Floating point numbers

Troels Henriksen

29-11-2021

Agenda

Preliminaries: biased numbers

Floating point arithmetic
    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

$$B2T(X) = -x_{w-1} \cdot 2^{w-1} + \sum_{i=0}^{w-2} x_i \cdot 2^i$$

Biased

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

- Typically

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

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

	B2U	B2I
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

$$B2T(X) = -x_{w-1} \cdot 2^{w-1} + \sum_{i=0}^{w-2} x_i \cdot 2^i$$

Biased

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

- Typically

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

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

	B2U	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

$$B2T(X) = -x_{w-1} \cdot 2^{w-1} + \sum_{i=0}^{w-2} x_i \cdot 2^i$$

Biased

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

- Typically

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

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

	B2U	B2I
00000000_2	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

$$B2T(X) = -x_{w-1} \cdot 2^{w-1} + \sum_{i=0}^{w-2} x_i \cdot 2^i$$

Biased

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

- Typically

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

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

	B2U	B2I
00000000_2	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

$$B2T(X) = -x_{w-1} \cdot 2^{w-1} + \sum_{i=0}^{w-2} x_i \cdot 2^i$$

Biased

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

- Typically

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

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

	B2U	B2I
00000000_2	0_{10}	-127_{10}
01111111_2	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

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

Two's complement

$$B2T(X) = -x_{w-1} \cdot 2^{w-1} + \sum_{i=0}^{w-2} x_i \cdot 2^i$$

Biased

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

- Typically

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

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

	B2U	B2I
00000000_2	0_{10}	-127_{10}
01111111_2	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

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

Two's complement

$$B2T(X) = -x_{w-1} \cdot 2^{w-1} + \sum_{i=0}^{w-2} x_i \cdot 2^i$$

Biased

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

- Typically

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

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

	B2U	B2I
00000000_2	0_{10}	-127_{10}
01111111_2	127_{10}	0_{10}
11111111_2	255_{10}	128_{10}

Preliminaries: biased numbers

Floating point arithmetic
    Background: Fractional binary numbers
    IEEE floating point standard
    Examples and properties
    Rounding, addition, and multiplication
    Floating point in C

Summary

We have seen that

$$10010101_2$$

is basically interpreted like

$$149_{10}$$

in particular "structure" is the same, just with 2 instead of 10.

We have seen that

$$10010101_2$$

is basically interpreted like

$$149_{10}$$

in particular "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}	$\cdots$	4	2	1	1/2	1/4	1/8	$\cdots$	2^{-n}
Digits	b_{m-1}	b_{m-2}	$\cdots$	b_2	b_1	b_0	b_{-1}	b_{-2}	b_{-3}	$\cdots$	b_{-n}

Representation

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

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

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 $0.111\ldots$ are just below 1.0.
  - $1/2 + 1/4 + 1/8 + \cdots 1/2^n + \cdots \sim 1.0$.
  - Use notation $1.0 - \epsilon$.

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!

Preliminaries: biased numbers

Floating point arithmetic
  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.

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.
  - ▶ **Beware the wrath of Kahan!**
  - ▶ http://people.eecs.berkeley.edu/~wkahan/

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).
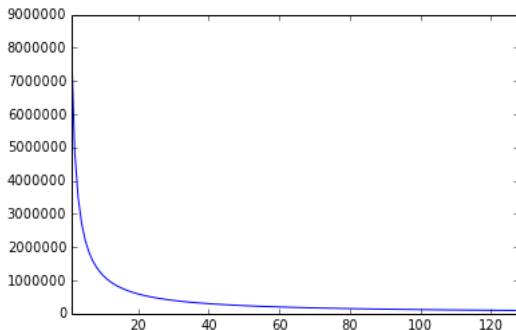- Frac field encodes M (but is not equal to M).

S	Exp	Frac
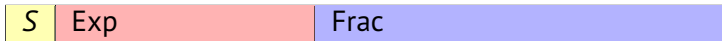
The point is floating

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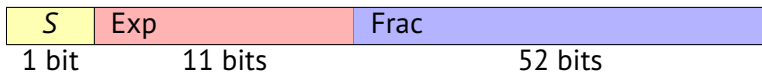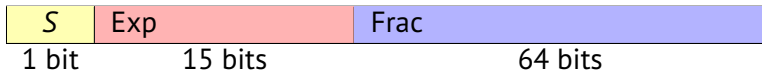- Symmetric around 0.

Density of floats

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

32-bit single precision: `float`

S	Exp	Frac
1 bit	8 bits	23 bits
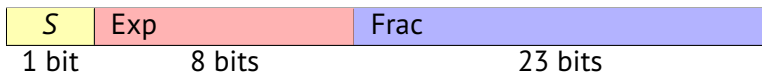
64-bit double precision: `double`

S	Exp	Frac
1 bit	11 bits	52 bits

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

S	Exp	Frac
1 bit	15 bits	64 bits

Normalised values when Exp $\neq 0 \cdots 0$ and Exp $\neq 1 \cdots 1$

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

S	Exp	Frac
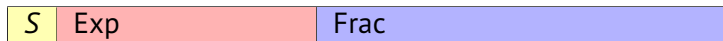
- **Exponent encoded as *biased* value**

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

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

- **Significand coded with implied leading 1:**

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

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

Normalised encoding example

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

Value: float F = 15213.0

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

Significand

$$M = 1.1101101101101_2$$
$$\text{Frac} = \quad 1101101101101 0000000000_2$$

Exponent

$$E = 13_{10}$$
$$\text{Bias} = 127_{10}$$
$$\text{Exp} = 140_{10} = 10001100_2$$

Result | 0 | 10001100 | 11011011011010000000000

Denormal values

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

Occur when Exp $= 000 \cdots 0_2$.

- **Exponent encoded as**

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

- **Significand coded with implied leading 0:**

$$M = 0.xxx \cdots x_2$$

- **Cases**
    - ▶ $\text{Exp} = 000 \cdots 0_2, \text{Frac} = 000 \cdots 0_2$
        - ▶ Represents zero value.
        - ▶ Note distinct values $-0, +0$ — why do you think that is?
    - ▶ $\text{Exp} = 000 \cdots 0_2, \text{Frac} \neq 000 \cdots 0_2$
        - ▶ Numbers closest to 0.0.
        - ▶ Called **Subnormal numbers.**
        - ▶ Ensure that $x \neq y \Rightarrow x - y \neq 0$, i.e. avoid overflow.

Special values

Occur when Exp $= 111\cdots1_2$.

When Exp $= 111\cdots1_2$, Frac $= 000\cdots0_2$
- 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 Exp $= 111\cdots1_2$, Frac $\neq 000\cdots0_2$
- Not A Number (NaN).
- Represents case when no numeric value can be determined.
- Examples:

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

The floating point number line

$\leftarrow$ very negative E   very positive E $\rightarrow$

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

| NaN | | NaN |

Note that NaNs are unordered:

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

Preliminaries: biased numbers

Floating point arithmetic
Examples and properties

Summary

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

Tiny 8-bit floating point example

s	Exp	Frac
1b	4b	3b
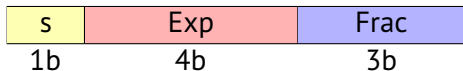
8-bit floating point representation

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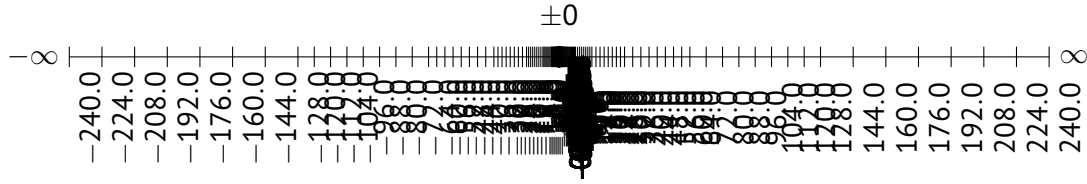- The last three bits are Frac.

Same general form as IEEE Format

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

Dynamic range of positive numbers

|	Sign	Exp	Frac	Exp	Value	
Denormalised	0	0000	000	-6	0	
	0	0000	001	-6	$1/8 \cdot 1/64 = 1/512$	closest to zero
	0	0000	010	-6	$2/8 \cdot 1/64 = 2/512$	
	...					
	0	0000	111	-6	$7/8 \cdot 1/64 = 7/512$	largest denorm
Normalised	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	$8/8 \cdot 1 = 1$	
	0	0111	001	0	$9/8 \cdot 9/8 = 1$	Closest to 1
	0	0111	010	0	$10/8 \cdot 10/8 = 1$	
	...					
	0	1110	110	7	$14/8 \cdot 128 = 224$	
	0	1110	111	7	$15/8 \cdot 128 = 240$	
	0	1111	000	N/A	∞	

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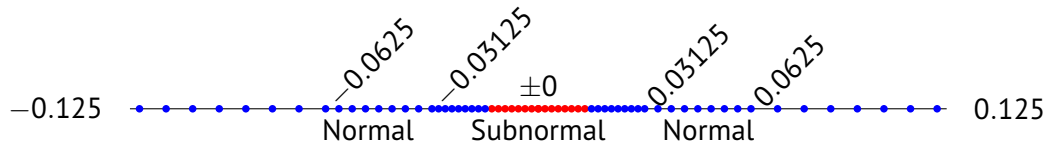
- Note how the distribution gets denser towards zero.
- Note the big gap there would be around 0 if we did not have subnormals.
- Each of the spans with same distance between neighbors corresponds to numbers with same Exp.

s	Exp	Frac
1b	4b	3b

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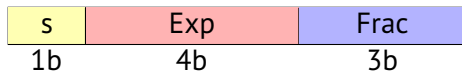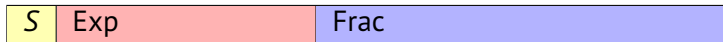
- **Floating-point zero same as integer zero**
    - ▶ All bits 0.
    - ▶ ...but negative zero is different.
- **Can almost compare floats with unsigned integer comparisons**
    - ▶ Must first compare sign bit.
    - ▶ Must consider $-0 = 0$.
    - ▶ NaNs problematic:
        - ▶ Greater than any other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

Preliminaries: biased numbers

Floating point arithmetic
Rounding, addition, and multiplication

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.

- 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					

- 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 $-\infty$					

- 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 ∞					

- 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					

- 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	3	-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

- **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

- **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				

- **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			

- **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	10.00_2	$(<1/2$ – down)	2

- **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	10.00_2	($< 1/2$ – down)	2
2 3/16				

- **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	10.00_2	($< 1/2$–down)	2
2 3/16	10.00110_2			

- **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	10.00_2	($< 1/2$–down)	2
2 3/16	10.00110_2	10.01_2	($> 1/2$–up)	2 1/4

- **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	10.00_2	($< 1/2$ – down)	2
2 3/16	10.00110_2	10.01_2	($> 1/2$ – up)	2 1/4
2 7/8				

- **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	10.00_2	$(<1/2-\text{down})$	2
2 3/16	10.00110_2	10.01_2	$(>1/2-\text{up})$	2 1/4
2 7/8	10.11100_2			

- **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	10.00_2	($< 1/2$ – down)	2
2 3/16	10.00110_2	10.01_2	($> 1/2$ – up)	2 1/4
2 7/8	10.11100_2	11.00_2	($1/2$ – up)	3

- **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	10.00_2	$(<1/2\text{--down})$	2
2 3/16	10.00110_2	10.01_2	$(>1/2\text{--up})$	2 1/4
2 7/8	10.11100_2	11.00_2	$(1/2\text{--up})$	3
2 5/8				

- **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	10.00_2	($< 1/2$–down)	2
2 3/16	10.00110_2	10.01_2	($> 1/2$–up)	2 1/4
2 7/8	10.11100_2	11.00_2	($1/2$–up)	3
2 5/8	10.10100_2			

- **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	10.00_2	($< 1/2$–down)	2
2 3/16	10.00110_2	10.01_2	($> 1/2$–up)	2 1/4
2 7/8	10.11100_2	11.00_2	($1/2$–up)	3
2 5/8	10.10100_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.

$\leftarrow E_1 - E_2 \rightarrow$

$-1^{S_1} \cdot M_1$

$+ \qquad -1^{S_2} \cdot M_2$

$-1^{S_3} \cdot M_3$

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

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

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

- **Compared to those of Abelian Group**
  - ▶ Closed under addition? **Yes**
    - ▶ But may generate infinity or NaN.
  - ▶ Commutative?

- **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.

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

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

- **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?

- **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=∞
    - ▶ 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=∞
    - ▶ 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 = NaN
- **Monotonicity**
  - ▶ $a \geq b \Rightarrow a + c \geq b + c$?

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

Preliminaries: biased numbers

Floating point arithmetic

Summary

- **C guarantees two types**
  - ▶ float: 32-bit single precision.
  - ▶ double: 64-bit single precision.
- **Conversions/casting**
  - ▶ Casting between int, float, and double changes bit represension.
  - ▶ double/float to int
    - ▶ Truncates fractional part.
    - ▶ Like rounding toward zero.
    - ▶ Not defined when out of range or NaN: generally sets to TMin.
  - ▶ int to double
    - ▶ Exact conversion as long as int fits in 53 bits.
    - ▶ We can represent integers as floats with $E = 0$ and then just put the integer in *Frac*.
  - ▶ int to float
    - ▶ Will round according to rounding mode.

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.

```
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

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)

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 ⇒ (d*2) < 0.0
- d > f ⇒ -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 ⇒ (d*2) < 0.0
- d > f ⇒ -f > -d
- d * d >= 0.0
- (d+f)-d == f

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.