Technical University of Cluj-Napoca
Faculty of Automation and Computer Science
Computer Science Department

Floating Point Arithmetic

Daniel-Peter Reckerth

Group 30434

Structure of Computing Systems (SCS)

Contents

1. Introduction

1.1 Context

This project aims at designing and implementing an arithmetic logic unit (ALU) system which implements the arithmetical operations – addition and subtraction – of single-precision floating point format numbers.

It can be further used in problems regarding floating point operations or implemented in a higher component as basic operations, consequently used in different applications. It can be also integrated in any sort of microprocessor unit which benefits from this kind of operations.

1.2 Specification

Floating point numbers and arithmetic use real numbers as approximation to support a trade-off between range and precision. As the name implies, *floating point* means that a number's radix point (decimal/binary point) can be placed anywhere relative to the significant digits of the numbers.

The designing phase will take into account the conceptual and logical representation of the system, i.e. correctness of operations. The implementation and simulation will be achieved in an IDE, in Xilinx environment and thereafter implemented on a Basys3 FPGA board.

Capabilities comprises correctly representation of the IEEE 754 single-precision binary floating-point format (binary32), and operations, namely addition and subtraction of such numbers (e.g. two 32-bit floating point numbers).

1.3 Objectives

Designing, simulating and implementing this application and testing it on different input test cases.

Study, design and implement methods of performing addition and subtraction. Compare and decide upon the best representation of the numbers: decimal radix with 7-digit precision. Define and decide how to implement and represent the numbers in the Xilinx environment.

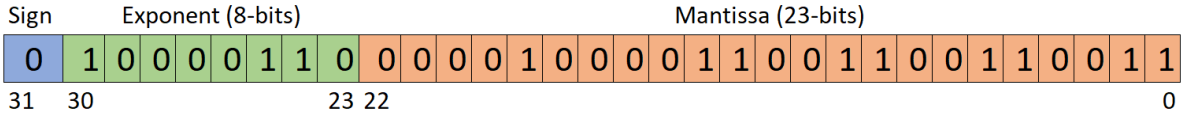
Simulate the project, verify the results of the testing cases and then implement it on a Basys3 FPGA board.

2. Analysis and Design

2.1 Theoretical Background

As we work with single-precision binary floating point format, we will present here the IEEE754 standard for floating-point numbers, as a technical standard. It addresses many problems found in different floating-point implementations in order to make everything more reliable and portably.

The floating-point number are represent using a short format on 4 Bytes (32-bits). The layout for this is the following.



Where:

- bit 32 (1 bit): sign bit (**S)**
- bits 30-23 (8 bits): exponent – magnitude of the number (E)
- bits 22-0 (23 bits): mantissa / fraction / significand (M)

The sign bit can be 0/1 (positive/negative number). It denotes the sign of the significand.

The exponent is a fixed number of bits (8) ranging from 0 to 255. It is represented using excess-n notation. This means that a number called *characteristic / biased exponent* is stored instead of the exponent itself. And it is calculated by adding 127 ($07F_{16}$) to the exponent. Biasing is done because exponents have to be signed values in order to be able to represent both very large and very small values. For single-precision number therefore the exponent is stored in range 1…254 and interpreted by subtracting the bias for an 8-bit exponent (127) to get an exponent in the range -126 … +127.

The significand includes 23 bits to the right of the binary point and an implicit leading bit, with value 1. The total precision is 24 bits. The mantissa is calculated by first normalizing the number: $NO = 1,<mantissa\_digits> \cdot 2^{exponent}$

The real value is computed as follows:

$$(-1)^{sign} \cdot (1, < mantissa\ digits >) \cdot 2^{exponent}$$

In our case:

1) Sign = 0
2) Exponent = 1000110 (B) = 134 (D)
   Biased exponent = 134 – 127 = 7
3) Mantissa = 00001000011001100110011 (B)
4) Number = $(-1)^0 \cdot (1,00001000011001100\ldots) \cdot 2^7$ = 10000100.0011001100110011 (B) = 132.1999969482421875 ≅ 132.2

2.2 Addition and Subtraction Algorithms

Both operations are complex when working with floating-point numbers. The mantissas must be taken care of, pre-processed so that the exponents are equal.

The addition and subtraction of two floating-point numbers implies that the mantissas must be aligned. Thereafter the exponents must be compared deciding which number is larger. If the difference of exponents is slight, then the number with the smaller exponent may be larger if the mantissas are not normalized, resulting in reduced precision.

We will present a general algorithm for addition, applied similar to subtraction.

1) load the operands
2) compare exponents
   - **case1**: $e_x = e_y$
     - add mantissas (Mx+My) – step 3)
     - copy exponent
   - **case2**: $e_x > e_y$ *and* $(d=e_x-e_y) <$ *# of mantissas bits*
     - mantissa My aligned by right-shifting with d
   - **case3**: $e_x >> e_y$ *(Y too small) and* $(d=e_x-e_y) >=$ *# of mantissas bits*
     - copy X in result
     - go to 4)
   - **case4**: $e_x < e_y$ *and* $(d=e_y-e_x) <$ *# of mantissas bits*
     - mantissa Mx aligned by right-shifting with d
     - add mantissas (Mx+My)
   - **case5**: $e_x << e_y$ *(X too small) and* $(d=e_y-e_x) >=$ *# of mantissas bits*
     - copy Y in result
     - go to 4)
3) add mantissas
4) realign result if necessary
   right/left-shift the resulting mantissas until the integer part is 0 and the 1st bit after the decimal point is 1
   increment or decrement the exponent in accordance with the shifting

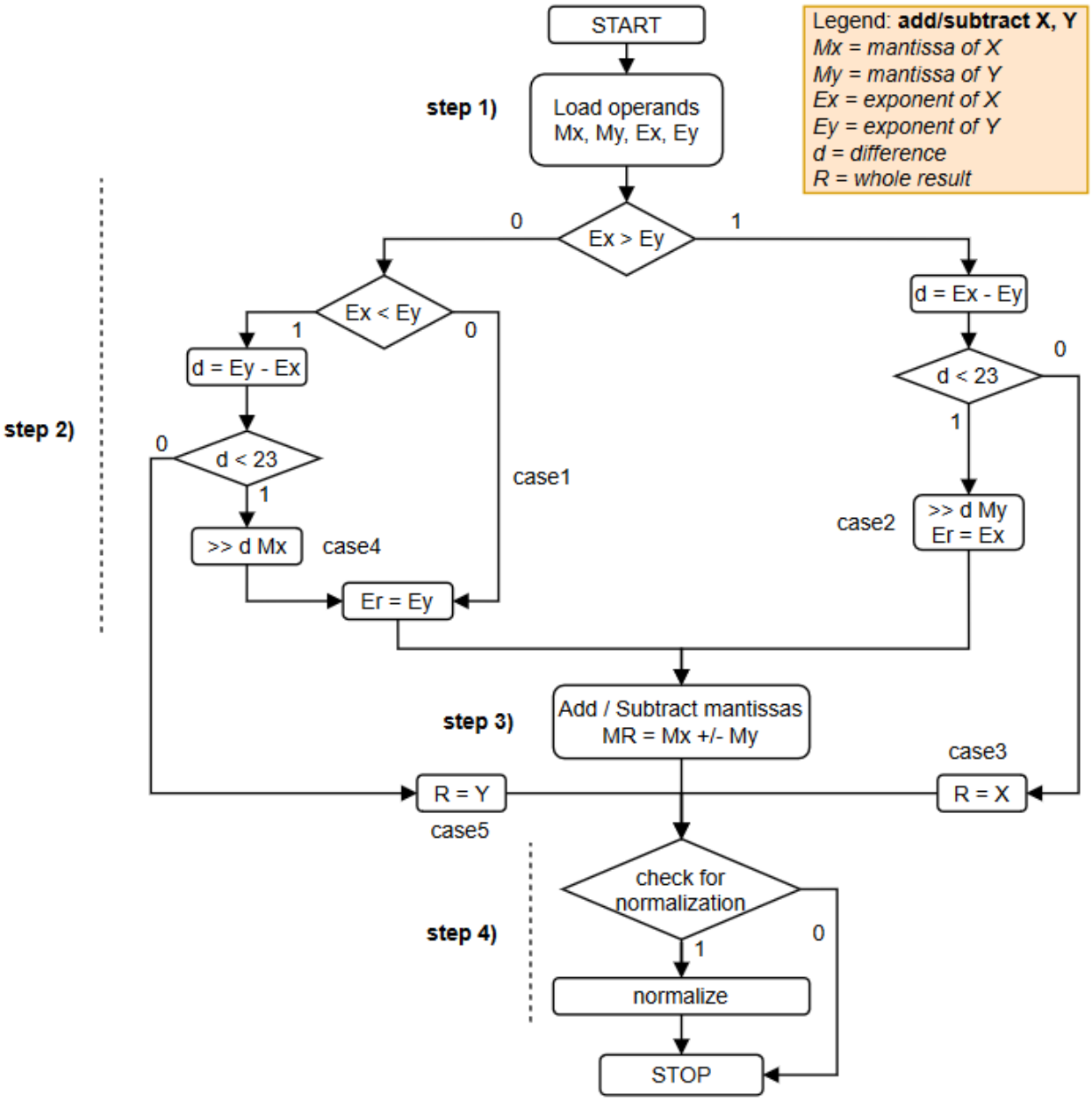
As we can notice the algorithm is straightforward. Its core is based on the compare cases of the exponents of the two floating-point numbers. By << and >> we refer to a very small number which is rounded up and when added or subtracted does not change the result.

If both exponents are equal than we simply add the mantissas and copy one of the exponents.

However, if the exponent of the first number is larger than the second exponent and of course the second number is not very small (the exponent's difference is smaller than 23-bit mantissa) than we have to align the second number's mantissa by right-shifting with the difference than the mantissas are added. If the second number is too small, we simply declare the first number to be the result.

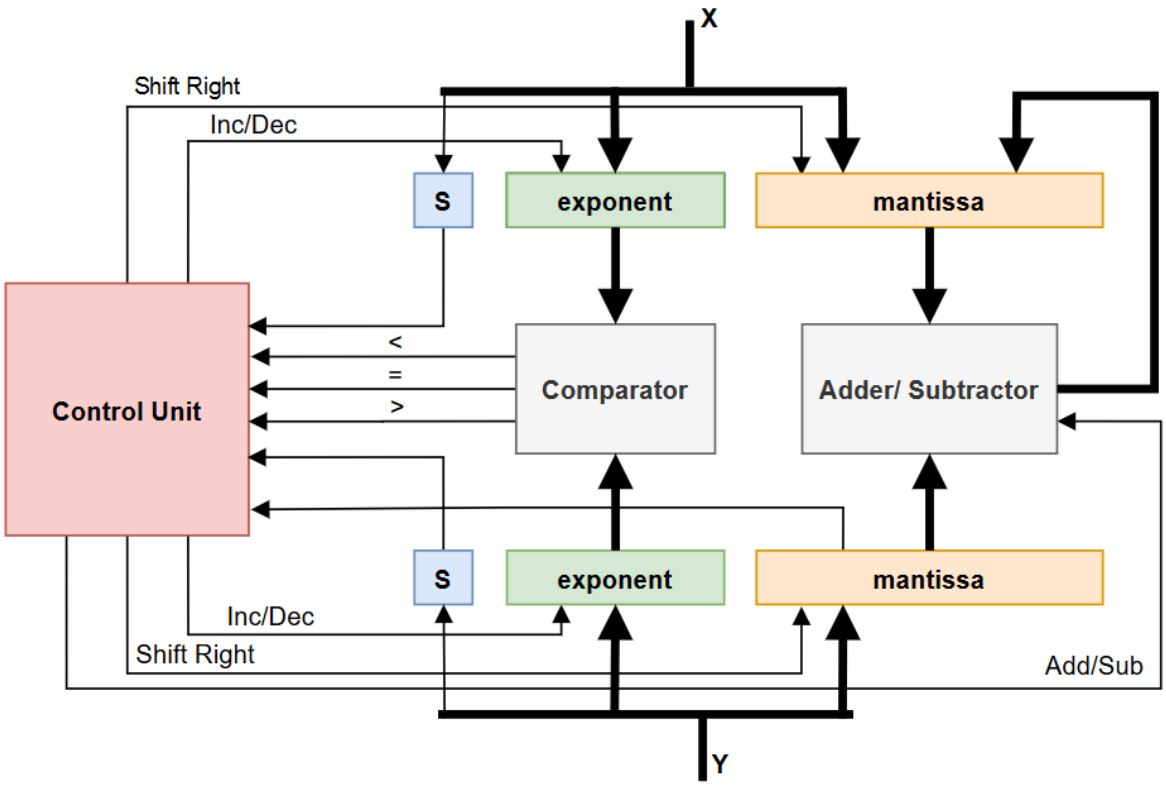
If the opposite case is true, i.e. the exponent of the first number is smaller than the second exponent and the first number is not too small, than we have to align the first number's mantissa by right-shifting with the difference. Otherwise, we simply declare the second number as the result.

A flowchart of the algorithm is provided for better understanding.



3. Implementation

As long as implementation criteria is concerned, we started from a logical scheme of the system. Below we provide the diagram.



As it can be seen we have depicted different modules. However, the implementation per se does not use this structural model fully. We had made a comparator component but we have included the control unit logic and the actual addition and subtraction in the main block file logic.

Our system receives as inputs two 32-bit operands X, Y and outputs the result of their addition R also on 32-bit. The structure is the one mentioned before (sign, exponent, mantissa). Although we speak of subtraction this is the same as addition with negative number so it is not a real subtraction. The two inputs can have the same sign (negative or positive) or different sign. So, for both addition and subtraction we have the same situation: either we add two same-sign numbers or two different-sign numbers.

Examples are presented below:

1) No normalization needed, same sign (positive)

X = 2.25	0	1	0	0	0	0	0	0	1	0	0	1	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
Y = 134.0625	0	1	0	0	0	0	1	1	0	1	0	0	0	0	1	1	0	0	0	0	1	0	0	0	0	0	0	0	0	0	0	0

40100000 (H)

Ex = 1000 0000 (B) = 128 (D) = 80 (H)

43061000 (H)

Ey = 1000 0110 (B) = 134 (D) = 86 (H)

d = 6, d < 23 => shift_right(Mx, 6)

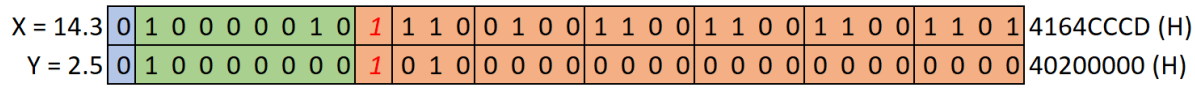
>> 6 Mx	0	0	0	0	0	1	0	0	1	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
My	1	0	0	0	0	1	1	0	0	0	0	0	1	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
R = 136.3125	0	1	0	0	0	0	1	1	0	1	0	0	0	1	0	0	0	0	1	0	1	0	0	0	0	0	0	0	0	0	0	0

+

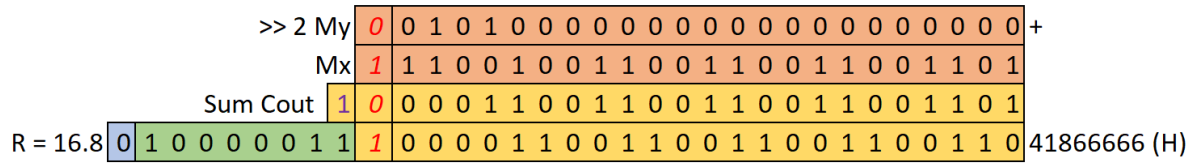
4308500 (H)

- the red one bit is actually the leading bit '1' which is imposed by the format, it is not included in the 23 bits of the mantissa, but need for computation

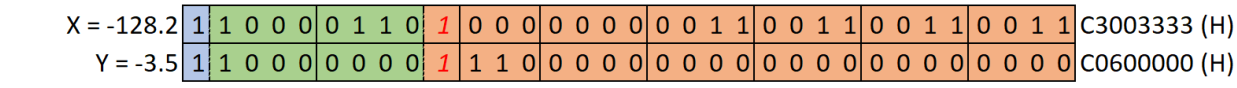
2) Normalization needed, same sign (positive)



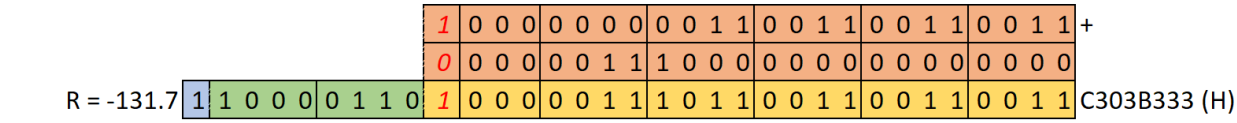
Ex = 10000010 (B) = 130 (D) = 82 (H)
Ey = 10000000 (B) = 128 (D) = 80 (H)
d = Ex -Ey
d = 00000010 (B) = 2 (D) = 2 (H)
d < 23 TRUE => shift_right(MY, 2)



3) No normalization, same sign (negative)

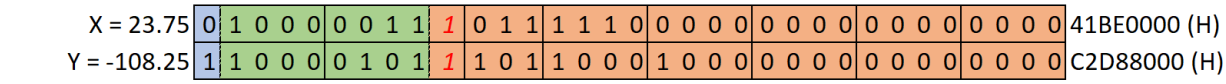


Ex = 1000 0110 = 86 (H)
Ey = 1000 0000 = 80 (H)
d = 6, d < 23 => shift_right(My, 6)

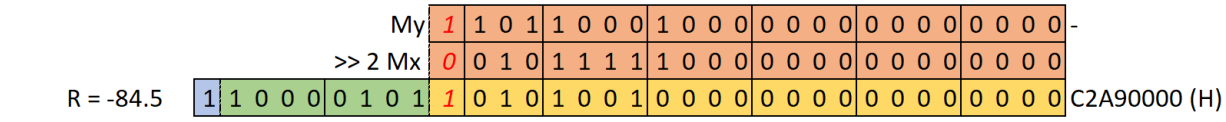


- as it can be seen even when both signs are '1', i.e negative same addition applies

4) Different signs



Ex = 1000 0011 = 83 (H)
Ey = 1000 0101 = 85 (H)
d = 2, d < 23 => shift_right(Mx, 2)



- we perform a subtraction from the greater mantissa instead of working with the 2's complement

Before presenting the actual source code, we will briefly present how the system is implemented.

The comparator receives the exponent of both numbers and outputs three different signals: EQ (equal), GT (greater than), LT (less than) with respect to X exponents. Below is the source code.

```vhdl
library IEEE;
use IEEE.STD_LOGIC_1164.ALL;
use IEEE.NUMERIC_STD.ALL;

entity cmp_exp is
  Port (Ex: in std_logic_vector(8 downto 0);
        Ey: in std_logic_vector(8 downto 0);
        EQ: out std_logic;
        GT: out std_logic;
        LT: out std_logic
        );
end cmp_exp;

architecture Behavioral of cmp_exp is

begin

    process(Ex, Ey)
    begin
        if Ex > Ey then
            GT <= '1';
        elsif Ex < Ey then
            LT <= '1';
        else
            EQ <= '1';
        end if;
    end process;

end Behavioral;
```

The main block is a finite state machine. We have defined the following states: waiting, align, add, normalize, output. As inputs we have declared besides the clock and the two 32-bit numbers a start and reset signal and we also have a finalise (done) signal besides the 32-bit output.

We have used internal signals in order to deconstruct the input. We have three 1-bit signal for the sign of the operands (`Sx, Sy, Sr`), three 9-bit signals for exponents (`Ex, Ey, Er`) and 25-bit signals for the mantissas (`Mx, My, Mr`). For the exponents we needed an additional bit in order to perform arithmetic and for the mantissas we needed two additional bits: one for the carry out of the arithmetic operation and one to prepend the leading bit '1'.

We instantiated the comparator for the exponent and we have internal signals for the outputs of the comparator.

In the main process we have declared a variable `d` which is used to hold the difference between the exponents as stated in the algorithm. Also, we have two variable `case3`, `case5` (flags) to signals when these cases were reached.

In the `WAITING` state if the start signal is high, we deconstruct both X, Y into sign, exponent mantissa. The `ALIGN` state is the actual flowchart of dealing with the mantissa shifting operations. Based on the outputs of the comparator and the dimensions of d we decide in which case we are and we perform the corresponding actions. The `ADD` state deals with the actual addition or subtraction. If the signs are the same then we add the mantissas, else we see which mantissa is greater and subtract from it the other one. The `NORMALIZE` state checks if something needs to be normalized. If we have a carry we right-shift, if not and the 23rd bit is low we have

to left-shift until the leading '1' appears. The exponents are incremented or decremented accordingly. In the `OUTPUT` state we simply build our output R. Below is the source code.

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

entity FPU is
  Port (clk:   in std_logic;
        rst:   in std_logic;
        start: in std_logic;
        X:     in std_logic_vector(31 downto 0);
        Y:     in std_logic_vector(31 downto 0);
        R:     out std_logic_vector(31 downto 0);
        done:  out std_logic
        );
end FPU;

architecture Behavioral of FPU is

type state_type is (WAITING, ALIGN, ADD, NORMALIZE, OUTPUT);
signal state: state_type := WAITING;

signal Sx, Sy, Sr: std_logic;
signal Ex, Ey, Er: std_logic_vector(8 downto 0);
signal Mx, My, Mr: std_logic_vector(24 downto 0);

component cmp_exp is
  Port (Ex: in std_logic_vector(8 downto 0);
        Ey: in std_logic_vector(8 downto 0);
        EQ: out std_logic;
        GT: out std_logic;
        LT: out std_logic
        );
end component;

signal expEQ, expGT, expLT: std_logic;

begin

    exponent_comp: cmp_exp port map(Ex => Ex, Ey => Ey, EQ => expEQ, GT => expGT, LT => expLT);

    process(clk, rst)
        variable d: signed(8 downto 0);
        variable case3, case5: std_logic := '0';
    begin
        if rst = '1' then
            state <= WAITING;
            done  <= '0';
        elsif rising_edge(clk) then
            case state is
                when WAITING =>
                    if start = '1' then
                        Sx <= X(31);
                        Sy <= Y(31);
                        Mx <= "01" & X(22 downto 0);
                        My <= "01" & Y(22 downto 0);
                        Ex <= '0' & X(30 downto 23);
                        Ey <= '0' & Y(30 downto 23);
                        state <= ALIGN;
                    else
                        state <= WAITING;
                    end if;
                when ALIGN =>
                    if expGT = '1' then
                        d := signed(Ex) - signed(Ey);
                        if d < 23 then
                            Er <= Ex;
                            My <= std_logic_vector(shift_right(unsigned(My), to_integer(d)));
                            state <= ADD;
                        else
                            R <= X;
                            case3 := '1';
                            state <= OUTPUT;
                        end if;
                    elsif expLT = '1' then
                        d := signed(Ey) - signed(Ex);
```

```
                    if d < 23 then
                        Er <= Ey;
                        Mx <= std_logic_vector(shift_right(unsigned(Mx), to_integer(d)));
                        state <= ADD;
                    else
                        R <= Y;
                        case5 := '1';
                        state <= OUTPUT;
                    end if;
                else
                    Er <= Ey;
                    state <= ADD;
                end if;
            when ADD =>
                state <= NORMALIZE;
                if (Sx xor Sy) = '0' then -- X, Y have same sign
                    Mr <= std_logic_vector((unsigned(Mx) + unsigned(My)));
                    Sr <= Sx;
                elsif (unsigned(Mx) >= unsigned(My)) then
                    Mr <= std_logic_vector((unsigned(Mx) - unsigned(My)));
                    Sr <= Sx;
                else
                    Mr <= std_logic_vector((unsigned(My) - unsigned(Mx)));
                    Sr <= Sy;
                end if;
            when NORMALIZE =>
                if unsigned(Mr) = to_unsigned(0, 25) then
                    Mr <= (others => '0');
                    Er <= (others => '0');
                    state <= OUTPUT;
                elsif (Mr(24) = '1') then
                    Mr <= '0' & Mr(24 downto 1);
                    Er <= std_logic_vector((unsigned(Er) + 1));
                    state <= OUTPUT;
                elsif (Mr(23) = '0') then
                    Mr <= Mr(23 downto 0) & '0';
                    Er <= std_logic_vector((unsigned(Er) - 1));
                        state <= NORMALIZE;
                    else
                        state <= OUTPUT;
                    end if;
               when OUTPUT =>
                    if (case3 = '0' and case5 = '0') then
                        R(31) <= Sr;
                        R(30 downto 23) <= Er(7 downto 0);
                        R(22 downto 0) <= Mr(22 downto 0);
                        done <= '1';
                    else
                        done <= '1';
                    end if;

                    if start = '0' then
                        done <= '0';
                        state <= WAITING;
                    end if;
               when others => state <= WAITING;
            end case;
        end if;
    end process;

end Behavioral;

```

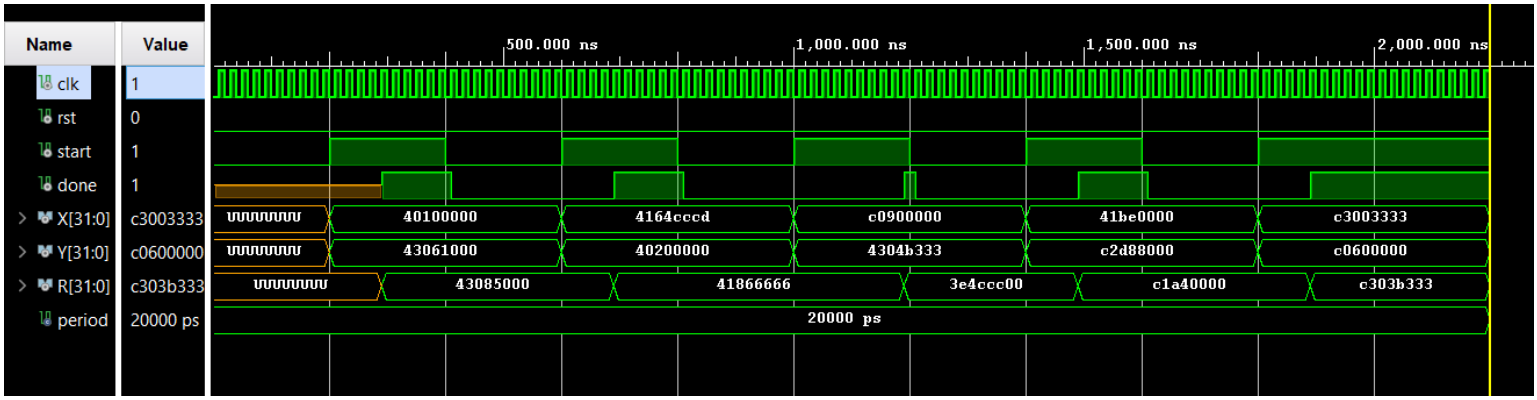
4. Test and results

In order to test the above, we have created a testbench in which, in the stimulus process, we have given X, Y different values. The values are the same as in the examples presented above. Below is the source code of the testbench.

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

entity tb is
--  Port ( );
end tb;

architecture Behavioral of tb is

component FPU is
  Port (clk:   in std_logic;
        rst:   in std_logic;
        start: in std_logic;
        X:     in std_logic_vector(31 downto 0);
        Y:     in std_logic_vector(31 downto 0);
        R:     out std_logic_vector(31 downto 0);
        done:  out std_logic
        );
end component;

signal clk, rst, start, done : std_logic := '0';
signal X, Y, R: std_logic_vector(31 downto 0);

constant period : time := 20 ns;
shared variable end_sim: boolean := false;

begin

    UUT: FPU port map(clk => clk, rst => rst, start => start, X => X, Y => Y, R => R, done => done);

    clk_generator: process
    begin
        if not end_sim then
            clk <= '0';
            wait for period / 2;
            clk <= '1';
            wait for period / 2;
        else wait;
        end if;
    end process clk_generator;


     stimulus: process
     begin
          -- X = 2.25 = 40100000 (H)
          -- Y = 134.0625 = 43061000(H)
          -- R = 136.3125 = 4308500 (H)
          -- OBS: no normalization
          start <= '0';
          wait for 10 * period;
          X <= x"40100000";
          Y <= x"43061000";
          start <= '1';
          wait for 200 ns;

          -- X = 14.3 = 4164CCCD (H)
          -- Y = 2.5 = 40200000(H)
          -- R = 16.8 = 41866666 (H)
          -- OBS: normalized
          start <= '0';
          wait for 10 * period;
          X <= x"4164CCCD";
          Y <= x"40200000";
          start <= '1';
          wait for 200 ns;
```

```
67
68          -- X = -4.5= C0900000 (H)
69          -- Y = 132.7 = 4304B333 (H)
70          -- R = 128.2 = 43003333 (H)
71          start <= '0';
72          wait for 10 * period;
73          X <= x"C0900000";
74          Y <= x"4304B333";
75          start <= '1';
76          wait for 200 ns;
77
78          -- X = +23.75= 41BE0000 (H)
79          -- Y = -108.25 = C2D88000 (H)
80          -- R = -84.5 = C2A90000(H)
81          start <= '0';
82          wait for 10 * period;
83          X <= x"41BE0000";
84          Y <= x"C2D88000";
85          start <= '1';
86          wait for 200 ns;
87
88          -- X = -128.2 = C3003333 (H)
89          -- Y = -3.5 = C0600000 (H)
90          -- R = -131.7 = C303B333  (H)
91          start <= '0';
92          wait for 10 * period;
93          X <= x"C3003333";
94          Y <= x"C0600000";
95          start <= '1';
96          wait for 200 ns;
97
98          wait for 200 ns;
99          end_sim := true;
100         wait;
101     end process;
102
103 end Behavioral;
```

Running the following testbench should produce the same result as described in the comments or as presented in the examples in chapter three (above). Here are the simulation results:



5. Conclusions

This project's goal of creating an adder for single precision floating point number was achieved. It was a good opportunity to learn more about how floating-point numbers are stored and how are the arithmetic operations performed on them.

This project can be further improved by implementing it on a real FPGA board in order to be physically tested. Also, different modules can be created in order to make the code more structural. Furthermore, multiplication, division, and other logical operation operations can be included, resulting in a more functional floating-point FPU.

6. Bibliography

- online resources (websites, articles, documents ..)
  - https://en.wikipedia.org/wiki/Floating-point_arithmetic
  - https://en.wikipedia.org/wiki/Single-precision_floating-point_format
  - https://www.cs.uaf.edu/2000/fall/cs301/notes/notes/node51.html
- books, papers, guides
  - UTCN-AC Structure of Computer Systems Laboratory Guide
  - Course slides of SCS (prof. Gheorghe Sebestyen) – ALU course
  - Floating Point – course notes Auckland CS