that NaNs are unordered:

- NaN is different from everything *even other NaNs!*
 - ▶ $\text{NaN} == \text{NaN}$ is false.
 - ▶ Floating-point equality is not reflexive!
- $\text{NaN} > x$ and $\text{NaN} < x$ is false for all x .

Floating point arithmetic

Biased numbers

Background: Fractional binary numbers

IEEE floating point standard

Examples and properties

Rounding, addition, and multiplication

Floating point in C

Summary

Play the game

<https://topps.diku.dk/compsys/floating-point.html>

Tiny 8-bit floating point example

yellow!30s	red!30Exp	blue!30Frac
1b	4b	3b

8-bit floating point representation

- Sign bit is the most significant bit (leftmost).
- The next four bits are Exp with a bias of 7.
- The last three bits are Frac.

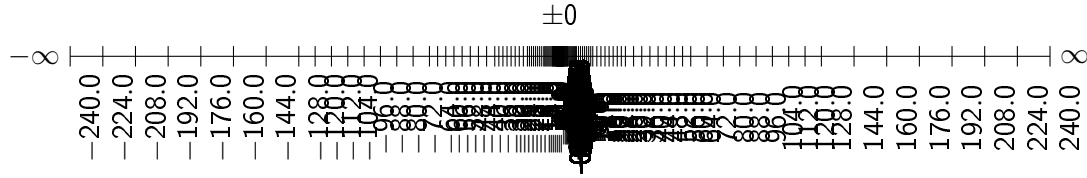
Same general form as IEEE Format

- Normalised, denormalised.
- Representation of 0, NaN, both infinities.

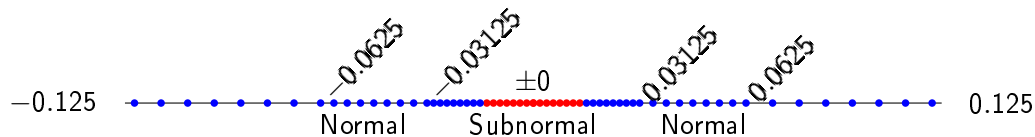
Let's look at their dynamic range.

		Sign	Exp	Frac	E	Value	
Denormalised		0	0000	000	-6	$0/8 \cdot 1/64 = 0/512$	zero
		0	0000	001	-6	$1/8 \cdot 1/64 = 1/512$	closest to zero
		0	0000	010	-6	$2/8 \cdot 1/64 = 2/512$	
		...					
yellow!30	Normalised	0	0000	111	-6	$7/8 \cdot 1/64 = 7/512$	largest denorm
		0	0001	000	-6	$8/8 \cdot 1/64 = 8/512$	smallest norm
		0	0001	001	-6	$9/8 \cdot 1/64 = 9/512$	
		...					
		0	0110	110	-1	$14/8 \cdot 1/2 = 14/16$	
		0	0110	111	-1	$15/8 \cdot 1/2 = 15/16$	closest to 1
		0	0111	000	0	$8/8 \cdot 1 = 8/8$	1
		0	0111	001	0	$9/8 \cdot 1 = 9/8$	closest to 1
		0	0111	010	0	$10/8 \cdot 1 = 10/8$	
		...					
yellow!30		0	1110	110	7	$14/8 \cdot 128 = 224$	
yellow!30		0	1110	111	7	$15/8 \cdot 128 = 240$	
red!30		0	1111	000	N/A	∞	
red!30		0	1111	001	N/A	NaN	
red!30		...				NaN	

Distribution of values



Distribution of values (zooming in)



- Note how the distribution gets denser towards zero.
- Note the big gap there would be around 0 if we did not have subnormals.
- Each of the spans with same distance between neighbors corresponds to numbers with same Exp.

yellow!30s	red!30Exp	blue!30Frac
1b	4b	3b

Useful properties of the IEEE encoding

yellow!30S	red!30Exp	blue!30Frac
------------	-----------	-------------

- **Floating-point zero same as integer zero**
 - ▶ All bits 0.
 - ▶ ...but negative zero is different.
- **Can almost compare floats with unsigned integer comparisons**
 - ▶ Must first compare sign bit.
 - ▶ Must consider $-0 = 0$.
 - ▶ NaNs problematic:
 - ▶ Greater than any other value (because $\text{Exp} = 111 \cdots 1_2$).
 - ▶ What should comparison yield?
 - ▶ Otherwise OK:
 - ▶ Normalised and denormalised compare as expected.
 - ▶ Infinities ordered properly relative to finities.

Floating point arithmetic

Biased numbers

Background: Fractional binary numbers

IEEE floating point standard

Examples and properties

Rounding, addition, and multiplication

Floating point in C

Summary

Basic idea behind floating point operations

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

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

- **Basic idea**

- ▶ First *compute exact result!*
- ▶ Then round it to fit into desired precision.
 - ▶ Overflow if exponent too large.
 - ▶ *Round to fit* into Frac.

Rounding and rounding modes

- There's more than one way to round a number, here to an integer.

	1.40	1.60	1.50	2.50	-1.50
Towards zero					

Rounding and rounding modes

- There's more than one way to round a number, here to an integer.

	1.40	1.60	1.50	2.50	-1.50
Towards zero	1	1	1	2	-1
Towards $-\infty$					

Rounding and rounding modes

- There's more than one way to round a number, here to an integer.

	1.40	1.60	1.50	2.50	-1.50
Towards zero	1	1	1	2	-1
Towards $-\infty$	1	1	1	2	-2
Towards ∞					

Rounding and rounding modes

- There's more than one way to round a number, here to an integer.

	1.40	1.60	1.50	2.50	-1.50
Towards zero	1	1	1	2	-1
Towards $-\infty$	1	1	1	2	-2
Towards ∞	2	2	2	3	-1
Nearest even					

Rounding and rounding modes

- There's more than one way to round a number, here to an integer.

	1.40	1.60	1.50	2.50	-1.50
Towards zero	1	1	1	2	-1
Towards $-\infty$	1	1	1	2	-2
Towards ∞	2	2	2	3	-1
Nearest even ∞	1	2	2	2	-2

- “Round to nearest, ties to even” is the default rounding mode.

Closer look at *nearest even*

- **Default rounding mode**

- ▶ But can be changed dynamically.
 - ▶ `https://www.gnu.org/software/libc/manual/html_node/Rounding.html`
 - ▶ Never do this.
- ▶ All others are statistically biased.
 - ▶ 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. rounding to nearest hundredth:
 - ▶ 7.8949999:

Closer look at *nearest even*

- **Default rounding mode**

- ▶ But can be changed dynamically.

- ▶ `https:`

- `//www.gnu.org/software/libc/manual/html_node/Rounding.html`

- ▶ Never do this.

- ▶ All others are statistically biased.

- ▶ 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. rounding to nearest hundredth:

- ▶ 7.8949999: 7.89

- ▶ 7.8990001:

Closer look at *nearest even*

- **Default rounding mode**

- ▶ But can be changed dynamically.
 - ▶ `https://www.gnu.org/software/libc/manual/html_node/Rounding.html`
 - ▶ Never do this.
- ▶ All others are statistically biased.
 - ▶ 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. rounding to nearest hundredth:
 - ▶ 7.8949999: 7.89
 - ▶ 7.8990001: 7.90
 - ▶ 7.8950000:

Closer look at *nearest even*

- **Default rounding mode**

- ▶ But can be changed dynamically.
 - ▶ `https://www.gnu.org/software/libc/manual/html_node/Rounding.html`
 - ▶ Never do this.
- ▶ All others are statistically biased.
 - ▶ 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. rounding to nearest hundredth:
 - ▶ 7.8949999: 7.89
 - ▶ 7.8990001: 7.90
 - ▶ 7.8950000: 7.90
 - ▶ 7.8850000:

Closer look at *nearest even*

- **Default rounding mode**

- ▶ But can be changed dynamically.

- ▶ `https:`

- `//www.gnu.org/software/libc/manual/html_node/Rounding.html`

- ▶ Never do this.

- ▶ All others are statistically biased.

- ▶ 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. rounding to nearest hundredth:

- ▶ 7.8949999: 7.89

- ▶ 7.8990001: 7.90

- ▶ 7.8950000: 7.90

- ▶ 7.8850000: 7.88

Rounding binary numbers

- **Binary fractional numbers**

- ▶ “Even” when least significant bit is 0.
- ▶ “Half way” when bits to right of rounding position are $100\cdots_2$.

- **Examples**

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

Value	Binary	Rounded	Action	Rounded value
-------	--------	---------	--------	---------------

Rounding binary numbers

- **Binary fractional numbers**

- ▶ “Even” when least significant bit is 0.
- ▶ “Half way” when bits to right of rounding position are $100\cdots_2$.

- **Examples**

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

Value	Binary	Rounded	Action	Rounded value
$2\frac{3}{32}$				

Rounding binary numbers

- **Binary fractional numbers**

- ▶ “Even” when least significant bit is 0.
- ▶ “Half way” when bits to right of rounding position are $100\cdots_2$.

- **Examples**

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

Value	Binary	Rounded	Action	Rounded value
$2\ 3/32$	10.00011_2			

Rounding binary numbers

- **Binary fractional numbers**

- ▶ “Even” when least significant bit is 0.
- ▶ “Half way” when bits to right of rounding position are $100\cdots_2$.

- **Examples**

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

Value	Binary	Rounded	Action	Rounded value
$2\ 3/32$	$10.00\textcolor{red}{011}_2$	10.00_2	($< 1/2$ —down)	2

Rounding binary numbers

- **Binary fractional numbers**

- ▶ “Even” when least significant bit is 0.
- ▶ “Half way” when bits to right of rounding position are $100\cdots_2$.

- **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
$2\frac{3}{16}$				

Rounding binary numbers

- **Binary fractional numbers**

- ▶ “Even” when least significant bit is 0.
- ▶ “Half way” when bits to right of rounding position are $100\cdots_2$.

- **Examples**

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

Value	Binary	Rounded	Action	Rounded value
$2\ 3/32$	$10.00\textcolor{red}{011}_2$	10.00_2	($< 1/2$ —down)	2
$2\ 3/16$	$10.00\textcolor{red}{110}_2$			

Rounding binary numbers

- **Binary fractional numbers**

- ▶ “Even” when least significant bit is 0.
- ▶ “Half way” when bits to right of rounding position are $100\cdots_2$.

- **Examples**

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

Value	Binary	Rounded	Action	Rounded value
$2\ 3/32$	$10.00\textcolor{red}{011}_2$	10.00_2	($< 1/2$ —down)	2
$2\ 3/16$	$10.00\textcolor{red}{110}_2$	10.01_2	($> 1/2$ —up)	$2\ 1/4$

Rounding binary numbers

- **Binary fractional numbers**

- ▶ “Even” when least significant bit is 0.
- ▶ “Half way” when bits to right of rounding position are $100\cdots_2$.

- **Examples**

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

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

Rounding binary numbers

- **Binary fractional numbers**

- ▶ “Even” when least significant bit is 0.
- ▶ “Half way” when bits to right of rounding position are $100\cdots_2$.

- **Examples**

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

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

Rounding binary numbers

- **Binary fractional numbers**

- ▶ “Even” when least significant bit is 0.
- ▶ “Half way” when bits to right of rounding position are $100\cdots_2$.

- **Examples**

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

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

Rounding binary numbers

- **Binary fractional numbers**

- ▶ “Even” when least significant bit is 0.
- ▶ “Half way” when bits to right of rounding position are $100\cdots_2$.

- **Examples**

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

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

Rounding binary numbers

- **Binary fractional numbers**

- ▶ “Even” when least significant bit is 0.
- ▶ “Half way” when bits to right of rounding position are $100\cdots_2$.

- **Examples**

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

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

Rounding binary numbers

- **Binary fractional numbers**

- ▶ “Even” when least significant bit is 0.
- ▶ “Half way” when bits to right of rounding position are $100\cdots_2$.

- **Examples**

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

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

Floating point multiplication (assuming operands are numbers)

$$((-1)^{S_3} \cdot M_3 \cdot 2^{E_3}) = ((-1)^{S_1} \cdot M_1 \cdot 2^{E_1}) \cdot ((-1)^{S_2} \cdot M_2 \cdot 2^{E_2})$$

- **Exact result**

$$S_3 = S_1 \oplus S_2$$

$$M_3 = M_1 \cdot M_2$$

$$E_3 = E_1 + E_2$$

where \oplus is exclusive-or.

- **Fixing**

- ▶ If $M_3 \geq 2$, shift M_3 right and increment E_e .
- ▶ If E_3 out of range, overflow to ∞ .
- ▶ Round M_3 to fit Frac precision.

- **Implementation**

- ▶ Biggest chore is multiplying significands.

Floating point addition (assuming operands are numbers)

$$((-1)^{S_3} \cdot M_3 \cdot 2^{E_3}) = ((-1)^{S_1} \cdot M_1 \cdot 2^{E_1}) + ((-1)^{S_2} \cdot M_2 \cdot 2^{E_2})$$

■ Approach

- ▶ Assume without loss of generality that $E_1 \geq E_2$.
- ▶ Rewrite smaller number such that its exponent matches E_1 :

$$((-1)^{S_3} \cdot M_3 \cdot 2^{E_3}) = ((-1)^{S_1} \cdot M_1 \cdot 2^{E_1}) + ((-1)^{S_2} \cdot M'_2 \cdot 2^{E_1})$$

■ Exact result

- ▶ Sign S_3 , significant M_3 :
 - ▶ Result of signed addition.

■ Fixing

- ▶ If $M_3 \geq 2$, shift M_3 right and increment E_3 .
- ▶ If $M_3 < 1$, shift M left k positions and decrement E_3 by k .
- ▶ If E_3 out of range, overflow to ∞ .
- ▶ Round M to fit Frac precision.

$$\begin{array}{r} \leftarrow E_1 - E_2 \rightarrow \\ \boxed{-1^{S_1} \cdot M_1} \\ + \quad \boxed{-1^{S_2} \cdot M_2} \\ \hline \boxed{-1^{S_3} \cdot M_3} \end{array}$$

Example of floating-point addition with a 2-bit significand

$$\begin{aligned} & (-1.01 \cdot 2^2) + (1.1 \cdot 2^4) \\ = & (-1.01 \cdot 2^2) + (110.0 \cdot 2^2) && \text{Align exponents} \\ = & (-1.01 + 110.0) \cdot 2^2 && \text{Distributivity} \\ = & 100.11 \cdot 2^2 && \text{Add significands} \\ = & 1.0011 \cdot 2^4 && \text{Normalise} \\ = & 1.01 \cdot 2^4 && \text{Perform rounding} \end{aligned}$$

Algebraic properties of floating-point addition

- **Compared to those of Abelian Group**
 - ▶ Closed under addition?

Algebraic properties of floating-point addition

- **Compared to those of Abelian Group**

- ▶ Closed under addition? **Yes**
 - ▶ But may generate infinity or NaN.
- ▶ Commutative?

Algebraic properties of floating-point addition

- **Compared to those of Abelian Group**

- ▶ Closed under addition? **Yes**
 - ▶ But may generate infinity or NaN.
- ▶ Commutative? **Yes**
- ▶ Associative?

Algebraic properties of floating-point addition

- Compared to those of Abelian Group

- ▶ Closed under addition? **Yes**
 - ▶ But may generate infinity or NaN.
- ▶ Commutative? **Yes**
- ▶ Associative? **No**
 - ▶ Due to overflow and inexactness of rounding.
 - ▶ $(3.14 + 1e10) - 1e10 = 0$
 - ▶ $3.14 + (1e10 - 1e10) = 3.14$
- ▶ 0 is additive identity?

Algebraic properties of floating-point addition

- **Compared to those of Abelian Group**

- ▶ Closed under addition? **Yes**
 - ▶ But may generate infinity or NaN.
- ▶ Commutative? **Yes**
- ▶ Associative? **No**
 - ▶ Due to overflow and inexactness of rounding.
 - ▶ $(3.14 + 1e10) - 1e10 = 0$
 - ▶ $3.14 + (1e10 - 1e10) = 3.14$
- ▶ 0 is additive identity? **Yes**
- ▶ Does every element have an additive inverse?

Algebraic properties of floating-point addition

■ Compared to those of Abelian Group

- ▶ Closed under addition? **Yes**
 - ▶ But may generate infinity or NaN.
- ▶ Commutative? **Yes**
- ▶ Associative? **No**
 - ▶ Due to overflow and inexactness of rounding.
 - ▶ $(3.14 + 1e10) - 1e10 = 0$
 - ▶ $3.14 + (1e10 - 1e10) = 3.14$
- ▶ 0 is additive identity? **Yes**
- ▶ Does every element have an additive inverse? **Almost**
 - ▶ Infinities and NaN do not have inverses.

■ Monotonicity

- ▶ $a \geq b \Rightarrow a + c \geq b + c$

Algebraic properties of floating-point addition

■ Compared to those of Abelian Group

- ▶ Closed under addition? **Yes**
 - ▶ But may generate infinity or NaN.
- ▶ Commutative? **Yes**
- ▶ Associative? **No**
 - ▶ Due to overflow and inexactness of rounding.
 - ▶ $(3.14 + 1e10) - 1e10 = 0$
 - ▶ $3.14 + (1e10 - 1e10) = 3.14$
- ▶ 0 is additive identity? **Yes**
- ▶ Does every element have an additive inverse? **Almost**
 - ▶ Infinities and NaN do not have inverses.

■ Monotonicity

- ▶ $a \geq b \Rightarrow a + c \geq b + c$? **Almost**
 - ▶ Infinities and NaNs are the exception.

Algebraic properties of floating-point multiplication

- Compared to those of a commutative ring
 - ▶ Closed under multiplication?

Algebraic properties of floating-point multiplication

- Compared to those of a commutative ring
 - ▶ Closed under multiplication? **Yes**
 - ▶ But may generate infinity or NaN.
 - ▶ Commutative?

Algebraic properties of floating-point multiplication

- Compared to those of a commutative ring
 - ▶ Closed under multiplication? **Yes**
 - ▶ But may generate infinity or NaN.
 - ▶ Commutative? **Yes**
 - ▶ Associative?

Algebraic properties of floating-point multiplication

- Compared to those of a commutative ring

- ▶ Closed under multiplication? **Yes**
 - ▶ But may generate infinity or NaN.
- ▶ Commutative? **Yes**
- ▶ Associative? **No**
 - ▶ Due to overflow and inexactness of rounding.
 - ▶ $(1e20 * 1e20) * 1e-20 = \infty$
 - ▶ $1e20 * (1e20 * 1e-20) = 1e20$
- ▶ 1 is multiplicative identity?

Algebraic properties of floating-point multiplication

- Compared to those of a commutative ring

- ▶ Closed under multiplication? **Yes**
 - ▶ But may generate infinity or NaN.
- ▶ Commutative? **Yes**
- ▶ Associative? **No**
 - ▶ Due to overflow and inexactness of rounding.
 - ▶ $(1e20 * 1e20) * 1e-20 = \infty$
 - ▶ $1e20 * (1e20 * 1e-20) = 1e20$
- ▶ 1 is multiplicative identity? **Yes**
- ▶ Multiplication distributes over addition?

Algebraic properties of floating-point multiplication

- Compared to those of a commutative ring

- ▶ Closed under multiplication? **Yes**
 - ▶ But may generate infinity or NaN.
- ▶ Commutative? **Yes**
- ▶ Associative? **No**
 - ▶ Due to overflow and inexactness of rounding.
 - ▶ $(1e20 * 1e20) * 1e-20 = \infty$
 - ▶ $1e20 * (1e20 * 1e-20) = 1e20$
- ▶ 1 is multiplicative identity? **Yes**
- ▶ Multiplication distributes over addition? **No**
 - ▶ Overflow and rounding again.
 - ▶ $1e20 * (1e20 - 1e20) = 0.0$
 - ▶ $1e20 * 1e20 - 1e20 * 1e20 = \text{NaN}$

Floating point arithmetic

Biased numbers

Background: Fractional binary numbers

IEEE floating point standard

Examples and properties

Rounding, addition, and multiplication

Floating point in C

Summary

Floating point in C

- **C guarantees two types**

- ▶ `float`: 32-bit single precision.
- ▶ `double`: 64-bit single precision.

- **Conversions/casting**

- ▶ Casting between `int`, `float`, and `double` changes bit representation.
- ▶ `double/float to int`
 - ▶ Truncates fractional part.
 - ▶ Like rounding toward zero.
 - ▶ Not defined when out of range or NaN: generally sets to `SMin`.
- ▶ `int to double`
 - ▶ Exact conversion as long as `int` fits in 53 bits.
- ▶ `int to float`
 - ▶ Will round according to rounding mode.

Floating point is exciting!



First “flight” of the Ariane 5 in 1996.

Floating point is exciting!



First “flight” of the Ariane 5 in 1996.

- A double storing horizontal velocity of the rocket was converted to a 16-bit signed integer.
- The number was larger than 32767 so the conversion failed, causing an exception, crashing the guidance module.

Floating point puzzles

For each of the following C expressions, either

- Argue that it is true for all argument values.
- Explain why it's not.

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

Assume neither `d` nor `t` is NaN.

Assume `int` is 32 bits.

Floating point puzzles

For each of the following C expressions, either

- Argue that it is true for all argument values.
- Explain why it's not.

■ `x == (int) (float) x`

```
int    x = ...;
```

```
float  f = ...;
```

```
double d = ...;
```

Assume neither `d` nor `t` is NaN.

Assume `int` is 32 bits.

Floating point puzzles

For each of the following C expressions, either

- Argue that it is true for all argument values.
- Explain why it's not.

- `x == (int) (float) x`
- `x == (int) (double) x`

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

Assume neither `d` nor `t` is NaN.

Assume `int` is 32 bits.

Floating point puzzles

For each of the following C expressions, either

- Argue that it is true for all argument values.
- Explain why it's not.

- `x == (int) (float) x`
- `x == (int) (double) x`
- `f == (float) (double) f`

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

Assume neither `d` nor `t` is NaN.

Assume `int` is 32 bits.

Floating point puzzles

For each of the following C expressions, either

- Argue that it is true for all argument values.
- Explain why it's not.

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

Assume neither `d` nor `t` is NaN.

Assume `int` is 32 bits.

- `x == (int) (float) x`
- `x == (int) (double) x`
- `f == (float) (double) f`
- `d == (double) (float) d`

Floating point puzzles

For each of the following C expressions, either

- Argue that it is true for all argument values.
- Explain why it's not.

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

Assume neither `d` nor `t` is NaN.

Assume `int` is 32 bits.

- `x == (int) (float) x`
- `x == (int) (double) x`
- `f == (float) (double) f`
- `d == (double) (float) d`
- `f == -(-f)`

Floating point puzzles

For each of the following C expressions, either

- Argue that it is true for all argument values.
- Explain why it's not.

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

Assume neither `d` nor `t` is NaN.

Assume `int` is 32 bits.

- `x == (int) (float) x`
- `x == (int) (double) x`
- `f == (float) (double) f`
- `d == (double) (float) d`
- `f == -(-f)`
- `2/3 == 2/3.0`

Floating point puzzles

For each of the following C expressions, either

- Argue that it is true for all argument values.
- Explain why it's not.

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

Assume neither `d` nor `t` is NaN.

Assume `int` is 32 bits.

- `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`

Floating point puzzles

For each of the following C expressions, either

- Argue that it is true for all argument values.
- Explain why it's not.

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

Assume neither d nor t is NaN.

Assume int is 32 bits.

- `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`

Floating point puzzles

For each of the following C expressions, either

- Argue that it is true for all argument values.
- Explain why it's not.

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

Assume neither d nor t is NaN.

Assume int is 32 bits.

- `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`

Floating point puzzles

For each of the following C expressions, either

- Argue that it is true for all argument values.
- Explain why it's not.

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

Assume neither d nor t is NaN.

Assume int is 32 bits.

- `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`

Floating point arithmetic

- Biased numbers

- Background: Fractional binary numbers

- IEEE floating point standard

- Examples and properties

- Rounding, addition, and multiplication

- Floating point in C

Summary

Summary

- **IEEE floating point has clear properties.**
 - ▶ But they may not match your intuition.
- **Represents numbers of the form $M \cdot 2^E$.**
- One can reason about operations independent of implementation.
 - ▶ Computed with perfect precision and then rounded.
 - ▶ But rounded after every “primitive” operation (e.g. addition, multiplication).
- **Not the same as \mathbb{Q}/\mathbb{R} arithmetic.**
 - ▶ Violates associativity and distributivity, mostly due to rounding.
 - ▶ Sometimes makes life difficult for heavy-duty numerical programming.
 - ▶ But carefully designed such that “naive” use mostly does what one expects.

Also try this tool: <https://evanw.github.io/float-toy/>
And read this: <https://moyix.blogspot.com/2022/09/someones-been-messing-with-my-subnormals.html>