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

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

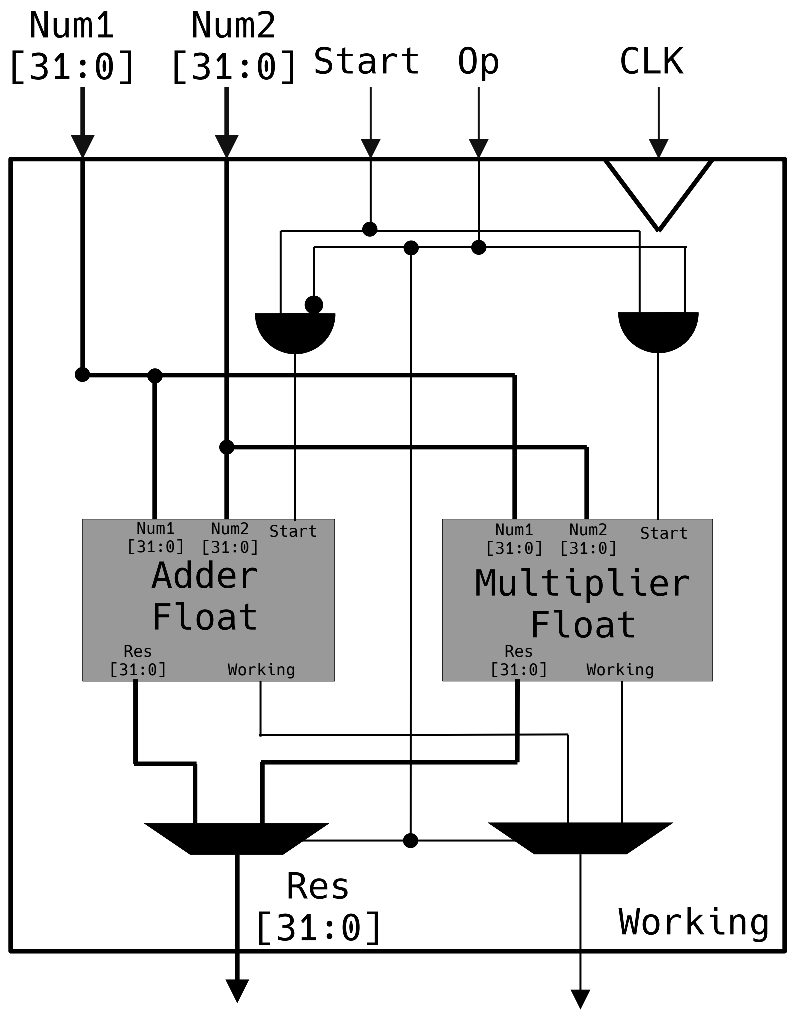
The floating point mutliplication subsection of the ALU must be implemented as a separate circuit and must be fully functional.

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 FP ALU is composed of the three parts - the adder, the multiplier, and a multiplexer to change between them. The inputs are the two 32 bit IEEE-754 numbers, the operation to be performed, and a start bit to start the addition/-multiplication.
The outputs are the resulting number, and a bit to signal when the ALU is working and when the operation is over.



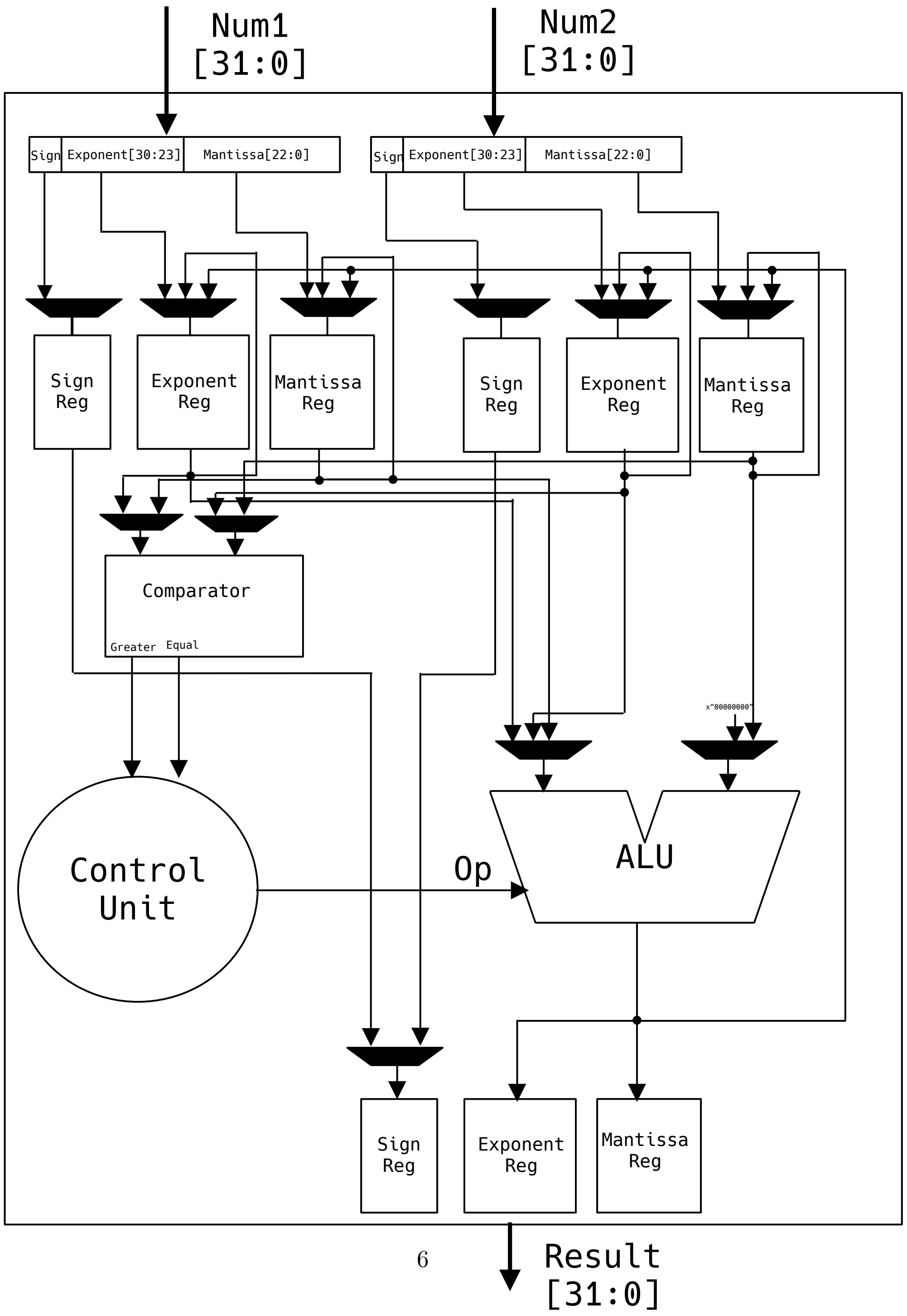
5 Floating Point Adder

5.1 Algorithm

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.
We also check for if the mantissa is equal to 0 and check for overflow and underflow of the exponent. If the mantissa is 0 or if we have an underflow of the exponent, we make the exponent and mantissa 0, so that there are only two values for the number 0. If there is an overflow of the exponent, we set the exponent to the maximum possible value and the mantissa to 0, for a standard infinity value.

5.2 Design

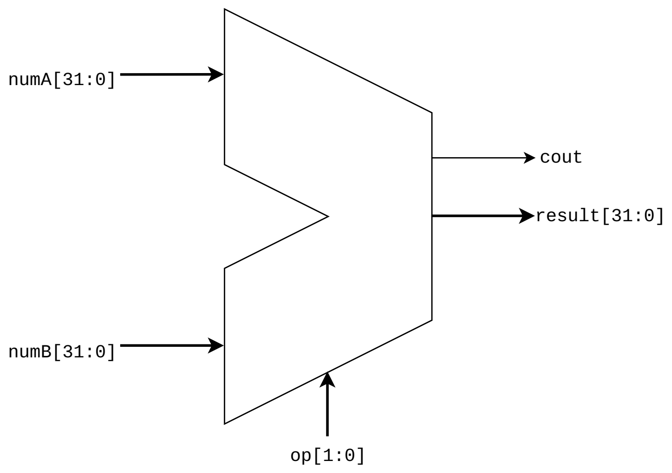
In order to implement the Floating Point Adder, we will use a 32 bit ALU, capable of addition, subtraction, incrementing, and decrementing. A comparator will also be needed which which can tell the greater number or whether they are equal. We will also use registers to store intermediate results, and a control unit to switch between the inputs and operations of the ALU.



5.3 Implementation

The top level code was written in a structural style, while the simpler components were written behaviourally. Every component was implement separately to match the specifications mentioned above.

5.3.1 ALU



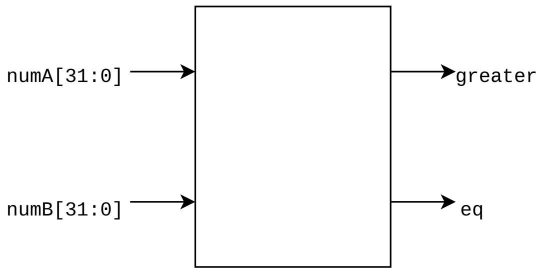
The ALU takes as input the two numbers, and a selection for which operation to be performed. Since it performs elementary operations, this is a combinational logic circuit, and requires no clock.
The ouputs are the result of the operation, and an additional carry out bit, marked as *cout*.
The ALU is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

5.3.2 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.



5.3.3 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 five states:

- idle_state:
  Machine is idle and not busy performing any operation.
- compare_state:
  Compare the two exponents and shift the mantissa until they are equal.
- add_state:
  Add the two mantissas together.
- align_state:
  Realign the resulting number to match the IEEE-754 standard.
- verify_overflow_state:
  Check for overflow and write the corresponding result.
- verify_underflow_state:
  Check for underflow and write the corresponding result.

The entity for the floating point adder, and the components used is available here.

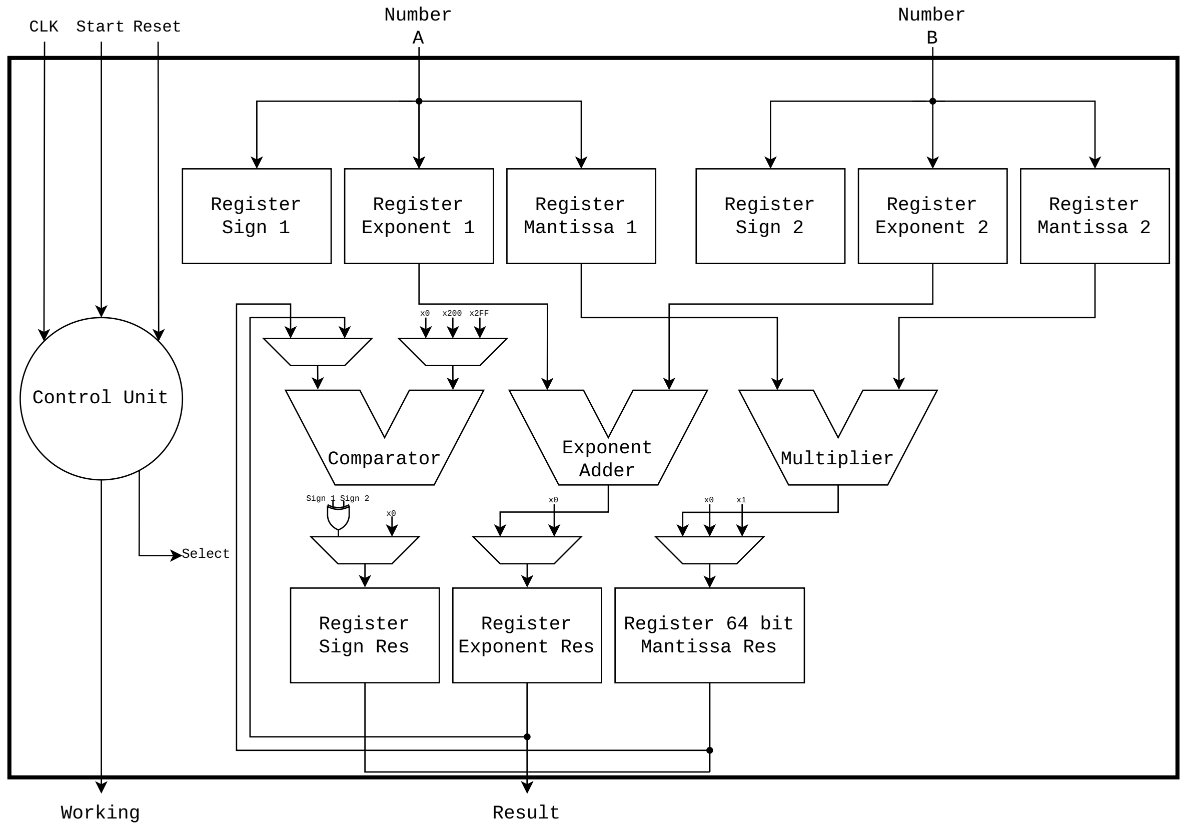
6 Floating Point Multiplier

6.1 Algorithm

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.
Simmilarly to the adder, we apply the same standardisation steps for 0 or infinity. If the mantissa is 0, or there is an exponent underflow, we set all the exponent and mantissa bits to 0. If there is an exponent overflow, we set all exponent bits to 1, and all mantissa bits to 0.
By using the steps described above, we can perform floating point multiplication in at most 5 clock cycles.

6.2 Design

In order to implement the Floating Point Multiplier, we will use a dedicated exponent adder to perform the exponent addition and subtracting 127, a 32 bit Wallace Tree Multiplier, and a Comparator. We also use registers to store intermediate values, and a control unit to give signals throughout the circuit.



6.3 Implementation

Simmilarly to the floating point adder, the top level was also written in a structural style, with the smaller components being written behaviourally.

6.3.1 Comparator

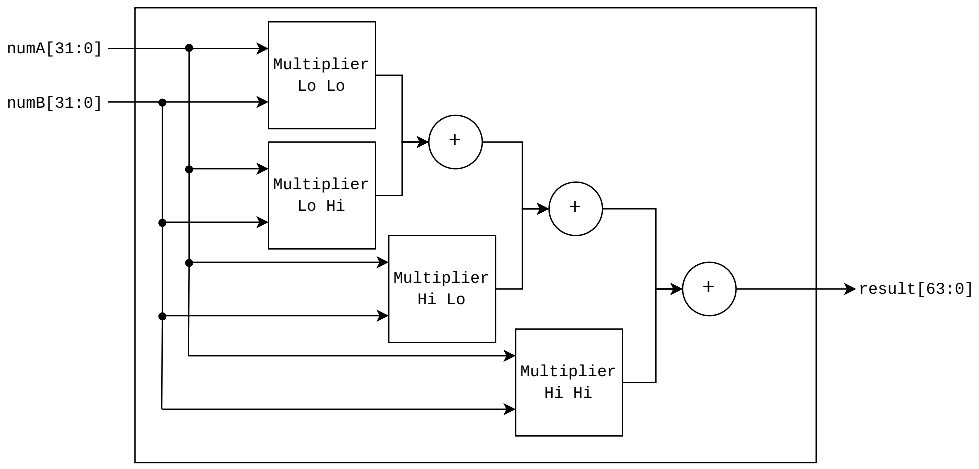
The comparator behaves the same as the one used for the adder, except it is capable of handling 64 bit numbers. It is used for checking the result for overflow or underflow in the final states of the process.

6.3.2 Exponent Adder

The exponent adder is a simple combinational logic circuit. It takes two 32 bit numbers, $exponent1$ and $exponent2$, and performs the operation
$result <= exponent1 + exponent2 - 127$.

6.3.3 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.



6.3.4 Control Unit

The Control Unit is similar to that of the floating point adder, though somewhat simpler. 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 just the select signals for the multiplexers and the write signals for the registers, with there being no operation signals.
As oposed to the adder, which requires shifting the result an unknown amount of bits, the floating point multiplier will always complete the operation in at most 6 clock cycles.
The Control Unit is a finite state machine with five states:

- idle_state:
  Machine is idle and not busy performing any operation.
- add_state:
  The exponent addition is performed
  ($exponentRes <= exponent1 + exponent2 - 127$).
- multiply_state:
  The two mantissas are multiplied together.
- align_state:
  Realign the resulting number to match the IEEE-754 standard.
- verify_overflow_state:
  Check for overflow and write the corresponding result.
- verify_underflow_state:
  Check for underflow and write the corresponding result.

The entity for the floating point multiplier, and the components used is available here.

7 Code

7.1 Floating Point Adder

7.1.1 Top level

```vhdl
entity adderFloat is
Port ( clk : in STD_LOGIC;
       reset:in STD_LOGIC;
       start: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);
       dbState:out std_logic_vector(3 downto 0);
       working : out STD_LOGIC);
end adderFloat;
```

7.1.2 ALU

```vhdl
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;
```

7.1.3 Comparator

```vhdl
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;
```

7.1.4 Control Unit

```vhdl
entity fpAdderControlUnit is
Port (
    clk:in std_logic;
    reset:in std_logic;

    comparatorEq:in std_logic;
    comparatorGreater:in std_logic;

    outSign1_0:in std_logic;
    outSign2_0:in std_logic;
    outMantissaRes24:in std_logic;
    outMantissaRes23:in std_logic;

    start:in std_logic;

    writeSign1:out std_logic;
```

```vhdl
        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(1 downto 0);
        selectExponentRes:out std_logic_vector(1 downto 0);
        selectMantissaRes:out std_logic_vector(1 downto 0);

        dbState:out std_logic_vector(3 downto 0);
        working:out std_logic;

        comparatorSelect:out std_logic_vector(2 downto 0);
        alu32bitSelect:out std_logic_vector(2 downto 0);
        alu32bitOp:out std_logic_vector(1 downto 0)
    );
    end fpAdderControlUnit;
```

7.1.5 Register with multiplexer

```vhdl
     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;
```

7.2 Floating Point Multiplier

7.2.1 Top level

```
    entity multiplierFloat is
    Port ( clk : in STD_LOGIC;
           reset:in STD_LOGIC;
           start: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);
           dbState:out std_logic_vector(3 downto 0);
           working : out STD_LOGIC);
    end multiplierFloat;
```

7.2.2 Exponent Adder

```
    entity exponentAdder is
    Port ( exponent1 : in STD_LOGIC_VECTOR (31 downto 0);
           exponent2 : in STD_LOGIC_VECTOR (31 downto 0);
           result: out std_logic_vector(31 downto 0));
    end exponentAdder;
```

7.2.3 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;
```

7.2.4 Control Unit

```
    entity fpMultiplierControlUnit is
    Port (
        clk:in std_logic;
        reset:in std_logic;

        comparatorEq:in std_logic;
        comparatorGreater:in std_logic;
        outMantissaRes63:in std_logic;

        start: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(1 downto 0);
        selectExponentRes:out std_logic_vector(1 downto 0);
        selectMantissaRes:out std_logic_vector(1 downto 0);

        dbState:out std_logic_vector(3 downto 0);
        working:out std_logic;
        comparatorSelect:out std_logic_vector(2 downto 0)
    );
    end fpMultiplierControlUnit;
```

7.3 ALU Top level

```
    entity aluFloat is
    Port ( clk : in STD_LOGIC;
           num1 : in STD_LOGIC_VECTOR (31 downto 0);
           num2 : in STD_LOGIC_VECTOR (31 downto 0);
           op : in STD_LOGIC;
           start:in std_logic;
           res : out STD_LOGIC_VECTOR (31 downto 0);
           working:out std_logic);
    end aluFloat;
```

8 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