TECHNICAL UNIVERSITY OF CLUJ-NAPOCA, FACULTY OF AUTOMATION AND COMPUTER SCIENCE

Arithmetic logic unit for floating point multiplication & division

Ștefan Ciuprina

05-Jan-21

Contents

1. Introduction

1.1 Context

The project consists of designing, implementing and testing the operations of multiplication and division on floating point numbers with single precision. This device will be useful for people in need of a calculator capable of doing these operations, or could be integrated in another project, like a processor, which will use this device in order to perform its tasks.

The floating point representation of a number is used in case of very big or very small numbers. In order to represent a floating point number, we need 3 fields: sign, exponent (the magnitude of the number) and the mantissa. What is to remember is that this representation is not the representation of real numbers from mathematics, but one that the computer can work with more easily. However, some problems may arise by using this, since many numbers have no precise representation in floating point (take for example the number 0.3), and therefore, the result will be an approximate one.

1.2 Specifications

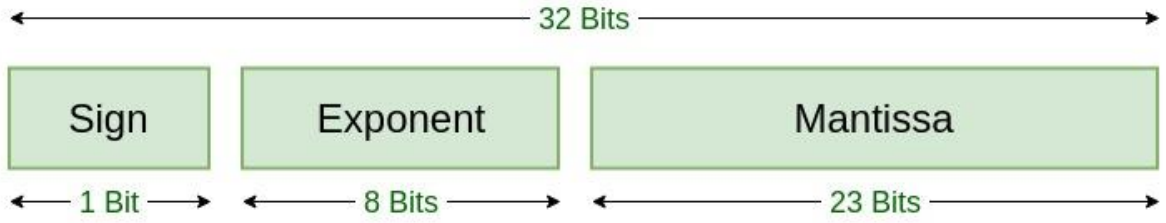
The project will be written in VHDL (Very High Speed Hardware Description Language), simulated in ActiveHDL and then programmed into a Basys3 Board using the Vivado Design Suite. The device will be able to represent internally the number in the Standard IEEE representation (the one mentioned earlier, with the sign, exponent and the mantissa; more on this in the next chapter), convert from and into two's complement representation, which will be helpful for the implementation of the operations, and present the result in a readable format.

1.3 Objectives

The objective of this project is to introduce and store two's complement fractional numbers in the floating point representation. From two's complement, we need to normalize the number and from that form, the extraction of the necessary components for making the floating point equivalent to the number. The device can then perform the operations. The main steps for multiplication are: adding the exponents, multiplying the mantissas and adjusting the result (shifting the mantissa to the left and decrement the exponent if necessary); for the division, the main steps are: substracting the exponents, dividing the mantissas, adjusting the result (if it is necessary). The final objective is to represent, using the 7 segment displays of the FPGA board, the result of the operation.

2. Bibliographic study

As mentioned earlier, the floating point numbers will be in the Standard IEEE representation, which is composed of the sign bit, the exponent and the mantissa. A visual representation is shown below.



Single Precision
IEEE 754 Floating-Point Standard

Source: [1]

As the picture above shows, we have 1 bit for the sign, 8 bits for the exponent and 23 bits for the mantissa, each having its own purpose:

- The sign – as the name suggests, represents the sign of the number, i.e. 0 represents a positive number and 1 represents a negative number
- The exponent – the exponent field needs to represent both positive and negative exponents; a bias is added to the actual exponent in order to get the stored exponent
- The mantissa – the mantissa is part of a number in scientific notation or a floating-point number, consisting of significant digits; here we have only 2 digits (0 and 1); a normalized mantissa is one with only one '1' to the left of the decimal

We can calculate the actual value of a floating point number from its IEEE representation using the following formula:

$$(-1)^{sign} * 2^{exponent-127} * 1.mantissa$$

Let's take for example the number 85.125.

We will first convert it to the Standard IEEE representation.

85 = 1010101 (in binary)
0.125 = 0.001 (in binary)
85.125 = 1010101.001 = 1.010101001 x 2^6

Sign: 0
Exponent: 127 + 6 = 133
Mantissa: 010101001 (to complete all 23 bits of the mantissa, we will fill the others with 0)

Finally, we have the IEEE representation:
0 10000101 01010100100000000000000

To convert back to the original number, we use the formula mentioned earlier.

$$(-1)^0 * 2^{133-127} * 1.33 = 85.12$$

Steps for the multiplication algorithm:

- Non-signed multiplication of mantissas: it must take account of the integer part, implicit in normalization. The number of bits of the result is twice the size of the operands (48 bits);
- Normalization of the result: the exponent can be modified accordingly
- Addition of the exponents, taking into account the bias;
- Calculation of the sign.

Source: [2]

Steps for the division algorithm:

- XOR the sign bits;
- If divisor is 0, the result is infinite (an "inf" message will be shown to the user on the 7 segment display);
- Subtract the exponents and bias to it;
- Divide the mantissa;
- Normalize the result if required.

Source: [3]

3. Analysis

3.1 Multiplication

In order to follow the multiplication algorithm, we need to design 2 components: a multiplicator (for the mantissas) and an adder (for adding the exponents).

To multiply the mantissas, we concatenate a '1' as most significant bit to both of them and then send them as input to the multiplicator. The result will therefore be on 48 bits (24 bit number x 24 bit number). We have 2 cases: if the first bit of the result is '1', we take the next 23 bits and normalize the result by adding one to the exponent; the other case is when the first bit is '0', when the second bit will be '1' every time, and therefore, we take the next 23 bits, following this '1'.

The next step of this algorithm is to add the two 8 bit exponents, which will be done using a simple adder on 8 bits.

The final step of this algorithm is to compute the sign, which will be done by making a XOR operation between the two sign bits.

However, there are some cases when this algorithm won't work properly, namely in an overflow situation (if the two operands are very big, the addition of the exponent will overflow, causing the result to be a very small number), or in an underflow situation (if the two operands are very small, causing the result to be a very big number).

For the multiplication component, I will use the *Shift and Add Multiplication* technique. The idea of this method is to take each digit of the multiplier in turn and multiply it with the multiplicand, shifting the multiplicand one position to the left after each step. We will start with the least significant bit of the multiplier, shifting it to the right after each operation, such that the next least significant bit will be taken in the next turn. The algorithm stops after the multiplier register is empty, taking as many clock cycles as there are bits in the multiplier (in our case, it will be a maximum of 24 bits).

For the addition of the exponents, I will use an 8 bit *Ripple Carry Adder* on 8 bits.

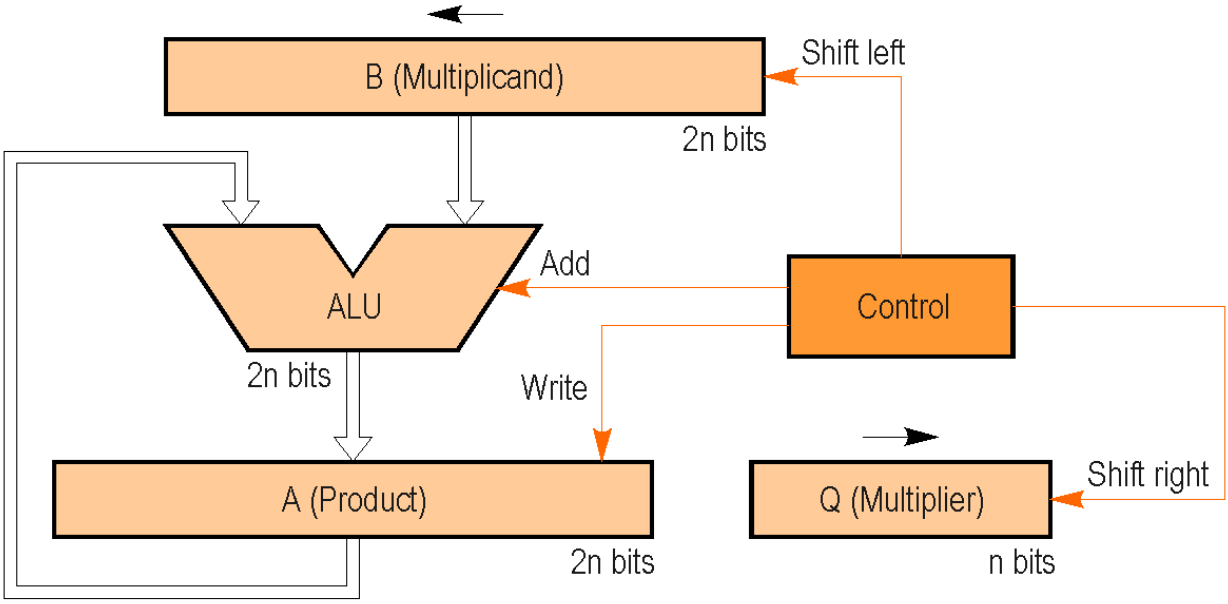
3.2 Division

The design of the division algorithm is similar to the one of the multiplication. We need to subtract the exponents, divide the mantissas and XOR the sign bits. For the subtraction, a subtractor will be used, while for the division of the mantissas, I will use a divider which will follow the algorithm of division as it is on paper (checking if the denominator fits in the numerator, if so shifting a '1' into the result and subtracting the numerator from the denominator, otherwise shifting a '0' into the result; in the next step, one more bit from the numerator will be take into consideration, until we reach 23 bits).

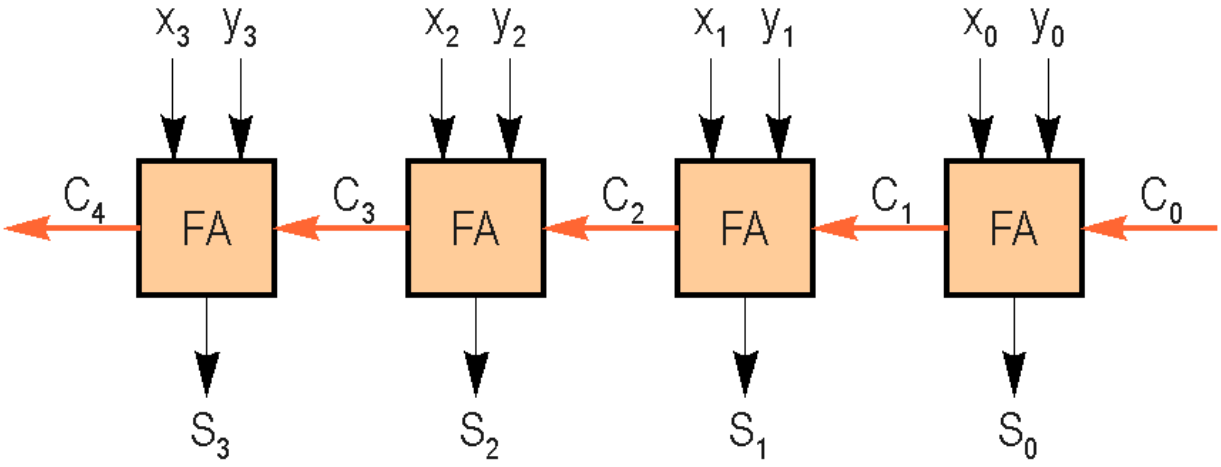
The drawbacks of this design are similar to those of the multiplication, namely exponent overflow (when dividing a very big number with a very small one, the result of exponent subtraction could overflow and a very small number would be shown), exponent underflow (when dividing a very small number with a very big one, the result of exponent subtraction could underflow and a very big number would be shown), but also division by zero, which will be solved by displaying an "inf" (infinity) message on the 7 segment display.

4. Design

The main components used for multiplication are the *Shift and Add Multiplicator* and the *Ripple Carry Adder*. The block diagrams of these components are shown below:



Shift and Add Multiplicator

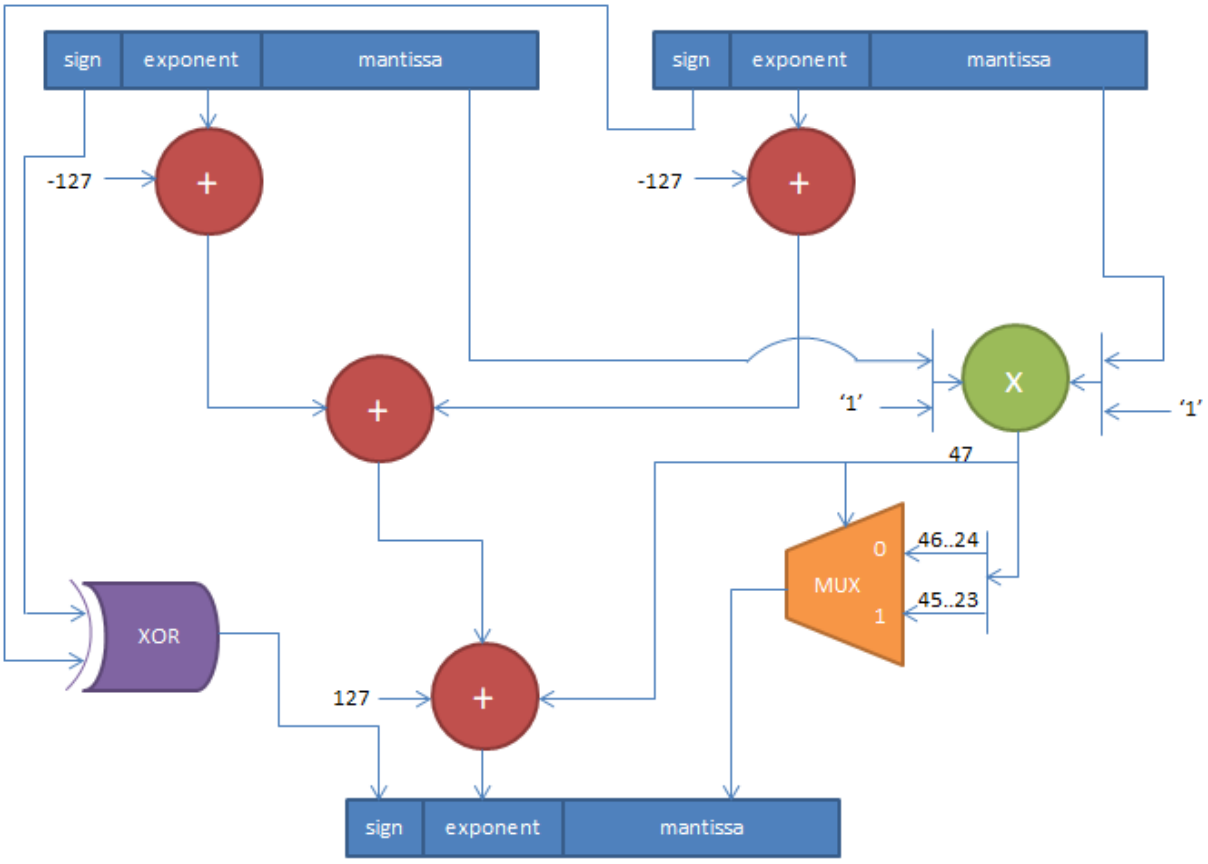


Ripple Carry Adder on 4 bits

-the block design for 8 bits is similar to this one, the only difference being the use of 8 full adders instead of 4-

Source: [4]

The block diagram for the whole multiplication algorithm explained above is shown below:



The diagram shows exactly how the algorithm we described works. We use adders to subtract the bias from the exponents, add the results and then add the bias back to the result, along with the normalization bit from the multiplication. The mantissas are being multiplied (after each being concatenated with a '1' as the most significant bit), and, depending on the most significant bit, we consider the bits starting from the second or third bit, as mentioned in **chapter 3.1**. The sign bit is computed by making a XOR operation between the sign bits of the two operands.

For the division, the design is pretty similar. The only notable differences are the use of a fixed point divider instead of the multiplicator and the transformation into *two's complement representation* of the second number, so that the second operand will be subtracted from the first one instead of being added to it.

The divider component will follow the division algorithm as on paper. We will take the first most significant bits of the numerator as the length of the denominator. We then subtract from these bits the denominator and add the next bit to the result of the subtraction, and also a '1' to the final result. If the result of the subtraction has fewer bits than the denominator or is smaller, we add a '0' to the final result and add the next bit of the nominator to the subtraction result. We repeat these steps until there are no more bits in the numerator.

5. Implementation

The implementation of this project consists of: creating the component for letting the user select each digit and the decimal point location (NumbersRegister), converting the inputted number into simple binary format and from that format to the IEEE representation (BCDtoIEEE). Then, having the two inputted numbers, the device can apply the multiplication/division on the two numbers, returning the result in the IEEE representation, having to convert it back to BCD for it to be displayed on the 7 segment display.

The implementation of the multiplicator will be presented below. It has been implemented using generics, being able to be used on as many bits as wanted and it follows the design presented in the previous chapter.

---------CODE---------

```
library IEEE;

use IEEE.STD_LOGIC_1164.ALL;

use IEEE.STD_LOGIC_UNSIGNED.ALL;


entity Multiplicator is

generic(n : natural);

port(

        multiplicand : in STD_LOGIC_VECTOR(2*n-1 downto 0);

        multiplier : in STD_LOGIC_VECTOR(n-1 downto 0);

        load : in STD_LOGIC;

        clk : in STD_LOGIC;

        product : out STD_LOGIC_VECTOR(2*n-1 downto 0);

        done : out STD_LOGIC

        );


end Multiplicator;
```

```
architecture Structural of Multiplicator is

component MultiplicandRegister is
generic(n : natural);
port(
        input : in STD_LOGIC_VECTOR(2*n-1 downto 0);
        write : in STD_LOGIC;
        shift_left : in STD_LOGIC;
        clk : in STD_LOGIC;
        output : out STD_LOGIC_VECTOR(2*n-1 downto 0)
        );
end component;

component MultiplierRegister is
generic(n : natural);
port(
        input : in STD_LOGIC_VECTOR(n-1 downto 0);
        write : in STD_LOGIC;
        shift_right : in STD_LOGIC;
        clk : in STD_LOGIC;
        output : out STD_LOGIC_VECTOR(n-1 downto 0)
        );
end component;

component ProductRegister is
```

```
generic(n : natural);
port(
        input : in STD_LOGIC_VECTOR(2*n-1 downto 0);
        write : in STD_LOGIC;
        reset : in STD_LOGIC;
        clk : in STD_LOGIC;
        output : out STD_LOGIC_VECTOR(2*n-1 downto 0)
        );
end component;

component Adder is
generic(n : natural);
port(
        input1 : in STD_LOGIC_VECTOR(2*n-1 downto 0);
        input2: in STD_LOGIC_VECTOR(2*n-1 downto 0);
        add : in STD_LOGIC;
        output : out STD_LOGIC_VECTOR(2*n-1 downto 0)
        );
end component;

signal multiplicand_reg, adder_output, product_output : STD_LOGIC_VECTOR(2*n-1 downto 0);
signal multiplier_reg : STD_LOGIC_VECTOR(n-1 downto 0);
signal multiplicand_write_signal, shift_left_signal, multiplier_write_signal, shift_right_signal,
product_write_signal, add_signal, reset : STD_LOGIC;

begin
```

```
MultiplicandReg : MultiplicandRegister generic map (n) port map (multiplicand,
multiplicand_write_signal, shift_left_signal, clk, multiplicand_reg);

MultiplierReg : MultiplierRegister generic map (n) port map (multiplier, multiplier_write_signal,
shift_right_signal, clk, multiplier_reg);

ProductReg : ProductRegister generic map (n) port map (adder_output, product_write_signal,
reset, clk, product_output);

Addr : Adder generic map (n) port map (multiplicand_reg, product_output, add_signal,
adder_output);



product <= product_output;



add_signal <= multiplier_reg(0);



process (load, multiplier_reg)

        begin

                if(load = '1') then

                        multiplicand_write_signal <= '1';

                        shift_left_signal <= '0';

                        multiplier_write_signal <= '1';

                        shift_right_signal <= '0';

                        product_write_signal <= '0';

                        done <= '0';

                        reset <= '1';

                else

                        reset <= '0';

                        multiplicand_write_signal <= '0';
```

```
                shift_left_signal <= '1';
                multiplier_write_signal <= '0';
                shift_right_signal <= '1';
                if (multiplier_reg = 0) then
                        product_write_signal <= '0';
                        done <= '1';
                else
                        product_write_signal <= '1';
                        done <= '0';
                end if;
        end if;
    end process;

end Structural;
```

The division algorithm follows the algorithm as on paper. Its implementation is presented below:

---CODE---

```
--divider for dividing numbers of the form (1...)/(1...) (same number of bits). the result will be of the form (1.something) or (0.something),
--the integer part being the MSB, the rest of the 23 bits being part of the fractional part
library IEEE;
use IEEE.STD_LOGIC_1164.ALL;
use IEEE.STD_LOGIC_UNSIGNED.ALL;

entity divider is
```

```
port(
        input1 : in STD_LOGIC_VECTOR(23 downto 0);
        input2 : in STD_LOGIC_VECTOR(23 downto 0);
        load : in STD_LOGIC;
        clk : in STD_LOGIC;

        result : out STD_LOGIC_VECTOR(24 downto 0);
        done : out STD_LOGIC
        );
end divider;

architecture Behavioral of divider is

        signal step : STD_LOGIC_VECTOR(5 downto 0);
        signal input1_signal, input2_signal, current_divident, result_signal : STD_LOGIC_VECTOR(47
downto 0); --current_divident delete
        signal approximate, done_signal : STD_LOGIC;

begin

        process(clk)
        begin
                if(rising_edge(clk)) then
                        if(load = '1') then
                                --input1_signal <= input1(22 downto 0) & '0';
```

```
                input2_signal(47 downto 24) <= (others => '0');
                input2_signal(23 downto 0) <= input2;
                result_signal <= (others => '0');
                step <= "101111"; --48 steps for 48 bits (from 0 to 47)
                done_signal <= '0';
                --current_divident(23 downto 1) <= (others => '0');
                --current_divident(0) <= input1(23);
                current_divident(47 downto 24) <= (others => '0');
                current_divident(23 downto 0) <= input1;
            else
                if(done_signal = '0') then
                    if(current_divident >= input2_signal) then
                        result_signal(conv_integer(step)) <= '1';
                        --current_divident <= (current_divident(22 downto 0) -
input2_signal(22 downto 0)) & input1_signal(23);
                        current_divident <= (current_divident(46 downto 0) -
input2_signal(46 downto 0)) & '0';
                    else
                        result_signal(conv_integer(step)) <= '0';
                        current_divident <= current_divident(46 downto 0) &
'0'; --shift right and add new bit to current divident
                    end if;
                    --input1_signal <= input1_signal(22 downto 0) & '0'; --shift right
                    if(step = "00000") then
                        done_signal <= '1';
                    else
                        step <= step - 1;
```

```
                                        end if;
                                end if;
                        end if;
                end if;
        end process;

        process(result_signal)
        begin
                approximate <= '0';
                for i in 22 downto 0 loop
                        if (result_signal(i) = '1') then
                                approximate <= '1';
                        end if;
                end loop;

                if(approximate = '1') then
                        result <= result_signal(47 downto 23) + 1;
                else
                        result <= result_signal(47 downto 23);
                end if;
        end process;

        done <= done_signal;

end Behavioral;
```

The IEEE multiplication and division has been made using the multiplicator and divider presented above, following the schematic presented in the previous chapter. The implementation is presented below:

IEEE multiplication:

---CODE---

```
library IEEE;
use IEEE.STD_LOGIC_1164.ALL;
use IEEE.STD_LOGIC_UNSIGNED.ALL;

entity IEEE_Multiplication is
port(
	input1, input2 : in STD_LOGIC_VECTOR(31 downto 0);
	load : in STD_LOGIC;
	clk : in STD_LOGIC;
	output : out STD_LOGIC_VECTOR(31 downto 0);
	done : out STD_LOGIC
	);
end IEEE_Multiplication;

architecture Behavioral of IEEE_Multiplication is

	component Multiplicator is
	generic(n : natural);
	port(
		multiplicand : in STD_LOGIC_VECTOR(2*n-1 downto 0);
		multiplier : in STD_LOGIC_VECTOR(n-1 downto 0);
```

```
            load : in STD_LOGIC;
            clk : in STD_LOGIC;
            product : out STD_LOGIC_VECTOR(2*n-1 downto 0);
            done : out STD_LOGIC
            );

    end component;

    signal signInput1, signInput2, signOutput : STD_LOGIC;
    signal exponentInput1, exponentInput2, exponentOutput : STD_LOGIC_VECTOR(7 downto 0);
    signal mantissaInput1, mantissaInput2, mantissaOutput : STD_LOGIC_VECTOR(22 downto 0);

    signal mantissaInput1_WITHONE : STD_LOGIC_VECTOR(47 downto 0);
    signal mantissaInput2_WITHONE : STD_LOGIC_VECTOR(23 downto 0);

    signal multiplicationResult : STD_LOGIC_VECTOR(47 downto 0);

begin

    --parsing the ieee numbers
    signInput1 <= input1(31);
    exponentInput1 <= input1(30 downto 23);
    mantissaInput1 <= input1(22 downto 0);

    signInput2 <= input2(31);
```

```
        exponentInput2 <= input2(30 downto 23);
        mantissaInput2 <= input2(22 downto 0);

        --computing the sign
        signOutput <= signInput1 xor signInput2;

        --instantiating the Multiplication component
        mantissaInput1_WITHONE(47 downto 24) <= (others => '0');
        mantissaInput1_WITHONE(23 downto 0) <= '1' & mantissaInput1;
        mantissaInput2_WITHONE <= '1' & mantissaInput2;

    MUL : Multiplicator generic map (24) port map (mantissaInput1_WITHONE,
mantissaInput2_WITHONE, load, clk, multiplicationResult, done);

        --computing the exponent
        exponentOutput <= (((exponentInput1 - 127) + (exponentInput2 - 127)) +
multiplicationResult(47) + 127);

        --computing the mantissa
        with multiplicationResult(47) select mantissaOutput <=
                multiplicationResult(45 downto 23) when '0',
                multiplicationResult(46 downto 24) when OTHERS;

        output <= signOutput & exponentOutput & mantissaOutput;

end Behavioral;
```

IEEE division:

---CODE---

```
library IEEE;
use IEEE.STD_LOGIC_1164.ALL;
use IEEE.STD_LOGIC_UNSIGNED.ALL;

entity IEEE_Division is
port(
	input1, input2 : in STD_LOGIC_VECTOR(31 downto 0);
	load : in STD_LOGIC;
	clk : in STD_LOGIC;
	output : out STD_LOGIC_VECTOR(31 downto 0);
	done : out STD_LOGIC
	);
end IEEE_Division;

architecture Behavioral of IEEE_Division is

	component divider is
	port(
		input1 : in STD_LOGIC_VECTOR(23 downto 0);
		input2 : in STD_LOGIC_VECTOR(23 downto 0);
		load : in STD_LOGIC;
		clk : in STD_LOGIC;
```

```
            result : out STD_LOGIC_VECTOR(24 downto 0);
            done : out STD_LOGIC
            );
    end component;

    signal signInput1, signInput2, signOutput, normalization : STD_LOGIC;
    signal exponentInput1, exponentInput2, exponentOutput : STD_LOGIC_VECTOR(7 downto 0);
    signal mantissaInput1, mantissaInput2, mantissaOutput : STD_LOGIC_VECTOR(22 downto 0);

    signal mantissaInput1_WITHONE, mantissaInput2_WITHONE : STD_LOGIC_VECTOR(23 downto
0);

    signal divisionResult : STD_LOGIC_VECTOR(24 downto 0);

begin

    --parsing the ieee numbers
    signInput1 <= input1(31);
    exponentInput1 <= input1(30 downto 23);
    mantissaInput1 <= input1(22 downto 0);

    signInput2 <= input2(31);
    exponentInput2 <= input2(30 downto 23);
    mantissaInput2 <= input2(22 downto 0);
```

```
--computing the sign

signOutput <= signInput1 xor signInput2;


--instantiating the division component

mantissaInput1_WITHONE <= '1' & mantissaInput1;

mantissaInput2_WITHONE <= '1' & mantissaInput2;

DIV : divider port map (mantissaInput1_WITHONE, mantissaInput2_WITHONE, load, clk,
divisionResult, done);


--computing the exponent

normalization <= not divisionResult(24);

exponentOutput <= (exponentInput1 - exponentInput2) + 127 - normalization;


--computing the mantissa

mantissaOutput <= divisionResult(23 downto 1) when normalization = '0' else divisionResult(22
downto 0);


output <= signOutput & exponentOutput & mantissaOutput;


end Behavioral;
```

---

6. Testing and validation

The workings of the project have been tested on the Basys 3 FPGA board and can be seen in the following Youtube video:

https://www.youtube.com/watch?v=H25zlllse5s

Simulations have also been made in ActiveHDL, proving the correctness of the algorithms for multiplication and division. An example for each is presented below:

- We'll use the following 2 numbers:

12.75 = (0x414c0000) IEEE

2.73 = (0x402eb852) IEEE

- Multiplication example

12.75 * 2.73 = 34.8075

34.8075 = (0x420b3ae1) IEEE

ActiveHDL multiplication simulation:

input1	414C0000
input2	402EB852
output	420B3AE1

- Division example

12.75 / 2.73 = 4.67032967032967

4.67032967032967 = (0x40957357) IEEE

ActiveHDL division simulation:

input1	414C0000
input2	402EB852
output	40957357

As testing above shows, the device works as expected.

7. Conclusions

Our goal for creating an arithmetic logic unit capable of making multiplications and divisions between floating point numbers with single precision has been reached. It provides full Basys 3 support, being ready for everyday use. Nevertheless, the components used here could also be used in other devices, other than the Basys 3, with better display formats and friendlier user designs.

8. Bibliography

[1]: https://www.geeksforgeeks.org/ieee-standard-754-floating-point-numbers/

[2]: http://lslwww.epfl.ch/pages/teaching/cours_lsl/sl_info/FPMultiplier.pdf

[3]: http://ijarece.org/wp-content/uploads/2015/07/IJARECE-VOL-4-ISSUE-7-1935-1939.pdf

[4]: https://moodle.cs.utcluj.ro/mod/resource/view.php?id=20490