Department of Computer Science
Technical University of Cluj-Napoca

Floating Point Arithmetic Logic Unit

Laboratory activity 2024-2025

Name: Suciu Andrei
Group: 30431
Email: suciu.se.an@student.utcluj.ro

Structure of Computer Systems

Contents

1 Project Overview

1.1 Specifications

Design an arithmetic logic unit on a 32-bit architecture capable of performing addition and multiplication operations on floating point numbers, encoded in the IEEE 754 standard. The ALU will be implemented on an FPGA, and must run and display various example operations. The ALU must be able to perform the following operations:

- 32-bit floating point addition
- 32-bit floating point multiplication

1.2 Design

The implementation of the arithmetic logic unit will be designed in VHDL, using Vivado. The ALU will be contained within a master project which will provide the input and output display functionalities accoring to the FPGA provided at the laboratory. The project will be broken down into multiple smaller sources, each of which will be implemented using a behavioural style. Finally, all the project sources will be combined structurally to form a cohesive unit.

2 Bibliographic Research

2.1 Arithmetic Logic Unit

In computing, an arithmetic logic unit (ALU) is a combinational digital circuit that is able to perform arithmetic on binary numbers. It is a fundamental building block of many other circuits, such as CPU or GPU. The ALU receives two input operands, an operation code, to decide which operation to perform, and optionally, a carry-in bit. Then, the result of the operation will be sent on the output signal.

2.2 Floating Point

In computing, numbers are represented in binary notation, as a series of bits, which can take values of either 1 or 0. In order to represent fractional parts, we must place a comma somewhere inside the number's representation. However, this greatly reduces the range of numbers which can be represented on a limitied amount of bits. In order to increase the range, we can imagine the comma moving (floating) left or right across the binary representation, according to what number we want to represent.
Thus, we will to reserve a few bits for storing the position of the comma, creating a floating point number.

2.3 IEEE 754

The IEEE 754 defines the most widealy used standard for floating point numbers. Instead of storing the position of the comma, and the number itself, we will take advantage of scientific notation. In decimal scientific notation, the magnitude of the number is represented as a power of 10, and the "mantissa," essentially the core of the number, is stored as a fractional number between 1 and 10.
Moving this format to binary, we can observe that no matter the number, the mantissa will always start with a leading "1." Thus, we can assume this part of the number and save 1 bit. The magnitude represents the exponent of a power of 2. Furthermore, there is one more bit for the sign of the number, 0 for positive, and 1 for negative.
Overall, for a 32 bit number, the IEEE 754 standard formats the bits as such:

- 1 bit for the sign of the number
- 8 bits for the exponent
- 23 bits for the mantissa

The IEEE 754 standard also mentions the format for special values. There are five such numbers:

- Zero — All exponent bits and all mantissa bits are set to 0. Can be either positive or negative depending on the sign bit.
- Infinity — All exponent bits are set to 1, and all mantissa bits are set to 0. Can be either positive or negative depending on the sign bit.

- NaN (Not a Number) — All exponent bits are set to 1, and at least one mantissa bit is set to 1. The sign bit is not taken into consideration.

3 Development Plan

3.1 Development Overview

It is proposed that by each project meeting a different subsection of the Floating Point ALU will be completed, to be tested at the laboratory, and any irregularities in behaviour to be elliminated. By the end of the semester, the whole project must be completed and must be presented by the last week.

3.2 Project Lab 1

The project overview and development plan will be finalised and presented.

3.3 Project Lab 2

The supporting environment for the project must be implemented and tested on the FPGA. It must be capable of displaying numbers stored in a Read-Only Memory using a Seven-Segment Display, and must be capable of switching between multiple display modes using switches or buttons as input.

3.4 Project Lab 3

The floating point addition subsection of the ALU must be implemented as a separate circuit and must be fully functional.

3.5 Project Lab 4

The floating point mutliplication subsection of the ALU must be implemented as a separate circuit and must be fully functional.

3.6 Project Lab 5

Any remaining part of the project will be implemented, and the board's functionality will be fully tested. Any errors in the circuit's operations will be removed. The documentation will be updated as well, should the need arise.

4 Floating Point ALU Structure

The Operation of the Floating Point ALU is composed of three steps:

1. Check the validity of the inputs and the operation to be performed, and either proceed to the next step, or write the appropriate value and end the algorithm.
2. Perform the operation, with all required steps.
3. Realign the number to the IEEE 754 standard, and check for exponent overflow or underflow.

In order to implement the Floating Point ALU, we will use Registers, Comparators, a smaller ALU capable of integer operations, a Wallace Tree Multiplier, and the Control Unit, implemented as a Finite State Machine.
We can speed up the circuit by utilizing multiple comparators. For this implementation, we will use two comparators.

4.1 Algorithm

4.1.1 Floating Point Adder

The Floating point Adder takes as inputs the two numbers, A and B, which are divided into their respective sign, exponent, and mantissa (with a 1 implicitly placed on the 24th bit.
First, we compare the exponents, and the mantissa of the number with the smaller exponent is shifted to the right and its exponent is incremented until the exponents are equal. This way, the numbers are aligned to the same exponent.
Next, we perform the addition of the mantissa. If the signs are equal, we simply add the two mantissas together, otherwise, we subtract the smaller mantissa from the larger one, and use the sign of the number with the larger mantissa for the result.
Finally, we realign the resulting number to the IEEE-754 standard. If the mantissa has become too large, and there is a '1' on the 25th bit, we shift the result right and increment the exponent. If the mantissa is too small, we shift left and decrement the exponent until there is a '1' on exactly the 24th bit.

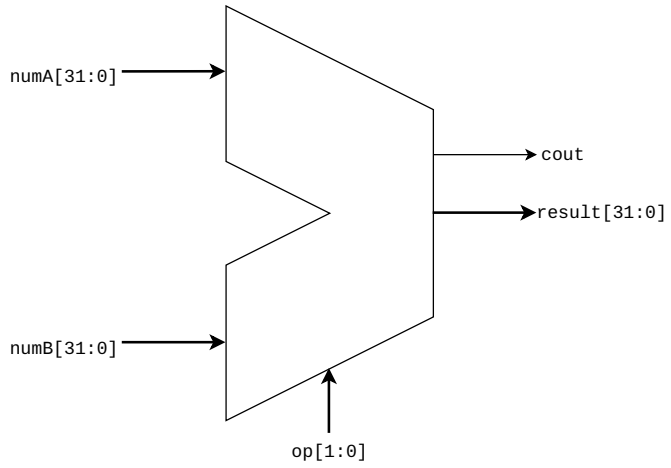
4.1.2 Floating Point Multiplier

The Floating Point Multiplier takes as inputs the two numbers, A and B, which are divided into their respective sign, exponent, and mantissa (with a 1 implicitly placed on the 24th bit.
First, we add the two exponents together and subtract 127 from the resulting sum. This is because the exponents are implicitly stored as (127 + exponent).
Next, we perform the multiplication on the mantissas. We can use a Wallace Tree Multiplier to perform the multiplication in a single clock cycle, which simplifies the circuit significantly.
Since we use a standard 32 bit Wallace Tree Multiplier and the mantissa has only 24 bits, we extend with zero to the right. The two mantissas will always start with a 1 on the MSB, and by multiplying them, the resulting number will also have a 1 on the 64th bit. We simply select the next most significatn 23 bits for the result mantissa, with no need to shift and align the number.

4.2 Design

4.2.1 ALU

Figure 1: Simple Integer ALU



The ALU takes as input the two numbers, and a selection for which operation to be performed. Since it performs elementary operations, this is a combinational logic circuit, and requires no clock.
The ouputs are the result of the operation, and an additional carry out bit, marked as *cout*.
The ALU is capable of performing four operations:

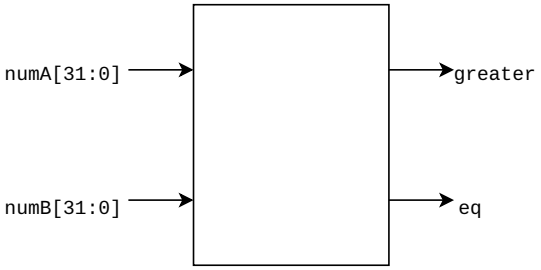
- Addition: *result = numA + numB*
- Subtraction: *result = numA - numB*
- Increment *result = numA + 1*
- Decrement *result = numA - 1*

In case of any overflow or underflow, the *cout* bit is set to 1, otherwise it remains 0.

4.2.2 Comparator

The comparator takes as input two numbers, and the outputs *eq* and *greater* are set to 1 if $numA > numB$ and if $numA = numB$ respectively.

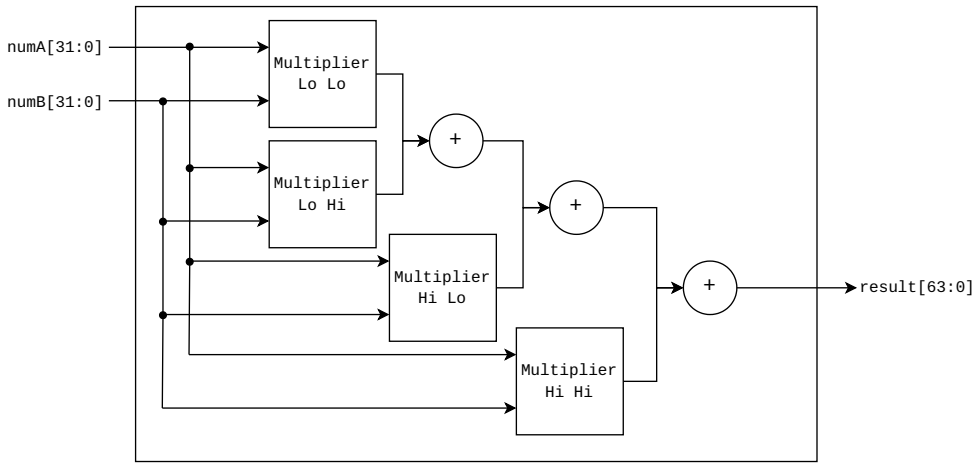
Figure 2: Integer Comparator



4.2.3 Multiplier

The mantissa multiplier was implemented as a wallace tree, capable of performing 32 bit multiplication. The result is on 64 bits. The desing was created recursively, starting from the 4 bit wallace tree multiplier. Each number is split into *Low* and $High$, and then multiplied, shifted, and added accordingly.

Figure 3: Structure of Wallace Tree Multiplier



The entity for the Components used is available here.

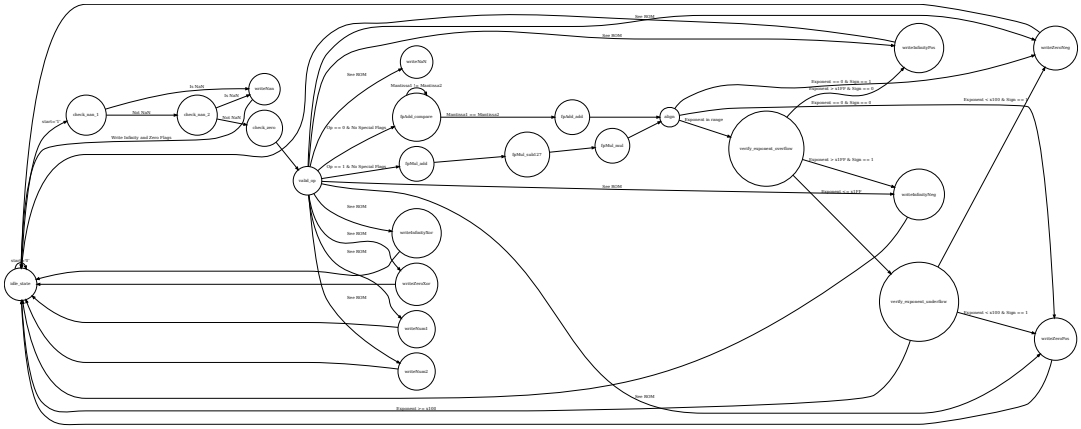
4.2.4 Control Unit

The Control Unit is the most complex part of the floating point adder. It is a finite state machine which controls all the select signals for the various components of the design. It switches states synchronously, and based on the state and

results from differing components, sets the select signals for the multiplexers, the write signals for the registers, and the operation for the ALU.
The Control Unit is a finite state machine with 22 states, divided into 6 groups:

- Idle: *idle_state*
  Machine is idle and not busy performing any operation.
- Check Input: *check_nan_1*, *check_nan_2*, *check_zero*, *valid_op*
  Check the validity of the inputs and either proceed with the algorithm or write the result directly.
- Perform FP Addition: $fpAdd\_compare$, $fpAdd\_add$
  Shift mantissa and increment exponents until they are equal, then add the mantissas.
- Perform FP Multiplication: $fpMul\_add$, $fpMul\_sub127$, $fpMul_mul$
  Add the two exponents together, subtract 127 from the result, and multiply the mantissas.
- Realign to Standard: $align$, $verify\_exponent\_overflow$, $verify\_exponent\_underflow$
  Realign mantissa to standard and check for overflow or underflow in the exponent.
- Write Special: $writeNan$, $writeInfinityXor$, $writeInfinityPos$, $writeInfinityNeg$, $writeZeroXor$, $writeZeroPos$, $writeZeroNeg$, $writeNum1$, $writeNum2$
  Used when operation is invalid.

Figure 4: The Control Unit transition diagram

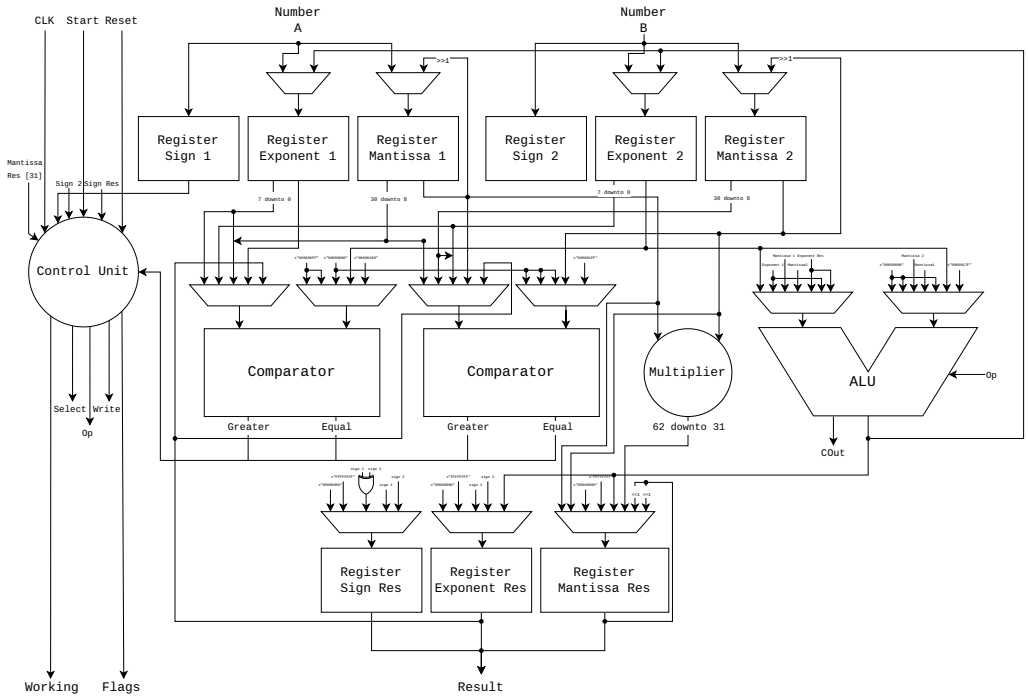


The entity for the Control Unit is available here.

4.2.5 Top Level Design

Following the design specifications, all the components were connected. A separate register was used for each part of the inputs and outputs, and two comparators were used to speed up the process. Multiplexers were used to switch between the inputs to the different components. Since there is more space in the register than is needed to store the mantissa, it is stored left aligned, and padded with 0. Only the most significant bits from the mantissa register and multiplier result are used, ensuring there is never a mantissa overflow.

Figure 5: The top level design



4.3 Implementation

The code for the Comparator, ALU, Control Unit and Registers was written in a behavioural stlye. The code for the Multiplier, and Top Level was written in a structural stlye.
For ease of implementation, separate register components were written with a $4:1$, and an $8:1$ multiplexer included.
The Top Level Design is mostly composed of port maps, with a couple of multiplexers for the ALU and Comparators.
The Control Unit is a Finite State Machine composed of two processes:

- The first porcess calculates the next state of the machine. The switching between states is done synchronously. When checking the inputs, internal flags are set to 1 or 0, These flags and the operation input then form an address in a ROM which gives the next state.
- The second process represents a multitude of multiplexers and simple logic gates which caluclate the output signals based on the current state and the inputs given. This process is asynchronous.

The code for the ROM is available here.

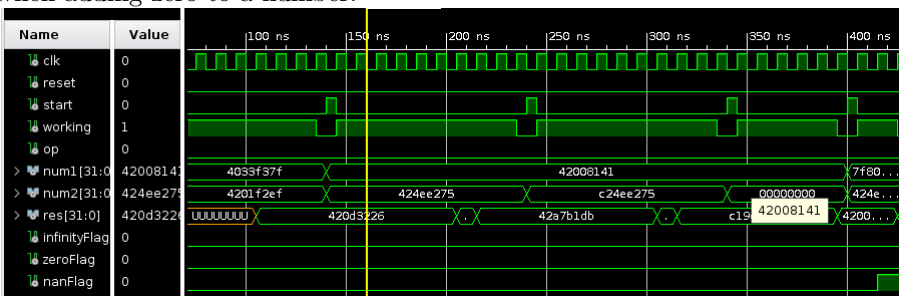
5 Testing

Testbenches were written for the Addition Operation and the Multiplication Operation. Multiple combinations of inputs were used, and the machines ability to detect wrong inputs was also tested.
The inputs are captured immediately as the start value is read as 1 on a rising edge of the clock. A change in inputs afterwards will not affect the performance. While the machine is busy, the working bit is set to 1, otherwise, while it is idle it remains 0. Once the algorithm is finished and the working bit is set to 0, the output will remain stable. If the machine is started again, the output will still remain stable for at least *one* clock cycle. Afterwards, it cannot be guaranteed what will appear on the output while the working flag is active.

5.1 Addition

5.1.1 Regular Behaviour

The FP ALU performs as expected under regular circumstances. It is performs as expected when adding positive numbers, when adding negative numbers, and when adding zero to a number.

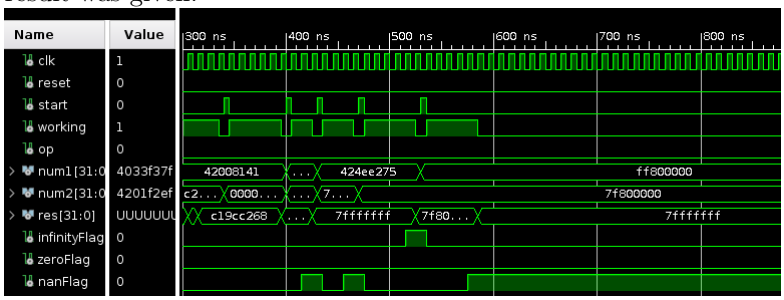


5.1.2 Special Cases

Based on the ROM in the Control Unit, the special cases for addition are as follows:

- $(+\infty) + (+\infty) = +\infty$
- $(+\infty) + (-\infty) = NaN$
- $(+\infty) + (\pm 0) = +\infty$
- $(\pm\infty) + x = \pm\infty$
- $(-\infty) + (-\infty) = -\infty$
- $(-\infty) + (\pm 0) = -\infty$
- $(\pm 0) + (\pm 0) = +0$
- $(\pm 0) + (\mp 0) = +0$
- $(\pm 0) + x = +0$

Also including their reciprocals. These special cases were tested, and the correct result was given.

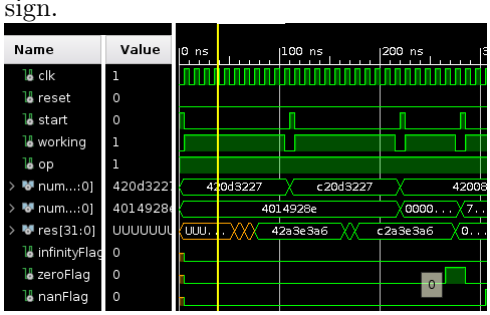


The code for the Add Testebench is available here.

5.2 Multiplication

5.2.1 Regular Beheviour

Under regular circumstances the multiplication is performed correctly. When multiplying numbers with the same sign, the resulting sign is positive, and when multiplying with 0, the result will also be 0, with the same logic applied for the sign.

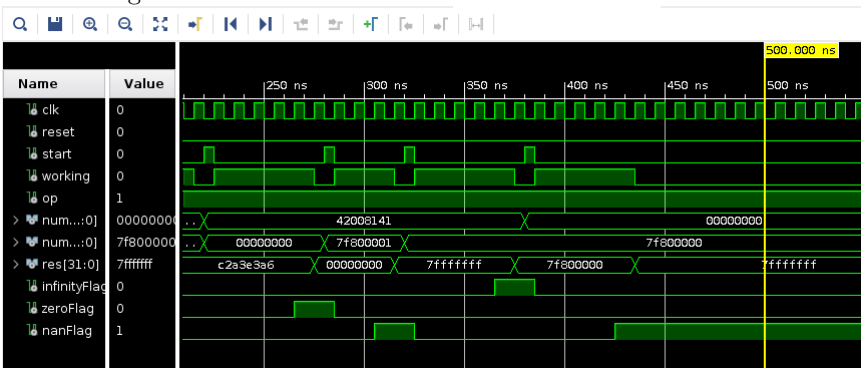


5.2.2 Special Cases

Based on the ROM in the Control Unit, the special cases for addition are as follows:

- $(\pm\infty) * (\pm\infty) = +\infty$
- $(\pm\infty) * (\mp\infty) = -\infty$
- $(\pm\infty) * (\pm 0) = NaN$
- $(\pm\infty) * (\mp 0) = NaN$
- $(\pm\infty) * x = (XOR)\infty$
- $(-\infty) + (\pm 0) = -\infty$
- $(\pm 0) * (\pm 0) = +0$
- $(\pm 0) * (\mp 0) = -0$
- $(\pm 0) * x = (XOR)0$

Also including their reciprocals. These special cases were tested, and the correct result was given.



The code for the Mul Testebench is available here.

6 Code

6.1 ALU

```
    entity alu32bit is
    Port (
            numA : in STD_LOGIC_VECTOR (31 downto 0);
            numB : in STD_LOGIC_VECTOR (31 downto 0);
            op : in STD_LOGIC_vector(1 downto 0);
            output : out STD_LOGIC_VECTOR (31 downto 0);
            cout: out std_logic);
    end alu32bit;
```

6.2 Comparator

```
    entity comparator32bit is
    Port ( numA : in STD_LOGIC_vector(31 downto 0);
           numB : in STD_LOGIC_vector(31 downto 0);
           eq : out STD_LOGIC;
           greater : out STD_LOGIC);
    end comparator32bit;
```

6.3 Register with multiplexer

```
     entity register32bitW4x1Mux is
    Port ( A : in STD_LOGIC_VECTOR (31 downto 0);
           B : in STD_LOGIC_VECTOR (31 downto 0);
           C : in STD_LOGIC_VECTOR (31 downto 0);
           D : in STD_LOGIC_VECTOR (31 downto 0);
           S: in std_logic_vector(1 downto 0);
           write : in STD_LOGIC;
           clk : in STD_LOGIC;
           output : out STD_LOGIC_VECTOR (31 downto 0));
    end register32bitW4x1Mux;
```

6.4 Wallace Tree Multiplier

```
    entity multiplier32bit is
    Port ( numA : in STD_LOGIC_VECTOR (31 downto 0);
           numB : in STD_LOGIC_VECTOR (31 downto 0);
           result : out STD_LOGIC_VECTOR (63 downto 0));
    end multiplier32bit;
```

6.5 Control Unit

```
entity controlUnit is
Port (
  clk : in STD_LOGIC;
  reset: in std_logic;
  start:in std_logic;
  op:in std_logic;

  writeSign1:out std_logic;
  writeSign2:out std_logic;
  writeSignRes:out std_logic;

  writeExponent1:out std_logic;
  writeExponent2:out std_logic;
  writeExponentRes:out std_logic;

  writeMantissa1:out std_logic;
  writeMantissa2:out std_logic;
  writeMantissaRes:out std_logic;

  selectSign1:out std_logic_vector(1 downto 0);
  selectExponent1:out std_logic_vector(1 downto 0);
  selectMantissa1:out std_logic_vector(1 downto 0);

  selectSign2:out std_logic_vector(1 downto 0);
  selectExponent2:out std_logic_vector(1 downto 0);
  selectMantissa2:out std_logic_vector(1 downto 0);

  selectSignRes:out std_logic_vector(2 downto 0);
  selectExponentRes:out std_logic_vector(2 downto 0);
  selectMantissaRes:out std_logic_vector(2 downto 0);

  outSign1:in std_logic;
  outSign2:in std_logic;
  outSignRes:in std_logic;
  outMantissaRes31:in std_logic;

  comparator1Select:out std_logic_vector(2 downto 0);
  comparator1Eq:in std_logic;
  comparator1Greater:in std_logic;

  comparator2Select:out std_logic_vector(2 downto 0);
  comparator2Eq:in std_logic;
  comparator2Greater:in std_logic;

  alu32bitOp:out std_logic_vector(1 downto 0);
  alu32bitSelect:out std_logic_vector(2 downto 0);
  alu32bitCout: in std_logic;

  infinityFlag:out std_logic;
  zeroFlag:out std_logic;
  nanFlag:out std_logic;

  working:out std_logic
);
end controlUnit;
```

6.6 ALU Top level

```
    entity aluFloat is
    Port (
        clk : in STD_LOGIC;
        start:in std_logic;
        reset:in std_logic;
        op : in STD_LOGIC;

        num1 : in STD_LOGIC_VECTOR (31 downto 0);
        num2 : in STD_LOGIC_VECTOR (31 downto 0);

        res : out STD_LOGIC_VECTOR (31 downto 0);
        working:out std_logic;
        infinityFlag:out std_logic;
        zeroFlag:out std_logic;
        nanFlag:out std_logic
    );
    end aluFloat;
```

6.7 ROM

```
    case internalFlags is
--      internalFlags<=op & infinityPos1 & infinityNeg1 &
    infinityPos2 & infinityNeg2 &
--              zeroPos1 & zeroNeg1 & zeroPos2 & zeroNeg2;
            when "000000000"=>
                state<=fpAdd_compare;

            -- +infinity + x
            when b"0_10_10_00_00"=> -- +inf + +inf
                state<=writeInfinityPos;
            when b"0_10_01_00_00"=> -- +inf + -inf
                state<=writeNan;
            when b"0_10_00_00_10"=> -- +inf + +0
                state<=writeInfinityPos;
            when b"0_10_00_00_01"=> -- +inf + -0
                state<=writeInfinityPos;
            when b"0_10_00_00_00"=> -- +inf + x
                state<=writeInfinityPos;

            -- -infinity + x
            when b"0_01_10_00_00"=> -- -inf + +inf
                state<=writeNan;
            when b"0_01_01_00_00"=> -- -inf + -inf
                state<=writeInfinityNeg;
            when b"0_01_00_00_10"=> -- -inf + +0
                state<=writeInfinityNeg;
            when b"0_01_00_00_01"=> -- -inf + -0
                state<=writeInfinityNeg;
            when b"0_01_00_00_00"=> -- -inf + x
                state<=writeInfinityNeg;

            -- +0 + x
            when b"0_00_10_10_00"=> -- +0 + +inf
                state<=writeInfinityPos;
            when b"0_00_01_10_00"=> -- +0 + -inf
                state<=writeInfinityNeg;
            when b"0_00_00_10_10"=> -- +0 + +0
                state<=writeZeroPos;
```

```
38              when b"0_00_00_10_01"=> -- +0 + -0
39                  state<=writeZeroPos;
40              when b"0_00_00_10_00"=> -- +0 + x
41                  state<=writeNum2;
42
43              -- -0 + x
44              when b"0_00_10_01_00"=> -- -0 + +inf
45                  state<=writeInfinityPos;
46              when b"0_00_01_01_00"=> -- -0 + -inf
47                  state<=writeInfinityNeg;
48              when b"0_00_00_01_10"=> -- -0 + +0
49                  state<=writeZeroPos;
50              when b"0_00_00_01_01"=> -- -0 + -0
51                  state<=writeZeroNeg;
52              when b"0_00_00_01_00"=> -- -0 + x
53                  state<=writeNum2;
54
55              -- x + special
56              when b"0_00_10_00_00"=> -- x + +inf
57                  state<=writeInfinityPos;
58              when b"0_00_01_00_00"=> -- x + -inf
59                  state<=writeInfinityNeg;
60              when b"0_00_00_00_10"=> -- x + +0
61                  state<=writeNum1;
62              when b"0_00_00_00_01"=> -- x + -0
63                  state<=writeNum1;
64
65  --      internalFlags<=op & infinityPos1 & infinityNeg1 &
        infinityPos2 & infinityNeg2 &
66  --              zeroPos1 & zeroNeg1 & zeroPos2 & zeroNeg2;
67              when "100000000"=>
68                  state<=fpMul_add;
69
70              -- +infinity * x
71              when b"1_10_10_00_00"=> -- +inf * +inf
72                  state<=writeInfinityPos;
73              when b"1_10_01_00_00"=> -- +inf * -inf
74                  state<=writeInfinityNeg;
75              when b"1_10_00_00_10"=> -- +inf * +0
76                  state<=writeNan;
77              when b"1_10_00_00_01"=> -- +inf * -0
78                  state<=writeNan;
79              when b"1_10_00_00_00"=> -- +inf * x
80                  state<=writeInfinityXor;
81
82              -- -infinity * x
83              when b"1_01_10_00_00"=> -- -inf * +inf
84                  state<=writeInfinityNeg;
85              when b"1_01_01_00_00"=> -- -inf * -inf
86                  state<=writeInfinityPos;
87              when b"1_01_00_00_10"=> -- -inf * +0
88                  state<=writeNan;
89              when b"1_01_00_00_01"=> -- -inf * -0
90                  state<=writeNan;
91              when b"1_01_00_00_00"=> -- -inf * x
92                  state<=writeInfinityXor;
93
94              -- +0 * x
95              when b"1_00_10_10_00"=> -- +0 * +inf
96                  state<=writeNan;
97              when b"1_00_01_10_00"=> -- +0 * -inf
98                  state<=writeNan;
```

```
99              when b"1_00_00_10_10"=> -- +0 * +0
100                 state<=writeZeroPos;
101             when b"1_00_00_10_01"=> -- +0 * -0
102                 state<=writeZeroNeg;
103             when b"1_00_00_10_00"=> -- +0 * x
104                 state<=writeZeroXor;
105
106             -- -0 * x
107             when b"1_00_10_01_00"=> -- -0 * +inf
108                 state<=writeNan;
109             when b"1_00_01_01_00"=> -- -0 * -inf
110                 state<=writeNan;
111             when b"1_00_00_01_10"=> -- -0 * +0
112                 state<=writeZeroNeg;
113             when b"1_00_00_01_01"=> -- -0 * -0
114                 state<=writeZeroPos;
115             when b"1_00_00_01_00"=> -- -0 * x
116                 state<=writeZeroXor;
117
118             -- x * special
119             when b"1_00_10_00_00"=> -- x * +inf
120                 state<=writeInfinityXor;
121             when b"1_00_01_00_00"=> -- x * -inf
122                 state<=writeInfinityXor;
123             when b"1_00_00_00_10"=> -- x * +0
124                 state<=writeZeroXor;
125             when b"1_00_00_00_01"=> -- x * -0
126                 state<=writeZeroXor;
127
128             when others=>
129                 state<=writeNan;
130         end case;
```

6.8 Testbenches

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

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

architecture Behavioral of tbAddFloat is

component aluFloat is
Port (
    clk : in STD_LOGIC;
    start:in std_logic;
    reset:in std_logic;
    op : in STD_LOGIC;

    num1 : in STD_LOGIC_VECTOR (31 downto 0);
    num2 : in STD_LOGIC_VECTOR (31 downto 0);

    res : out STD_LOGIC_VECTOR (31 downto 0);
    working:out std_logic;
    infinityFlag:out std_logic;
    zeroFlag:out std_logic;
    nanFlag:out std_logic
);
end component;

signal clk,reset,start,working,op:std_logic:='0';
signal num1,num2,res:std_logic_vector(31 downto 0);

signal infinityFlag,zeroFlag,nanFlag:std_logic:='0';

function to_string ( a: std_logic_vector) return string is
variable b : string (1 to a'length) := (others => NUL);
variable stri : integer := 1;
begin
    for i in a'range loop
        b(stri) := std_logic'image(a((i)))(2);
    stri := stri+1;
    end loop;
return b;
end function;


begin

clk<=not clk after 5ns;

tb:aluFloat port map(
    clk=>clk,
    reset=>reset,
    start=>start,
    op=>op,

    num1=>num1,
    num2=>num2,
    res=>res,
    working=>working,

    infinityFlag=>infinityFlag,
```

```
61      zeroFlag=>zeroFlag,
62      nanFlag=>nanFlag
63  );
64
65  process
66  begin
67
68      num1<=b"0_10000000_01100111111001101111111"; --
            2.811736822182958984375
69      num2<=b"0_10000100_00000011111001011101111"; --
            32.487239837646484375
70      start<='1';
71
72      --expecting 35.29897665977478027344
73      -- 420d3226
74      wait on clk;
75      start<='0';
76      wait until working='0';
77
78      report to_string(res);
79      wait on clk;
80      --============================================================
81      num1<=b"01000010000000001000000101000001"; -- 32.126225
82      num2<=b"01000010010011101110001001110101"; -- 51.721149
83      start<='1';
84
85      -- expecting 83.847374
86      -- 42a7b1db
87      wait on clk;
88      start<='0';
89      wait until working='0';
90      report to_string(res);
91      wait on clk;
92
93      --============================================================
94      num1<=b"01000010000000001000000101000001"; -- 32.126225
95      num2<=b"11000010010011101110001001110101"; -- -51.721149
96      start<='1';
97
98      -- expecting -19.594924
99      -- c19cc268
100     wait on clk;
101     start<='0';
102     wait until working='0';
103     report to_string(res);
104     wait on clk;
105
106     --============================================================
107     num1<=b"01000010000000001000000101000001"; --32.126225
108     num2<=b"00000000000000000000000000000000"; --0
109     start<='1';
110
111     -- expecting 32.126225
112     -- 42008141
113     wait on clk;
114     start<='0';
115     wait until working='0';
116     report to_string(res);
117     wait on clk;
118
119
120
```

```
121     --=========================================================
122     num1<=b"0_11111111_00000000000000000000001"; --NaN
123     num2<=b"01000010010011101110001001110101"; --51.721149
124     start<='1';
125 
126     -- expecting NaN
127     -- 7fffffff
128     wait on clk;
129     start<='0';
130     wait until working='0';
131     report to_string(res);
132     wait on clk;
133     --=========================================================
134     num1<=b"01000010010011101110001001110101"; --51.721149
135     num2<=b"0_11111111_00000000000000000000001"; --NaN
136     start<='1';
137 
138     -- expecting NaN
139     -- 7fffffff
140     wait on clk;
141     start<='0';
142     wait until working='0';
143     report to_string(res);
144     wait on clk;
145     --=========================================================
146     num1<=b"01000010010011101110001001110101"; --51.721149
147     num2<=b"0_11111111_00000000000000000000000"; -- +inf
148     start<='1';
149 
150     -- expecting +inf
151     -- 7f800000
152     wait on clk;
153     start<='0';
154     wait until working='0';
155     report to_string(res);
156     wait on clk;
157 
158     --=========================================================
159     num1<=b"1_11111111_00000000000000000000000"; -- -inf
160     num2<=b"0_11111111_00000000000000000000000"; -- +inf
161     start<='1';
162 
163     -- expecting +NaN
164     -- 7fffffff
165     wait on clk;
166     start<='0';
167     wait until working='0';
168     report to_string(res);
169     wait on clk;
170 
171     wait for 1000ns;
172 end process;
173 
174 end Behavioral;
```

Listing 1: Add Testbench

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

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

architecture Behavioral of tbMulFloat is

component aluFloat is
Port (
    clk : in STD_LOGIC;
    start:in std_logic;
    reset:in std_logic;
    op : in STD_LOGIC;

    num1 : in STD_LOGIC_VECTOR (31 downto 0);
    num2 : in STD_LOGIC_VECTOR (31 downto 0);

    res : out STD_LOGIC_VECTOR (31 downto 0);
    working:out std_logic;
    infinityFlag:out std_logic;
    zeroFlag:out std_logic;
    nanFlag:out std_logic
);
end component;

signal clk,reset,start,working,op:std_logic:='0';
signal num1,num2,res:std_logic_vector(31 downto 0);

signal infinityFlag,zeroFlag,nanFlag:std_logic:='0';

function to_string ( a: std_logic_vector) return string is
variable b : string (1 to a'length) := (others => NUL);
variable stri : integer := 1;
begin
    for i in a'range loop
        b(stri) := std_logic'image(a((i)))(2);
    stri := stri+1;
    end loop;
return b;
end function;


begin

clk<=not clk after 5ns;

tb:aluFloat port map(
    clk=>clk,
    reset=>reset,
    start=>start,
    op=>op,

    num1=>num1,
    num2=>num2,
    res=>res,
    working=>working,

    infinityFlag=>infinityFlag,
    zeroFlag=>zeroFlag,
    nanFlag=>nanFlag
```

```
63  );
64
65  process
66  begin
67      op<='1';
68      start<='1';
69      num1<="01000010000011010011001000100111"; --
            35.298976898193359375
70      num2<="01000000000101001001001010001110"; --
            2.321444988250732421875
71
72      --expecting 81.94463245721374278219
73      --42a3e3a6
74      wait on clk;
75      start<='0';
76      wait until working='0';
77
78      report to_string(res);
79      wait on clk;
80      --============================================================
81
82      op<='1';
83      start<='1';
84      num1<="11000010000011010011001000100111"; --
            -35.298976898193359375
85      num2<="01000000000101001001001010001110"; --
            2.321444988250732421875
86
87      --expecting -81.94463245721374278219
88      --c2a3e3a6
89      wait on clk;
90      start<='0';
91      wait until working='0';
92
93      report to_string(res);
94      wait on clk;
95      --============================================================
96      num1<=b"01000010000000001000000101000001"; --32.126225
97      num2<=b"00000000000000000000000000000000"; --0
98      start<='1';
99
100     -- expecting 0
101     -- 00000000
102     wait on clk;
103     start<='0';
104     wait until working='0';
105     report to_string(res);
106     wait on clk;
107
108     --============================================================
109
110     num1<=b"01000010000000001000000101000001"; --32.126225
111     num2<=b"0_11111111_00000000000000000000001"; --Nan
112     start<='1';
113
114     -- expecting NaN
115     -- 7fffffff
116     wait on clk;
117     start<='0';
118     wait until working='0';
119     report to_string(res);
120     wait on clk;
```

```
121
122     --=========================================================
123
124     num1<=b"01000010000000001000000101000001"; --32.126225
125     num2<=b"0_11111111_00000000000000000000000"; --+inf
126     start<='1';
127
128     -- expecting +inf
129     -- 7f800000
130     wait on clk;
131     start<='0';
132     wait until working='0';
133     report to_string(res);
134     wait on clk;
135
136     --=========================================================
137
138     num1<=b"00000000000000000000000000000000"; --0
139     num2<=b"0_11111111_00000000000000000000000"; --+inf
140     start<='1';
141
142     -- expecting NaN
143     -- 7fffffff
144     wait on clk;
145     start<='0';
146     wait until working='0';
147     report to_string(res);
148     wait on clk;
149
150     --=========================================================
151     wait for 1000ns;
152
153 end process;
154 end Behavioral;
```

Listing 2: Mul Testbench

7 Bibliography

References

[1] Steven Petryk, Adding IEEE-754 Floating Point Numbers, Oct 2015

[2] Wikipedia, IEEE 754, Oct 2024

[3] Jan Misali, how floating point works, May 2022