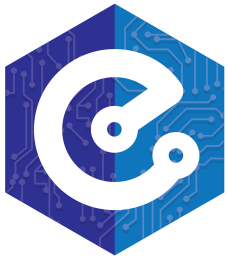
Ho Chi Minh University of Technology
Department of Electronics



DIGITAL SYSTEM DESIGN AND VERIFICATION
(EE3213)

Big Project 1
Floating Point Adder and Subtractor

Instructor: Hieu Nguyen

Email: hieunt@hcmut.edu.vn

October 27, 2025

Contents

1 Introduction

1.1 Floating-point Format

While integers offer precise representations of numerical values, they have two key limitations: they cannot express fractional numbers and have a restricted range. For instance, an n-bit binary string can represent:

- values from 0 to $2^n - 1$ if they are in unsigned integer format.
- values from -2^{n-1} to $2^{n-1} - 1$ if they are in signed integer two's complement format.

Then, doing arithmetic on the integer format also has these limitations. Floating-point arithmetic addresses both issues, though it does so by sacrificing some accuracy and, in some processors, computational speed. The main benefit of floating-point representation over fixed-point or integer formats is its ability to represent a much broader range of values.

A floating-point format is a way to represent real numbers in computers, using a sign, an exponent, and a mantissa (also called a significand), similar to scientific notation. This allows for a wide range of values, from very small to very large, by trading off some precision for a broader dynamic range. Standard formats include the IEEE 754 single-precision (32-bit) and double-precision (64-bit) standards. It also has 8-bit and 16-bit floating-point formats, but they are less commonly used.

In this project, we mainly focus on the single-precision (32-bit) standards. We have a sign bit, 8 bits for the exponent, and the remaining 23 bits for the mantissa.

S	E: 8-bit Exponent	M: 23-bit Mantissa

Using this format, the floating-point numbers are of the form:

$$(-1)^S * 1.M * 2^{(E-BIAS)}$$

- S: is the sign bit, which determines whether the number is positive or negative.
- M: holds the significand bits of the floating point number. More bits in the significand provide greater accuracy.
- E: is the exponent that (1+M) is raised to. More bits for exponents increase the range.

The exponents are biased to make sorting easier. With a 32-bit floating-point format, the bias value is 127, and E values range from 0 to 255; hence, the number is in the $[2^{-128} : 2^{128}]$ range.

Example 1.1 *Convert 2025 to 32-bit floating-point format*

- *Step 1 - Convert to binary:* $2025_{10} = 11111101001_2$
- *Step 2 - Normalize the binary number: Write it in normalized scientific notation (base 2)*

$$11111101001 * 2^0 = 1.1111101001 * 2^{10}$$

- *Step 3 - Compute each field:*
  Sign bit: 0 *(since 2025 is positive)*
  Exponent: $10 + 127 = 137 \rightarrow 137_{10} = 10001001_2$
  Mantissa: the fractional part of $1.1111101001 \rightarrow 11111010010000000000000$ *(padded to 23 bits)*
- *Step 4 - Combine all fields:* 0 10001001 11111010010000000000000

Example 1.2 *Convert -25.125 to 32-bit floating-point format*

- *Step 1 - Convert to binary:* $25.125_{10} = 11001.001_2$
- *Step 2 - Normalize the binary number: Write it in normalized scientific notation (base 2)*

$$11001.001 * 2^0 = 1.1001001 * 2^4$$

- *Step 3 - Compute each field:*
  Sign bit: 1 *(since the number is negative)*
  Exponent: $4 + 127 = 131 \rightarrow 131_{10} = 10000011_2$
  Mantissa: the fractional part of $1.1001001 \rightarrow 10010010000000000000000$ *(padded to 23 bits)*
- *Step 4 - Combine all fields:* 1 10000011 10010010000000000000000

Example 1.3 *Convert 0.03125 to 32-bit floating-point format*

- *Step 1 - Convert to binary:* $0.03125_{10} = 0.00001_2$
- *Step 2 - Normalize the binary number: Write it in normalized scientific notation (base 2)*

$$0.00001 * 2^0 = 1.0 * 2^{-5}$$

- *Step 3 - Compute each field:*
  Sign bit: 0 *(since the number is positive)*
  Exponent: $-5 + 127 = 122 \rightarrow 122_{10} = 01111010_2$
  Mantissa: the fractional part of $1.0 \rightarrow 00000000000000000000000$ *(padded to 23 bits)*
- *Step 4 - Combine all fields:* 0 01111010 00000000000000000000000

Example 1.4 *Convert* $0x40E40000$ *in 32-bit floating-point format to decimal*

- *Step 1 - Write the hex in binary:* $40E40000_{16} = 01000000111001000000000000000000_2$
- *Step 2 - Split into IEEE 754 fields:*
  Sign: 0 *- positive*
  Exponent: $10000001_2 = 129_{10}$
  Mantissa: 11001000000000000000000
- *Step 3 - Compute value:*

$$(-1)^0 * 1.11001000000000000000000_2 * 2^{(129-127)} = +1.78125 * 2^2 = 7.125$$

Example 1.5 *Convert* $0x80D50000$ *in 32-bit floating-point format to decimal*

- *Step 1 - Write the hex in binary:* $80D50000_{16} = 10000000110101010000000000000000_2$
- *Step 2 - Split into IEEE 754 fields:*
  Sign: 1 *- negative*
  Exponent: $00000001_2 = 1_{10}$
  Mantissa: 10101010000000000000000
- *Step 3 - Compute value:*

$$(-1)^1 * 1.10101010000000000000000_2 * 2^{(1-127)} = -1.6640625 * 2^{-126}$$

1.2 Rounding in Floating-point Format

In the previous examples, we have seen how to convert from decimal to single-precision format. In examples 1.2 and 1.3, when converting from a decimal to a binary number, the result is not an infinite number. When the result is infinite, we have to round it because the mantissa can store 23 bits. There are many rounding modes, including rounding down, rounding up, rounding to nearest, and rounding to even. In this project, we mainly focus on the nearest rounding mode. Given the bit string before reduction has the form:

$$1.\boxed{m_1 m_2 \dots m_{23}}\underbrace{m_{24} m_{25} \dots}_{\text{dropped part}}$$

- If what is left over is $< 1/2$: just drop the bits beyond position 23

$$1.\boxed{m_1 m_2 \dots m_{23}}\ \underbrace{0 \dots\dots\dots}_{\substack{\text{round down by} \\ \text{truncating}}}$$

- If what is left over is $> 1/2$: add 1 to bit 23 carrying as necessary

$$1.\boxed{m_1 m_2 \dots m_{23}} \underbrace{1\dots\dots\dots}_{\text{round up by adding 1}}$$

- If what is left over is $= 1/2$:

$$1.\boxed{m_1 m_2 \dots m_{23}}1 \underbrace{0\dots\dots\dots}_{\text{all 0's}}$$

If $m_{23} = 1$: add 1 to m_{23} (round up)
If $m_{23} = 0$: only cut parts from m_{24} onwards (round down)

These bits are also called the Guard bit (m_{23}), the Round bit (m_{24}), and the Sticky bit (OR of remaining bits). We summarize the previous rules into a Table 1:

GRS	Increase?
$x0x$	N
010	N
110	Y
$x11$	Y

Table 1: Rounding Rules

Example 1.6 *Convert 9.6781 to 32-bit floating-point format*

- *Step 1 - Convert to binary:* $9.6781_{10} \approx 1001.10101101100101111111\dots_2$
- *Step 2 - Normalize the binary number: Write it in normalized scientific notation (base 2)*

$$1001.10101101100101111111\dots * 2^0 = 1.00110101101100101111111\dots * 2^3$$

- *Step 3 - Compute each field:*
  Sign bit: 0 *(since the number is positive)*
  Exponent: $3 + 127 = 130 \rightarrow 130_{10} = 10000010_2$
  Mantissa: from the normalized form, the fraction bits start: 00110101101100101111111110110...
  Take the first 23 bits (then apply rounding rule):
  First 23 bits: 00110101101100101111111
  Guard bit - bit 23th: 1
  Round bit - 24th: 0
  Sticky bit (OR of remaining bits): 1
  Because the Round bit is 0, no rounding up is required, so the 23-bit mantissa remains: 00110101101100101111111
- *Step 4 - Combine all fields:* 0 10000010 00110101101100101111111

Example 1.7 *Convert -0.1 to 32-bit floating-point format*

- *Step 1 - Convert to binary:* $0.1_{10} \approx 0.0001100110011001100110011001100110011\dots_2$
- *Step 2 - Normalize the binary number: Write it in normalized scientific notation (base 2)*

$$0.0001100110011001100110011001100110011\dots * 2^0 = 1.1001100110011001100110011\dots * 2^{-4}$$

- *Step 3 - Compute each field:*
  Sign bit: 1 *(since the number is negative)*
  Exponent: $-4 + 127 = 123 \rightarrow 123_{10} = 01111011_2$
  Mantissa: from the normalized form, the fraction bits start: 10011001100110011001100110...
  Take the first 23 bits (then apply rounding rule):
  First 23 bits: 10011001100110011001100
  Guard bit - bit 23th: 0
  Round bit - 24th: 1
  Sticky bit (OR of remaining bits): 1
  Because GRS is 011, rounding up is required, so the 23-bit mantissa remains: 10011001100110011001101
- *Step 4 - Combine all fields:* 1 01111011 10011001100110011001101

1.3 Special Cases

Two terms appear in the representation of a 32-bit floating-point number: **overflow** and **underflow**. These two terms describe what happens when a computation produces a number that is too large or too small to be represented within the format's range.

Overflow: occurs when the magnitude of the computed value is too large to be represented.
Example: its exponent exceeds the maximum exponent field (255) for normalized numbers.
Max representable normalized value:

$$1.11111111111111111111111_2 * 2^{127} \approx 3.4028235 * 10^{38}$$

If a result exceeds this value, it **overflows**.

- The exponent field becomes all ones ($11111111_2 = 255$).
- Mantissa $= 0 \rightarrow$ Infinity ($\pm\infty$)
- Mantissa $\neq 0 \rightarrow$ NaN (Not a Number)

Underflow: occurs when the magnitude of the computed value is too small (close to zero) to be represented as a normalized number.
Example: its exponent drops below -127 for normalized numbers.
There are two possibilities:

- Representable as subnormal $\rightarrow$ exponent field $= 0$, leading 1 is not assumed

$$(-1)^S * (0.M) * 2^{-126}$$

  These are gradually underflowed numbers (very tiny values).
- Too small even for subnormal $\rightarrow$ result rounded to zero (± 0).

The graphical visualization showing the representable range is shown in Figure 1.

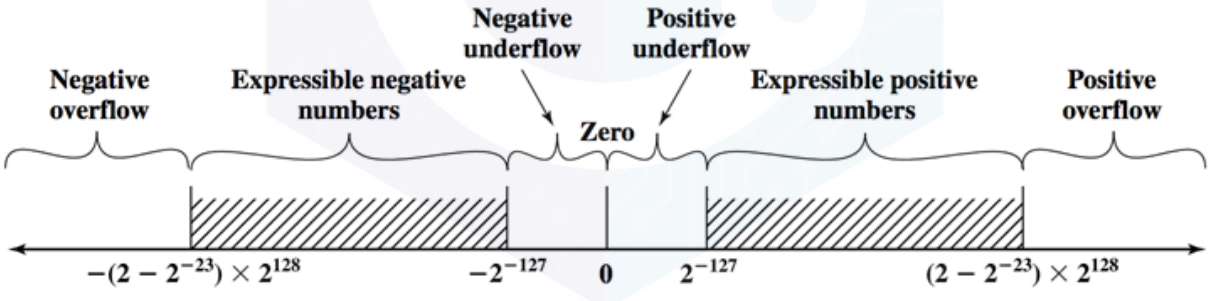


Figure 1: The Representable Range of 32-bit Floating-point Numbers

In short, a 32-bit floating-point format has several special cases, as shown in Table 2.

Exponent (E)	Mantissa (M)	Meaning	Example (Hex)	Notes
E = 0	M = 0	Zero	$+0 = 0x00000000$ $-0 = 0x80000000$	Sign bit distinguishes +0 and −0
E = 0	M $\neq$ 0	Subnormal (Denormalized)	$0x80000600$	
E = 255	M = 0	Infinity ($\pm\infty$)	$+\infty = 0x7F800000$ $-\infty = 0xFF800000$	Represents overflow or division by zero
E = 255	M $\neq$ 0	NaN (Not a Number)	$0x7FC00000$	Result of undefined ops like $0/0$ or $\infty - \infty$

Table 2: Special Cases in 32-bit floating-point format

1.4 Adding two floating-point numbers

To add two floating-point numbers, you first align their exponents by shifting the mantissa of the number with the smaller exponent to the right. Second, you have to add the mantissas. After these steps, you normalize the result by adjusting the exponent and mantissa. Potential issues during this process include overflow, underflow, and truncation errors. The detailed addition of two floating-point numbers is shown in Figure 2.

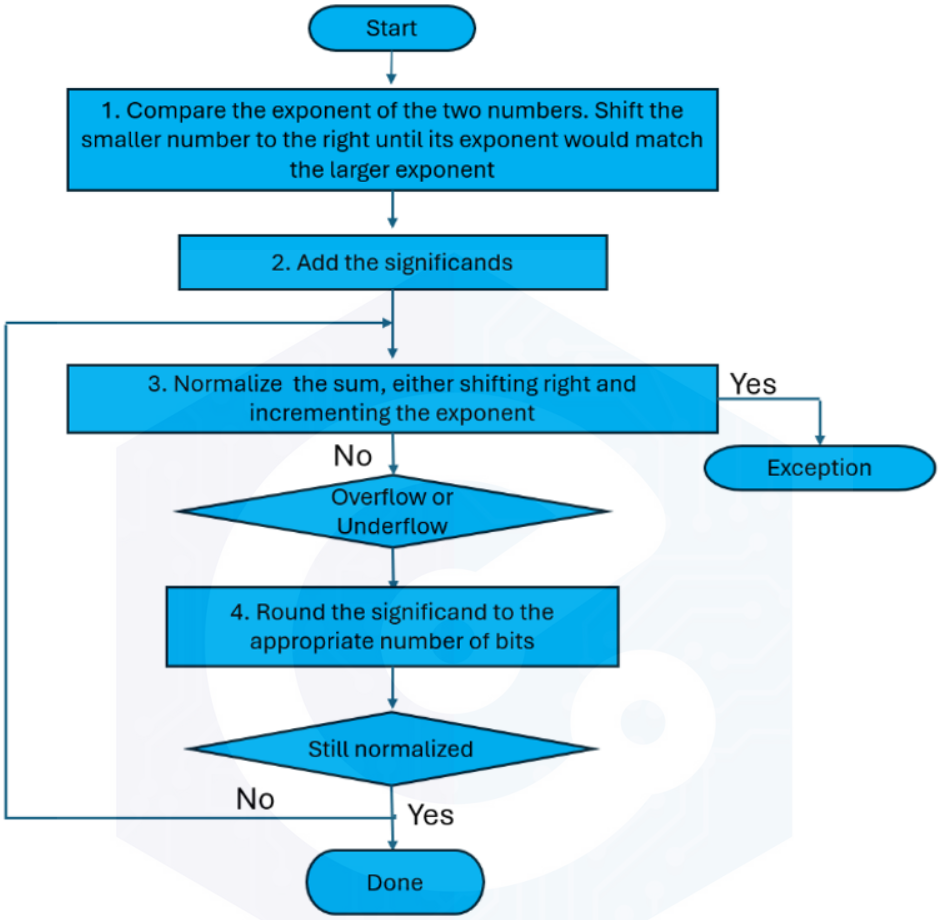


Figure 2: The Adding Arithmetic Algorithm

Detailed steps for adding floating-point numbers:

- **Extract and unpack:** Separate the sign, exponent, and mantissa from both floating-point numbers.
- **Align exponents:**
  - Find the number with the smaller exponent.
  - Shift its mantissa to the right until its exponent matches the larger one. The shift amount determines how many bits are shifted to the right.
- **Add mantissas:** Add the two mantissas together. If the signs are different, you will perform an addition with 2's complement numbers, which can result in a negative number.
- **Normalize the result:**
  - If adding the mantissas results in a value that is not normalized (for instance, a carry-out or a number with a leading zero), adjust the exponent and shift the mantissa to normalize it.
  - If the sum is too large or too small for the available bits, this is an overflow or underflow error.
- **Round the result:** Round the final result to fit the required precision, which can introduce inaccuracies.

Example 1.8 *Demonstrate 32-bit floating point addition step by step for 28 + 3.75*

- *Step 1 - Convert both numbers to 32-bit floating point format:*
  $28_{10} = 11100_2 = 1.1100_2 * 2^4 = 1.1100_2 * 2^{131-127} \rightarrow 0\ 10000011\ 11000000000000000000000$
  $3.75_{10} = 11.11_2 = 1.111_2 * 2^1 = 1.111_2 * 2^{128-127} \rightarrow 0\ 10000000\ 11100000000000000000000$
- *Step 2 - Align exponents:*
  - Larger exponent = 131 (from 28). Difference = 131 - 128 = 3.
  - Shift the smaller number's significand (3.75) right by 3 bits:
  $1.11100000000000000000000_2 * 2^1 = 0.00111100000000000000000_2 * 2^4$
- *Step 3 - Add mantissas:*
  Before performing this step, we add two 0 bits at the beginning of the bit sequence.
  $28_{10} \rightarrow 001.11000000000000000000000$
  $3.75_{10} \rightarrow 000.00111100000000000000000$

$$\begin{array}{r} 001.11000000000000000000000 \\ +000.00111100000000000000000 \\ \hline 001.11111100000000000000000 \end{array}$$

  The first bit is 0, which means the sign of the result is positive.
  The result is: $001.11111100000000000000000_2 * 2^4$
- *Step 4 - Normalize the result: The result must be normalized to have one leading 1.*
  The sum already has the form $1.xxxxxx$*, so it is normalized — no exponent change.*
  The result's unbiased exponent = 4, then the biased exponent $= 131_{10} = 10000011_2$*.*
- *Step 5 - Round the result: There are no nonzero bits beyond the 23 stored fraction bits here (we padded zeros), so no rounding.*
  The result in 32-bit floating-point format is: 0 10000011 11111100000000000000000
  Decimal value: 31.75

Example 1.9 *Demonstrate 32-bit floating point addition step by step for 15 + 9.125*

- *Step 1 - Convert both numbers to 32-bit floating point format:*
  $15_{10} = 1111_2 = 1.111_2 * 2^3 = 1.111_2 * 2^{130-127} \rightarrow 0\ 10000010\ 11100000000000000000000$
  $9.125_{10} = 1001.001_2 = 1.001001_2 * 2^3 = 1.001001_2 * 2^{130-127} \rightarrow 0\ 10000010\ 00100100000000000000000$
- *Step 2 - Align exponents: Exponents are equal (biased value:* $130_{10} = 10000010_2$ *and unbiased value:* 3_{10}*), so no shifting is necessary.*
- *Step 3 - Add mantissas:*
  Before performing this step, we add two 0 bits at the beginning of the bit sequence.
  $15_{10} \rightarrow 001.11100000000000000000000$
  $9.125_{10} \rightarrow 001.00100100000000000000000$

$$\begin{array}{r} 001.11100000000000000000000 \\ +001.00100100000000000000000 \\ \hline 011.00000100000000000000000 \end{array}$$

  The first bit is 0, which means the sign of the result is positive.
  The result is: $011.00000100000000000000000_2 * 2^3$
- *Step 4 - Normalize the result: The result must be normalized to have one leading 1.*
  $011.00000100000000000000000_2 * 2^3 = 1.10000010000000000000000_2 * 2^4$
  The result's unbiased exponent = 4, then the biased exponent $= 131_{10} = 10000011_2$*.*
- *Step 5 - Round the result: There are no nonzero bits beyond the 23 stored fraction bits here (we padded zeros), so no rounding.*
  The result in 32-bit floating-point format is: 0 10000011 10000010000000000000000
  Decimal value: 24.125

Example 1.10 *Demonstrate 32-bit floating point addition step by step for 13 + (-0.0625)*

- *Step 1 - Convert both numbers to 32-bit floating point format:*
  $13_{10} = 1101_2 = 1.101_2 * 2^3 = 1.101_2 * 2^{130-127} \rightarrow 0\ 10000010\ 10100000000000000000000$
  $-0.0625_{10} = -0.0001_2 = -1.0_2 * 2^{-4} = -1.0_2 * 2^{123-127} \rightarrow 1\ 01111011\ 00000000000000000000000$

- *Step 2 - Align exponents:*
  - Larger exponent = 130 (from 13). Difference = 130 - 123 = 7.
  - Shift the smaller number's significand (-0.0625) right by 7 bits:
  $1.00000000000000000000000_2 * 2^{-4} = 0.00000010000000000000000_2 * 2^3$
- *Step 3 - Add mantissas:*
  Before performing this step, we add two 0 bits at the beginning of the bit sequence.
  $13_{10} \rightarrow 001.10100000000000000000000$
  $-0.625_{10} \rightarrow 000.00000010000000000000000$
  Because -0.0625 is negative, take its 2's complement:
  $000.00000010000000000000000 \rightarrow 111.11111110000000000000000$

$$\begin{array}{r} 001.10100000000000000000000 \\ +111.11111110000000000000000 \\ \hline (1)001.10011110000000000000000 \end{array}$$

  There's a carry out, which we drop because we're working in fixed-width arithmetic (25 bits).
  The first bit is 0, which means the sign of the result is positive.
  The result is: $1.10011110000000000000000_2 * 2^3$
- *Step 4 - Normalize the result: The result must be normalized to have one leading 1.*
  The sum already has the form $1.xxxxxx$*, so it is normalized — no exponent change.*
  The result's unbiased exponent = 3, then the biased exponent $= 130_{10} = 10000010_2$*.*
- *Step 5 - Round the result: There are no nonzero bits beyond the 23 stored fraction bits here (we padded zeros), so no rounding.*
  The result in 32-bit floating-point format is: 0 10000010 10011110000000000000000
  Decimal value: 12.9375

Example 1.11 *Demonstrate 32-bit floating point addition step by step for -10.125 + 5.25*

- *Step 1 - Convert both numbers to 32-bit floating point format:*
  $-10.125_{10} = -1010.001_2 = -1.010001_2 * 2^3 = 1.010001_2 * 2^{130-127} \rightarrow 1\ 10000010\ 01000100000000000000000$
  $5.25_{10} = 101.01_2 = 1.0101 * 2^2 = 1.0101 * 2^{129-127} \rightarrow 0\ 10000001\ 01010000000000000000000$
- *Step 2 - Align exponents:*
  - Larger exponent = 130 (from -10.125). Difference = 130 - 129 = 1.
  - Shift the smaller number's significand (5.25) right by 1 bit:
  $1.01010000000000000000000_2 * 2^2 = 0.10101000000000000000000_2 * 2^3$
- *Step 3 - Add mantissas:*
  Before performing this step, we add two 0 bits at the beginning of the bit sequence.
  $-10.125_{10} \rightarrow 001.01000100000000000000000$
  $5.25_{10} \rightarrow 000.10101000000000000000000$
  Because -10.125 is negative, take its 2's complement:
  $001.01000100000000000000000 \rightarrow 110.10111100000000000000000$

$$\begin{array}{r} 110.10111100000000000000000 \\ +000.10101000000000000000000 \\ \hline 111.01100100000000000000000 \end{array}$$

  The first bit is 1, which means the sign of the result is negative.
  Take its 2's complement, the result is: $000.10011100000000000000000_2 * 2^3$
- *Step 4 - Normalize the result: The result must be normalized to have one leading 1.*
  $000.10011100000000000000000_2 * 2^3 = 1.00111000000000000000000_2 * 2^2$
  The result's unbiased exponent = 2, then the biased exponent $= 129_{10} = 10000001_2$*.*
- *Step 5 - Round the result: There are no nonzero bits beyond the 23 stored fraction bits here (we padded zeros), so no rounding.*
  The result in 32-bit floating-point format is: 1 10000001 00111000000000000000000
  Decimal value: -4.875

1.5 Subtracting two floating-point numbers

Similar to addition, the subtraction of two 32-bit floating-point numbers also follows similar steps as below:

- **Extract and unpack:** Separate the sign, exponent, and mantissa from both floating-point numbers.
- **Align exponents:**
  - Find the number with the smaller exponent.
  - Shift its mantissa to the right until its exponent matches the larger one. The shift amount determines how many bits are shifted to the right.
- **Add mantissas:** Add the two mantissas together. Remember, subtracting a number by a negative number is equivalent to addition.
- **Normalize the result:**
  - If adding the mantissas results in a value that is not normalized (for instance, a carry-out or a number with a leading zero), adjust the exponent and shift the mantissa to normalize it.
  - If the sum is too large or too small for the available bits, this is an overflow or underflow error.
- **Round the result:** Round the final result to fit the required precision, which can introduce inaccuracies.

Example 1.12 *Demonstrate 32-bit floating point subtraction step by step for 0.75 - 11.125*

Remember: $0.75 - 11.125 = 0.75 + (-11.125)$

- *Step 1 - Convert both numbers to 32-bit floating point format:*
  $0.75_{10} = 0.11_2 = 1.1_2 * 2^-1 = 1.1_2 * 2^{126-127} \rightarrow 0\ 01111110\ 10000000000000000000000$
  $-11.125_{10} = -1011.001_2 = -1.011001{*}2^3 = -1.011001{*}2^{130-127} \rightarrow 1\ 10000010\ 01100100000000000000000$
- *Step 2 - Align exponents:*
  - Larger exponent = 130 (from -11.125). Difference = 130 - 126 = 4.
  - Shift the smaller number's significand (0.75) right by 4 bit:
  $1.10000000000000000000000_2 * 2^{-1} = 0.00011000000000000000000_2 * 2^3$
- *Step 3 - Add mantissas:*
  Before performing this step, we add two 0 bits at the beginning of the bit sequence.
  $0.75_{10} \rightarrow 000.00011000000000000000000$
  $-11.125_{10} \rightarrow 001.01100100000000000000000$
  Because -11.125 is negative, take its 2's complement:
  $001.01100100000000000000000 \rightarrow 110.10011100000000000000000$

$$\begin{array}{r} 000.00011000000000000000000 \\ +110.10011100000000000000000 \\ \hline 110.10110100000000000000000 \end{array}$$

  The first bit is 1, which means the sign of the result is negative.
  Take its 2's complement, the result is: $001.01001100000000000000000_2 * 2^3$
- *Step 4 - Normalize the result: The result must be normalized to have one leading 1.*
  The sum already has the form $1.xxxxxx$*, so it is normalized — no exponent change.*
  The result's unbiased exponent = 3, then the biased exponent $= 130_{10} = 10000010_2$*.*
- *Step 5 - Round the result: There are no nonzero bits beyond the 23 stored fraction bits here (we padded zeros), so no rounding.*
  The result in 32-bit floating-point format is: 1 10000010 01001100000000000000000
  Decimal value: -10.375

Example 1.13 *Demonstrate 32-bit floating point subtraction step by step for -8.875 - (-25.25)*

Remember: $-8.875 - (-25.25) = -8.875 + 25.25$

- *Step 1 - Convert both numbers to 32-bit floating point format:*
  $-8.875_{10} = -1000.111_2 = -1.000111_2{*}2^3 = -1.000111_2{*}2^{130-127} \rightarrow 1\ 10000010\ 00011100000000000000000$
  $25.25_{10} = 11001.01_2 = 1.100101 * 2^4 = 1.011001 * 2^{131-127} \rightarrow 0\ 10000011\ 10010100000000000000000$
- *Step 2 - Align exponents:*
  - Larger exponent = 131 (from 25.25). Difference = 131 - 130 = 1.
  - Shift the smaller number's significand (-8.875) right by 1 bit:
  $1.00011100000000000000000_2 * 2^3 = 0.10001110000000000000000_2 * 2^4$
- *Step 3 - Add mantissas:*
  Before performing this step, we add two 0 bits at the beginning of the bit sequence.

$-8.875_{10} \rightarrow 000.10001110000000000000000$
$25.25_{10} \rightarrow 001.10010100000000000000000$
Because -8.875 is negative, take its 2's complement:
$000.10001110000000000000000 \rightarrow 111.01110010000000000000000$

$$\begin{array}{r} 111.01110010000000000000000 \\ +001.10010100000000000000000 \\ \hline (1)001.00000110000000000000000 \end{array}$$

There's a carry out, which we drop because we're working in fixed-width arithmetic (25 bits).
The first bit is 0, which means the sign of the result is positive.
The result is: $001.00000110000000000000000_2 * 2^4$

- *Step 4 - Normalize the result: The result must be normalized to have one leading 1.*
  The sum already has the form $1.xxxxxx$*, so it is normalized — no exponent change.*
  The result's unbiased exponent = 4, then the biased exponent $= 131_{10} = 10000011_2$*.*
- *Step 5 - Round the result: There are no nonzero bits beyond the 23 stored fraction bits here (we padded zeros), so no rounding.*
  The result in 32-bit floating-point format is: 0 10000011 00000110000000000000000
  Decimal value: 16.375

Note: When performing addition and subtraction on floating-point numbers, there are special cases defined by the conventions in the Table 3.

Operation	Number 1	Number 2	Answer
+	$+\infty$	$+\infty$	$+\infty$
+	$+\infty$	$-\infty$	NaN
+	$\pm\infty$	0	$\pm\infty$
-	$+\infty$	$+\infty$	NaN
-	$+\infty$	$-\infty$	$+\infty$
-	$\pm\infty$	0	$\pm\infty$
-	0	$\pm\infty$	$\mp\infty$

Table 3: Special Cases in Addition and Subtraction

.

2 Requirements, implementation procedure, and grading criteria

2.1 Requirements

Design a floating-point unit (FPU) that performs addition and subtraction of two 32-bit floating-point numbers. The design method used is a purely combinational logic design.

After completing the design, students must:

- Write a verification program to verify the design.
- Implement the design on an FPGA board to validate its functionality.
- Perform synthesis of the code using standard cell libraries at different corners and eliminate glitches.

Input and output ports for your design are described in Table 4.

Name	Port Type	Width	Description
i_32_a	Input	32 bits	32-bit floating-point number A
i_32_b	Input	32 bits	32-bit floating-point number B
i_add_sub	Input	1 bit	Operation: 0 = add, 1 = sub
o_32_s	Output	32 bits	32-bit floating-point output
o_ov_flag	Output	1 bit	Overflow Flag: 1 = overflow, 0 = no overflow occurs
o_un_flag	Output	1 bit	Underflow Flag: 1 = underflow, 0 = no underflow occurs

Table 4: Input and Output Ports

2.2 Implementation Procedure

2.2.1 Design FPU

Students can refer to the diagram in Figure 3 to carry out the design. Alternatively, they may consult any reference materials available on the Internet.

The design may have some sub-modules that perform floating point calculations:

- Identify which number is larger, which number is smaller.
- Identify the amount to shift the operand which has smaller exponent.
- Right shift fraction value of the smaller operand to align decimal points.
- Calculate the two's complement of the shifted fraction, only needed in the case of subtraction or equivalent case.
- Add the two fractions together.
- Normalize the fraction and exponent value so it's back in floating point representation.
- Determine sign of the final value.
- Detect zero: the result is zero if the signs of A and B are different and there is no difference in the fraction and exponent.

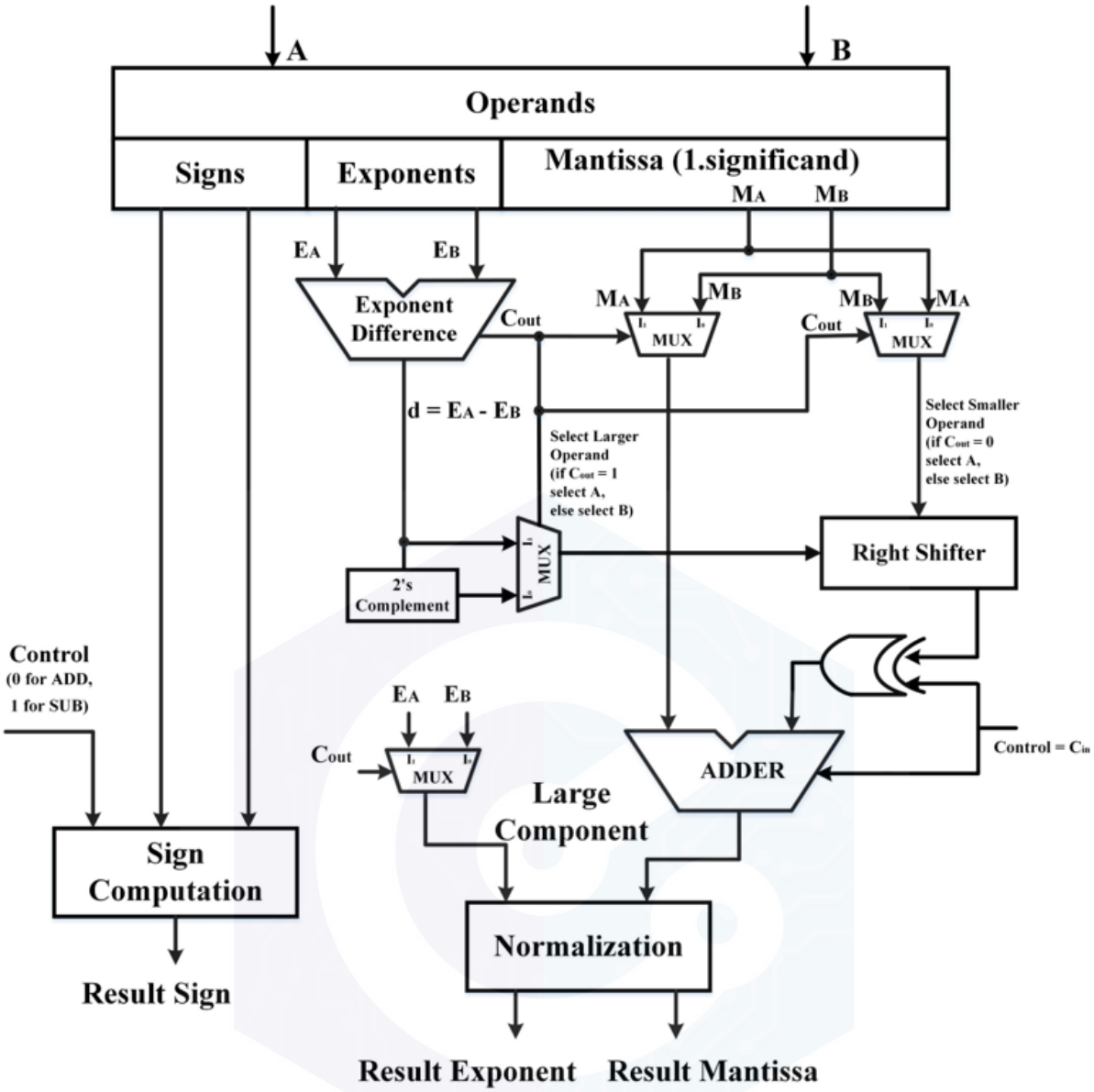


Figure 3: The Adding Arithmetic Algorithm

2.2.2 Simulation and Verification

To verify the design, students must:

- Write a program that generates 100 random input combinations and applies them to the design for testing.
- A task should be written to perform addition and subtraction of two floating-point numbers for comparison with the design's output.
- The special cases of addition and subtraction must be verified at this stage.
- All programs must be written and executed using the Xcelium tool.

2.2.3 Implement on FPGA

Students may choose either the DE10 or DE2 kit to verify whether the design functions correctly.

2.2.4 Synthesis your design

Students shall use the Genus tool on the server to synthesize their design. Eight standard cell libraries are used, as follows:

- fast_vdd1v0_basicCells_hvt.lib
- fast_vdd1v2_basicCells_hvt.lib
- slow_vdd1v0_basicCells_hvt.lib
- slow_vdd1v2_basicCells_hvt.lib
- fast_vdd1v0_basicCells_lvt.lib
- fast_vdd1v2_basicCells_lvt.lib
- slow_vdd1v0_basicCells_lvt.lib
- slow_vdd1v2_basicCells_lvt.lib

When inputs are applied to the circuit after synthesis (netlist), the outputs should have as few glitches as possible.

2.2.5 Reports and Design Defense

All groups must submit a report and conduct a design defense with the instructor regarding their own design. The group report should clearly and thoroughly present the details of the group's design.The groups must draw the design hierarchy from the top module to the sub-modules and finally to the leaf modules. For each block (top, sub-module, and leaf), in addition to the diagram describing the core design, a table must be provided listing the inputs, outputs, signal widths, and signal descriptions.

2.2.6 Grading Criteria

Your project will be evaluated based on the following criteria:

Criteria	Grade
Design an FPU that functions correctly and fully verify all cases	6 pts
Implement the design on an FPGA board to verify that it operates correctly	2 pts
Synthesis and verification of the post-synthesis design	2 pts

Table 5: Grading Criteria