# Introduction to Computer Arithmetic

Floating-point arithmetic and error analysis (AFAE)

Sorbonne Université

## Contents

# 1 Exercise 1: Representation of Signed Integers

## 1.1 Question 1: Three ways to represent an 8-bit signed integer

### 1.1.1 Method 1: Sign-Magnitude

- **Format**: 1 bit for sign (0=positive, 1=negative) + 7 bits for magnitude
- **Range**: $-(2^7-1)$ to $+(2^7-1) = -127$ to $+127$
- **19**: `0001 0011` (sign=0, magnitude=19)
- **-19**: `1001 0011` (sign=1, magnitude=19)

### 1.1.2 Method 2: One's Complement

- **Format**: Positive numbers as usual; negative numbers by flipping all bits
- **Range**: $-(2^7-1)$ to $+(2^7-1) = -127$ to $+127$
- **19**: `0001 0011`
- **-19**: `1110 1100` (flip all bits of 19)

### 1.1.3 Method 3: Two's Complement (Most Common)

- **Format**: Positive numbers as usual; negative numbers by flipping bits and adding 1
- **Range**: $-2^7$ to $+(2^7-1) = -128$ to $+127$
- **19**: `0001 0011`
- **-19**: `1110 1101` (flip bits: 1110 1100, then add 1)

**Advantages of Two's Complement**

- No duplicate zero
- Simpler arithmetic circuits
- Addition/subtraction use the same hardware

# 2 Exercise 2: IEEE-754 Single Precision Representation

## 2.1 IEEE-754 Single Precision Format

| S | Exponent | Mantissa |
|---|---|---|
| 1 bit | 8 bits | 23 bits |

**Formula**: $(-1)^S \times 1.M \times 2^{E-127}$

## 2.2 Number Representations

### 2.2.1 1. Number: 13

**Binary conversion**: $13 = 1101_2 = 1.101_2 \times 2^3$

- **Sign (S)**: 0 (positive)
- **Exponent (E)**: $3 + 127 = 130 = 10000010_2$
- **Mantissa (M)**: 101 followed by 20 zeros

**IEEE-754**: `0 10000010 10100000000000000000000`
**Hex**: `0x41500000`

### 2.2.2 2. Number: 0.4375

**Binary conversion**:

$$\begin{gathered} 0.4375 \times 2 = 0.875 \to 0 \\ 0.875 \times 2 = 1.75 \to 1 \\ 0.75 \times 2 = 1.5 \to 1 \\ 0.5 \times 2 = 1.0 \to 1 \end{gathered}$$

$0.4375 = 0.0111_2 = 1.11_2 \times 2^{-2}$

- **Sign (S)**: 0
- **Exponent (E)**: $-2 + 127 = 125 = 01111101_2$
- **Mantissa (M)**: 11 followed by 21 zeros

**IEEE-754**: `0 01111101 11000000000000000000000`
**Hex**: `0x3EE00000`

### 2.2.3 3. Number: $-0.4375$

Same as 0.4375 but with sign bit = 1
**IEEE-754**: `1 01111101 11000000000000000000000`
**Hex**: `0xBEE00000`

### 2.2.4 4. Number: $1 + 2^{-24}$

This is exactly representable in single precision!
$1 + 2^{-24} = 1.000000000000000000000001_2 \times 2^0$

- **Sign (S)**: 0
- **Exponent (E)**: $0 + 127 = 127 = 01111111_2$
- **Mantissa (M)**: 23 zeros followed by 1 in the last position

**IEEE-754**: `0 01111111 00000000000000000000001`
**Hex**: `0x3F800001`

### 2.2.5 5. Number: $1 + 2^{-24} - 2^{-25}$

$= 1 + 2^{-25}(2 - 1) = 1 + 2^{-25}$

This cannot be exactly represented (mantissa only has 23 bits). The exact binary would need a 1 in position 25.

**Rounding to nearest**: This rounds to 1.0 (the $2^{-25}$ is below the precision)

**IEEE-754**: `0 01111111 00000000000000000000000`

**Hex**: `0x3F800000`

### 2.2.6 6. Number: $1 + 2^{-24} + 2^{-25}$

$= 1 + 2^{-25}(2 + 1) = 1 + 3 \times 2^{-25} = 1 + 1.5 \times 2^{-24}$

This is exactly halfway between $1 + 2^{-24}$ and $1 + 2 \times 2^{-24}$.

**Rounding to nearest (even)**: Rounds to $1 + 2 \times 2^{-24} = 1 + 2^{-23}$

**IEEE-754**: `0 01111111 00000000000000000000010`

**Hex**: `0x3F800002`

### 2.2.7 7. Number: $1/7$

$1/7 = 0.142857142857\ldots$ (repeating)

Converting to binary: $1/7 = 0.001001001001\ldots_2$ (repeating pattern: 001)

$= 1.001001001\ldots_2 \times 2^{-3}$

- **Sign (S)**: 0
- **Exponent (E)**: $-3 + 127 = 124 = 01111100_2$
- **Mantissa (M)**: `00100100100100100100100`

**IEEE-754**: `0 01111100 00100100100100100100100`

**Hex**: `0x3E124925`

**Warning**

Note: 1/7 cannot be exactly represented - it's a repeating binary fraction

### 2.2.8 8. Number: $2^{-130}$

This is a **denormalized (subnormal) number** because the exponent would be $-130 + 127 = -3$, which is less than 0.

For denormalized numbers:

- Exponent bits = 00000000
- Value = $(-1)^S \times 0.M \times 2^{-126}$

$2^{-130} = 2^{-126} \times 2^{-4} = 0.0001_2 \times 2^{-126}$

- **Sign (S)**: 0
- **Exponent (E)**: 00000000 (denormalized)
- **Mantissa (M)**: 0001 followed by 19 zeros

**IEEE-754**: `0 00000000 00010000000000000000000`

**Hex**: `0x00080000`

### 2.3 Question 2: Product of $a = 4097$ and $b = 8449$

**Given**:

- $a = 4097 = 2^{12} + 1 = 1.00000000001_2 \times 2^{12}$
- $b = 8449 = 2^{13} + 2^8 + 1 = 1.0000001000001_2 \times 2^{13}$

**Step 1: Represent $a$ in single precision**

- Sign: 0
- Exponent: $12 + 127 = 139 = 10001011_2$
- Mantissa: `00000000001` + 20 zeros

**Step 2: Represent $b$ in single precision**

- Sign: 0
- Exponent: $13 + 127 = 140 = 10001100_2$
- Mantissa: `0000001000001` + 18 zeros

**Step 3: Compute exact product**

$$\begin{aligned} a \times b &= (2^{12} + 1) \times (2^{13} + 2^8 + 1) \\ &= 2^{25} + 2^{20} + 2^{12} + 2^{13} + 2^8 + 1 \\ &= 2^{25} + 2^{20} + 2^{13} + 2^{12} + 2^8 + 1 \\ &= 34{,}603{,}009 \end{aligned}$$

In binary: $\texttt{10000100000011000100000001}_2$
**Step 4: Normalize**
$= 1.0000100000011000100000001_2 \times 2^{25}$
**Step 5: Round to 23 bits (single precision)**
The mantissa has more than 23 bits. Keeping first 23 bits after the implicit 1: `00001000000110001000000`
Looking at bit 24: 0, and bit 25: 1. Since we need to round and bit 24 is 0, we round down (keep as is).
**Result** $c$:

- Sign: 0
- Exponent: $25 + 127 = 152 = 10011000_2$
- Mantissa: `00001000000110001000000`

**IEEE-754**: `0 10011000 00001000000110001000000`
**Hex**: `0x4C080620`

## 3 Exercise 3: Problem with Double Rounding

### 3.1 Given

- $x = 1.0110101000000100111100110111111111111111001010111111110_2$
- $y = 1.1010100000001111111111111111111111111111111111111111_2 \times 2^{-39}$

### 3.2 Question 1: Compute $x + y$ exactly

Since $y$ is multiplied by $2^{-39}$, we need to align the exponents.

$x$ has exponent 0 (assuming normalized form with exponent bits decoded). $y$ needs to be shifted right by 39 positions to align with $x$.

**Exact sum**:

$$x = 1.0110101000001001111001101111111111111100101011111110_2$$
$$y = 0.000000000000000000000000000000000000001101010000000111\ldots_2$$

Adding these together:

$$x + y = 1.011010100000100111100110111111111111111000010111111111011\ldots_2$$

The exact result has more than 53 bits of precision in the fractional part.

### 3.3 Question 2: Rounding to double precision, then to single precision

**Step A: Round $x + y$ to double precision (53-bit mantissa)**

The exact sum needs to be rounded to 53 bits after the binary point.

Looking at the 53 bits of mantissa: `0110101000001001111001101111111111111110000101111111011`...

The 54th bit is used for rounding. Applying round-to-nearest:

**Double precision result**: $1.0110101000001001111001101111111111111110000101111111011_2$

After rounding, we keep 53 bits: `0110101000001001111001101111111111111110000101111111011`

**Step B: Round this double precision result to single precision (24-bit mantissa)**

Now we take the double precision result and round to 23 bits (+ 1 implicit bit).

Mantissa in double: `0110101000001001111001101111111111111110000101111111011`

Keep first 23 bits: `01101010000010011110011`

The 24th bit is 0, the 25th bit is 1, and there are more non-zero bits after. Since bit 24 is 0, we round down.

**Single precision result from double rounding**: $1.01101010000010011110011_2$

### 3.4 Question 3: Direct rounding from exact sum to single precision

Taking the exact sum and rounding directly to single precision (23-bit mantissa):

Exact: $1.0110101000001001111001101111111111111110000101111111011\ldots_2$

First 23 bits of mantissa: `01101010000010011110011`

Bit 24: 0, Bit 25: 1, Following bits: have 1's

For round-to-nearest with bit 24 $= 0$, we round down.

**Single precision result from direct rounding**: $1.01101010000010011110011_2$

### 3.5 Observation

**Double Rounding Problem**

**Key insight**: Double rounding can produce different results than direct rounding when:

1. The value is exactly halfway between two representable values at the intermediate precision
2. The tie-breaking rule (round to even) at the intermediate precision affects the final result

In this specific case, the bits after position 53 in the exact sum create a situation where:

- Rounding to double precision first may round up
- Then rounding that double to single may go in a different direction
- Than directly rounding the original to single precision

**This demonstrates the "double rounding problem":**
Rounding a value twice (to intermediate then final precision) can give a different result than rounding directly to the final precision.
This is why FPU operations should avoid double rounding when possible.

## 4 Exercise 4: Computation of Square Root and Division

### 4.1 Question 1: Newton-Raphson for Square Root

**Goal**: Compute $\sqrt{a}$
**Method**: Find the root of $f(x) = x^2 - a$
**Newton-Raphson formula**:

$$x_{n+1} = x_n - \frac{f(x_n)}{f'(x_n)} \tag{1}$$

**Derivatives**:

- $f(x) = x^2 - a$
- $f'(x) = 2x$

**Iteration formula**:

$$\begin{aligned} x_{n+1} &= x_n - \frac{x_n^2 - a}{2x_n} \\ &= x_n - \frac{x_n}{2} + \frac{a}{2x_n} \\ &= \frac{x_n}{2} + \frac{a}{2x_n} \end{aligned}$$

**Simplified**:

$$\boxed{x_{n+1} = \frac{1}{2}\left(x_n + \frac{a}{x_n}\right)} \tag{2}$$

This is the classic formula for computing square roots!
**Example**: $\sqrt{2}$ with $x_0 = 1.5$

- $x_1 = \frac{1}{2}(1.5 + 2/1.5) = \frac{1}{2}(1.5 + 1.333\ldots) = 1.4166\ldots$
- $x_2 = \frac{1}{2}(1.4166\ldots + 2/1.4166\ldots) = 1.4142\ldots$
- $\sqrt{2} \approx 1.41421356\ldots$

### 4.2 Question 2: Convergence Rate

Newton-Raphson has **quadratic convergence**: the number of correct bits approximately doubles at each iteration.

**Given**: $x_0$ has 4 bits of accuracy

| Iteration | Bits of accuracy |
|---|---|
| 0 | 4 |
| 1 | $\approx 8$ |
| 2 | $\approx 16$ |
| 3 | $\approx 32$ |
| 4 | $\approx 64$ |

**For 24 bits (single precision)**:

- Start: 4 bits
- After iteration 1: $\approx 8$ bits
- After iteration 2: $\approx 16$ bits
- After iteration 3: $\approx 32$ bits ✓

**Answer**: **3 iterations** needed for 24-bit accuracy

**For 53 bits (double precision)**:

- Start: 4 bits
- After iteration 1: $\approx 8$ bits
- After iteration 2: $\approx 16$ bits
- After iteration 3: $\approx 32$ bits
- After iteration 4: $\approx 64$ bits ✓

**Answer**: **4 iterations** needed for 53-bit accuracy

### 4.3 Question 3: Division using Newton-Raphson

**Goal**: Compute $a/b$

**Method**: Instead of dividing, compute $1/b$ and then multiply by $a$.

**Find root of**: $f(x) = 1/x - b = 0$, which means $x = 1/b$

Alternatively, use $f(x) = b - 1/x = 0$

**Better formulation**: $f(x) = 1 - bx = 0$

- This avoids division in the iteration itself!
- $f'(x) = -b$

**Newton-Raphson formula**:

$$\begin{aligned} x_{n+1} &= x_n - \frac{1 - bx_n}{-b} \\ &= x_n + \frac{1 - bx_n}{b} \end{aligned}$$

**Simplified**:

$$\boxed{x_{n+1} = x_n(2 - bx_n)} \tag{3}$$

This formula computes $1/b$ using only multiplications and subtractions!
**Final step**: Once we have $1/b$, compute $a/b = a \times (1/b)$
**Algorithm**:

1. Choose initial approximation $x_0 \approx 1/b$
2. Iterate: $x_{n+1} = x_n(2 - bx_n)$ until convergence
3. Result: $a/b \approx a \times x_n$

**Example**: Compute 7/3

- Find 1/3 first using $b = 3$
- $x_0 = 0.3$ (initial guess)
- $x_1 = 0.3(2 - 3 \times 0.3) = 0.3(2 - 0.9) = 0.33$
- $x_2 = 0.33(2 - 3 \times 0.33) = 0.33(2 - 0.99) = 0.3333$
- Then: $7/3 = 7 \times 0.3333\ldots \approx 2.333\ldots$

**Advantage**

This method avoids hardware division circuits entirely, using only multiplication and subtraction!

## 5 Summary

### 5.1 Key Takeaways

1. **Integer Representation**: Two's complement is the standard due to its simplicity and no duplicate zero
2. **IEEE-754 Floating-Point**: Understanding the bit layout (sign, exponent, mantissa) is crucial for:
   - Recognizing representable numbers
   - Understanding rounding behavior
   - Identifying subnormal numbers
3. **Double Rounding Problem**: Demonstrates that rounding precision matters and can affect final results
4. **Newton-Raphson**:
   - Quadratic convergence (bits double each iteration)
   - Can be adapted for both square root and division
   - Division method avoids hardware division circuits

### 5.2 Important Formulas

**Square Root**:

$$x_{n+1} = \frac{1}{2}\left(x_n + \frac{a}{x_n}\right) \tag{4}$$

**Division** (computing $1/b$):

$$x_{n+1} = x_n(2 - bx_n) \tag{5}$$

---

*Tutorial solved with Claude by Giulia Lionetti - Sorbonne Université*