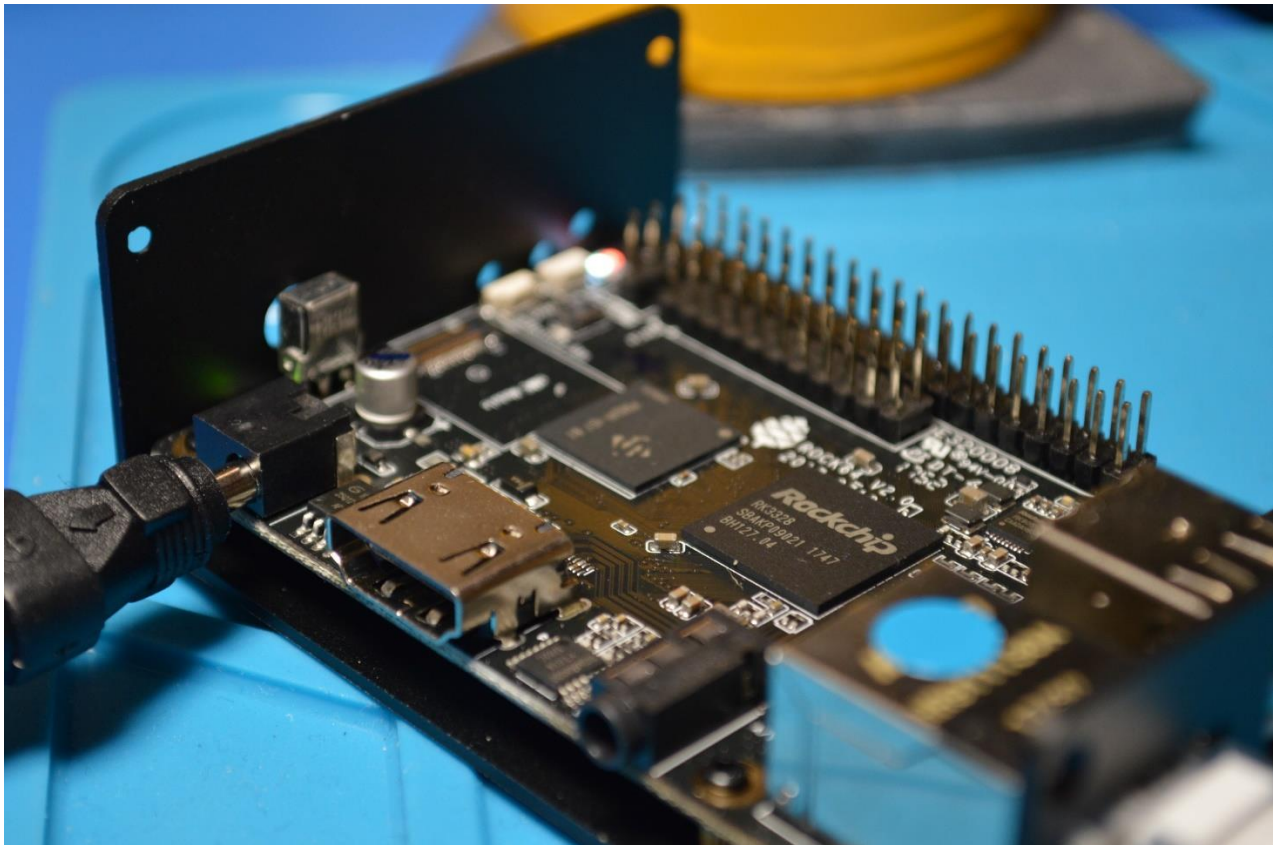


FPU: multiplication & division



Tilea Anda-Corina

Group 30431

05.01.2021

Contents

1. Introduction

1.1	FPU Definition	3
1.2	Floating-Point Numbers	3
1.3	2's Complement Numbers	4
1.4	Standard IEEE	4
1.5	Special Numbers	4-5
1.6	Project Proposal	6
1.7	Plan	6

2. [Bibliographic study](#)

3. Analysis

3.1	Multiplication	7-12
3.2	Proper Example	8
3.3	Division	13-15

4. Design

4.1	Multiplication	15-18
4.2	Division	19-20
4.3	FPU	20-21

5. Implementation & Testing

5.1	Multiplication	21-23
5.2	Division	24-26
5.3	Special Cases	26-28
5.4	Appendix 1	28-30

6. [Conclusions](#)

7. [References](#)

1. Introduction – Floating Point Unit

1.1 Definition

FPU (Floating Point Unit) is designed to carry out operations for numbers in floating point representation. Some of typical operations we are going to talk about are **multiplication** and **division**.

1.2 Floating-point Numbers

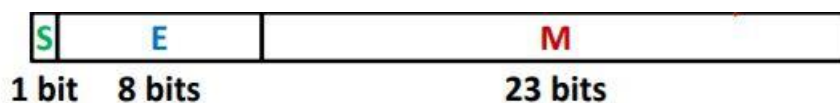
Floating point describes a system for representing numbers that would be too large or too small to be represented as integers. The representation of a number is the significant digit multiplied by its base with exponent power

Ex:

$$+7.23 * 10^{-23}$$

All the floating-point numbers will contain the following components:

- sign: indicate the sign of the number (1 – negative, 0 – positive);
- exponent: containing the value of the base;
- mantissa: setting the value of the number;
- base: it is also called radix, it is common to all the numbers;



In the digital system, the IEEE 754 standard is used for representing floating point numbers.

The proposed environment for floating point will be **VHDL**. The use of VHDL is considered appealing since it provides a formal description of the system and allows specific description styles.

1.3 2's Complement Numbers

In this project the conversion from and into 2's complement representation will take place. 16 bits will be assigned for the integer part and the other 16 bits will represent the fractional part.

An n-bit, 2's complement number can represent the range $[-2^{n-1}, 2^{n-1} - 1]$. The asymmetry of this range is caused by the fact that there exists one more negative number.

The objective of the project is to implement a method in which 2's complement fractional numbers will be stored in the floating point representation and the correct result must be displayed. The numbers need to be normalized in order to make the equivalent floating point representation.

An example of multiplication on 4-bit numbers in 2's complement will look like this:

<p>with correct sign extension</p> <pre> 11100 x 00011 ----- 111111100 111111100 ----- 111110100 (-12) RIGHT!</pre>	<p>with correct adjustment</p> <pre> 11101 (-3) x 11101 (-3) ----- (get additive inverse of both) 00011 (+3) x 00011 (+3) ----- 00011 + 00011 ----- 001001 (+9) (right!)</pre>
---	--

1.4 Standard IEEE

Standard IEEE 754 specifies formats and methods in order to operate with floating point arithmetic.

This Standard can define arithmetic formats such as sets of binary and decimal floating-point data (finite numbers and infinite ones). It is also capable of performing interchange formats like encodings (used to exchange floating-point data in an efficient and compact form). The most important feature which is going to be used is the ability to perform operations: arithmetic operations (like multiplication or division).

A special case which may be encountered here, is the exception handling, providing indication of exceptional conditions (such as division by 0).

1.5 Special numbers

- **Overflow** = it occurs when the number is too large to fit in the frame (between largest norm and infinity);

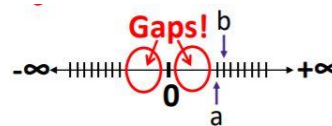
- **Underflow** = it occurs when the number is too small to fit in the frame (between zero and smallest denorm);

Zero:

IEEE standards proposed zero to become a special case (if every bit is zero, then the numbers is considered zero – E and M all zeros - the sign bit being irrelevant).

Numbers closest to 0:

- $a = 1.0...0_2 \times 2^{-126} = 2^{-126}$
- $b = 1.0...01_2 \times 2^{-126} = 2^{-126} + 2^{-149}$



A special case is also the result of an operation like infinity times zero:

$$E = 0xFF, M = 0: \pm \infty$$

Some other special cases are the representations for positive and negative infinity, and also for a not-a-number (NaN). **NaN** values can be obtained in the square root of negative numbers, the division 0/0 or infinity / infinity (the value of M will tell the user the cause of NaN). A number is considered to be NaN if every bit of the exponent is 1 plus any mantissa bits are 1.

$$E = 0xFF, M \neq 0: \text{Not a Number (NaN)}$$

The sign will distinguish positive and negative infinity and +/- NaN.

A number is said to be infinite if every bit of the exponent is 1, so the new largest value will have the form:

$$E = 0xFE \text{ has largest: } 1.1...1_2 \times 2^{127} = 2^{128} - 2^{104}$$

Any non-zero number that is smaller than the smallest normal number is a **denormalized number** ($E = 0, M \neq 0$). Denormalized numbers close the gap between zero and the smallest normalized number (there will still be a gap between zero and the smallest denormalized number).

- Smallest norm: $\pm 1.0...0_{\text{two}} \times 2^{-126} = \pm 2^{-126}$
- Smallest denorm: $\pm 0.0...01_{\text{two}} \times 2^{-126} = \pm 2^{-149}$

So much closer to 0

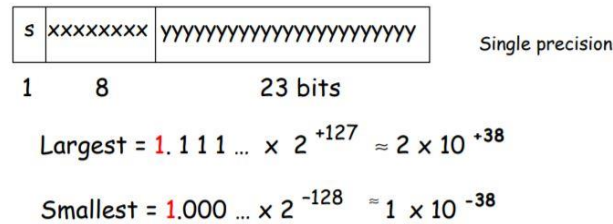
An Encoding Summary will look like:

Exponent	Mantissa	Object
0	0	Zero
0	Nonzero	Denormalized number*
1-254	Anything	+/- FP number
255	0	+ / - infinity
255	Nonzero	NaN like o/o or ox inf

1.6 Project Proposal

In this project, a Floating-Point Unit, designed using VHDL code will be presented and the two operations (multiplication and division), on the Standard IEEE format on 32 bits, will be tested and verified.

This Unit will perform the mentioned operations only on floating-point numbers with single precision.



The implemented device can be used as a calculator in order to multiply or divide 32-bit numbers.

Talking further away, it can be connected as a peripheral device for executing these two operations or it can even be integrated in a processor where the multiplication and division are needed.

The simulation will take place in the IDE provided by Vivado.

This project will allow to represent the numbers in the Standard IEEE, compute the operations in the floating-point representation and visualize the result in a readable format.

1.7 Plan

In this section, the project presents a scheduling of all the steps involved in order to reach our goal. There will be five project meetings, as established.

In the second meeting, a short VHDL programming recap will take place in order to remember what was previously studied. Also, the requirements of the problem will be analysed once more for a better understanding.

In the third meeting, the analysis of the project will begin. Firstly, the multiplication's and division's algorithms need to be truly understood in order to know if they work correctly.

In the fourth meeting, after the analysis part is done, the design chapter will begin. Also, the documentation will be updated at each step.

In the fifth meeting, the design part of the problem will be done and the next chapter should be started: the full implementation of the designed components and the results (the Testing part of the project).

In the sixth meeting, all the chapters should be refined and finalized. The Conclusions part should be completed.

- Calculation of the Sign;
- Composition of the result (as a conclusion);

We will take a **proper example** for a better understanding of what this operation requires:

A = - 18.0, B = 9.5

1. Put the operands in a binary representation:

A = - 10010.0, B = + 1001.1

2. Then transform it to the normalized representation of the operands:

A = - 1.001 x 2⁴, B = + 1.0011 x 2³

3. Then compute it to the IEEE representation of the operands:

A = 1 10000011 001000000000000000000000, B = 0 10000010 001100000000000000000000

After we obtain the numbers in the wanted format, we can start the algorithm steps for the multiplication:

1. The mantissas are extracted, adding an 1 as MSB, for normalization:

A = 1 001000000000000000000000, B = 1 001100000000000000000000

2. The result of the multiplication will be on 48 bits: 0x558000000000
3. After normalization (elimination of the most significant 1), the 23-bit mantissa of the result is obtained:

01 010101100000000000000000 000000000000000000000000

4. Addition of the exponents:

The sum of the operands exponents will be equal to the exponent of the result.

The exponent field are biased, so it must be removed in order to do the addition. It will be added later, for obtaining the exponent field of the result

$$\text{Exp_result} = (\text{Exp_a} - 127) + (\text{Exp_b} - 127) + 127 = \text{Exp_a} + \text{Exp_b} - 127$$

In this case the calculus will look like this:

Exp_a 10000011

Exp_b 10000010

-127 10000110

Exp_result 10000110

5. Calculation of the sign of the result:

The sign will be given by the xor operation between the signs of the operands:

$$S_{\text{result}} = S_a \text{ xor } S_b = 1 \text{ xor } 0 = 1 \text{ (negative)}$$

6. Composition of the result:

Set the sign, exponent and mantissa and the final result will be:

$$1 \ 10001110 \ 010101100000000000000000 \sim (-171.0)$$

After seeing all the necessary steps for computing a multiplication between 2 floating-point numbers, we can design a block diagram for this operation:

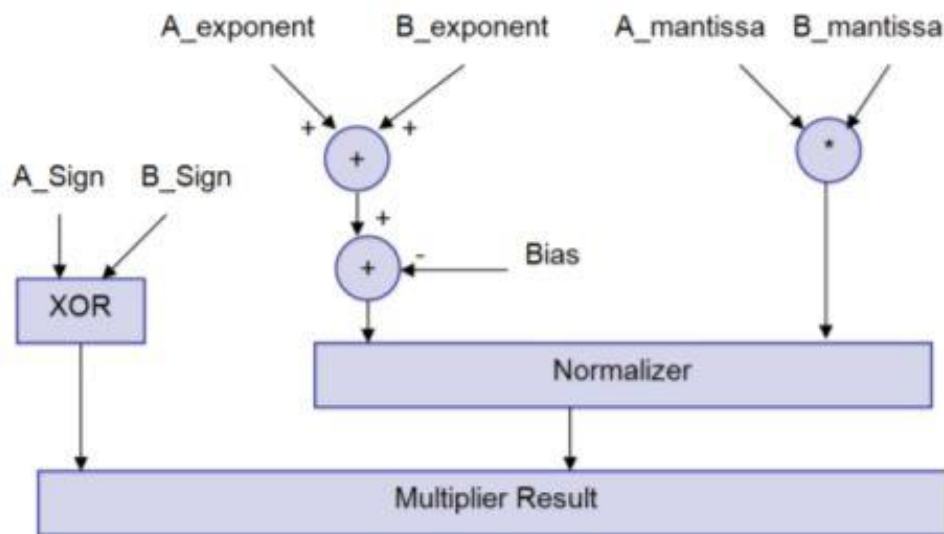


Fig. 1. – Floating point multiplier block diagram

Firstly we will need to design **an Adder for 8 bits numbers** (the Exponents), and **a Multiplier of 24 bit numbers** (Mantissas).

We saw from the previous example that the Adder should be used to compute the exponent of the result (adding the exponents of the two operands taking into account the bias).

The multiplier will be used for the mantissas. An extra '1' will be added as a MSB for both of the operand's mantissas. If the first bit of the result will be 1 then we take the 23 bits for the result's mantissa from the second bit and we will need to normalize the result (adding one to the exponent). If the first bit is equal to zero, then we will take the 23 bits from the third bit.

After computing the exponent and the mantissa of the result, the sign will be calculated as a xor operation between the operand's signs. Considering this steps a flowchart for the multiplication operation can be created:

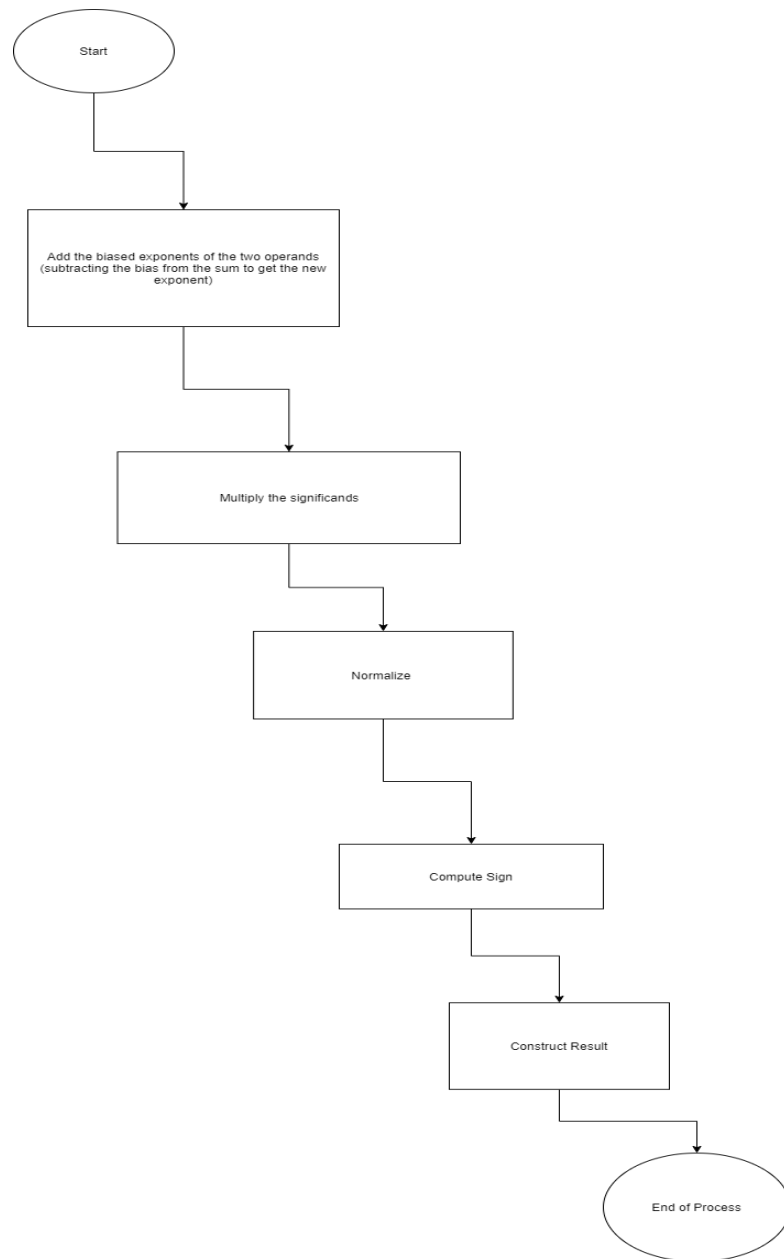


Fig.2. – Flowchart for multiplication

This approach will have to consider to following scenarios:

Overflow

The addition of the operand's exponents can overflow if the numbers are very big (a very small number will be obtained);

Underflow

The addition of the operand's exponents can underflow if the numbers are very small (they present exponents with low negative values). In this case the result can be obtained as a very big number;

Solutions:

- After some research, I found out that for the Adder, a ripple-carry adder or a carry look-ahead adder will represent valid solutions. The decision was to choose the **carry look-ahead adder**. This component will generate the carry in advance, reducing the time required to form carry signals. This type of adders use a separate block which calculates the carry output of each full adder. So, the delay for carries to ripple from stage to stage (like in the ripple carry adder's case) will be avoided.

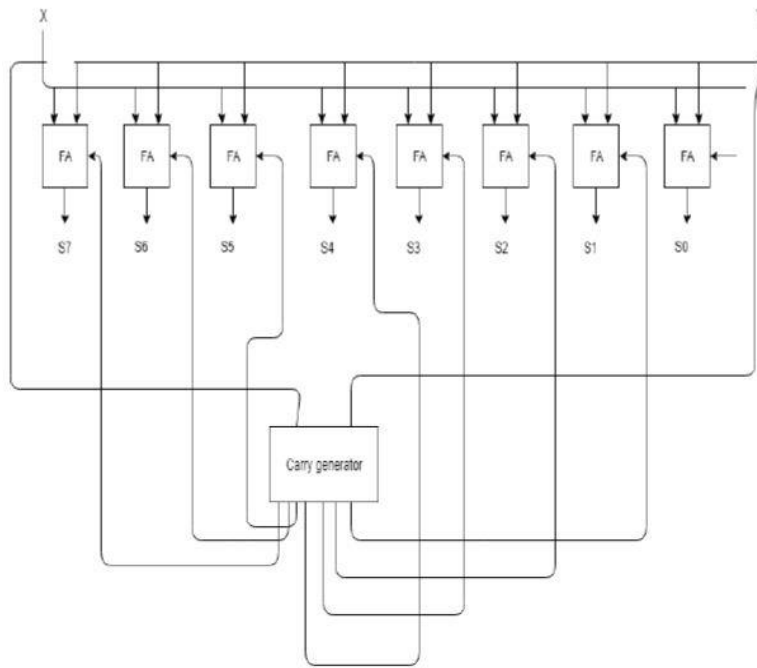


Fig.3. – Carry look-ahead Adder for 8 bit

- For the multiplication operation of the mantissas, some valid solutions would be represented by the Wallace tree or the **Booth's Algorithm**. The decision was to choose the Booth's Algorithm method, since the algorithm was meant to increase the speed, presenting a shorter period of time.
- The algorithm presents a multiplicand and a multiplier and also 2 registers : a 1-bit register (initialized with 0, responsible of storing the last bit, being capable of shifting – when the 2 operands have the same sign, adding and subtracting operations – when the operands have different signs: 0 -> 1 or 1 -> 0) and a register with the size of the result (also initialized with 0);
- **Algorithm's pseudocode:**
 - Put multiplicand in BR and multiplier in QR, the algorithm will work per following conditions:
 - 1. If Q_n and Q_{n+1} are the same (00 or 01) -> perform arithmetic shift by one bit;

- 2. If Q_n and $Q_{n+1} = 10$ -> do $A = A + BR$ and perform arithmetic shift by one bit;
 - 3. If Q_n and $Q_{n+1} = 01$ -> do $A = A - BR$ and perform arithmetic shift by one bit;
- Booth's Algorithm calculates the product in n steps (n – being the number of bits used to represent the numbers). For the final result, the algorithm quickly shifts through sequences of form 1-0 / 0-1.
 - As noticed, the Algorithm works on signed numbers, so we will add an extra bit '0' to the mantissas in order to represent only positive numbers.

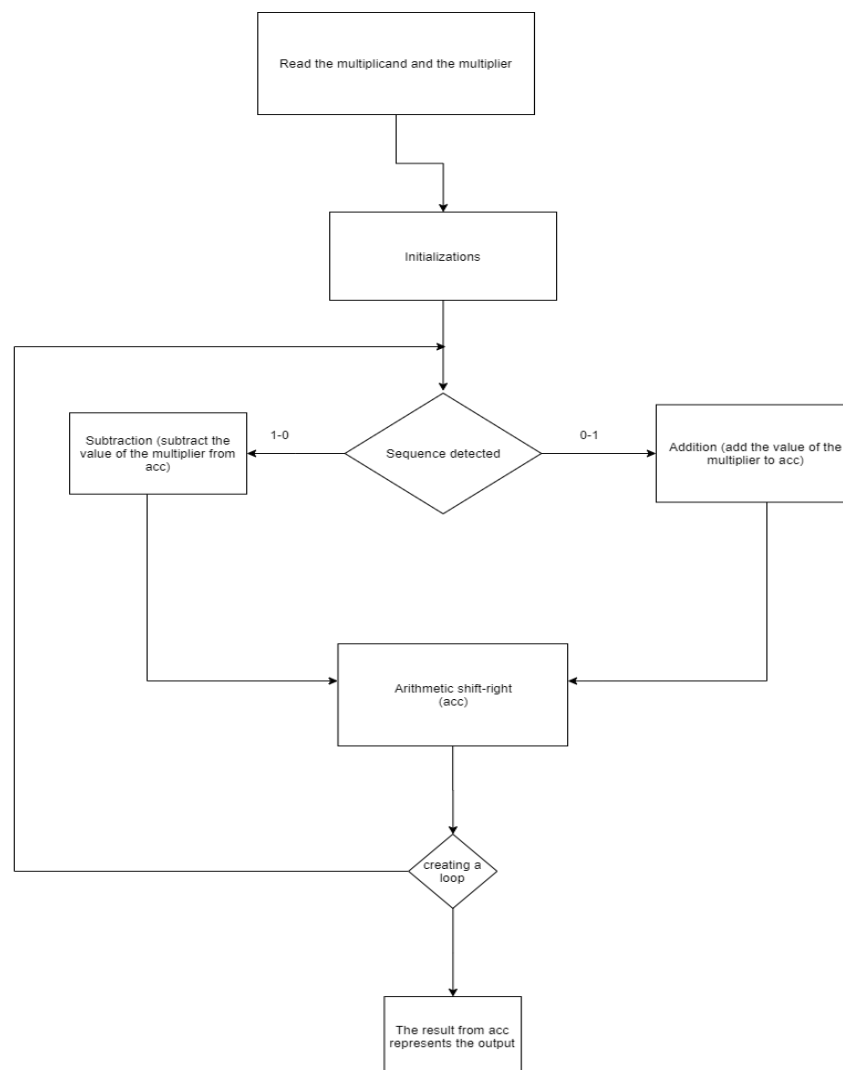


Fig.4. – Booth's Algorithm - Flowchart

3.2 Division

The four steps discussed in the previous section can also be applied in this situation.

The user has the same attributions: computing the exponents (implying the bias, but in this case the subtraction operation will take place), computing the mantissas (in this case the arithmetic operation is the division one), then normalize (if needed) and then compute the sign of the result (the same logical xor function).

A proper example (in the case in which the exponent has 4 bits and the mantissa 11) will be considered:

- The initial values are:

$$u = -6.5, v = 3.5$$

- Transforming in the needed format (S E M) we will obtain:

$$u = 1\ 1001\ 10100000000, v = 0\ 1000\ 11000000000$$

- Subtracting the exponents, we will obtain:

Exp_u	1001 –
Exp_v	1000
<hr/>	
	1001 +
	0111
<hr/>	
	1000

- The next step is represented by the computation of the mantissas:

$$110100000000 / 111000000000 = 11010 / 11100$$

11010 (dividend, u)	1110 (divisor, v)
1110 (shift and subtract)	0.1
11000	
1110 (shift and subtract)	0.11
10100	
1110 (shift and subtract)	0.111
011000	
1110 (shift twice and subtract)	0.11101
10100	
1110 (shift and subtract)	0.111011
011000	
1110 (shift twice and subtract)	0.11101101
10100	
1110 (shift and subtract)	0.111011011
011000	
1110 (shift twice and subtract)	0.11101101101 (u/v)
1010	

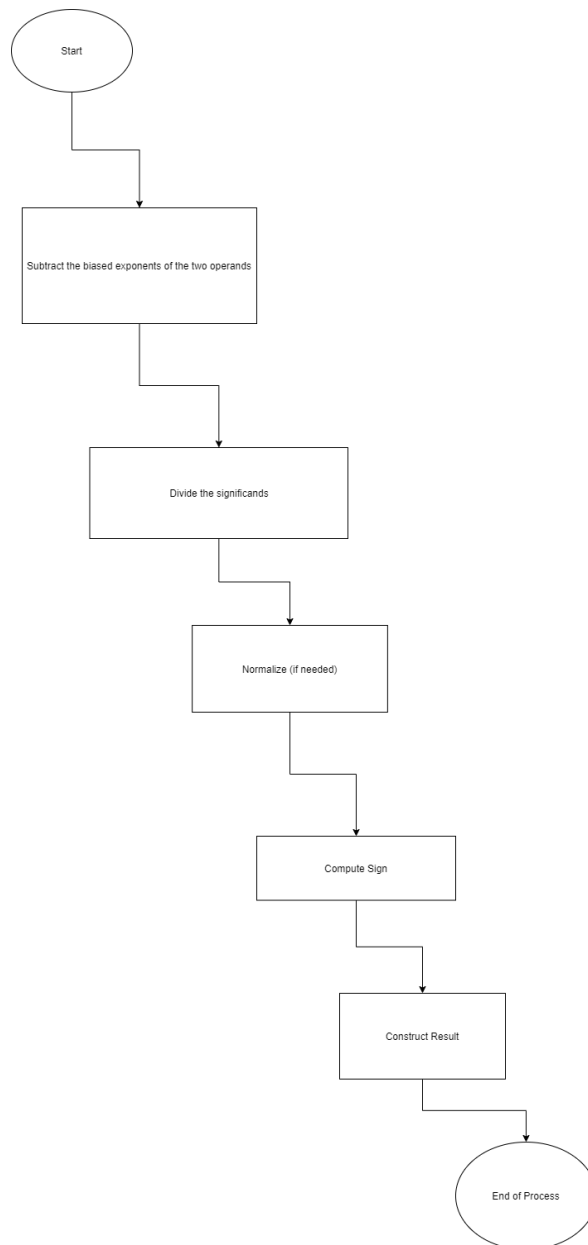


Fig.5. – Flowchart for division

This approach will have to consider to following scenarios:

Underflow

The result of the exponents subtraction (when the second operator is a very large number) will lead to an underflow, showing a bigger result than it is supposed to;

Division by zero

Solutions:

The subtraction will behave just like in the previous case in which we encountered the addition. So in this case we will also need an Adder with the configuration on 8 bits. This Adder will be created out of a cascade of full-adders.

The division will be implemented as similarly as possible to the multiplication scenario. If the division can take place, the '1' bit is shifted into the result and then the subtraction takes place. If not, the '0' bit is shifted in the result and an extra bit will be added for the next computation. The loop is maintained until the 23 bits bound is reached.

4. Design

Multiplication

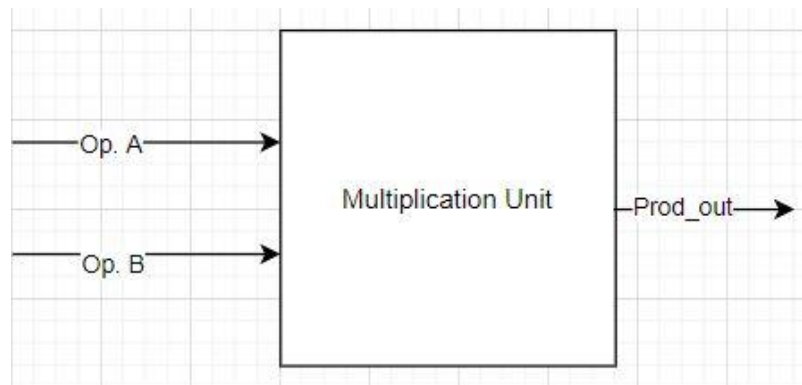


Figure 1. Design of the multiplication FPU component

The main parts for the multiplication algorithm will be represented by the Booth's Algorithm and the cascade of adders used in order to implement the Carry-Look-Ahead Adder (Figure. 3. – page 11), an adder used for the addition of the two exponents.

The justification for the adder choice is the following: this type of adder will avoid the propagation of the carry which will determine a faster computation. It will be created by cascading 8 Full-Adders (at the end, an 8-bit adder is needed) and it will also contain a separate block for the carry generation. Thus, there will be no “wait time” from carries to ripple from stage to stage (like in the ripple carry adder design).

For this component, a cascade of classical full adders will be used.

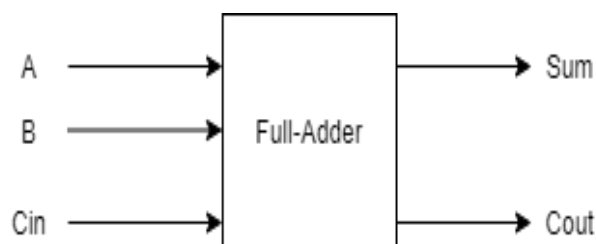
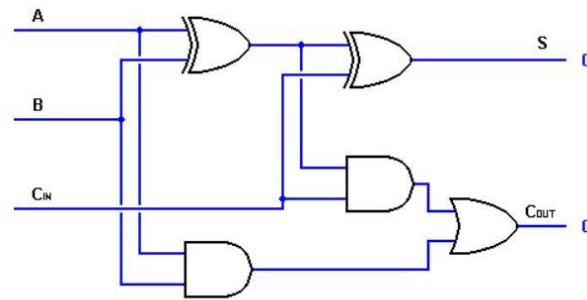


Fig.2. – One-bit Full Adder

Inputs			Outputs	
A	B	C _{in}	Sum	Carry
0	0	0	0	0
0	0	1	1	0
0	1	0	1	0
0	1	1	0	1
1	0	0	1	0
1	0	1	0	1
1	1	0	0	1
1	1	1	1	1

Fig.3. – Truth Table for a FA



Full Adder Circuit

They will be implemented based on the truth table, and a carry generation component. The carry for the next adder will be computed based on the formula: $c_{i+1} = g_i \text{ or } (p_i \text{ and } c_i)$;

In this case, two functions will be used:

- Generate signal(g): $g_i = x_i \text{ and } y_i$;
- Propagate signal(p): $p_i = x_i \text{ or } y_i$;

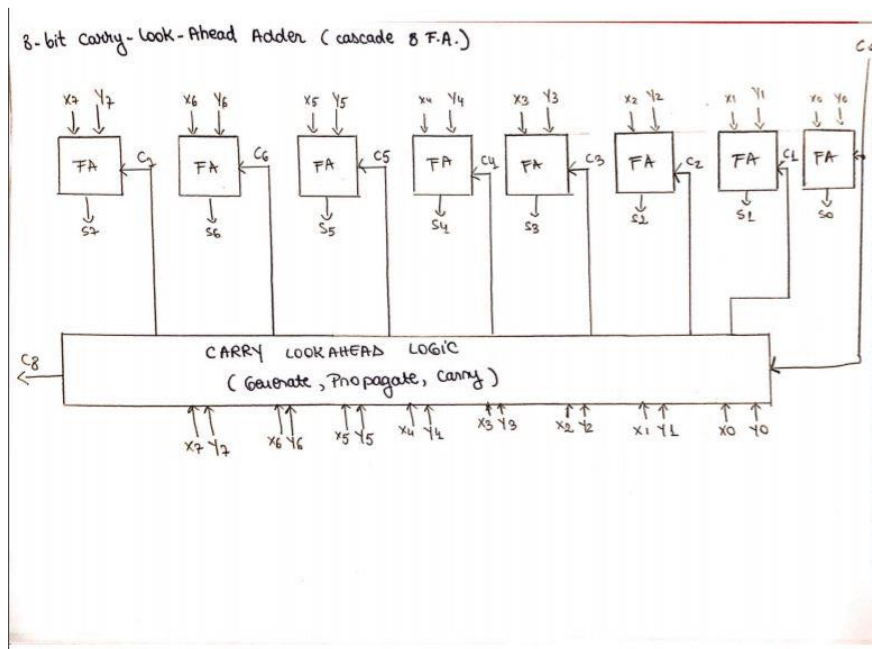


Figure 4. Design of the addition component for the exponents (Carry-look-ahead Adder).

The implementation of the adder will help in the step in which we need to compute the addition of the exponents. Their values will be biased, so we will need to subtract the bias from both values. At the final result, we will need to add back the bias along with the normalization bit obtained from the multiplication of mantissas.

Another important step will be to take care of the multiplication of the mantissas. The algorithm's flowchart can be seen in Figure 4 (page 12). The 25-bit component will be used in order to multiply the mantissas, to which we will add an extra 1-bit as the most significant one (for dealing with the signed numbers issue). So, the final result will contain a number of 48 bits. In the result, if the first bit is 1, then the number needs to be normalized. Otherwise, if the first bit is 0, then the normalization process will not be needed.

Finally, the sign will be obtained by applying the xor operation on the signs of the two operands.

After a better understanding of the required operations and after approaching the adder's situation, we need to describe the chosen components for the Booth's Algorithm.

It will be designed as a multiplier for two 25-bit numbers, and it can be described as following:

- A register will be needed with the size of our operands and one register of 1 bit responsible for the shifting part and for the addition/subtraction if needed.
- The registers will be initialized with "0" value;
- The algorithm will perform as many steps as the size of the operands (n-steps algorithm);
- If in this loop, there will be a change in the sequence of bits, the register will take care of it and for the 0-1 case (the addition takes place) and for the 1-0 case (the subtraction is performed). After the specific arithmetic operation is computed, the shift will take place.

For this, we will need an accumulator in order to perform the shifting operation and to store all the intermediary results. This accumulator will have the same size with the resulting product. An extra bit will be added for storing the last shifted out bit in order to check the changes of '0-1' or '1-0'. A real example containing small numbers will help with the understanding of the algorithm:

$$A = 0011\ 0000\ 0, S = 1101\ 0000\ 0, P = 0000\ 1100\ 0$$

The first bit will initially be zero, the low part of the accumulator is initialized with the multiplicand, and then the main loop can start. The loop will repeat itself as many times as the size of the operands (in our example that will be 4 times; in the project the loop will iterate 25 times).

1. $P = 0000\ 1100\ 0$: last two bits are 00 -> just shift: $0000\ 0110\ 0$
2. $P = 0000\ 0110\ 0$: last two bits are 00 -> just shift: $0000\ 0011\ 0$
3. $P = 0000\ 0011\ 0$: last two bits are 10 -> $(P+S)$ -> $P = 1101\ 0011\ 0$ -> shift: $1110\ 1001\ 1$
4. $P = 1110\ 1001\ 1$: last two bits are 11 -> just shift: $1111\ 0100\ 1$
 $P = 1111\ 0100$ (final result)

In each iteration, we will check if there are bits fluctuations (01 – store in the high part of the accumulator the addition between itself and the multiplier or 10 – store in the high part of the accumulator the difference between itself and the multiplier). If the last two bits are the same, then we only do the shifting part (keeping the sign). At the end, the result is represented by the two parts of the accumulator (high and low).

```
graph TD
    Multiplier["Multiplier  
(2nd operand)"] --> Arithmetic["Arithmetic operation"]
    Multiplicand["Multiplicand  
(1st operand)"] --> HighLow[High | Low]
    Multiplicand --> Control
    Arithmetic --> HighLow
    HighLow --> Product["Product"]
    HighLow --> Control
    Control -- "write" --> HighLow
    Control -- "shift" --> HighLow
    Control --> Arithmetic
    Control --> Product
```

Figure 5. Design of the multiplication's Booth Algorithm.

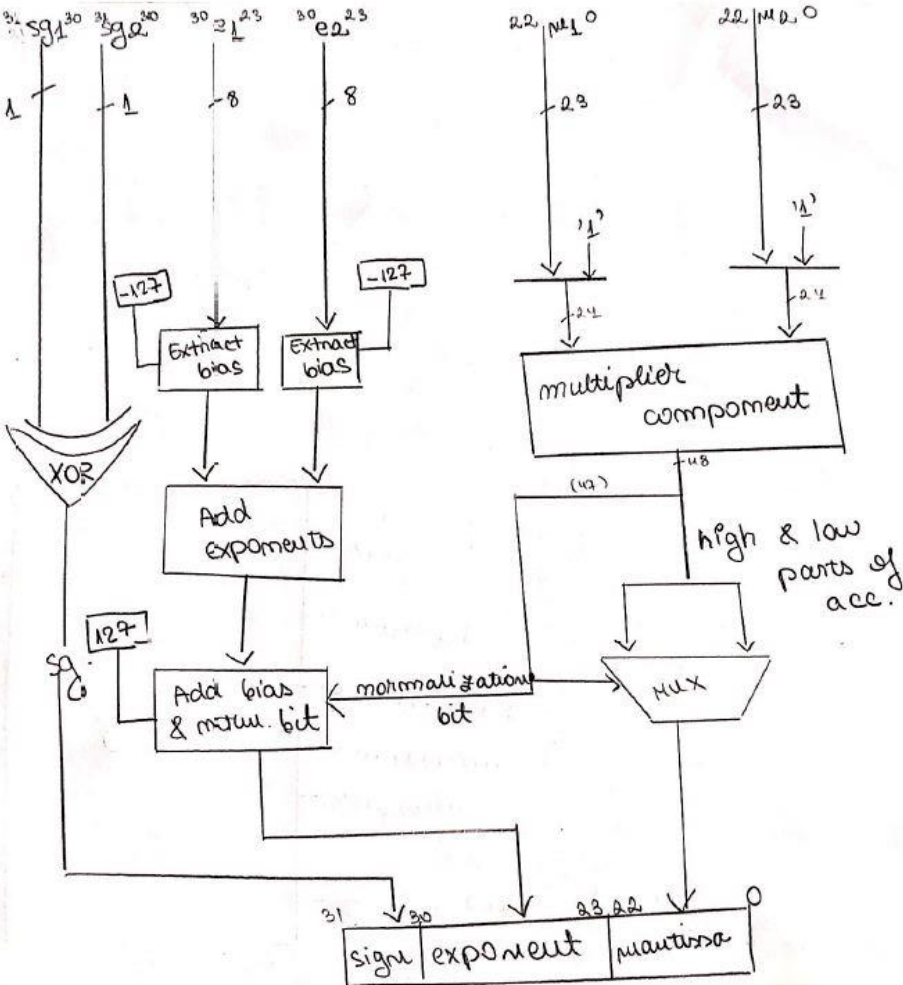


Figure 6. High-Level Design of the multiplication FPU component.

Division

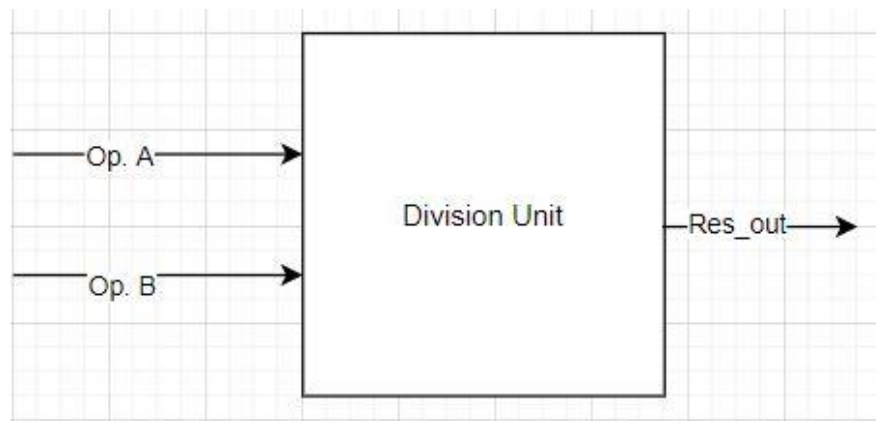


Figure 7. Design of the division FPU component.

In this case, the exponents will have to perform the subtraction operation. As a better solution, to reduce the number of external components, we can obtain the two's complement of the second operand and perform the addition of the two numbers (since the addition component already exists and it is described). The exponents will be complemented for obtaining their negative value. After that, they can represent the inputs for the carry-look-ahead adder (in this way a subtraction can be performed).

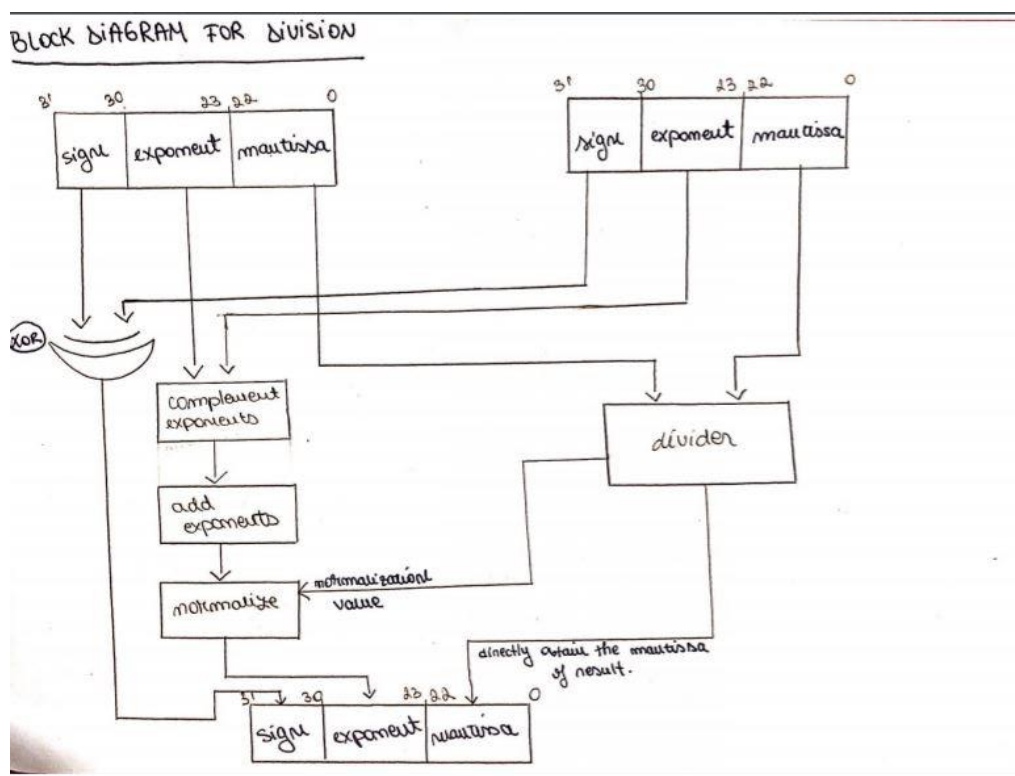


Figure 8. High-Level Design of the division FPU component

The project will need to contain a separate method for dividing the two mantissas. The component is designed to produce the division result of the mantissas and the normalization value. This value will be used to reduce the exponent (in order for the result to be normalized).

The two numbers become the dividend and the divisor. The method starts by checking if the divisor fits in the dividend (if the two numbers can successfully be divided). If the condition is satisfied then we add a '1' for the result and then subtract the divisor from the dividend.

The new result will take the place of the first dividend and the steps will repeat. Otherwise, a '0' will be added to the result. Then, we will move to the right with one position and the steps will be repeated. At the end, the normalization value will be obtained by the difference between the initial point that was found and the result. It will be used in the exponent's computation.

The FPU core was designed to be as modular as possible. The current core will support the two arithmetic operations: multiplication and division. The two arithmetic operations will be connected using a Control Unit.

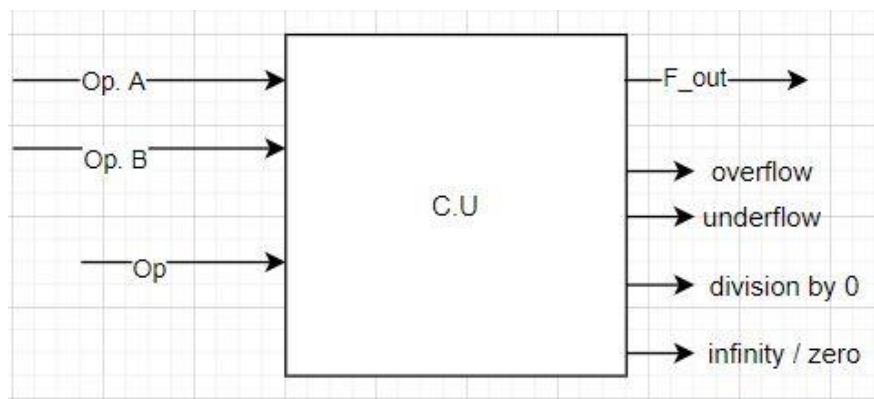


Figure 9. Initial Design of the Control Unit.

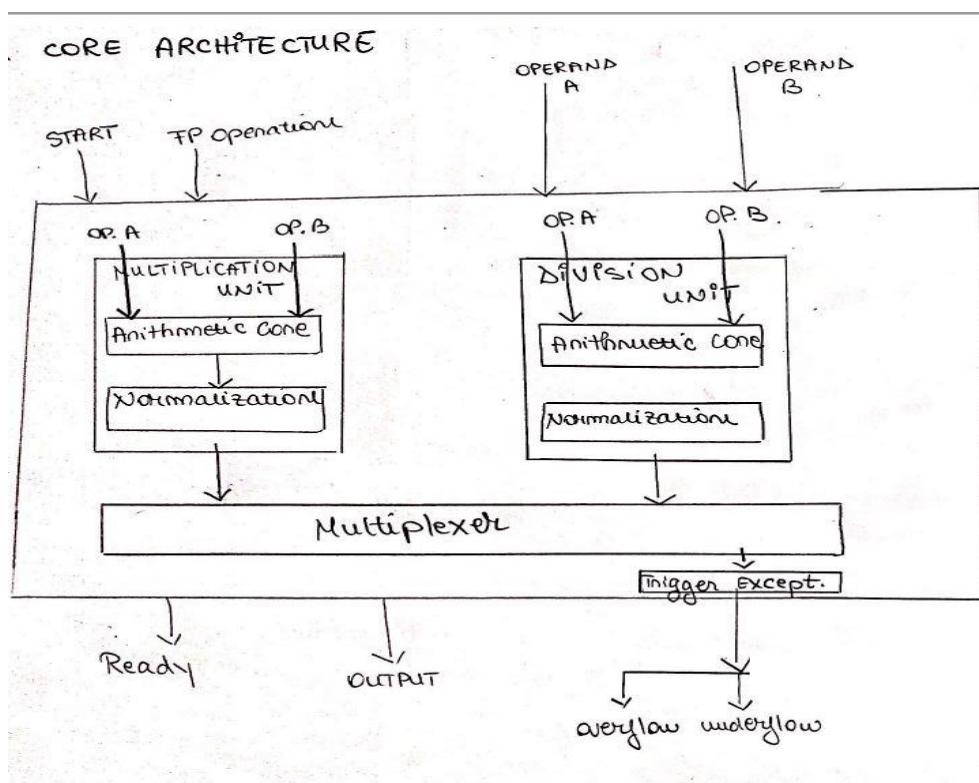


Figure 10. Core Architecture of FPU. How C.U communicate with the arithmetic blocks.

The Control Unit is the principal “force” which will control how to components are being handled. It defines what goals are being performed and what the next requirement has to do.

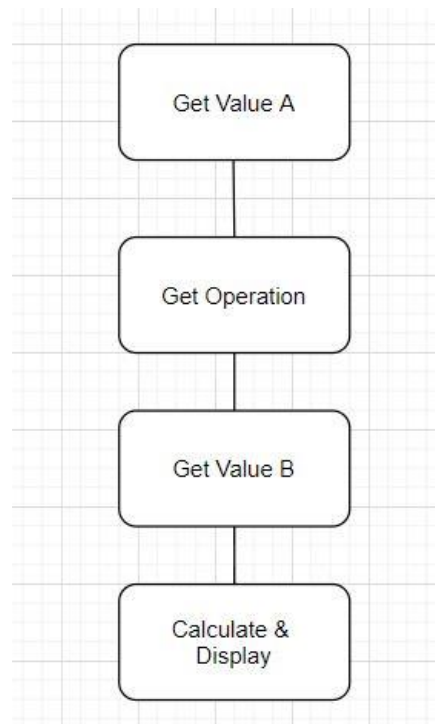


Figure 11. The Order of Steps for the Control Unit.

It will make sure that the correct operation is computed, making use of all the involved components. The other arithmetic unit will be “disabled” by modifying the output multiplexer code (since the two units are totally independent, the only similarity being that they use a common component).

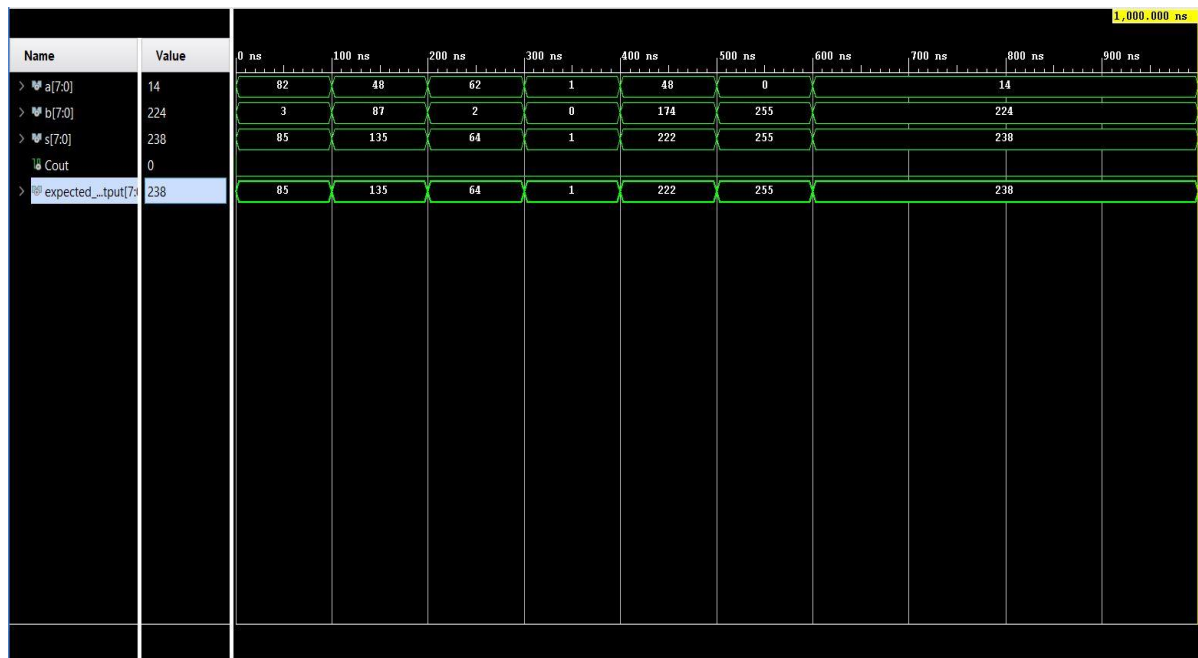
5. Implementation & Testing

Multiplication

As discussed in the Design section, the components were structured following the provided diagrams.

A Full Adder, on 1 bit, was implemented in order to form a loop of 8 steps, obtaining a cascading of eight full-adders. They will be connected to the carry component, which will generate the carry in advance. The carry generator will also present an 8 steps loop for a faster approach (than implementing 8 1-bit carry generators).

Then, by connecting the two addition components, the Adder required for the addition of the exponents will be implemented.

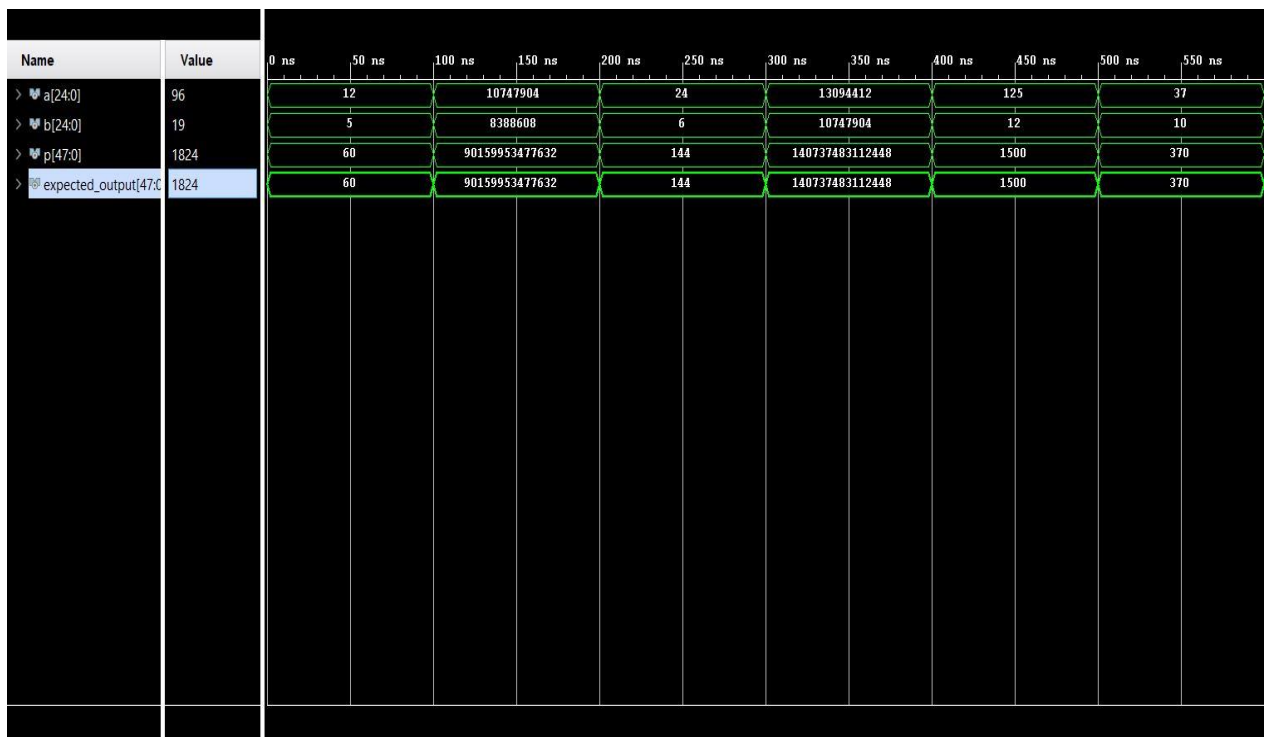


Then, the implementation will continue focusing on the multiplication component, designed for the mantissas.

The *Booth's Algorithm* will be designed in a behavioural architecture, by following the analysis and design's pseudocode and the described steps.

The accumulator will be declared as a signal (acc) and it will have the length of the obtained product along with an extra bit for the shifting procedures.

The bit fluctuations (0-1 or 1-0) will be checked and the proper arithmetic operation will take place, the final step being the shifting of the entire variable by 1 bit which will take place regarding of the case in which we are in.



Finally, the two components together will form the *FPU Multiplication feature*. We will extract the number components of the two inputs (sign, exponent, mantissa) and then we will apply the proper operation on them. The product obtained from the multiplication of the mantissas will be checked in order to know if the final result needs to be normalized or not (if yes, an ‘1’ will be added to the final sum of the exponents using the adder component).

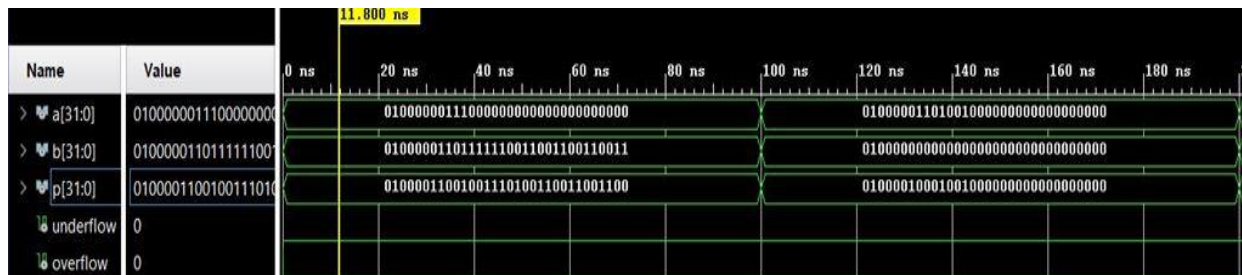
Using the “port map” method, the exponents will perform all the arithmetic additions (removing the bias, adding the unbiased exponents, adding back the bias and normalize if needed).

The *FPU Multiplication* component will also check if the exponents will create an overflow or an underflow situation. If there are some C-out signals remaining then the operation encountered an issue. We mark the issue by attributing ‘1’ to the signal of the current problem.

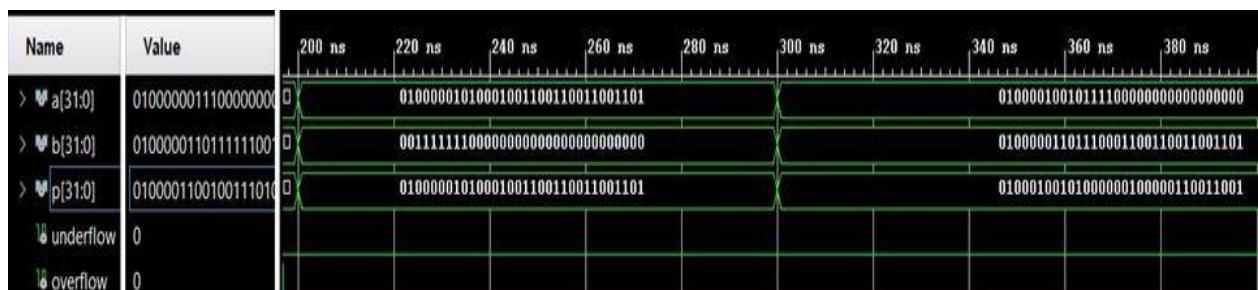
Finally, the sign will be computed by using the “xor” logical gate.

The result will be represented by the product “p” variable which will contain all the components combined in a full representation of a floating point number.

1. $a = 7, b = 23.9, p = 167.3$
2. $a = 20.5, b = 2, p = 41$



3. $a = 12.3, b = 1, p = 12.3$
4. $a = 55.5, b = 23.1, p = 1282.05$



More examples in the floating-point representation will be given in the Appendix 1.

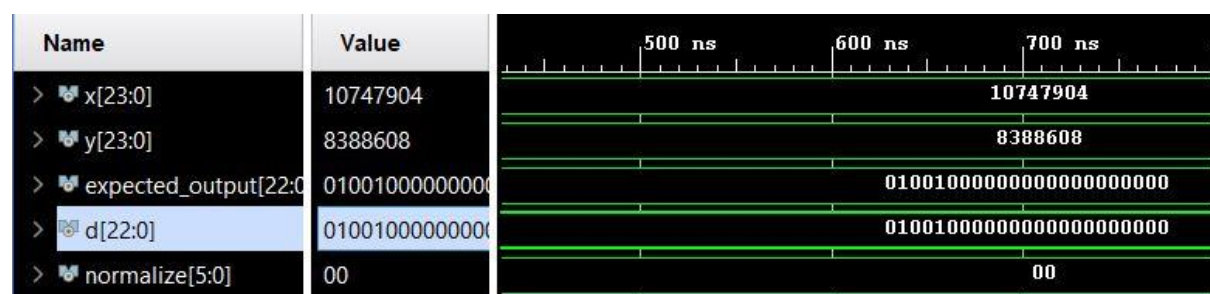
Division

The other operation will also be implemented as discussed in the Design section, the components being structured by following the provided diagrams.

The difference from the first operation presented is that the exponents need to be complemented in order to obtain their negative value. Only after this step, they will enter the port-map operations. In this way, we can use the adder already implemented to perform the “subtraction” of the exponents.

Another difference will be the separate division component. This component will produce as a result the final mantissa representation and the normalization value, used later in the exponents operation (it represents the number of times in which we have to reduce the exponent in order for the result to be normalized).

The *Division Component* will be used for the division of the two mantissas of the input values. The division will present a series of steps implemented after the idea of how a normal division between two numbers takes place.



The two components together (taking the Adder already presented in the Multiplication Section) will form the FPU Division feature. We will extract the number components of the two inputs (sign, exponent, mantissa) and then we will apply the proper operation on them.

From the division of the mantissas we can check what the normalization value will be. This result will be subtracted from the exponents final result. In order to do this, we will have to transform it to its 2's complement representation.

Using the “port map” method, the exponents will perform all the arithmetic additions (removing the bias, adding the unbiased exponents, adding back the bias and normalize if needed).

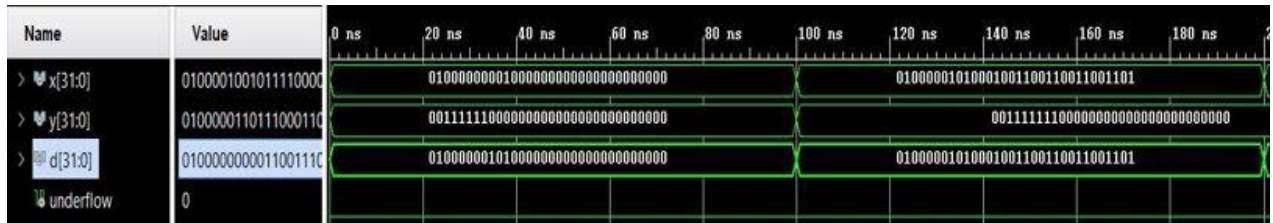
In this case, the exponents should be subtracted, but we can perform this operation using the Adder that we already have along with a conversion of the second exponent.

The *FPU Division component* will check if the exponents will create an underflow situation. If there are some C-out signals remaining then the operation encountered an issue. We mark the issue by attributing ‘1’ to the signal of the current problem.

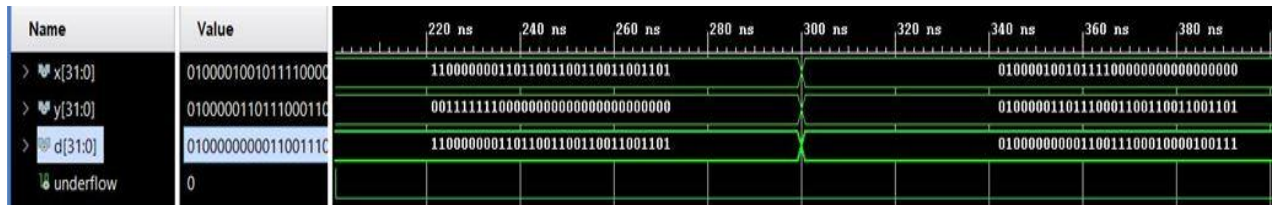
Finally, the sign will be computed by using the “xor” logical gate.

The result will be represented by the “d” variable which will contain all the components combined in a full representation of a floating point number.

1. $x = 2.5$, $y = 0.5$, $d = 5$
2. $x = 12.3$, $y = 1.0$, $d = 12.3$



3. $x = -3.7$, $y = 1.0$, $d = -3.7$
4. $x = 55.5$, $y = 23.1$, $d = 2.40$



More examples in the floating-point representation will be given in the Appendix 1.

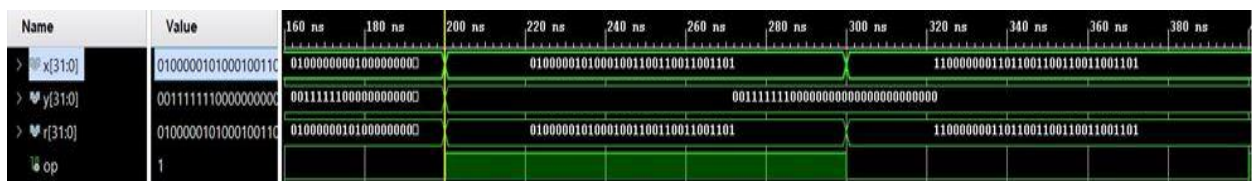
Finally, the two FPU components will be connected in a top level file. As inputs this *Control Unit* will present the two 32 bits numbers, the result obtained after performing a certain operation and a 1-bit selection which decides what operation is requested.

Using the port-map method, we perform the operation and obtain the wanted result.

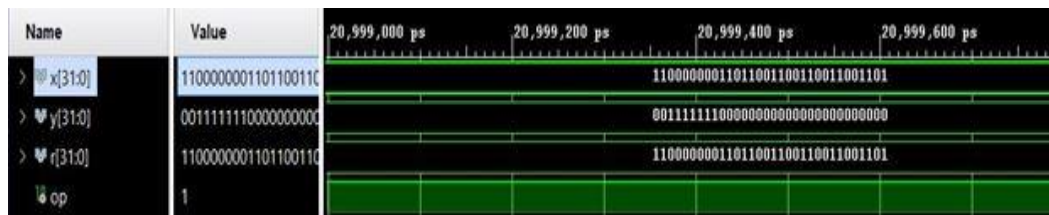
A process will be created in order for the program to know which result is wanted by the user in the simulation. So, we decided that if the “selection” will take the value ‘1’ then the multiplication of the two numbers will take place. Otherwise, if the “selection” is equal to ‘0’ then the division is requested and performed.

The flags for special cases were also included so the simulation can be as accurate as possible to any combination of numbers.

1. $a = 12.3$, $b = 1.0$, $op = '1'$ (the multiplication of the two numbers is requested).
2. $a = -3.7$, $b = 1.0$, $op = '0'$ (division)



3. $a = -3.7$, $b = 1.0$, $op = '1'$ (multiplication)

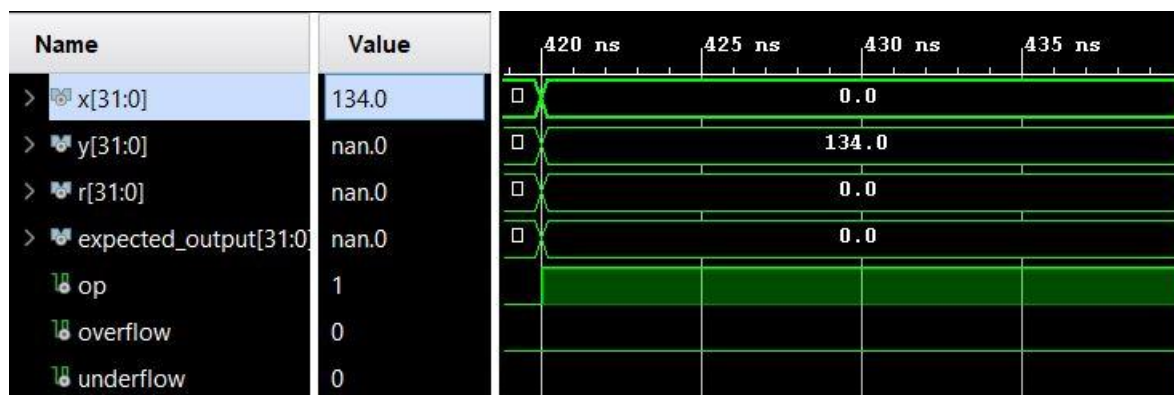


More examples in the floating-point representation will be given in the Appendix 1.

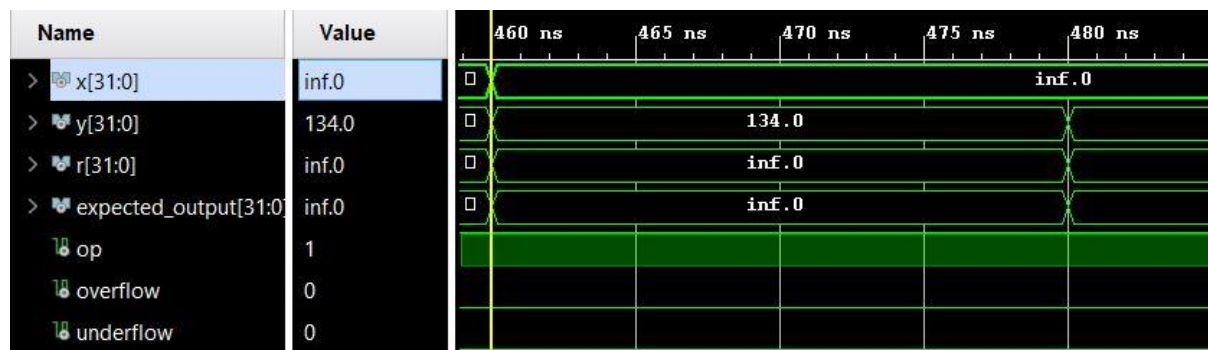
SPECIAL CASES

The 2 operands will be represented by the floating point numbers: x, y;

- *Multiplication with 0* : x = 0, y = 134;



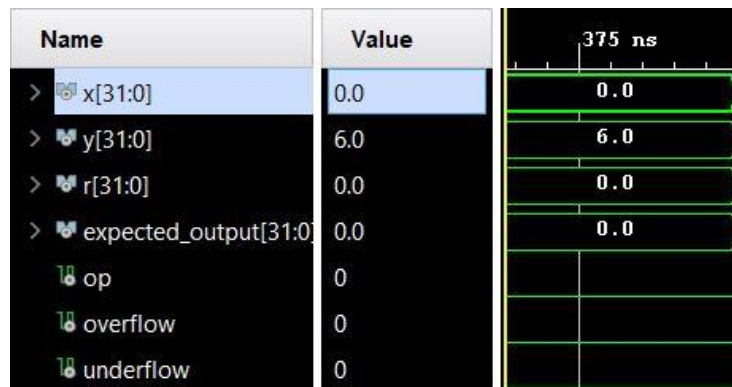
- *Multiplication with infinity*: x = + inf, y = 134;



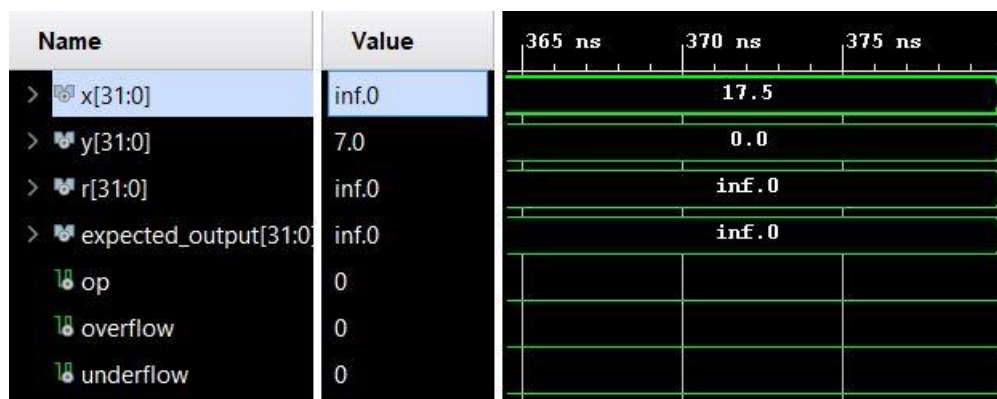
- *Multiplication with NaN*: x = 134, y = NaN



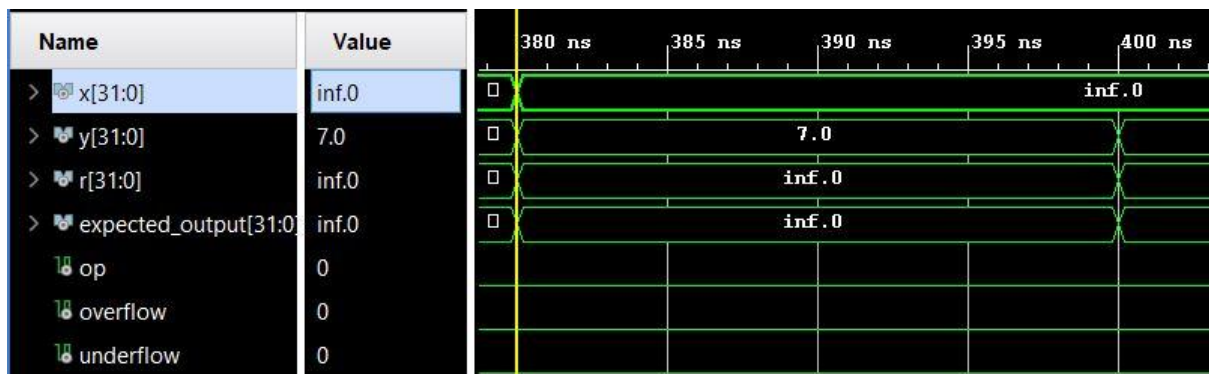
➤ *Division with 0: $x = 0$, $y = 6$*



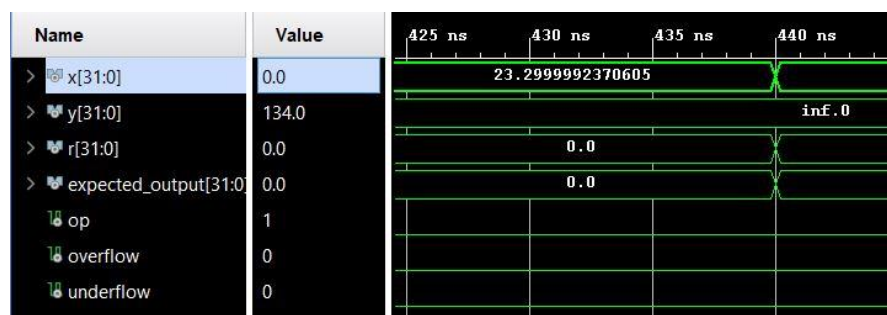
➤ *Division by 0: $x = 17.5$, $y = 0$*



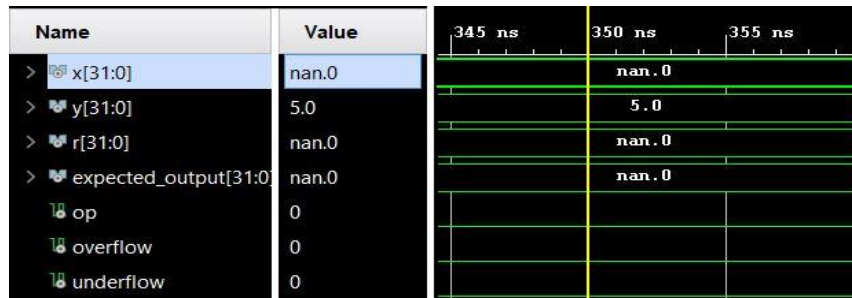
➤ *Division with infinity: $x = +\text{inf}$, $y = 7$*



➤ *Division by infinity: $x = 23.2999992370605$, $y = +\text{inf}$*



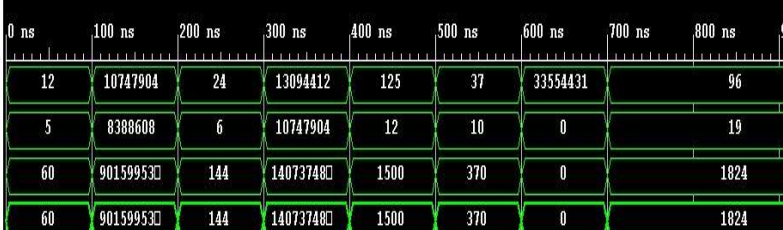
➤ *Division with NaN:* $x = \text{NaN}$, $y = 5$

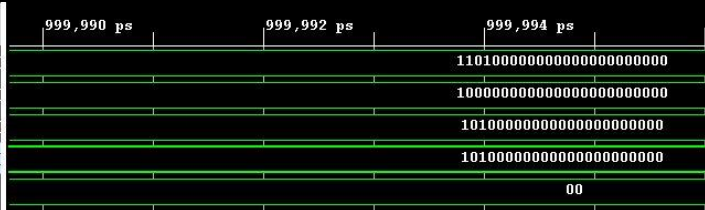


APPENDIX 1

In this section we will represent the results obtained in all the previous cases. An expected output variable will be created and it will contain the correct result of the performed operation.


Component	We will start with the first component implemented: The Addition Component.																																																																											
Explanations	Different numbers will be chosen in the testbench and the addition will be performed. We can see that the sum and the expected output have the same values, with no remaining C-out, so the operation is well managed.																																																																											
Picture	<table><tr><th>Name</th><th>Value</th><th>0 ns</th><th>100 ns</th><th>200 ns</th><th>300 ns</th><th>400 ns</th><th>500 ns</th><th>600 ns</th><th>700 ns</th><th>800 ns</th></tr><tr><td>> a[7:0]</td><td>14</td><td>82</td><td>48</td><td>62</td><td>1</td><td>48</td><td>0</td><td colspan="3">14</td></tr><tr><td>> b[7:0]</td><td>224</td><td>3</td><td>87</td><td>2</td><td>0</td><td>174</td><td>255</td><td colspan="3">224</td></tr><tr><td>> s[7:0]</td><td>238</td><td>85</td><td>135</td><td>64</td><td>1</td><td>222</td><td>255</td><td colspan="3">238</td></tr><tr><td>↓ Cout</td><td>0</td><td></td><td></td><td></td><td></td><td></td><td></td><td></td><td></td><td></td></tr><tr><td>> expected_output[7:0]</td><td>238</td><td>85</td><td>135</td><td>64</td><td>1</td><td>222</td><td>255</td><td colspan="3">238</td></tr></table>										Name	Value	0 ns	100 ns	200 ns	300 ns	400 ns	500 ns	600 ns	700 ns	800 ns	> a[7:0]	14	82	48	62	1	48	0	14			> b[7:0]	224	3	87	2	0	174	255	224			> s[7:0]	238	85	135	64	1	222	255	238			↓ Cout	0										> expected_output[7:0]	238	85	135	64	1	222	255	238		
Name	Value	0 ns	100 ns	200 ns	300 ns	400 ns	500 ns	600 ns	700 ns	800 ns																																																																		
> a[7:0]	14	82	48	62	1	48	0	14																																																																				
> b[7:0]	224	3	87	2	0	174	255	224																																																																				
> s[7:0]	238	85	135	64	1	222	255	238																																																																				
↓ Cout	0																																																																											
> expected_output[7:0]	238	85	135	64	1	222	255	238																																																																				

Component	The second component implemented were: The Multiplication Component.																				
Explanations	Different numbers will be chosen in the testbench and the multiplication of the mantissas will be performed. We can see that the product and the expected output have the same values, so the operation is well managed.																				
Picture	<table><thead><tr><th>Name</th><th>Value</th></tr></thead><tbody><tr><td>> a[24:0]</td><td>96</td></tr><tr><td>> b[24:0]</td><td>19</td></tr><tr><td>> p[47:0]</td><td>1824</td></tr><tr><td>> expected output[47:0]</td><td>1824</td></tr></tbody></table>		Name	Value	> a[24:0]	96	> b[24:0]	19	> p[47:0]	1824	> expected output[47:0]	1824									
	Name	Value																			
	> a[24:0]	96																			
	> b[24:0]	19																			
	> p[47:0]	1824																			
> expected output[47:0]	1824																				

Component	For the division operation the component implemented were: The Division Component.														
Explanations	Different numbers will be chosen in the testbench and the division of the mantissas will be performed. We can see that the result and the expected output have the same values, and the numbers do not need to be normalized so the operation is well managed.														
Picture	<table><tr><th>Name</th><th>Value</th></tr><tr><td>> x[23:0]</td><td>1101000000000</td></tr><tr><td>> y[23:0]</td><td>1000000000000</td></tr><tr><td>> expected_output[22:0]</td><td>1010000000000</td></tr><tr><td>> d[22:0]</td><td>1010000000000</td></tr><tr><td>> normalize[5:0]</td><td>00</td></tr></table>	Name	Value	> x[23:0]	1101000000000	> y[23:0]	1000000000000	> expected_output[22:0]	1010000000000	> d[22:0]	1010000000000	> normalize[5:0]	00		
Name	Value														
> x[23:0]	1101000000000														
> y[23:0]	1000000000000														
> expected_output[22:0]	1010000000000														
> d[22:0]	1010000000000														
> normalize[5:0]	00														

After all the components were implemented and tested, the two main parts (responsible for the arithmetic operations) of the FPU need to be checked.

Component	The FPU Multiplication Component.																																																																																							
Explanations	<p>The Multiplication operation will be performed fully by the FPU. Some arithmetic operations will take place on all the components (exponent, mantissa) and the xor gate for the signs.</p> <p>Some numbers of different forms (negative/positive, integers/real) shown in the floating point representation were chosen to confirm the behaviour of the project.</p> <p>The numbers are written in the floating point single precision representation.</p>																																																																																							
Picture	<table><thead><tr><th>Name</th><th>Value</th></tr></thead><tbody><tr><td>> a[31:0]</td><td>7.0</td></tr><tr><td>> b[31:0]</td><td>13.0</td></tr><tr><td>> p[31:0]</td><td>91.0</td></tr><tr><td>> expected_output[31:0]</td><td>91.0</td></tr><tr><td>underflow</td><td>0</td></tr><tr><td>overflow</td><td>0</td></tr></tbody></table>		Name	Value	> a[31:0]	7.0	> b[31:0]	13.0	> p[31:0]	91.0	> expected_output[31:0]	91.0	underflow	0	overflow	0	<table><thead><tr><th>0 ns</th><th>100 ns</th><th>200 ns</th><th>300 ns</th><th>400 ns</th><th>500 ns</th><th>600 ns</th><th>700 ns</th><th>800 ns</th></tr></thead><tbody><tr><td>7.0</td><td>20.5</td><td>-14.5</td><td>55.5</td><td>-23.7000</td><td>224.0</td><td colspan="3">255.0</td></tr><tr><td>13.0</td><td>2.0</td><td>-0.375</td><td>-23.0</td><td>-18.0</td><td>14.0</td><td colspan="3">255.0</td></tr><tr><td>91.0</td><td>41.0</td><td>5.4375</td><td>-1276.5</td><td>426.6000</td><td>3136.0</td><td colspan="3">65025.0</td></tr><tr><td>91.0</td><td>41.0</td><td>5.4375</td><td>-1276.5</td><td>426.6000</td><td>3136.0</td><td colspan="3">65025.0</td></tr><tr><td></td><td></td><td></td><td></td><td></td><td></td><td colspan="3"></td></tr><tr><td></td><td></td><td></td><td></td><td></td><td></td><td colspan="3"></td></tr></tbody></table>									0 ns	100 ns	200 ns	300 ns	400 ns	500 ns	600 ns	700 ns	800 ns	7.0	20.5	-14.5	55.5	-23.7000	224.0	255.0			13.0	2.0	-0.375	-23.0	-18.0	14.0	255.0			91.0	41.0	5.4375	-1276.5	426.6000	3136.0	65025.0			91.0	41.0	5.4375	-1276.5	426.6000	3136.0	65025.0																				
Name	Value																																																																																							
> a[31:0]	7.0																																																																																							
> b[31:0]	13.0																																																																																							
> p[31:0]	91.0																																																																																							
> expected_output[31:0]	91.0																																																																																							
underflow	0																																																																																							
overflow	0																																																																																							
0 ns	100 ns	200 ns	300 ns	400 ns	500 ns	600 ns	700 ns	800 ns																																																																																
7.0	20.5	-14.5	55.5	-23.7000	224.0	255.0																																																																																		
13.0	2.0	-0.375	-23.0	-18.0	14.0	255.0																																																																																		
91.0	41.0	5.4375	-1276.5	426.6000	3136.0	65025.0																																																																																		
91.0	41.0	5.4375	-1276.5	426.6000	3136.0	65025.0																																																																																		

Component	The FPU Division Component.																																																																											
Explanations	<p>The Division operation will be performed fully by the FPU. Some arithmetic operations will take place on all the components (exponent, mantissa) and the xor gate for the signs.</p> <p>Some numbers of different forms (negative/positive, integers/real) were chosen to confirm the behaviour of the project.</p>																																																																											
Picture	 <table border="1"> <thead> <tr> <th>Name</th> <th>Value</th> <th>0 ns</th> <th>100 ns</th> <th>200 ns</th> <th>300 ns</th> <th>400 ns</th> <th>500 ns</th> <th>600 ns</th> <th>700 ns</th> <th>800 ns</th> </tr> </thead> <tbody> <tr> <td>x[31:0]</td> <td>1000.0</td> <td>7.0</td> <td>2.5</td> <td>-3.70000</td> <td>1000.0</td> <td>113.0</td> <td>224.0</td> <td>255.0</td> <td></td> <td>20.5</td> </tr> <tr> <td>y[31:0]</td> <td>10.0</td> <td>2.0</td> <td>0.5</td> <td>1.0</td> <td>10.0</td> <td>25.0</td> <td>14.0</td> <td>255.0</td> <td></td> <td>2.0</td> </tr> <tr> <td>d[31:0]</td> <td>100.0</td> <td>3.5</td> <td>5.0</td> <td>-3.70000</td> <td>100.0</td> <td>4.519999</td> <td>16.0</td> <td>1.0</td> <td></td> <td>10.25</td> </tr> <tr> <td>expected_output[31:0]</td> <td>100.0</td> <td>3.5</td> <td>5.0</td> <td>-3.70000</td> <td>100.0</td> <td>4.519999</td> <td>16.0</td> <td>1.0</td> <td></td> <td>10.25</td> </tr> <tr> <td>underflow</td> <td>0</td> <td></td> <td></td> <td></td> <td></td> <td></td> <td></td> <td></td> <td></td> <td></td> </tr> </tbody> </table>										Name	Value	0 ns	100 ns	200 ns	300 ns	400 ns	500 ns	600 ns	700 ns	800 ns	x[31:0]	1000.0	7.0	2.5	-3.70000	1000.0	113.0	224.0	255.0		20.5	y[31:0]	10.0	2.0	0.5	1.0	10.0	25.0	14.0	255.0		2.0	d[31:0]	100.0	3.5	5.0	-3.70000	100.0	4.519999	16.0	1.0		10.25	expected_output[31:0]	100.0	3.5	5.0	-3.70000	100.0	4.519999	16.0	1.0		10.25	underflow	0									
Name	Value	0 ns	100 ns	200 ns	300 ns	400 ns	500 ns	600 ns	700 ns	800 ns																																																																		
x[31:0]	1000.0	7.0	2.5	-3.70000	1000.0	113.0	224.0	255.0		20.5																																																																		
y[31:0]	10.0	2.0	0.5	1.0	10.0	25.0	14.0	255.0		2.0																																																																		
d[31:0]	100.0	3.5	5.0	-3.70000	100.0	4.519999	16.0	1.0		10.25																																																																		
expected_output[31:0]	100.0	3.5	5.0	-3.70000	100.0	4.519999	16.0	1.0		10.25																																																																		
underflow	0																																																																											

Component	Final FPU.																																																																																																																												
Explanations	<p>Finally, the two components were connected using a top-level file. This file will take the role of a Control Unit in which a selection is presented. This will let the user chose the wanted operation. Same here, an expected output variable will be present to check if the obtained result is the right one.</p> <p>Some numbers of different forms (negative/positive, integers/real) shown in the floating point representation were chosen to confirm the behaviour of the project.</p>																																																																																																																												
Picture	<table><tr><th>Name</th><th>Value</th><th>0 ns</th><th>20 ns</th><th>40 ns</th><th>60 ns</th><th>80 ns</th><th>100 ns</th><th>120 ns</th><th>140 ns</th><th>160 ns</th><th>180 ns</th><th>200 ns</th><th>220 ns</th></tr><tr><td>x[31:0]</td><td>17.5</td><td>1000.0</td><td>2.5</td><td>134.0</td><td>255.0</td><td>224.0</td><td>20.5</td><td>255.0</td><td>113.0</td><td>-14.5</td><td>7.0</td><td></td><td></td></tr><tr><td>y[31:0]</td><td>0.0</td><td>10.0</td><td>379.0</td><td>0.5</td><td>-2.5</td><td>255.0</td><td>14.0</td><td>2.0</td><td>255.0</td><td>25.0</td><td>-0.375</td><td>2.0</td><td>13.0</td></tr><tr><td>r[31:0]</td><td>inf.0</td><td>100.0</td><td>379000.0</td><td>5.0</td><td>-335.0</td><td>1.0</td><td>3136.0</td><td>10.25</td><td>65025.0</td><td>4.519999</td><td>5.4375</td><td>3.5</td><td>91.0</td></tr><tr><td>expected_output[31:0]</td><td>inf.0</td><td>100.0</td><td>379000.0</td><td>5.0</td><td>-335.0</td><td>1.0</td><td>3136.0</td><td>10.25</td><td>65025.0</td><td>4.519999</td><td>5.4375</td><td>3.5</td><td>91.0</td></tr><tr><td>op</td><td>0</td><td></td><td></td><td></td><td></td><td></td><td></td><td></td><td></td><td></td><td></td><td></td><td></td></tr><tr><td>overflow</td><td>0</td><td></td><td></td><td></td><td></td><td></td><td></td><td></td><td></td><td></td><td></td><td></td><td></td></tr><tr><td>underflow</td><td>0</td><td></td><td></td><td></td><td></td><td></td><td></td><td></td><td></td><td></td><td></td><td></td><td></td></tr></table>													Name	Value	0 ns	20 ns	40 ns	60 ns	80 ns	100 ns	120 ns	140 ns	160 ns	180 ns	200 ns	220 ns	x[31:0]	17.5	1000.0	2.5	134.0	255.0	224.0	20.5	255.0	113.0	-14.5	7.0			y[31:0]	0.0	10.0	379.0	0.5	-2.5	255.0	14.0	2.0	255.0	25.0	-0.375	2.0	13.0	r[31:0]	inf.0	100.0	379000.0	5.0	-335.0	1.0	3136.0	10.25	65025.0	4.519999	5.4375	3.5	91.0	expected_output[31:0]	inf.0	100.0	379000.0	5.0	-335.0	1.0	3136.0	10.25	65025.0	4.519999	5.4375	3.5	91.0	op	0													overflow	0													underflow	0												
Name	Value	0 ns	20 ns	40 ns	60 ns	80 ns	100 ns	120 ns	140 ns	160 ns	180 ns	200 ns	220 ns																																																																																																																
x[31:0]	17.5	1000.0	2.5	134.0	255.0	224.0	20.5	255.0	113.0	-14.5	7.0																																																																																																																		
y[31:0]	0.0	10.0	379.0	0.5	-2.5	255.0	14.0	2.0	255.0	25.0	-0.375	2.0	13.0																																																																																																																
r[31:0]	inf.0	100.0	379000.0	5.0	-335.0	1.0	3136.0	10.25	65025.0	4.519999	5.4375	3.5	91.0																																																																																																																
expected_output[31:0]	inf.0	100.0	379000.0	5.0	-335.0	1.0	3136.0	10.25	65025.0	4.519999	5.4375	3.5	91.0																																																																																																																
op	0																																																																																																																												
overflow	0																																																																																																																												
underflow	0																																																																																																																												

4. Conclusions

This project had as a main objective designing, implementing and testing two of the operations on the standard IEEE format on 32 bits (multiplication and division). The difference from the usual implementations of multiplication and division is that this project will perform them on floating point numbers with single precision. This unit will take care of the proper 32-bit numbers as well as of the special cases which were integrated in the FPU.

5. References

- [1] A.P. Shanthi, “*Floating Point Arithmetic Unit*”,
<http://www.cs.umd.edu/~meesh/411/CA-online/chapter/floating-point-arithmetic-unit/index.html>
- [2] “*Floating Point Multiplier*”,
https://lslwww.epfl.ch/pages/teaching/cours_lsl/sl_info/FPMultiplier.pdf
- [3] “*Floating Points*”, <https://www.rfwireless-world.com/Tutorials/floating-point-tutorial.html>
- [4] Naresh Grover, M.K. Soni, “*Floating Point Unit*”,
<https://core.ac.uk/download/pdf/190942867.pdf>
- [5] “*Single-precision Floating-Point format*”, https://en.wikipedia.org/wiki/Single-precision_floating-point_format
- [6] Jean-Michel Muller, Nicolas Brisebarre, “*Handbook of Floating-Point Arithmetic*”,
https://doc.lagout.org/science/0_Computer%20Science/3_Theory/Handbook%20of%20Floating%20Point%20Arithmetic.pdf
- [7] “*Floating-Point Tutorial*”, <https://www.rfwireless-world.com/Tutorials/floating-point-tutorial.html>
- [8] “*Floating-Point Unit – an overview*”,
<https://www.sciencedirect.com/topics/computer-science/floating-point-unit>
- [9] “*What is a floating point overflow?*”,
https://water.usgs.gov/nrp/gwsoftware/ModelMuse/Help/index.html?what_is_a_floating_point_overflow.htm
- [10] “*Numerical Computation Guide – Underflow Thresholds*”,
https://docs.oracle.com/cd/E37069_01/html/E39019/z4000ac019659.html
- [11] “*IEEE Arithmetic*”, https://docs.oracle.com/cd/E19957-01/806-3568/ncg_math.html#415