# FLOATING POINT UNIT (FPU)

Addition & Subtraction

ABSTRACT

This paper will describe all the steps required to design an FPU which performs addition and subtraction of Floating Point .

Anamaria Raita

Structure of Computer Systems

Table of Contents



# Introduction

## Context

The purpose of this project is to implement an FP Unit which has an architecture and a functionality a little bit more complex than the basic ALU that every engineer student knows to implement. This FPU will execute the basic operations: Addition and Subtraction on Floating Point Numbers with single precision. These numbers will respect the standard IEEE 32 bits format. Also, I want to make a project that can be used by everyone, regardless the fact that they own a board or not, so it will also contain a simulation of the project.

Even if the project can be used only to perform basic calculations, I do not think that this should or will be its main purpose, but it would be more probably to integrate it in the microprocessor of some other device. Also, it could be a starting point in implementing a full FPU.

## Specification

As we will see more precisely in the next chapter, these numbers we have talked about use the Standard IEEE format, so the application should be able to represent them internally in this way, to perform addition and subtraction on them, to store the operands and the result and also to display each one of them in a readable format, such as everyone can read it easily without having to make conversions.

The project will be simulated using the tools Vivado provides.

### Objectives

Design a way in which introduced numbers in binary form and store them internally in the floating point representation and to display the result of the operation. Also, design a user-friendly control-unit which gives the user the possibility to choose the desired operation: introduce operands, store operands, display operands, addition, subtraction, store result and display result. All the 'display' options should be done using a 7-segment display. A simulation for validating the functional requirements is as well needed.

## Bibliographic Study

### Number Representation

The first and most important thing before starting to perform the algorithms is to see and understand how to store the numbers used by us in everyday life in the Single Precision IEEE 754 Floating-Point Standard.

The numbers are represented using 32 bits and each number can be split in 3 parts:

- The first bit represents the sign.
- Next eight bits represent the exponent
- Last 23 bits representing the mantissa

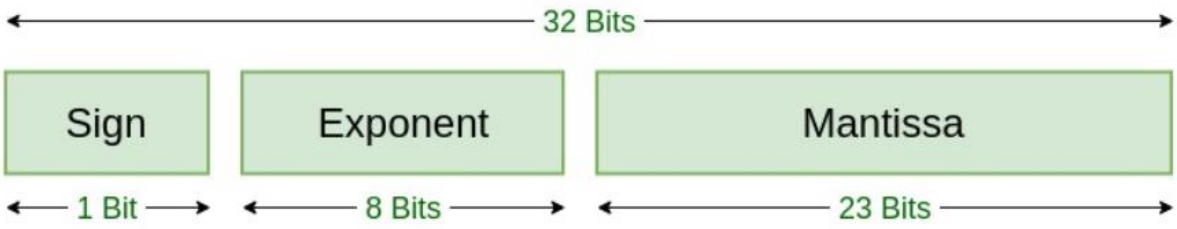


*Figure 1 – Representation of numbers in IEE 754*

The exponent is an 8-bit unsigned integer from 0 to 255, in biased form: an exponent value of 127 represents the actual zero, that is why in the formula below we see that from the exponent the value 127 is subtracted.

The mantissa represents the actual binary digits of the floating-point number.

$$\text{value} = (-1)^{\text{sign}} \times 2^{(E-127)} \times \left(1 + \sum_{i=1}^{23} b_{23-i} 2^{-i}\right)$$

*Figure 2– Formula for the value of numbers*

For the purpose of clarity, we will convert the number 85.125 in the form described above to see all the steps needed to be performed also by the FPU:

1. We separate in two parts: 85 and 0.125 and convert each of them to their binary form.
2. So now, we have 1010101 for 85 and .001 for 0.125.
3. Now we combine the two parts again an obtain 1010101.001 .
4. As the mantissa always starts with one, we move the point six places in the left and obtain 1.010101001
5. The number of places we have moved the point to the left will be the exponent, so the representation of the initial number up to this point is 1.010101001* 2^6
6. Establish the bit for the sign, which will be 0 for our example
7. The true exponent used in the formula above will be obtain by adding 6 and 127, thus we will get 133.
8. Transform the exponent into its binary representation 10000101.
9. Now it is time to combine the parts:

   First bit for the sign will be 0.

   The next 8 are the ones obtained at step 8: 10000101

   And for the mantissa, we go back to the number at step 4: 1.010101001 drop the first '1'

   and then pad with 0 on the remaining bits of the mantissa.

The result will be:

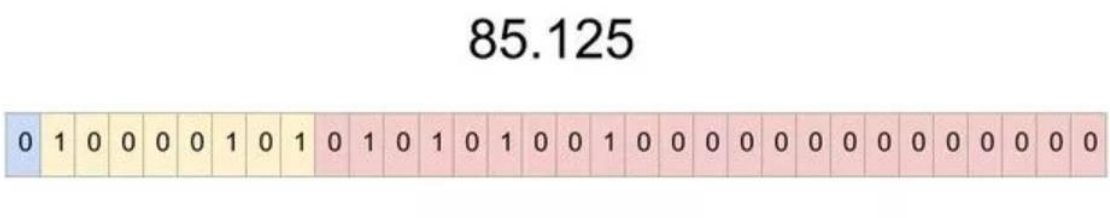


*Figure 3 – Representation of a number*

Special Cases of representation:

-∞: 1 11111111 00000000000000000000000

+∞ : 0 11111111 00000000000000000000000

NotANumber (Nan values) : sign = either 0 or 1.

biased exponent = all 1 bits.

fraction = anything except all 0 bits (since all 0 bits represents infinity)

## Addition Algorithm

For the purpose of making the explanation easier, we will denote the 2 operands with X1, X2 and the result with S.

Firstly: X1 and X2 can only be added if the exponents are the same i.e E1=E2

If this is not the case, we will see later in the algorithm that we will need some shifts.

1. We assume that X1 has the larger absolute value of the 2 numbers. Absolute value of of X1 should be greater than absolute value of X2, else swap the values such that Abs(X1) is greater than Abs(X2).
   Abs(X1) > Abs(X2)
2. Initial value of the exponent (before the shift) should be the larger of the 2 numbers, since we know exponent of X1 will be bigger , hence Initial exponent result E3 = E1.
3. Calculate the exponent's difference i.e. Exp_diff = (E1-E2).
4. Left shift the decimal point of mantissa (M2) by the exponent difference. Now the exponents of both **X1 and X2 are same.**
5. Compute the sum/difference of the mantissas depending on the sign bit S1 and S2.
   If signs of X1 and X2 are equal (S1 == S2) then add the mantissas
   If signs of X1 and X2 are not equal (S1 != S2) then subtract the mantissas
6. Normalize the resultant mantissa (M3) if needed. (1.m3 format) and the initial exponent result E3=E1 needs to be adjusted according to the normalization of mantissa.
7. If any of the operands is infinity or if (E3>Emax) , overflow has occurred ,the output should be set to infinity. If(E3 < Emin) then it's a underflow and the output should be set to zero.
8. Null values are not supported.

We will take an example for more clarity:

A = 9.75
B = 0.5625

X1=0 10000010  0011100…

X2=0 01111110  0010000…

1) Abs (A) > Abs (B)?

Yes.

2) Result of Initial exponent E3 = E1 = 10000010 = 130(10)

3) E1 - E2 = (10000010 - 01111110) => (130-126)=4

4) Shift the mantissa M2 by (E1-E2) so that the exponents are same for both numbers.

MX2=1. 0010000…

=00001. 001000…

=0. 0001001000…

5) Sign bits of both are equal? Yes. Add the mantissa's

1.001110000000…..(MX1)

0.000100100000…..(aligned mantissa of X2)

---

1.010010100000…..(mantissa of X3)

6) Normalization needed? No, (if Normalization was required for M3 then the initial exponent result E3=E1 should be adjusted accordingly)

7) Convert the result back into decimals which is 10.3125

### Subtraction Algorithm

For the subtraction the same algorithm as for the addition will be used, but firstly the sign bit of the second operand will be changed.

### Special Cases

There will be also special cases to be taken in consideration besides the -,+ infinite and NaN values.

Overflow and Underflow situations will be discussed later.

## Analysis

As the two algorithms are almost similar, only a change of sign being needed, the flowchart in Figure 4 shows how the algorithm will work.

Almost the entire diagram will represent one component, the Adder itself, and to perform a subtraction we will only add a Not Gate for the sign of the second number.

Also, there are a few important prerequisites to be mentioned before talking about the operations themselves:

-Firstly, we need to read the operands and check if both of them are valid values (i.e. not infinite or Nan values) and how should we treat them.

-Secondly, we need to see if the accumulator should be loaded with the previous result or with an independent operand.

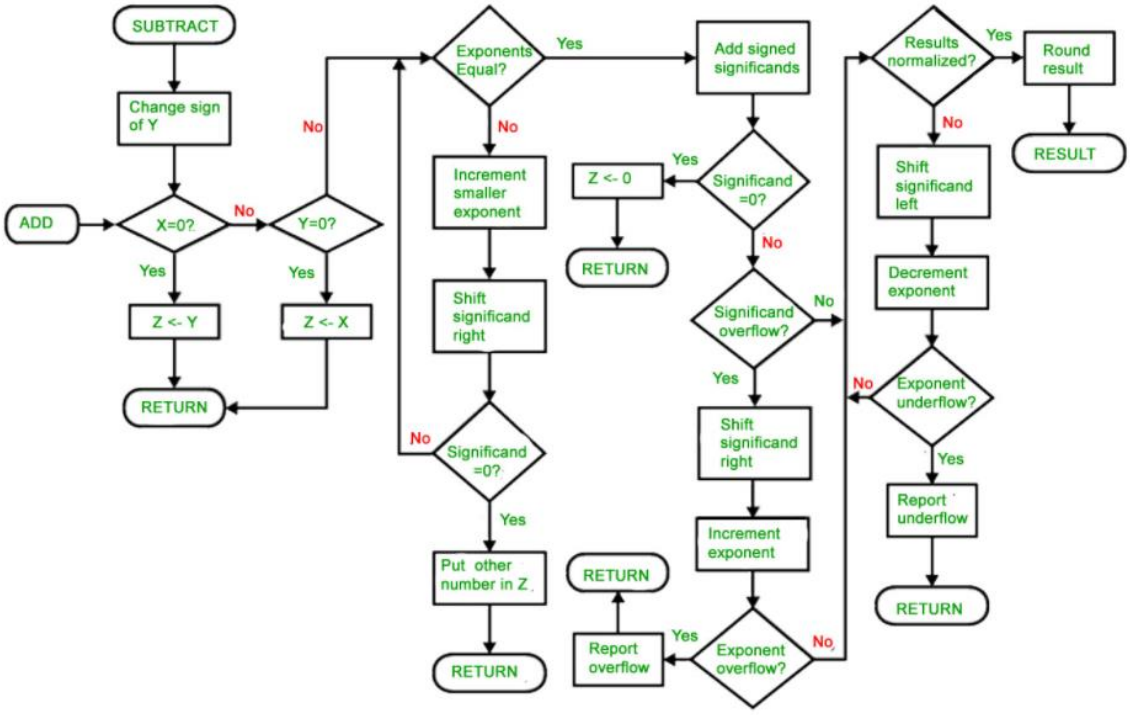


*Figure 4- Floating point addition algorithm flowchart diagram*

I have decided that to have a clearer view, I should compose the adder from three parts:

1.Decomposition part

2.Actual Adder

The decomposition part will be responsible with treating special cases such as:

-one operand being +- infinity or Nan

-one or both operands being 0, that meaning that the result is very easy to compute

Also, there is one more cases:

-one or both operands are normal numbers and not exceptions

We also have 2 special cases:

-exponent overflow (all exponent bits are 1 and mantissa is 0) and underflow

-significand overflow and underflow

## Design

The Higher-Level diagram of the entire design looks like in Figure 5 and below I will write a brief description of each component:

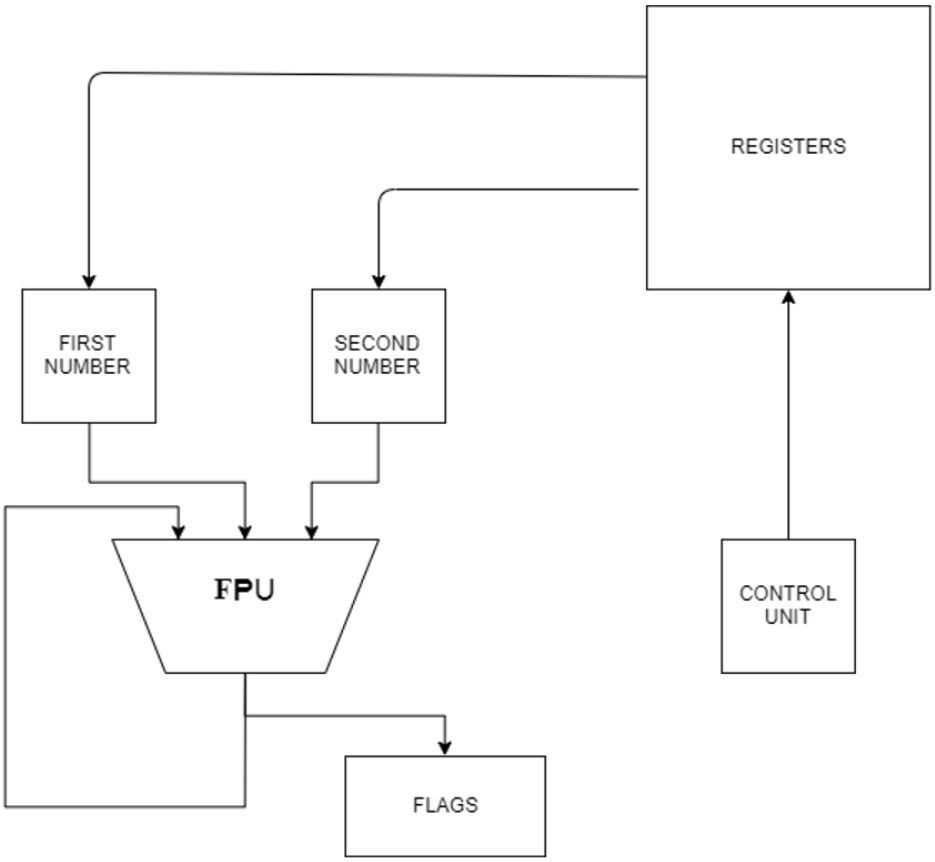


*Figure 5 – System's architecture*

### 1.The Control Unit

-This component will have a role similar to how its name suggests: this is where we choose the operation to be performed: addition or subtraction, also it will help us choose what numbers we

want to perform the operation on. The second part will become cleared as I will describe the second component.

**2. The registers**

As for this component, it will behave like a ROM memory. We need to simulate the FPU and because the numbers are represented on 32 bits, to avoid writing multiple times when testing and writing simulation, this will be used to store values.

**3.The Floating-Point Unit**

This will be the part which implements the algorithm described in the introduction.

This will consist of 2 smaller components:

-The Decomposition and Special Cases (DSC)

This is going to treat the special cases and pass further the 3 parts of the number: sign, exponent and mantissa.

| Exponent | Mantissa | Output | Output Code |
|---|---|---|---|
| 0 | 0 | Zero | 00 |
| 0<=e<255 | >0 | Normal | 01 |
| 255 | =0 | Infinity | 10 |
| 255 | >0 | Nan | 11 |

When both of numbers are normal, we will have an Enable button which allows us to pass further the separate parts of the number.

If not, we will be in one of the cases below, which allows us to display instantly the result:

| First Number | Second Number | Output |
|---|---|---|
| 0 | 0 | 0 (Addition and Subtraction) |
| 0 | Finite | Finite (Addition ) |
| 0 | Finite | -Finite (Subtraction) |
| Finite | 0 | Finite (Addition and Subtraction) |
| Finite | +-Infinite | +-Infinite (Addition and Subtraction) |
| +Infinite | +Infinite | Nan (Addition) |
| +Infinite | -Infinite | Nan (Addition) |
| -Infinite | -Infinite | Nan (Subtraction) |
| -Infinite | -Infinite | Nan (Subtraction) |
| Nan | Nan | Nan (Addition and Subtraction) |

Is both numbers are normal, then we proceed to the next component which is the Adder.

This component will perform the algorithm described in the introduction following mainly the flowchart presented in Figure 4, except the 0-verification part which was done in the previous

stage and will reverse the sign bit of the second number in case the selected operation is subtraction.

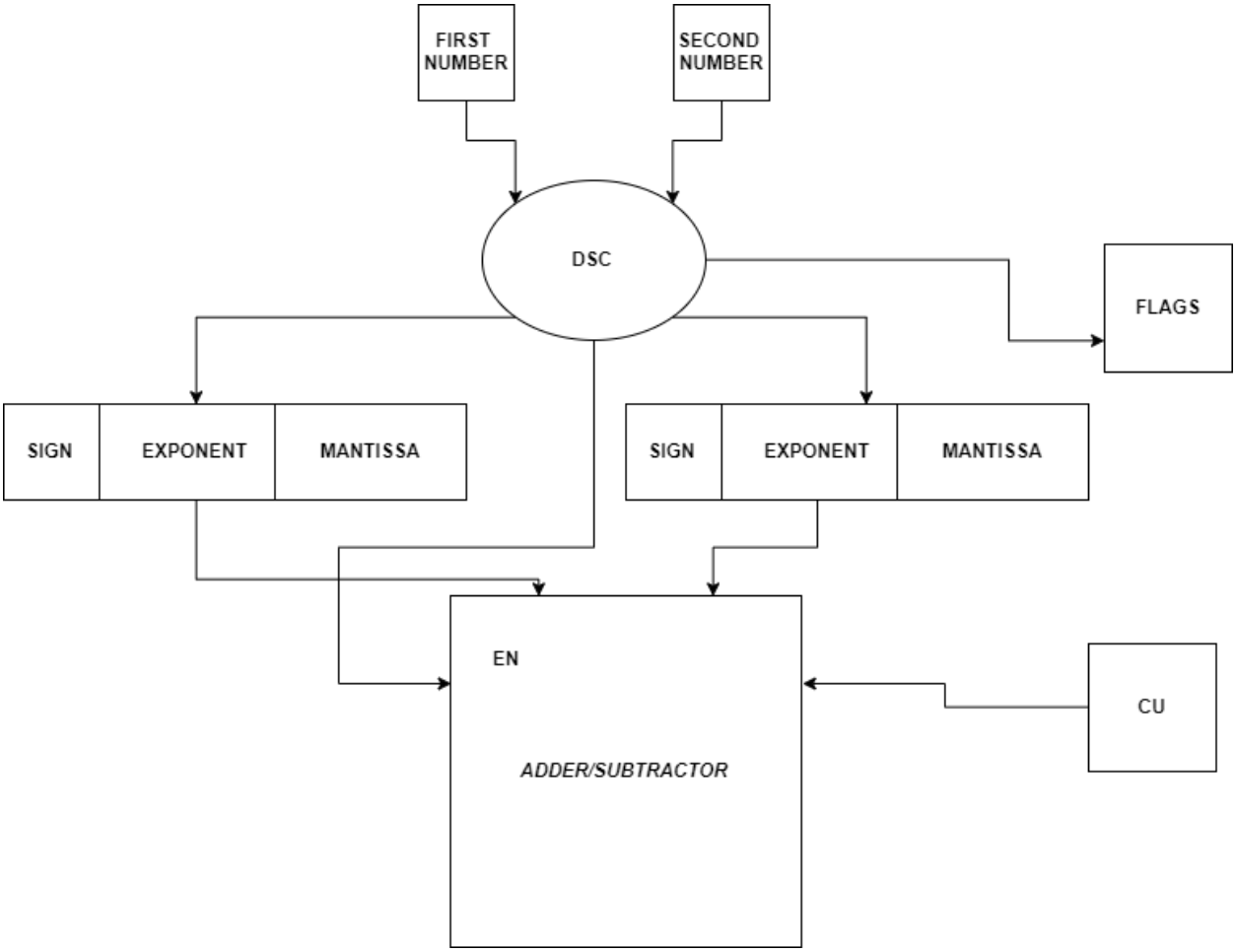


*Figure 6– System's blocks*

**4.The Flags component**

This component is responsible also with special cases, and will have the flags for overflow, zero flag and other special cases which includes that one or both operands are infinity or Nan .

## Implementation

When describing the implementation I will start from general to particular.

Firstly, I will describe how the Top module works, and after that each component.

It is important to mention that the ROM component was not used in the designing part, but in the simulation part. I wrote in ROM 15 values in order to make the testing part easier. We work with numbers on 32 bits, but the address of the ROM is an integer, so it made the entire process of assigning values much easier.'

**The TOP module**

This is the big black box of the project, with the final inputs and outputs.

The numbers, the operation, output result, flags, a reset button and of course the clock.

I've declared some additional intermediary signals in the architecture and the process looks like this:

```
Label1: PreAdder port map (clk,A,B,Operation,sc,zero,sum1);
Label2: CU port map (A,B,clk,enable,Operation,reset,'1',done,sum2,ovf);

FlagReg(0)<=zero;
FlagReg(1)<=sc;
FlagReg(2)<=ovf;

process(zero,sc,done,ovf)
begin
if (zero='1' or sc='1' ) then
DisplayResult<=sum1;
else
if(done='1') then
DisplayResult<=sum2;
else
DisplayResult<="00000000000000000000000000000000";
end if;
end if;
end process;
```

*Figure 7- The main process in top module*

So basically, there is a process on the flags and on the done signal, which tells us when the second component finished calculations.

Firstly, outside the process I assign to the output values of the flags their intermediary signal, and after that comes the process.

When the Zero Flag or the Special Case Flag are 1, it means that the input numbers are in one of the cases treated by the PreAdder and we can take the result directly from there.

Otherwise, it means that the CU component does the calculations. While the calculations are still rendering, the output is a dummy one, zero. When done is one, we display the result from that component.

**The PreAdder**

The PreAdder is a rather primitive component, as it is only composed from if/else sets.

```
begin
process(Number1,Number2,Operation)
variable var: std_logic_vector(2 downto 0);
begin
var:="000";
-- if both are 0 or -0 the result is 0
if   ( Number1 = "00000000000000000000000000000000" and Number2 = "00000000000000000000000000000000" )    then
    DisplayResult<= "00000000000000000000000000000000";
    ZeroFlag<='1';
    SpecialCase<='0';
    var:="001";
end if;
if   ( Number1 = "10000000000000000000000000000000" and Number2 = "10000000000000000000000000000000" )    then
    DisplayResult<= "00000000000000000000000000000000";
    ZeroFlag<='1';
    SpecialCase<='0';
    var:="001";
end if;
if   ( Number1 = "00000000000000000000000000000000" and Number2 = "10000000000000000000000000000000" )    then
    DisplayResult<= "00000000000000000000000000000000";
```

*Figure 8- Section of code from PreAdder*

As you can see in the section above, I have tested some of the simpler cases, in this particular example if we have a combination of +0 and -0. If that is the case, then the result is 0 and we mark the 0 flag with one. Also, I used a helper variable, in order to keep track on the cases and for avoiding confusions.

If it stays until the end, then there is no special case or zero flag, and the output is a dummy one only to avoid conflicts, because it will take its value from the other component.

**The CU**

This component is responsible with all the part that is not suited for the PreAdder.

This works going through 5 states: WAIT_STATE, ALIGN_STATE, ADDITION_STATE, NORMALIZE_STATE, OUTPUT_STATE.

I also took some intermediary signals for the sign, mantissa and exponent of each number and the sum.

In the Waiting state, we initialize overflow with ‘0’ and we split the numbers in their parts.

At the exponent, we pad with a 0, in order to perform the difference of the exponents.

At the mantissa, we pad with “01”, 0 for carry and the 1 is the leading one of the mantissa.

After that, we check the Operation input, and is it is ‘1’, we change the sign of B.

In the align state, is where we need to align exponents. If the exponents difference is greater than 23, we consider that the other number is very insignificant, and the output is the greater number.

```
when ALIGN_STATE =>
  if unsigned(A_exp) > unsigned(B_exp) then
    diff := signed(A_exp) - signed(B_exp);
    if diff > 23 then
      sum_mantissa <= A_mantissa;
      sum_exp <= A_exp;
      sum_sgn      <= A_sgn;
      state      <= OUTPUT_STATE;
    else
      sum_exp <= A_exp;
      B_mantissa(24-to_integer(diff) downto 0)  <= B_mantissa(24 downto to_integer(diff));
      B_mantissa(24 downto 25-to_integer(diff)) <= (others => '0');
      state                                     <= ADDITION_STATE;
    end if;
```

*Figure 9- Exponent align example*

If not, then we downshift the needed exponent, A if its exponent is smaller or B's otherwise.

Because as stated before, these operations can only be performed if the numbers have equal exponents.

If they are equal, we go in the next state transmitting the exponent of A, but we could also transmit the exponent of B, it was only a matter of choice.

Next comes the Addition state, in which we add the mantissas.

The code here is straight-forward, we make the equality check which was implemented by using an exclusive or function compared with 0. If the exponents are equal, we add the mantissas. Otherwise, we perform a subtraction, which has as first term (minuend) the greater mantissa.

```
when ADDITION_STATE =>                    --Mantissa addition
  state <= NORMALIZE_STATE;
  if (A_sgn xor B_sgn) = '0' then
    sum_mantissa <= std_logic_vector((unsigned(A_mantissa) + unsigned(B_mantissa)));
    sum_sgn      <= A_sgn;
  elsif unsigned(A_mantissa) >= unsigned(B_mantissa) then
    sum_mantissa <= std_logic_vector((unsigned(A_mantissa) - unsigned(B_mantissa)));
    sum_sgn      <= A_sgn;
  else
    sum_mantissa <= std_logic_vector((unsigned(B_mantissa) - unsigned(A_mantissa)));
    sum_sgn      <= B_sgn;
  end if;
```

*Figure 10- Adding the mantissas*

After that, before we are done, it comes the Normalization state.

If the first bit of the mantissa is one, then it overflowed and we need to downshift the mantissa and increase the exponent. Also here we give a '1' to the overflow flag.

```
when NORMALIZE_STATE =>
  if(sum_mantissa(24) = '1') then
  sum_mantissa <= '0' & sum_mantissa(24 downto 1);
   sum_exp        <= std_logic_vector((unsigned(sum_exp)+ 1));
   off:='1';
   state      <= OUTPUT_STATE;
  elsif(sum_mantissa(23) = '0') then
      sum_mantissa <= sum_mantissa(23 downto 0) & '0';
      sum_exp <= std_logic_vector((unsigned(sum_exp)-1));
      state<= NORMALIZE_STATE;
  else
    state <= OUTPUT_STATE;
```

*Figure 11-The normalization state implementation*

If we don't have an overflow, but the 23-rd bit of the intermediary signal of the mantissa is 0, then we need to shift until the leading one appears (remember that the mantissa is of the form 1. value).

If no normalization is needed, we go in the Output state.

Here we only assign to the output signals the intermediary signals in order to display the final result

and we change the value of done signal to '1' so we know that the calculations are finished.

## Testing and Validation

The following examples were used to test the correct functionality of the algorithm using numbers in the floating-point representation:

- A=2.5 = 01000000001000000000000000000000 = 40200000 hex

  B=3 =0 10000000100000000000000000000000 = 40400000 hex

  Operation= 1 (subtraction)

  Expected result=-0.5 = 10111111000000000000000000000000 = bf000000 hex

  Flag(0)= zero flag =0

  Flag(1)= special case = 0

  Flag(2)= overflow = 0

  Actual Result

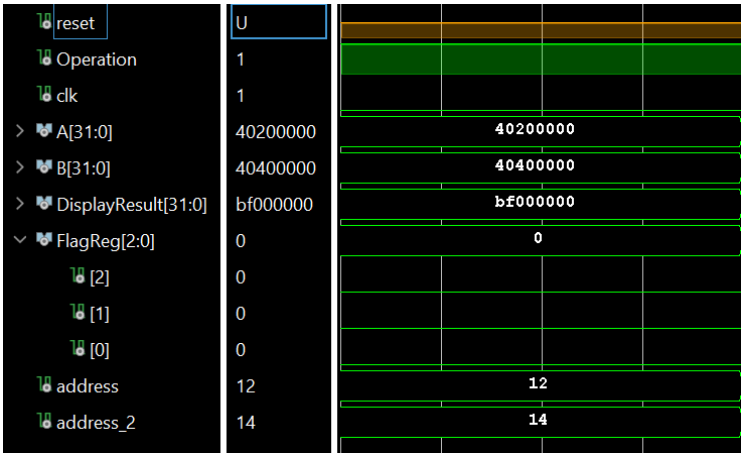


*Figure 12– Simulation1*

Test passed

- A= -11.5 = 11000001001110000000000000000000 = c1380000 hex

  B= 117 = 01000010111010100000000000000000 = 42ea0000 hex

  Operation= 1 (subtraction)

  Expected result= -128.5 = 11000011000000001000000000000000= c3008000 hex

  Flag(0)= zero flag =0

  Flag(1)= special case = 0

  Flag(2)= overflow = 0

  Actual Result

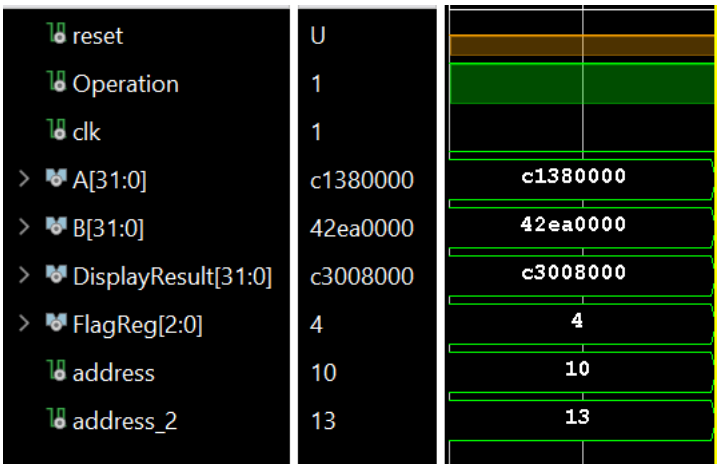


*Figure 13– Simulation2*

Test passed

- A= 1000 = 01000100011110100000000000000000= 447a0000 hex

B= 1000 = 01000100011110100000000000000000= 447a0000 hex

Operation= 1 (subtraction)

Expected result= 0 = 00000000000000000000000000000000= 00000000hex

Flag(0)= zero flag =1

Flag(1)= special case = 0

Flag(2)= overflow = 0

Actual Result

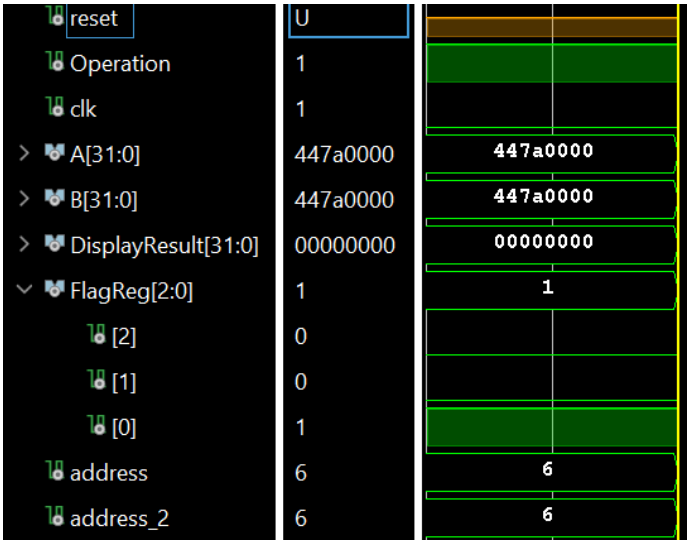


*Figure 14– Simulation3*

Test passed

- A= +inf = 01111111100000000000000000000000= 7f800000 hex

  B=+inf = 01111111100000000000000000000000= 7f800000 hex

  Operation= 0 (addition)

  Expected result= +inf = 01111111100000000000000000000000= 7f800000 hex

  Flag(0)= zero flag =0

  Flag(1)= special case = 1

  Flag(2)= overflow = 1

  Actual Result

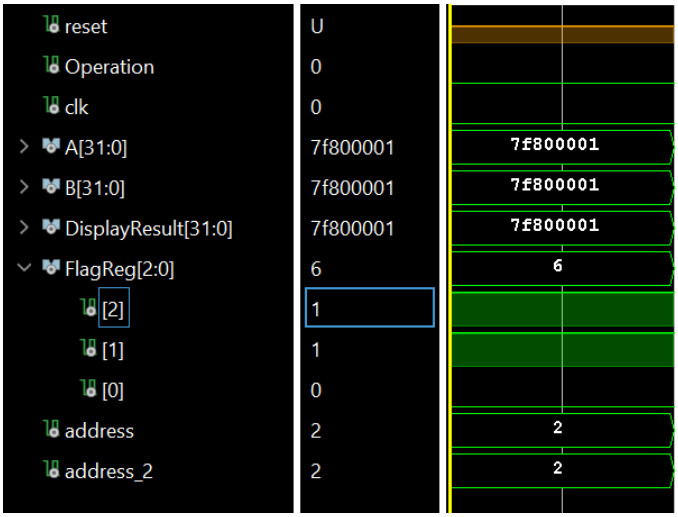


*Figure 15– Simulation4*

Test passed.

- A= 3.40282346639e+38 = 01111111011111111111111111111111= 7f7fffff hex

  B= 3.40282346639e+38 = 01111111011111111111111111111111= 7f7fffff hex

  Operation= 0 (addition)

  Expected result= NaN = 01111111111111111111111111111111= 7fffffff hex

  Flag(0)= zero flag =0

  Flag(1)= special case = 0

  Flag(2)= overflow = 1

  Actual Result

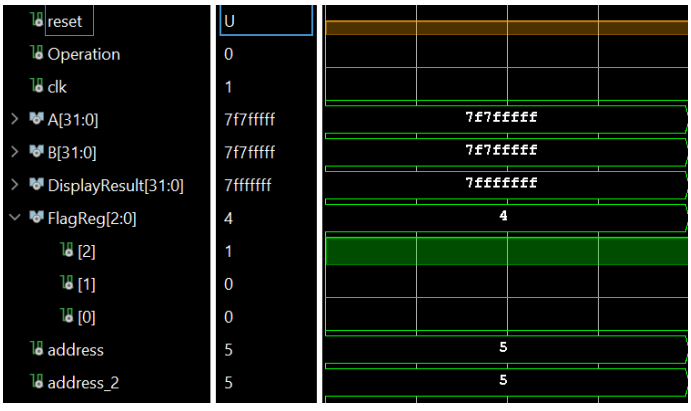


*Figure 16– Simulation5*

- Test Passed

- A= 3.40282346639e+38 = 01111111011111111111111111111111= 7f7fffff hex

  B= 0.25 = 10111110100000000000000000000000 = be800000 hex

  Operation= 0 (addition)

- Expected result= 3.40282346639e+38 = 01111111011111111111111111111111= 7f7fffff hex

  Flag(0)= zero flag =0

  Flag(1)= special case = 0

  Flag(2)= overflow = 0

  -That is because when we sum up a very big number with a very small one, we neglect de very small number and keep only the bigger one.

  Actual Result

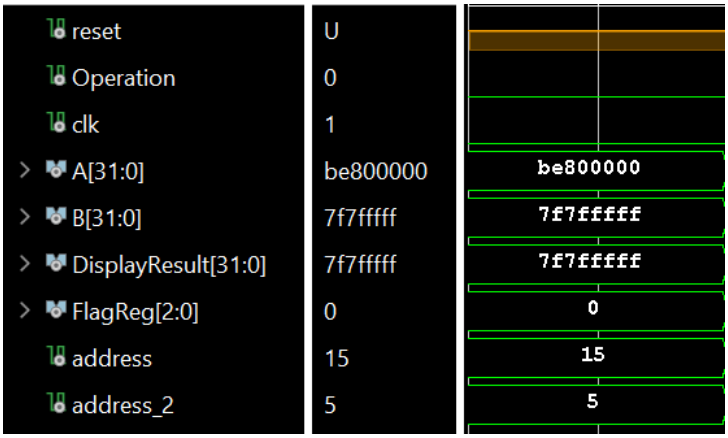


*Figure 17– Simulation6*

Test passed.

## Conclusions

The goal of this project was to design, implement and test 2 operations on the standard IEEE format on 32 bits. The difference from the usual implementations of addition and subtraction is that this unit will perform them on floating point numbers with single precision.

The difficult part was to implement the algorithm which I knew how to perform on paper, in VHDL. Also, I also needed to take into account that many things here does not execute sequentially and for the signals the value is assigned only at the end of the process. Also, the synchronization between components was also difficult.

Vivado in particular did not make implementing this project any easier, because it has a lot of bugs and the errors there are very vague compared to Visual Studio or InteliJ. The good part was that I learned to use debugging mode and breakpoints in order to see where and what was I doing wrong.

The documentation on the internet is also rather poor on this subject, more exactly it only states the steps on the algorithm in general and you can't find very many examples to see exactly how it behaves on edge cases.

But, putting together algorithms found on the internet, the YouTube videos and other ideas of implementation I managed to implement a mostly functional FPU . I say mostly because in every application testing can show only the presence of bugs not the absence of them, and it was impossible to test each and every test possible.

## Bibliography


1. "Design and Implementation of Floating Point ALU with Parity Generator Using Verilog HDL" [Online] Available: https://www.iosrjournals.org/iosr-jvlsi/papers/vol5-issue5/Version-1/I05515459.pdf
2. "Signed Binary Numbers" [Online] Available: https://www.electronics-tutorials.ws/binary/signed-binary-numbers.html
3. Adding IEEE-754 Floating Point Numbers [Online] Available: https://www.youtube.com/watch?v=mKJiD2ZAlwM
4. Floating Point Tutorial [Online] Available: https://www.rfwireless-world.com/Tutorials/floating-point-tutorial.html
5. Floating Point Operations [Online] Available: https://www.geeksforgeeks.org/computer-arithmetic-set-2/
6. IEE Standard Floating Point Numbers [Online] Available: https://www.geeksforgeeks.org/ieee-standard-754-floating-point-numbers/
7. Design and implementation of IEEE-754 Decimal Floating Point adder, subtractor and multiplier, Dr. Srinivas Badithela [Online] Available: https://www.researchgate.net/profile/Srinivas-B
8. Design Of High Performance IEEE- 754 Single Precision (32 bit) Floating Point Adder Using VHDL, Preethi Sudha Gollamudi, M. Kamaraju [Online] Available:

   https://www.ijert.org/research/design-of-high-performance-ieee-754-single-precision-32-bit-floating-point-adder-using-vhdl-IJERTV2IS70837.pdf
9. Addition of IEEE 754 Single Precision Floating Point Numbers[Online] Available:

   https://www.youtube.com/watch?v=DuOwT2hZOiw

Table of Figures