

# RISC-V Floating-Point CPU project

Yuval Gerzon 205738024

David Jotkowitz 323601013

RISC-V CPU with floating point support implementation

Under the supervision of Dr. Amit Berman

## Table of contents

...

Abstract.....	<b>3</b>
Symbols and abbreviations .....	4
Introduction and previous work review.....	5-12
CPU flow .....	<b>13</b>
Verifying software reference model .....	14-16
Hardware planning and implementation .....	17-21
Hardware tests .....	22-39
Synthesis.....	40-42
Summary.....	43
Sources and bibliography .....	44

## Abstract

In our project, we have implemented a CPU supporting arithmetic Floating-Point operations defined by RISC-V ISA.

We planned a reference software model of our chip in order to test our hardware implementation and tested the model itself with complex arithmetic computations. We did this by using Taylor expansions and other calculus methods to compute values manually and then made sure our model was consistent with the calculations.

The hardware itself was logically designed using Verilog, and used us to implement complexed arithmetic operations, such as addition, subtraction, multiplication, division and square root, as well as non-arithmetic operations, such as comparison and MIN/MAX.

In some cases, the implementation of different operations and behavior of our system required complex planning. For example, adding square root required a support of multi-clock operations to our design, which resulted in a stall mechanism for the pipeline excepting only the ALU, which had to be enabled to work for a few clock cycles with the same input to then generate a complex operation. This change required complex control logic for the whole pipeline.

## Symbols and abbreviations

...

ADD - addition

ALU – arithmetic logic unit

CPU – central processing unit

DIV - division

FP - Floating point

FP-32 – 32-bit floating point

ISA – instruction set architecture

MUL - multiplication

RISC – reduced instruction architect

SQRT – square root

Spec – specifications

SUB - subtraction

...

## Introduction and previous work review

### RISC architecture

The RISC (reduced instruction set computer) architecture is an improvement on existing CISC (complex instruction set computer) architecture that was introduced in the early 1980's. The first microprocessors developed a decade earlier did not have the capability to store in its RAM too many instructions at once, so the architecture utilized in those microprocessors emphasized the ability to unpack few complex instructions to long procedures, each taking several clock-cycles to perform.

With the advance of RAM technology, the need for short programs became obsolete, and the development began of a new architecture that could improve the time efficiency of programs running on it. This was manifested in the RISC architecture - each command in its instruction set runs on just one clock cycle. This allowed for an efficient pipelining of programs, reducing the time it took for each program to run.

The inherent downside of the RISC ISA is that its commands must be simple, so each program must be expanded to delineate its process in bare-bones operations on and between its general-purpose registers, and memory-addressing requests. Also, more memory caches and general-purpose registers are needed, so that between commands not too much time will be spent retrieving data. Nevertheless, today many chip developers have decided to stick with RISC architecture anyway.

A table comparing the RISC and CISC ISAs:

	RISC	CISC
Amount of general-purpose registers necessary	More	Less
Datatypes	Usually two (int and float)	Usually more than two
Instructions	Load/store general registers, operations on datatypes in registers	Correspond to datatypes
Instruction formats	Fixed length, two main types: Load/store, $R = R \text{ op } R$	Variable length, many types: Load/store, $R = R \text{ op } R$ , $R = \text{Mem op } R$ , $M = \text{Mem op Mem}$
Encoding	1 instruction= 1operand or 1 operation	1 instruction= 1 statement
Design objective	Trade off program length, minimize time to execute instruction	Minimum program length Maximum work/instruction
Implementation	Hard-wired processor and software. Fast processor and fast cache for instructions; instructions take one clock cycle; simple pipeline	Microprogrammed processor; slow primary memory and fast clock: instructions take variable time; pipeline is complex; larger implementation

		may result in longer design time
Caching	Essential for instructions	Useful
Compiler design	Should stress best ordering	Should stress finding right instructions
Philosophy	Move all functions to software	Move any useful software function into hardware, including diagnostics.

### The RISC-V ISA

RISC-V is a free and open ISA first developed by engineers at Berkley university of California in 2010 as part of the Parallel Computing Laboratory (Par Lab) originally with academic intentions in mind (research and education). Today, the RISC-V ISA is used both in academia and in the hardware industry for various needs. The RISC-V (often referred to as RV) ISA, unlike other ISAs, is not protected by corporate copyright but rather guaranteed by Berkley university to be free of use and accessible to all.

It has two main memory addressing variants: 32-bit and 64-bit, as well as 128-bit expansion with less practical uses nowadays.

RISC-V ISA supports multicore implementations, which can allow better overall latency and throughput to the microprocessor when comparing to a single-core implementation.

RISC-V supports the 2008 IEEE-754 floating-point standard<sup>1</sup> which allows us to plan a micro-architecture that supports floating point operations. The support includes single-precision floating-point registers denoted by “F” (for floating-point) for four byte (32-bit) floating point number representation, double-precision floating-point registers denoted by “D” (for double-precision) for eight byte (64-bit) floating point number representation, as well as the less commonly used quad-precision floating-point registers denoted by “Q” (for Quad-precision) for sixteen byte (128-bit) floating point number representation. The “Q” extension requires RV64 and would not operate on RV32. In addition to the floating-point extension, RISC-V ISA holds support for an integer register extension denoted by “I”, multiplication and division for integers denoted by “M”, and the standard atomic instruction extension denoted by “A” which supports inter-processor synchronization operations. All the above except “Q” are the standard extensions “IMAFD” abbreviated as “G” (for example RV32G).

RV has various lengths for instructions, but our main focus will be on the 32-bit instructions, which are commonly most used. The 32-bit instruction will be of the following format:

$$\underbrace{xxx \dots xxx}_{27 \text{ bits}} bbb11, \quad \text{where } bbb \neq 111$$

Other formats are one of the following:

---

<sup>1</sup> According to The RISC-V Instruction Set Manual Version 2.2, May 7, 2017

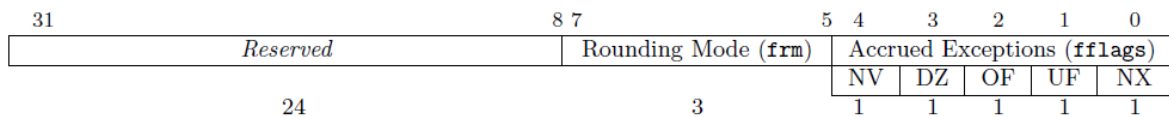


## Memory of RISC-V

As mentioned before, RV ISA has two main address space variants, 32-bit and 64-bit, as well as the less commonly used 128-bit. We will focus on the first two and on floating-point extension registers, meaning “F”, “D” and “Q” extensions.

Register file – “F” extension has a register-file of thirty-two registers, each 32 bits long. The registers are marked f0-f31. The “D” extension widens the register file so that each register has 64 bits. Similarly to the “D” extension, the “Q” extension widens the registers to have 128-bits in order to represent Quad-precision floating point numbers. Nevertheless, all standard operations remain 32-bit.

Other than that, both RV32G and RV64G include an additional 32-bit Floating-Point Control and Status Register (FCSR). It holds values that can determine the flow of the processor and has bits that are automatically raised during the flow, such as when there occurs an overflow in the arithmetic operation. This register is a part of “F” extension and in use also for “D” and “Q” extensions. The FCSR has the following form:



With the following uses:

For arithmetic operations, if the RM field of the instruction is set to "dynamic rounding mode" then the ALU will read the rounding mode from the FCSR. The general rounding modes are given in the following table:

Rounding Mode	Mnemonic	Meaning
000	RNE	Round to Nearest, ties to Even
001	RTZ	Round towards Zero
010	RDN	Round Down (towards $-\infty$ )
011	RUP	Round Up (towards $+\infty$ )
100	RMM	Round to Nearest, ties to Max Magnitude
101		<i>Invalid. Reserved for future use.</i>
110		<i>Invalid. Reserved for future use.</i>
111		In instruction's <i>rm</i> field, selects dynamic rounding mode; In Rounding Mode register, <i>Invalid</i> .

The automatically raised flags in the FCSR are as follows:

Flag Mnemonic	Flag Meaning
NV	Invalid Operation
DZ	Divide by Zero
OF	Overflow
UF	Underflow
NX	Inexact

Overflow occurs when a result is too large to represent in the supported format, due to limitations of floating-point representation (see relevant section below). Similarly, underflow occurs when the absolute value of the result is too small to be represented by the supported format.

The Inexact flag is raised occurs when the result is rounded.

Cache – cache memory is not a part of the RV architecture, but an addition, with various implementations. Different cache technologies allow architectures to plan and manufacture cache memory that would benefit their micro-architecture, and there is no single standard to logic or physical design of this part of memory.

Ram - Addresses compatible to 32- or 64-bit format respectively to RV32G and RV64G. Note that a register containing an address cannot be of a floating-point format register, but rather an address. Therefore, a distinct set of registers is required for operations requiring addresses, with compatibility to either 32- or 64-bit format.

### Floating point

The floating-point type has a twofold advantage over the integer type. It can represent numbers well over the simple binary range in a given number of bits and it can represent fractions. The disadvantage of floating points is that they are usually not completely accurate.

The floating point splits the bits into three parts: the sign (which is always one bit), the exponent (which length varies according to the type of floating-point precision), and the mantissa (also called the fraction, which length also varies). The exponent is represented in regular binary. The mantissa is a fraction, with each bit representing two to the negative power of the bit's significance. A base number is also assigned that corresponds to half the value of the upper range of the exponent.

The value of the floating point is calculated thus:



$$Value = (-1)^{Sign} \cdot 2^{Exponent - Base} \cdot (1 + Mantissa)$$

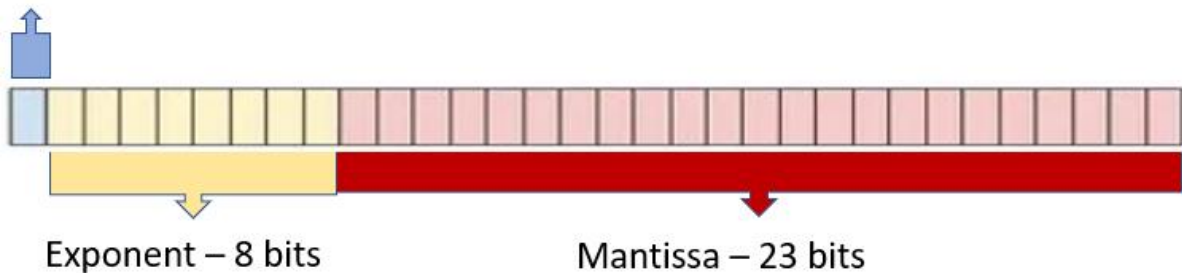
There are exceptional cases: if the exponent is at its highest value (all ones) then the floating point will represent infinite if the mantissa is all zeros, otherwise it will represent a non-existent value (NaN). Also, if the exponent is at its lowest value (all zeros), then the floating point will take on the value:

$$Value = (-1)^{Sign} \cdot 2^{-Base+1} \cdot Mantissa$$

It is understood that the hardware actions necessary for operations on floating points are different than those for integers, thereby needing a different control for the data path of the processor.

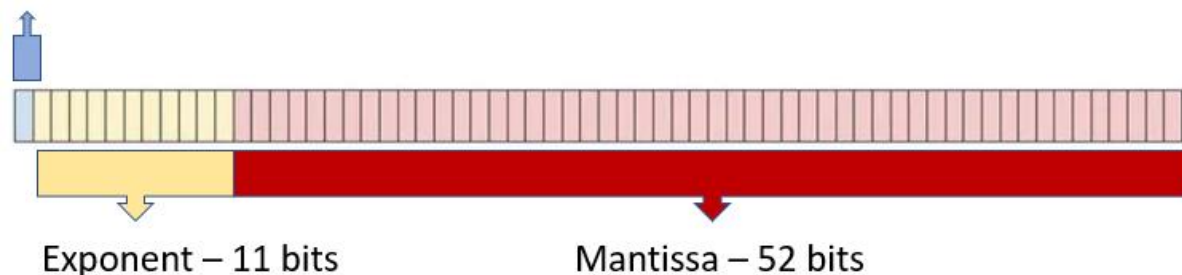
The format of a single-precision (32 bit) floating point is:

Sign – 1 bit



And the format for a double-precision (64 bit) floating point is:

Sign – 1 bit



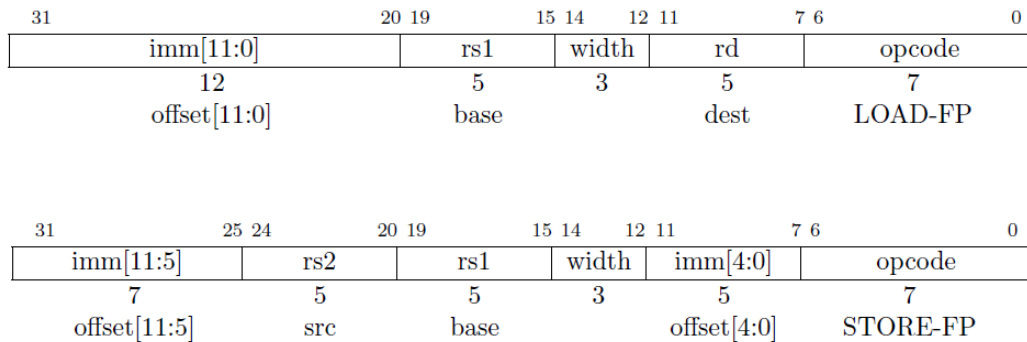
In the same vein, there is a quad-precision (128 bit) type of floating point, but it is less commonly used. Theoretically, there is no end to the progress towards greater and greater levels of precision, which might be utilized in the future.

### Implementing floating point instructions on RISC-V

All floating-point operations involve two registers at least. In some exceptional cases, the register-register operations involve more than two registers, though the common operations involve only two. All following operations contain 32-bit instruction; therefore, we will present both RV32G and RV64G at once.

As mentioned before, different floating-point representations vary from 32-bit to 128-bit, and so do the instructions. For different operations, we will declare the length of the used register by using different bits, as we will explain.

For load and store operations (memory operations):



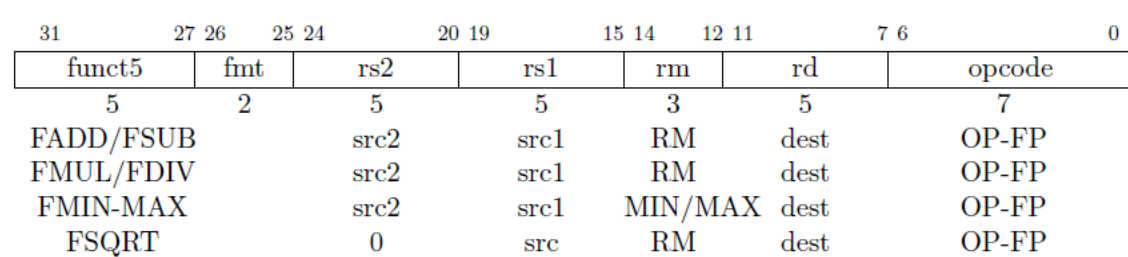
Where “width” field varies for different extensions (“F”/”S”, ”D”, ”Q”)

The table below gives us additional information about the op-codes for different formats in all the following instructions:

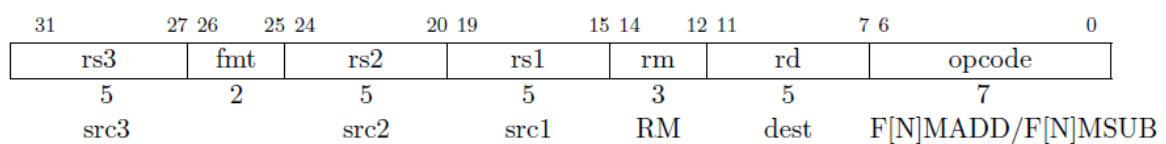
Fmt field	Mnemonic	Meaning
00	S	32-bit single-precision
01	D	64-bit double-precision
10	-	-
11	Q	128-bit quad-precision

Where the fmt field varies according to the relevant extension in the arithmetic operation.

The general form for the arithmetic operations is:



There is also a unique arithmetic operation for floating points that multiplies two numbers and then adds or subtracts a third number from the result. It is in a similar format as the other arithmetic instructions:



Where MADD stands for multiplication-addition and MSUB stands for multiplication-substitution.

In some cases, we need to be able to convert from a floating-point representation to an integer representation or vice versa. For that, we will use the different FCVT instructions:

31	27 26	25 24	20 19	15 14	12 11	7 6	0
funct5	fmt	rs2	rs1	rm	rd	opcode	
5	2	5	5	3	5	7	
FCVT.int.fmt		W[U]/L[U]	src	RM	dest	OP-FP	
FCVT.fmt.int		W[U]/L[U]	src	RM	dest	OP-FP	

If the floating-point number is out of range of the required integer, then the relevant flag will signal the exception and the integer will be set to one of the values according to the next table:

	FCVT.W.S	FCVT.WU.S	FCVT.L.S	FCVT.LU.S
Minimum valid input (after rounding)	$-2^{31}$	0	$-2^{63}$	0
Maximum valid input (after rounding)	$2^{31} - 1$	$2^{32} - 1$	$2^{63} - 1$	$2^{64} - 1$
Output for out-of-range negative input	$-2^{31}$	0	$-2^{63}$	0
Output for $-\infty$	$-2^{31}$	0	$-2^{63}$	0
Output for out-of-range positive input	$2^{31} - 1$	$2^{32} - 1$	$2^{63} - 1$	$2^{64} - 1$
Output for $+\infty$ or NaN	$2^{31} - 1$	$2^{32} - 1$	$2^{63} - 1$	$2^{64} - 1$

Similarly, we can convert double to single-precision, quad to double-precision, quad to single-precision and vice versa using the following operands:

31	27 26	25 24	20 19	15 14	12 11	7 6	0
funct5	fmt	rs2	rs1	rm	rd	opcode	
5	2	5	5	3	5	7	
FCVT.S.D	S	D	src	RM	dest	OP-FP	
FCVT.D.S	D	S	src	RM	dest	OP-FP	

31	27 26	25 24	20 19	15 14	12 11	7 6	0
funct5	fmt	rs2	rs1	rm	rd	opcode	
5	2	5	5	3	5	7	
FCVT.S.Q	S	Q	src	RM	dest	OP-FP	
FCVT.Q.S	Q	S	src	RM	dest	OP-FP	
FCVT.D.Q	D	Q	src	RM	dest	OP-FP	
FCVT.Q.D	Q	D	src	RM	dest	OP-FP	

In all conversions the result will be rounded according to the rm field, just as in arithmetic operations. Note that quad-precision is more precise than both double and single-precision, and double-precision is more precise than single-precision, therefore rounding will be essential when reducing precision.

Comparing floats is done using the FCMP instruction. It compounds three different comparison operations: marked with EQ (equal) LT (less-then) and LE (less or equal):

31	27 26	25 24	20 19	15 14	12 11	7 6	0
funct5	fmt	rs2	rs1	rm	rd	opcode	
5	2	5	5	3	5	7	
FCMP		src2	src1	EQ/LT/LE	dest	OP-FP	

The sign injunction instruction FSGNJ takes the mantissa and exponent bits from the first register but will take the sign bit according to the RM field. In J mode the second register's

sign bit is used, in JN mode the second register's not-sign bit is used, and in JX mode the sign will be the result of a xor operation between the two registers' signs. The instruction is defined to all three extensions ("F", "D", "Q") and is of the form:

31	27 26	25 24	20 19	15 14	12 11	7 6	0
funct5	fmt	rs2	rs1	rm	rd	opcode	
5	2	5	5	3	5	7	
FSGNJ		src2	src1	J[N]/JX	dest	OP-FP	

The classify instruction FCLASS examines a floating point register to determine its class and marks a specific bit in the destination register to mark the register's class according to the table:

rd bit	Meaning
0	<i>rs1</i> is $-\infty$ .
1	<i>rs1</i> is a negative normal number.
2	<i>rs1</i> is a negative subnormal number.
3	<i>rs1</i> is $-0$ .
4	<i>rs1</i> is $+0$ .
5	<i>rs1</i> is a positive subnormal number.
6	<i>rs1</i> is a positive normal number.
7	<i>rs1</i> is $+\infty$ .
8	<i>rs1</i> is a signaling NaN.
9	<i>rs1</i> is a quiet NaN.

The general form of the instruction is:

31	27 26	25 24	20 19	15 14	12 11	7 6	0
funct5	fmt	rs2	rs1	rm	rd	opcode	
5	2	5	5	3	5	7	
FCLASS		0	src	001	dest	OP-FP	

In our project, we implemented a RISC-V Floating-Point CPU supporting both arithmetic operations and logic operations (such as FCMP and MIN/MAX).

## CPU flow

The full flow for an instruction, from reading it in the instruction memory to writing its result either in the system memory or the register file, was not integral to our project. We focused more on the aspect of the actual operations on floating point registers. However, we did implement a full CPU in the reference model and the hardware.

As a reference model for testing our hardware, we implemented a software model of a full RISC-V pipelined CPU. The model was written in Python3.

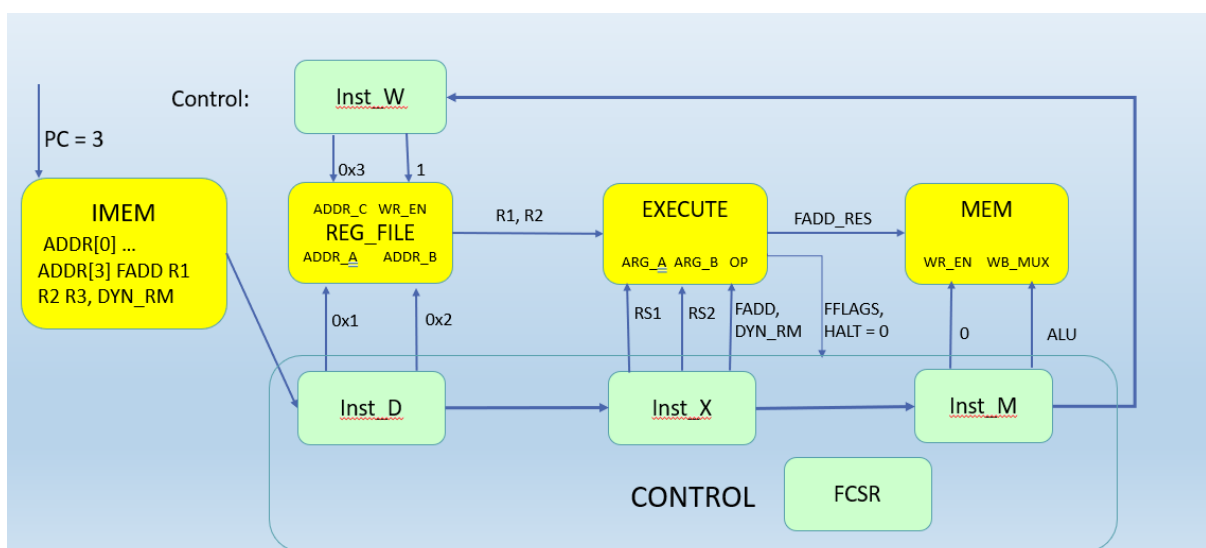
We simulated in our model and finally implemented in our hardware the following five stages for the micro-processor:

- Instruction fetch
- Instruction decode
- Execute
- Memory
- Write back

In addition, we also implemented a control unit, which enabled the work of the software model. The control logic included a special operation which enabled a “halt” for all the CPU units except the ALU, in order to implement operations that required multiple clock cycles (for example SQRT). That way, we entered “bubbles” all along the pipeline while long operations were executed.

Moreover, we implemented a register file that represented floating-points bitwise. By doing so, we were able to access every bit of data, so comparing the results of the software model and the results of the hardware implantation were significantly more accessible and the work of verification more efficient. The instructions themselves were also implemented bitwise in the simulation.

As an example, the full flow of an FADD instruction would go as follows:



The PC counter automatically increments by 4 at every clock (unless stopped by the control). When the PC reaches the FADD instruction address, the instruction is read into the control unit. In the following cycle, the control unit gives the addresses for RS1 and RS2 to the REG\_FILE to be read. Next, it gives 'RS1' and 'RS2' to the MUXs that determine the arguments going into the ALU and sends the operation type 'FADD' and the RM 'DYN\_RM' to the ALU. The DYN\_RM itself is read from the FCSR. The ALU outputs the operation result, and flags for the FCSR that were raised during the operation, such as the OF flag. Next, the control sends '0' to the WR\_EN for the system memory (we only want to write into a register) and 'ALU' for the WB into the REG\_FILE. Finally, the control sends the address for 'RD' and '1' for the 'WR\_EN' to the REG\_FILE.

After finishing both reference model and hardware planning, we compared the results of each test for the hardware and verified it with the software model.

## Verifying software reference model

In order to verify the correctness of the software reference model, we used known calculations that require a large amount of operations. By showing that those calculation converge to the manually-computed value we deduced that the model is correct.

Taylor expansions and geometric series are a convenient way to get such tests. The solution can be calculated by a long series of calculations, or by an easily solved formula. To test all our mathematical operations, we chose one Taylor expansion and two series, which covered all our operations. The massive repetition of calculations needed in each expansion or series was crucial to the test, due to the diversity of cases it presented.

These are the expansions and series we implemented:

1.

$$\text{Exponent: } e^x = \sum_{n=0}^{\infty} \frac{x^n}{n!} = 1 + x + \frac{x^2}{2!} + \frac{x^3}{3!} + \dots$$

This test checks addition (meaning same signs), multiplication (for x different than 1) and division.

Test for x=1 (basic) and for x=1.11

Implemented by:

```
ADDi R1 R0 x //R1 is forever x
```

```
ADDi R2 R0 1 //current element, starts as 1
```

```
ADDi R3 R0 1 //R3 = 1//iterator, starts as 1
```

```
ADDi R4 R0 1 //R4 = 1//overall sum, starts as 1
```

```
ADDi R5 R0 1 //R5 is forever 1
```

Loop:

```
Mul R2 R2 R1 //R2 = R2*R1
```

```
Div R2 R2 R3 //R2 = R2/R3
```

```
Add R4 R4 R2 //R4+=R2
```

```
Add R3 R3 R5 //R3++
```

Simulation Result (with x=1, 20 iterations):

Final result: 2.7182819843292236

Actual value: 2.718281828...

2.

Infinite geometric series (with sqrt):  $\sum_{n=0}^{\infty} \sqrt{a}^n = \sum_{n=0}^{\infty} (\sqrt{a})^n = 1 + \sqrt{a} + \sqrt{a}^2 + \dots = \frac{1}{1-\sqrt{a}}$

This test checks addition (meaning same signs) and sqrt.

Implemented by:

ADDI R1 R0 x //R1 is forever x = a

ADD R2 R1 R0 // (current element) ^2, starts as x

//no R3 yet

ADDI R4 R0 1 //R4 = 1//overall sum, starts as 1

Loop:

Sqrt R3 R2 //R3 = sqrt(R2)

Mul R2 R2 R1 //R2 = R2\*x

Add R4 R4 R3 //R4+=R3

Simulation Result (using a=0.25, 50 iterations):

Final result: 1.9999998807907104

Actual value: 2

3.

Infinite geometric series (with an alternating sign):  $\sum_{n=0}^{\infty} (-|a|)^n = 1 - |a| + |a|^2 - \dots = \frac{1}{1-(-|a|)} = \frac{1}{1+|a|}$

Checks addition (meaning same signs) and subtraction (meaning different signs, only for  $a < 0$ ).

Implemented by:

ADDI R1 R0 x //R1 is forever x

ADD R2 R1 R0 //|current element|, starts as x

//no R3

ADDI R4 R0 1 //R4 = 1//overall sum, starts as 1



Loop:

Sub R4 R4 R2

Mul R2 R2 R1

Add R4 R4 R2

Mul R2 R2 R1

Simulation result (using  $a = 0.25$ , 20 iterations):

Final result: 0.800000011920929

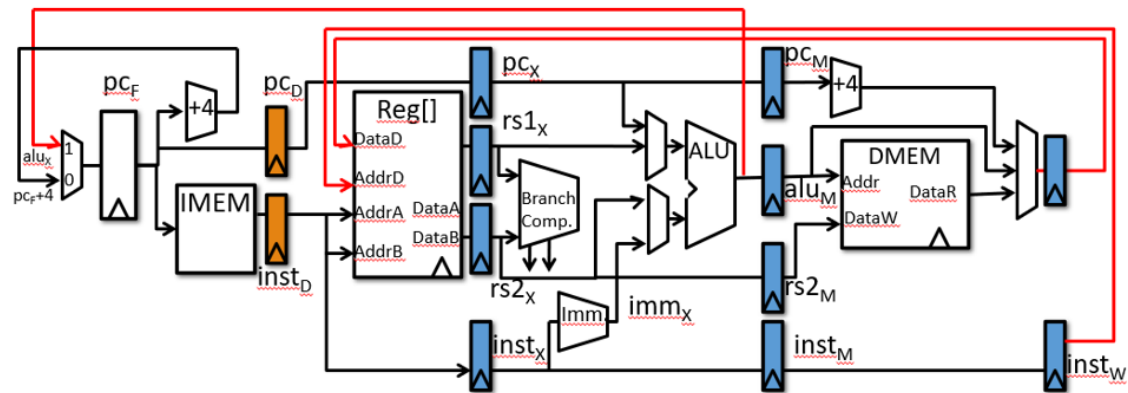
Actual value: 0.8

Note that the value is never the exact mathematical value, but it is still correct for the following reasons:

- Taylor expansions, as well as the sum of geometric series, is only equal to the mentioned value for  $\#iterations \rightarrow \infty$ , hence at the limit. Therefore, we should not expect it to converge to the exact value, but it should be close enough.
- The precision of FP-32 (and any representation actually) is limited. In our case, FP-32 can be accurate at about 7-8 digits (varies between different cases), and therefore our conversion is good enough, and achieved the needed result. We would like to mention that the error in its Lagrange form gives us the boundaries of the error, and our error was always in between those boundaries.

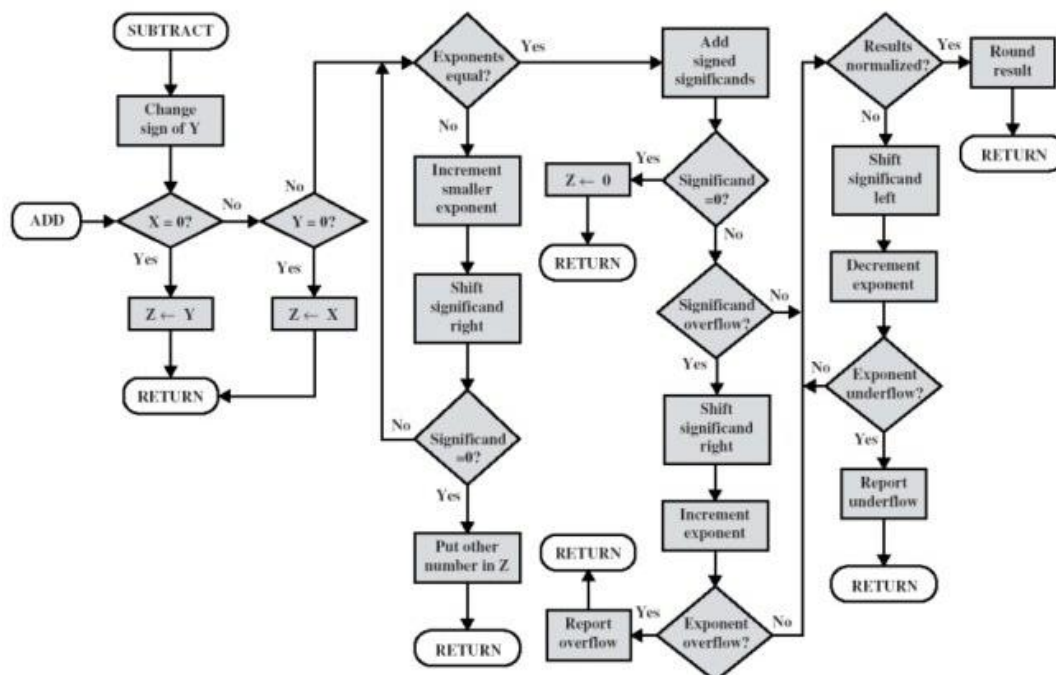
As for the non-arithmetic operations, those were checked separately with different inputs in order to achieve full coverage.

We implemented a five-stage pipelined RISC-V FP-32 CPU with the following design:

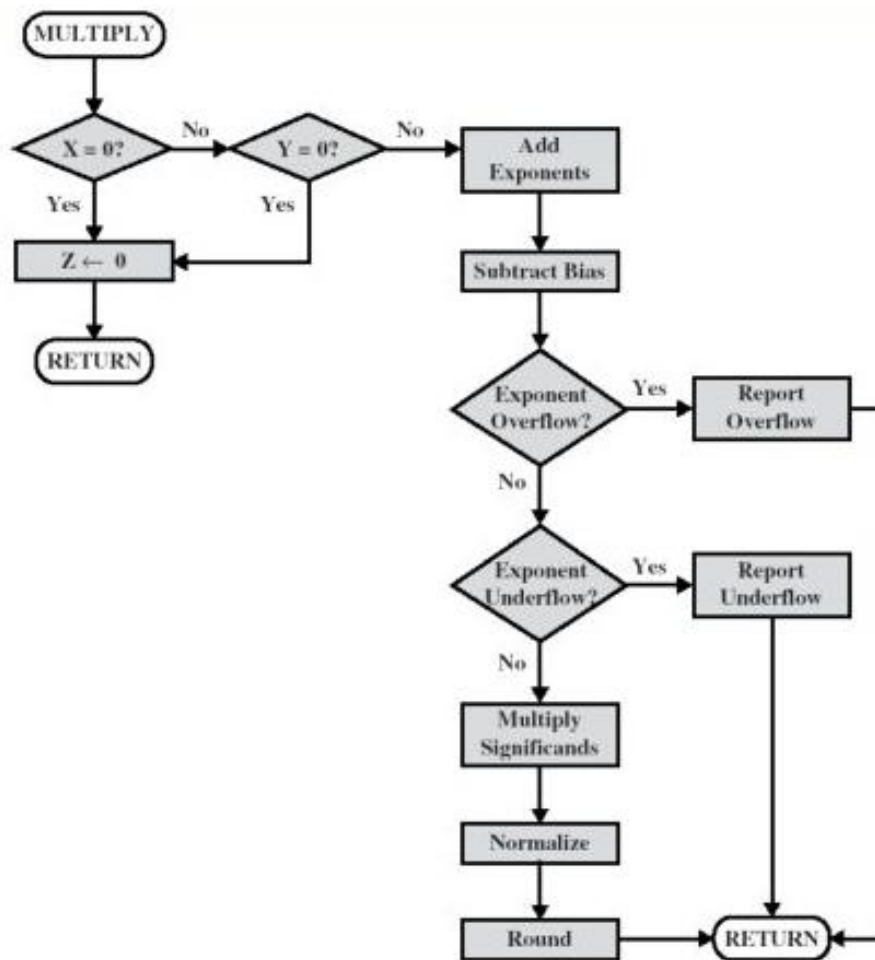


We implemented the following arithmetic operations:

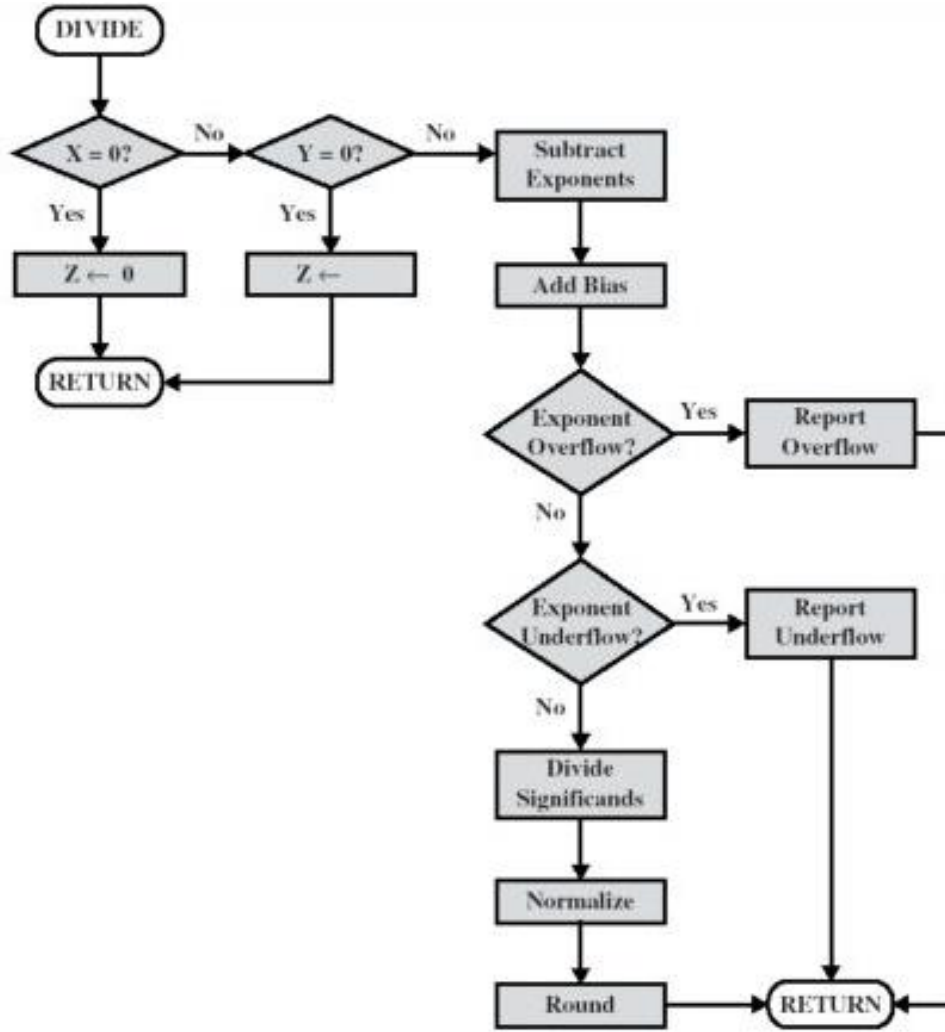
### Addition and Subtraction:



**Multiplication:**



Division:



Both the addition and the subtraction models presented a challenge: to find the first bit that is “one” using hardware components is not so simple. To overcome this problem, we chose to implement hardware that is essentially a giant MUX. This forwards the proper bits to the final mantissa according to where the first “one” in the calculated mantissa is. To facilitate the RTL writing we wrote a script that automatically writes System-Verilog code for us.

#### SQRT:

The square-root (SQRT) was implemented as a multi-clock operation, implemented by a Taylor series with no error within the resolution of single-precision registers. We used the following formula:

$$\sqrt{b} = Y_0 \left( 1 - \frac{1}{2} \left( 1 - \frac{b}{Y_0^2} \right) - \frac{1}{8} \left( 1 - \frac{b}{Y_0^2} \right)^2 - \frac{1}{16} \left( 1 - \frac{b}{Y_0^2} \right)^3 - \frac{5}{128} \left( 1 - \frac{b}{Y_0^2} \right)^4 - \frac{7}{256} \left( 1 - \frac{b}{Y_0^2} \right)^5 \right)$$

In our case, we chose  $Y_0$  to be half of the exponent, because it is a good approximation of the expected result, and therefore a good point for the Taylor expansion (the simplicity of

generating a close approximation of a square root is one of the great advantages of floating-point architecture).

Although  $\left(1 - \frac{b}{Y_0^2}\right)$  seems to use a division, this can be implemented by subtraction of exponents alone, because  $Y_0^2$  consists only of an exponent twice as big as  $Y_0$ , hence is the exact exponent of the input number or smaller by 1. By knowing that, we could save resources and time of execution.

Moreover, the numbers  $\left\{\frac{5}{128}, \frac{7}{256}\right\}$  are constants that are frequently used, and therefore hard coded to the module to save time and energy for re-use.

Other than that, we used our multiplication module to get  $\left(1 - \frac{b}{Y_0^2}\right)^n$ ,  $n \in \{2,3,4,5\}$  and our subtraction to accumulate the results.

The final multiplication with  $Y_0$  can also be implemented with exponent addition, because once again it is a number consisting of an exponent only.

After calculating the Lagrange error and the terms of the Taylor expansion, we concluded that six cycles of the Execution unit were the minimum needed to output an error-less square root value. The determined stages go as follows:

1. Calculation of  $Y_0$  and  $Y_0^2$ , division of  $\frac{b}{Y_0^2}$  and subtraction of  $1 - \frac{b}{Y_0^2}$ . This will be the basis of all the following calculations.
2. Calculation of  $\frac{1}{8}\left(1 - \frac{b}{Y_0^2}\right)^2$  and  $1 - \frac{1}{2}\left(1 - \frac{b}{Y_0^2}\right)$
3. Calculation of  $\frac{1}{16}\left(1 - \frac{b}{Y_0^2}\right)^3$  and  $1 - \frac{1}{2}\left(1 - \frac{b}{Y_0^2}\right) - \frac{1}{8}\left(1 - \frac{b}{Y_0^2}\right)^2$
4. Calculation of  $\frac{5}{128}\left(1 - \frac{b}{Y_0^2}\right)^4$  and  $1 - \frac{1}{2}\left(1 - \frac{b}{Y_0^2}\right) - \frac{1}{8}\left(1 - \frac{b}{Y_0^2}\right)^2 - \frac{1}{16}\left(1 - \frac{b}{Y_0^2}\right)^3$
5. Calculation of  $\frac{7}{256}\left(1 - \frac{b}{Y_0^2}\right)^5$  and  $1 - \frac{1}{2}\left(1 - \frac{b}{Y_0^2}\right) - \frac{1}{8}\left(1 - \frac{b}{Y_0^2}\right)^2 - \frac{1}{16}\left(1 - \frac{b}{Y_0^2}\right)^3 - \frac{5}{128}\left(1 - \frac{b}{Y_0^2}\right)^4$
6. Calculation of the final result:

$$\sqrt{b} = Y_0 \left( 1 - \frac{1}{2}\left(1 - \frac{b}{Y_0^2}\right) - \frac{1}{8}\left(1 - \frac{b}{Y_0^2}\right)^2 - \frac{1}{16}\left(1 - \frac{b}{Y_0^2}\right)^3 - \frac{5}{128}\left(1 - \frac{b}{Y_0^2}\right)^4 - \frac{7}{256}\left(1 - \frac{b}{Y_0^2}\right)^5 \right)$$

We chose to use multiplication for the final result, because the multiplying module was already in place at the SQRT module.

Note that while the SQRT is working, the control halts the whole pipeline.

Non-arithmetic operations:

In addition to the arithmetic operations, we also implemented the MIN-MAX operation and the Comparison operation. Both consist of the following logic, with different outputs:

If the sign of the inputs is different, the negative is the smaller, and the positive is the larger.

Else, If the exponent of the inputs is different, the one with the larger exponent is the larger in case of common positive sign and the smaller in case of common negative sign.

Else, If the mantissa of the inputs is different, the one with the larger mantissa is the larger in case of common positive sign and the smaller in case of common negative sign

Else, the two inputs are equal.

We used a few flags:

- Equal
- Less than

Those two alone are set by the logic presented above; they determine the output.

The compare function has three rounding modes:

- LE (less or equal)
- LT (less than)
- EQ (equal)

The output is determined by the mode and the two flags mentioned above, where LE check for Equal **or** less than while the other two check a single flag alone.

The output would be accordingly 1 if true or 0 if false.

For MIN-MAX, a similar logic to what we presented will choose the correct register input and send it to the output instead of a binary result. In case of two equal inputs, the output can be either one of the inputs.

## Hardware tests

ADD:

Addition, without needing to add to the larger exponent, where the second operand is larger because of its exponent:

```
first arg: [0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 1, 1, 1, 1, 1]
second arg: [0, 0, 0, 0, 0, 0, 0, 0, 1, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 1, 1]
result: [0, 0, 0, 0, 0, 0, 0, 0, 1, 1, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 1, 0, 0, 1, 0]
```

[illegible]




Addition, with needing to add to the larger exponent, because of the exponent, where the first is larger because of its mantissa:

```
first arg: [0, 0, 0, 0, 0, 0, 0, 0, 1, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 1, 1, 1, 1]
second arg: [0, 0, 0, 0, 0, 0, 0, 0, 1, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 1, 1]
result: [0, 0, 0, 0, 0, 0, 0, 1, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 1, 0, 0, 1, 0]
```

first_operand	000000001000000000000000001111
second_operand	000000001000000000000000000011
calculation output	00000000100000000000000000001001

Addition, with needing to add to the larger exponent, because of the mantissa (this is not a special case, but we will include it anyway), where the first is larger because of its exponent:

```
first arg: [0, 0, 0, 0, 0, 0, 0, 1, 0, 1, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0]
second arg: [0, 0, 0, 0, 0, 0, 0, 1, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0]
result: [0, 0, 0, 0, 0, 0, 0, 1, 1, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0]
```

	> first_operand	00000001010000000000000000000000
	> second_operand	00000000100000000000000000000000
	> calculation_output	00000001100000000000000000000000

No change, because the first operand is more than 24 powers of 2 larger than the second operand:

[illegible]

first_operand	00001100100000000000000000000000
second_operand	00000000100000000000000000000000
calculation_output	00001100100000000000000000000000

On the verge of there being no change:

```
first arg: [0, 0, 0, 0, 1, 1, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0]
second arg: [0, 0, 0, 0, 0, 0, 0, 0, 1, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0]
result: [0, 0, 0, 0, 1, 1, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 1]
```

in	> first_operand	00001100000000000000000000000000
in	> second_operand	00000000100000000000000000000000
out	> calculation_output	00001100000000000000000000000001

Overflow of the exponent:

```
first arg: [0, 1, 1, 1, 1, 1, 1, 1, 1, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0]
second arg: [0, 1, 1, 1, 1, 1, 1, 1, 1, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0]
result: [0, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1]
```

in	> first_operand	01111111100000000000000000000000
in	> second_operand	01111111100000000000000000000000
out	> calculation_output	01111111111111111111111111111111

Subtraction, where the larger exponent remains the same:

```
first arg: [0, 0, 0, 0, 0, 0, 0, 0, 1, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0]
second arg: [1, 0, 0, 0, 0, 0, 0, 1, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0]
result: [1, 0, 0, 0, 0, 0, 0, 1, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0]
```

in	> first_operand	00000000100000000000000000000000
in	> second_operand	10000010010000000000000000000000
out	> calculation_output	10000010001100000000000000000000

Subtraction, where the larger exponent falls by 1:

```
first arg: [0, 0, 0, 0, 0, 0, 0, 1, 0, 1, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0]
second arg: [1, 0, 0, 0, 0, 0, 0, 1, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0]
result: [0, 0, 0, 0, 0, 0, 0, 0, 1, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0]
```

in	> first_operand	00000001010000000000000000000000
in	> second_operand	10000001000000000000000000000000
out	> calculation_output	00000000100000000000000000000000

```
first arg: [0, 1, 1, 1, 1, 0, 0, 0, 1, 1, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0]
second arg: [1, 1, 1, 1, 1, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 1]
result: [0, 1, 1, 1, 1, 0, 0, 0, 0, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1]
```

in	> first_operand	01111000110000000000000000000000
in	> second_operand	11111000000000000000000000000001
out	> calculation_output	01111000011111111111111111111111

Subtraction, where the larger exponent falls by 5 (arbitrarily):

```
first arg: [0, 0, 0, 0, 0, 1, 1, 1, 1, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0]
second arg: [1, 0, 0, 0, 0, 1, 1, 1, 1, 1, 1, 1, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0]
result: [0, 0, 0, 0, 0, 1, 0, 1, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0]
```

in	> first_operand	00000111100000000000000000000000
in	> second_operand	10000111011110000000000000000000
out	> calculation_output	00000101000000000000000000000000

Subtraction, where the mantissa falls to zero because of the sum of the mantissas is zero:

```
first arg: [0, 1, 1, 1, 1, 0, 0, 0, 1, 1, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0]
second arg: [1, 1, 1, 1, 1, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0]
result: [0, 1, 1, 1, 1, 0, 0, 0, 1, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0]
```





MUL:

Infinity:

```
first arg: [0, 1, 1, 1, 1, 1, 1, 1, 1, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 1, 1, 1, 1]
second arg: [0, 1, 1, 1, 1, 1, 1, 1, 1, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 1, 1, 1, 1]
result:     [0, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1]
```

		ops	
out	> calculation_output	B 011111...	01111111111111111111111111111111
in	> first_operand	B 011111...	011111111000000000000000000001111
in	> second_operand	B 011111...	011111111000000000000000000001111

Zero:

```
first arg: [0, 1, 1, 1, 1, 1, 1, 1, 1, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 1, 1, 1, 1]
second arg: [0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0]
result:     [0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0]
```

	Name	ops	
out	> calculation_output	B ...	00000000000000000000000000000000
in	> first_operand	B ...	011111111000000000000000000001111
in	> second_operand	B ...	00000000000000000000000000000000

Infinity X zero:

```
first arg: [0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0]
second arg: [0, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1]
result:     [0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0]
```




out	> calculation_output		00000000000000000000000000000000
in	> first_operand		00000000000000000000000000000000
in	> second_operand		01111111111111111111111111111111

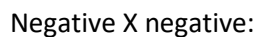
Big X small:

```
first arg: [0, 1, 1, 1, 1, 1, 1, 1, 1, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 1, 1, 1, 1]
second arg: [0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 1]
result:     [0, 1, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 1, 0, 0, 0]
```

		ops	
out	> calculation_output	B ...	010000000000000000000000000010000
in	> first_operand	B ...	011111111000000000000000000001111
in	> second_operand	B ...	000000000000000000000000000000001

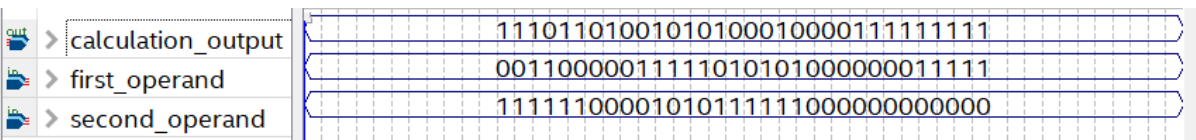
Small X small = zero

	Name	0 ps	160,0 ns	320,0 ns	480,0 ns	640,0 ns	800,0 ns	960,0 ns
		0 ps						
	> calculation_output	00000000000000000000000000000000						
	> first_operand	000010000000000000000000000000001111						
	> second_operand	000000000000000000000000000000000001						

[illegible]

> calculation_output	01100001001010100010000111101011
> first_operand	10111100011111010101000000000001
> second_operand	11100100001010111111000000000000

```
first arg: [0, 0, 1, 1, 0, 0, 0, 0, 0, 1, 1, 1, 1, 0, 0, 1, 0, 1, 0, 1, 0, 0, 0, 0, 0, 0, 1, 1, 1, 1, 1]
second arg: [1, 1, 1, 1, 1, 1, 0, 0, 0, 0, 1, 0, 1, 0, 1, 1, 1, 1, 1, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0]
result:     [1, 1, 1, 0, 1, 1, 0, 1, 0, 0, 1, 0, 1, 0, 1, 0, 0, 0, 1, 0, 0, 0, 0, 1, 1, 1, 1, 1, 1, 1, 1, 1]
```



```
first arg: [0, 0, 1, 1, 1, 1, 1, 1, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0]
second arg: [0, 1, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0]
result:     [0, 0, 1, 1, 1, 1, 1, 1, 1, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0]
```

	Name	Value at 0 ps	0 ps	320,0 ns	640,0 ns	960,0 ns
			0 ps			
out	> calculation_output	B 001111...		00111111100000000000000000000000		
in	> first_operand	B 001111...		00111111100000000000000000000000		
in	> second_operand	B 010000...		01000000000000000000000000000000		

Regular multiplication of 13.78 and 207.13 with the expected result of 2854.2515:

```
first arg: [0, 1, 0, 0, 0, 0, 0, 1, 0, 1, 0, 1, 1, 1, 0, 0, 0, 1, 1, 1, 1, 0, 1, 0, 1, 1, 1, 0, 0, 0, 0, 1]
second arg: [0, 1, 0, 0, 0, 0, 1, 1, 0, 1, 0, 0, 1, 1, 1, 1, 0, 0, 1, 0, 0, 0, 0, 1, 0, 1, 0, 0, 1, 0, 0, 0]
result:    [0, 1, 0, 0, 0, 1, 0, 1, 0, 0, 1, 1, 0, 0, 1, 0, 0, 1, 1, 0, 0, 1, 0, 0, 0, 0, 0, 0, 0, 0, 1, 0, 1]
```

out	> calculation_output	B.	01000101001100100110010000000101
in	> first_operand	B.	01000001010111000111101011100001
in	> second_operand	B.	01000011010011110010000101001000

Cutting the mantisa:

The multiplication of  $1.5621186 \cdot 10^{-2}$  with  $1.5621185 \cdot 10^{-2}$  should yeild  $2.440214364 \cdot 10^{-4}$ ,

But instead, it will yield a smaller value ( $2.4402143 \cdot 10^{-4}$ ) due to the finite representation span of the matissa bits.

```
first arg: [0, 0, 1, 1, 1, 1, 0, 0, 0, 1, 1, 1, 1, 1, 1, 1, 1, 1, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 1]
second arg: [0, 0, 1, 1, 1, 1, 0, 0, 0, 1, 1, 1, 1, 1, 1, 1, 1, 1, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0]
result:    [0, 0, 1, 1, 1, 1, 0, 0, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0]
```

out	> calculation_output		00111001011111111110000000000001
in	> first_operand		00111100011111111111000000000001
in	> second_operand		00111100011111111111000000000000

DIV:

Divide a number by itself results in 1:

```
first arg: [0, 1, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 1, 0, 0, 0, 0, 1, 0, 0, 0, 0, 0, 1, 1, 0]
second arg: [0, 1, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 1, 0, 0, 0, 0, 1, 0, 0, 0, 0, 0, 1, 1, 0]
result:    [0, 0, 1, 1, 1, 1, 1, 1, 1, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0]
```

out	> calculation_output		00111111100000000000000000000000
in	> first_operand		010000000000000000010000100000110
in	> second_operand		010000000000000000010000100000110

Divide a negative number by itself results as 1:

```
first arg: [1, 1, 1, 1, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 1, 0, 0, 0, 0, 1, 0, 0, 0, 0, 0, 1, 1, 0]
second arg: [1, 1, 1, 1, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 1, 0, 0, 0, 0, 1, 0, 0, 0, 0, 0, 1, 1, 0]
result:    [0, 0, 1, 1, 1, 1, 1, 1, 1, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0]
```

out	> calculation_output	00111111100000000000000000000000
in	> first_operand	1111000000000000000010000100000110
in	> second_operand	1111000000000000000010000100000110

Divide a positive number with a negative with same absolute value:

```
first arg: [1, 1, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 1, 0, 0, 0, 0, 1, 0, 0, 0, 0, 0, 1, 1, 0]
second arg: [0, 1, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 1, 0, 0, 0, 0, 1, 0, 0, 0, 0, 0, 1, 1, 0]
result:    [1, 0, 1, 1, 1, 1, 1, 1, 1, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0]
```

out	> calculation_output	10111111100000000000000000000000
in	> first_operand	1100000000000000000010000100000110
in	> second_operand	0100000000000000000010000100000110

Divide a positive number with a negative with same absolute value, reversed:

```
first arg: [0, 1, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 1, 0, 0, 0, 0, 1, 0, 0, 0, 0, 0, 1, 1, 0]
second arg: [1, 1, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 1, 0, 0, 0, 0, 1, 0, 0, 0, 0, 0, 1, 1, 0]
result:    [1, 0, 1, 1, 1, 1, 1, 1, 1, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0]
```

out	> calculation_output	10111111100000000000000000000000
in	> first_operand	0100000000000000000010000100000110
in	> second_operand	1100000000000000000010000100000110

Positive infinite result:(very large number divided by very small number):

```
first arg: [0, 1, 1, 1, 1, 1, 1, 1, 1, 1, 0, 0, 0, 0, 0, 0, 0, 0, 1, 0, 0, 0, 0, 1, 0, 0, 0, 1, 0, 0, 0, 1]
second arg: [0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 1, 0, 1, 0, 0, 0, 0, 1, 0, 0, 1, 0, 0, 1, 1, 0]
result:    [0, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1]
```

out	> calculation_output	01111111111111111111111111111111
in	> first_operand	01111111110000000010000100010001
in	> second_operand	0000000000000000001010000100100110

Positive zero result:(very small number divided by very large number)

```
first arg: [0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 1, 0, 1, 0, 0, 0, 0, 1, 0, 0, 1, 0, 0, 1, 1, 0]
second arg: [0, 1, 1, 1, 1, 1, 1, 1, 1, 1, 0, 0, 0, 0, 0, 0, 0, 0, 1, 0, 0, 0, 0, 1, 0, 0, 0, 1, 0, 0, 0, 1]
result:    [0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0]
```

out	> calculation_output	00000000000000000000000000000000
in	> first_operand	0000000000000000001010000100100110
in	> second_operand	01111111110000000010000100010001

Zero divide by any number:

Variable	Value (Hex)
calculation_output	00000000000000000000000000000000
first_operand	00000000000000000000000000000000
second_operand	01111000110000000010110100010001

[illegible]

```
first arg: [0, 1, 1, 1, 1, 1, 0, 0, 0, 0, 0, 0, 0, 0, 0, 1, 0, 0, 1, 0, 0, 0, 0, 0, 0, 0, 1, 1]
second arg: [0, 1, 1, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 1, 1, 0, 0, 0, 1, 0, 0, 1, 0, 1, 0, 0, 0, 0]
result:     [1, 1, 0, 1, 1, 0, 1, 1, 0, 1, 1, 1, 0, 1, 0, 1, 0, 1, 1, 0, 0, 0, 1, 1, 1, 0, 1, 1, 1, 1]
```

```
first arg: [1, 1, 1, 1, 1, 1, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 1, 0, 0, 1, 0, 0, 0, 0, 0, 0, 0, 0, 1, 1]
second arg: [0, 1, 1, 1, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 1, 1, 1, 1, 0, 1, 0, 0, 0, 0, 0, 0, 0]
result:     [1, 1, 0, 0, 1, 0, 1, 1, 1, 0, 0, 0, 0, 0, 0, 0, 1, 0, 1, 0, 0, 1, 0, 1, 1, 0, 1, 1, 0, 1, 1]
```

```
first arg: [1, 1, 1, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 1, 0, 0, 1, 0, 0, 0, 0, 0, 0, 0, 0, 0, 1, 1]
second arg: [0, 1, 1, 1, 1, 1, 1, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 1, 1, 1, 1, 0, 1, 0, 0, 0, 0, 0, 0, 0, 0]
result:     [1, 0, 1, 0, 1, 0, 0, 1, 1, 0, 0, 0, 0, 0, 0, 0, 0, 1, 0, 1, 0, 0, 1, 0, 1, 1, 0, 1, 1, 0, 1, 1]
```

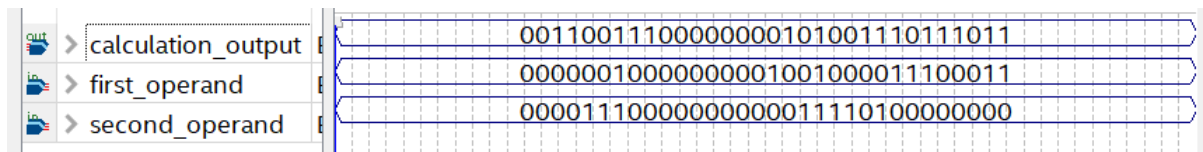
30



```

first arg: [0, 0, 0, 0, 0, 0, 1, 0, 0, 0, 0, 0, 0, 0, 0, 1, 0, 0, 1, 0, 0, 0, 0, 0, 0, 0, 0, 1, 1]
second arg: [0, 0, 0, 0, 1, 1, 1, 0, 0, 0, 0, 0, 0, 0, 0, 0, 1, 1, 1, 1, 0, 1, 0, 0, 0, 0, 0, 0, 0]
result:    [1, 0, 1, 1, 0, 0, 1, 1, 1, 0, 0, 0, 0, 0, 0, 0, 1, 0, 1, 0, 0, 1, 1, 1, 0, 1, 1, 1, 0, 1, 1]

```



## SQRT:

Square root of 4047.891845703125 = 63.62304493

Test results:

```

reg_orig      = 0100010101111100111111001000101
end_of_first  = 10111100010000101011100000000000
end_of_second = 00111111100000001100001010111000
end_of_third  = 00111111100000001100001000100100
end_of_fourth = 00111111100000001100001000100100
end_of_fifth  = 00111111100000001100001000100100
final         = 01000010011111100111110111111101

```

Design result:

Original: B 0100010101111100111111001000101

1: B 10111100010000101011100000000000

2: B 00111111100000001100001010111000

3: B 00111111100000001100001000100100

4: B 00111111100000001100001000100100

5: B 00111111100000001100001000100100

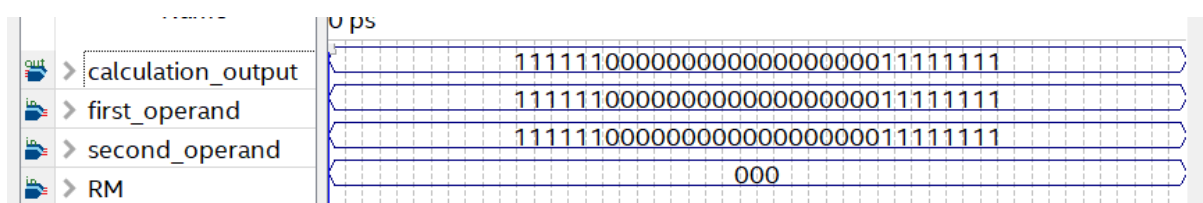
Final: B 01000010011111100111110111111101

The final result is: 63.6230...

MIN\_MAX:

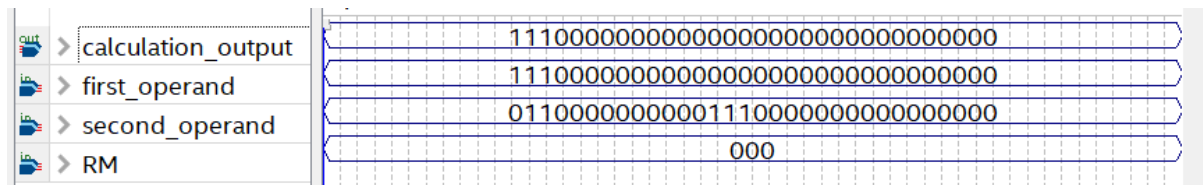
**FMIN: (RM=000)**

Same number:



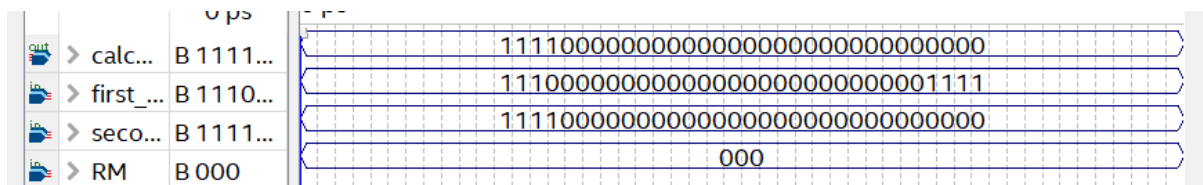
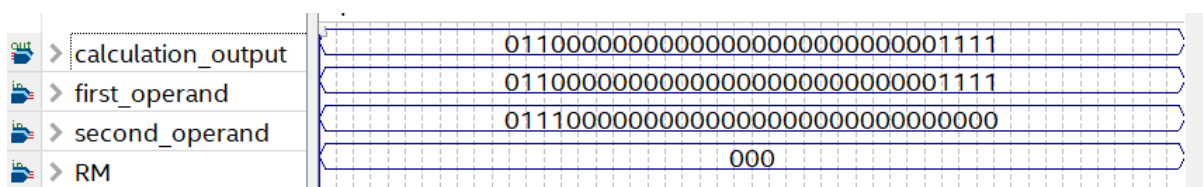
[illegible]

Negative and positive:

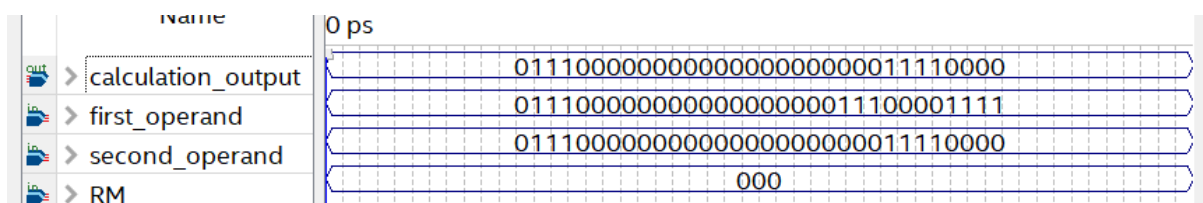
[illegible]

Same sign:

Determine by exponent:

[illegible][illegible]

Determine by mantisa:

[illegible]



[illegible]

Same number:

[illegible]

```
result:      [0, 1, 1, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 1, 1, 1, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0]
first arg:   [1, 1, 1, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0]
second arg:  [0, 1, 1, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 1, 1, 1, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0]
```

Determine by exponent:

[illegible]



equal:

### Less Than: (RM=001)

Determine only by sign:

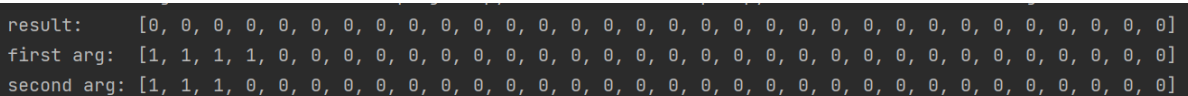
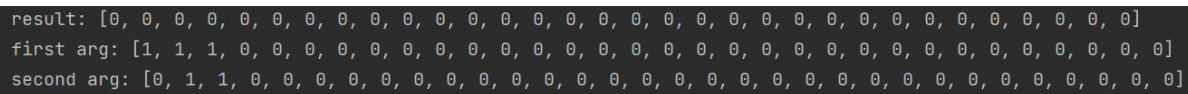
Determine by exponent (for same sign):

Determine by mantisa (for same sign and exponent):

35

equal:

Determine only by sign:





Rs2 (register 2) = 11000100000011110001101011110010

Instructions (FADD, FSUB, FMUL, FDIV, FSQRT ,FCMP):

00000000001000001111000111010011

00001000001000001111000111010011

00010000001000001111000111010011

00011000001000001111000111010011

01011000000000001111000111010011

10100000001000001111000111010011

Simulation:

FADD result: [0 ,1 ,0 ,0 ,1 ,0 ,1 ,0 ,1 ,0 ,1 ,1 ,0 ,0 ,1 ,0 ,0 ,0 ,1 ,0 ,0 ,1 ,0 ,0 ,1 ,0 ,1 ,0 ,0 ,1 ,1 ,0]

FSUB result: [0 ,1 ,0 ,0 ,1 ,0 ,1 ,0 ,1 ,0 ,1 ,1 ,0 ,0 ,1 ,0 ,0 ,0 ,1 ,0 ,1 ,1 ,0 ,1 ,1 ,0 ,0 ,1 ,0 ,1 ,1 ,1]

FMUL result: [1 ,1 ,0 ,0 ,1 ,1 ,1 ,1 ,0 ,1 ,0 ,0 ,0 ,1 ,1 ,1 ,0 ,0 ,1 ,0 ,1 ,1 ,1 ,1 ,0 ,1 ,1 ,1 ,0 ,0 ,0 ,1]

FDIV result: [1 ,1 ,0 ,0 ,0 ,1 ,1 ,0 ,0 ,0 ,0 ,1 ,1 ,1 ,1 ,0 ,1 ,0 ,1 ,1 ,0 ,1 ,0 ,1 ,1 ,1 ,0 ,1 ,0 ,1 ,1 ,1]

FSQRT result: [0 ,1 ,0 ,0 ,0 ,1 ,0 ,1 ,0 ,0 ,0 ,1 ,0 ,1 ,1 ,1 ,0 ,0 ,0 ,0 ,0 ,0 ,1 ,1 ,1 ,0 ,0 ,1 ,1 ,1 ,0 ,1]

FCMP result: [0 ,0]

Design:

FADD:

> calculation_output	B 01001010101100100010010010100110
----------------------	------------------------------------

FSUB:

> calculation_output	B 01001010101100100010110110010111
----------------------	------------------------------------

FMUL:

> calculation_output	B 11001111010001110010111101110001
----------------------	------------------------------------

FDIV:



```
> calculation_output B 11000110000111110101101011101011
```

FSQRT:

```
> halt B 0000
> calculation_output B 01000101000101110000001110011101
```

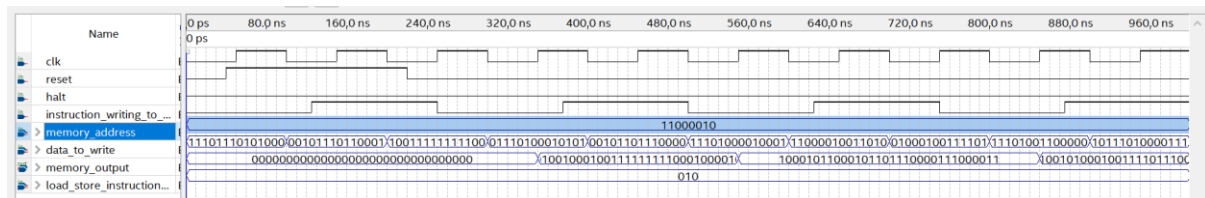
FCMP:

```
> calculation_output B 00000000000000000000000000000000
```

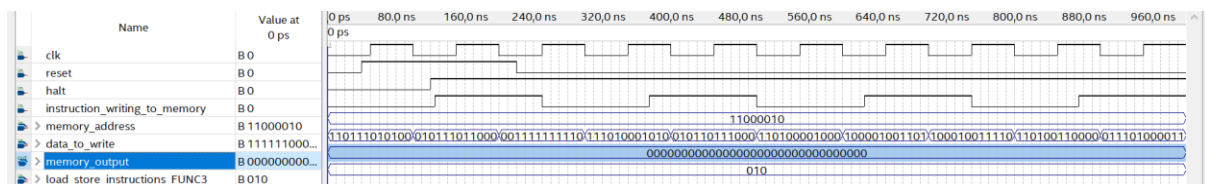
Memory:

Single address, reading and writing (there was a bug in the previous project!)

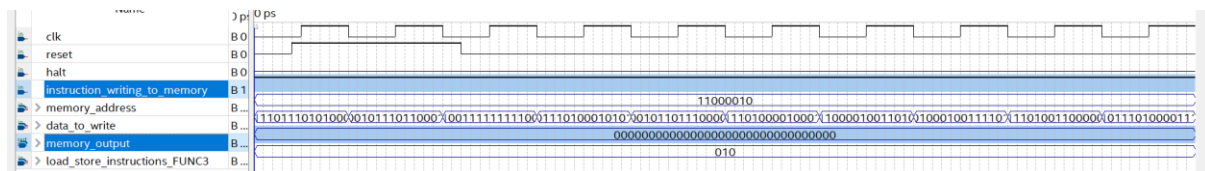
Notice that the data is zero at first, but we get the correct values after inserting them.



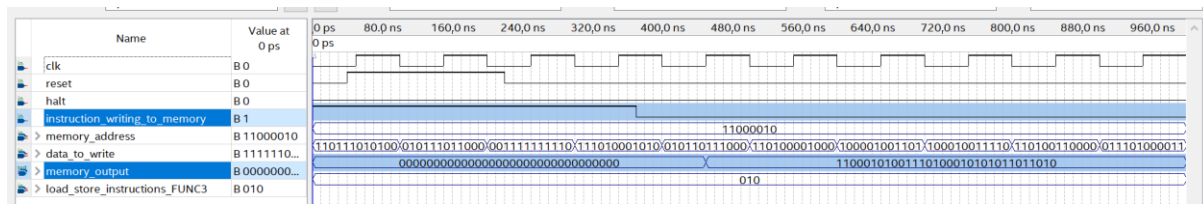
When halt is used, no data is written into the memory or from the memory, as needed:



When we only write to the memory, nothing is read outside



When writing and following the input we try reading, the last value is saved and steady, as needed:



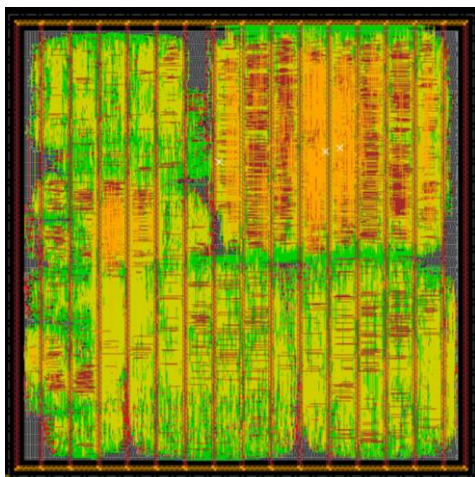
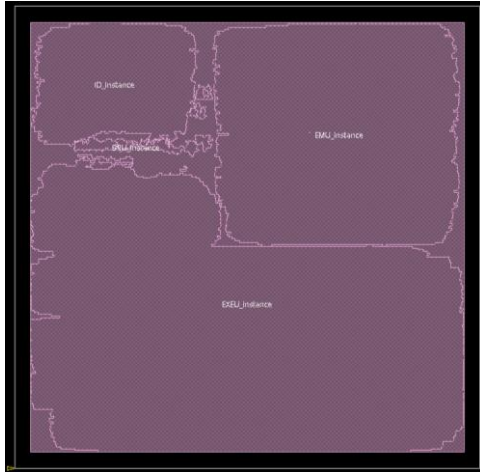
...



## Synthesis

After completing the design and verification of our CPU, we compiled and synthesized our design. We got the following results:

As for the physical layout, we have achieved the following:



Moreover, overall we had the following area, power, and timing:

Area:

\*\*\*\*\*

Report : area

Design : RISC\_V\_Top\_FP

Version: R-2020.09-SP2

Date : Mon Jan 10 10:10:00 2022

\*\*\*\*\*

Library(s) Used:

tsl18fs120\_typ (File:  
/tools/kits/tower/PDK\_TS18SL/FS120\_STD\_Cells\_0\_18um\_2005\_12/DW\_TOWER\_tsl18fs  
120/2005.12/synopsys/2004.12/models/tsl18fs120\_typ.db)

Number of ports:	14432
Number of nets:	77928
Number of cells:	64600
Number of combinational cells:	60588
Number of sequential cells:	3615
Number of macros/black boxes:	0
Number of buf/inv:	14427
Number of references:	23

Combinational area:	84086.750000
Buf/Inv area:	12769.000000
Noncombinational area:	17594.000000
Macro/Black Box area:	0.000000
Net Interconnect area:	39417.492536

Total cell area:	101680.750000
Total area:	141098.242536

Power:

Global Operating Voltage = 1.8

Power-specific unit information:

Voltage Units = 1V

Capacitance Units = 1.000000pf

Time Units = 1ns

Dynamic Power Units = 1mW (derived from V,C,T units)

Leakage Power Units = 1pW

Cell Internal Power = 30.5327 mW (90%)

Net Switching Power = 3.2875 mW (10%)

-----

Total Dynamic Power = 33.8201 mW (100%)

Cell Leakage Power = 2.2477 uW

	Internal	Switching	Leakage	Total
Power Group	Power	Power	Power	Power ( % ) Attrs
-----				
io_pad	0.0000	0.0000	0.0000	0.0000 ( 0.00%)
memory	0.0000	0.0000	0.0000	0.0000 ( 0.00%)
black_box	0.0000	0.0000	0.0000	0.0000 ( 0.00%)
clock_network	0.0000	0.0000	0.0000	0.0000 ( 0.00%)
register	28.5306	3.7978e-02	4.6768e+05	28.5690 ( 84.47%)
sequential	0.0000	0.0000	0.0000	0.0000 ( 0.00%)
combinational	2.0016	3.2495	1.7800e+06	5.2529 ( 15.53%)
-----				
Total	30.5322 mW	3.2875 mW	2.2477e+06 pW	33.8219 mW

timing:

data arrival time 26.71ns

The final clock cycle we achieved was 26.71 nano-second.

Timing is quite long due to complicated calculations, especially in the division and multiplication. We chose to implement this operation on a single clock for simplicity and better power performance, due to the absence of temporal registers in the process.

## Summary

The complexity of FP-32 CPU implementation was caused mainly due to the complex computation of the different mathematical computation, in comparison to Integer operations.

For the simple operations, such as comparison, addition and subtraction, the implementation was mostly straight forward and did not require much complexed planning.

On the other hand, for the more complex operations we had a few decisions to make regarding the logical implementation method. We chose to try and implement different operations using different approaches, and we can see different pros and cons for each one:

- For multiplication and division, we decided (along with our advisor) to implement the operations in a single clock cycle. While keeping the logic of the whole CPU simpler and reducing the use of temporal registers needed for calculations, we also paid a price by extending our clock cycle due to the long calculation process of these two operations. The trade of was between shorter cycle and more complex (and therefore more power absorbent) implementation and we chose to implement these operations on a single cycle.
- For square-root (SQRT) we decided to implement the operation with Taylor expansion by approximating the value around the square root of the exponent alone. We did it for two reasons:
  - Unlike all other arithmetic operations, there is no simple logic flow for square root, due to the complexity of the operation itself.
  - When implementing an operation that is complicated, the cost of not dividing it to a few cycles will be too high, because the latency would be at least  $\times 6$  as much as the other operations. On the other hand, dividing it into a few cycles does not require many additional resources. Therefore, the tradeoff can be advantageous.

As for the non-arithmetic operations, their design was relatively simple to plan and implement. They served as helpful self-validation that we were on the right track.

## Sources and bibliography

[1]The RISC-V Instruction Set Manual  
Volume I: User-Level ISA  
Document Version 2.2

[2]William Stallings, computer organization and architecture 6th edition

[3]Floating-Point Division and Square Root using a Taylor-Series Expansion Algorithm,  
Taek-Jun Kwon, Jeff Sondeen, Jeff Draper University of Southern California / Information  
Sciences Institute Marina del Rey, U.S.A. {tjkwon, sondeen, draper}@ISI.EDU

[4]Floating-point Square Root Calculation Arithmetic Based on Taylor-Series Expansion and  
Region Division, Jianglin Wei, A. Kuwana, H. Kobayashi K. Kubo, Y. Tanakas