Floating-Point Unit

David Oliver Șipoș
Group 30433

Contents

1 Introduction

1.1 What is a Floating-Point Unit?

In computer architecture a **Floating-Point Unit**[4], also known as a math coprocessor is the part of the computer specifically designed for performing operations on floating-point numbers. Typical operations include addition, subtraction, multiplication, division and taking the square root. Some floating point units can compute more complex functions like exponentials, logarithms, and trigonometric functions. There are multiple ways to provide this unit. For early **Intel** processors a separate coprocessor could be purchased, but nowadays hardware **FPUs** are integrated on the same chip with the main processor. There exist also software libraries for emulating such a unit. Processors designed for embedded applications do not have support for such floating point operations, for example some low-power **ARM Cortex-M**[1] CPU cores support no instructions involving these kinds of numbers.

1.2 The IEEE-754 standard

One might wonder how real numbers are represented in computer science. There are a multitude of schemes to choose from. Nowadays computers use the floating-point format described by the **IEEE-754**[7] standard. However for some applications where precision is not important other formats are used, one such format is the **bfloat16**[3] developed by **Google Brain** for use in machine learning algorithms. The standard describes binary and decimal exchange formats and some fundamental operations which mainly include rounding and conversions. For this project the binary formats are the most relevant ones, more specifically the so-called **binary32** variation. Numbers are represented using normalized scientific notation, because we are using base 2, and the only digit different from 0 is 1 the most significant digit is not stored in memory, it's presence being implicit. As it's name suggests the **binary32** uses 32-bits to represent a number:

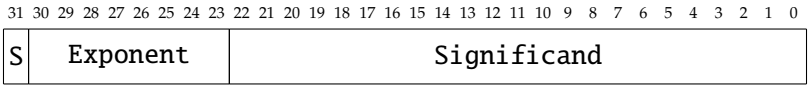

Figure 1: binary32

The `S` gives the sign of the number. In order to ease comparisons involinv floating-point numbers the 8-bit `exponent` is biased (with 127) to obtain a positive number. The `significand` contains the fractional part of our number with the implied 1 in case of normalized values.

One might raise the question of representing 0 if there is always a 1 before the `significand`. The standard describes reserved exponents used to represent special values, like $\pm$ 0, `NaN`, $\pm\infty$, and denormals. Denormals are so hard to deal with that hardware often does not provide support for them, instead some software libraries are used to complement the hardware and offer complete support of the standard.

1.3 Project overview

This project consists of implementing a floating-point unit capable of performing additon and multiplication using the wide-spread **binary32** format described by the **IEEE 754** standard.

The product of this project can be used as a separate floating-point coprocessor connected to the main processor of a computer, data exchange being based on the host operating system's serial communication API.

1.4 Project specification

The description of the unit shall be done using `VHDL`. Validation includes simulation in `Vivado 2023.1` and testing on a `Digilent Nexys A7` board equipped with a `Xilinx Artix-7 FPGA`. The results computed by the unit will be compared to ones given by other calculators to assess the precision of the designed logic circuits.

1.5 Project objectives

The main objective of the project is to design and implement logic circuits performing operations on floating-point values using the format provided by **IEEE 754**. Support for $\pm\infty$, `NaN` and denormals are excluded, considering that this is mainly an educational project and the implementation does not provide additional insight into floating-point arithmetic because they represent special cases requiring treatment with the help of dedicated condition checking circuitry. The standard specifies several rounding modes, but only one such mode will be implemented namely the default round to nearest mode.

The floating-point unit shall have two main components: a **control unit** and an **execution engine**, containing a register file, adders and multipliers. The register file will be endowed with 32 32-bit general purpose registers identified by their number starting from 0 to 31. These register will be written using special load instructions.

The control unit will take the form of a finite state machine, while the execution engine will be mainly combinational with some sequential elements. The control logic shall contain finite automata for transmission and reception using `UART` through the `USB` protocol, the conversion being carried out by the chip supplied by **FTDI** found on the development board. The transmission and reception shall be realized using clock signals generated by frequency dividers. For reception an oversampling approach will be employed to ensure data correctness. The chosen baud rate is of `9600` bits per second, with bytes of 8 bits in lengths, one stop bit and no parity bit.

In order to fulfill the main objective of the project the execution unit will contain a foating-point adder and multiplier, with a design trying to achieve peak performance in the given development environment. Addition will be based on **32-bit carry-lookahead adders**, with **combinational barrel shifters**, realized with multiplexers, used for normalizing and rounding the operands after addition. Multiplication shall be implemented using a **tree multiplier** with **full/half adders**. The normalizing and rounding circuits will be similar to the ones used for addition. Because subtraction is easy to support given an adder this operation will also be implemented.

Besides the aforementioned operations support for additional computations may be provided, including division and square rooting.

1.5.1 Instruction set

As previously hinted a simple **instruction set** will be devised and implemented in order to ease the workflow with the coprocessor. The **ISA** will be adhering to the **RISC** paradigm of computing. Instruction shall have the same 48-bit length regardless of their type and the operation they represent. Two major encodings are developed. The first one will use a register number and a floating-point immediate, while the second one two register numbers. This being a primitive instruction set, support for memory addressing will not be provided.

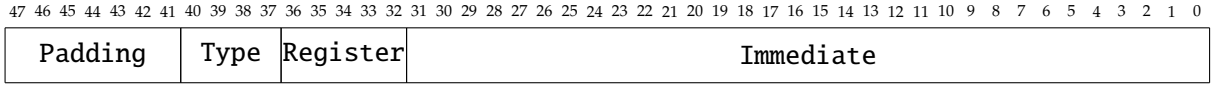

Figure 2: Register-Immediate format

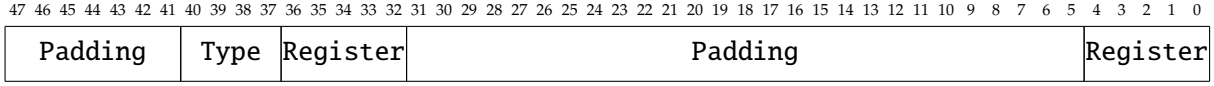

Figure 3: Register-register format

Supported instructions:

- `fldi reg imm` - load `imm` into the `reg` register
- `fldr reg0 reg1` - copy into `reg0` the value of `reg1`
- `ftx reg` - transmit the value of `reg` over `UART`
- `fadd reg0 reg1` - add to `reg0` the value of `reg1`
- `fsub reg0 reg1` - subtract from `reg0` the value of `reg1`
- `fmul reg0 reg1` - multiply `reg0` by the value of `reg1`

Other proposed instructions (they might not be supported):

- `fdiv reg0 reg1` - divide `reg0` by the value of `reg1`
- `fsqrt reg` - calculate the square root of `reg`

1.6 Project plan

In order keep track of development, the project shall be realized incrementally adhering to the following per project meeting plan:

Project meeting 3. complete implementation of a floating-point adder and testing

Project meeting 4. `UART` transmitting and receiving FSMs and `C++` program using the `Win32` API for communication with the board

Project meeting 5. control unit implementation and demonstration of the workflow with the unit

Project meeting 6. floating-point multiplier and testing

The present documentation will be expanded throughout the development of the project to include the most recent state of design and testing.

2 Bibliographic study

To document myself about floating-point numbers I started with the **IEEE-754** standard. I studied the parts related to the binary exchange formats which include data representation, operations, rounding modes, and special values. After I understood the fundamentals I went on to study possible implementations of floating-point adders and multipliers.

I mainly used two books: [6] and [5]. They provide detailed explanations of different architectures for foating-point operations. My approach is based on fast designs, which I simplified, sacrificing a bit of speed, to ease development.

After choosing my architecture I dug in the details, searching for the required fast adders, shifers and multipliers. From [6] I read the chapters about addtion, multiplication and floating-point operations. In this book there are some guidelines for designing such type of circuitry with high-level schematics. The second book I consulted for information is [5], more specifically Appendix J. In this appendix one can find algorithms for the operations, and rounding modes with clear descriptions.

For the design of the control unit and implementation of `UART` communication I used the knowledge gained in the previous years of study. These are simple finite-state machines for which no additional resources were needed.

3 Analysis

In what follows I describe the algorithms implemented by my circuits for every major component of the floating-point unit. The floating-point format was described in the Introduction of this document so I will not describe it again.

3.1 Addition

I found addition the hardest operation to implement. It has several steps during which exceptions can happen and checks must be performed to provide the correct result. This operation is carried out in a natural way, meaning that the operands are aligned to the same exponent, then their is sum is computed, normalized and rounded. I separated addition in the following steps:

1. Unpack
2. Exponent subtract
3. Preshift and complement
4. Add
5. Pre-round normalize
6. Round
7. Post-round normalize
8. Pack result

3.1.1 Unpack

This is one of the simplest stages of addition, the operands' components are separated and the hidden 1 is reinstated in the significand. The inputs to this step are our operands and the outputs represent their defining features: `sign` (1-bit), `exponent` (8-bits), and the `significand` (24-bits). These results are sent to the circuit performing the next operation.

3.1.2 Exponent subtract

During this step the exponents are subtracted. Following this subtraction a choice is made based on the sign of the difference. The first tentative exponent and the operand, for which alignment needs to be performed are determined. A flag is also generated indicating whether alignment will destroy the operand, more specifically the situation in which all bits are shifted out is detected.

3.1.3 Preshift and complement

One of the operands has suppport for preshift and the other one for complementation. The operand with the smaller exponent is shifted to the right by the amount computed in the previous stage. The other operand is negated if needed. It is possible that the operands will get exchanged because the one that needs alignment might not have support for it. Three bits used for normalization and rounding are computed: g (guard bit), r (round bit), s (sticky bit). The guard bit represents the most-significant shifted out bit, while the round is equal to the second most-significant shifted out bit. The sticky bit is equal to the logical or of the remaining lost bits. This step produces 32-bit numbers which are then forwarded to the adder.

3.1.4 Add

A 32-bit carry-lookahead is used to add the operands in 2's complement representation. The carry in to the adder will depend on the sign of the operands. This step produces the initial result of our operation which is complemented in the case it is negative and sent to be processed to obtain the final significand recognized by the standard.

3.1.5 Pre-round normalize

Because our operands are normalized they are in the interval $[1, 2)$. After the preshift stage they will be in $(-2, 2)$, with their sum found in $(-4, 4)$. This implies that following addition and negation our initial sum will be in the range given by $[0, 4)$. Taking into consideration this conclusion we have two major cases. If the operand is greater than equal to 2 we have to shift it right by one bit, and increment the tentative exponent. The other case occurs when the result is less than 1, meaning we have to shift left until a 1 appears in front of the radix point. This also involves subtracting from the exponent the shift amount, this amount equaling the number of leading zeros of the result. During this stage, shifting modifies the round and sitcky bits. The value of the guard bit is used during left shifting, as it's name suggests, to guard against loss of precision.

3.1.6 Round

Based on the current rounding mode, the values of the round and sitcky bits we perform a rounding, by adding *ulp* if needed. Because we might modify the significand of the result we need another normalization step.

3.1.7 Post-round normalize

After rounding the result can become greater than equal to 2. This step detects the need of right shifting by one bit. Again shifting implies modifying the tentative exponent.

3.1.8 Pack

This is the final step of addition, in which the components of the result are assembled. Here we perform a check to see if the result is zero, because zero has a special representation. Also during this stage the final overflow and underflow bits are generated, based on the flags obtained by previous steps which modify the exponent.

3.2 Multiplication

Multiplication is much simpler to implement than addition, because there is no need for preshift alignment. One just adds the exponents and multiplies the significands. The process of multiplication can be grouped in the following steps:

1. Unpack (identical to addition)
2. Add exponents
3. Multiply
4. Pre-round normalize
5. Round and normalize (identical to addtion)
6. Pack (identical to addition)

3.2.1 Add exponents

The exponents are added and the bias is subtracted. An overflow or underflow might occur, so in this we output the required flags.

3.2.2 Multiply

The 24-bit significands are multiplied producing a 48-bit product which is then forwarded to the normalization step.

3.2.3 Pre-round normalize

The significands are in the interval $[1, 2)$, this means the product will be in the interval $[1, 4)$. Thus at most one right shift and exponent increment might be needed thus an overflow flag is output by this stage.

3.3 Control Unit

The control unit is a finite-state machine. It waits for the reception of a 48-bit instruction then enters the decode state. In this state the operands are fectched, and the proper future state is selected based on the operation. An execute state follows after which the instruction execution is ended by a write or transmit state.

4 Design

4.1 Adder

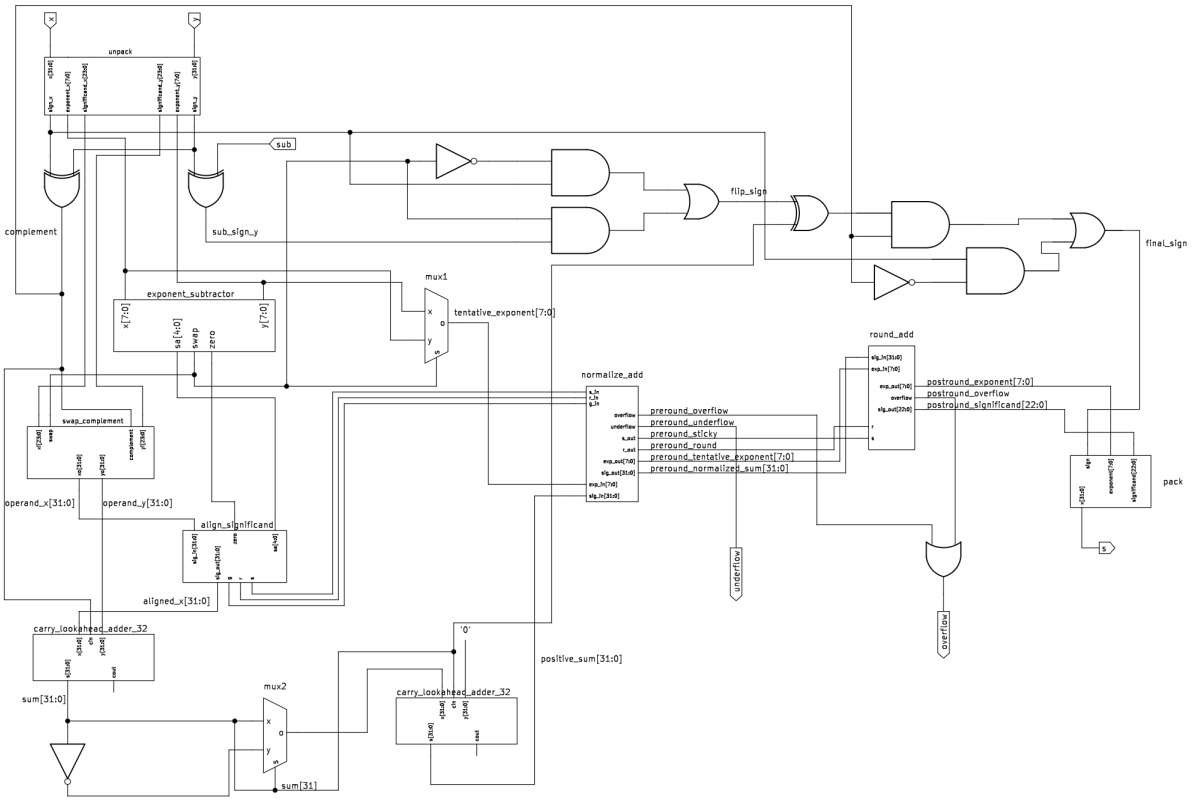

Figure 4: Adder schematic

One can observe that there is a component for each stage mentioned above. Additional logic is introduced for determining the sign of the result, whether the result is zero or whether there is an overflow or underflow. This circuits is combinational and computes the sum in one clock cycle.

4.2 Multiplier

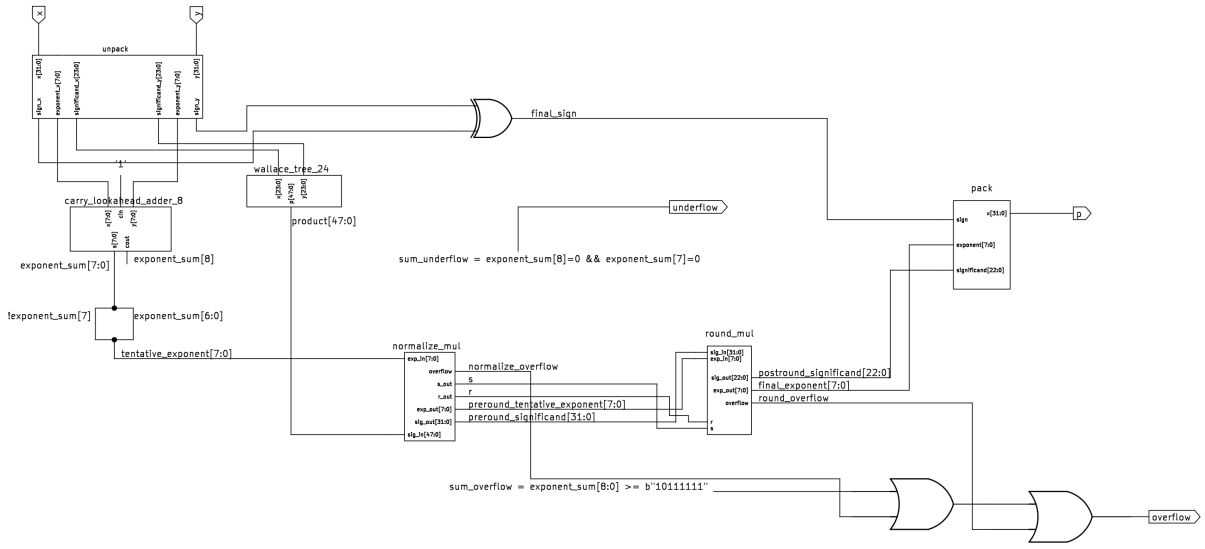

Figure 5: Multiplier schematic

4.3 Control Unit

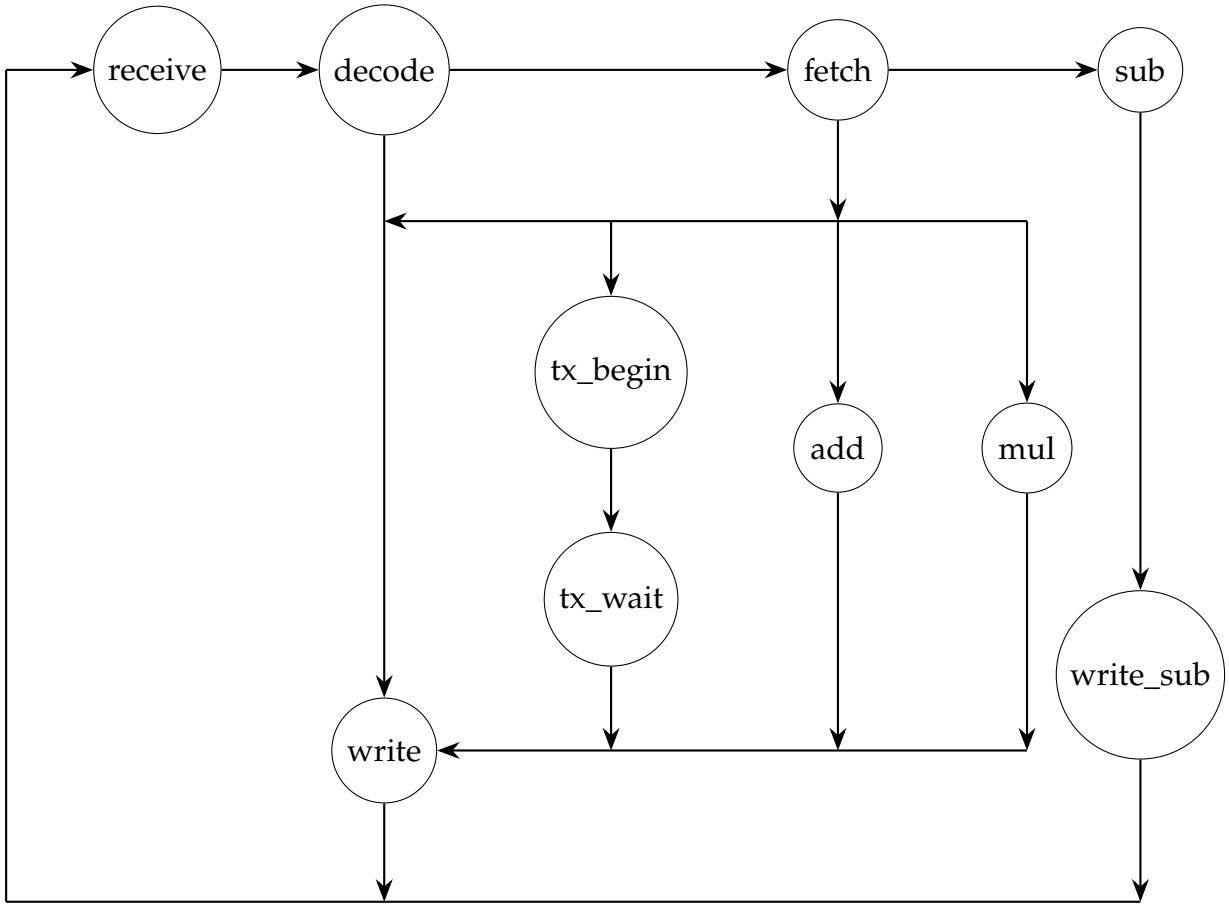

Figure 6: Control automaton

State	Function
receive	Wait for instruction
decode	Decode instruction
fetch	Fetch operands from the register file
add	Execute an addition
sub	Execute a subtraction
mul	Execute a multiplication
tx_begin	Begin register data transmission
tx_wait	Wait for transmission to finish
write	Write result to the register file
write_sub	Write result to the register file with the sub signal asserted

4.4 Floating-Point Unit

The black box of the unit can be seen in the upper left corner, while the rest of the figure depicts the high-level structure of the floating-point unit.

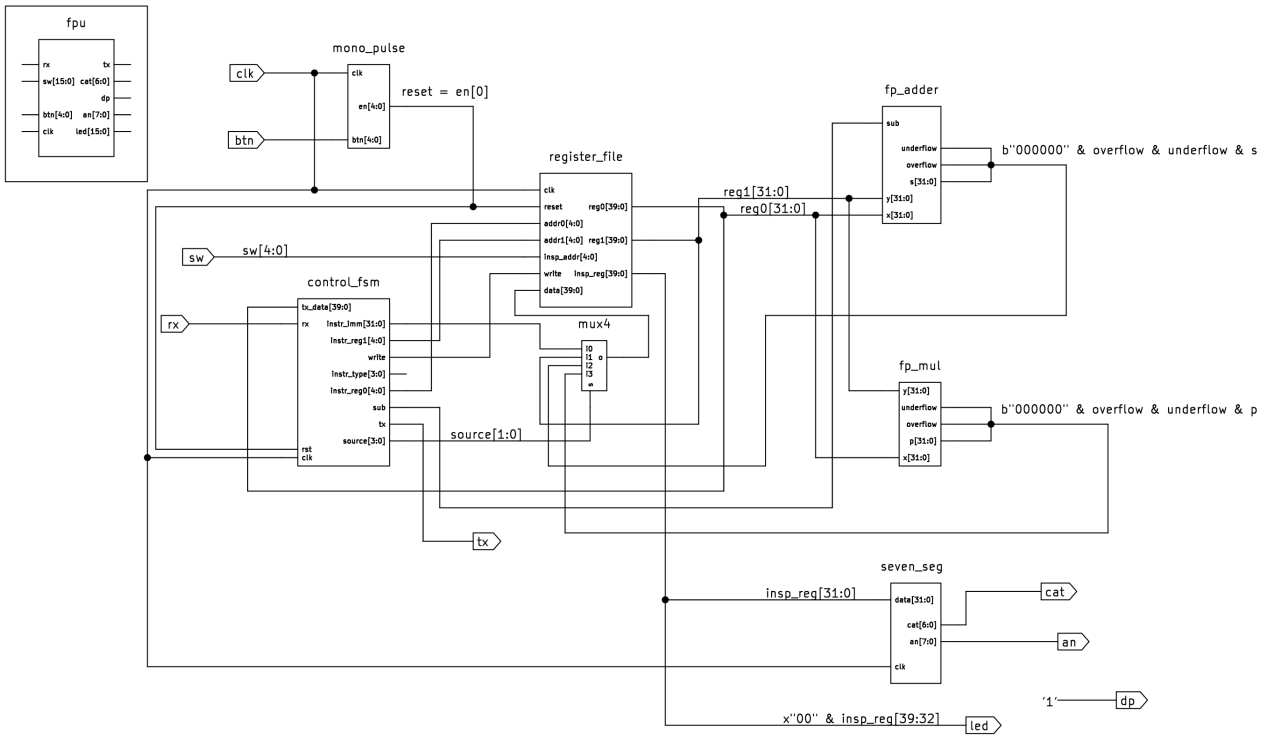

Figure 7: Floating-Point Unit schematic

5 Implementation

In the following section the interface and function of the main subcomponents will be presented.

5.1 Adder

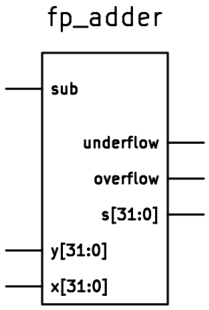

Figure 8: Adder unit

Port	Direction	Description
x[31:0]	IN	Operand x
y[31:0]	IN	Operand y
sub	IN	Subtract
underflow	OUT	Underflow
overflow	OUT	Overflow
s[31:0]	OUT	Sum

5.1.1 Unpack

Unpacks inputs and outputs their constituent parts.

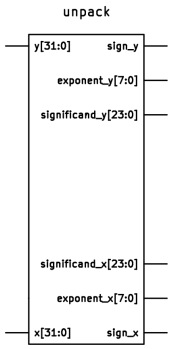

Figure 9: Unpack unit

Port	Direction	Description
x[31:0]	IN	Operand x
y[31:0]	IN	Operand y
sign_x	OUT	Sign of x
exponent_x[7:0]	OUT	Exponent of x
significand_x[23:0]	OUT	Significand of x, hidden 1 included
sign_y	OUT	Sign of y
exponent_y[7:0]	OUT	Exponent of y
significand_y[23:0]	OUT	Significand of y, hidden 1 included

5.1.2 Exponent subtractor

Subtracts the exponents, determines which operand needs to be aligned and calculates the shift amount.

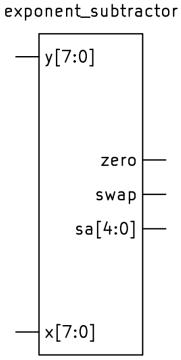

Figure 10: Exponent subtractor unit

Port	Direction	Description
x[7:0]	IN	Exponent of x
y[7:0]	IN	Exponent of y
zero	OUT	Indicates wether x will be shifted out completely
swap	OUT	Indicates the case in which y needs to be shifted(swap operands)
sa[4:0]	OUT	Shift amount required for proper alignment

5.1.3 Swap and Complement

Based on the output of the previous component it swaps the operands and may also perform negation depending on their signs and the sub input signal to the adder.

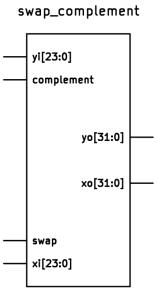

Figure 11: Swap and complement unit

Port	Direction	Description
xi[23:0]	IN	Operand x
yi[23:0]	IN	Operand y
complement	IN	Complement y
swap	IN	Swap operands
xo[31:0]	OUT	Output operand x
yo[31:0]	OUT	Output operand y

5.1.4 Align significand

Using the shift amount calculated by the exponent subtractor it shifts right one of the operands to align with the other also producing the initial value of the bits needed for rounding.

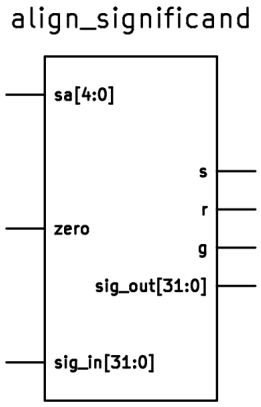

Figure 12: Align significand unit

Port	Direction	Description
sa[4:0]	IN	Shift amount
zero	IN	Indicates whether sig_in will be zeroed
sig_in[31:0]	IN	Input significand
s	OUT	Sticky bit
r	OUT	Round bit
g	OUT	Guard bit

5.1.5 Carry-lookahead adder

Simple 32-bit carry-lookahead adder.

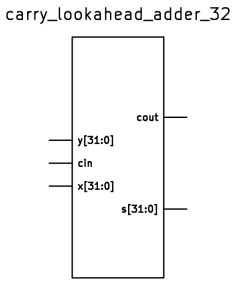

Figure 13: Carry-lookahead adder unit

Port	Direction	Description
x[31:0]	IN	Operand x
y[31:0]	IN	Operand y
cin	IN	Carry in
s[31:0]	OUT	Sum
cout	OUT	Carry out

5.1.6 Normalize Add

This unit performs normalization after addition by shifting the result. It may also modify the bits used for rounding and the exponent. Because the exponent of the result changes an overflow or an underflow might occur.

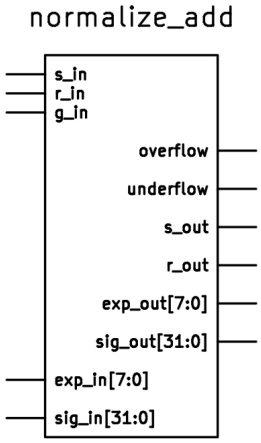

Figure 14: Normalize add unit

Port	Direction	Description
exp_in[7:0]	IN	Tentative exponent
sig_in[31:0]	IN	Input significand
s_in	IN	Input sticky bit
r_in	IN	Input round bit
g_in	IN	Input guard bit
overflow	OUT	Overflow during normalization
underflow	OUT	Underflow during normalization
s_out	OUT	Output sitcky bit
r_out	OUT	Output round bit
exp_out[7:0]	OUT	Output exponent
sig_out[31:0]	OUT	Normalized significand

5.1.7 Round Add

It rounds the result according to the default rounding mode, round to nearest even, described by **IEEE**. Rounding involves addition, thus the result needs to go through another normalization, which might trigger an overflow.

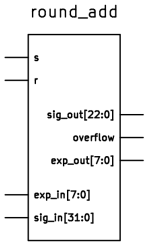

Figure 15: Round add unit

Port	Direction	Description
exp_in[7:0]	IN	Tentative exponent
sig_in[31:0]	IN	Input significand
s	IN	Input sticky bit
r	IN	Input round bit
overflow	OUT	Overflow during rounding
exp_out[7:0]	OUT	Output exponent
sig_out[22:0]	OUT	Rounded significand, no hidden 1, final form

5.1.8 Pack

Assembles the final result, additional logic is used to detect zero.

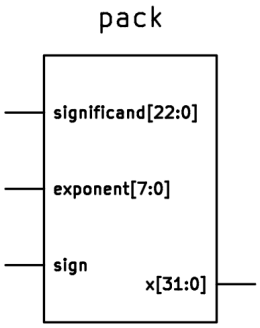

Figure 16: Pack unit

Port	Direction	Description
exponent[7:0]	IN	Input exponent
significand[31:0]	IN	Input significand
sign	IN	Input sign
x[31:0]	OUT	Final result

5.2 Multiplier

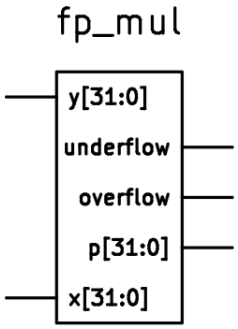

Figure 17: Multiplier unit

Port	Direction	Description
x[31:0]	IN	Operand x
y[31:0]	IN	Operand y
underflow	OUT	Underflow
overflow	OUT	Overflow
p[31:0]	OUT	Product

5.2.1 Wallace tree

24-bit Wallace Tree multiplier generated using a custom C++ class template, which can be used to generate Wallace Trees of any width.

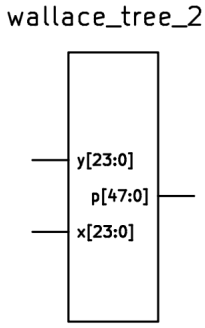

Figure 18: Wallace tree unit

Port	Direction	Description
x[23:0]	IN	Operand x
y[23:0]	IN	Operand y
p[47:0]	OUT	Product

5.2.2 Normalize Mul

Product is normalized. During normalization the operand is shifted right thus only overflow might occur. The round and sticky bits are also produced by this unit, we do not need a guard bit due to the fact that we are always right shifting during normalization and rounding.

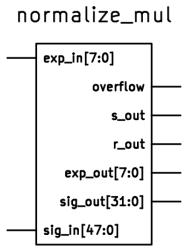

Figure 19: Normalize mul unit

Port	Direction	Description
exp_in[7:0]	IN	Tentative exponent
sig_in[47:0]	IN	Input significand
overflow	OUT	Overflow during normalization
s_out	OUT	Output sitcky bit
r_out	OUT	Output round bit
exp_out[7:0]	OUT	Output exponent
sig_out[31:0]	OUT	Normalized significand

5.2.3 Round Mul

The result is rounded and further normalized according to the **IEEE** round to nearest even mode. Again overflow might occur meaning that a flag needs to generated.

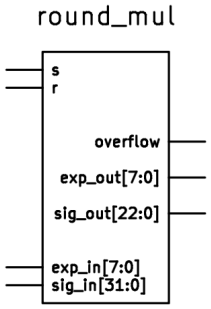

Figure 20: Round mul unit

Port	Direction	Description
exp_in[7:0]	IN	Tentative exponent
sig_in[31:0]	IN	Input significand
s	IN	Input sitcky bit
r	IN	Input round bit
overflow	OUT	Overflow during rounding
exp_out[7:0]	OUT	Output exponent
sig_out[22:0]	OUT	Rounded significand, no hidden 1, final form

5.3 Control Unit

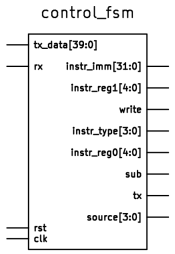

Figure 21: Control Unit

Port	Direction	Description
tx_data[39:0]	IN	Data to transmit
rx	IN	UART rx
rst	IN	Reset
clk	IN	Clock
instr_imm[31:0]	OUT	Instruction immediate
instr_reg0[4:0]	OUT	Operand register
instr_reg1[4:0]	OUT	Operand register
instr_type[3:0]	OUT	Instruction type
write	OUT	Write to register file
sub	OUT	Subtract
tx	OUT	UART tx
source[3:0]	OUT	Register source selection

5.3.1 RX automaton

Finite state machine used to receive a byte bit by bit through UART.

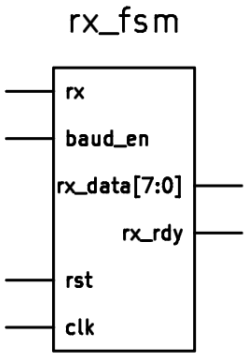

Figure 22: rx automaton

Port	Direction	Description
rx	IN	UART rx
rst	IN	Reset
clk	IN	Clock
baud_en	IN	Baud clock, generated by another unit
rx_rdy	OUT	Indicates the reception of one byte
rx_data[7:0]	OUT	Received data

5.3.2 TX automaton

Finite state machine used to send a byte bit by bit using UART.

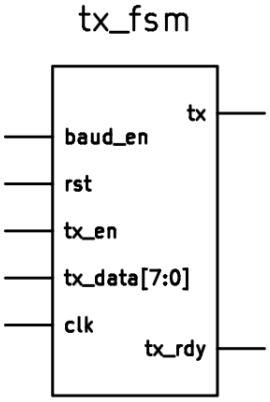

Figure 23: tx automaton

Port	Direction	Description
tx_en	IN	A pulse on this port will start transmission
rst	IN	Reset
clk	IN	Clock
baud_en	IN	Baud clock, generated by another unit
tx_data[7:0]	IN	Byte to transmit
tx	OUT	UART tx
tx_rdy	OUT	Indicates the idle state, meaning ready to transmit

5.3.3 Receive instruction

It controls the aforementioned reception automaton by generating the oversampling baud rate clock and accumulating a 6-byte instruction.

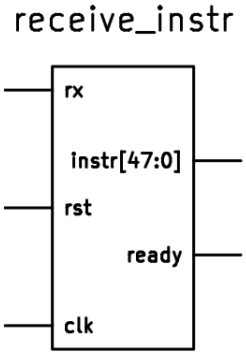

Figure 24: Receive instruction automaton

Port	Direction	Description
rx	IN	UART rx
rst	IN	Reset
clk	IN	Clock
ready	OUT	Indicates a complete instruction
instr[47:0]	OUT	Received instruction

5.3.4 Transmit float

This unit, with the help of `tx_fsm`, transmits the contents of one register, representing a 32-bit float and an additional flags byte.

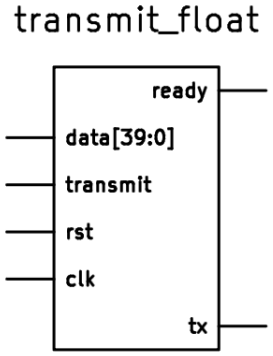

Figure 25: Transmit float and flags automaton

Port	Direction	Description
`data[39:0]`	IN	Data to transmit
`transmit`	IN	A pulse on this port triggers transmission
`clk`	IN	Clock
`rst`	IN	Reset
`ready`	OUT	Indicates successful transmission
`tx`	OUT	UART tx

5.4 Register file

Simple register file, it contains 32 40-bit general purpose registers, each storing a float and a flags byte. An additional read port can be used to inspect the contents of the registers.

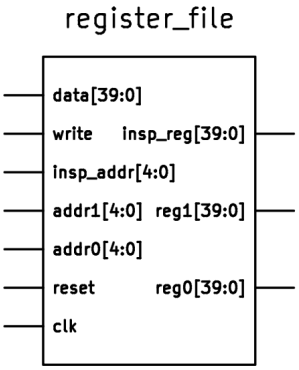

Figure 26: Register file

Port	Direction	Description
data[39:0]	IN	Data to write
write	IN	Write data
insp_addr[4:0]	IN	Address used for inspecting register contents
addr1[4:0]	IN	Address of register 1
addr0[4:0]	IN	Address of register 0
reset	IN	Reset
clk	IN	Clock
insp_reg[39:0]	OUT	Inspected register contents
reg1[39:0]	OUT	Contents of register 1
reg0[39:0]	OUT	Contents of register 0

5.5 Floating-Point Unit

This represents the main enitity of the whole project. It is the top-level interface of the Floating-Point Unit.

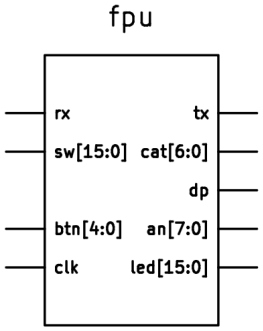

Figure 27: Floating-Point Unit

Port	Direction	Description
rx	IN	UART rx
sw[15:0]	IN	Switches
btn[4:0]	IN	Buttons
clk	IN	Clock
tx	OUT	UART tx
cat[6:0]	OUT	Seven-segment display cathodes
dp	OUT	Decimal point, set to HIGH
an[7:0]	OUT	Seven-segment display anodes
led[15:0]	OUT	LEDs

6 Tests and Experiments

This section contains screenshots of the C++ program used to communicate with the board, representing moments of the testing phase. Each operation was tested in three ways. First with reasonable operands, then computations which involved zero followed. Lastly inputs which generate exceptions were provided.

6.1 Addition

```
Connected to the board
Port state:
        BaudRate: 9600
        ByteSize: 8
        Parity: 0
        StopBits: 0
fldi 0 -1
Sending instruction: 0x0000bf800000
fldi 2 1
Sending instruction: 0x00023f800000
fadd 2 0
Sending instruction: 0x006200000000
ftx 2
Sending instruction: 0x002200000000
Received: 0 (0x0 Overflow: 0 Underflow: 0)
fldi 5 3.14159
Sending instruction: 0x000540490fd0
fldi 8 2.71828
Sending instruction: 0x0008402df84d
fldr 10 8
Sending instruction: 0x004a00000008
ftx 10
Sending instruction: 0x002a00000000
Received: 2.71828 (0x402df84d Overflow: 0 Underflow: 0)
fadd 10 5
Sending instruction: 0x006a00000005
ftx 10
Sending instruction: 0x002a00000000
Received: 5.85987 (0x40bb840e Overflow: 0 Underflow: 0)
fldr 21 10
Sending instruction: 0x00550000000a
ftx 21
Sending instruction: 0x003500000000
Received: 5.85987 (0x40bb840e Overflow: 0 Underflow: 0)
fldi 0 0.85
Sending instruction: 0x00003f59999a
fsub 21 0
Sending instruction: 0x009500000000
ftx 21
Sending instruction: 0x003500000000
Received: 5.00987 (0x40a050db Overflow: 0 Underflow: 0)
```

(a) Addition with normal values

```
        BaudRate: 9600
        ByteSize: 8
        Parity: 0
        StopBits: 0
fldi 5 1.17549463108e-38
Sending instruction: 0x000500800002
fldi 7 1.17549449095e-38
Sending instruction: 0x000700800001
fsub 5 7
Sending instruction: 0x008500000007
ftx 5
Sending instruction: 0x002500000000
Received: 1.62259e+32 (0x75000000 Overflow: 0 Underflow: 1)
fldi 10 340282346638528859811704183484516925440.0000000000000000
Sending instruction: 0x000a7f7fffff
fldi 11 340282346638528859811704183484516925440.0000000000000000
Sending instruction: 0x000b7f7fffff
fadd 10 11
Sending instruction: 0x006a0000000b
ftx 10
Sending instruction: 0x002a00000000
Received: nan (0x7fffffff Overflow: 1 Underflow: 0)
fldi 20 -340282346638528859811704183484516925440.0000000000000000
Sending instruction: 0x0014ff7fffff
fldi 21 -340282346638528859811704183484516925440.0000000000000000
Sending instruction: 0x0015ff7fffff
fadd 20 21
Sending instruction: 0x007400000015
ftx 20
Sending instruction: 0x003400000000
Received: -nan (0xffffffff Overflow: 1 Underflow: 0)
fldi 30 -340282346638528859811704183484516925440.0000000000000000
Sending instruction: 0x001eff7fffff
fldi 31 340282346638528859811704183484516925440.0000000000000000
Sending instruction: 0x001f7f7fffff
fadd 30 31
Sending instruction: 0x007e0000001f
ftx 30
Sending instruction: 0x003e00000000
Received: -0 (0x80000000 Overflow: 0 Underflow: 0)
fldi 14 340282346638528859811704183484516925440.0000000000000000
Sending instruction: 0x000e7f7fffff
fldi 15 340282346638528859811704183484516925440.0000000000000000
Sending instruction: 0x000f7f7fffff
fsub 14 15
Sending instruction: 0x008e0000000f
ftx 14
Sending instruction: 0x002e00000000
Received: 0 (0x0 Overflow: 0 Underflow: 0)
```

(b) Addition with exceptions

```
Connected to the board
Port state:
        BaudRate: 9600
        ByteSize: 8
        Parity: 0
        StopBits: 0
fldi 0 0
Sending instruction: 0x000000000000
fldi 1 0
Sending instruction: 0x000100000000
fadd 0 1
Sending instruction: 0x006000000001
ftx 0
Sending instruction: 0x002000000000
Received: 0 (0x0 Overflow: 0 Underflow: 0)
fadd 0 0
Sending instruction: 0x006000000000
ftx 0
Sending instruction: 0x002000000000
Received: 0 (0x0 Overflow: 0 Underflow: 0)
fsub 0 1
Sending instruction: 0x008000000001
ftx 0
Sending instruction: 0x002000000000
Received: 0 (0x0 Overflow: 0 Underflow: 0)
```

(c) Addition involving zero

Figure 28: Addition tests

1. $A = -1, B = -1, Operation = A + B, Result = 0$ - test if the adder detects zero result, and outputs zero's special representation
2. $A = 3.14159, B = 2.71828, Operation = A + B, Result = 5.85987$ - I just like π and e
3. $A = 1.17549463108\text{e}{-38}, B = 1.17549449095\text{e}{-38}, Operation = A - B, Result = Underflow$ - Subtracting the two smallest representable numbers should lead to underflow
4. $A = 340282346638528859811704183484516925440$,
   $B = 340282346638528859811704183484516925440, Operation = A + B, Result = Overflow$ - Adding the two largest representable numbers should lead to overflow
5. $A = -340282346638528859811704183484516925440$,
   $B = -340282346638528859811704183484516925440, Operation = A + B, Result = Overflow$ - Adding the two largest representable numbers in absolute value should lead to overflow, because the exponent is large
6. $A = 340282346638528859811704183484516925440$,
   $B = -340282346638528859811704183484516925440, Operation = A + B, Result = 0$
7. $A = 340282346638528859811704183484516925440$,
   $B = 340282346638528859811704183484516925440, Operation = A - B, Result = 0$
8. $A = 0, B = 0, Operation = A + B, Result = 0$ - Test if adder recognizes zero at the input
9. $A = 0, B = 0, Operation = A - B, Result = 0$

6.2 Multiplication

```
Connected to the board
Port state:
        BaudRate: 9600
        ByteSize: 8
        Parity: 0
        StopBits: 0
fldi 0 3.14159
Sending instruction: 0x000040490fd0
fldi 1 2.71828
Sending instruction: 0x0001402df84d
fmul 0 1
Sending instruction: 0x00a000000001
ftx 0
Sending instruction: 0x002000000000
Received: 8.53972 (0x4108a2b3 Overflow: 0 Underflow: 0)
fldi 5 0.5
Sending instruction: 0x00053f000000
fmul 5 9
Sending instruction: 0x00a500000009
ftx 5
Sending instruction: 0x002500000000
Received: 0 (0x0 Overflow: 0 Underflow: 0)
fldi 5 0.5
Sending instruction: 0x00053f000000
fmul 5 0
Sending instruction: 0x00a500000000
ftx 5
Sending instruction: 0x002500000000
Received: 4.26986 (0x4088a2b3 Overflow: 0 Underflow: 0)
fldi 4 -0.1
Sending instruction: 0x0004bdcccccd
fmul 4 0
Sending instruction: 0x00a400000000
ftx 4
Sending instruction: 0x002400000000
Received: -0.853972 (0xbf5a9deb Overflow: 0 Underflow: 0)
```

(a) Multiplication with normal values

```
Port state:
        BaudRate: 9600
        ByteSize: 8
        Parity: 0
        StopBits: 0
fldi 0 340282346638528859811704183484516925440.0000000000000000
Sending instruction: 0x00007f7fffff
fldi 1 340282346638528859811704183484516925440.0000000000000000
Sending instruction: 0x00017f7fffff
fmul 0 1
Sending instruction: 0x00a000000001
ftx 0
Sending instruction: 0x002000000000
Received: 1 (0x3f7ffffe Overflow: 1 Underflow: 0)
fldi 5 340282346638528859811704183484516925440.0000000000000000
Sending instruction: 0x00057f7fffff
fldi 6 -340282346638528859811704183484516925440.0000000000000000
Sending instruction: 0x0006ff7fffff
fmul 5 6
Sending instruction: 0x00a500000006
ftx 5
Sending instruction: 0x002500000000
Received: -1 (0xbf7ffffe Overflow: 1 Underflow: 0)
fldi 25 1.17549449095e-38
Sending instruction: 0x001900800001
fldi 26 1.17549449095e-38
Sending instruction: 0x001a00800001
fmul 25 26
Sending instruction: 0x00b90000001a
ftx 25
Sending instruction: 0x003900000000
Received: 16 (0x41800002 Overflow: 0 Underflow: 1)
fldi 9 -1.17549449095e-38
Sending instruction: 0x000980800001
fldi 10 1.17549449095e-38
Sending instruction: 0x000a00800001
fmul 10 9
Sending instruction: 0x00aa00000009
ftx 10
Sending instruction: 0x002a00000000
Received: -16 (0xc1800002 Overflow: 0 Underflow: 1)
```

(b) Multiplication with exceptions

```
Connected to the board
Port state:
        BaudRate: 9600
        ByteSize: 8
        Parity: 0
        StopBits: 0
fldi 1 2
Sending instruction: 0x000140000000
fmul 0 1
Sending instruction: 0x00a000000001
ftx 0
Sending instruction: 0x002000000000
Received: 0 (0x0 Overflow: 0 Underflow: 0)
fmul 8 9
Sending instruction: 0x00a800000009
ftx 0
Sending instruction: 0x002000000000
Received: 0 (0x0 Overflow: 0 Underflow: 0)
ftx 8
Sending instruction: 0x002800000000
Received: 0 (0x0 Overflow: 0 Underflow: 0)
```

(c) Multiplication involving zero

Figure 29: Multiplication tests

1. $A = 3.14159, B = 2.71828, Operation = A * B, Result = 8.53972$ - I just like π and e
2. $A = 0.5, B = 0, Operation = A * B, Result = 0$ - Test if multiplier knows to work with zero
3. $A = 0.5, B = 8.53972, Operation = A * B, Result = 4.26986$ - Test if multiplier can divide
4. $A = -0.1, B = 8.53972, Operation = A * B, Result = -0.853972$ - Test if multiplier can divide
5. $A = 1.17549449095\text{e}{-38}, B = 1.17549449095\text{e}{-38}, Operation = A * B, Result = Underflow$ - Multiplying the two smallest representable numbers should lead to underflow

6. $A = -1.17549449095\text{e}{-38}, B = -1.17549449095\text{e}{-38}, Operation = A*B, Result = Underflow$ - Multiplying the two smallest representable numbers should lead to underflow
7. $A = 340282346638528859811704183484516925440$,
   $B = 340282346638528859811704183484516925440, Operation = A * B, Result = Overflow$ - Multiplying the two largest representable numbers should lead to overflow
8. $A = -340282346638528859811704183484516925440$,
   $B = 340282346638528859811704183484516925440, Operation = A * B, Result = Overflow$ - Multiplying the two largest representable numbers in absolute value should lead to overflow

7 Conclusion

In conclusion, the proposed objectives of the project were achieved. A description of the **IEEE 754 floating-point standard** was given, followed by specifications of algorithms for performing **addition** and **multiplication**. The main stages of each algorithm were described in detail, then in the design and implementation phases complete schematics were presented, with specifications of every major subcomponent. The resulting floating-point unit was validated, extensive testing proving the correctness of the implementation.

References

[1] *ARM Cortex-M*. https://en.wikipedia.org/wiki/ARM_Cortex-M.

[2] *Barrel shifter*. https://en.wikipedia.org/wiki/Barrel_shifter.

[3] *bfloat16 floating-point format*. https://en.wikipedia.org/wiki/Bfloat16_floating-point_format.

[4] *Floating-Point Unit*. https://en.wikipedia.org/wiki/Floating-point_unit.

[5] David A. Patterson John L. Hennessy. *Copmuter Architecture: A Quantitative Approach, Sixth Edition*. 50 Hampshire Street, 5th Floor, Cambridge, MA 02139, USA: Elsevier, 2019.

[6] Behrooz Parhami. *Computer Arithmetic, Algorithms and Hardware Designs, Second Edition*. 198 Madison Avenue, New York, NY 10016, USA: Oxford University Press, 2010.

[7] IEEE Computer Society. *IEEE Standard for Floating-Point Arithmetic*. 3 Park Avenue, New York, NY 0016-5997, USA: IEEE, 2008.