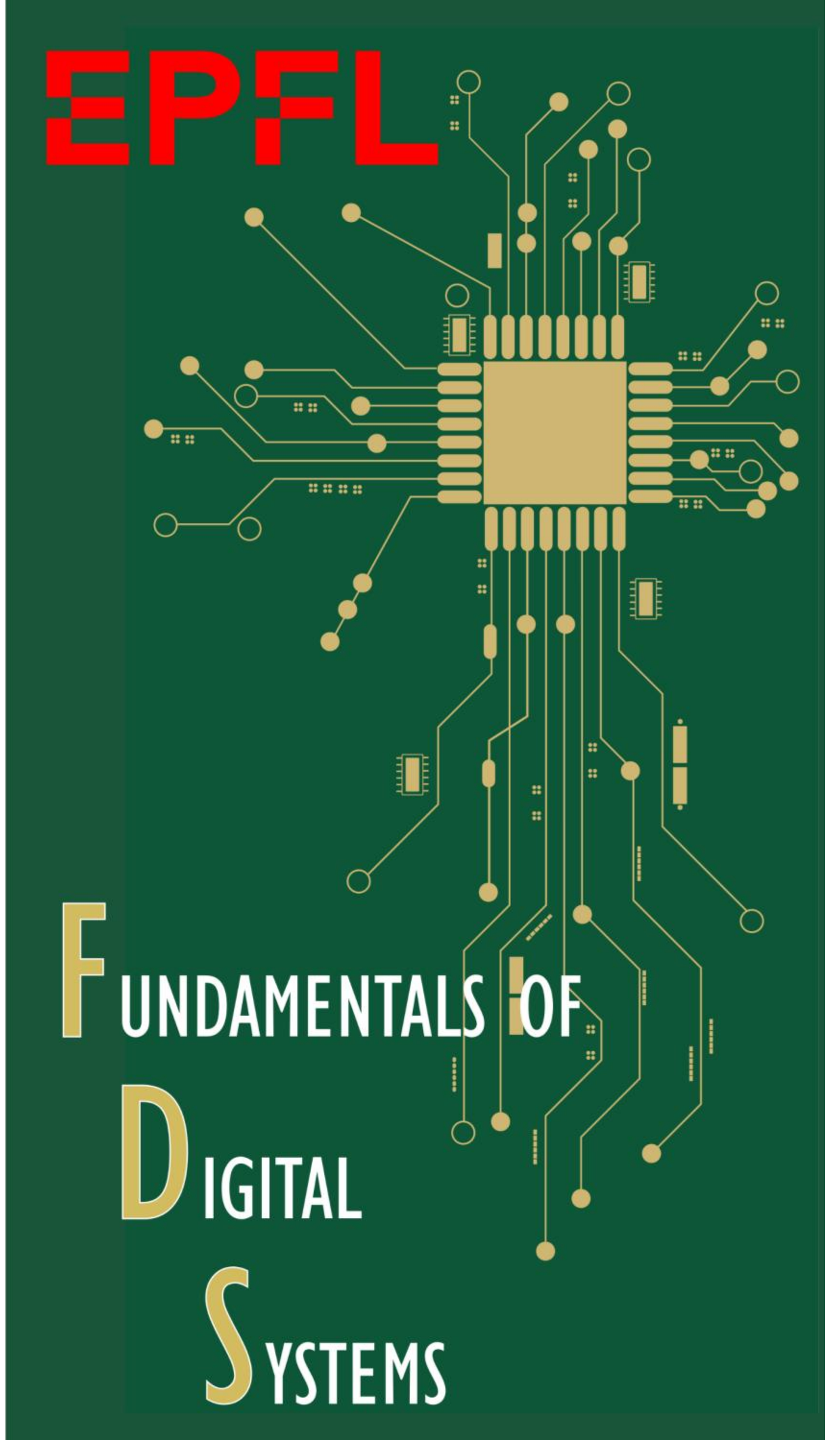


Number Systems

Arithmetic Operations with Fractional Numbers

CS-173 Fundamentals of Digital Systems

Mirjana Stojilović

Spring 2025

Previously on FDS

…Fixed- and Floating-Point Representations

Previously

- Discovered the notion of radix point
- Learned two representations for fractional numbers
  - Fixed-point
  - Floating-point
    - IEEE 754 standard
- Evaluated and analyzed precision, resolution, range, accuracy, dynamic range, rounding

Fixed- and Floating-Point Representations

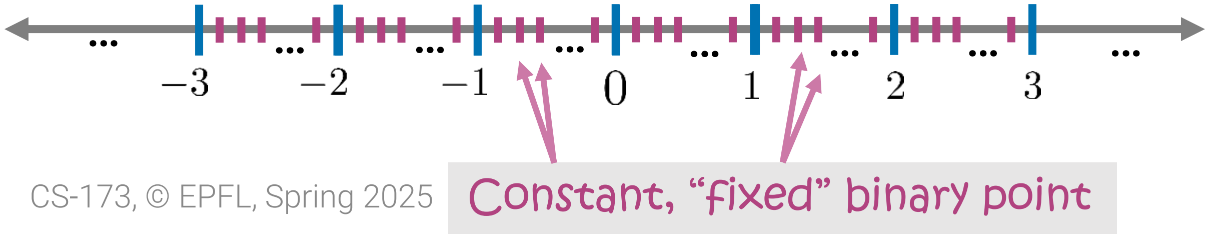
- Fixed-point

$$X = (X_{m-1}X_{m-2}\ldots X_1X_0.X_{-1}X_{-2}\ldots X_{-f})$$

Integer component | Radix point | Fractional component

- Value (two's complement)

$$x = -X_{m-1}2^{m-1} + \sum_{i=-f}^{m-2} X_i 2^i$$



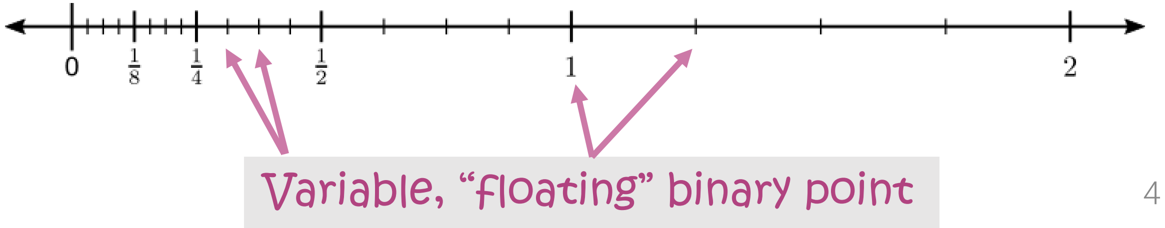
- Floating-point

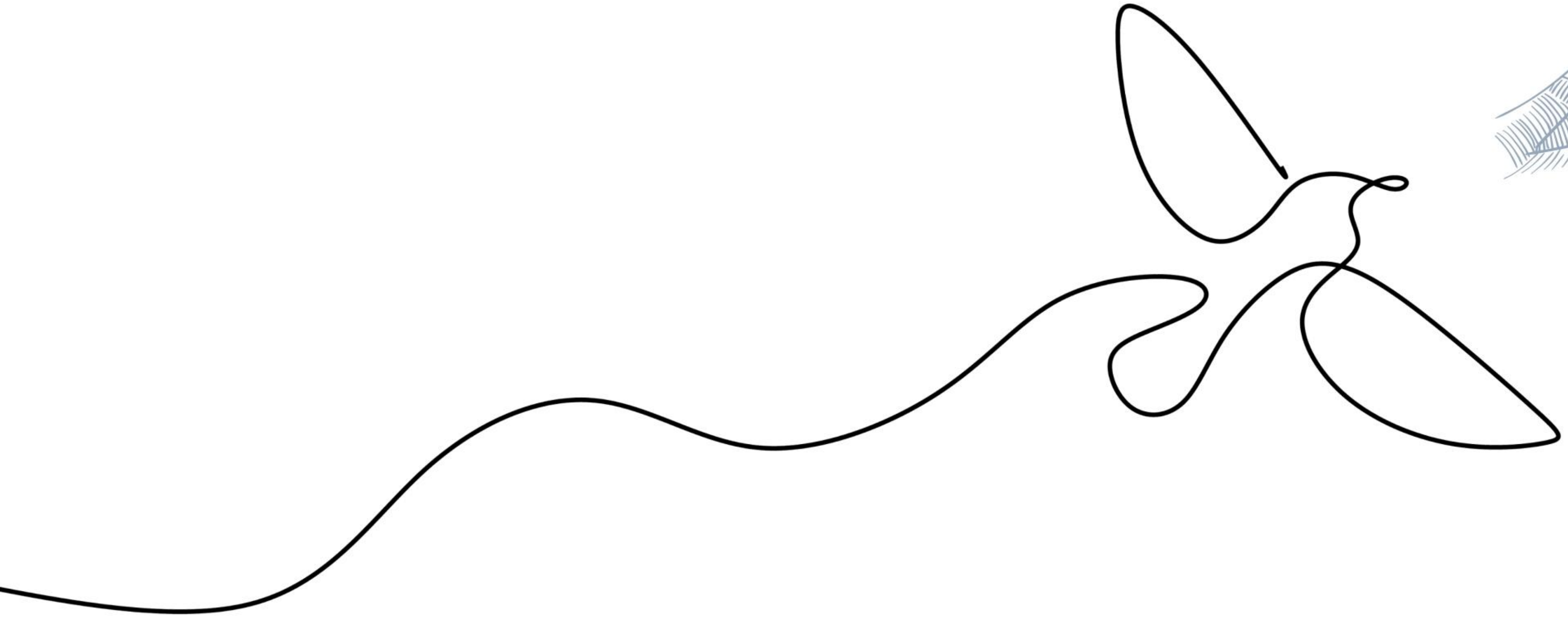
$$X = (SE_{m-1}E_{m-2}\ldots E_1E_0M_{n-1}M_{n-2}\ldots M_0)$$

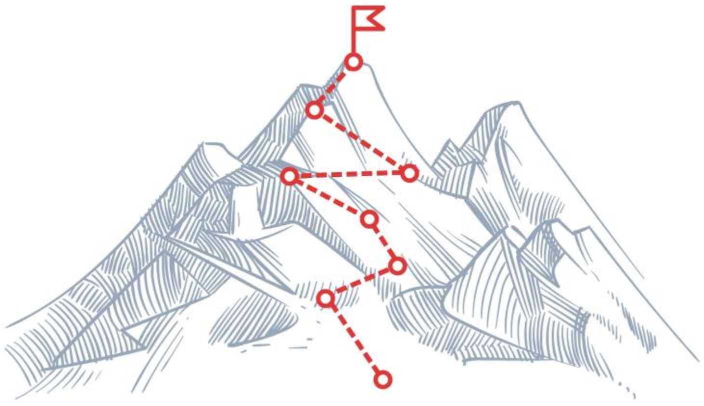
Sign | Exponent | Magnitude

- Value (sign-and-magnitude mantissa)

$$x = (-1)^S \times M \times b^E$$

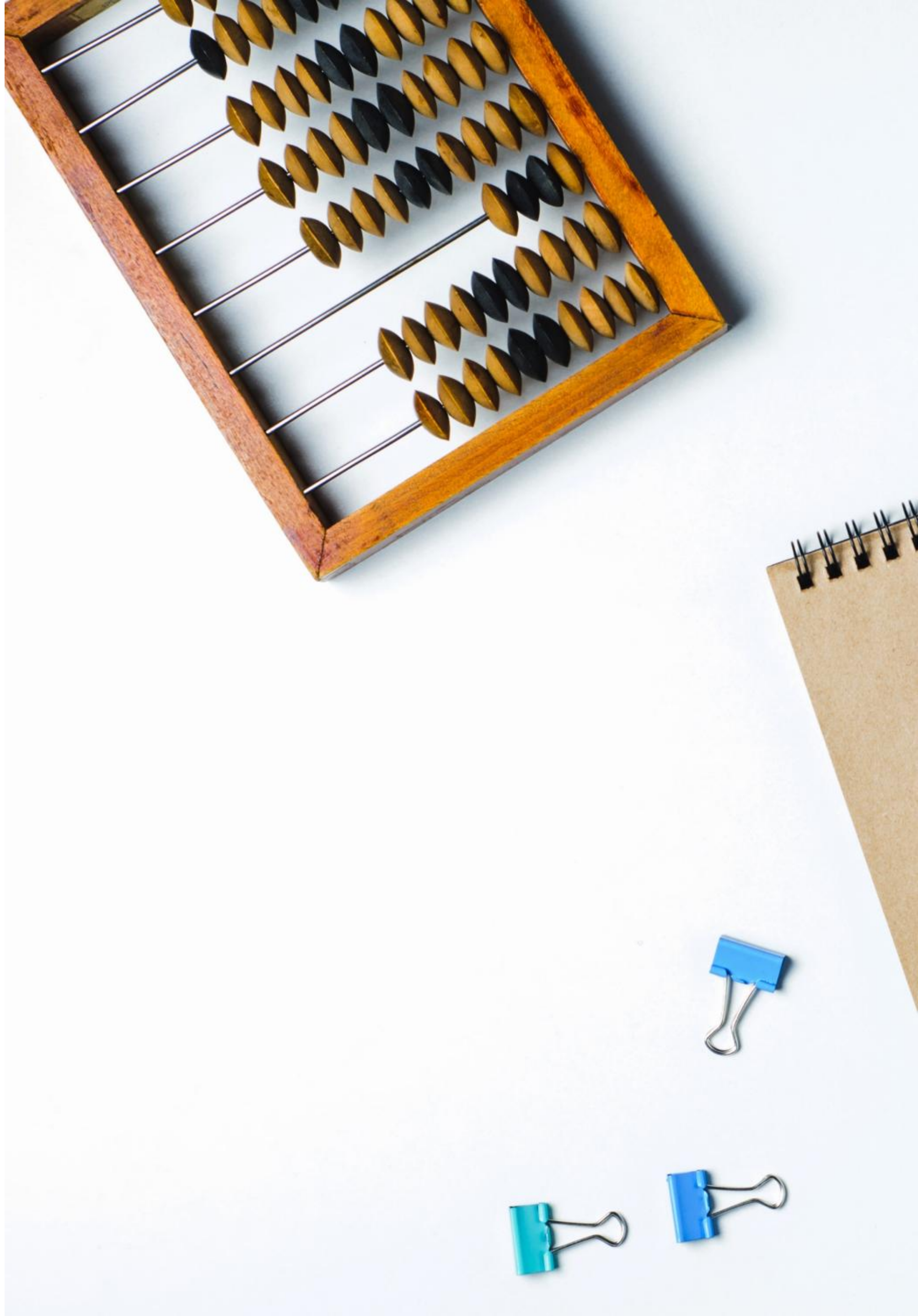




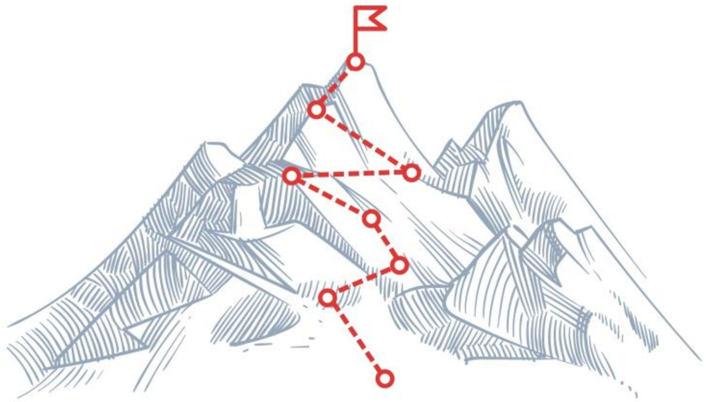


Let’s Talk About…

…Performing arithmetic operations with fractional numbers



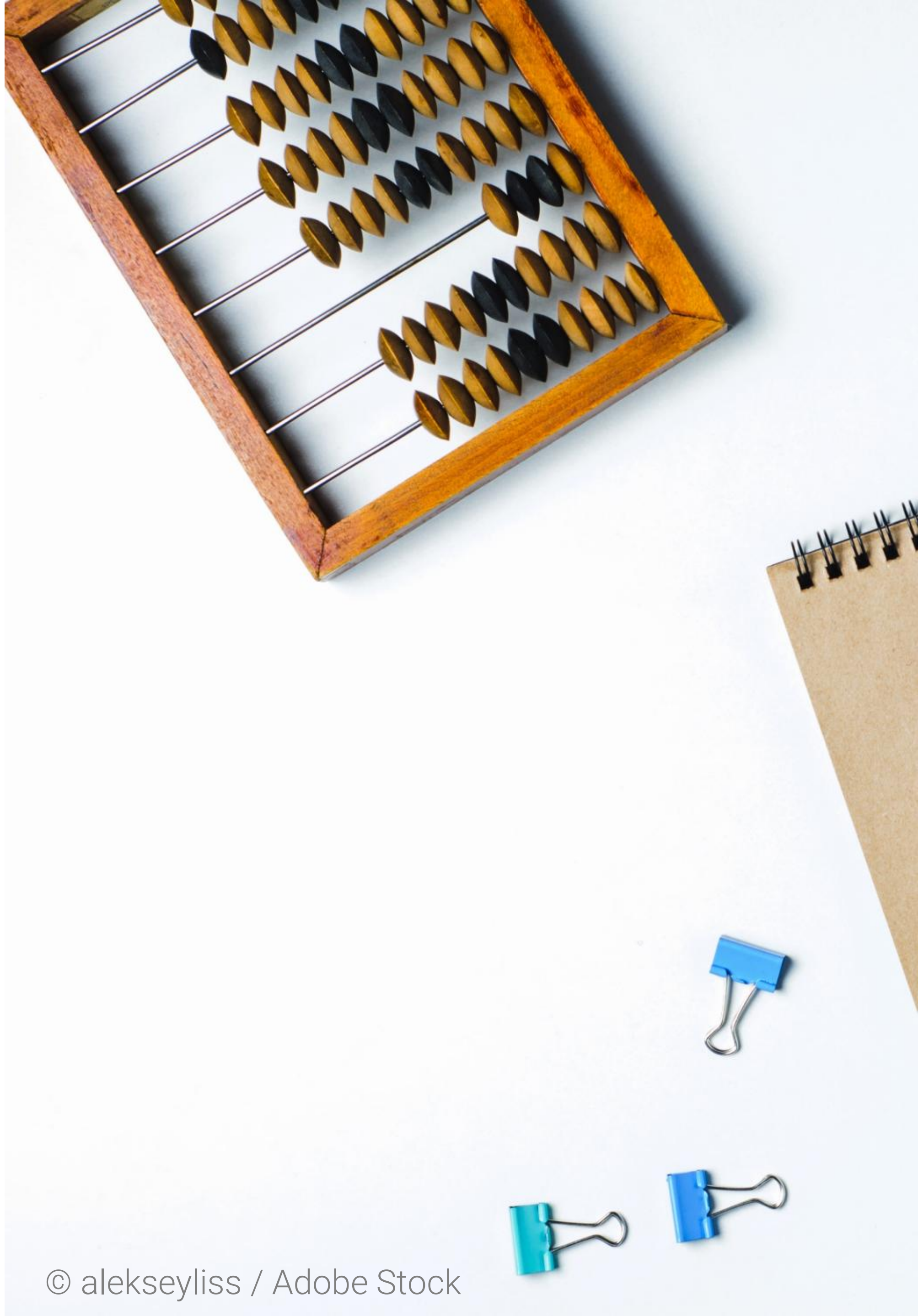
Learning Outcomes



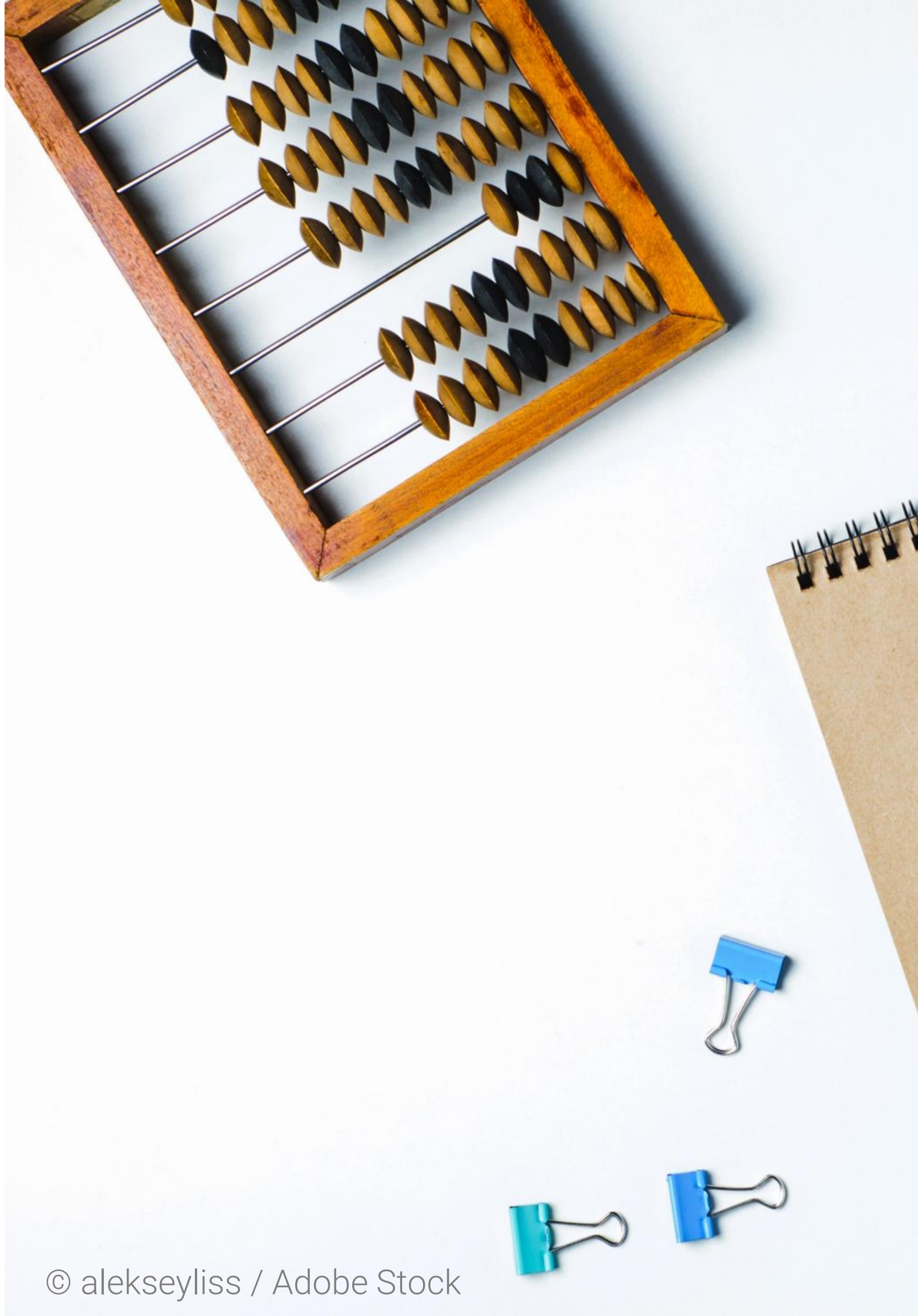
- Perform +/-/× pen-and-paper style
  - Fixed-point
  - Floating-point

Quick Outline

- Fixed-point arithmetic
- Floating-point arithmetic



Fixed-Point Arithmetic



Fixed-Point Arithmetic

Addition/Subtraction

- Performing **+** or **-** on two binary numbers $x(m,f)$ and $y(m,f)$ is done in **the same way** as if the operands were integers
  - Overflow can happen

$$x + y = x_{\text{int}} + x_{\text{fr}} + y_{\text{int}} + y_{\text{fr}}$$

$$x - y = x_{\text{int}} + x_{\text{fr}} - (y_{\text{int}} + y_{\text{fr}})$$

Fixed-Point Arithmetic

Example: Addition/Subtraction

- Addition

$$\begin{array}{rr} & X \\ + & Y \\ \hline & X+Y \end{array} \Rightarrow \begin{array}{rl} & 011011\ 100 \leftarrow \text{Carry} \\ & 000101.110 = 5.75 \\ + & 001100.011 = 12.375 \\ \hline & 010010.001 = 18.125 \end{array}$$

- Subtraction

$$\begin{array}{rr} & X \\ - & Y \\ \hline & X-Y \end{array} \Rightarrow \begin{array}{rl} & 1110000\ 110 \leftarrow \text{Borrow} \\ & 000101.110 = 5.75 \\ - & 001100.011 = 12.375 \\ \hline & 111001.011 = -6.625 \end{array}$$

Fixed-Point Arithmetic

Addition/Subtraction in Two's Complement

- How many bits to represent the integer and fractional parts?

$$x \pm y = \left(-X_{(m_x-1)}2^{(m_x-1)} + \sum_{i=-f_x}^{m_x-2} X_i 2^i\right) \pm \left(-Y_{(m_y-1)}2^{(m_y-1)} + \sum_{i=-f_y}^{m_y-2} Y_i 2^i\right)$$

The largest integer-part exponent: $\max(m_x - 1, m_y - 1)$
Consequently: $m_{x\pm y} = \max(m_x, m_y) + 1$

Number of bits for the integer component

The smallest fractional-part exponent: $\min(-f_x, -f_y)$
Consequently: $f_{x\pm y} = \max(f_x, f_y)$

Number of bits for the fractional component

Fixed-Point Addition

Example: Analogy With Decimal Numbers

- How many bits to represent the integer and fractional parts?

$$\begin{array}{r} 9.99 \\ +\ 999.9999 \\ \hline 1009.9899 \end{array} \qquad \begin{array}{l} m_x = 1, f_x = 2 \\ m_y = 3, f_x = 4 \end{array}$$

$$m_{x \pm y} = \max(m_x, m_y) + 1 \qquad f_{x \pm y} = \max(f_x, f_y)$$

$$m_{x+y} = \max(1, 3) + 1 = 4 \qquad f_{x+y} = \max(2, 4) = 4$$

Fixed-Point Arithmetic

Multiplication, Same Operand Format

- Multiplication on two binary numbers $x(m, f)$ and $y(m, f)$
  - Same algorithm as if the operands were integers
  - **Binary point location changes;** overflow can happen
  - How many bits to represent the integer and fractional parts?
- In two's complement:

$$x \cdot y = \left(-X_{m-1}2^{m-1} + \sum_{i=-f}^{m-2} X_i 2^i \right) \cdot \left(-Y_{m-1}2^{m-1} + \sum_{i=-f}^{m-2} Y_i 2^i \right)$$

The largest integer-part exponent: $(m-1)+(m-1)$
Consequently: $m_{xy} = 2m$

The smallest fractional-part exponent: $(-f)+(-f)$
Consequently: $f_{xy} = 2f$

Fixed-Point Arithmetic

Multiplication, Generalization

- Multiplication on two binary numbers $x(m_x, f_x)$ and $y(m_y, f_y)$
  - How many bits to represent the integer and fractional parts?

$$x \cdot y = (x_{\text{int}} + x_{\text{fr}}) \cdot (y_{\text{int}} + y_{\text{fr}})$$

- In two's complement:

$$x \cdot y = \left(-X_{m_x-1} 2^{m_x-1} + \sum_{i=-f_x}^{m_x-2} X_i 2^i \right) \cdot \left(-Y_{m_y-1} 2^{m_y-1} + \sum_{i=-f_y}^{m_y-2} Y_i 2^i \right)$$

The largest integer-part exponent: $(m_x - 1) + (m_y - 1)$
Consequently: $m_{xy} = m_x + m_y$

The smallest fractional-part exponent: $(-f_x) + (-f_y)$
Consequently: $f_{xy} = f_x + f_y$

Fixed-Point Multiplication

Example: Analogy With Decimal Numbers

- How many bits to represent the integer and fractional parts?

$$
\begin{array}{r}
9.99 \\
\times \quad 999.9999 \\
\hline
9989.999001
\end{array}
\qquad
\begin{array}{l}
m_x = 1, f_x = 2 \\
m_y = 3, f_x = 4
\end{array}
$$

$$m_{xy} = m_x + m_y \qquad f_{xy} = f_x + f_y$$

$$m_{xy} = 1 + 3 = 4 \qquad f_{xy} = 2 + 4 = 6$$

EXAMPLES

EXAMPLES

Fixed-Point Arithmetic

Example: Multiplication

- Format

$$m_x = m_y = 3$$
$$f_x = f_y = 2$$

- Example

$$\begin{array}{r} X \\ \times \quad Y \\ \hline X \times Y \end{array} \qquad \begin{array}{r} 2.75 \\ \times \quad 3.25 \\ \hline 8.9375 \end{array}$$

010.11	Multiplicand
× 011.01	Multiplier
000000	First partial product (always zero), sign-extended
+ 001011	1 x multiplicand, sign-extended
0001011	Intermediate result, sign-extended
+ 000000	0 x multiplicand, left-shifted by 1 place and sign-extended
00001011	Intermediate result, sign-extended
+ 001011	1 x multiplicand, left-shifted by 2 places and sign-extended
000110111	Intermediate result, sign-extended
+ 001011	1 x multiplicand, left-shifted by 3 places and sign-extended
0010001111	Intermediate result, sign-extended
+ 000000	0 x multiplicand, left-shifted by 4 places and sign-extended
0010001111	Result, integer → convert to fixed-point now

Integer Multiplication

Example Contd.

- Integer result 0010001111 needs now to be converted to fixed-point representation
  - Assuming we can use as many bits as required to represent the integer and fractional parts

$$m_{xy} = m_x + m_y = 6$$
$$f_{xy} = f_x + f_y = 4$$

$$001000.1111 = 8.9375$$

  - Assuming the **same** format for multiplicand, multiplier, and the result

$$m_{xy} = m_x = m_y = 3$$
$$f_{xy} = f_x = f_y = 2$$

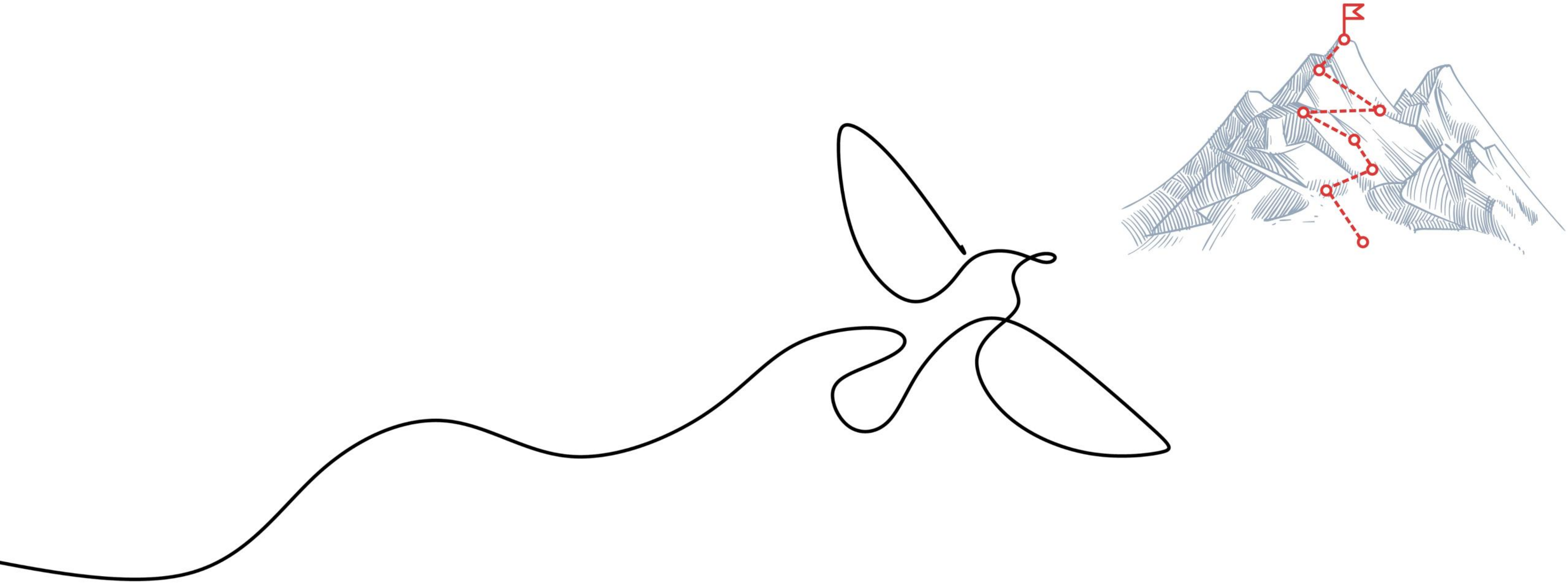
~~001~~000.11~~11~~ $= 0.75$

$\neq 8.9375$

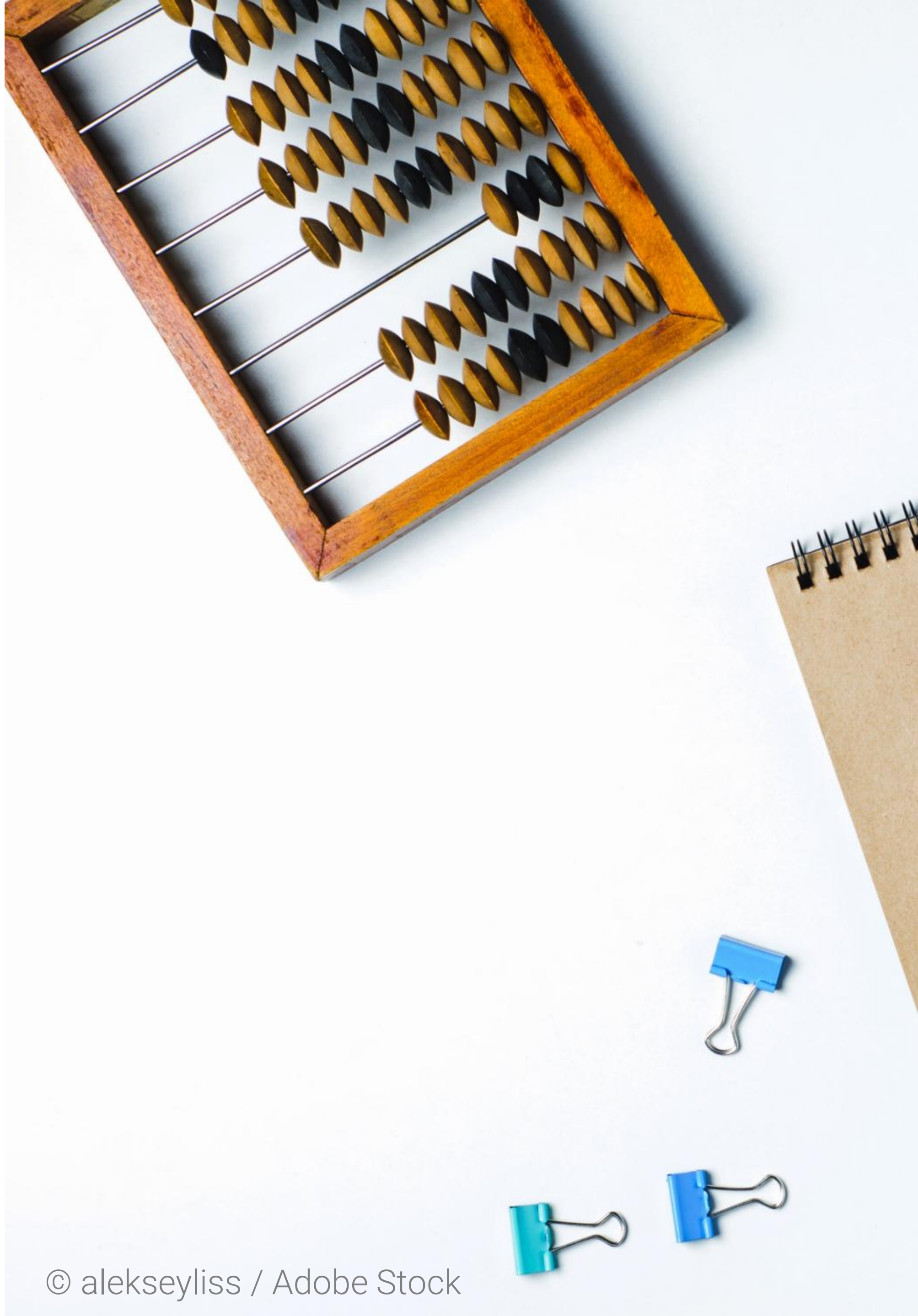
In practice, the format is fixed, and erroneous results may happen

Pros and Cons of Fixed-Point Representation

- 👍 Arithmetic operations on integers can be applied to fixed-point numbers without modifications
  - 👍 Portable: we can reuse the same integer processing digital hardware
  - 👍 Like with integers, arithmetic operations are performed efficiently (fast)
  - 👍 Used in image and signal processing and communication
- 👎 Complex data and algorithm analysis
  - 👎 Where to put the binary point to achieve good accuracy?
- 👎 There are other number formats (floating-point) that provide more extensive dynamic range



Floating-Point Arithmetic



© alekseyliss / Adobe Stock

Floating-Point Arithmetic
Addition/Subtraction

- Let x and y be represented as (S_x, M_x, E_x) and (S_y, M_y, E_y)
  - The signed significands $M^* = (-1)^S M$ are normalized

- Addition/subtraction result is z, also represented as (S_z, M_z, E_z)

$$z = x \pm y = M_x^* \times 2^{E_x} \pm M_y^* \times 2^{E_y}$$

  - The significand of the result is also normalized

$$z = M_z^* \times 2^{E_z}$$

Floating-Point Addition/Subtraction

Algorithm

- Four main steps to compute and produce the result of +/-

1
  - Add/subtract significand (mantissa) and set the exponent

    The mantissa of the number with the **smaller** exponent has to be multiplied by two to the power of the difference between the exponents (this operation is called **alignment**) and then added/subtracted to the mantissa of the other number

$$M_z^* = \begin{cases} M_x^* \pm \left(M_y^* \times 2^{(E_y - E_x)}\right) & \text{if } E_x \geq E_y \\ \left(M_x^* \times 2^{(E_x - E_y)}\right) \pm M_y^* & \text{if } E_x < E_y \end{cases}$$

$$E_z = \max(E_x, E_y)$$

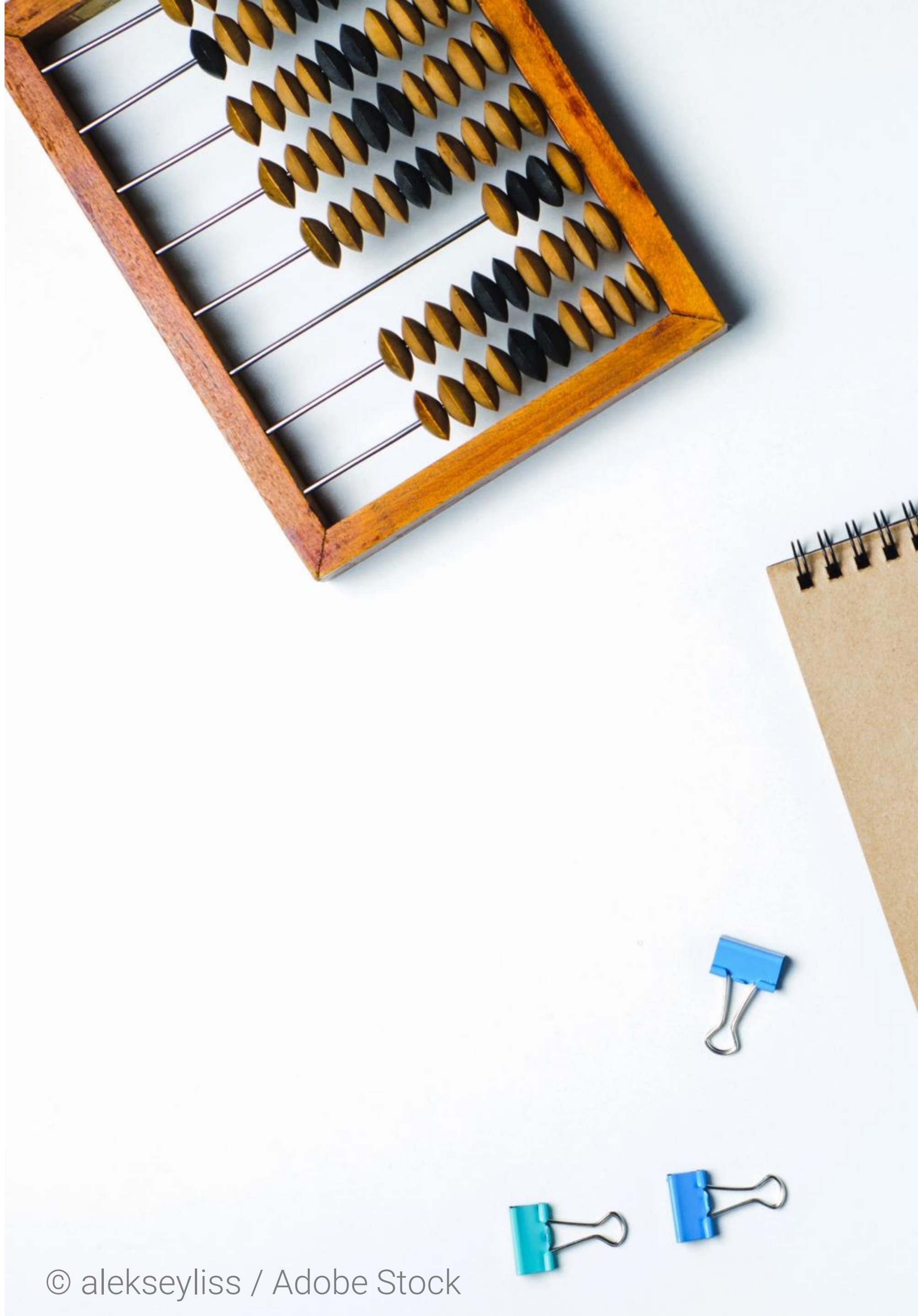
2
  - **Normalize** the result and then, if required, **adjust the exponent**

3
  - **Round** the result and then, if required, **normalize** it and **adjust the exponent**

4
  - Set flags for **special values**, if required

Floating-Point +/-

Step 1: Align and +/-



Alignment

Example: Analogy with Decimal Numbers

- Example

Normalized, 3-bit fraction

$$1.895 \times 10^5 + 5.440 \times 10^3 = (194940)_{10}$$

  - Approach 1: Align the two operands to a common exponent, e.g., zero

Shifted left (<< 3)

Shifted left (<< 5)

Not normalized

$$1.895 \times 10^5 + 5.440 \times 10^3 = (189500.000 + 5440.000) \times 10^0 = 194940.000 \times 10^0$$

  - Cons: as the exponents of the operands are different from zero, both significands need adjusting/shifting (unnecessary additional work)
  - The result needs to be normalized, and the exponent adjusted

Alignment

Analogy with Decimal Numbers, Contd.

- Example
  - Approach 2: align to the **common exponent—min of the two**

After left shift (<< |5-3|)

Not normalized

$$1.895 \times 10^5 + 5.440 \times 10^3 = (189.500 + 5.440) \times 10^3 = 194.940 \times 10^3$$

  - Pros: Only one alignment (one adjustment of the significand)
  - The result needs to be normalized, and the exponent adjusted
  - Left shift: some of the most significant bits of one of the two significands are lost in the process; **potentially a large error**

$$1.895 \times 10^5 + 5.440 \times 10^3 = (\cancel{1}89.500 + 5.440) \times 10^3$$
$$= (14940)_{10} \neq (194940)_{10}$$

Alignment

Analogy with Decimal Numbers, Contd.

- Example
  - Approach 3: align to the **common exponent—max of the two**

After right shift (>> |5-3|)

Normalized

$$1.895 \times 10^5 + 5.440 \times 10^3 = (1.895 + 0.0544) \times 10^5 = 1.9494 \times 10^5$$

  - Pros: only one alignment (one adjustment of the significand)
  - The result needs to be normalized, and the exponent adjusted
  - Right shift: Some least-significant bits of one of the two significands may get lost in the process, but the potential **error is much smaller**

$$1.895 \times 10^5 + 5.440 \times 10^3 = (1.895 + 0.054\cancel{4}) \times 10^5$$

$$= (194900)_{10} \approx (194940)_{10}$$

Floating-Point Addition/Subtraction

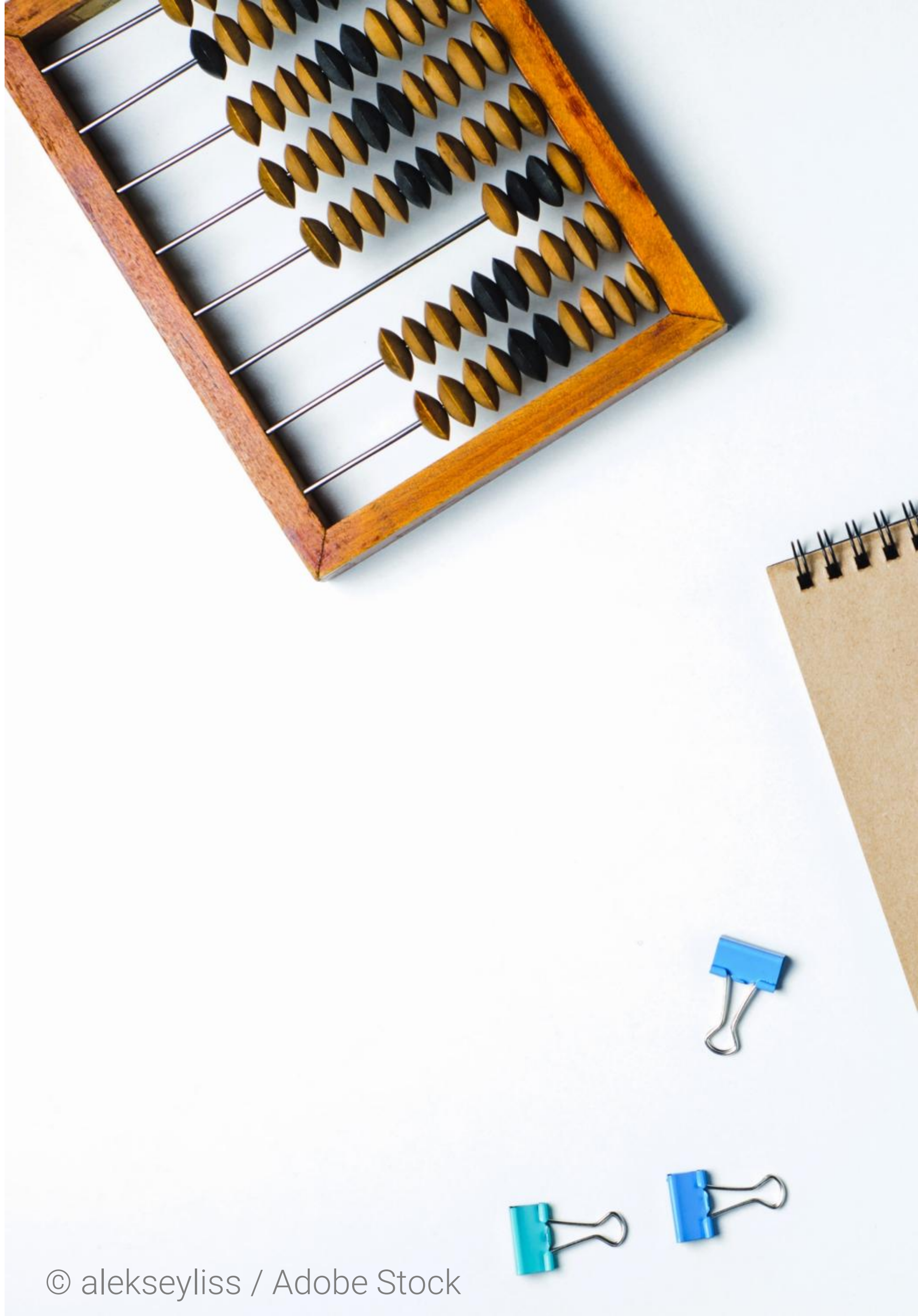
Step 1: Recap

- Recall Step 1: Add/subtract significand and set exponent
- Algorithm:
  - Subtract exponents $d = |E_x - E_y|$
  - Align significands (mantissas)
    - Compare the exponents of the two operands
    - Shift right d positions the significand of the operand with the smaller exponent
    - Select as the exponent of the result the larger exponent
  - Add/subtract signed significands and produce the sign of the result

FP operation	Signs of the operands	Effective operation
+	=	add
+	≠	subtract
-	=	subtract
-	≠	add

Floating-Point +/-

Step 2: Normalization



Floating-Point Addition/Subtraction
Normalization

- Various situations may occur
  - Scenario 1:
    - The result is already normalized. No action is needed.
  - Example:

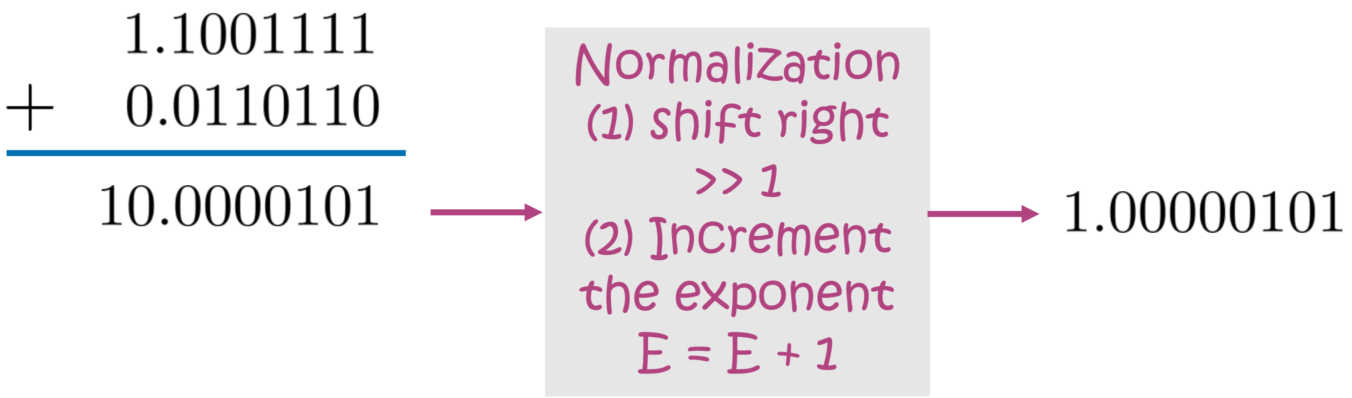
$$\begin{array}{r} 1.10011111 \\ +\quad 0.00101011 \\ \hline 1.11001010 \end{array}$$

Normalized

EXAMPLES

Floating-Point Addition/Subtraction
Normalization, Contd.

- Various situations may occur
  - Scenario 2: When **adding**, the significand might **overflow**
  - Steps to perform normalization:
    - Shift right the result by one position
    - Increment the exponent by one
  - Example:



Floating-Point Addition/Subtraction

Normalization, Contd.

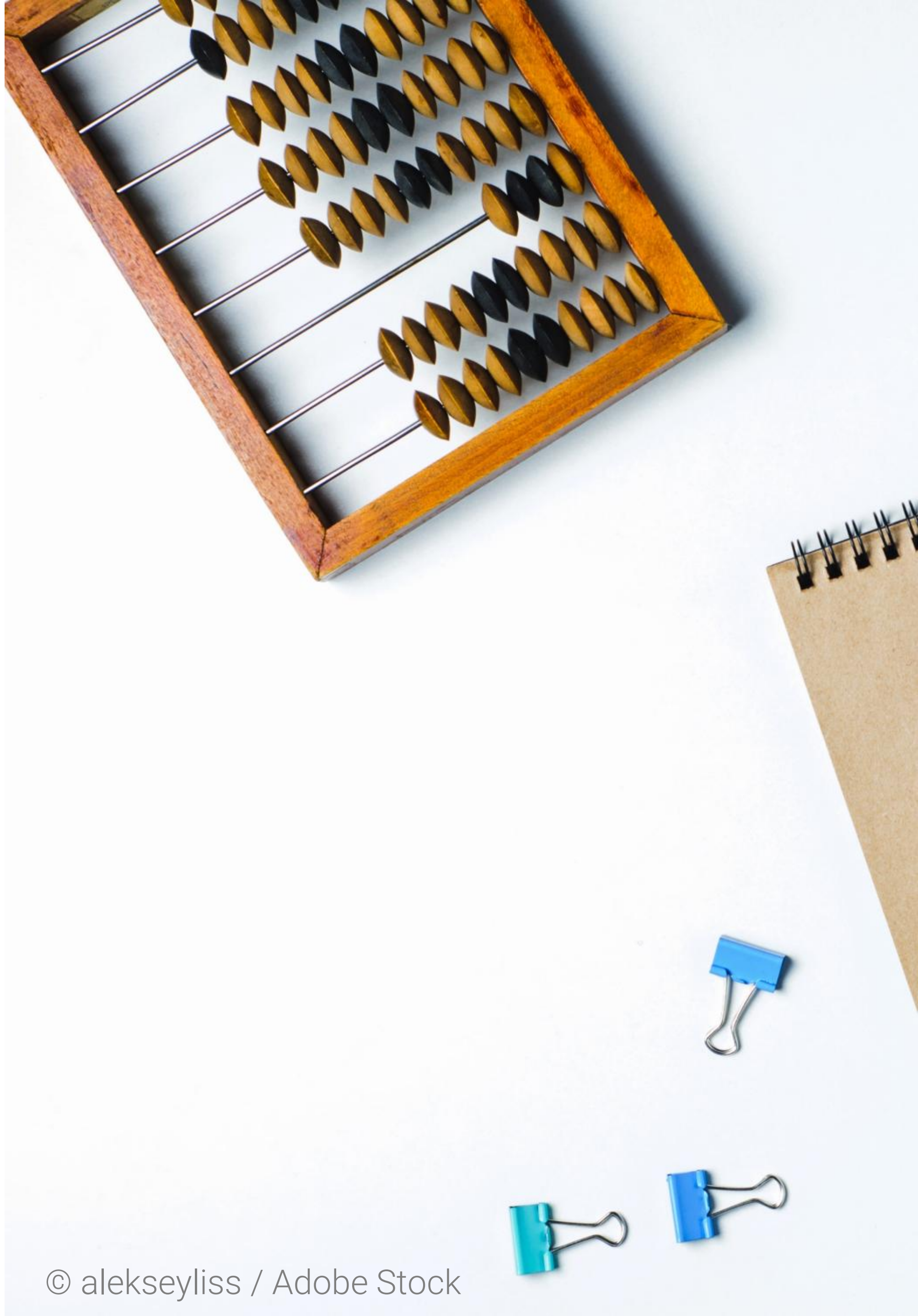
- Various situations may occur
  - Scenario 3: When **subtracting**, the result might have **leading zeros**
  - Steps to perform normalization:
    - Shift left the result by as many positions as there are leading zeros
    - Decrement the exponent by the number of leading zeros
  - Example:

$$\begin{array}{r} 1.1001111 \\ -\quad 1.1001010 \\ \hline 0.0000101 \end{array} \longrightarrow \boxed{\text{Normalization}} \longrightarrow 1.0100000$$

Normalization
(1) count leading zeros
p = 5
(1) shift left
<< p
(2) Decrement
the exponent
E = E - p

Floating-Point +/-

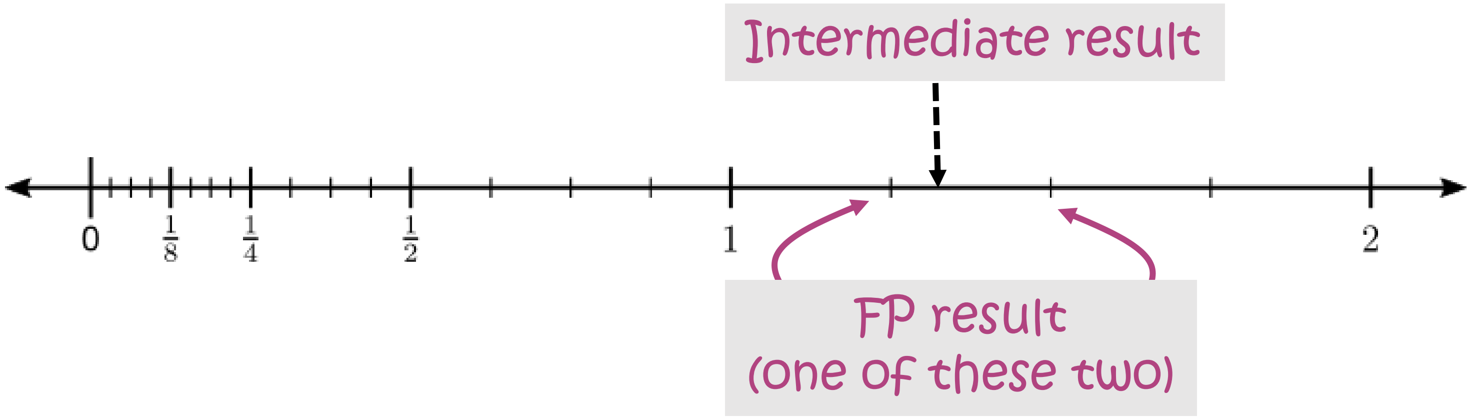
Step 3: Rounding



© alekseyliss / Adobe Stock

Floating-Point Addition/Subtraction

Rounding



- The result may not be representable in the given number format
- Perform rounding
  - Towards zero: truncate the least-significant bits
  - Towards ±∝ : requires addition
  - **[default]** To nearest, to even when tie: requires addition

Rounding to Nearest

To Even if Tie

- The FP result is as close as possible to the exact value
  - Minimized roundoff error (**default** rounding mode in IEEE 754)
  - **Tie to even** is preferred because it leads to smaller errors when the result is divided by two—a frequent operation
- Assuming a significand of infinite precision and radix r, **round to the nearest** can be obtained by **adding** $(r^{-f})/2$ to the infinite precision significand and keeping the resulting f fractional digits
  - If overflow: normalization and the exponent adjustments are needed

Rounding to Nearest

To Even if Tie

- Round the given value to the nearest 8-bit fraction:

$$\begin{array}{r} 1.100100011101 \\ +\quad 1 \\ \hline 1.10010010 \end{array}$$

Keep 8 bits

Exact value, but not representable

Addition with $(2^{-8})/2 = 2^{-9}$

Result, after rounding

$$\begin{array}{r} 1.100100001101 \\ +\quad 1 \\ \hline 1.10010001 \end{array}$$

Keep 8 bits

Exact value, but not representable

Addition with $(2^{-8})/2 = 2^{-9}$

Result, after rounding

Rounding to Nearest

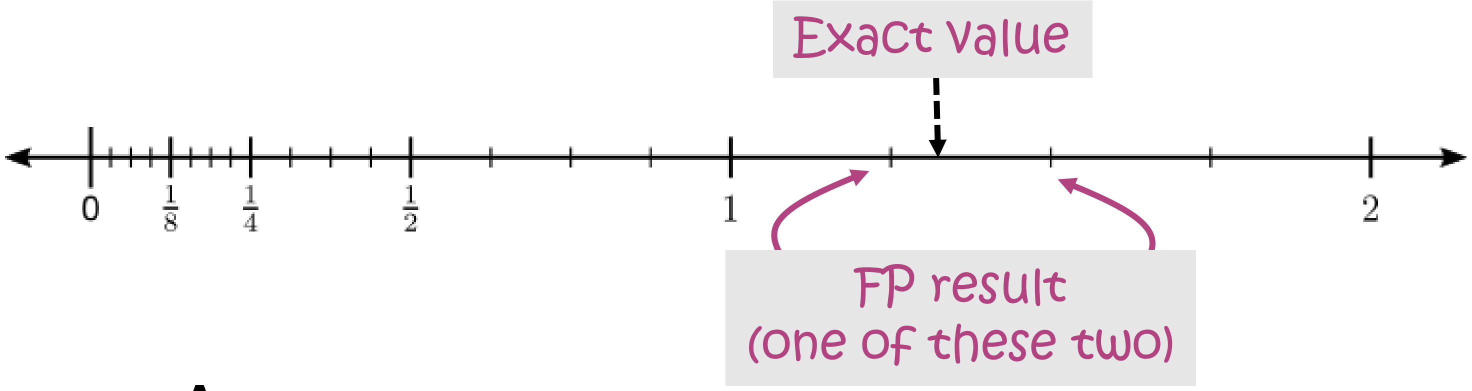
To Even if Tie

- **Q:** Round the value 1.100100001 to the nearest 8-bit fraction

- **A:** 1.10010000
  - Looking at 1.100100001, notice
    - It's a tie
    - If we ignore the tie bits, what is left is an even number
  - If we were to add anything, we'd end up rounding to the nearest odd number
    - Therefore, in this example, it suffices to truncate the "tie" bits

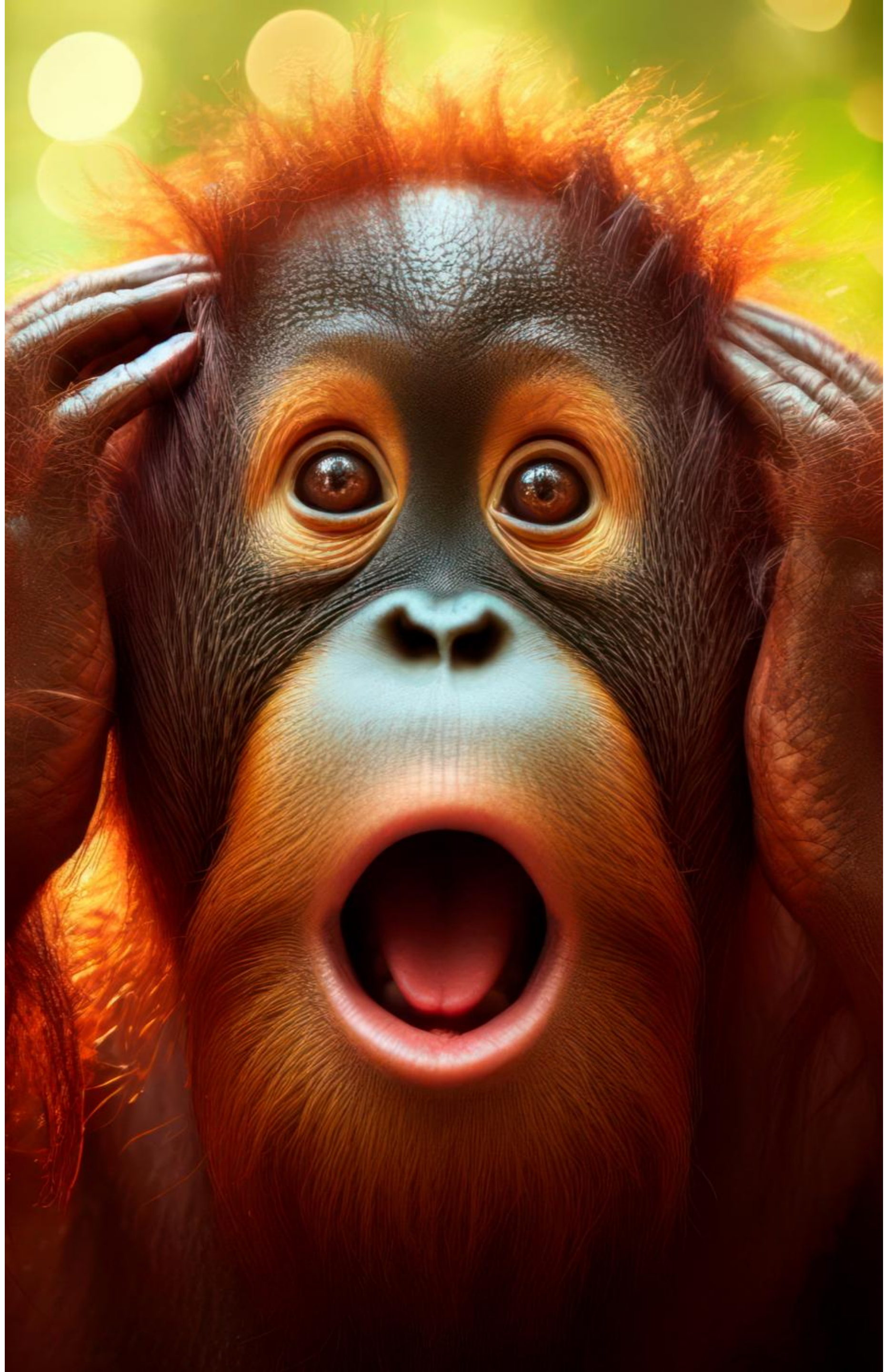
Max Round-off Error

- **Q:** Rounding to nearest, f fractional digits. What is the **maximum** difference between the exact value and its FP representation?



- **A:**
  - When the exact value is in the middle and the exponent is the max

$$\frac{2^{-f}}{2} \times 2^{E_{\max}}$$

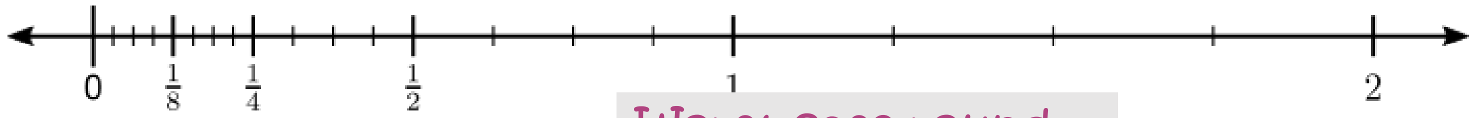


Max Round-off Error

Example Floating-Point

- Rounding to nearest, f fractional digits. Find the worst-case round-off error
- A: Max round-off error occurs for the largest positive exponent

Exponent increases



Worst-case round-off error increases

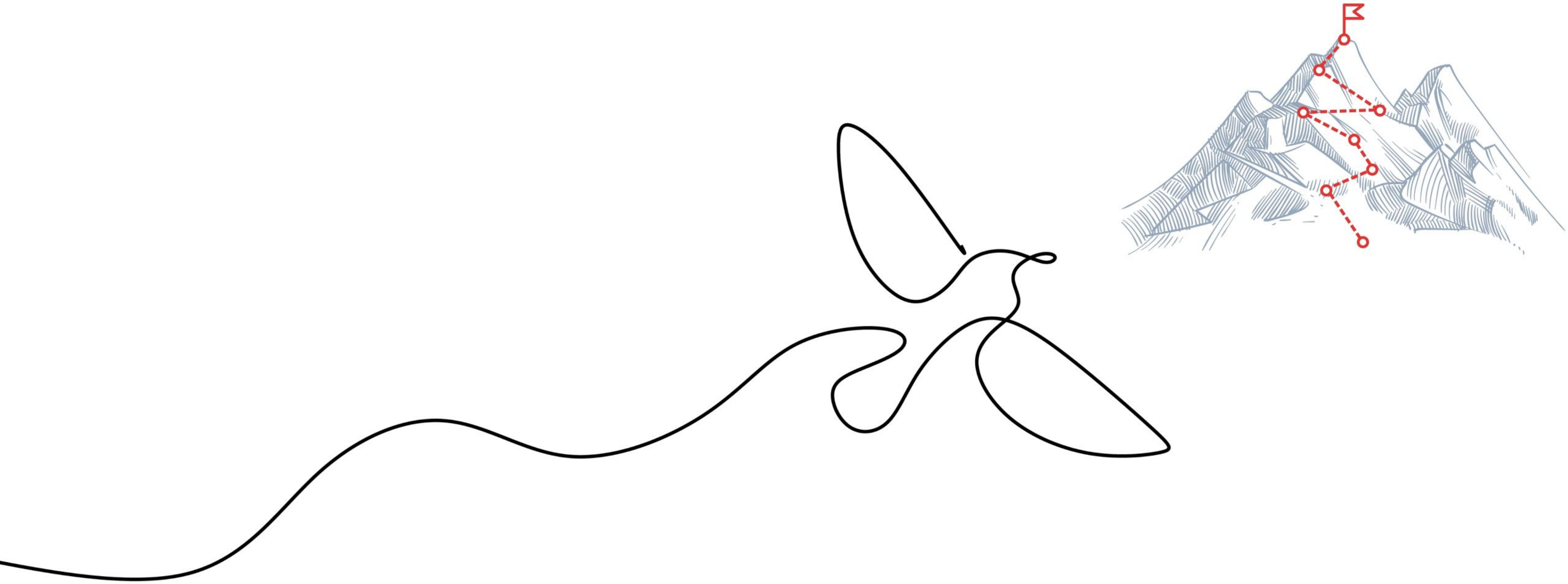
½ for the worst-case, when the real value is in the middle between two consecutive FP numbers

$$\max(\text{Error}_{\text{round-off}}) = \frac{1}{2}\left(x_{\max} - x_{(\text{imediately preceeding the max})}\right)$$

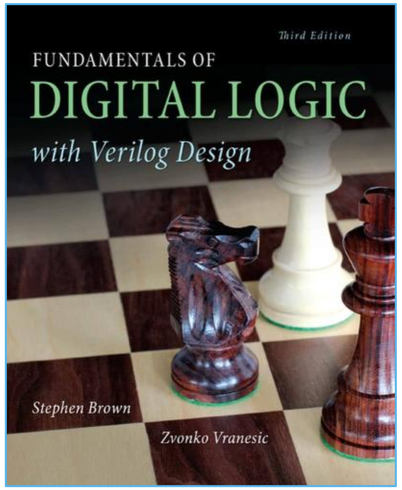
$$\max(\text{Error}_{\text{round-off}}) = \frac{1}{2}\left(1.11...11 \times 2^{E_{max}} - 1.11...10 \times 2^{E_{max}}\right) = \frac{1}{2}2^{-f} \times 2^{E_{max}}$$

! Computing with large FP numbers may lead to (very) unexpected results

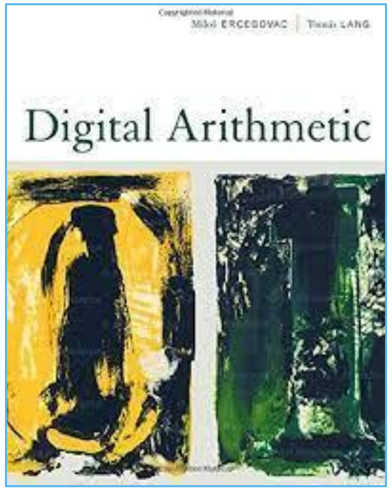
EXAMPLES



Literature



- Chapter 3: Number Representation and Arithmetic Circuits
  - 3.7.1
  - 3.7.2
- On the web: Wiki, IEEE 754 [link]



- Chapter 1: Preview of Basic Number Representations and Arithmetic Algorithms
  - 1.2.5
- Chapter 8: Floating-Point Representation, Algorithms, and Implementations
  - 8.1–8.3
  - 8.4.1
  - 8.5.1