Floating Point Arithmetic Unit

DOCUMENTATION

Structure of Computer Systems

STUDENT: DOMȘA-STÎNĂ ANDA
GROUP: 30433

1 TABLE OF CONTENTS

1. Introduction:

1.1. Context:

The goal of this project is to design, implement and test an arithmetic unit that performs two operations on 32 bit format floating point numbers. This device can be used as a calculator that can perform addition and subtraction of 32 bit floating point numbers in 2's complement.

This device can be used as an addition to some other devices. For example, it could be integrated in any sort of processor or microprocessor that could use or benefit from having the capability of doing these operations.

1.2. Project description:

The main objective of the project is to implement a user friendly device that can perform the addition and subtraction operations based on the input. The user should be able to choose the operation he wants to perform and to input the numbers to be used in the computation step.

The device will be simulated in Vivado and tested on the Basys 3 board. It will be able to perform addition and subtraction in two's complement representation and present them in a readable format.

2. Bibliographic research:

In order to start the design of the arithmetic unit, we need to understand the way to represent 32-bit floating point numbers in 2's complement (IEEE Standard 32 bit for Floating Point Binary Arithmetic).

Regarding this fact, we acknowledge that to represent such a number we will need a sign bit (S - 0 for positive, 1 for negative), an exponent (E - weights value by a power of 2) and a mantissa (M – also called significant or base number), as referenced in [1].

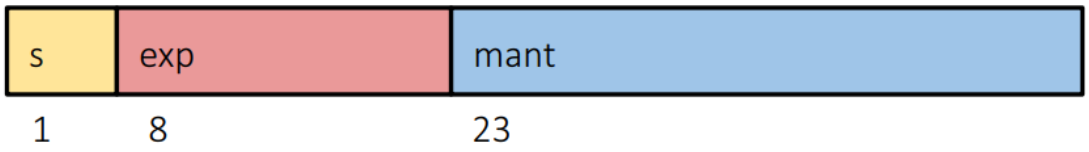


Figure 1. IEEE Standard floating point numbers representation [1]

The value of the floating point number is computed using the following formula:

$$value = (-1)^S * M * 2^E$$

The steps to convert a floating point 32-bit number into IEEE 754 Standard 32 bit for Floating Point Binary Arithmetic are the following [2]:

1. Determine the sign bit

2. Convert to pure binary

3. Normalize to determine the mantissa and the unbiased exponent (place the binary point after leftmost 1)

4. Determine the biased exponent (add 127 then convert to an unsigned binary integer)

5. Remove the leading 1 from the mantissa

Special cases:

- Not a number (NAN): - E = all ones, M = not all zeros
- Infinity: - E = all ones, M = all zeros
- 0: - E = all zeros, M = all zeros

Next we will discuss the way to implement the required operations [2]:

1. **Addition**:
   - First we align the exponents if it is necessary (shift left the floating point of the mantissa having the smaller exponent until we reach the value of the higher one – right shift of the mantissa). Then we add the base numbers and finally we normalize the result.
   - Downsides: it is possible to loose precision due to truncation errors.
   - We need to also take into consideration the possible overflow that is signaled by a difference between the last two carries.

Special cases:

While adding the two floating point numbers, two cases may arise.

Case 1: when both the numbers are of same sign i.e. when both the numbers are either +fp or -fp. In this case MSB of both the numbers are either 1 or 0.

Case II: when both the numbers are of different sign i.e. when one number is +fp and other number is –fp. In this case the MSB of one number is 1 and other is 0.

So when we firstly check the sign of the two numbers, if the Sign of any of the two number is different than the other, we take the 2's complement of the respective number and then add the two numbers.

Other special operations:

- Infinity + Infinity = Infinity
- Infinity – -Infinity = Infinity
- -Infinity - Infinity = -Infinity
- -Infinity + -Infinity = Infinity
- Nan == Nan => FALSE

2. **Subtraction**:
    - Or addition of negative quantities, we are going to use the negation trick to implement it.
    - So we are going to add a "phantom" –2 in front (–1*2 ^ 1) and then add as usual.
    - So we will do the same as for addition, first we will align the exponents, but then we will negate the number to be subtracted. After, we add the mantissas and normalize if needed.
    - The overflow will be signaled by a difference between the last two borrows.

Hardware resources needed:

- The device will use the input registers to load the data, the control unit will choose the operation to be performed based on the input data and the result will be saved on the output registers.
- Also needed: arithmetic unit (addition, shift), comparator.

3. Analysis:

In addition or subtraction operations, first of all we have check the leftmost bit which is called the sign bit. If the bit is 0, then the decimal equivalent of the binary number will be positive. But if the leftmost bit is 1, then the decimal equivalent of the binary number will be negative. The addition or subtraction of both positive or both negative numbers take place according to the normal addition or subtraction rules (where a carry of 1 and a sum of 0 is produced when two 1s are added).

Use Cases:

Scenario: The user will input two floating point numbers and the operation to be performed (addition/subtraction). The user won't input the number in floating point, but an index to a value stored in a memory using the switches. For the selection of the operation, we will also use a switch that will be 0 for addition and switched to 1 for subtraction.

Expected results: The device will perform the floating point addition/subtraction and display the correct result in the IEEE Standard floating point numbers representation using the leds of the board. The result will be displayed on the Seven Segment Display.

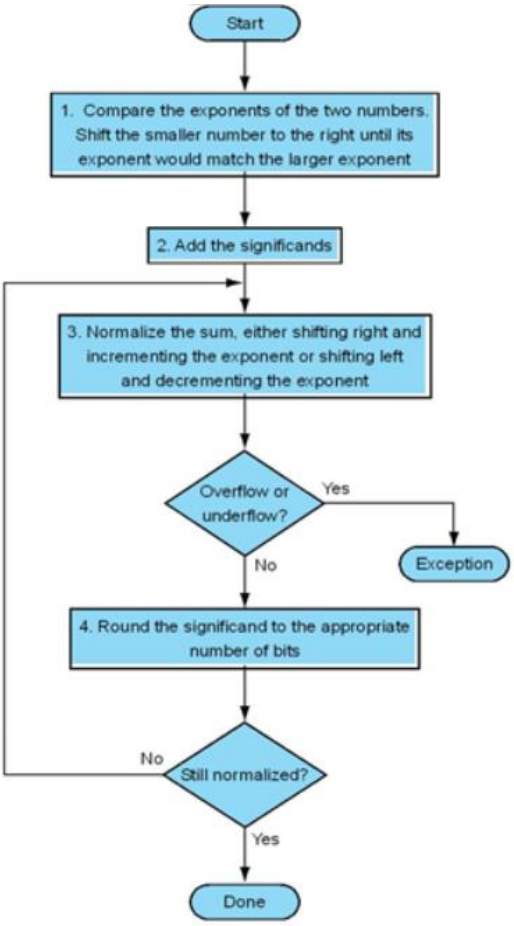


Figure 2. [3]. Floating Point Addition/Subtraction Flowchart

Converting fixed point numbers into floating point numbers:

In most hardware, the data of some signal or a quantity is stored in a format called "Q" fixed point format (Q(m.n) represents there would be m bits before the decimal point and n bits after the decimal point). The embedded systems usually wouldn't have the complexity of storing decimals in floating-point format. They store numbers in the registers mostly 16 bits to 32 bits.

$$\text{Range} \quad : \quad [-2^{m-1}, 2^{m-1} - 2^{-n}]$$

$$\text{Resolution:} \quad [-2^{-n}]$$

Figure 3. [5]. Signed Qm.n

So there would be a need to convert from floating point to fixed point and vice versa when storing and retrieving decimal values from embedded systems that store the data at fixed length memory locations or registers.

The general workflow would be [5]:

1. Read data from the registers in fixed-point format.
2. Convert to floating-point
3. Perform floating-point arithmetic to process the data.
4. Convert the result back to the fixed point format.
5. Write the result back to the registers

Converting from floating-point to fixed-point

Let F be the floating-point number to convert it to the fixed point number

1. Multiply it by 2^n
2. Round the value to the nearest integer
3. If F is negative take two's complement of the value arrived at step 2.

Converting from fixed-point to floating-point

Let X be the fixed-point number to convert it into floating-point

1. Convert the fixed-point number as an integer.
2. Divide the number by 2^n (2 to the power of n).

If the Q format is signed and the x is negative then convert "X" to its two-complement equivalent before performing step 1 above.

3.1 Addition:

By analyzing the algorithm for addition we see that in order to design the device we need to design a 32-bit adder. Since the adder is at the core of this operation, the logical impulse is to design a good adder, so the focus will be in implementing an optimum adder since everything will benefit from this.

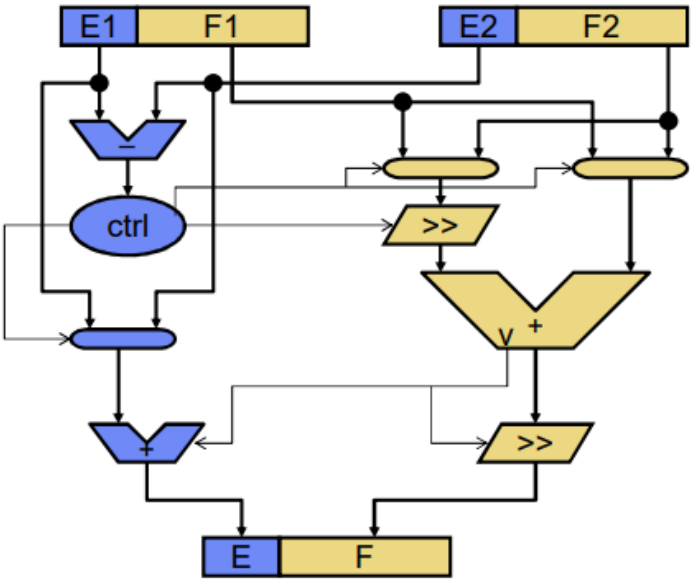


Figure 4. [2]. Floating Point Addition HW

At first, for both operations, we need to convert the floating point 32-bit number into IEEE standard 32 bit for Floating Point Binary Format, so we will need a separate unit for this step.

Then, we can start thinking about how to implement the addition algorithm. For this step we will implement an ALU that can perform either addition or subtraction.

If the desired operation is addition, the ALU will take the 2 numbers, and compare the exponents. At this step the device will decide if the operation can take place or not. For example, if the difference between the exponents is higher than 24, the operation doesn't make sense because the mantissa will be completely lost after the shifts. Then, if the operation can be performed, it will shift left the floating point of the mantissa having the smaller exponent until the value of the higher one is reached. For this we will need a comparator and a shifter.

After the numbers are aligned, here comes the need of the adder; we add the mantissas bit by bit and normalize the result.

Example: **Let the two numbers be**: x = 9.75, y = 0.5625

9.75's representation in 32-bit format = **0 10000010 00111000000000000000000**

0.5625's representation in 32-bit format = **0 01111110 00100000000000000000000**

Now we get the difference of exponents to know how much shifting is required.

(**10000010 – 01111110**)2 = (**4**)10

Now, we shift the mantissa of lesser number right side by 4 units.

Mantissa of **0.5625 = 1.00100000000000000000000**
Shifting right by **4** units, we get **0.00010010000000000000000**
Mantissa of **9.75** = **1. 00111000000000000000000**

Adding mantissa of both:

0. 00010010000000000000000 + 1. 00111000000000000000000 = **1. 01001010000000000000000**

In the final answer, we take the exponent of the bigger number:

Sign bit = **0**
Exponent of bigger number = **10000010**
Mantissa = **01001010000000000000000**
32 bit representation of answer = **x + y** = **0 10000010 01001010000000000000000**

3.2 Subtraction:

From the study of floating point subtraction, one can observe that the algorithm is almost identical to that of the addition one, when viewed from a high level perspective. The flow of the operation resembles that of addition and is comprising.

The algorithm for subtraction has the same steps as the one for the addition, because for subtraction we will simply perform an addition between the first number and the 2's complement of the second one. So we are going to use the same resources we used for addition.

Example: Let the two numbers be: x = 9.75 and y = – 0.5625

9.75's representation in 32-bit format = **0 10000010 00111000000000000000000**
– 0.5625's representation in 32-bit format = **1 01111110 00100000000000000000000**

Now, we find the difference of exponents to know how much shifting is required.

(**10000010 – 01111110**)2 = (**4**)10

Now, we shift the mantissa of lesser number right side by 4 units.
Mantissa of **– 0.5625 = 1.00100000000000000000000**
Shifting right by **4** units, **0.00010010000000000000000**
Mantissa of **9.75**= **1. 00111000000000000000000**
Subtracting mantissa of both

0. 00010010000000000000000 – 1. 00111000000000000000000 = **1. 00100110000000000000000**
Sign bit of bigger number = **0**
So, finally the answer = **x – y = 0 10000010 00100110000000000000000**

4. Design

The black box view and block diagram of the single precision floating point unit is shown in the picture below. The input operands are firstly divided into their sign, mantissa and exponent components. This module has as inputs the first number and the second number which are operands of 32-bit width. Before computing the required operation, the device should take into account some special cases and treat them separately.

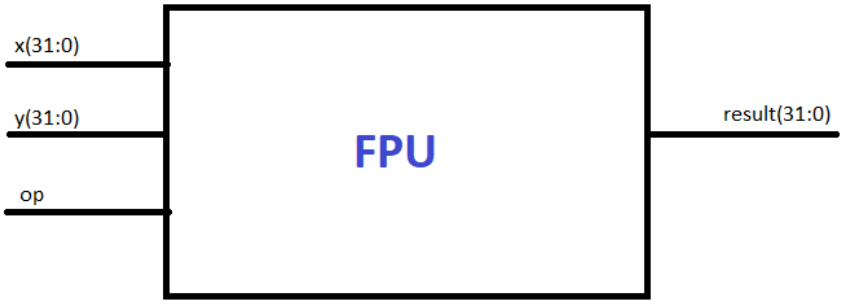


Figure 5. FPU Black Box

Components:

- **The Rom memory unit:** - used for storing some values that the user can access through their address, since the board doesn't give us the possibility to input 32 bit numbers.
- **The number processing unit: -** various functions, prepares the numbers for the operation: **Comparator + RightShifter**
  - Compares the exponents of the numbers and determines the higher one
  - Performs the needed right shifting of the mantissa by the difference between the exponents
  - Determines the higher number between the two
  - Obtains the output exponent (the higher one)
- **The addition/subtraction unit: -** performs the selected operation.
- **Adder Block:** - This block only implements the operation (addition or subtraction). It can be said the adder block is the ALU (Arithmetic Logic Unit) of the project because it oversees the arithmetic operations.

  Two functions are implemented in this part of the code:

  1. Obtaining the output's sign
  2. Implementing the desired operation.

  A Carry Look Ahead structure has been implemented. This structure allows a faster addition than other structures. It improves by reducing the time required to determine carry bits. This is achieved by calculating the carry bits before the sum which reduces the wait time to calculate the result of the large value bits.

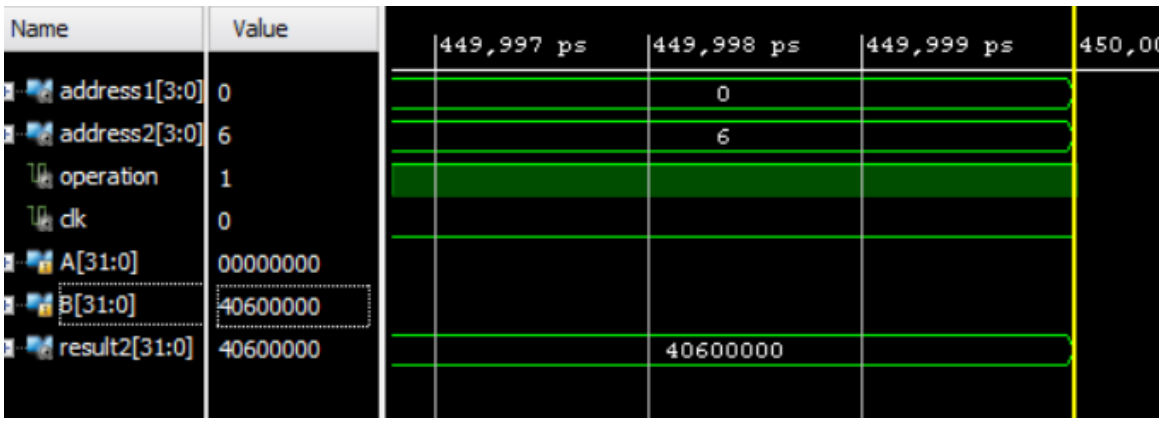
Figure 6 Diagram

5. Implementation:

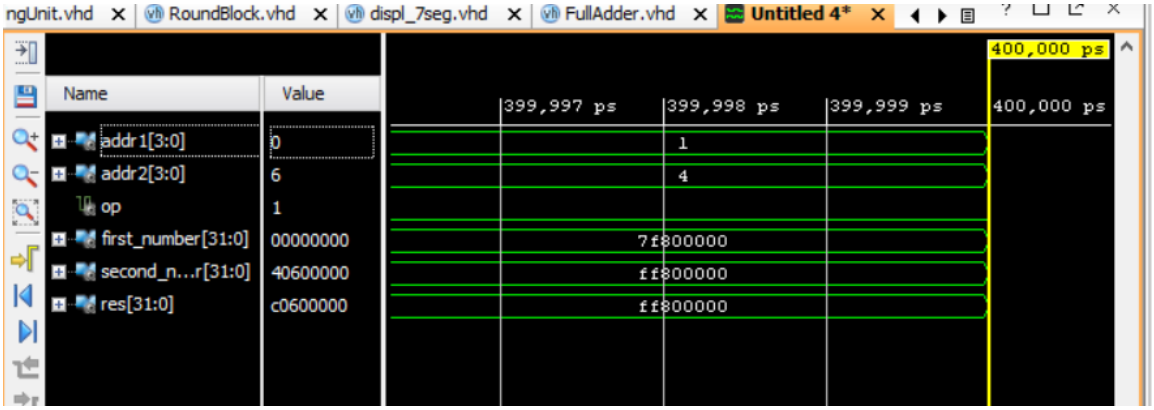
- **Top FPU Module:**
  - The FPU takes in two inputs, addr1 and addr2, which are 4-bit std_logic_vectors used to address two internal read-only memories (ROMs) that store the first and second operands. The FPU also takes in an input signal op, which determines whether the operation is addition or subtraction. The FPU uses the operands retrieved from the ROMs and the operation signal to perform the addition or subtraction operation using a component called FPAdder. The result of the operation is then output as a 32-bit std_logic_vector.
- **OpEnable:**
  - This component gets two numbers A and B. It checks whether one of them or both are 0, Infinity or NaN the Result will be directly calculated, and the enable will be set to 0, otherwise the enable is 1.
- **ROM Memory Unit:**
  - I implemented a ROM Memory to store the numbers and choose them directly, since the board doesn't have the possibility to read 32 bits.
- **Number Processing Unit:** (comparator + shifter) – register + comparator + shifter
  - 2 main elements: exponents' comparator and right shifter.
  - It prepares the numbers for add/subtract operation by calculating the maximum exponent, shifting the mantissa of the other, and getting the signs of each number. The mantissas will be extended on 28 bits for a more precise calculation.
  - The role of the comparator is to break the numbers into sign, exponent and mantissa and compute the signs, the maximum exponent, the difference of the exponents and the mantissas.
  - The shifters are used to perform the normalization. Depending on the shift value generated by the Control unit, the Mantissa/Exponents are shifted accordingly. The purpose of the right shifter here is to shift the mantissa of the numbers with the smaller exponent by the difference of the exponents.
  - **Comparator:** The role of this component is to break the numbers into sign, exponent and mantissa and return the signs, the maximum exponent, the difference of the exponents and the mantissas.
  - **RightShifter:** Its purpose is to shift the mantissa of the numbers with the small exponent by the difference of the exponents.
- **Adder:**
  - The Adder is using multiple Carry Look Ahead Adders to compute the sum/difference of the mantissas, computes the final sign and returns the carry out of the operation.

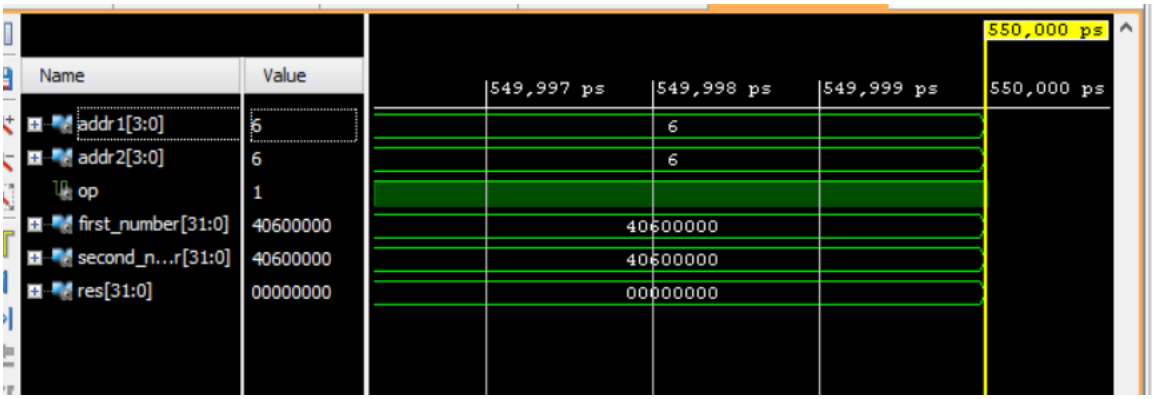
6. Testing and validation:

1. Addition:



3. Subtraction:





7. Conclusions:

This project aimed to design and implement two operations, addition and subtraction, in single precision floating point format using 32 bits in the standard IEEE format. The project presented a challenge as it required a thorough understanding of floating point representation in order to successfully design and implement the operations. Despite the difficulties, I found the project to be enjoyable as it provided an opportunity to apply my logic and knowledge of VHDL language and computer systems. Through research and experimentation, I was able to develop my own implementation for the floating point addition and subtraction, ultimately achieving correct results.

8. Bibliography:

[1] https://courses.cs.washington.edu/courses/cse351/17wi/sections/03/CSE351-S03-2cfp_17wi.pdf

[2] http://people.ee.duke.edu/~sorin/prior-courses/ece152-spring2011/lectures/3.3-arith.pdf

[3] https://www.cs.umd.edu/~meesh/411/CA-online/chapter/floating-point-arithmetic-unit/index.html

[4] https://www.sciencedirect.com/science/article/abs/pii/S0925231219308884

[5] https://medium.com/incredible-coder/converting-fixed-point-to-floating-point-format-and-vice-versa-6cbc0e32544e