Floating-point numbers

Troels Henriksen

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

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*, raw bits are interpreted as unsigned, then a *bias* is subtracted.

Unsigned

$$\text{Bits2N}(X) = \sum_{i=0}^{w-1} x_i \cdot 2^i$$

Two's complement

$$\text{TC2Int}(X) = -x_{w-1} \cdot 2^{w-1} + \sum_{i=0}^{w-2} x_i \cdot 2^i$$

Biased

$$\text{B2Int}(X) = \text{Bits2N}(x) - b$$

- Typically $b = 2^{w-1} - 1$
- **Examples for** $w = 8, b = 127$

	B2U	B2I
$\langle 00000000 \rangle$		

Biased number representation

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

Unsigned

$$\mathrm{Bits2N}(X) = \sum_{i=0}^{w-1} x_i \cdot 2^i$$

Two's complement

$$\mathrm{TC2Int}(X) = -x_{w-1} \cdot 2^{w-1} + \sum_{i=0}^{w-2} x_i \cdot 2^i$$

Biased

$$\mathrm{B2Int}(X) = \mathrm{Bits2N}(x) - b$$

- Typically $b = 2^{w-1} - 1$
- **Examples for** $w = 8, b = 127$

	B2U	B2I
$\langle 00000000 \rangle$	0_{10}	

Biased number representation

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

Unsigned

$$\mathrm{Bits2N}(X) = \sum_{i=0}^{w-1} x_i \cdot 2^i$$

Two's complement

$$\mathrm{TC2Int}(X) = -x_{w-1} \cdot 2^{w-1} + \sum_{i=0}^{w-2} x_i \cdot 2^i$$

Biased

$$\mathrm{B2Int}(X) = \mathrm{Bits2N}(x) - b$$

- Typically $b = 2^{w-1} - 1$
- **Examples for** $w = 8, b = 127$

	B2U	B2I
$\langle 00000000 \rangle$	0_{10}	-127_{10}
$\langle 01111111 \rangle$		

Biased number representation

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

Unsigned

$$\mathrm{Bits2N}(X) = \sum_{i=0}^{w-1} x_i \cdot 2^i$$

Two's complement

$$\mathrm{TC2Int}(X) = -x_{w-1} \cdot 2^{w-1} + \sum_{i=0}^{w-2} x_i \cdot 2^i$$

Biased

$$\mathrm{B2Int}(X) = \mathrm{Bits2N}(x) - b$$

- Typically $b = 2^{w-1} - 1$
- **Examples for** $w = 8, b = 127$

	B2U	B2I
$\langle 00000000 \rangle$	0_{10}	-127_{10}
$\langle 01111111 \rangle$	127_{10}	

Biased number representation

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

Unsigned

$$\text{Bits2N}(X) = \sum_{i=0}^{w-1} x_i \cdot 2^i$$

Two's complement

$$\text{TC2Int}(X) = -x_{w-1} \cdot 2^{w-1} + \sum_{i=0}^{w-2} x_i \cdot 2^i$$

Biased

$$\text{B2Int}(X) = \text{Bits2N}(x) - b$$

- Typically $b = 2^{w-1} - 1$
- **Examples for** $w = 8, b = 127$

	B2U	B2I
$\langle 00000000 \rangle$	0_{10}	-127_{10}
$\langle 01111111 \rangle$	127_{10}	0_{10}
$\langle 11111111 \rangle$		

Biased number representation

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

Unsigned

$$\text{Bits2N}(X) = \sum_{i=0}^{w-1} x_i \cdot 2^i$$

Two's complement

$$\text{TC2Int}(X) = -x_{w-1} \cdot 2^{w-1} + \sum_{i=0}^{w-2} x_i \cdot 2^i$$

Biased

$$\text{B2Int}(X) = \text{Bits2N}(x) - b$$

- Typically $b = 2^{w-1} - 1$
- **Examples for** $w = 8, b = 127$

	B2U	B2I
$\langle 00000000 \rangle$	0_{10}	-127_{10}
$\langle 01111111 \rangle$	127_{10}	0_{10}
$\langle 11111111 \rangle$	255_{10}	

Biased number representation

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

Unsigned

$$\text{Bits2N}(X) = \sum_{i=0}^{w-1} x_i \cdot 2^i$$

Two's complement

$$\text{TC2Int}(X) = -x_{w-1} \cdot 2^{w-1} + \sum_{i=0}^{w-2} x_i \cdot 2^i$$

Biased

$$\text{B2Int}(X) = \text{Bits2N}(x) - b$$

- Typically $b = 2^{w-1} - 1$
- **Examples for** $w = 8, b = 127$

	B2U	B2I
$\langle 00000000 \rangle$	0_{10}	-127_{10}
$\langle 01111111 \rangle$	127_{10}	0_{10}
$\langle 11111111 \rangle$	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

Integral binary numbers

We have seen that

$$10010101_2$$

is basically interpreted like

$$149_{10}$$

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

Integral binary numbers

We have seen that

$$10010101_2$$

is basically interpreted like

$$149_{10}$$

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

$$\underbrace{b_{m-1}\cdots b_0}_{\text{integral part}} . \underbrace{b_{-1}\cdots b_{-n}}_{\text{fraction part}} = \sum_{i=-n}^{m-1} b_i \cdot 2^i$$

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 $1.0 - \epsilon$.

Representable numbers

Limitation #1

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

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

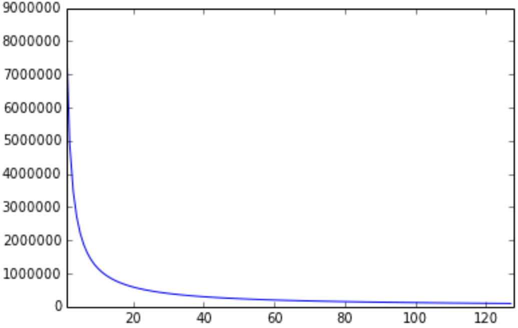
- Most significant bit is sign bit.
- E field encodes e (but is not equal to e).
- T field encodes m (but is not equal to m).

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`

S	*E*	*T*
1 bit	8 bits	23 bits

64-bit double precision: `double`

S	*E*	*T*
1 bit	11 bits	52 bits

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

S	*E*	*T*
1 bit	15 bits	64 bits

Normalised values when $E \neq \langle 0 \cdots 0 \rangle$ and $E \neq \langle 1 \cdots 1 \rangle$

$$\boxed{v = (-1)^s \cdot m \cdot 2^e}$$

S	E	T

- **Exponent encoded as *biased* value**

$$e = \mathrm{Bits2N}(E) - b$$

  - $b = e_{\mathrm{max}}$.
    - Single precision: 127 ($E \in [-126, 127]$).
    - Double precision: 1023 ($E \in [-1022, 1023]$).

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

$$m = 1.xxx \cdots x_2 = 1 + \mathrm{Bits2N}(\langle T \rangle) \cdot 2^{1-p}$$

  - $xxx \cdots x$: bits of T.
  - Get extra leading bit for free.
  - Precision
    - Single precision: $p = 24$.
    - Double precision: $p = 53$.
  - Minimum value when $T = \langle 0000 \cdots 0 \rangle$ ($m = 1$).
  - Maximum value when $T = \langle 1111 \cdots 1 \rangle$ ($m = 2 - \epsilon$).

Normalised encoding example

$$\boxed{v = (-1)^S \cdot m \cdot 2^e} \qquad \boxed{e = \mathrm{Bits2N}(E) - b}$$

Value: `float F = 15213.0`

$$\begin{aligned} 15213_{10} &= 11101101101101_2 \cdot 2^0 \\ &= 1.1101101101101_2 \cdot 2^{13} \end{aligned}$$

Significand

$$\begin{aligned} m &= 1.1101101101101_2 \\ \mathrm{T} &= \langle\ 11011011011010000000000 \rangle \end{aligned}$$

Exponent

$$\begin{aligned} e &= 13_{10} \\ b &= 127_{10} \\ E &= N2Bits(e+b) = \langle 10001100 \rangle \end{aligned}$$

Result			
	0	10001100	11011011011010000000000

Denormal values

$$\boxed{v = (-1)^s \cdot m \cdot 2^e} \quad \boxed{e = 1 - b}$$

Occur when $E = \langle 000 \cdots 0 \rangle$.

- **Exponent encoded as**

$$e = 1 - b$$

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

$$m = 0.xxx \cdots x = \mathrm{Bits2N}(\langle T \rangle) \cdot 2^{1-p}$$

- **Cases**
  - $E = \langle 000 \cdots 0 \rangle, T = \langle 000 \cdots 0 \rangle$
    - Represents zero value.
    - Note distinct values $-0, +0$ – why do you think that is?
  - $E = \langle 000 \cdots 0 \rangle, T \neq \langle 000 \cdots 0 \rangle$
    - 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 $E = \langle 111 \cdots 1 \rangle$.

When $E = \langle 111 \cdots 1 \rangle, T = \langle 000 \cdots 0 \rangle$

- 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:

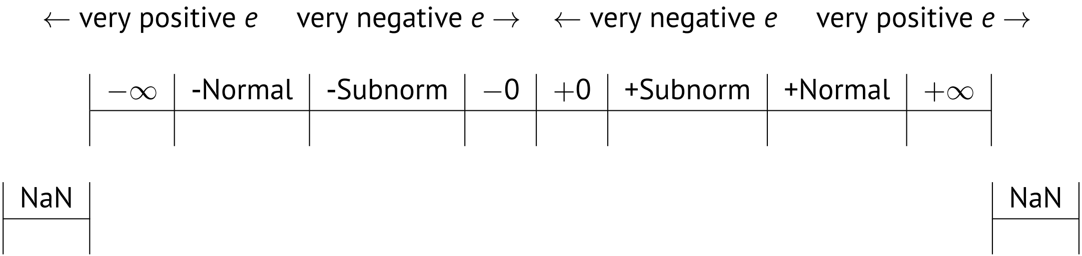
$$\frac{1}{0} = \frac{-1}{-0} = \infty \qquad \frac{1}{-0} = -\infty$$

When $E = \langle 111 \cdots 1 \rangle, T \neq \langle 000 \cdots 0 \rangle$

- 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



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
    - 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

S	E	T
1b	4b	3b

8-bit floating-point representation

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

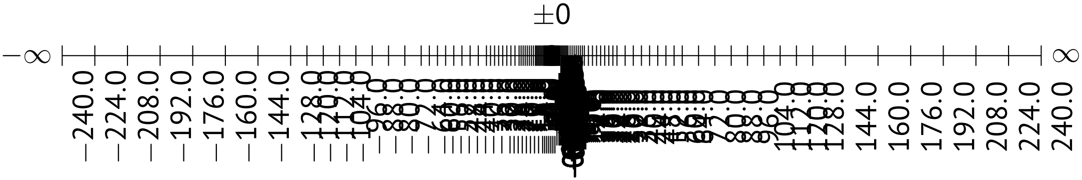
Same general form as IEEE Format

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

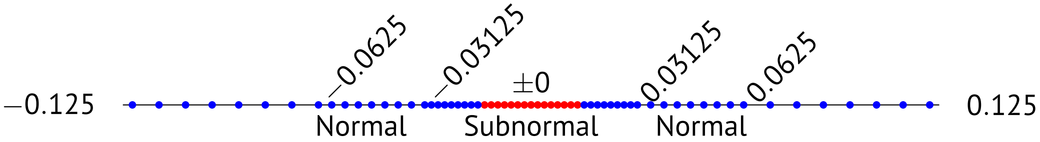
Dynamic range of positive numbers

	S	E	T	e	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



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 *E*.

s	*E*	*T*
1b	4b	3b

Useful properties of the IEEE encoding

- **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 $E = \langle 111 \cdots 1 \rangle$).
    - What should comparison yield?
  - Otherwise OK:
    - Normalised and denormalised compare as expected.
    - Infinities ordered properly relative to finities.

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

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 T.

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 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	10.00011_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 3/32	10.00011_2	10.00_2	$(< 1/2-\text{down})$	2
2 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	10.00011_2	10.00_2	$(< 1/2\text{–down})$	2
2 3/16	10.00110_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	10.00011_2	10.00_2	$(< 1/2 -$down)	2
2 3/16	10.00110_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	10.00011_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				

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	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			

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	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

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	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				

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	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			

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	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**

$$\begin{aligned} s_3 &= s_1 \oplus s_2 \\ m_3 &= m_1 \cdot m_2 \\ e_3 &= e_1 + e_2 \end{aligned}$$

  where $\oplus$ is exclusive-or.
- **Fixing**
  - If $m_3 \geq 2$, shift m_3 right and increment e_3.
  - If e_3 out of range, overflow to ∞.
  - Round m_3 to fit T 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, significand 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_3 to p bits.

$$\leftarrow e_1 - e_2 \rightarrow$$

$$\boxed{-1^{s_1} \cdot m_1}$$

$$+ \quad \boxed{-1^{s_2} \cdot m_2}$$

$$\boxed{-1^{s_3} \cdot m_3}$$

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

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

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=∞`
    - `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=`∞
    - `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`

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

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 represensation.
  - `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.
  - `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.

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

Assume neither `d` nor `t` is NaN.
Assume `int` is 32 bits.

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

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)`

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` ⇒ `(d*2) < 0.0`
- `d > f` ⇒ `-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`

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

Summary

- **IEEE floating-point has clear properties.**
  - But they may not match your intuition.
- **Represents numbers of the form $(-1)^s \cdot m \cdot 2^e$.**
- 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`