VLSI LAB REPORT | 2024

(CO3098) LSI logic design

Floating-point ALU Specification

Submitted to: Mr. Huynh Phuc Nghi

Submitted by: To Hoang Phong - 2112012
Vu Huynh Tan Phat - 2114391

Ho Chi Minh University of Technology (HCMUT)

The faculty of Computer Science and Engineering

Author	Group 1
Date	2024/04/08
Version	1

Contents

1 Interface

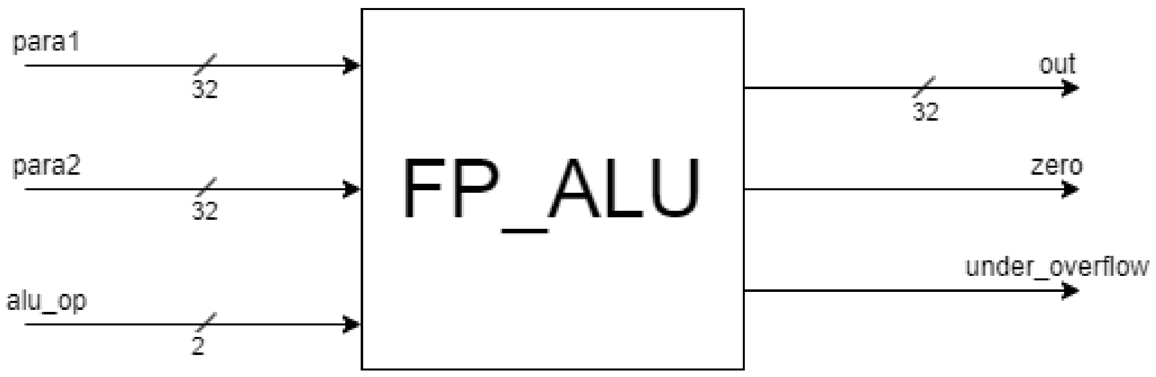


Figure 1: The figure of floating-point ALU System

signal	Width	In/Out	Description
para1	32	Input	The first parameter for the calculation that is presented in the single precision floating point form (a 32-bit approximation of a real number)
para2	32	Input	The second parameter for the calculation that is presented in the single precision floating point form
alu_op	2	Input	The signal for choosing operators in ALU. The next section will describe this signal in detail
out	32	Output	The result of calculation processes that is presented in the single precision floating point form
zero	1	Output	The signal for users to notice whether all bits in the result is 0 or not
under_overflow	1	Output	The signal for users to notice underflow or overflow cases

Table 1: Descriptions of signals in floating-point ALU

2 Functional implementation

- A float-point arithmetic logic unit system that is the operation of two single precision floating point numbers.
- For more details in single precision floating point numbers, please read the Appendix.
- System's operation based on three input signals
  - para1
  - para2
  - alu_op
- The system specification
  - para1, para2 are 32-bit floating point numbers
  - The operator will be decided by **alu_op** signal
    alu_op = 00_b: out = para1 + para2 (the addition operator)
    alu_op = 01_b: out = para1 - para2 (the subtraction operator)
    alu_op = 10_b: out = para1 * para2 (the multiplication operator)
    Others: out = 0
  - The system can handle all floating point numbers, except for infinity, NA (not-a-number) and denormalized number cases
- zero = 1 when out = 0000_h (0 value in the floating point number set)
- Underflow or overflow cases:
  - In underflow cases
    The out signal will equal to $FF800000_h$ $(-\infty)$ and under_overflow = 1
  - In overflow cases
    The out signal will equal to $7F800000_h$ $(+\infty)$ and under_overflow = 1
  - **In multiplication operators**
    Overflow or underflow cases are defined in exponent parts, disregard for the sign part.
    E.g: Overflow case: $-2^{120} * 2^{120} = -2^{240}$, the absolute value is greater than the maximum number.
    Underflow case: $2^{-120} * 2^{-120} = 2^{-240}$, the absolute value is less than the minimum positive number.
    The output value in over/underflow cases is same as the above rule.

3 Diagram

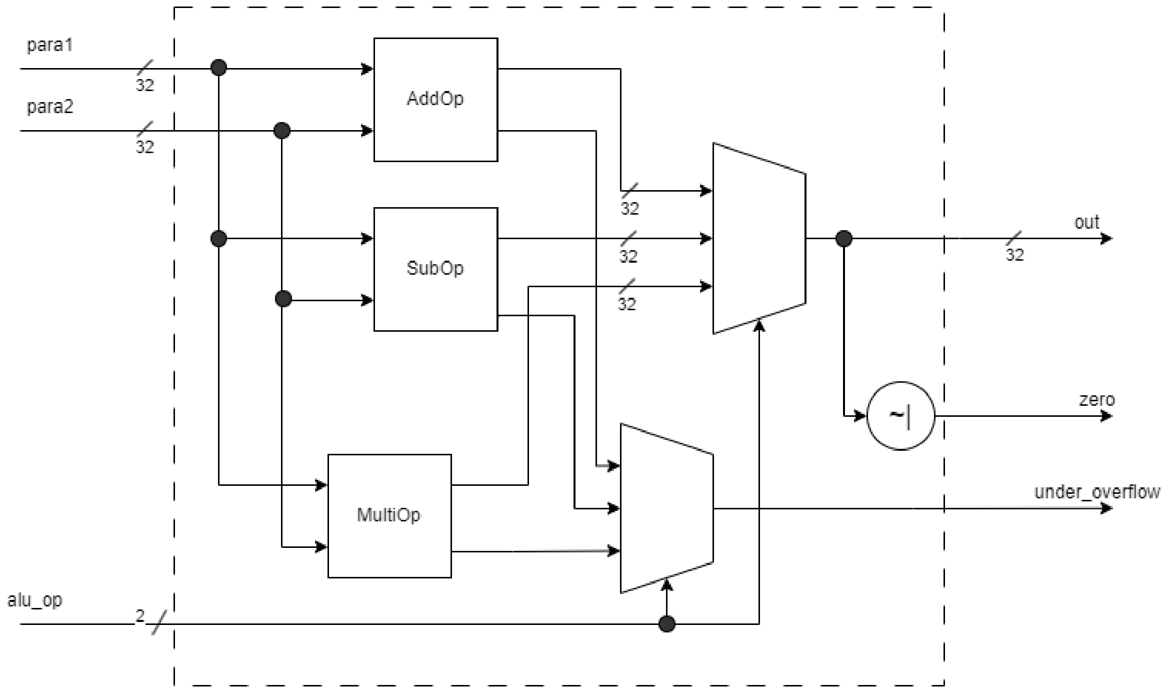


Figure 2: The system diagram

Details

Sub-modules	Interface	Descriptions
AddOp	input [31:0] para1 input [31:0] para2 output [31:0] out output under_overflow	The sub-module handles addition operators between two 32-bit inputs. There is an under_overflow signal for verify underflow or overflow cases
SubOp	input [31:0] para1 input [31:0] para2 output [31:0] out output under_overflow	The sub-module handles subtraction operators between two 32-bit inputs. There is an under_overflow signal for verify underflow or overflow cases
MultiOp	input [31:0] para1 input [31:0] para2 output [31:0] out output under_overflow	The sub-module handles multiplication operators between two 32-bit inputs. There is an under_overflow signal for verify underflow or overflow cases
$\sim \mid$	input [31:0] number output zero	The sub-module is a reduction 'nor' operator that help to notice whether bits of number are all 0

Table 2: Descriptions of sub-modules in floating-point ALU

4 Appendix

Reference: David A.Patterson & John L. Hennessy, *Computer Organization and Design RISC-V edition*

Single precision floating point numbers

Floating-point numbers are usually a multiple of the size of a word. The representation of a RISC-V floating-point number is shown below, where s is the sign of the floating-point number (1 meaning negative), exponent is the value of the 8-bit exponent field (including the sign of the exponent), and fraction is the 23-bit number.

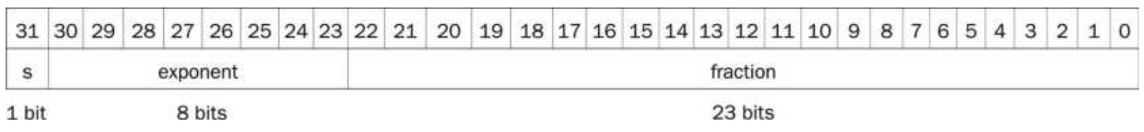


Figure 3: The number format

In general, floating-point numbers are of the form:

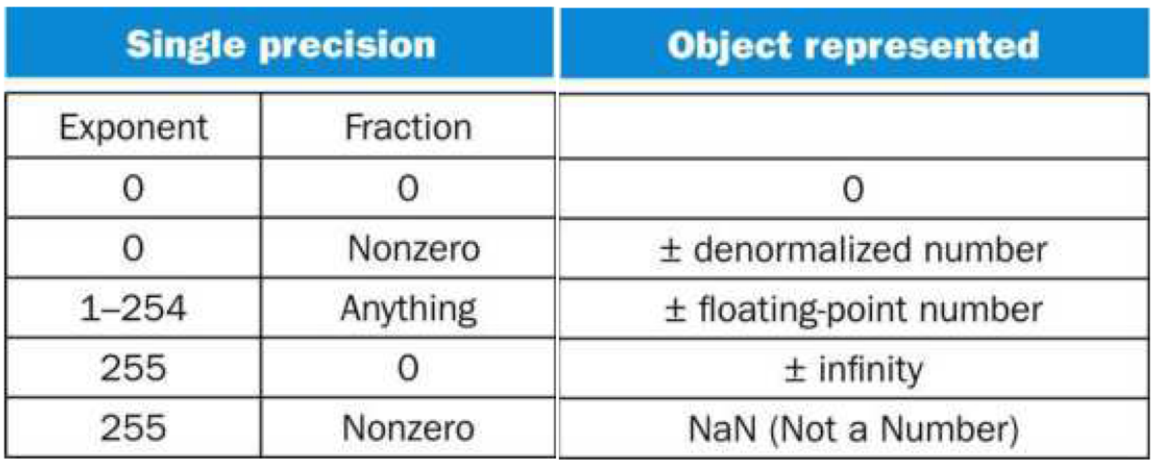
$$(-1)^S * F * 2^E \tag{1}$$

F involves the value in the fraction field and E involves the value in the exponent field. However, to pack even more bits into the number, IEEE 754 makes the leading 1 bit of normalized binary numbers implicit. Hence, the number is actually 24 bits long in single precision (implied 1 and a 23-bit fraction).

$$(-1)^S * (1 + Fraction) * 2^E \tag{2}$$

where the bits of the fraction represent a number between 0 and 1 and E specifies the value in the exponent field, to be given in detail shortly. If we number the bits of the fraction from left to right f1, f2, f3, ..., then the value is

$$(-1)^S * (1 + f_1 * 2^{-1} + f_2 * 2^{-2} + f_3 * 2^{-3} + ...) * 2^E \tag{3}$$



Single precision		Object represented
Exponent	Fraction	
0	0	0
0	Nonzero	± denormalized number
1–254	Anything	± floating-point number
255	0	± infinity
255	Nonzero	NaN (Not a Number)

Figure 4: IEEE754 number encoder

IEEE 754 uses a bias of 127 for single precision, so an exponent of -1 is represented by the bit pattern of the value $-1 + 127_d$, or $126_d = 01111110_b$, and +1 is represented by $1 + 127$, or $128_d = 10000000_b$. Biased exponent means that the value represented by a floating-point number is really:

$$(-1)^S * (1 + Fraction) * 2^{E-127} \tag{4}$$

The minimum positive number is 2^{-126}

The maximum positive number is $(2 - 2^{-23}) * 2^{127}$

Addition and subtraction operators

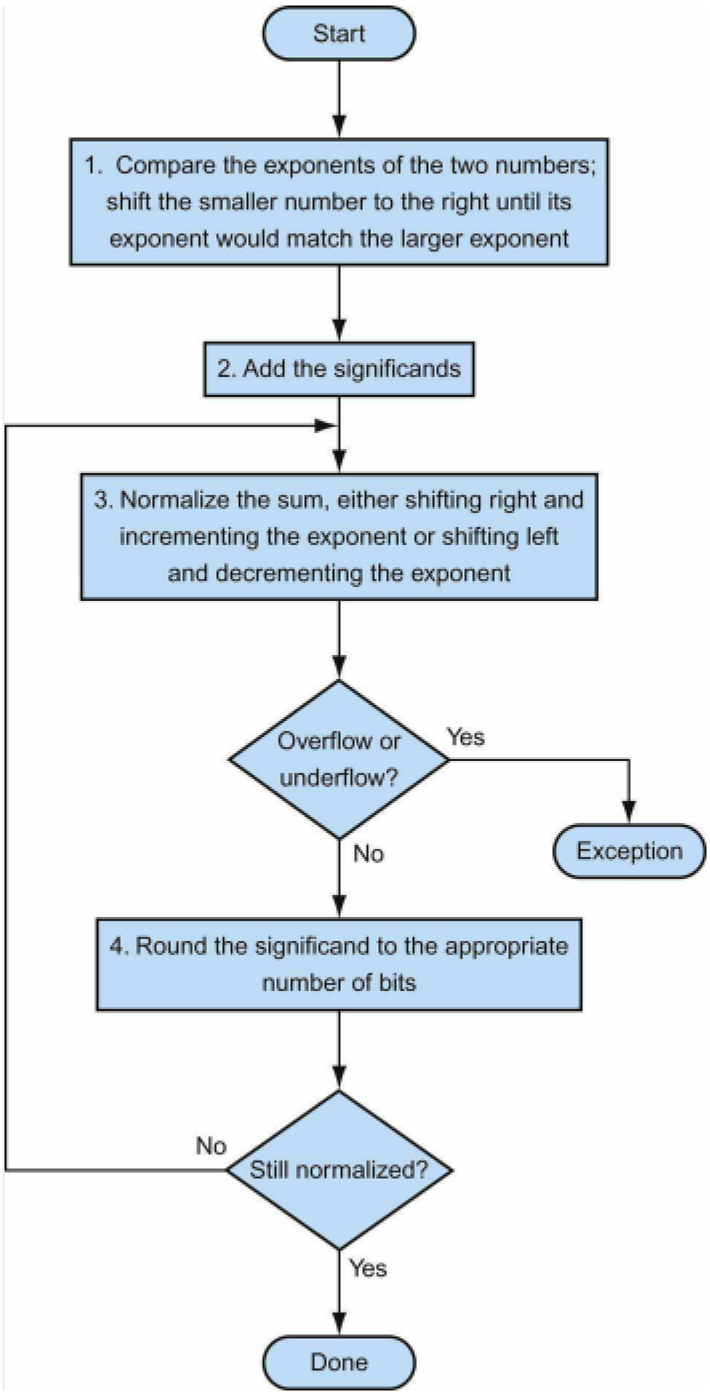

Start
1. Compare the exponents of the two numbers; shift the smaller number to the right until its exponent would match the larger exponent
2. Add the significands
3. Normalize the sum, either shifting right and incrementing the exponent or shifting left and decrementing the exponent
Overflow or underflow?
Yes
Exception
No
4. Round the significand to the appropriate number of bits
No
Still normalized?
Yes
Done

Figure 5: Floating-point addition and subtraction algorithm

Multiplication algorithm

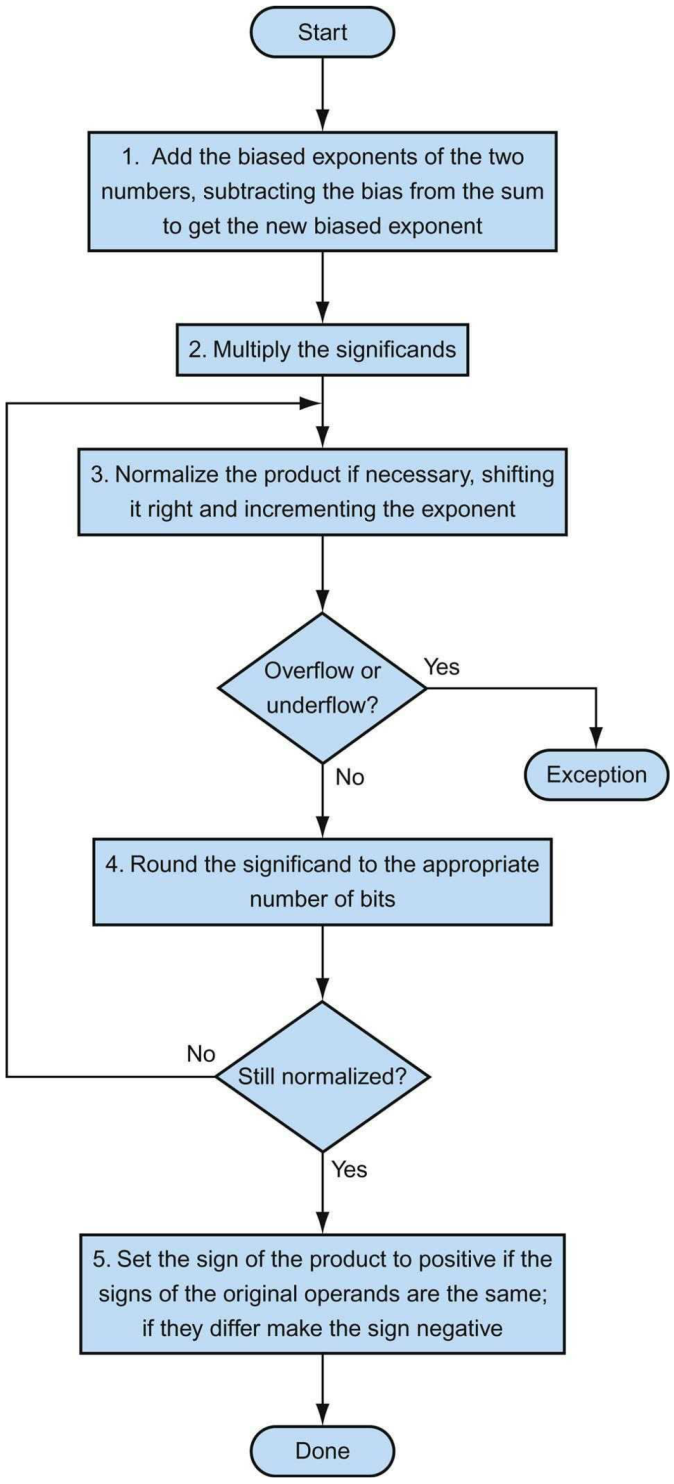

Start
1. Add the biased exponents of the two numbers, subtracting the bias from the sum to get the new biased exponent
2. Multiply the significands
3. Normalize the product if necessary, shifting it right and incrementing the exponent
Overflow or underflow?
Yes
Exception
No
4. Round the significand to the appropriate number of bits
No
Still normalized?
Yes
5. Set the sign of the product to positive if the signs of the original operands are the same; if they differ make the sign negative
Done

Figure 6: Floating-point multiplication algorithm