COMP36212 Mathematical Systems and Computation 2020/21

Week 2: Floating-Point Arithmetic

Mantas Mikaitis, Department of Mathematics

Email: mantas.mikaitis@manchester.ac.uk

Goals

- In the previous week, fixed-point arithmetic—a way to represent real numbers with integers—was presented.

- Our aim this week is to learn about *floating-point* arithmetic—a different (better) approach to represent a subset of real numbers on computers.

- The main advantages of floating-point over fixed-point arithmetic are:

  - ❑ Wider range of representable values (underflow or overflow in FP computations is much less common).

  - ❑ *Relative* accuracy of representation rather than absolute—this is achieved by widening the gaps between the numbers as they advance from 0 towards $\pm\infty$.

  - ❑ Well-defined behaviour by the IEEE 754 standards.

  - ❑ Error analysis of FP arithmetic is more widely used and more of it was developed over the years.

- The main disadvantage: it is more expensive to implement.

Literature

The following material, and the references therein, can be used for further reading.

- Chapters 1, 2, and 27 of the *Accuracy and Stability of Numerical Algorithms* by N. J. Higham, 2nd edition, 2002 (ASNA02).

- Chapters 1–3, 7, and Appendix B of the *Handbook of Floating-Point Arithmetic* by J.-M. Muller and others, 2nd edition, 2018 (HFPA18).

- Chapter 8 of the *Digital Arithmetic* by M. Ercegovac and T. Lang, 2004 (DA04).

- 754-2019 - IEEE Standard for Floating-Point Arithmetic, published by IEEE, available on IEEE Xplore, 2019.

Contents

- General definitions and properties of floating-point arithmetic.

- IEEE 754 standardized floating-point arithmetic.

- Non-standard floating-point data types.

- Floating-point arithmetic algorithms: addition/subtraction, multiplication.

- Methods for error analysis.

Some remarks

- We refer to floating point with an abbreviation "FP".

- We refer to FP operations with $\circ\ (x\ \mathbf{op}\ b)$ with $\circ \in \{\mathrm{RN}, \mathrm{RZ}, \mathrm{RU}, \mathrm{RD}\}$ and $\mathbf{op} \in \{+, -, \times, /, \sqrt{}\}$.

- We refer to exact operations with $x\ \mathbf{op}\ b$.

- We use the *italic* font to emphasize the important terms.

- We use the `typewriter` font for algorithms and code.

- For the multiplication we use $\star$ in the algorithms and $\times$ in the equations.

Floating-point representation

A set of FP numbers $\mathcal{F} \subset \mathbb{R}$ is a subset of real numbers partially characterized by

- ❏ a *radix or base* $\beta \geq 2$ (nowadays $\beta = 2$),
- ❏ a *precision* $p \geq 2$, and
- ❏ extremal *exponents* e_{min} and e_{max}.

Usually $e_{min} < e_{max}$ and $e_{min} = 1 - e_{max}$.

Elements of $\mathcal{F}$ have the form $m{\times}\beta^{e-p+1}$, with $0 \leq |m| \leq \beta^p - 1$ (an integer *significand* or *mantissa*), and $e_{min} \leq e \leq e_{max}$ (an integer *exponent*).

Full specification also includes special values and binary encodings of the exponent and significand, which we will see later.

Floating-point representation

- For example, consider a small unsigned FP system with $\beta = 2, p = 3,$ $e_{min} = -2, e_{max} = 3$.
- Some example numbers in this system:
  - ❑ $m = 2^2 = 4, e = 0$. This represents $m \times 2^{e-3+1} = 2^2 \times 2^{-2} = 1$.
  - ❑ $m = 2^1 = 2, e = 1$. This represents $2^1 \times 2^{-1} = 1$.
  - ❑ $m = 2^0 = 1, e = 2$. This represents $2^0 \times 2^0 = 1$.
  - ❑ $m = 2^2 + 2^1 = 6, e = 0$. This represents $(2^2 + 2^1) \times 2^{-2} = 1.5$.
  - ❑ $m = 2^1 + 2^0 = 3, e = 1$. This represents $(2^1 + 2^0) \times 2^{-1} = 1.5$.
- There is a problem—same numbers can be represented in multiple ways!

Floating-point representation

- Elements of a certain $\mathcal{F}$ may be expressed in more than one combination of e and m.

- Usually it is desired to *normalize* FP systems so that each number has no more than one representation.

- This is achieved by making sure that $\beta^{p-1} \leq m \leq \beta^p - 1$.

- Any number with m satisfying this is said to be a *normal* number.

- Otherwise, when $m < \beta^{p-1}$, necessarily $e = e_{min}$ and the number is said to be *denormal* or *subnormal*.

- With these constraints $\mathcal{F}$ is normalized, and every FP number $x \in \mathcal{F}$ has a single unique representation.

- Usually the first digit of m is implicit, and not stored, since it does not change for normal values.

Floating-point representation

- Back to the examples shown before in the small FP system, now we can cross out all the representations where $m < 2^{p-1}$, unless $e \neq -2$ (none here):
  - ❑ $m = 2^2 = 4$, $e = 0$. This represents $m \times 2^{e-3+1} = 2^2 \times 2^{-2} = 1$.
  - ❑ ~~$m = 2^1 = 2$, $e = 1$. This represents $2^1 \times 2^{-1} = 1$.~~
  - ❑ ~~$m = 2^0 = 1$, $e = 2$. This represents $2^0 \times 2^0 = 1$.~~
  - ❑ $m = 2^2 + 2^1 = 6$, $e = 0$. This represents $(2^2 + 2^1) \times 2^{-2} = 1.5$.
  - ❑ ~~$m = 2^1 + 2^0 = 3$, $e = 1$. This represents $(2^1 + 2^0) \times 2^{-1} = 1.5$.~~

Floating-point representation

- Some notable floating-point numbers in $\mathcal{F}$:
  - ❑ Smallest positive FP number (subnormal) is $\beta^{e_{min}} \times \beta^{1-p} = \beta^{1+e_{min}-p}$.
  - ❑ Largest positive FP number is $\beta^{e_{max}} \times (\beta - \beta^{1-p})$.
  - ❑ Smallest positive normal FP number is $\beta^{e_{min}}$.

- When some computed FP number is smaller than the smallest subnormal number, the result is said to cause an *underflow* which is usually flagged in the processors.

- However, note that *rounding* (see later) can "bump up" the values that underflow to the smallest representable number, thus causing no signalled underflow (due to this underflow requires a precise definition—before or after rounding).

- Similarly, overflow occurs when some computed value becomes larger than the largest FP number after rounding.

Floating-point representation

Special FP values are

❑ zero, most commonly represented with $m = 0$ and $e = e_{min}$ (usually both positive and negative zeros are available),

❑ infinities ($\pm\infty$), for representing values that overflow. Infinities are usually represented with with $e = e_{max} + 1$ and $m = 0$, and

❑ NaNs (not-a-number), for operations that do not produce meaningful values (such as $\sqrt{-2}$). These are usually represented with $e = e_{max} + 1$ and $m \neq 0$.

Small floating-point system

- Again consider a small unsigned FP system with $\beta = 2$, $p = 3$, $e_{min} = -2$, $e_{max} = 3$.

- We now bridge the mathematical definition and the bit-level representation of this binary FP system (necessary to implement a practical arithmetic on a computer).

- This will require us to choose the layout of binary FP data and the binary encoding of various parts, as well as, later, implement arithmetic operations.

- The data is represented, in an IEEE 754 fashion (see later), with the exponent bits (3 bits) followed by $p - 1 = 2$ significand bits (with one bit implicit and not stored).

- In a signed FP system, an extra sign bit would be added before the exponent.

Small floating-point system

- The layout of the data is as follows.
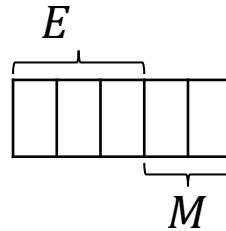
- Here E is a binary exponent and M is a binary significand.
- The encoding of e is denoted as $e = E - e_{max}$ (*biased exponent*).
- The encoding of m is denoted as $m = 2^{p-1} + M$ ($m = M$ for subnormals).
- Examples:
  - ❑ Binary value 01100_2 represents $m \times \beta^{e-p+1} = 2^2 \times 2^{-2} = 1$.
  - ❑ Binary value 01000_2 represents $2^2 \times 2^{-3} = 0.5$.
  - ❑ Binary value 01010_2 represents $(2^2 + 2^1) \times 2^{-3} = 0.75$.

Small floating-point system

- The value 00000_2 is reserved for representing zero.
- The values 00001_2 to 00011_2 are reserved for subnormal numbers with $e = e_{min} = -2$.
- The values 00100_2 to 11011_2 represent normalized numbers as per definition.
- The value 11100_2 represents $+\infty$.
- The values 11101_2 to 11111_2 represent `NaNs`.

Small floating-point system

The list of values representable in our small FP system. Subnormals in blue.

Binary value	FP value	Binary value	FP value	Binary value	FP value	Binary value	FP value
00000	0	01000	0.5	10000	2	11000	8
00001	0.0625	01001	0.625	10001	2.5	11001	10
00010	0.125	01010	0.75	10010	3	11010	12
00011	0.1875	01011	0.875	10011	3.5	11011	14
00100	0.25	01100	1	10100	4	11100	$+\infty$
00101	0.3125	01101	1.25	10101	5	11101	NaN
00110	0.375	01110	1.5	10110	6	11110	NaN
00111	0.4375	01111	1.75	10111	7	11111	NaN

Small floating-point system

0 0.5 1 1.5 2 2.5 3 3.5 4 5 6 7 8 10 12 14
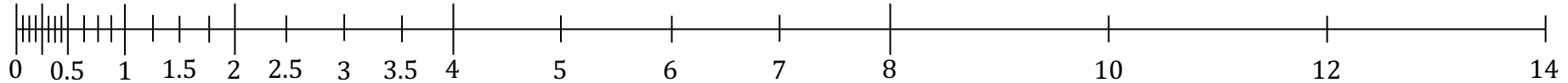
- The subset of FP numbers is marked on a real number axis above.
- The gaps between numbers increase $2\times$ on every power of 2, except 0.25.
- The gaps between subnormals and the first and second power-of-2 FP numbers are equivalent to $2^{e_{min}}\times 2^{1-p} = 0.0625$.
- In general, the gap in front of any binary FP number $x = 2^{e-p+1}\times m$ is equal to $2^e\times 2^{1-p}$.
- This is commonly called a *unit of least precision (ulp)*[*] and written as a function $ulp(x)$ (beware of different definitions of the ulp in literature).
- Relative errors are sometimes measured in $\mathbf{ulps}$ to hide the information about the sizes of gaps—e.g. above, 4 is $8\mathbf{ulps}$ (of different sizes) away from 1.
- The value 2^{1-p} is commonly called a *machine epsilon* of an FP arithmetic, and denoted as ε.
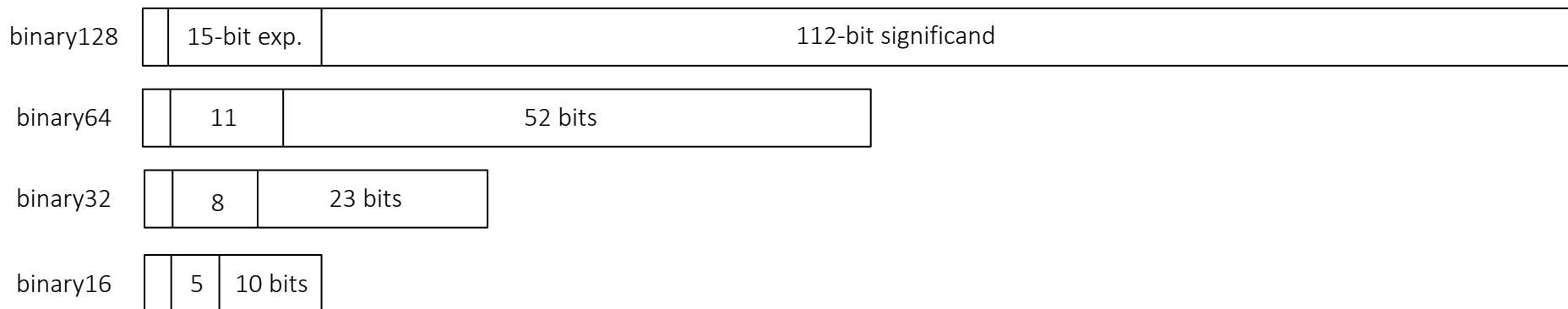
IEEE 754 FP arithmetic standard

- IEEE 754 is a standard for FP arithmetic.

- First version in 1985, followed by revisions in 2008 and 2019.

- It specifies, among other things,
  - ❑ four binary FP formats, three for computation and one for data interchange;
  - ❑ decimal FP formats (we will only focus on binary formats here);
  - ❑ add, subtract, multiply, divide, square root, compare, and various other operations on FP numbers that are *required* for an arithmetic to be IEEE 754 compliant;
  - ❑ floating-point exceptions and their handling, particularly including NaN FP values which are categorized into quiet NaNs and signalled NaNs;
  - ❑ rounding modes of FP arithmetics;
  - ❑ a concept of *correctly rounded* FP data; and
  - ❑ *Recommended* functions, such as **exp**, **log**, **pow**, **tan**, **cos** and others.

IEEE 754 fused multiply-add

- IEEE 754 defines an operation that comprises two basic arithmetic operations in one: $\circ (x{\times}y + z)$ where all the inputs as well as the result are FP numbers—this is called a *fused multiply-add* (FMA).

- Notice that there is only a single rounding, therefore this is not the same as performing the multiplication $t = \circ (x{\times}y)$ followed by addition $\circ (t + z)$.

- An FMA is better in terms of accuracy and sometimes can result in speeding up the code.

- Compilers usually take advantage of the FMA instruction wherever possible.

- However, see Sec. 2.6 in ASNA02 for some tricky situations that the use of FMA can present.

IEEE 754 FP arithmetic standard

| binary128 | | 15-bit exp. | 112-bit significand |

| binary64 | | 11 | 52 bits |

| binary32 | | 8 | 23 bits |

| binary16 | | 5 | 10 bits |

- The formats above are all signed, having a sign bit at the front.
- The values including sign are expressed with $(-1)^s \times m \times 2^{e-p+1}$, where s is a sign bit with $s = 1$ if the FP number is negative and $s = 0$ otherwise.
- Binary16 is defined only for memory operations, not computation, however, various hardware devices can perform computation.
- Binary128 is rare in hardware.
- Most hardware these days is equipped with binary64 and binary32 arithmetics, with various operations (in C language `double` and `float` respectively).
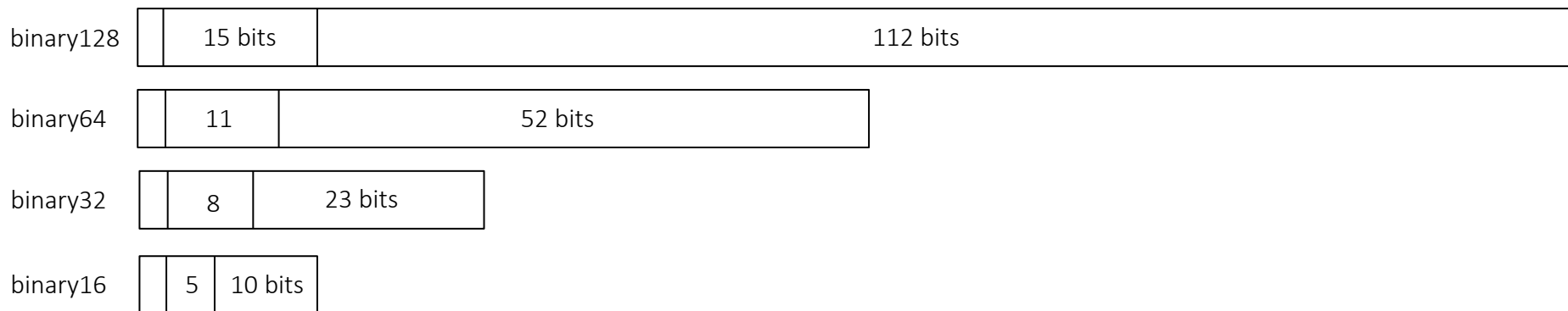
Intel's extended precision

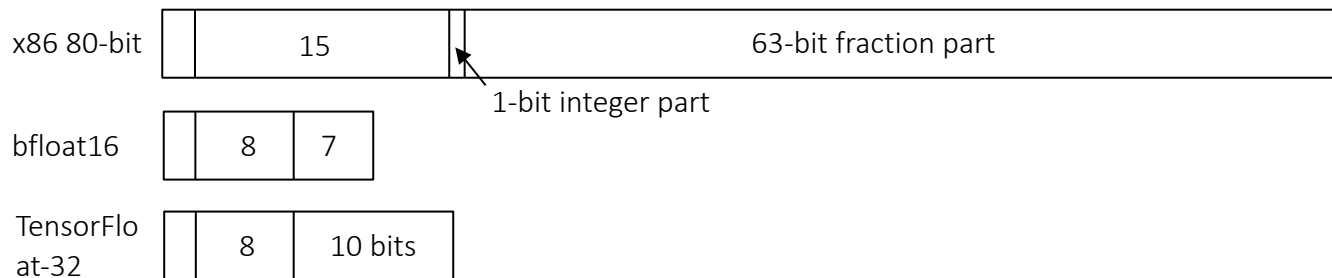| x86 80-bit		15		63-bit fraction part

1-bit integer part
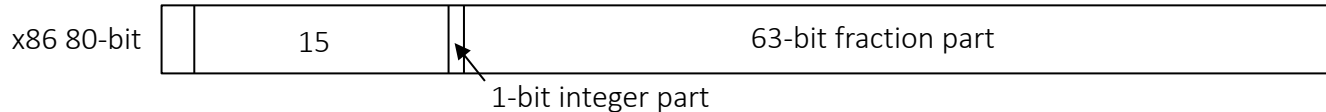
- Intel's x86 instruction sets provide an 80-bit FP data type, starting with the Intel 8087 math coprocessor (~1980).

- It is based on the extended precision specification of the IEEE 754 standard.

- This format is encoded similarly to binary32/64 formats, except there is no implicit bit and bit '1' for normalized numbers is stored before the significand.

- Floating-point registers are 80 bits wide to accommodate this format, so operations with any lower precision formats usually also maintain 80 bits while data stays in registers, unless specified on compilation not to do that (speed penalty added due to extra rounding after each operation).

List of all common FP datatypes

IEEE 754 datatypes

binary128 | 15 bits | 112 bits

binary64 | 11 | 52 bits

binary32 | 8 | 23 bits

binary16 | 5 | 10 bits

Other FP datatypes that appear in various Intel, ARM, and NVIDIA devices

x86 80-bit | 15 | 63-bit fraction part

1-bit integer part

bfloat16 | 8 | 7

TensorFloat-32 | 8 | 10 bits

List of FP arithmetics in hardware

- The table below provides a list of main FP arithmetic types and various values.
- Here f_{min} and s_{min} are minimum positive normal and subnormal FP values resp.

	binary16	bfloat16	TensorFloat-32	binary32	binary64
p	11	8	11	24	53
$e_{\max}$	15	127	127	127	1023
$e_{\min}$	-14	-126	-126	-126	-1022
ε	2^{-10}	2^{-7}	2^{-10}	2^{-23}	2^{-52}
$f_{\min}$	2^{-14}	2^{-126}	2^{-126}	2^{-126}	2^{-1022}
$s_{\min}$	2^{-24}	2^{-133}	2^{-136}	2^{-149}	2^{-1074}

Optional exercise: Derive all the values in rows 3-5 of the table using the general FP arithmetic definitions in previous slides.

FP arithmetic in C

- Example code in `fp_arithmetic_intro.c` shows some usage of binary32 (`float`) and binary64 (`double`) data types. Output below.

```
1.75 in float is represented as:
3fe00000
0 01111111 11000000000000000000000

2^(-55) in float is represented as:
24000000
0 01001000 00000000000000000000000

Positive infinity in float is represented as:
7f800000
0 11111111 00000000000000000000000

1.75 in double is represented as:
3ffc000000000000
0 01111111111 1100000000000000000000000000000000000000000000000000

2^(-55) in double is represented as:
3c80000000000000
0 01111001000 0000000000000000000000000000000000000000000000000000

Positive infinity in double is represented as:
7ff0000000000000
0 11111111111 0000000000000000000000000000000000000000000000000000
```

- Some values held in `float` and `double` data types, including infinity.
- We printed out the encoding of those values in hexadecimal and binary.
- The binary printout is split into sign, exponent and significand with a space between.
- Remember that there is an implicit extra bit in the significand that is not shown.

Optional exercise

Determine on paper or calculator, using the definitions above, what binary32 FP numbers are represented by the following binary patterns.

- 0 10000000 01000000000000000000000
- 1 10001111 00000000000000000000000
- 0 11111111 10000000011111100000011
- 1 11111111 00000000000000000000000
- 1 10000010 11100100000000000000000
- 0 10010011 00000000000000001000000
- 1 01111111 00000000000000000000000

Rounding

- Arithmetic operations or functions that take FP inputs and produce FP outputs can internally produce outputs in higher precision.

- An operation that maps some value into a target FP arithmetic is called *rounding.*

- Rounding is performed as part of every FP operation as well as in conversion between different data types.

- We denote rounding with $\circ\,(x\ \mathrm{op}\ b)$ where $\mathrm{op} \in \{+, -, \times, /, \sqrt{}\}$, or $\circ\,(x)$ for rounding in conversion, and $\circ \in \{\mathrm{RN}, \mathrm{RZ}, \mathrm{RU}, \mathrm{RD}\}$ (see next slide).

- In fixed-point arithmetics, after rounding, usually *saturation* is performed, which returns a maximum representable value if x overflows.

- In FP arithmetic, overflows (including due to rounding) automatically return infinity—this is suitable in most cases.
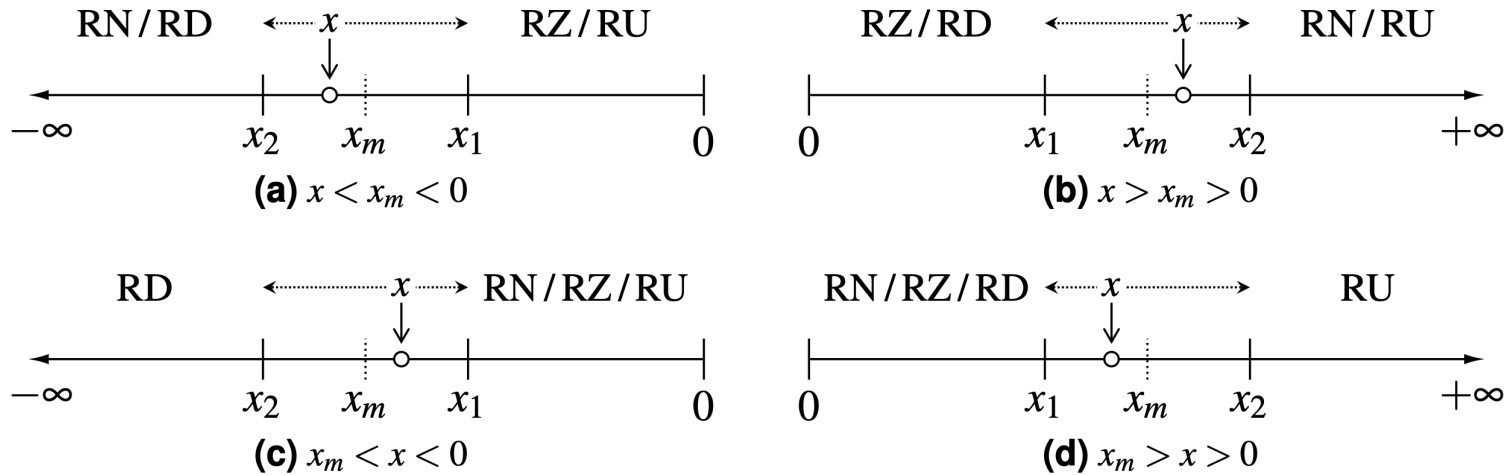
Rounding

IEEE 754-2019 specifies the following rounding modes for binary FP arithmetic.

- ❑ Round toward $-\infty$: the rounding function $\mathrm{RD}(x)$ returns a maximum FP value, in the target precision, that is not greater than x.
- ❑ Round toward $+\infty$: $\mathrm{RU}(x)$ returns the smallest FP value that is not less than x.
- ❑ Round toward zero: $\mathrm{RZ}(x)$ returns the closest FP value that is not greater in magnitude than x.
- ❑ Round to nearest, ties to even: $\mathrm{RN}(x)$ returns the closest FP value, and if there are two that are equally close, the one that has an even significand is returned (*least significant bit* of the significand is zero). This is usually a default mode in most hardware.

Consider three consecutive values in some precision-p FP arithmetic: 1, $1 + \varepsilon$, and $1 + 2\varepsilon$. The first and the third have even significands, while the second odd. Then, $\mathrm{RN}(1 + \varepsilon/2) = 1$, whereas $\mathrm{RN}(1 + \varepsilon + \varepsilon/2) = 1 + 2\varepsilon$.

Rounding

(a) $x < x_m < 0$

(b) $x > x_m > 0$

(c) $x_m < x < 0$

(d) $x_m > x > 0$

- Here various rounding cases are shown.
- The value x is rounded to either of the two neighbouring FP values x_1, x_2.
- The value x_m is the middle point, which is not an FP value here.

Rounding

There are two terms commonly appearing when talking about rounding.

- *Correct rounding*—when some FP operation or function, for all possible inputs, produces values as if the function was computed in infinite precision and unbounded range and then rounded, it is said to be a *correctly rounded* FP operation for a given rounding mode.

- *Faithful rounding*—common misuse of *faithful arithmetic,* which says that either of the two FP values that surround the exact result are returned by FP operations, not necessarily conforming with the definition of any rounding mode.

- Any arithmetic that provides the correctly rounded results is faithful.

Table Maker's dilemma

- Correct rounding for an arithmetic is a much more strict requirement than it being a faithful arithmetic.

- IEEE 754 requires only addition/subtraction, multiplication, division, square root, fused multiply-add, and conversion operations correctly rounded.

- More complex functions, such as $\exp$, $\log$, $\sin$, $\cos$ and others, are *recommended* by the IEEE 754, not required, since they are expensive to implement, especially with correct rounding.

- The problem comes down to the fact that, in general, outputs of elementary functions can produce outputs that are non-terminating decimals which might require infinitely many digits for deciding the correct rounding.

- The term is said to be coined by William M. Kahan, a Turing Award winner and one of the main people behind the floating-point standardization.

Some properties of rounding

- If some number x is an FP number in the target precision, then $\circ(x) = x$.

- Rounding is a *monotonic* or a *nonincreasing* mapping of real numbers to FP numbers—this means that if we take any two real numbers x, y such that $x \leq y$, then we have that $\circ(x) \leq \circ(y)$, and similarly for when $x \geq y$.

- RN is an *unbiased* rounding scheme, whereas RZ, RD, RU are biased (*directed rounding* methods).

- RU cannot round to $-\infty$, while RD cannot round to $+\infty$.

- RN and RZ are symmetric with regards to zero.

- $\text{RU}(x) = -\text{RD}(-x)$ and $\text{RD}(x) = -\text{RU}(-x)$.

- In case of a tie when, for example, x is halfway between the largest FP value and infinity, RN rounds to infinity.

FP arithmetic operations

- FP numbers cannot be manipulated with integer arithmetic units such as adders and multipliers, as can be done for fixed-point numbers.

- Algorithms for manipulating FP numbers can be implemented by using the integer arithmetic units with some additional shifting and and bit-masking.

- If an FP arithmetic needs to be computed in hardware, for speed purposes, then a separate *Floating-Point Unit (FPU)* is most commonly designed in processors with a dedicated register bank and instructions.

- Next we look at algorithms for adding and multiplying FP numbers in order to get a better sense of what is involved in designing an FP arithmetic library.

- Basic principles are similar for both software and hardware implementations, but some low-level details differ (those interested refer to HFPA18).

FP arithmetic operations: Add

For simplicity, we work with two positive normalized precision-p FP numbers $x = 2^{e_x} \times m_x$ and $y = 2^{e_y} \times m_y$—in case of signed data some sign manipulation would have to be added at various steps (see the literature).

1. If $e_y > e_x$ swap x and y. We then have $e_x \geq e_y$.

2. Perform binary shift right of m_y by $e_x - e_y$ steps to compute $2^{-(e_x - e_y)} \times m_y$. This step is called a *significand alignment* step.

3. Since the significands were aligned, we set $e_y = e_x$.

4. Addition now comes down to adding the significands: $2^{e_x} \times m_x + 2^{e_x} \times 2^{-(e_x - e_y)} \times m_y = 2^{e_x} \times (m_x + 2^{-(e_x - e_y)} \times m_y)$. Since the significands are binary integers, this addition is a binary integer addition of $p + 1$ bits (extra bit is required for a possible carry out).

FP arithmetic operations: Add

5. We now have the sum $2^{e_r} \times m_r$ with the significand m_r stored in $p + 1 + 3$ bits, and $e_r = e_x$. The extra 3 bits come from the significand alignment step—when shifting m_y right, instead of throwing out the bits that fall off, we keep the last 2 ones and form a third bit by an OR of the rest that fall off in shifting. Those extra bits later are used for rounding.

6. Recall that precision-p significands of normalized FP numbers always have 1 as the most significant bit. In m_r we have an extra bit at the front which could be 1 (in the case of carry out in step 4), and if that is the case, we shift m_r right by one bit to satisfy the definition; otherwise we remove the most significant bit (which is zero) and keep $p + 3$ bits of m_r. This is called a *normalization* step.

7. Perform rounding of m_r to p bits. If rounding causes a carry out, do the normalization one more time.

FP arithmetic operations: Add

8. Finally, if $e_r > e_{max}$, overflow occurs. Otherwise the final answer
   ◦ $(x + y) = 2^{e_r} \times m_r$ is returned.

A few things worth noting.

- We do not handle subnormal inputs here. In that case, some preprocessing of the inputs would have to be done (some more details in the literature).

- Notice how the whole algorithm is performed using basic operations: comparison, binary shifting, binary integer addition. This demonstrates that FP arithmetic, at the core, is fixed-point arithmetic with some extra bit manipulations around the integer operations.

- During the normalization, the three extra bits for rounding have to be recomputed.

FP arithmetic ops.: multiply

For multiplication we also simplify and take two positive normalized precision-p FP numbers $x = 2^{e_x} \times m_x$ and $y = 2^{e_y} \times m_y$. Multiplication of those numbers shows that exponents need to be added while the significands multiplied: $2^{e_x} \times m_x \times 2^{e_y} \times m_y = 2^{e_x + e_y} \times m_x \times m_y$.

1. Multiply the integer significands: $m_r = m_x \times m_y$. This requires $2p$ bits to store.

2. Add the exponents (here we assume the exponent bias of e_{max} as in IEEE 754): $e_r = e_x + e_y = E_x - e_{max} + E_y - e_{max}$. Since the binary encoding of the exponent does not include the bias, $E_r = E_x + E_y - e_{max}$.

3. Since $1 \leq m_x, m_y < 2$, we have $1 \leq m_r < 4$, and thus we might need to shift m_r right by one step and increase the exponent by one (normalize).

FP arithmetic ops.: multiply

4. Perform rounding of m_r to fit it into the precision-p (after the normalization it is stored $2p - 1$ bits and we need the top p bits for the result).

5. If no exceptions occur (such as overflow or underflow) then we have the final answer $\circ (x \times y) = 2^{e_r} \times m_r$.

A few things worth noting.

- We do not handle any exceptions here: for example, the exponent can become higher than the maximum exponent after steps 2 or 3, which would mean overflow. Similarly, after step 2 we might find that the resultant exponent is smaller than e_{min} which would cause an underflow.

- See HFPA18 and DA04 for further details.

Measuring accuracy

- We talked about *precision*, but how about *accuracy*?

- *Precision* usually refers to the number of bits in a numerical format or the number of bits in the fractional part of the numerical format.

- *Accuracy* is a measure of how well an algorithm utilizes a given numerical precision to represent some output when compared to some ideal algorithm using the same or higher precision data type.

- In other words, how well does it set up the bits in a data type to minimize the numerical error?

- While the basic arithmetic operations in some IEEE 754-compliant arithmetic have the maximum possible accuracy due to correct rounding, it might not always be the case.

- We look into how to measure the accuracy of FP results.

Measuring accuracy

- To measure accuracy of FP arithmetic operations or functions, we go back to the concept of the *unit of least precision (ulp)*.

- First of all, we need to pick some reference for making the comparison when measuring accuracy.

- For example, if we want to develop an approximation to e^x in binary32 arithmetic, say $e_s^{x_s}$, we should have an approximation of it in some higher precision arithmetic ($e_r^{x_r}$).

- We then do (possibly for all possible inputs):

  1. Evaluate the exponential in higher precision ($y_r = e_r^{x_r}$) and in binary32 ($y_s = e_s^{x_s}$), by ensuring that $x_s = x_r$ to avoid introducing errors in the input.

  2. Convert y_r to binary32 ($\hat{y}_r$) and compute $\frac{1}{2}\mathrm{ulp}(\hat{y}_r) = h.$

Measuring accuracy

3. Then, if $y_r - h \leq y_s \leq y_r + h$, we declare that the accuracy of our binary32 exponential is within $0.5\mathrm{ulp}$ (a closest possible binary32 answer is always returned).

4. Otherwise, $h = 2h$ and $h = h + \mathrm{ulp}(\hat{y}_r)$ thereafter, and each time we repeat step 3, and similarly declare that the accuracy is within $1\mathrm{ulp}$, $2\mathrm{ulp}$, $3\mathrm{ulp}$, … once we find that y_s lies within the given range.

There is a complication that the sizes of ulp change around the powers of 2 (see the axis of numbers in the small FP system shown previously)—not shown here for simplicity, but this has to be taken into account if $\hat{y}_r$ is a power of 2 or if, in step 3, one of the bounds crosses a power of 2.

Measuring accuracy
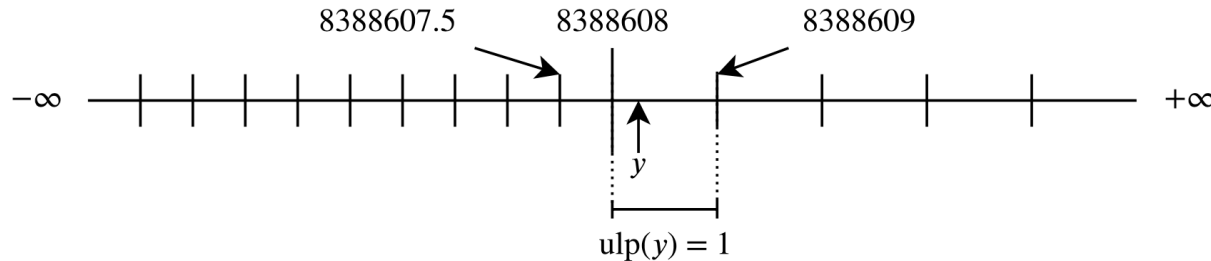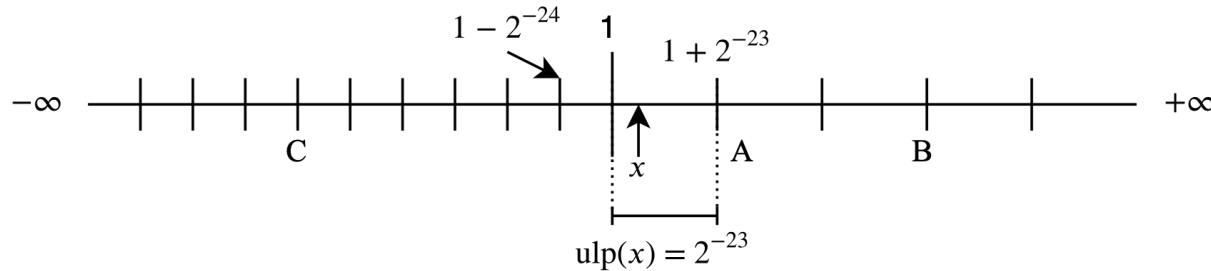
- A few interesting examples using $\mathbf{ulp}$ in binary32. If an approximation to x evaluates to 1, then the accuracy is $0.5\mathbf{ulp}$, if A it's $1\mathbf{ulp}$, if B it's $3\mathbf{ulps}$, and if C it's $7\mathbf{ulps}$.
- The size of $\mathbf{ulp}$ around y is quite big, it is 1, but if, for example, 8388608 is an approximation to y, then we are still accurate to $0.5\mathbf{ulp}.$

Measuring accuracy

A few things worth noting.

- Note that all calculations in this type of error analysis have to be done in higher precision than the target (target precision was in this case binary32).

- You will notice that the resultant error is an *error bound* rather than an actual numerical error. This is usually sufficient and is much easier to disseminate—if we claim that our developed library is returning answers within $0.5\mathrm{ulp}$ that is sufficient to inform the readers that it is correctly rounded without plotting all the values of errors.

- Measuring in ulps is most common in computer arithmetic literature and some software documentation, for example see GNU GCC: https://www.gnu.org/software/libc/manual/html_node/Errors-in-Math-Functions.html.

Rounding error analysis

- If we know the error bounds of various basic operations, such as IEEE 754 arithmetic, then we can perform rounding error analysis of algorithms that utilize them to analyse error accumulation and other effects.

- Rounding error analysis allows us to derive *worst case error bounds*, which evaluates the numerical behaviour of the algorithms in general rather than specific cases.

- This is in contrast with running an algorithm many times with a set of data points and observing the errors—usually we cannot cover the whole input space in order to detect the worst case errors.

- Rounding error analysis is done on paper, and, if we can make assumptions about the accuracy of basic operations, does not require a computer.

- Usually the bounds are tested by a few *numerical experiments*.

Rounding error analysis

For FP arithmetic, one *standard model* is as follows. We assume IEEE 754 arithmetic with round-to-nearest, and denote rounding to floating-point with $\mathrm{fl}(x)$.

Given FP numbers $x, y \in \mathcal{F}$,

$$\mathrm{fl}(x \text{ op } y) = (x \text{ op } y)(1 + \delta), \qquad |\delta| \leq u, \qquad \text{op} \in \{+, -, \times, /\}.$$

Here $u = 2^{-p}$, for a precision-p FP arithmetic, is called the *unit roundoff*.

Square root and the FMA can be expressed similarly. Analysis using this model is out of scope here—those interested, see ASNA02 for many examples.