

# Tutorial 1 - Introduction to Computer Arithmetic

## Solutions to Exercise 5 (Parts 1 and 3)

Floating-point Arithmetic and Error Analysis (AFAE)

Master of Computer Science 2nd year - CCA  
Year 2025/2026

### Exercise 5: Floating-Point Arithmetic

#### IEEE-754 Double Precision Format Reminder

IEEE-754 double precision (binary64) format consists of:

- **1 bit** for sign ( $s$ )
- **11 bits** for biased exponent ( $e$ ), bias =  $2^{11-1} - 1 = 1023$
- **52 bits** for mantissa/significand ( $f$ )

Memory layout (64 bits total):

Sign (1 bit)	Exponent (11 bits)	Mantissa (52 bits)
63	62–52	51–0

#### Value Interpretation

For **normalized numbers** ( $1 \leq e \leq 2046$ ):

$$\text{value} = (-1)^s \times 2^{e-1023} \times (1.f) \quad (1)$$

For **subnormal numbers** ( $e = 0, f \neq 0$ ):

$$\text{value} = (-1)^s \times 2^{-1022} \times (0.f) \quad (2)$$

#### Special values:

- Zero:  $e = 0, f = 0$  (signed:  $\pm 0$ )
- Infinity:  $e = 2047, f = 0$  (signed:  $\pm \infty$ )
- NaN:  $e = 2047, f \neq 0$

# 1 Problem 1: Decode IEEE-754 Double Precision Numbers

## Problem Statement

Decode by hand the following double precision numbers given in hexadecimal:

1. 0x3ff00000000000000
2. 0x0000000000000000
3. 0x8000000000000000
4. 0x0000000000000001
5. 0xc00a000000000000
6. 0x3ff6a09e667f3bcd
7. 0x7ff0000000000000

## Solution

1. 0x3ff00000000000000

### Binary representation:

$$\begin{aligned} \text{Hex: } & 3ff0\ 0000\ 0000\ 0000_{16} \\ \text{Binary: } & 0011\ 1111\ 1111\ 0000\ \dots 0000_2 \end{aligned}$$

### Decomposition:

- Sign bit:  $s = 0$  (positive)
- Exponent bits:  $011\ 1111\ 1111_2 = 1023_{10}$
- Mantissa bits: all zeros

### Calculation:

$$e = 1023 \tag{3}$$

$$\text{True exponent} = e - 1023 = 0 \tag{4}$$

$$\text{Mantissa} = 1.0 \text{ (implicit leading 1, all fraction bits are 0)} \tag{5}$$

$$\text{Value} = (-1)^0 \times 2^0 \times 1.0 = \boxed{1.0} \tag{6}$$

### Answer

$$0x3ff00000000000000 = \boxed{1.0}$$

2. 0x0000000000000000

### Binary representation:

All bits are 0

**Decomposition:**

- Sign bit:  $s = 0$
- Exponent bits: all zeros ( $e = 0$ )
- Mantissa bits: all zeros ( $f = 0$ )

**Special case:** When  $e = 0$  and  $f = 0$ , this represents **zero**.

Answer

$0x0000000000000000 = +0.0$  (positive zero)

3.  $0x8000000000000000$

**Binary representation:**

Hex: 8000 0000 0000 0000<sub>16</sub>  
Binary: 1000 0000 0000 0000 ...0000<sub>2</sub>

**Decomposition:**

- Sign bit:  $s = 1$  (negative)
- Exponent bits: all zeros ( $e = 0$ )
- Mantissa bits: all zeros ( $f = 0$ )

**Special case:** When  $e = 0$  and  $f = 0$ , with sign bit = 1, this represents **negative zero**.

Answer

$0x8000000000000000 = -0.0$  (negative zero)

4.  $0x0000000000000001$

**Binary representation:**

Hex: 0000 0000 0000 0001<sub>16</sub>  
Binary: 0000 ...0001<sub>2</sub>

**Decomposition:**

- Sign bit:  $s = 0$  (positive)
- Exponent bits: all zeros ( $e = 0$ )
- Mantissa bits: only the last bit is 1

**Calculation (Subnormal):** Since  $e = 0$  but  $f \neq 0$ , this is a **subnormal number**.

$$\text{Mantissa} = 0.\underbrace{00\dots01}_{52 \text{ bits}} = 2^{-52} \quad (7)$$

$$\text{Value} = (-1)^0 \times 2^{-1022} \times 2^{-52} \quad (8)$$

$$= 2^{-1074} \quad (9)$$

$$\approx 4.94 \times 10^{-324} \quad (10)$$

This is the **smallest positive subnormal number** in IEEE-754 double precision.

Answer

$$0x0000000000000001 = \boxed{2^{-1074} \approx 4.94 \times 10^{-324}}$$

This is the smallest positive representable number (machine epsilon for subnormals).

## 5. 0xc00a000000000000

**Binary representation:**

Hex: c00a 0000 0000 0000<sub>16</sub>

Binary: 1100 0000 0000 1010 0000 ...0000<sub>2</sub>

**Decomposition:**

- Sign bit:  $s = 1$  (negative)
- Exponent bits: 100 0000 0000<sub>2</sub> = 1024<sub>10</sub>
- Mantissa bits: 0000 1010 0000...<sub>2</sub>

**Calculation:**

$$e = 1024 \quad (11)$$

$$\text{True exponent} = e - 1023 = 1 \quad (12)$$

$$\text{Mantissa} = 1.0000101000\dots_2 = 1 + \frac{1}{2^5} + \frac{1}{2^7} \quad (13)$$

$$= 1 + 0.03125 + 0.0078125 \quad (14)$$

$$= 1 + \frac{5}{128} = 1 + \frac{10}{256} = \frac{266}{256} = 1.0390625 \quad (15)$$

$$\text{Value} = (-1)^1 \times 2^1 \times 1.0390625 \quad (16)$$

$$= -2 \times 1.0390625 \quad (17)$$

$$= -2.078125 \quad (18)$$

$$= \boxed{-\frac{133}{64}} \quad (19)$$

Answer

$$0xc00a000000000000 = -2.078125 = -\frac{133}{64}$$

## 6. 0x3ff6a09e667f3bcd

**Binary representation:**

Hex: 3ff6 a09e 667f 3bcd<sub>16</sub>

**Decomposition:**

- Sign bit:  $s = 0$  (positive)
- Exponent bits: 011 1111 1111<sub>2</sub> = 1023<sub>10</sub>
- Mantissa bits: 0110101000001001...

**Calculation:**

$$e = 1023 \quad (20)$$

$$\text{True exponent} = e - 1023 = 0 \quad (21)$$

$$\text{Mantissa in hex} = 1.6a09e667f3bcd_{16} \quad (22)$$

Converting the fractional part:

$$6a09e667f3bcd_{16} = \frac{6a09e667f3bcd_{16}}{2^{52}} \quad (23)$$

$$\approx 0.41421356237309515 \quad (24)$$

Therefore:

$$\text{Value} = 2^0 \times (1 + 0.41421356237309515) \quad (25)$$

$$= 1.41421356237309515 \quad (26)$$

$$\approx \boxed{\sqrt{2}} \quad (27)$$

Answer

0x3ff6a09e667f3bcd  $\approx 1.4142135623730951 \approx \sqrt{2}$

This is the IEEE-754 double precision representation of  $\sqrt{2}$ .

## 7. 0x7ff0000000000000

**Binary representation:**

Hex: 7ff0 0000 0000 0000<sub>16</sub>

Binary: 0111 1111 1111 0000 ...0000<sub>2</sub>

**Decomposition:**

- Sign bit:  $s = 0$  (positive)
- Exponent bits: 111 1111 1111<sub>2</sub> = 2047<sub>10</sub> (all ones)
- Mantissa bits: all zeros

**Special case:** When  $e = 2047$  (all exponent bits are 1) and  $f = 0$ , this represents **infinity**.

Answer

$0x7ff0000000000000 = +\infty$  (positive infinity)

## Summary Table

Hexadecimal	Value	Type
0x3ff0000000000000	1.0	Normalized
0x0000000000000000	+0.0	Zero
0x8000000000000000	-0.0	Signed zero
0x0000000000000001	$2^{-1074} \approx 4.94 \times 10^{-324}$	Subnormal
0xc00a000000000000	-2.078125	Normalized
0x3ff6a09e667f3bcd	$\sqrt{2} \approx 1.41421$	Normalized
0x7ff0000000000000	$+\infty$	Infinity

Table 1: Decoded IEEE-754 double precision values

## 2 Problem 3: Decompose Double Precision by Iteration

### Problem Statement

By successive iterations, propose a way to decompose a double precision number  $x > 2^{-1021}$  into  $E \in \mathbb{Z}$  and  $m \in \mathbb{F}$ ,  $1 \leq m < 2$  such that  $x = 2^E \cdot m$ .

### Solution

#### Understanding the Goal

We want to find the **normalized scientific notation** of a floating-point number:

$$x = 2^E \times m \quad \text{where } 1 \leq m < 2 \quad (28)$$

This decomposition is also known as finding the **exponent** and **mantissa** of the number.

#### Algorithm Overview

The idea is to iteratively:

1. Determine if  $x$  is too large ( $x \geq 2$ ) or too small ( $x < 1$ )
2. Scale  $x$  by powers of 2 to bring it into the range  $[1, 2)$
3. Track the exponent as we scale

#### Method 1: Binary Search Approach

**Initial bounds:** Since  $x > 2^{-1021}$  is a normalized double precision number, we know:

$$2^{-1022} \leq x < 2^{1024} \quad (29)$$

Thus:  $-1022 \leq E \leq 1023$ .

#### Algorithm:

1. Initialize:  $E = 0$ ,  $m = x$

2. While  $m \geq 2$ :

- $m \leftarrow m/2$
- $E \leftarrow E + 1$

3. While  $m < 1$ :

- $m \leftarrow m \times 2$
- $E \leftarrow E - 1$

4. Return  $(E, m)$

```

C Implementation:
1 void decompose_iterative_v1(double x, int *E, double *m) {
2     *E = 0;
3     *m = x;
4
5     // Scale down if too large
6     while (*m >= 2.0) {
7         *m /= 2.0;
8         (*E)++;
9     }
10
11    // Scale up if too small
12    while (*m < 1.0) {
13        *m *= 2.0;
14        (*E)--;
15    }
16}

```

**Complexity:** In the worst case, we need  $O(\log_2 |E|)$  iterations, which is at most about 10-11 iterations for double precision.

## Method 2: Optimized Binary Search

We can optimize by using larger jumps initially, then refining.

```

1 void decompose_iterative_v2(double x, int *E, double *m) {
2     *E = 0;
3     *m = x;
4
5     // Coarse adjustment by powers of 2^k
6     for (int k = 10; k >= 0; k--) {
7         int step = 1 << k; // 2^k
8         double scale = 1.0;
9
10        // Compute 2^step
11        for (int i = 0; i < step; i++) {
12            scale *= 2.0;
13        }
14
15        // Scale down if possible
16        while (*m >= scale) {
17            *m /= scale;
18            *E += step;
19        }
20
21        // Scale up if needed
22        while (*m < 1.0 && *m * scale < 2.0) {
23            *m *= scale;
24            *E -= step;
25        }
26    }
27
28    // Final adjustment

```

```

29     while (*m >= 2.0) {
30         *m /= 2.0;
31         (*E)++;
32     }
33
34     while (*m < 1.0) {
35         *m *= 2.0;
36         (*E)--;
37     }
38 }
```

### Method 3: Using frexp (Standard Library)

For reference, the C standard library provides `frexp` which does exactly this:

```

1 #include <math.h>
2
3 void decompose_using_frexp(double x, int *E, double *m) {
4     *m = frexp(x, E);
5     // Note: frexp returns m in [0.5, 1), so we adjust:
6     *m *= 2.0;
7     (*E)--;
8 }
```

### Verification Examples

**Example 1:**  $x = 10.0$

$$10.0 = 2^E \times m \quad (30)$$

$$\text{Iterations: } 10.0 \rightarrow 5.0 \rightarrow 2.5 \rightarrow 1.25 \quad (31)$$

$$E = 3 \quad (32)$$

$$m = 1.25 \quad (33)$$

$$\text{Check: } 2^3 \times 1.25 = 8 \times 1.25 = 10.0 \quad \checkmark \quad (34)$$

**Example 2:**  $x = 0.125$

$$0.125 = 2^E \times m \quad (35)$$

$$\text{Iterations: } 0.125 \rightarrow 0.25 \rightarrow 0.5 \rightarrow 1.0 \quad (36)$$

$$E = -3 \quad (37)$$

$$m = 1.0 \quad (38)$$

$$\text{Check: } 2^{-3} \times 1.0 = 0.125 \quad \checkmark \quad (39)$$

**Example 3:**  $x = \sqrt{2} \approx 1.41421$

$$\sqrt{2} \approx 1.41421 \quad (40)$$

Already in range [1, 2) (41)

$$E = 0 \quad (42)$$

$$m = 1.41421 \quad (43)$$

$$\text{Check: } 2^0 \times 1.41421 = 1.41421 \quad \checkmark \quad (44)$$

## Complete Working Example

```
1 #include <stdio.h>
2
3 void decompose_iterative(double x, int *E, double *m) {
4     *E = 0;
5     *m = x;
6
7     // Scale down if too large
8     while (*m >= 2.0) {
9         *m /= 2.0;
10        (*E)++;
11    }
12
13    // Scale up if too small
14    while (*m < 1.0) {
15        *m *= 2.0;
16        (*E)--;
17    }
18}
19
20 int main() {
21     double test_values[] = {1.0, 10.0, 0.125, 3.14159, 1024.0};
22     int n = sizeof(test_values) / sizeof(test_values[0]);
23
24     for (int i = 0; i < n; i++) {
25         int E;
26         double m;
27         decompose_iterative(test_values[i], &E, &m);
28         printf("x = %.6f = 2^%d * %.6f\n", test_values[i], E, m);
29
30         // Verify
31         double reconstructed = m;
32         for (int j = 0; j < E; j++) reconstructed *= 2.0;
33         for (int j = 0; j > E; j--) reconstructed /= 2.0;
34         printf("Verification: 2^%d * %.6f = %.6f\n\n",
35               E, m, reconstructed);
36     }
37
38     return 0;
39 }
```

### Output:

```
x = 1.000000 = 2^0 * 1.000000
Verification: 2^0 * 1.000000 = 1.000000

x = 10.000000 = 2^3 * 1.250000
Verification: 2^3 * 1.250000 = 10.000000

x = 0.125000 = 2^-3 * 1.000000
Verification: 2^-3 * 1.000000 = 0.125000

x = 3.141590 = 2^1 * 1.570795
Verification: 2^1 * 1.570795 = 3.141590

x = 1024.000000 = 2^10 * 1.000000
Verification: 2^10 * 1.000000 = 1024.000000
```

### Key Points

- The iterative approach repeatedly multiplies or divides by 2
- Complexity is  $O(\log |E|)$  which is typically 10-11 iterations max
- Works for all normalized numbers ( $x > 2^{-1021}$ )
- Division/multiplication by 2 is exact in floating-point (no rounding error)
- Can be optimized using larger steps (binary search on exponent)

### Why This Works

1. **Multiplication/division by 2 is exact:** In IEEE-754, multiplying or dividing by 2 only changes the exponent, not the mantissa. This is exact (no rounding error).
2. **Convergence guarantee:** Each iteration either:
  - Halves the number if  $m \geq 2$ , bringing it closer to  $[1, 2]$
  - Doubles the number if  $m < 1$ , bringing it closer to  $[1, 2]$
3. **Termination:** Since exponents are bounded ( $-1022 \leq E \leq 1023$ ), the algorithm terminates in finite time.