# CEG3156-Computer System Design

12/2/2024

# Lab#1 :

## Floating-Point Multiplication

**Group membres :**

| | |
|---|---|
| Gbegbe Decaho - Jacques | 300094197 |
| Ismaël Pedro - | 300242291 |
| Abdoulaye Diallo - | 7935327 |

**Assistant TA –** Pulkit Arora

Abdoulaye Djela Diallo
DEPARTEMENT COMP. ENG UNIVERSITY OF OTTAWA

## Table of Contents

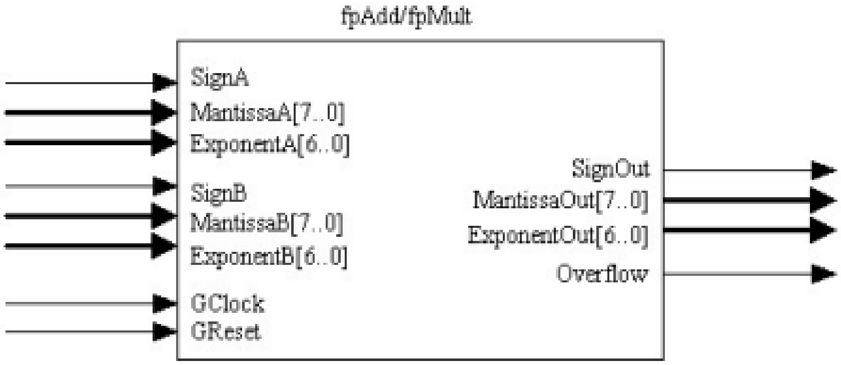
# I Objectif

The objective of this laboratory is to design and build a floating-point multiplier using VHDL coding implementation.  be able to Design, realize and test a floating-point adder unit  Design, realize and test a floating-point multiplication unit; demonstrate a complete understanding for floating-point arithmetic.

# II Theoretical Part

## 1 Introduction of the Problem

In this laboratory, the main goal is to design a multiplier multiplier and an additionner where each components uses a different logic of implementation but both possess the same input and output components as shown in the following figure:



## 2 Problem identification and discussion

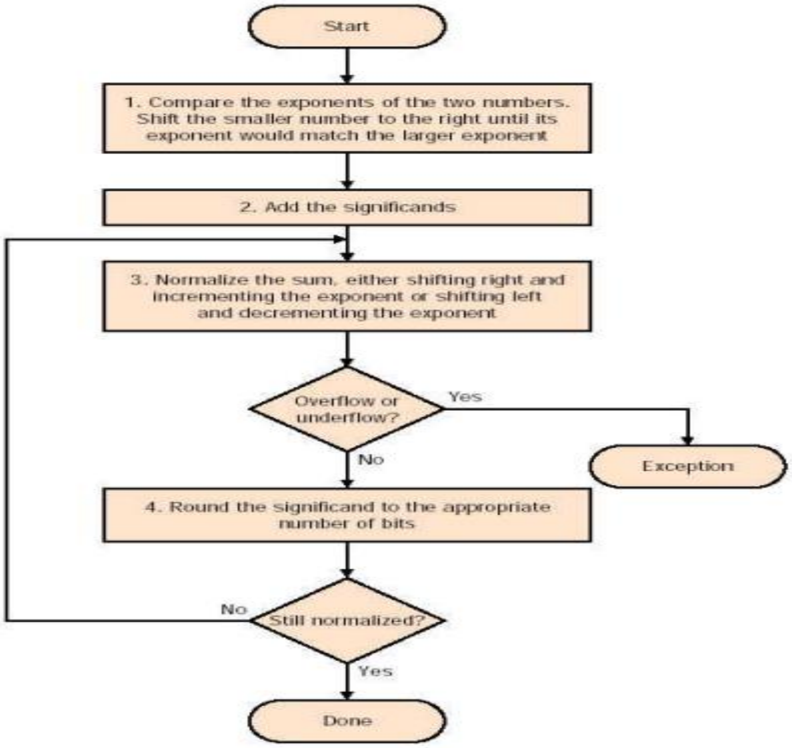
The two entities that we had to implement during this laboratory are :

- The addition:

The algorithm consists of choosing the number that has the smallest exponent and shift its mantissa to the right with the difference of exponents then define the result of the exponent as being equal to the largest exponent. Also we have to perform an addition on the mantissa to determine the sign of the result and then normalize our results if necessary then check if there is an overflow or underflow if this is the case, then it is an exception and the process ends otherwise we round the mantissa and recheck if it is always normalize after these steps if this is the case, the algorithm ends otherwise we have to redo the process from the step of standardization

As shown in the following ASM diagram:

**Addition**

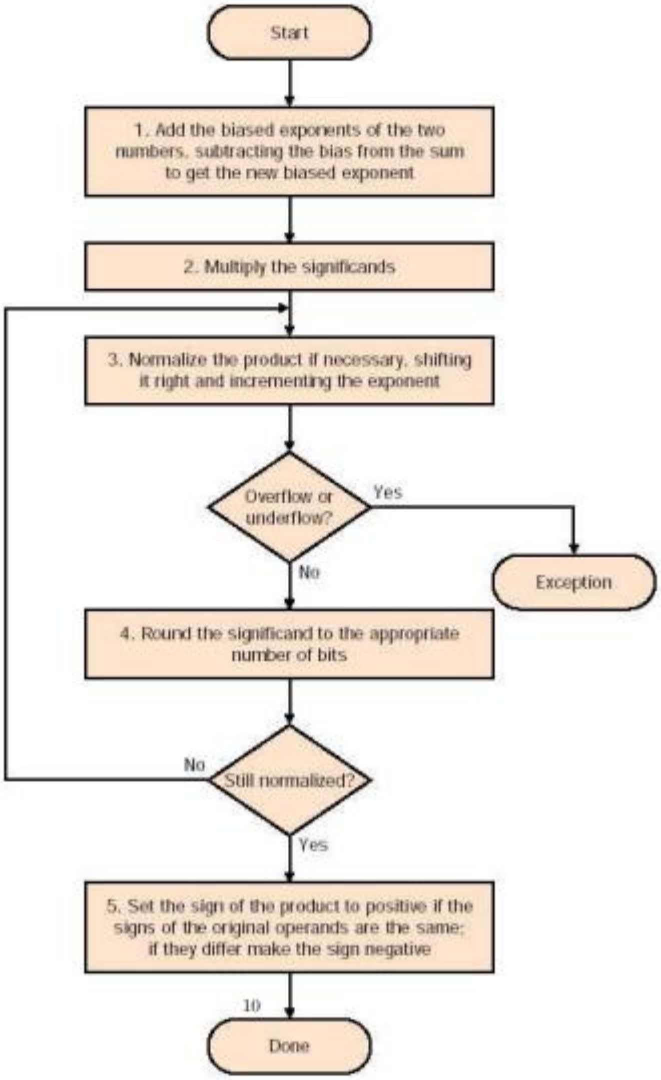

Start
1. Compare the exponents of the two numbers. Shift the smaller number to the right until its exponent would match the larger exponent
2. Add the significands
3. Normalize the sum, either shifting right and incrementing the exponent or shifting left and decrementing the exponent
Overflow or underflow?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

## Multiplication

We added the two exponents then subtracted 127 then we multiplied the two masteries and determined the sign resulting and finally we had normalize the resulting values if this proves necessary then we check if there is an overflow or underflow if this is the case then it is an exception and we finish our process otherwise we round the mantissa and we recheck if it is still normalized after these steps if it is in the case we put the sign of positive products if they have the same signs and negative if they are of different signs and then comes the end of our algorithm otherwise (if it is not normalized) we redo process from normalization stage

As shown in the following diagram:

ASM Diagram chart of Multiplication


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
No
Exception
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
10
Done

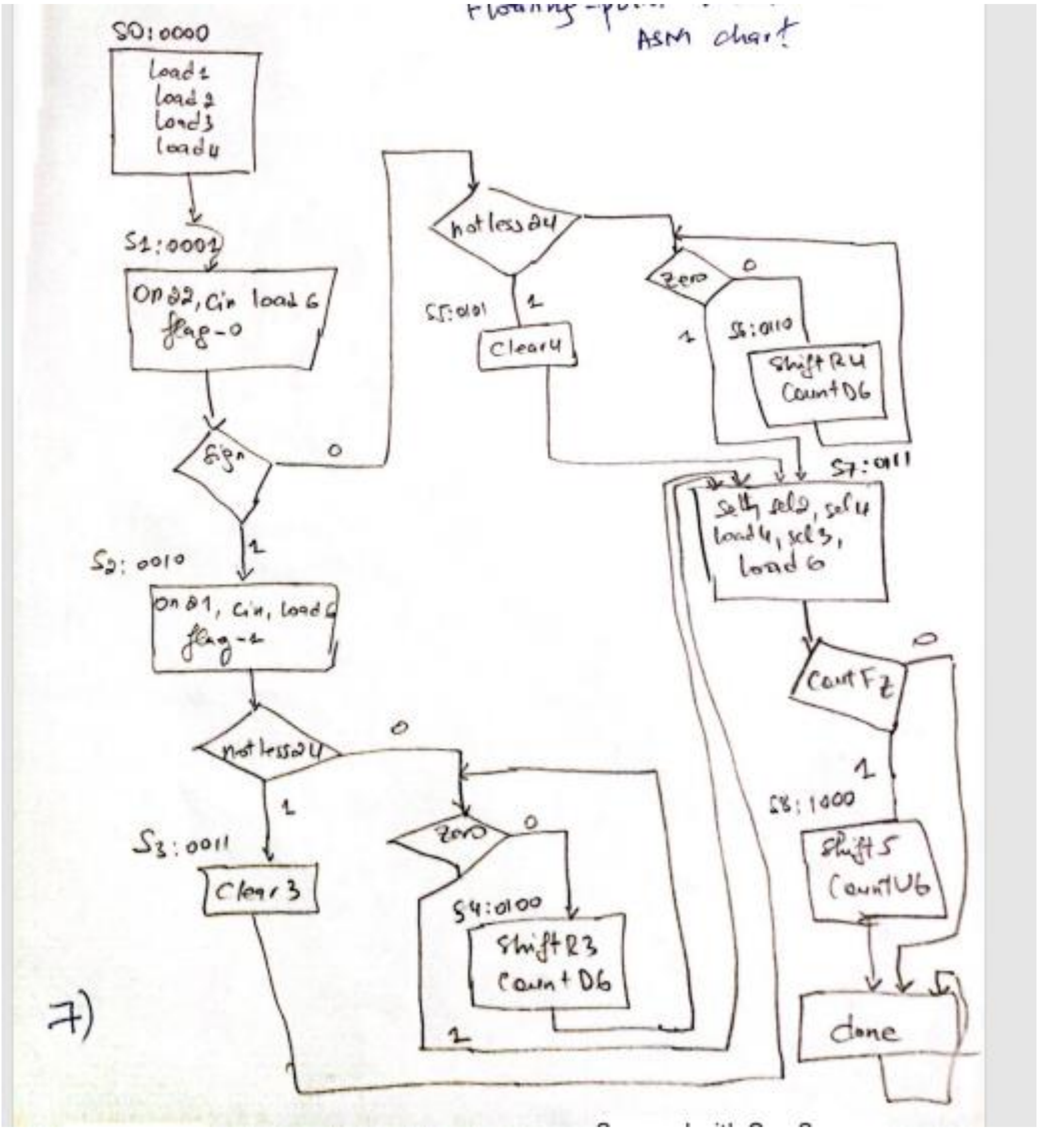
## 3 Algorithmic procedures and discusion

Regarding our implementation for both the addition and the multiplication units, we used the method of the 5 stages which are mainly the characteristics of the algorithm to use (which was shown previously) then it was a necessity to rely on this algorithm to make our ASM when this was carried out we tried to find the registers that were associated with the logic of our ASM to build our Datapath then input and output of our Datapath we were able to do the detailed ASM and finally we were able to get out our control path the realization of these 2 units differs due to their logic.

## Addition

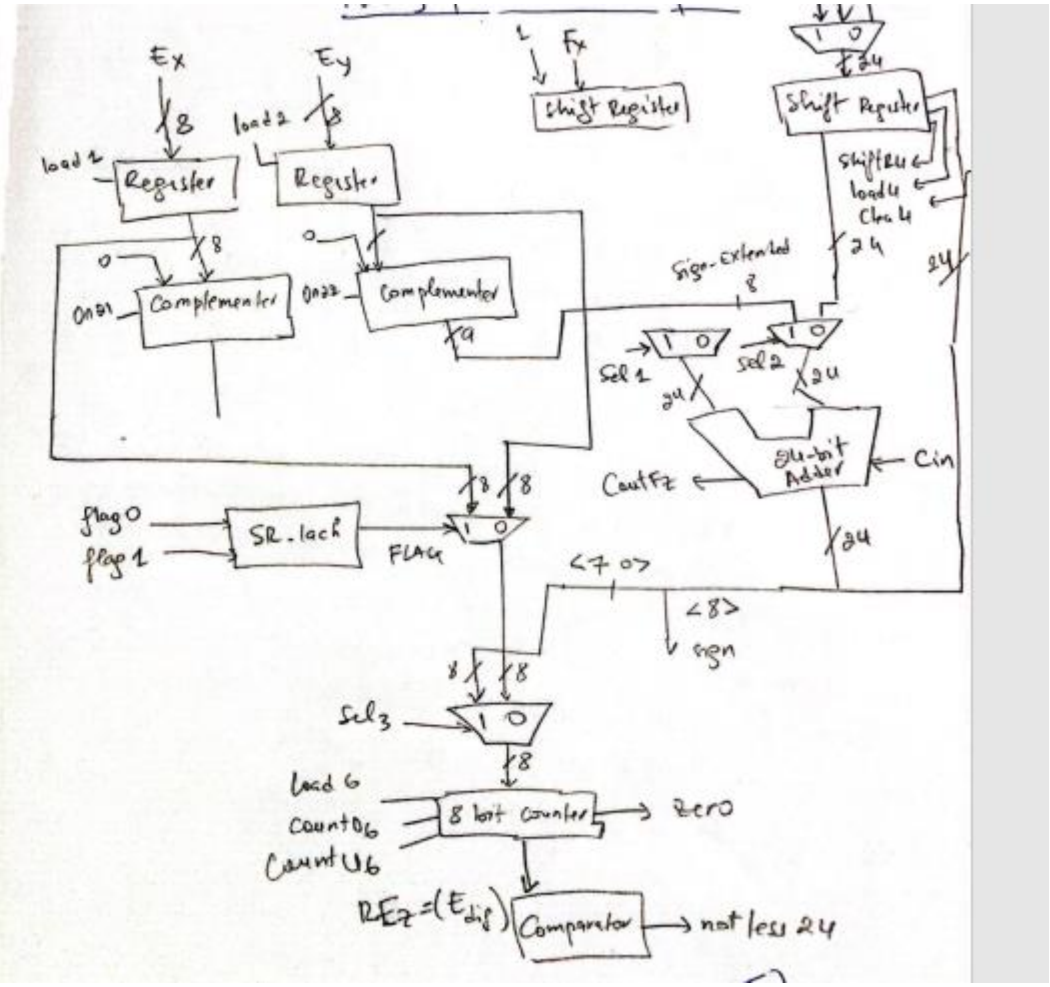
## ASM Chart

The ASM diagram follows the logic of our algorithms seen above, at the beginning we load the values of the exponents and mantissas then we make the difference of the two exponents to compare them if it is positive we continue our algorithms if it is negative we reverse the positions of the two exponents in the difference and we continues our algorithm as follows: If it is positive we recheck if the difference in exponents is less than 9 and we do this with the comparator because its output allows us to be informed if it is less than or greater than 9, if our result is greater than or equal then we make a clear for the second mantissa and we move on to the next step otherwise we check if the difference of the exponents is equal to zero through the 7-bit counter output if it is true we move on to the next step otherwise we decrements the difference in exponents and we shift the second mantissa on the left and we do this again operation until the difference in exponent is equal to 0 so that we move on to step next . If it is negative, we recheck if the difference in exponents is less than 9 and we do this with the comparator because its output allows us to be informed if it is lower or higher, if our result is greater than or equal to 9 then we make a clear for the first mantissa and we move on to the next step otherwise we check if the difference of the exponents is equal to zero through the 7-bit counter output if it is true we move on to the next step otherwise we decrements the difference in exponents and we shift the first mantissa on the left and we do this again operation until the difference in exponents is equal to 0 so that we move to the next step . After this we will add the value of the two mantissas and the value of the exponent will depend on the path we followed during the first condition (flag 0 or flag 1) then, we move on to the last step, that of standardization. We check that Rfz is indeed in the good form, i.e. less than 0. If this is not the case, we increment the exponent by one and shift Rfz to the right. We repeats until the Rfz<1, in which case we have finished.



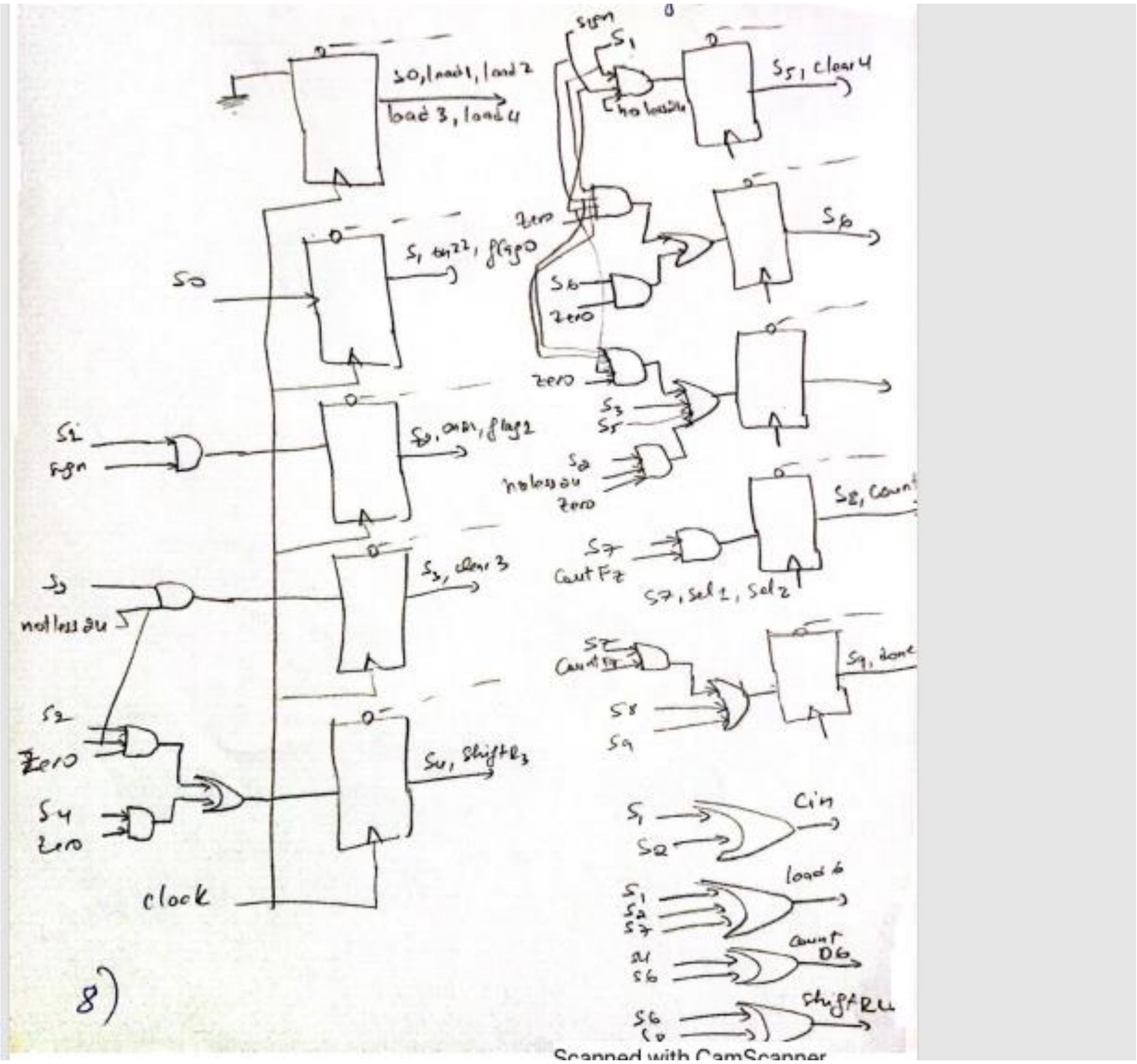
## Datapath

The datapath is made up of 2 7-bit registers which we allows us to load our two exponents and which are connected respectively to 2 8-bit complementors which are like entering the exponent and the 0 that it will add and also (on21,on22). The latter two are connected to a 8-bit adder since our values were in complemented before 8 bits which allows us to make the difference between the exponents and has as input the carry in and as output the sign of the operation the latter is itself connected to a meter which allows us to store the value sent by the 8-bit adder then and a count entry to decrement the value of 1 and a zero output to signal the end of the operation and finally the counter is connected to a comparator 8 bits to compare the value obtained (the difference of exhibitors), we also connected our register which contains the value of our exponent to a multiplexer which will choose one of the exponents depending on the flag and the difference state then this multiplexer is connected to a 7-bit composter which allows the value to be stored chosen (with the load) and the output is the result of the exhibitors. Concerning the mantissas, we simply have use 2 9-bit shift registers to load the values mantissas and also shifters if necessary we also have added the clear entry which allows us to set our mantissa in the case where the difference of the exponents is greater than 9 then these two registers are connected to a 9-bit adder which has as output the results of these two mantissa and this adder has a named output coutfz which allows us to know if the result always need to be standardized the latter itself connected to a 9-bit shift register to store the result obtained by the adder with (load) and the shifter either to the left or to right depending on the situation, we also find a clear to clear the value obtained in need and its output is the final answers for the mantissas.



## Controlpath

From the detailed asm we can derive the following control path:

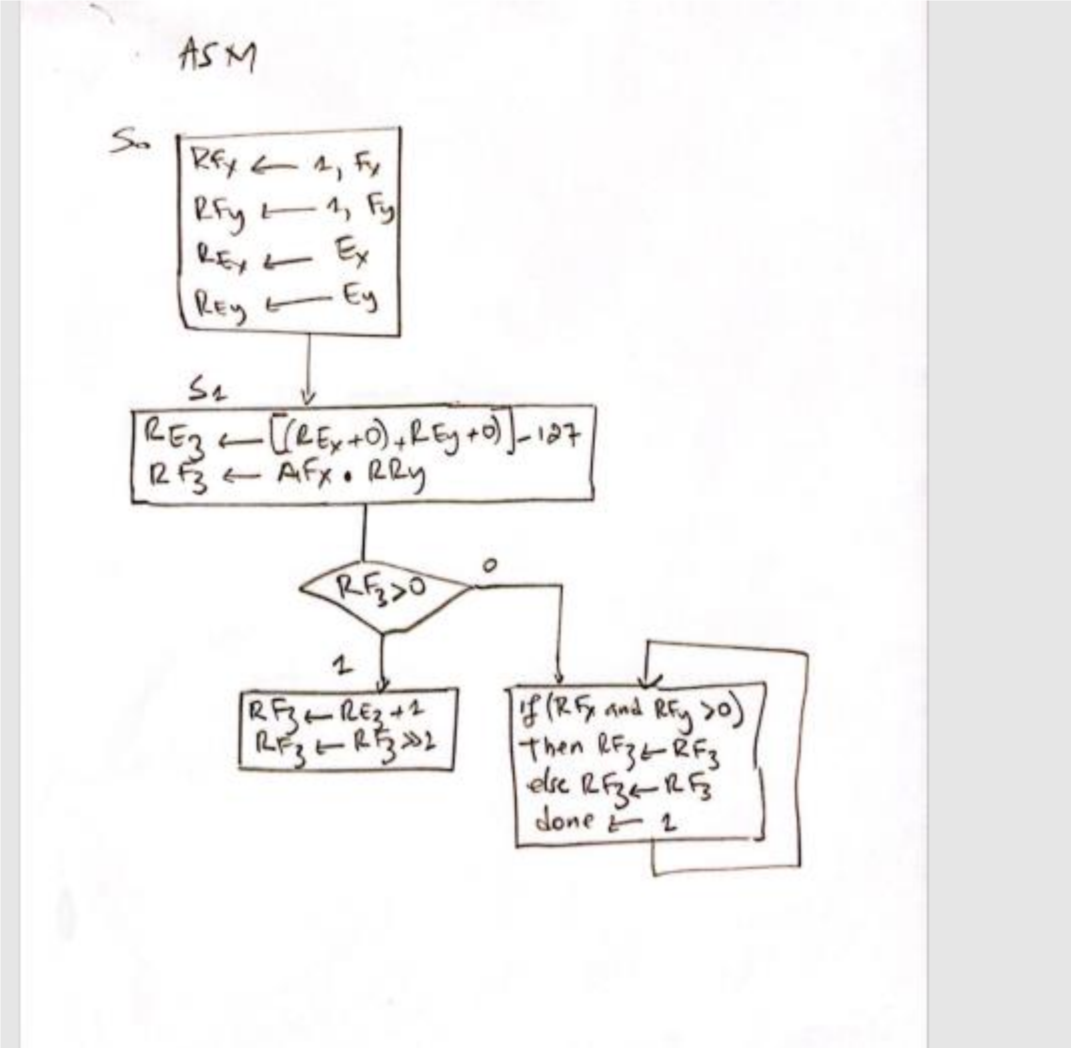


## Multiplication

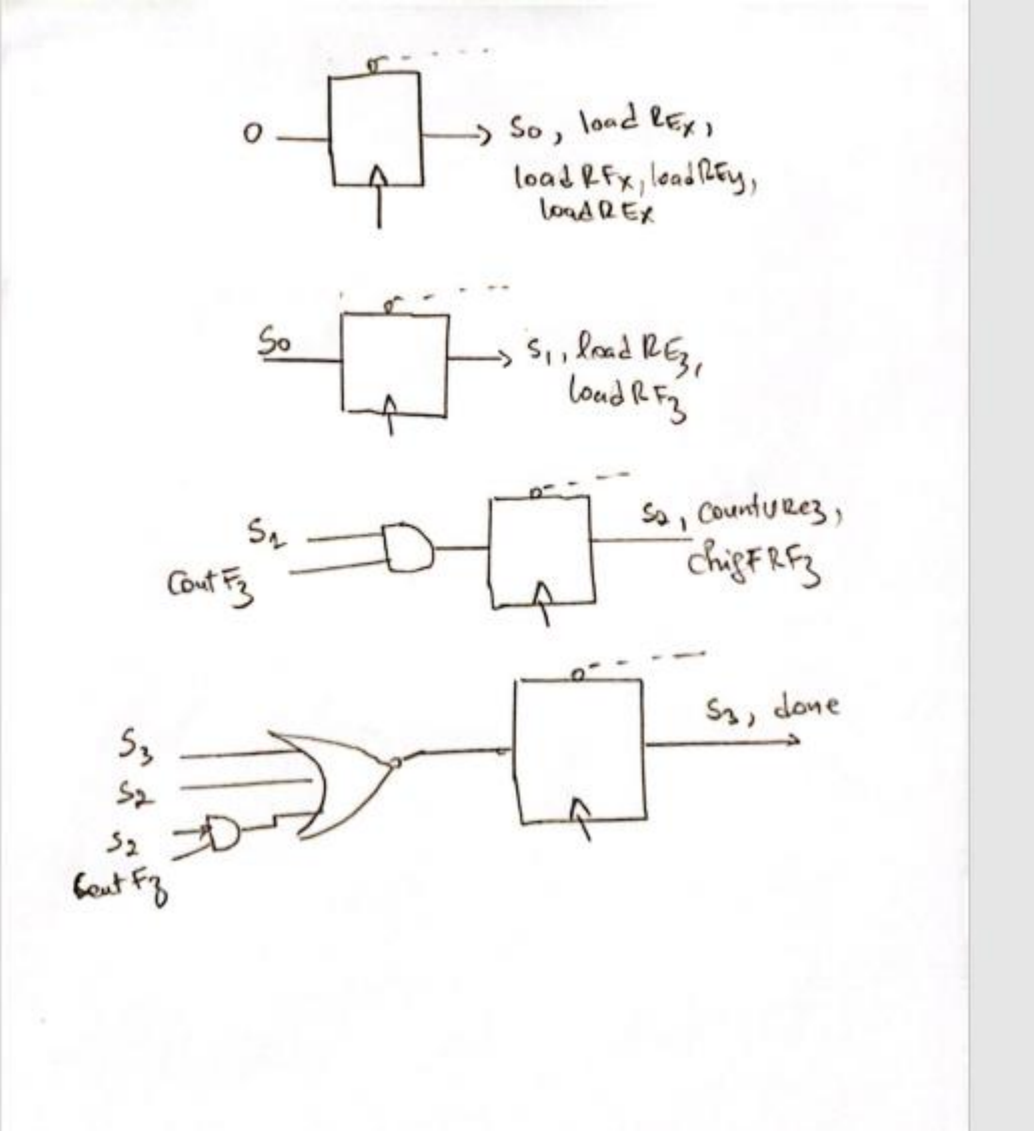
### ASM

The  ASM diagram  chart follows the logic of our algorithms shown in the following diagram, we load at start the values of the exponents (...) and the mantissas (...)

then we added the two exponents and subtract 127 then we multiply the two mantissas after having done this we did  a check to see if we need to make a normalization if this is the case, we do a right shift to the exponent and then increment. Otherwise, we ensure for the sign of the mantissas if the last bits contain the same sign, then they are positive otherwise they are negative.

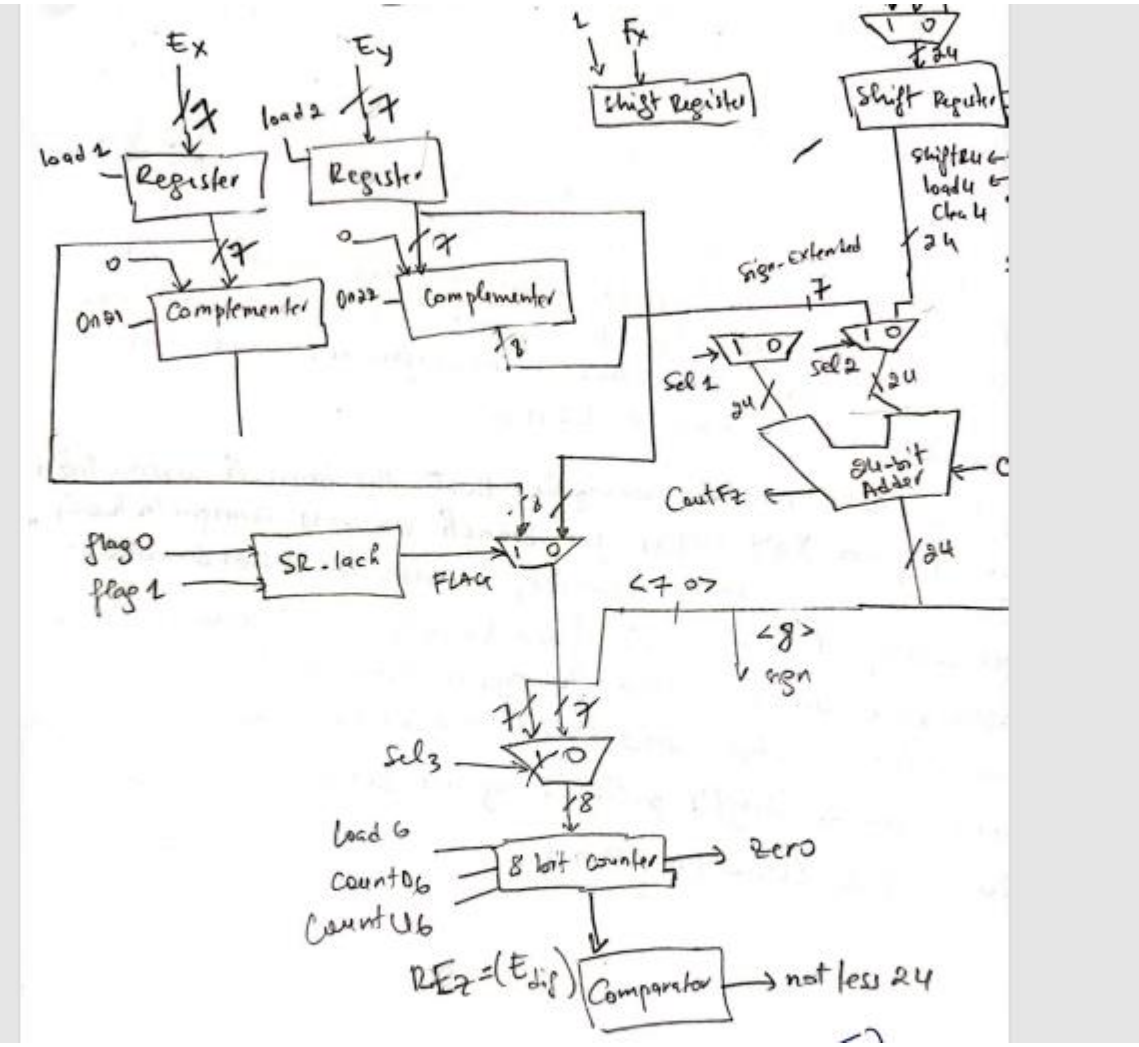


## Controlpath



## Datapath

The Datapath consists of 2 registers of  7-bits to enter the values of the two exponents then these two registers are connected respectively with 2 complementor to 8 bits which as input 0 the latter are connected with an 8-bit adder and those for the purpose of adding the two exhibiting as we explained in our algorithm it is itself connected to a complementor 9 bits which like between the result of the addition and a 0 this last one is connected to a 9-bit subtractor and those for subtract 127 from the results of the two added exponents and then we put our results in a 7-bit counter which has as output the resulting exponent. While for the mantissa we find 2 9-bit registers which have as input a mantissa and 1 and those for store the values of the 2 different mantissas then these last two are connected with a 9-bit multiplier whose output is coutfz which allows us to know if the result always needs to be normalized the latter even connected to a 9-bit shift register to store the result obtained by the adder with (load) and the shifter on the right, its output is the final answers for the mantissas. Concerning the part of the sign we just put an “XOR” between the two signs entered at the beginning.



# III Design Part

## 1 Discussion of Used Components

### 1.1 Additionner

Control path:

They essentially consist of D flip flops to manage the different states as well as control signals. logic gates “and” and “or” to manage the conditions of transitions between States and obtain the different control signals.

### 1.2 Multiplication

Datapath:

Regarding the Datapath we will need 2 registers of 7-bits to load the value of the exhibitor. An 8-bit adder to add the two exhibitors A 9-bit subtractor to subtract 127 from the addition made previously of the exponents to make this we used a 9 bit adder and the 1's complement of 127. A 7-bit counter which is mainly connected to a subtractor the load of the register is connected to the input signal and the count is used to decrement the register value of 1’s Two complementary 8 bits which takes as input and exponent value which is 7 bits its goal is to make it 8 bits. 1 complementary of 9 bits which has as input 0 and the result of the 8-bit addition, its goal is to make the output of 8-bit addition to 9-bit 2 registers of 9-bits  to load the values of both mantissas. A 9-bit Multiplier that multiplies both mantissas And bits shift-register to shift the result which must be done in the standardization stage and has as output the results of the mantissas

Control path:

They essentially consist of D flip flops to manage the different states as well as control signals. logic gates “and” and “or” to manage the conditions of transitions between States and obtain the different control signals.

## 2 Discussion of  encountered problems

We encountered several problems in this laboratory by example we found it difficult to test our designs especially when Quartus gave us errors and we arrived at advanced stages in our code we were obliged to return to view entity by entity and make additional outputs to try to debug our code.

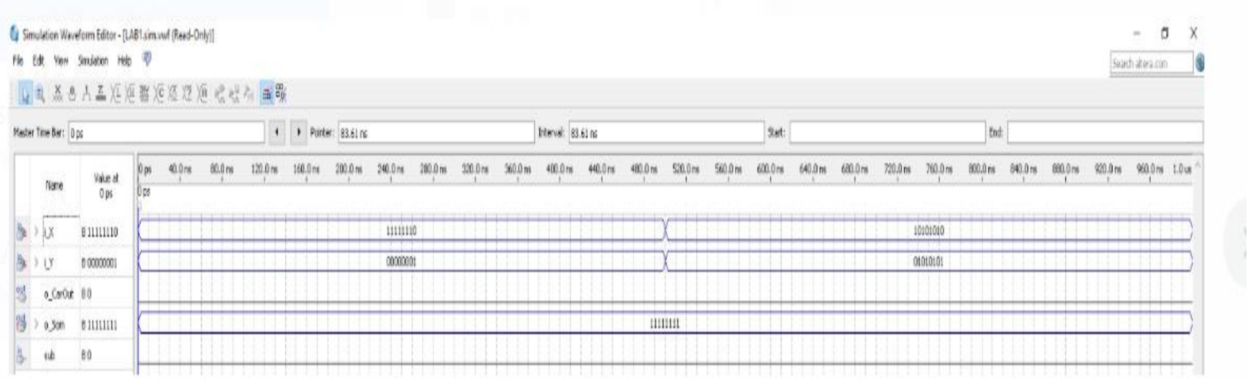
## 3 Actual Solution Discussion

In the submission for this lab, you will find attached in a compressed zip document all the files and components used in the lab experience. Also the lab report for this lab.

# IV Real Implementation and Simulation

## 1 Adition and multiplication simulation

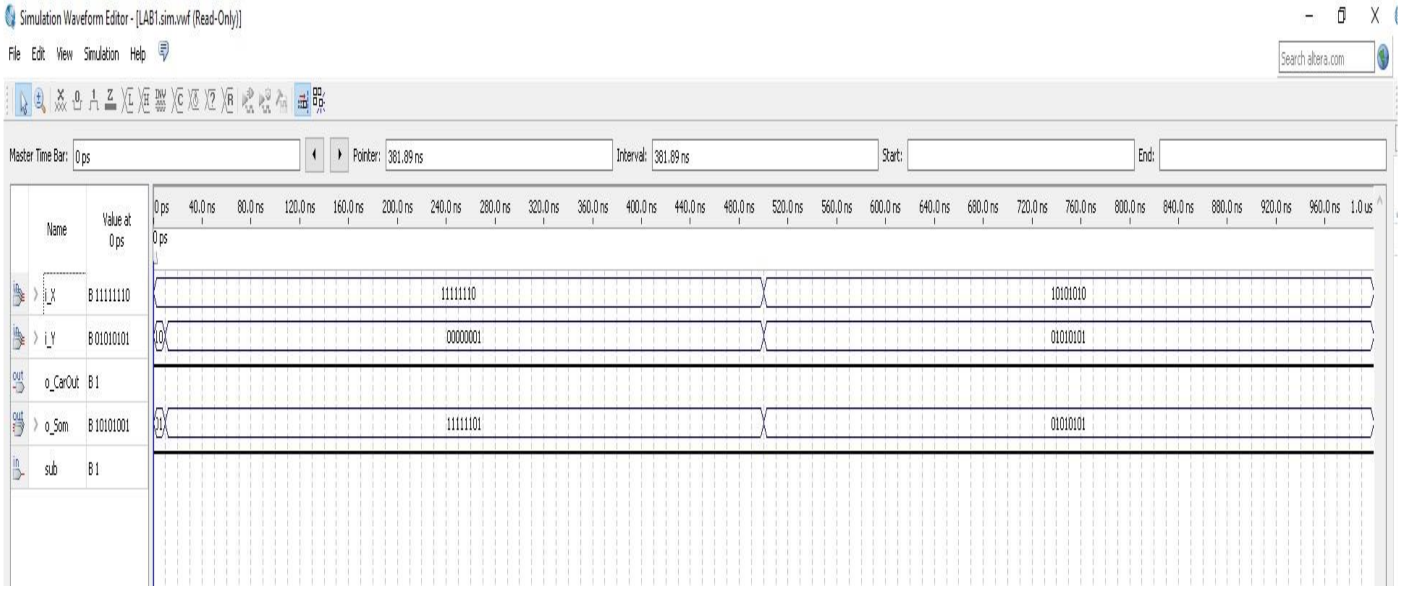
### 1.1 First addition simulation


Simulation Waveform Editor - [LAB1.sim.vwf (Read-Only)]
File Edit View Simulation Help
Search altera.com
0 ps 40.0 ns 80.0 ns 120.0 ns 160.0 ns 200.0 ns 240.0 ns 280.0 ns 320.0 ns 360.0 ns 400.0 ns 440.0 ns 480.0 ns 520.0 ns 560.0 ns 600.0 ns 640.0 ns 680.0 ns 720.0 ns 760.0 ns 800.0 ns 840.0 ns 880.0 ns 920.0 ns 960.0 ns 1.0 us
0 ps
11111110
10101010
00000001
01010101
11111111

Master Time Bar: 0 ps
Pointer: 83.61 ns
Interval: 83.61 ns
Start:
End:

| Name | Value at 0 ps |
|---|---|
| i_X | B 11111110 |
| i_Y | B 00000001 |
| o_CarOut | B 0 |
| o_Som | B 11111111 |
| sub | B 0 |

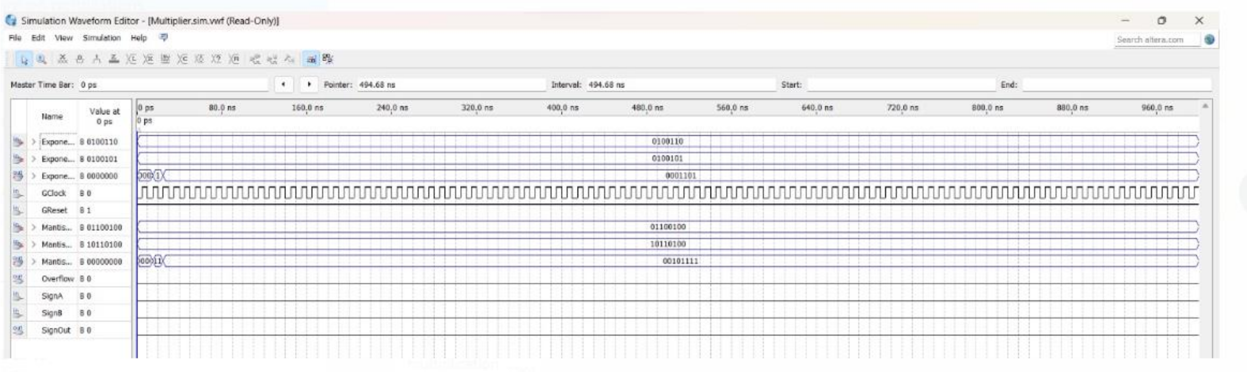
## 1.2 Second Addition simulation


Simulation Waveform Editor - [LAB1.sim.vwf (Read-Only)]
File Edit View Simulation Help
Search altera.com

Master Time Bar: 0 ps Pointer: 381.89 ns Interval: 381.89 ns Start: End:

| Name | Value at 0 ps | Waveform |
|---|---|---|
| i_X | B 11111110 | 11111110 → 10101010 |
| i_Y | B 01010101 | 10 → 00000001 → 01010101 |
| o_CarOut | B 1 | |
| o_Som | B 10101001 | 01 → 11111101 → 01010101 |
| sub | B 1 | |

## 1.3 Multiplication simulation


Simulation Waveform Editor - [Multiplier.sim.vwf (Read-Only)]
File Edit View Simulation Help
Search altera.com

Master Time Bar: 0 ps Pointer: 494.68 ns Interval: 494.68 ns Start: End:

| Name | Value at 0 ps | Waveform |
|---|---|---|
| Expone... | B 0100110 | 0100110 |
| Expone... | B 0100101 | 0100101 |
| Expone... | B 0000000 | 000 → 1 → 0001101 |
| GClock | B 0 | |
| GReset | B 1 | |
| Mantis... | B 01100100 | 01100100 |
| Mantis... | B 10110100 | 10110100 |
| Mantis... | B 00000000 | 000 → 11 → 00101111 |
| Overflow | B 0 | |
| SignA | B 0 | |
| SignB | B 0 | |
| SignOut | B 0 | |

We can notice that our simulation gives results consistent concerning the signs of the two values because we put the value of 0 in both expressions and we obtain 0 which is logic concerning the exponents we notice that the result (exponent result) is the addition of the two exponent ( exponent and expose Y) and we subtract 127 from this result for the result of the mantissa we notice that the result is correct because the output (Mantissa_Result) is just the multiplication of the two input mantissa (MantissaX and MantissaY)

### 2 Design verification

# V Discussion

Each component of our arithmetic units (addition and multiplication) has been simulated and tested. Although we had some left for demonstration at the map level, the addition and multiplication units were operational and producing simulation results. On the other hand, there was no time left for us to continue with the demonstration.

# VIConclusion

Finally, we can conclude that this laboratory experience is important because it allowed us to get familiarized with designing and implementing arithmetic floating points units practically on altera board. We learned and understood the concept of VHDL structural programming and gained knowledge of hardware system design through implementation of hardware programs.

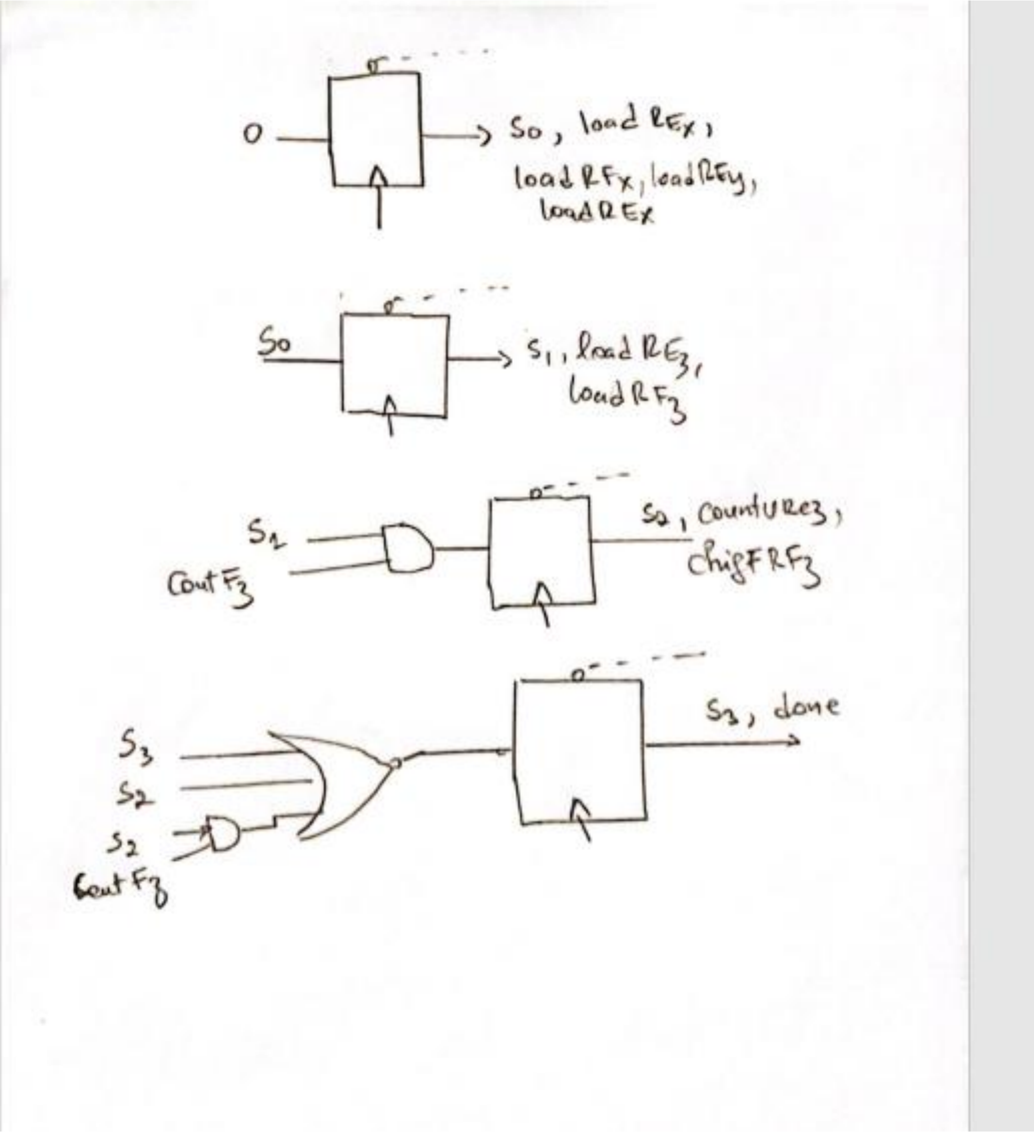
However, we can justify by the objective of the laboratory that we have learned how to design, realize and test a floating-point addition unit and also design, realize and test a floating-point multiplication unit. We were able to demonstrate a complete understanding for floating-point arithmetic through a Quartus simulations and compilations.

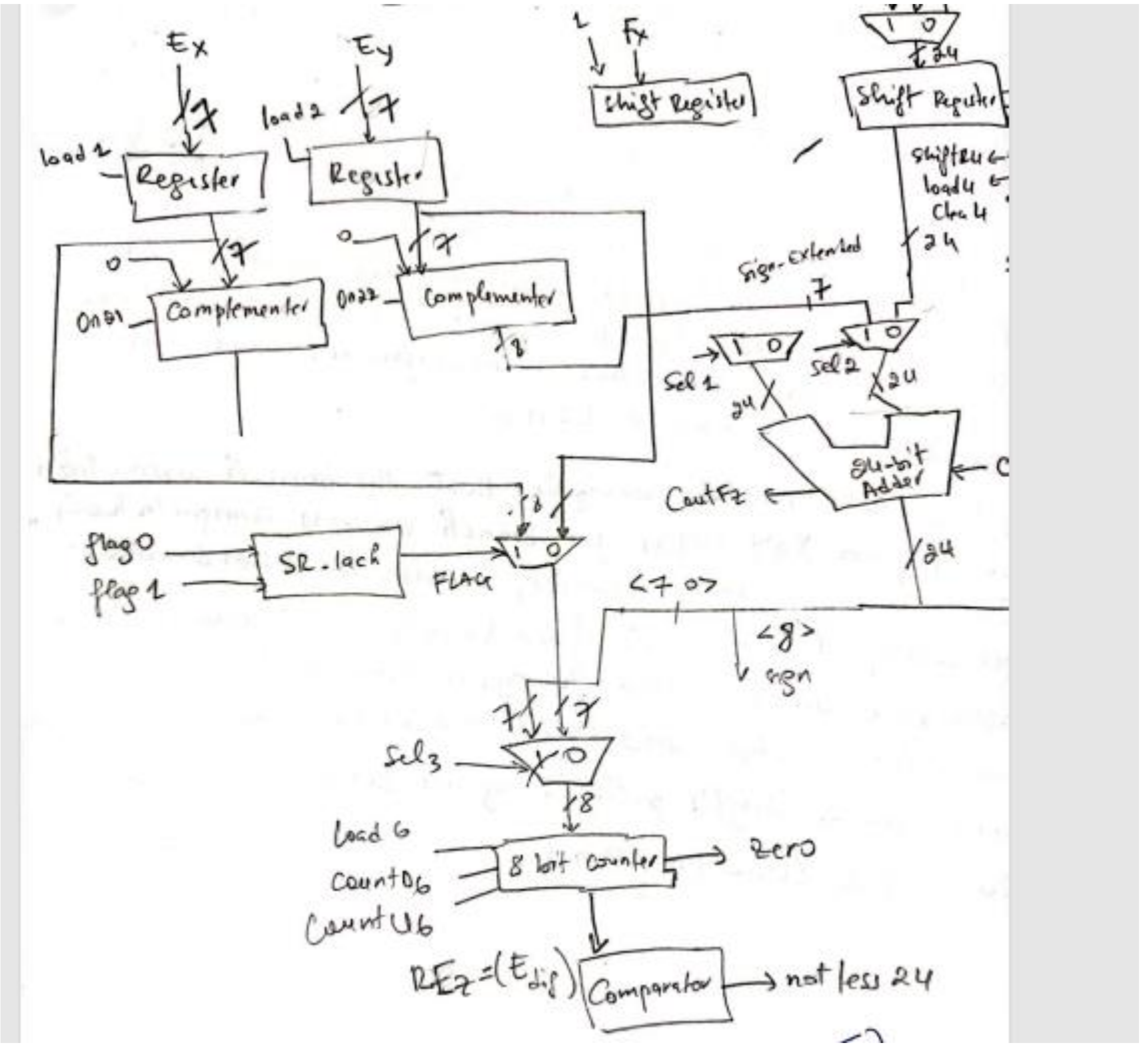
# VII Reference

The courses manual and code provided for the laboratory.

# VIII Annexe

Captures of some designs and diagrams used during the lab session


0
So, load REx,
load RFx, load REy,
load REx
So
S1, load REz,
load RFz
S1
Cout Fz
S2, CountUREz,
ChifFRFz
S3
S2
S2
Cout Fz
S3, done


Ex
Ey
Fx
1
load 1
load 2
7
7
Register
Register
Shift Register
Shift Register
1 0
24
shiftRu
load4
Clr4
24
0
0
On01
On02
Complementer
Complementer
7
7
8
Sign-extended
7
1 0
1 0
Sel 1
Sel2
24
24
24-bit Adder
C
CoutFz
8
flag0
flag1
SR-lach
FLAG
1 0
24
<7 0>
<8>
sign
7
7
Sel3
1 0
8
load 6
CountD6
CountU6
8 bit counter
zero
REz =(E_dif)
Comparator
not less 24