ISTANBUL TECHNICAL UNIVERSITY
ELECTRICAL-ELECTRONICS FACULTY

EXTENDING THE INSTRUCTION SET OF RISC-V PROCESSOR FOR FLOATING-POINT ARITHMETIC

SENIOR DESIGN PROJECT

Salih Daysal
Mehmet Emin Tuzcu

ELECTRONICS AND COMMUNICATION ENGINEERING DEPARTMENT

JUNE 2022

ISTANBUL TECHNICAL UNIVERSITY
ELECTRICAL-ELECTRONICS FACULTY

EXTENDING THE INSTRUCTION SET OF RISC-V PROCESSOR FOR FLOATING-POINT ARITHMETIC

SENIOR DESIGN PROJECT

Salih Daysal
040160256

Mehmet Emin Tuzcu
040170023

ELECTRONICS AND COMMUNICATION ENGINEERING DEPARTMENT

Project Advisor: Prof. Dr. Sıddıka Berna Örs Yalçın

JUNE 2022

İSTANBUL TEKNİK ÜNİVERSİTESİ
ELEKTRİK-ELEKTRONİK FAKÜLTESİ

KAYAN NOKTA ARİTMETİĞİ İÇİN RISC-V İŞLEMCİSİNİN KOMUT SETİNİ GENİŞLETME

BİTİRME TASARIM PROJESİ

Salih Daysal
040160256

Mehmet Emin Tuzcu
040170023

ELEKTRONİK VE HABERLEŞME MÜHENDİSLİĞİ BÖLÜMÜ

Proje Danışmanı: Prof. Dr. Sıddıka Berna Örs Yalçın

Haziran 2022

We are submitting the Senior Design Project Report entitled as “Extending the Instruction Set of RISC-V Processor for Floating-Point Arithmetic”. The Senior Design Project Report has been prepared as to fulfill the relevant regulations of the Electronics and Communication Engineering Department of Istanbul Technical University. We hereby confirm that we have realized all stages of the Senior Design Project Interim Report by ourselves, and we have abided by the ethical rules with respect to academic and professional integrity .

Salih Daysal .....
040160256

Mehmet Emin Tuzcu .....
040170023

FOREWORD

We would like to express our gratitude to our advisor Prof. Dr. Sıddıka Berna Örs Yalçın, who dedicated her time for us and provided all the help she could. We also acknowledge the support our families gave us; without which we would not succeed.

June 2022

Salih DAYSAl
Mehmet Emin TUZCU

TABLE OF CONTENTS

Page

ABBREVIATIONS

ISA : Instruction Set Architecture

ALU : Arithmetic Logic Unit

DIV : Division

CPU : Central Processing Unit

RISC : Reduced Instruction Set Computer

HDL : Hardware Description Language

SQRT: Square Root

ASM : Algorithmic State Machine

MSB : Most-significant bit

LSB : Least-significant bit

LIST OF TABLES

Page

LIST OF FIGURES

Page

EXTENDING THE INSTRUCTION SET OF RISC-V PROCESSOR FOR FLOATING-POINT ARITHMETIC

SUMMARY

The development of the Internet and communication devices with scientific studies until today has enabled the speed and amount of communication between people to reach a different dimension compared to the past. Thanks to this increase in communication, the idea that solutions which used to produced separately for similar problems can be produced together led people to create the open-source concept. Open-source designs have become a concept that designers have the unconditional right to use, develop and change over time. Among these designs, researchers at the University of California, Berkeley developed a Processor Instruction Set Architecture called "RISC-V", which is an interface between software and hardware with no license requirements. Interested by researchers and companies, the RISC-V Instruction Set was designed by many different institutions and individuals. It emerged as open source after being approved by the RISC-V organization. A 32-bit RISC-V kernel was designed at Istanbul Technical University by senior students named Hornet. The hornet has an RV32I basic instruction set and M extension with detailed documentation and approved by RISC-V.

The Hornet processor already includes hardware that can perform basic mathematical and logical operations with integers. Nevertheless, integers cannot provide an adequate approximation to the set of real numbers, which makes the processor's ability to operate with floating decimal numbers, which is a different number representation. In the RISC-V Instruction Set content, the F extension offers an instruction set that allows operations to execute with floating-point numbers. In this context, we decided to improve the competence of the processor by implementing F instructions on the Hornet core for operations with floating-point number sets. By the courtesy of, users will be able to use the Hornet core in applications that require operations with real numbers.

Before we started adding the F instruction set, we tested the features of the Hornet processor with simulations on Linux by writing C programs, thus verifying that the core is functional. Afterwards, in order to increase our knowledge in the field of computer architecture, we read the basic books written about computer architecture and design. In the light of what we have learned, we decided on the blocks that we think we need to add to the processor and the order in which these blocks should be added. Since the 'F' extension is separated as Load/Store instructions, computational instructions, conversion instructions and move instructions , compare instructions and sign injection instructions, we added the instructions to the processor under these headings. In particular, the circuits that perform the multiplication, division and square rooting instructions among the computational commands are designed to work together in terms of control. We designed the circuits that make the implementations in accordance with the IEEE 754 standard and the RISC-v manual, and tested the accuracy of the results at every stage of the design. After the design phase was completed, we tested all the commands on the processor with the codes wroted in assembly language and verified core’s functionality.

KAYAN NOKTA ARİTMETİĞİ İÇİN RISC-V İŞLEMCİSİNİN KOMUT SETİNİ GENİŞLETME

ÖZET

İnternetin ve haberleşme aygıtlarının günümüze kadar olan bilimsel çalışmalar ile geliştirilmesi insanlar arasındaki iletişim hızı ve miktarı eskiye göre farklı bir boyuta ulaşmasın sağladı. İletişimdeki bu artış sayesinde benzer problemler için ayrı ayrı üretilmesi gereken çözümlerin birlikte üretilebileceği fikri insanları açık kaynak konseptini oluşturmaya yönlendirdi. Açık kaynak tasarımlar zaman içerisinde tasarımcıların koşulsuz şartsız kullanma, geliştirme ve değiştirme hakkına sahip oldukları bir konsept halini aldı. Bu tasarımların arasında California, Berkeley Üniversitesi 'indeki araştırmacılar tarafından adı ''RISC-V'' olan, lisans koşulu olmayan yazılım ile donanım arasında ara yüz olan bir Komut Kümesi Mimarisi geliştirildi. Araştırmacılar ve şirketler tarafından ilgi duyulan RISC-V Komut Seti Kümesi birçok farklı kurum ve kişi tarafından tasarlandı. RISC-V organizasyonu tarafından tescillendikten sonra açık kaynak olarak ortaya çıktı. Hornet'te RISC-V tarafından onay almış RV32I temel komut kümesine ve M uzantısına sahip açık kaynaklı ayrıntılı dokümantasyonu olan bir İstanbul Teknik Üniversitesi öğrencileri tarafından tasarlanmış 32-bitlik bir RISC-V çekirdeği.

Hornet işlemcisi temel matematiksel ve mantıksal işlemleri tam sayılar ile yapabilen donanımları bünyesinde hâlihazırda bulunduruyor fakat tam sayılar gerçek sayı kümesine yeterli yaklaşımı sağlayamıyor bu da işlemcinin daha farklı bir sayı gösterimi olan kayan ondalıklı sayılar ile işlem yapabilme yeteneklerini gerekli hale getiriyor. RISC-V Komut Kümesi içeriğinde F uzantısı adında kayan ondalıklı sayılar ile işlemlerin yapılabilmesi sağlayan komut kümesini sunuyor. Bu bağlamda işlemcinin kullanım alanını geliştirmek ve gerçek sayı kümesi ile yapılacak işlemlerde hassasiyeti çok daha yüksek sonuçlar elde etmek için Hornet çekirdeği üzerine F

komutlarının gerçeklemesiyle işlemcinin yeteneklerini geliştirmeye karar verdik. Bu sayede kullanıcılar gerçek sayılar ile işlem gerektiren uygulamalarında Hornet çekirdeğini kullanabilir hale gelecek.

F komut kümesini eklemeye başlamadan önce Hornet işlemcisinin özelliklerini C programları yazarak Linux üzerinde simülasyonlar ile test ettik bu sayede çekirdeğin fonksiyonel olduğunu doğruladık. Daha sonrasında Bilgisayar mimarisi alanında olan bilgi birikimimizi arttırmak amacıyla bu alan yazılan temel kitaplardan faydalandık. Öğrendiklerimizin ışığında işlemci üzerine eklememizin gerekli olduğunu düşündüğümüz blokları ve bu blokların hangi sıra ile eklenmesi gerektiğine karar verdik. Temel olarak 'F' uzantısı hafıza komutları, hesaplama komutları, dönüştürme ve hareket ettirme komutları, karşılaştırma komutları ve işaret enjeksiyon komutları olarak ayrıldığı için komutları yine işlemci üzerine bu başlıklar altında ekledik. Özellikle hesaplamalı komutlar arasından çarpma bölme ve karekök alma komutlarının gerçeklemesini sağlayan devreler kontrol açısından beraber çalışacak şekilde tasarlandı. Gerçeklemeleri yapan devreleri IEEE 754 standardına ve Rısc-v manuel'ine uygun olacak biçimde tasarlayıp sonuçların doğruluğunu tasarımın her aşamasında test ettik. Tasarım aşaması tamamlandıktan sonra komutların hepsini işlemci üzerine assembly dilinde yazılan komutlar ile test ederek işlevselliğini doğruladık.

1. INTRODUCTION

Computers became important for humankind in favour of their computational and memorial capabilities. These capabilities are ensured with components which are processor and memory; while the processor computes and controls the system, memory keeps the data that is the processor's input or output. Since computer systems are designed in the 1940's different design architectures are introduced by academia and industry to optimize processors for purposes. Rısc-v is one of the open-source instruction set architecture that developed at the University of California Berkeley. This project is aiming to extend instruction set of RISC-V core with 'F' Single-Precision Floating-Point instructions. 'F' instruction set includes load and store, computational, conversion and move, comparison and classify floating-point instructions. These instructions will be implemented with considering IEEE-754 floating point standards for single precision (32 bit) as requested in the RISC-V Instruction Set Manual. It is intended to add the Floating Point Unit (FPU) block to the pre-designed RISCV processor named HORNET to implement the given instructions. A FPU is a a part of computer system designed specifically for manipulating floating point numbers. Even without a floating-point unit, a CPU can handle both integer and floating point (non-integer) calculations. However, integer operations use significantly different logic than floating point operations. While it is possible to handle floating-point operations through software emulation, the goal of this project is to actually add a FPU to the processor that can handle the instructions as part of the processor. A FPU provides a faster way to handle calculations with non-integer numbers. The project consists of three main stages that are divided the instructions in the 'F' extension of the RISCV manual into 3 different groups that has tested on HORNET at each stage. Each student performed some of the different types of instruction groups and integrate them into HORNET.The first stage is to design and implement the data transfer instructions and adder/subtractor module.The second stage is design and implementation of multiplication/division, square root and comparison modules. The third stage is design and implementation of conversion and move

instructions. After design and implementations finalizied processor core tested for each instruction ‘F’ extension has but the fused multiply add and min-max instructions.

1.1 About RISC-V and HORNET

RISC-V is an open source instruction set architecture developed by researchers at the University of California Berkeley for use in lectures and research projects [1]. The areas reached by the project were later expanded and turned into an architecture accepted all over the world. In addition to creating a safe zone for software platforms and developers by freezing its basic features (frozen set), studies have also been made to expand it for many applications thanks to its flexible architecture. In addition, HORNET is a 32-bit processor with ‘Base Integer’ and ‘M’ set instructions designed by ITU students[2].

1.2 About IEEE-754

IEEE-754 is the most used standard for arithmetic of floating-point numbers. The formats and methods of floating-point numbers used in computer systems are specified in this standard's documentation. Examples of these specifications include rounding methods, arithmetic operations, exception handling, and how to encode exceptions (such as NaN, infinity, zero)[3]. The FPU that we will design during the project will be designed completely in accordance with these specifications and will be integrated into HORNET.

1.3 The RISC-V ‘F’ Instruction Set

Single-precision floating-point computational instructions of RISC-V, designated as "F" and conforming to the IEEE 754-2008 arithmetic standard, are briefly elaborated in Table 1.1[4].

Table 1. 1 : Instructions of 'F' Standard Extension.

Instructions	Executing Operations
FLW, FSW	Loads/Stores Floating-point data to/from destination.
FMADD.S, FMSUB.S	Multiplies source1, source2, adds source3, stores in destination.
FNMADD.S, FNMSUB.S	Multiplies source1, source2, negates, adds source3, stores in destination
FADD.S, FSUB.S, FMUL.S, FDIV.S	Adds/Subtracts/Multiplies/Divides source1, source2, stores in destination.
FSQRT.S	Computes square root of source1, stores in destination.
FSGNJ.S, FSGNJN.S, FSGNJX.S	Takes all bits from source1 except sign bit, which is determined by the sign of source2, the opposite sign of source2, or XOR of signs of source1 and source2, stores in destination.
FMIN.S, FMAX.S	Takes min/max of source1 and source2, stores in destination.
FCVT.W.S, FCVT.WU.S	Converts floating-point source1 value to signed/unsigned integer value, stores in destination.
FMV.X.W, FMV.W.X	Moves floating-point value from source1 to lower 32 bits of integer register destination, or vice versa.
FEQ.S, FLT.S, FLE.S	Equality/Less than/Less than or equal to of source1, source2, stores in destination.
FCLASS.S	Examines value in source1, stores 10-bit mask in destination that indicates class of floating-point number.
FCVT.S.W, FCVT.S.WU	Converts signed/unsigned source1 value to floating-point value, stores in destination.

There are several considerations when adding an FPU to the processor that will add considerable complexity. First, we will need to implement changes to the Control Unit to correctly interpret this extended set of instructions and store information about whether operations are integer or floating-point. The RISC-V floating-point extension uses 32 additional 32-bit registers for floating-point operations, so we'll need to include a second register file for these floating-point registers.

1.3.1 Single-precision load and store instructions

The load and store instructions, named as FLW and FSW can be seen at Tables 1.2 and 1.3, loads a single-precision floating-point value to floating-point register from memory and stores a single precision value of floating-point register to memory, respectively. Load and store instructions use a base address in source register and 12-bit signed byte offset as base+offset addressing method as integer base ISA does. In

the implementation of these commands, it is not much different from normal load and store as only the correct register bank should be selected.

Table 1. 2 : Single precision floating-point load instruction.

Immediate[11:0]	rs	width	rd	opcode
12	5	3	5	7
Offset	base	W	dest	LOAD-FP

Table 1. 3 : Single-precision floating-point store instruction.

Imm[11:0]	rs 2	rs1	width	Imm[4:0]	opcode
7	5	5	3	5	7
Offset	src2	src1	width	dest	STORE-FP

1.3.2 Single-precision floating-point computational instructions

Computational instructions can be classified in 2 different categories according to the instruction type. The first of these groups includes the R-type commands FADD.S-FSUB.S, FMUL.S-FDIV.S, FSQRT.S and lastly FMIN.S/FMAX.S shown in Table 1.4. Except for FMIN.S and FMAX.S instructions, as the name implies, other instructions do basic four operations on floating-point numbers and FSQRT.S take square root of a floating-point number. FMIN.S and FMAX.S, on the other hand, saves the smaller or larger of the two floating-point numbers, respectively, to the destination register. The other group is R4-type fused instructions, FMADD.S, FMSUB.S, FMNADD.S and FMNSUB.S, which, unlike the previous one, use three source registers, instead of two. Fused multiply and add instructions make multiplication and addition operations together to obtain the result, which causes it to differ from other computational instructions. Since these commands go through similar stages while running, common units with small varieties are used for rounding and normalizing the result.

Table 1. 4 : Single-precision floating-point computational instructions.

funct5	fmt	rs2	rs1	rm	rd	opcode
5	2	5	5	3	5	7
FADD/FSUB	S	src2	src1	RM	dest	OP-FP
FMUL/FDIV	S	src2	src1	RM	dest	OP-FP
FSQRT	S	src2	src1	RM	dest	OP-FP
FMIN/FMAX	S	src2	src1	RM	dest	OP-FP

1.3.3 Single-precision floating-point conversion and move instructions

Format conversion instructions are used to convert different data formats to each other. These are FCVT.W.S, FCVT.WU.S, FCVT.S.W and FCVT.S.WU are given below in Table 1.5 . Respectively, the functions of these commands are to convert a floating-point number to its closest signed or unsigned integer representation, or to convert a signed or unsigned integer to its floating-point counterpart. Since HORNET is a 32-bit architecture, conversions related to 64-bit integer numbers are not performed.

Table 1. 5 : Single-precision floating-point conversion instructions.

funct5	fmt	rs2	rs1	rm	rd	opcode
5	2	5	5	3	5	7
FCVT.int.fmt	S	W[U]/L[U]	src	RM	dest	OP-FP
FCVT.int.fmt	S	W[U]/L[U]	src	RM	dest	OP-FP

Floating-point sign injection instructions take interest on sign of sources rs1 and rs2 while rd takes all bits of rs1 but sign bit, can be seen in Table 1.6. There are 3 different sign injection instructions which are FSGNJ.S that sets sign bit of rd as sign of rs2, FSGNJN.S that sets sign bit of rd as opposite of rs2, FSGNJX.S that sets sign bit of rd as exored signs of rs1 and rs2.

Table 1. 6 : Single-precision floating-point sign injection Instructions.

funct5	fmt	rs2	rs1	rm	rd	opcode
5	2	5	5	3	5	7
FSGNJ	S	src2	src1	J[N]/JX	dest	OP-FP

The floating-point move instructions, FMV.X.W and FMV.W.X in Table 1.7, are used to transfer contents of a floating-point register to a general purpose(integer) register or contents of a general purpose register to a floating-point register without changing or converting the data, respectively.

Table 1. 7 : Single Precision Floating Point move Instructions.

funct5	fmt	rs2	rs1	rm	rd	opcode
5	2	5	5	3	5	7
FMV.X.W	S	0	src	000	dest	OP-FP
FMV.W.X	S	0	src	000	dest	OP-FP

1.3.4 Single-precision floating-point compare instructions

There are 3 different instructions for comparison in F extension that are FEQ.S, FLT.S and FLE.S showed in Table 1.8, stands for equal, less than, less than or equal, respectively. If the conditions are met, the integer destination register is written 1, otherwise 0.

Table 1. 8 : : Single Precision Floating-Point Compare Instructions.

funct5	fmt	rs2	rs1	rm	rd	opcode
5	2	5	5	3	5	7
FCMP	S	src2	src1	EQ/LT/LE	dest	OP-FP

1.3.5 Single-precision floating-point classify instruction

FCLASS.S is an instruction for classifying the contents of a floating-point register by writing a 10-bit mask to the integer register. The instruction classifies floating-point numbers by setting the corresponding bit in the integer register according to the Table 1.9 below. Classify instruction sets none of the floating-point exception flags. Instruction tpye and bit distrubutions given below in Table 1.10

Table 1. 9 : Classification Mask of floating-point numbers.

R_d bit	Meaning
0	is $-\infty$
1	is a negative normal number.
2	is asubnormal number.
3	is -0.
4	is $+0$.
5	is a positive subnormal number.
6	is positive normal number.
7	is is $+\infty$
8	is a signaling NaN.
9	is a quiet NaN.

Table 1. 10 : Single-precision floating-point classify instruction.

funct5	fmt	rs2	rs1	rm	rd	opcode
5	2	5	5	3	5	7
FCLASS	S	0	src	001	dest	OP-FP

1.4 Multiplication

Multiplication has main steps that are a generation of partial products and the addition of them [5]. Multiplication circuits are generally designed while considering the trade of in-between execution time and implementation cost. Multiplications for small operands are generally implemented with considering the implementing cost due to their low latency advantages. On the other hand, while implementing multipliers for big operands due to use latency advantages of small operands, big operands are divided into smaller operands to generate partial products with low latency. Generated partial products aligned with proper shifting operations and then aligned products are added to obtain the final result of multiplication. Latency of addition operation has also been considered to optimize the implementation of multiplication. Varying methods are available to optimize the addition of partial products using Carry Save Adders, like Wallace Tree [5].

1.5 Division

Division operation generally considered as an infrequent operation that does not have to be implemented as primary. On the other hand, ignoring the implementation of division can acquire a decrease in system performance for various applications [6]. Division algorithms can be categorized into five classes digit recurrence, functional iteration, very high radix, table lookup, and variable latency [7]. Implementing division algorithms generally includes variations from their classes. All these classes consider trade of in-between time and area constraints, concerning operational priorities division algorithm classes, can be chosen.

1.6 Square Root

Computers architecture was not capable of doing square root operation before computers became capable of executing functions in hardware implementations of processors directly. These hardware implementations of functions ensure executions in less clock cycle but they increase the complexity of the total implementation of processors. That necessitates the designing of time/area-efficient square root

implementation algorithms. There are iterative approximations and recurrence algorithms designed to implement the square root function. These algorithms are mainly differing with respect to the convergence to result. Recurrence algorithms are mainly assuming the upcoming bit of partial square root as 1 and check whether the remainder is negative or positive after subtraction to adjust the partial square root and remainder respectively for each two-bit of operands orderly.

2. BACKGROUND INFORMATION

2.1 Floating-Point Numbers

Floating-point numbers have emerged as a result of the projection of real numbers in the real world to computer systems with a certain precision. Since computer memory is limited, you cannot store numbers with infinite precision, no matter whether you use binary fractions or decimal ones: at some point you have to cut off. Therefore, a method similar to scientific notation is used to encode these numbers shown in Figure 2.1. In this way, we maximize the range of real numbers can be represented.

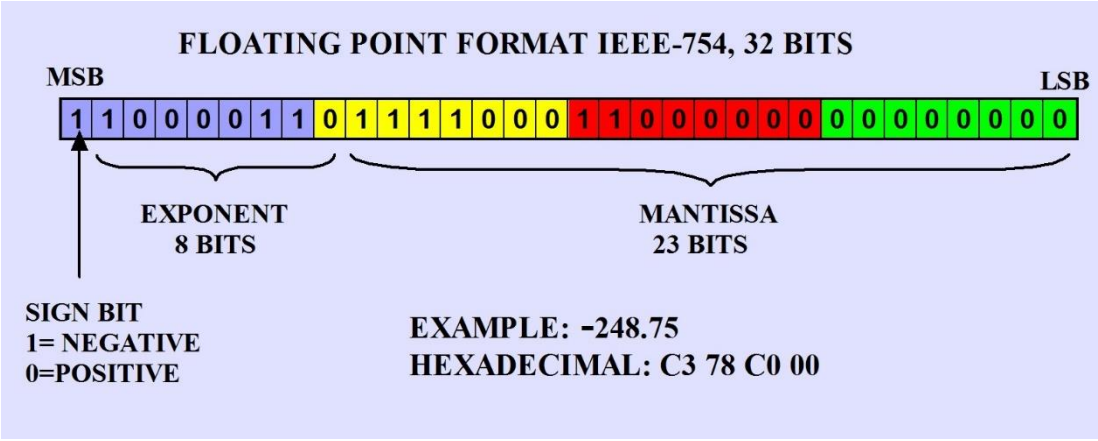

Figure 2. 1 : 32-bit representation of floating-point numbers.

Floating-point numbers can be represented by the following formula (2.1):

$$(-1)^s \times b^q \times c \quad (2.1)$$

Where,

— s is sign and 0 (positive) or 1 (negative).

— q is any integer exponent_min ≤ $q + p - 1$ ≤ exponent_max.

— c is a number represented by a digit string of the form $d_0\, d_1\, d_2 \ldots d_{p-1}$ where d_i is an integer digit $0 \leq d_i$

Since we will design for single precision in IEEE standards, 23 bits will be used as precision, 8 bits as exponent and 1 bit as sign, a total of 32 bits will be used.

The set of representable floating-point numbers can be classified into different groups in which they have different special cases. These cases include ±infinity , not-a-number(NAN) and ±0. The encoding formats of the entire set of single precision floating-point numbers can be viewed from Table 2.1.

Table 2. 1 : Binary encoding for Floating-Point Datum

Exponent	Mantissa	Object Represented
0	0	±0.0
255	0	±∞
255	1.0…xx	SNaN
255	1.1…xx	QNaN
1 to 254	1.x…xx	Normal Numbers
0	0.x…xx	Subnormal Numbers

When binary encoded floating-point numbers are expressed using scientific notation, the digit value of the integral part is hidden according to the IEEE standard, since the digit values are always 0 or 1. In addition to this and the fractional part after the decimal point is kept in the mantissa segment. If the value of this hidden digit is 1, they are called normalized numbers, and 0 are called denormalized numbers[2].

2.2 Exceptions

Exceptions are important factors in the standard to signal the system about some operations and results.

When an exception occurs, the following action should be taken:

- A status flag is set.
- The implementation should provide the users with a way to read and write the status flags.
- The Flags are ``sticky'' which means once a flag is set it remains until its explicitly cleared.

Common exceptions in floating-point numbers are listed below and represented in Table 2.2:

- ***Overflow, underflow and division by zero:*** As is obvious from the table below, the distinction between Overflow and division by zero is to give the ability to distinguish between the source of the infinity in the result.
- ***Invalid:*** This exception is generated upon operations that generate NaN results. But this is not a reversible relation (i.e. if the output is NaN because one of the inputs is NaN this exception will not raise).
- ***Inexact:*** It is raised when the result is not exact because the result can not be represented in the used precision and rounding cannot give the exact result.

Table 2. 2 : Exceptions in IEEE 754 standard

Exception	Caused By	Result
Overflow	Operation produce large number	$\pm\infty$
Underflow	Operation produce small number	0
Divide by Zero	x/0	$\pm\infty$
Invalid	Undefined Operations	NaN
Inexact	Not exact results	Round(x)

2.3 Rounding

Rounding is the one of the major problems of floating-point arithmetic which has to be considered for all arithmetic operations as well as the addition and subtraction.

Rounding is also a necessity due to higher precision which is one of the main advantages of floating-point numbers. To ensure precision our calculation will be implemented with extra tree binary digits which are called guard, round and sticky bits. Rounding also has 5 different modes which packed within the instruction that executed in processor defined in RISC-V manuals. IEEE 754 standards have conventions for rounding attributes used as modes of our rounder design. These modes can be viewed from Table 2.3.

Table 2. 3 : Rounding mode table.

Rounding Mode	Mnemonic	Meaning
000	RNE	Round to Nearest, ties to Even
001	RTZ	Round towards Zero
010	RDN	Round Down
011	RUP	Round Up
100	RMM	Roundto Nearest, ties to Max Magnitude
101		Invalid
110		invalid
111	DYN	In instructions's rm field, selects dynamic rounding mode; In Rounding Mode register, Invalid.

The guard, round and sticky bits processed during the preliminary normalization and are output as Round and Sticky after some manipulations. Then, in the rounding unit, the necessary rounding is done using these bits, the rounding mode and the sign of the output.

3. LITERATURE REVIEW

Reduced instruction set computers are alternative to complex ones which are getting more complicated nowadays. Reducing the instruction sets makes computer architecture plain in this way designing process becomes shorter related to normal

computers also reducing instruction sets reduces memory and process loads which decreases to energy consumption and cost of core. There for in developing IoT industry this kind of processors will be demanding cost effective application in near feature.

Computer instruction set architecture RISC-V is one of the upcoming topics of the last decade. Rısc-v processor cores can be implemented with a base integer instruction set however there are 16 different extensions for RISC-V ISA that were introduced by the rısc-v manual. Implementation of the processor varies with respect to aim of the project each instruction set extends the capability of cores while increasing the complexity of implementation. As we aim to extend the core with F extension that ensures working with floating-point number set to be capable to work with processes including real numbers. There are 10 different available Rısc-v cores that have F extension in for now. These cores are designed by 5 different organizations [4].

Codasip have four different core that have F extension are A70X, H50XF, L30F, L50F these cores are written in Verilog. L30F and L50F are 32-bit processor which have M, F, C extension on RV32I base instruction sets. A70x and H50XF are 64-bit processor which have M, A, F, D, C, extensions on RV64I base instruction set.

Sifive Founding have 3 different cores that have F extension are E2, E3, E7 all of these cores written in Verilog and have E, M, A, F, D, C extension on RV32I base instruction sets.

Fraunhofer IPMS have a core that named as EMSA5-FS written in System Verilog and have E, M, A, F, D, C extension on RV32I base instruction sets.

Openhw have a core that named as CV32E40P have F extension written in System Verilog and have M, F, C extensions on RV32I base instruction set.

MIT CSAIL CSG have a core that named as RiscyOO have F extension written in Bluespec and have M, F, C extensions on RV32I base instruction set.

4. DESIGN AND IMPLEMENTATION OF THE 'F' INSTRUCTION SET

4.1 Implementation of Load/Store Instructions

In the 'F' extension of the RISC-V specification document, that there should be a separate register file for floating-point arithmetic and it denotes that the following format in Table 2.4 and 2.5, should be applied for floating-point data transfer instructions**:**

Table 4. 1 : Single-precision floating-point store instruction.

Immediate[11:0]	rs	width	rd	opcode
12	5	3	5	7
Offset	base	W	dest	LOAD-FP

Table 4. 2 : Single-precision floating point load instruction.

Imm[11:0]	rs 2	rs1	width	Imm[4:0]	opcode
7	5	5	3	5	7
Offset	src2	src1	width	dest	STORE-FP

Then, when we examined the load/store instruction format for integer numbers, we realized that these two instruction sets have same formats. In this way, we completed the first instructions by making a few changes in some original design files.

4.1.1 Reorganize control

We made very little to no changes on the control side. Control signals for integer type load/store can also be applied in floating-point load/store instructions. But we added extra two multiplexers to the design to control which register file our outputs are taken from. In this, we defined two different signals for Select inputs of multiplixers that select data1 and data2 according to the instruction.

For example, when *fsw f15, -24(x8)* instruction is executed in assembly language, data1 should be selected from integer bank and data2 should be selected from floating-point register bank. In addition, when a LOAD instruction executed, which register bank the data will be loaded into at the WB(write back) stage is decided by the

registerbank_sel signal, which travels in the pipeline stage until the WB stage. Figure 4.1 shows that this change has created a diagram.

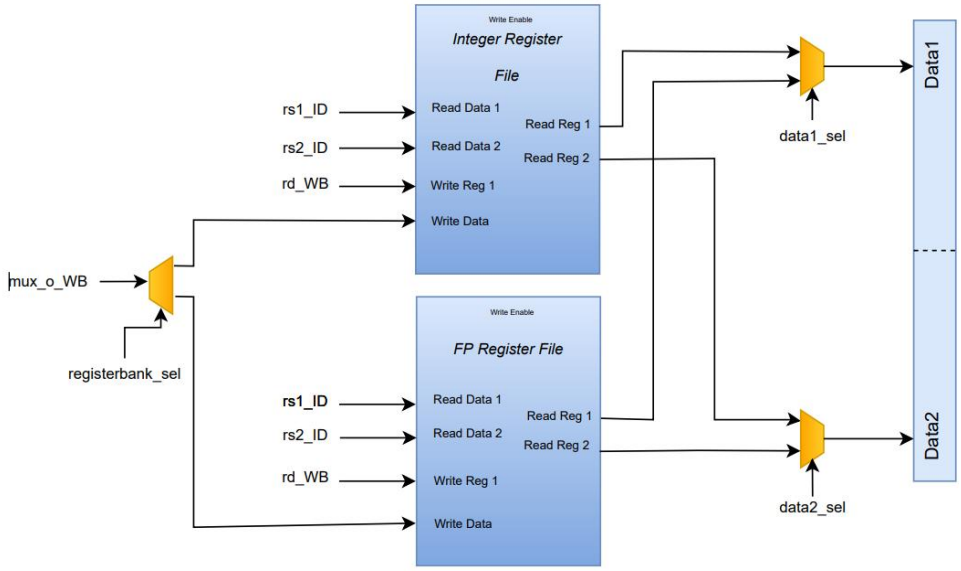

Figure 4. 1 : Block diagram of reorganized design of registers

```
if(rs1_ID == 5'b0)
    IDEX_preg_data1 <= 32'b0;
else
begin
    if(ctrl_unit_IDEX_data1_sel)
        IDEX_preg_data1 <= f_register_bank[rs1_ID];
    else
        IDEX_preg_data1 <= register_bank[rs1_ID];
end
if(rs2_ID == 5'b0)
    IDEX_preg_data2 <= 32'b0;
else
begin
    if(ctrl_unit_IDEX_data2_sel)
        IDEX_preg_data2 <= f_register_bank[rs2_ID];
    else
        IDEX_preg_data2 <= register_bank[rs2_ID];
end
```

Figure 4. 2 : Code section modified in the design file for data selection

The implementation of the extra multiplexers for selection of data1 and data2 can be seen from figure 4.2.

4.1.2 Verification of the implementation

Before we synthesized the implementation, we ran several tests to verify its behavioral functionality. we wrote a one basic C programs: a bubble sort algorithm, an integer

multiplication algorithm and a Fibonacci Sequence generator. We compiled these programs with RISC-V GNU C compiler, generated the opcodes and loaded them into the memory. We compiled and loaded these programs into the processor for simulation. Testing code given in figure 4.3 below.

```
1 void tst()
2 {
3         volatile float a = 2.4; // in hex format -> 0x4019999a
4         volatile float b = 4;   // '' ''  ''      -> 0x40800000
5 }
6
7 int main()
8 {
9         tst();
10
11 }
```

Figure 4. 3 : C code of test program.

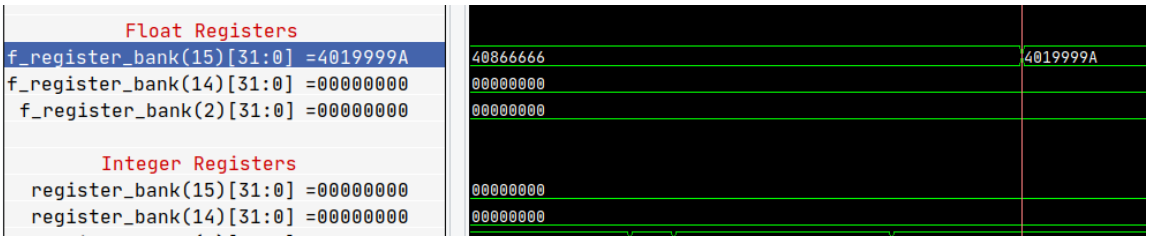

Figure 4. 4 : Values on FP registers.

As can see from Figure 4.4 and Figure 4.5, the variables a and b that we randomly numbers are assigned to floating-point register bank and then moved to memory correctly.

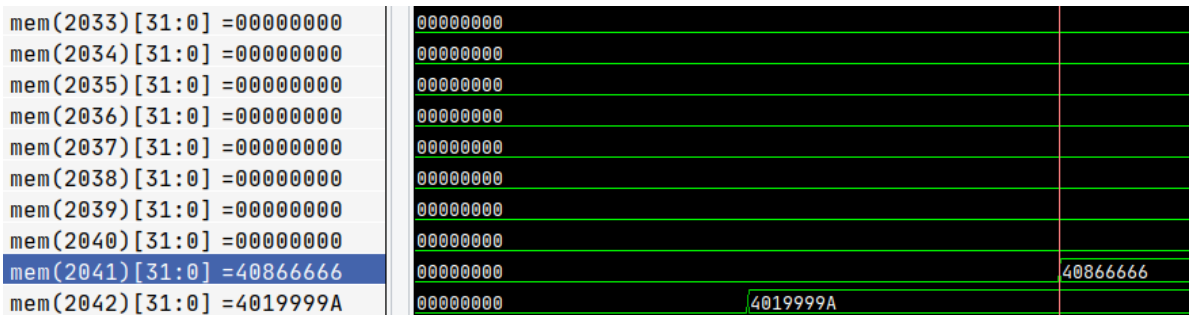

Figure 4. 5 : Value on memory.

After making sure that the Load/Store instructions are working properly, we start to think that the remaining commands will be implemented more quickly as we learn how to verify our design .

4.2 Floating-Point Computational Instructions

Floating-point instrucitons are generally instructions that perform mathematical operations for floating-point numbers. Among these instructions, the circuits that perform the sign, exponential and mantissa calculation of the addition and subtraction commands have many difference with the circuits that perform the multiplication, division and square root instructions. For this reason, the addition/subtraction block was designed and developed separately, while the other blocks were developed together. However the block that allow the decode of the 32-bit floating-point numbers into sign, exponential and mantissa, which is necessary for all mathematical operations, is designed to be used for all computational operations. Multiplication, division and square root of floating-point numbers are examined under the same heading as they are similar in terms of both sign and exponent handling. For the multiplication or division of mantissas, the same algorithm used in the existing muldiv module is used. For normalization, 3 different normaliser designs named sqrtNorm, divNorm and mulNorm are designed.

4.2.1 Decoding of operands for computational instructions

Floating point numbers are constructed as parts of privileged notation as mentioned before and represented in figure 2.1. This binary array includes sign, exponential and mantissa parts in addition to these there is a hidden one for floating numbers. This hidden one expands precision of floating point numbers by changing significant value with respect to the exponent. Significant value is stands for 24-bit binary number that integrate mantissa and hidden one of floating point numbers and used as binary operands of arithmetic operations. Floating point numbers are favourable approach to real numbers ergo considering infinite expansion and precision of real numbers with limited bits exceptions are inevitable. The exceptions which are element of real number set but not set element of floating point numbers . This numbers grouped as ±infinity , not-a-number(NAN) and ±0. There is also subnormal which is also

considered as underflow for floating numbers. By appointment of particular exponential value for subnormal hidden 1 expression ignored and 23-bit mantissa used as precision increasing. This classification and disjunction of operands part are essential for all arithmetic operations that leads us to design decoder unit which decodes the 32-bit floating point numbers for all arithmetic operations.

4.2.1.1 Implementation of decoder

Decoder units used as pair due to ensure two source operands unpacked properly for each arithmetic operation. Decoders are used for decoding source 1 and source 2 respectively. These decoder units take source operands used as input; give sign, exponential, significant and exceptions as outputs. Decoder units also coping with hidden one for sources which have zero exponentials and nonzero mantissa left most bit of significant assigned as 0 otherwise 1, with this appointment contradiction of significant settled. As significant values determined exponential differences of subnormal exponential and minimum exponential values are settled same as minimum exponential value. Decoding of floating point number enables to determining of exception cases with respect to the sign, exponential and mantissa. Combinations of maximum and zero values of exponential and zero value of mantissa used as indicators of exceptions. Using these indicators subnormal, zero, infinity and not a number(NaN) signals are generated.

4.2.2 Floating point addition and substraction

First, we did research on how to implement an algorithm for adding and subtracting floating-point numbers. Accordingly, we reviewed the following book, Computer Arithmetic and Hardware Designs and we follow the following algorithm[5]. Since the subtraction operation is essentially an addition, it is converted into an addition operation after the necessary sign arrangements. Flow chart of adder/substracter can be seen in Figure 4.6 below.

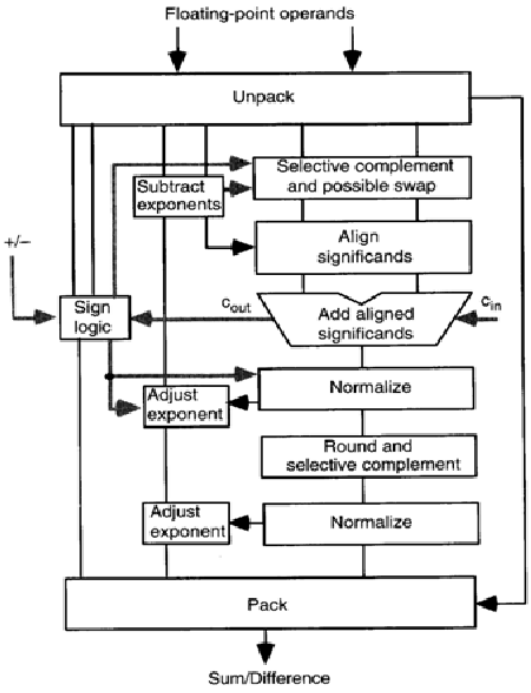

Figure 4. 6 : Floating point adding algorithm.

4.2.2.1 Designing the algorithm

The steps of the algorithm we will use are as follows:

- ***Step 1:*** Compare the exponent of the numbers in the entries and get the absolute value of the difference between them. This value will be the temporary exponent value of our result.
- ***Step 2:*** Shift the significand part of the input with the smaller exponent to the right by the exponent difference.
- S***tep 3:*** Select the required signals for addition or subtraction. So for subtraction, take the 2's complement of original number. Then keep the result in your RESULT variable.
- ***Step 4:*** Check MSB bit of RESULT during addition. If this value is 1, shift RESULT right by 1 and increase its exponent by 1. During subtraction, check RESULT for leading zeros. Shift RESULT left until the MSB of the shifted result is a 1. Subtract the leading zero count from tentative exponent.

- ***Step 5:*** Set the bits required for rounding in normalize and do the necessary rounding. After rounding, subject the result to the final normalize and handle the exceptions.

The last two steps are done by the normalizer, which is a separate module to encode floating-point numbers with a certain format. We skipped the rounding operations in this section, as we will implement the rounding instructions later.

4.2.2.2 Verification of the implementation

Before integrating the module we designed into Hornet, we applied behavioral simulation with Vivado Design Suite to make sure it was working properly. The testbench in Figure 4.7 is used result aquried as Figure 4.8 illustrates.

```
`timescale 1ns / 1ps
module sim1_tb();

reg     clk;
reg[31:0] A,B;
wire[31:0] OUT;
integer  i = 0;
fpu_adder UUT(.clk(clk), .A(A), .B(B), .OUT(OUT));

initial
begin
      A = 32'h3fe00000; // 1.75
      B = 32'h40800000; // 4.0
      // OUT = 40b80000, // 5.75
      #10;
      A = 32'hc2f71062; // -123.53
      B = 32'h43fa281d; // 500.3134
      // OUT = 43bc6446, // 376.7834
      #10
      A = 32'h00000000; // 0.0
      B = 32'hc20a0000; // -34.50
      // OUT = c20a0000, // -34.5
      #10;
      A = 32'h3e7020c5; // 0.2345
      B = 32'hc0add2f2; // -5.432
      // OUT = c0a651ec, // -5.1975
      #10
    $finish;
end
```

Figure 4. 7 : Test bench of adder/subtracter

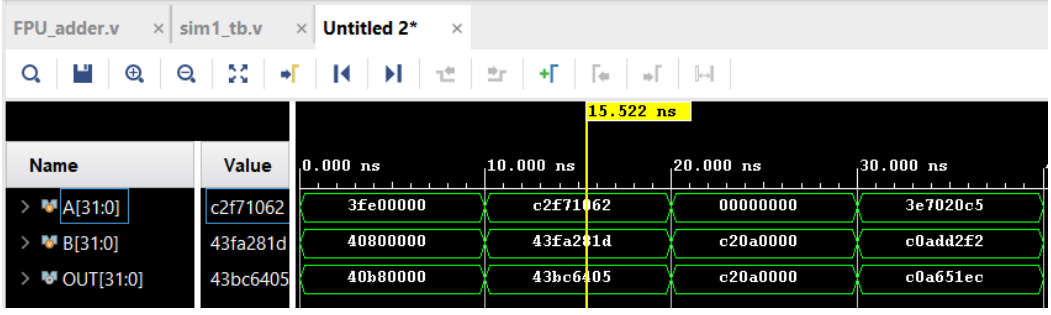

Figure 4. 8 : Simulation results on Vivado.

As can be seen from above Figure 4.8 the simulation results, the outputs give accurate results without rounding. In the next stage, clock and necessary rounding operations must be done to synchronize the circuit.

After completing rounding and handling of exceptions in this module, apart from this test, we tested 1 million different input combinations of random floating-point numbers that we created on matlab.Subsequently, saving these combinations to the txt file, we checked it over the tcl console in vivado by doing so, we fixed many errors that we did not notice during the design stage. Matlab code of testing variables are given in Figure 4.9 below.

```
for test = 1:1000000
    V = ['0':'9','A':'F'];
    X1 = randi(16,1,8);
    X2 = randi(16,1,8);
    V(X1);
    V(X2);
    N1 = typecast(uint32(hex2dec(V(X1))),'single');
    N2 = typecast(uint32(hex2dec(V(X2))),'single');
    hexstr1 = num2hex(N1);
    hexstr2 = num2hex(N2);
    hexstrsum = num2hex(N1/N2);
    fprintf(fileid, '%s %s %s \n', hexstr1, hexstr2, hexstrsum);
end
```

Figure 4. 9 : Matlab code of random FP number generator.

Finally, we also tested the numbers in some edge regions. For example, cases where the sum of the minimum subnormal number and the maximum subnormal number is normal, or the result is normal with the sum of two maximum subnormal numbers.

4.2.2.3 Verification on HORNET

After testing the Adder/Substractor module stand-alone on vivado, the block needs to be combined with HORNET as the next step. To do this, new control signals have been added with the purpose that this process can be done properly. These control signals are INTorFloat, FPU_func and FPU_roundingMode respectively can be seen in Figure 4.10. INTorFloat is a signal used to forward data from the correct output (ALU or FPU) in the pipeline. FPU_func is used to determine the addition or

subtraction operation, and FPU_roundingMode as the name implies, is used to control the rounding mode.

```
output reg INTorFloat,
output reg FPU_func,
output reg [2:0] FPU_roundingMode,
```

Figure 4. 10 : Control Signals for FADD/FSUB**.**

Also, the forwarding unit and hazard detection unit modules have been revised for possible data forwarding and pipeline stall situations that come with the "F" instruction set.

```
int main()
{

        volatile float a = 2.4; // 0x4019999a in IEEE-754 "single" format
        volatile float b = 4.2; // 0x40866666
        volatile float c = a-b; // 6.6 → 0xbfe66666
        volatile float f1 = 2.345; // 0x4016147b
        volatile float f2 = 5.344; // 0x40ab020c
        volatile float f3 = f1+f2; // 7.689 → 0x40f60c4a
        return 0;

}
```

Figure 4. 11 : Example C code for FADD and FSUB instructions.

As seen in the waveform in Figure 4.12,that we created using gtkwave, the correct values were calculated in the processor and then saved in the memory respectively.

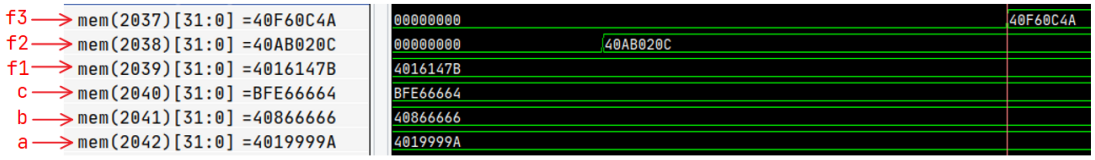

Figure 4. 12 : Simulation results on HORNET for FADD/FSUB instructions.

4.2.3 Floating-point multiplication

The representation of a floating-point number was like equation 4.1. Using this notation, the result obtained by multiplying two floating-point numbers can be represented as follows:

$$\boldsymbol{X \cdot Y = Z_M = (-1)^{(s_x \oplus s_y)} \cdot 2^{(e_x + e_y - 127)} \cdot (M_x \cdot M_x)} \qquad (4.1)$$

The sign part is pretty straightforward. The sign of the result is determined by EXORing the signs of the inputs. In the exponent part, it is obtained by adding two exponents. But the exponent we obtained from the product are biased two times, so 127 is subtracted from the result to obtain a value which has the bias is applied once.

4.2.3.1 Multiplication of significands

The mantissa part of the result is obtained by directly multiplying it. In order not to further complicate the design while multiplying the significands, we changed the design codes of the 32 bit multiplier circuit to operate in 24 bits in place of 32 bits, instead of directly using the previous MULDIV block in the processor. Flow chart of multiplication operation illustrated with Figure 4.13.

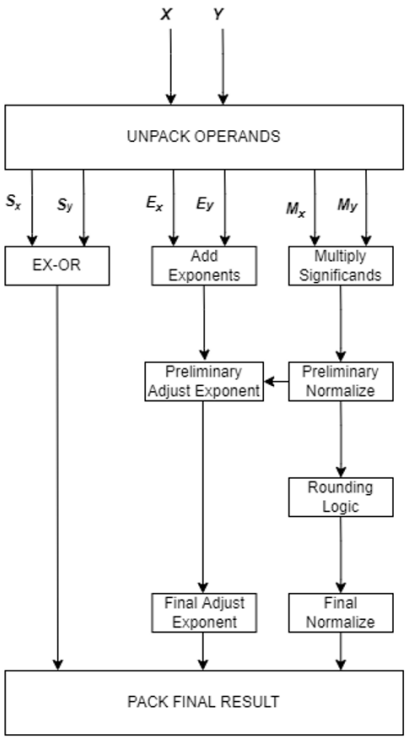

Figure 4. 13 : Flowchart of floating-point multiplication.

4.2.3.2 The multiplication algorithm

Two 24-bit significands can be written as the sum of two stage-1 partitions according to the following equation 4.2.

$$A = A_H 2^{12} + A_L \quad B = B_H 2^{12} + B_L \tag{4.2}$$

Using this notation, the product of two significands can be represented as:

$$A \times B = (A_H 2^{12} + A_L) \times (B_H 2^{12} + B_L) = A_H B_H 2^{24} + (A_H B_L + A_L B_H) 2^{12} + A_L B_L \quad (4.3)$$

Moreover, these partitions can be further split up into smaller stage-2 partitions.

$$A_H = A_{HH} 2^6 + A_{HL} \qquad B_H = B_{HH} 2^6 + B_{HL} \quad (4.4)$$

For example the expression $A_H B_H$ can be represented as

$$A_H \times B_H = (A_{HH} 2^6 + A_{HL}) \times (B_{HH} 2^6 + B_{HL}) \quad (4.5)$$
$$= A_{HH} B_{HH} 2^{12} + (A_{HH} B_{HL} + A_{HL} B_{HH}) 2^6 + A_{HL} B_{HL}$$

With the use of equations (4.2), (4.3), (4.4) and (4.5), operands are partitioned to A_{HH}, A_{HL}, A_{LH} and A_{LL} for A, and B_{HH}, B_{HL}, B_{LH} and B_{LL} for B. These partitions are 8-bit, and are multiplied in an order for generating the second level partitions $A_H B_H$, $A_H B_L$, $A_L B_H$ and $A_L B_L$. These second level partitions are then added in an order to generate the final result.

4.2.3.3 Normalization of Significands

Normalizer For Multiplication Although the normalizer of the multiplier circuit works with logic similar to the adder, some changes need to be made. The pseudo code of the multiplication normalizer can be seen in Figure 4.14.

For example, when we subtract bias from the sum of the exponents, if the result is underflow and the hidden digit is 0, it means that our number has fall into subnormal range. In this case, our output exponent should be set to 0. Also, exceptions that may occur during the normalization step, such as overflow and underflow, due to exponancial changes, are checked.

```
 //inSignificand -> Significand before normalization
 //        inExp -> Exponent before normalization
//outSignificand -> Significand after normalization
//        outExp -> Exponent after normalization
//  ExpUnderflow -> signal that checks if the exponent falls into the subnormal range
//  zeroCount    -> Variable that holds how many leading zeros in significand

if(!ExpUnderflow )

    if(inSignificand == 1x.xxxx...xx)
        outSignificand = inSignificand >> 1;
        outExp         = inExp + 1;

    else if (inSignificand == 01.xxxx...xx)

        outSignificand = inSignificand;
        outExp         = inExp;
    // when input is subnormal or when input exponent is not enough to normalize)
    else if (inExp == 1 & inExp <= zeroCount)
        outSignificand = inSignificand << inExp - 1;
        outExp         = 0;
    // when input exponent is suffient for normalization
    else
        outSignificand = inSignificand - zeroCount;
        outExp         = inExp - zeroCount;

else
    // if ExpUnderflow then shift the number by the absolute value of the exponent and
    // check if the number is normal, if normal then set Exponent 1 otherwise set to 0
        outSignificand = inSignificand >> inExp2C - 1;
        outExp         = |outSignificand[24:23] ? 1 : 0;
```

Figure 4. 14 : Pseudo code of Multiplication Normalizer.

4.2.3.4 Rounding logic for multiplication

After the Significands are multiplied, the Guard, Round and Sticky bits need to be properly determined in order to be able to round correcly. The Guard bit is a digit that gains importance when it is necessary to shift left only once while normalizing the significand. The Round bit is used to make the rounding more precise. Lastly, Sticky bit is used to give a general intuition about the values of the bits to be discarded from a certain precision.

$$\begin{array}{l} \boldsymbol{M_{x24} \cdot M_{x23} \ldots\ldots\ldots\ldots\ldots M_{x1} M_{x0}} \\ \times\ \boldsymbol{M_{y24} \cdot M_{y23} \ldots\ldots\ldots\ldots\ldots M_{y1} M_{y0}} \\ \hline \underbrace{\boldsymbol{M_{z47} M_{z46} \cdot M_{z45} \ldots\ldots\ldots\ldots\ldots}}_{\textit{24 bits}}\ \underbrace{\boldsymbol{M_{z1} M_{z0}}}_{\textit{24 bits}} \\ \textit{(actual significand)} \qquad \textit{(used for rounding)} \end{array} \tag{4.6}$$

G = Guard Bit = $\boldsymbol{M_{z23}}$

R = Round Bit = $\boldsymbol{M_{z22}}$

S = Sticky Bit = OR of less significand 22 bits of $\boldsymbol{Z_M}$

$$M_{z47}M_{z46} \cdot M_{z45}\text{.....}M_{z24} \;\text{—}\; g\ r\ s$$

Case a: $M_{z47} = 0$, $Z_M = M_{z46} \cdot M_{z45}\text{.....} M_{z24}\ g$ R = r S = s

Case b: $M_{z47} = 1$, $Z_M = M_{z47} \cdot M_{z46}\text{.....} M_{z25}\, M_{z24}$ R = g S = r OR s

Case c: M_{z47}, $M_{z46} = 0$ Z_M is shifted until MSB is 1. Then R is equal to 23nd bit of shifted version of Z_M and S is equal to OR of less significand 22 bits of shifted version Z_M.

After the normalization step, the final Round and Sticky bit are specified using below cases. Then, the R and S bits are rounded according to the rounding mode used. Eventually, after the appropriate rounding is done, the final normalization and exponent adjustment processes are done, and our output is ready.

4.2.3.5 Testing the floating-point multiplication circuit

During the test phase, we tried the random floating-point numbers we generated with MATLAB, as in the previous units, with simulation over Vivado. We reviewed the design by analyzing incorrect results and updated it until finally no errors.

Afterwards, we would integrate this circuit directly in Hornet, but since it will have common signals with the Division block, we decided to do it after designing the division circuit.

4.2.4 Floating-Point Division

In floating-point division arithmetic, the output sign is calculated by EXORing the input signs, as in multiplication. In floating-point division arithmetic, the exit sign is calculated by EXORING the input signs, as in multiplication. Since the exponents will be subtracted from each other in division, the resulting exponent will be unbiased. For this reason, the value of 127 is added to bias the output exponent once.

$$X \div Y = Z_D = (-1)^{(s_x \oplus s_y)} \cdot 2^{(e_x - e_y + 127)} \cdot (M_x \div M_x) \qquad (4.7)$$

4.2.4.1 Division of Significands

We used same algorithm that Hornet does due to pave the way for further works. In addition to this the algorithm have been chosen with consideration of area and time coefficients. Trade of between area and time concluded with regard of frequency of division operations comes while processor working [2]. After determining division algorithm of significant part of floating point we take in to account exponential and sign operations. These operations are very straight forward for division as well as multiplication.

Unlike integer division, for the division of 24-bit significands, the dividend and divisor must be must be converted to the appropriate format before starting the operation. First, the dividend is shifted to the left by 26 bits and 26 bits of 0 are padded to the beginning of the divisor. The reason for shifting 26 bits instead of 23 bits is that the extra 3 bits are used for rounding. Then offset is calculated to get the output correctly for subnormal divisors.

The formatting is as follows. Assume that M_x is dividend, M_y is divisor and M_D is quetient.

$$\frac{M_x}{M_y} = M_D \tag{4.8}$$

Can be written as:

$$\frac{M_x}{M_y} \times 2^{26} = M_D \times 2^{26} = M_{shifted} \tag{4.9}$$

equation 4.9 can also be written as equation 4.10.

$$M_d = M_{shifted} \times 2^{-26} \tag{4.10}$$

Accordingly, $M_d$is the 26-bit right shifted version of the $M_{shifted}$. For subnormal divisors, $M_{shifted}$ needs to be shifted to the left by $24 - offset$ amount. Normalizing and rounding is done with the similar logic as in the multiplication.

4.2.5 Floating point square root

Calculating the square root of single-precision floating-point numbers is an unary operation unlike other computational instructions. Since the square root operation is valid only in the domain of positive real numbers, sign of input operand is just an indicator to determine whether square root operation is valid or not. The exponent is divided in half, as can be seen from equation 4.11. The point to consider is when the exponent is odd. In these cases, operations are performed by considering the significand shifted 1 digit to the left.

$$\sqrt{X} = \sqrt{M_X \cdot 2^{e_x}} = \begin{cases} \sqrt{M_X} \cdot 2^{\left(\frac{e_x}{2}\right)} & if\ e_x\ is\ even, \\ \sqrt{2 \cdot M_X} \cdot 2^{\left(\frac{e_x - 1}{2}\right)} & if\ e_x\ is\ odd. \end{cases} \quad (4.11)$$

Nontrivial calculation of square root operation is the computation of significand value's square roots. Since the significant value is 24-bit binary array which is interpreted differently for floating-point arithmetic, approximations for integer square root operations are also applicable to significant value.

4.2.5.1 Square Root of Binary Numbers

Square root operation has only one operand unlikely to other mathematical operations. The operand expected to be equal to the square of the final result that leads design to be different from analytic approaches. Therefore, in square root operation result has to be calculated with algorithms which are mainly divided into iterative approximations and recurrence methods. Since our concern is calculating square root of floating point numbers with binary 32-bit representation. Recurrence algorithms are chosen due to constant execution time which has critical significance for pipelined processor structure. Recurrence algorithms generate square root bit by bit, with subtracting the square of instant square root from related remainder. Since these operations employ large number of different units, square root calculation algorithms have been developed to reduce the number and complexity of components. By courtesy of these algorithms square root operation can be implemented with simple digital circuitry. 'A new non-restoring square root algorithm introduced by Yamin Li and Wanming Chu is very efficient to implement. Many existing design are less area and time efficient than the implementation of new non-restoring square root algorithm [10].

4.2.5.2 Mathematical expression of binary square rooting operation

Square root operations start with calculating most significant bit of square root result and to be continued till least significant bit of square root calculated. While calculating square root of binary numbers new non-restoring algorithm investigate on remainder for each step of calculation.

Suppose D is input operand, q_k is partial square root and $r_k ab$ is the partial remainder at step k where ab is the next pair of operand, $a = D_{2k-1}$ and $b = D_{2k-2}$. Note that q_k, r_k are binary sequences and multiplying these sequence with 2 corresponds shifting these sequences left by one bit.

$$qrt(D) = \sqrt{D_{2k-1}D_{2k-2}D_{2k-3}\cdots\cdots D_1D_0} \tag{4.12}$$

$$= Q_{k-1}Q_{k-2}Q_{k-3}\cdots\cdots Q_1Q_0 = q_k$$

Square rooting $2 \times$ k bit binary number resulted with k bit binary sequence. Initially, the MSB of the square root result and first partial remainder is calculated by subtracting 01 from the $D_{2k-1}D_{2k-2}$ pair. Other bits of remainder and next significant bit pairs of D are concenatated as $r_k ab$. The remaining bits of the square root result are obtained sequentially by subtracting or adding the current partial remainder from the partial square root result, according to the MSB value of the partial remainder calculated in the previous state. Partial tables truth table represented below Table 4.3.

Let's assume that the initial partial square root and the partial remainder are calculated according to the figure A, where $D_{2k-1}D_{2k-2}$ is first pair of binary input, Q_{k-1} is the MSB of square root and $R_{k+1}R_k$ first pair of partial remainder .

Table 4. 3 : Truth table of partial root and related square root bit.

$D_{2k-1}\, D_{2k-1}$	Q_{k-1}	$R_{k+1}\, R_k$
00	0	00
01	1	00
10	1	01
11	1	10

The next bit of square root and partial square root are calculated as in Equation 4.13 and Equation 4.14.

$$Q_{k-2} = \begin{cases} 1, & if\ r_{k-2} = r_{k-1}ab - {q_{k-1}}^2 \geq 0, \\ 0, & otherwise. \end{cases} \quad (4.13)$$

$$q_{k-2} = \begin{cases} q_{k-1}1, & if\ Q_{k-2} = 1, \\ q_{k-1}0, & otherwise. \end{cases} \quad (4.14)$$

cd is the next pair of D. If the remainder r_{k-2} is non-negative, next partial root is equal to following expression.

$$r_{k-3} = r_{k-2}cd - {q_{k-2}}^2 \quad (4.15)$$

If the remainder r_{k-2} is negative then next partial root equals to:

$$r_{k-3} = r_{k-1}abcd - {q_{k-2}}^2 \quad (4.16)$$

As can be seen from the expressions above, the new non-restoring algorithm results in two different expressions for negative partial remainder and positive partial remainder.These expressions will be implemented with digital circuitry as adder and shift registers. For $2 \times k$ bit binary number k + 2 bit adder/subtracter can be used with shift registers to compute square root.

4.2.5.3 Square root of significand

Since the square root of a 24-bit significand will be 12-bit, there will be a loss of precision, so it needs to be extended further. For that purpose, along with the bits required for rounding, the significand is extended to 54 bits by shifting it to the left. In each clock cycle two bits of square root are calculated and the final result is calculated in 27 cycle. Also, the hidden bit of the square root result can only be 0 in the subnormal number set, so normalization is not performed for normal inputs. Hence, normalization in square root aritmetic is relatively simpler than other operations.

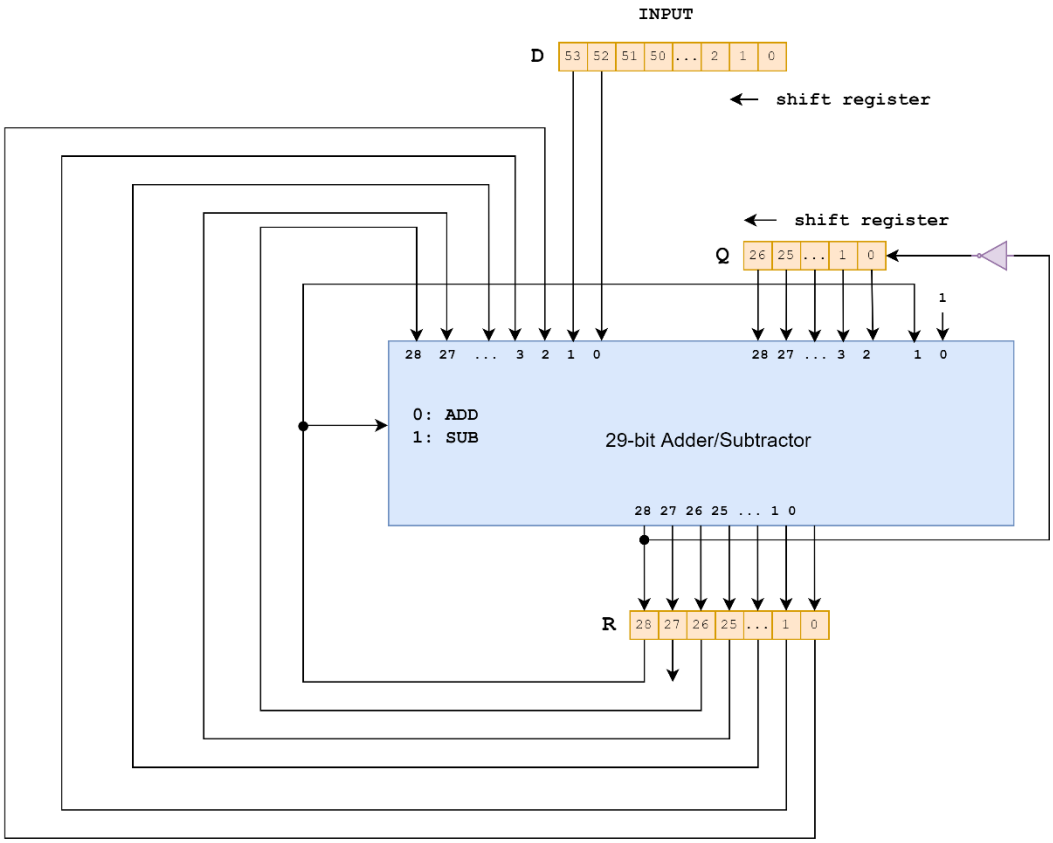

Figure 4. 15 : Implementation of square root algorithm.

As can be seen from above Figure 4.15 acquired result of square root mathematical expressions has implemented with 29-bit Adder/Substracter block and tree shift registers for remainder, quantinent and input operands.

4.2.6 Control unit

The Control Unit is responsible for making sure that the rest of the circuit is operating correctly. It controls which signals will be inputted to the Square Root Block, and where the outputs from the Divider Block will go. The Control Unit is an Algorithmic State Machine (ASM), and it controls the circuit by determining the values of the control signals. The State Diagram of the circuit is given in Figure 4.16.

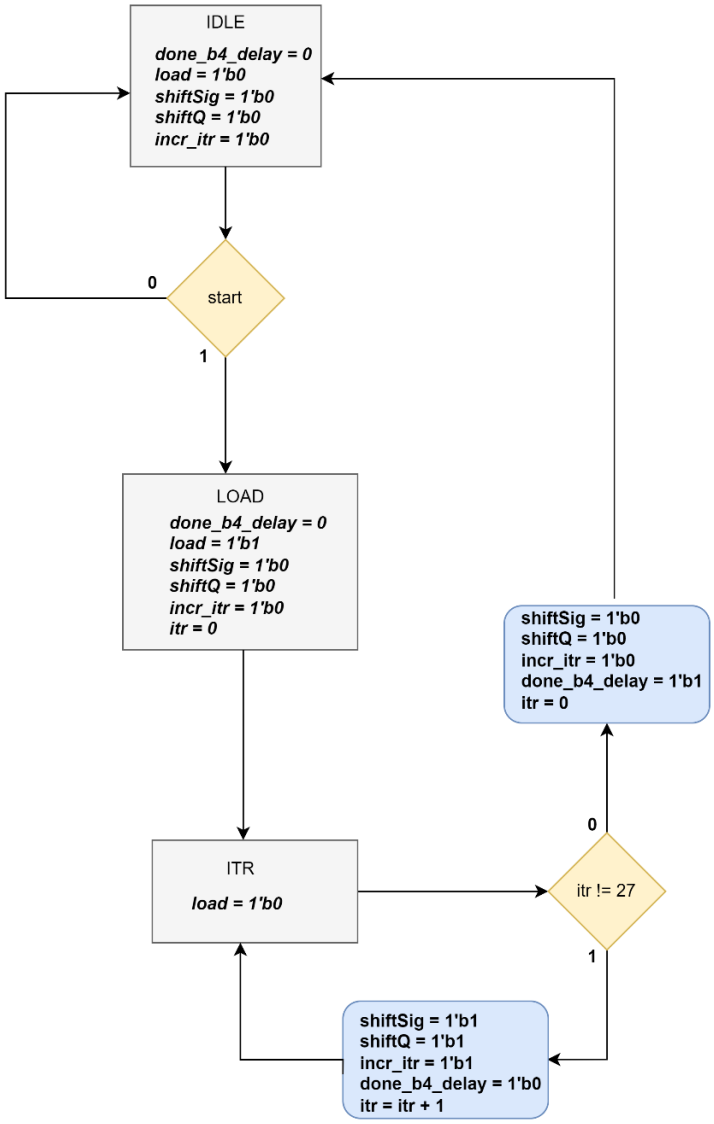

Figure 4. 16 : The State Diagram of the circuit.

The circuit waits for the operation in the state IDLE. When the start signal is asserted, the square root operation begins and the circuit goes through 27 iterations, during where it calculates required results and transfers them to the next round. In the final iteration, which is the 27th round, it asserts a ready signal and the final result for square root is ready.

4.2.7 The Floating-Point Muldivsqrt Circuit

4.2.7.1 Datapath

Floating-point multiplication, division, and square root circuits are sequential circuits and take multiple clock periods to calculate. For this reason, after these circuits were designed and tested separately, they were combined in a module called MDS. mds contains the following submodules:

- **An exponent handler:** Calculates the pre-normalization exponent according to the type of operation.
- **Rounders**: Each operation round differently the significand according to the rounding mode.
- **A final normaliser:** Performs possible normalization of significand after rounding.
- **A sign handle**r: Determines the sign of the floating-point output with respect to operation type.

Each operation block contains submodules that do the initial normalization. Exception flags are also set in the relevant operation block when necessary. What is meant by necessity for example, division-by-zero exception is calculated only for floating-point division. In other operations, it is set to 0 directly. The simplified diagram of the circuit can be seen from Figure 4.17.

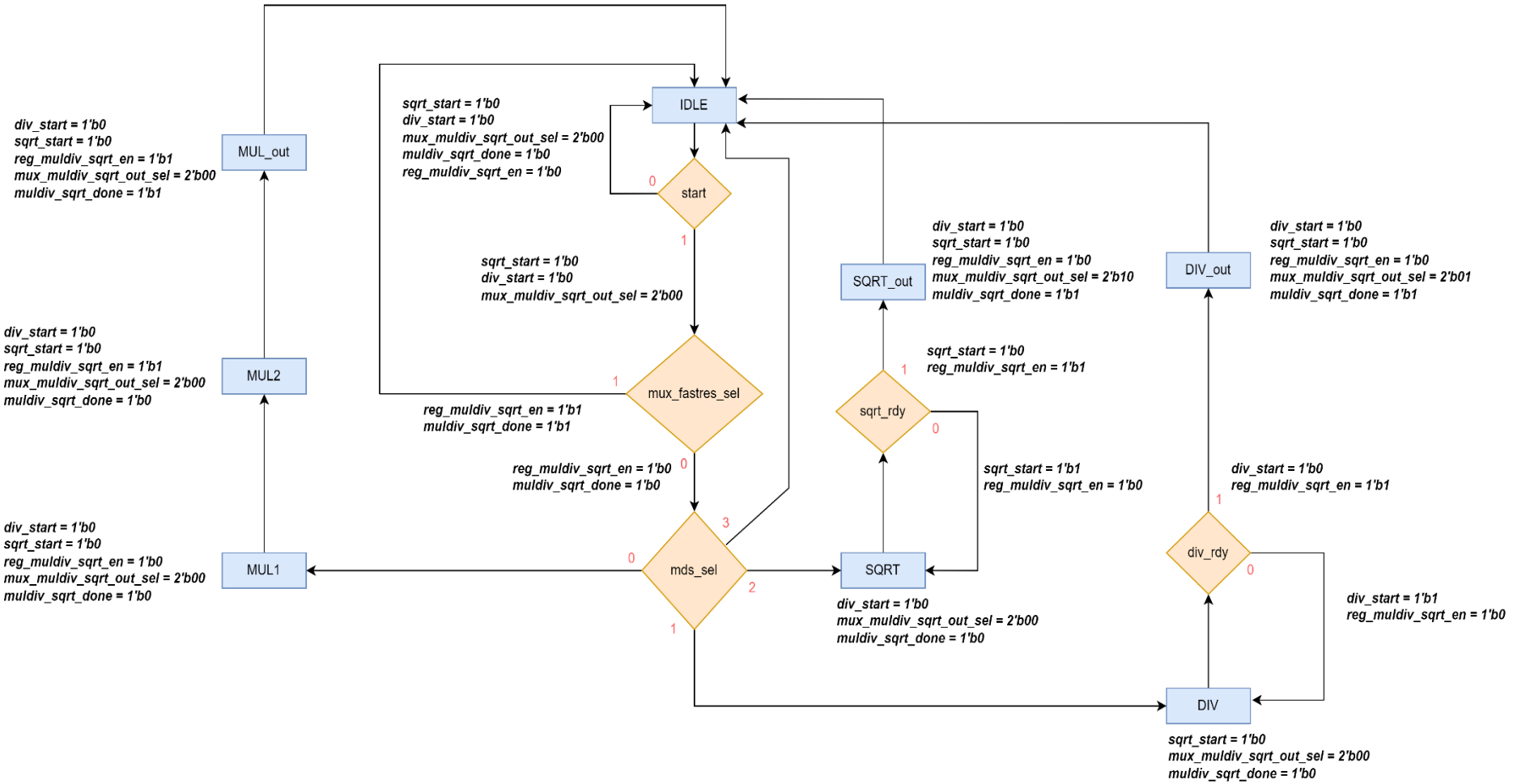

Figure 4. 17 : Square root and division control diagram.

4.3 Implementation of Conversion, Move and Compare Instructions

As mentioned earlier, conversion instruction are used to convert a floating-point number to the its nearest integer equivalent, or to the floating-point equivalent of an integer number in the IEEE-754 standard. In computing , since all 32-bit signed or unsigned integer numbers have an unique floating-point representation, there are no restrictions or special conditions for inputs of integer-to-float conversions. However, in the float-to-integer conversion, there are special cases in floating-point domain, such as INF or NAN, or cases where the floating-point number to be converted cannot be represented by the signed or unsigned 32-bit integer. These particular cases can be viewed in the table below Table 4.4.

Table 4. 4 : Special cases for conversion instructions

	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	-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 infinity or NaN	$2^{31}-1$	$2^{32}-1$	$2^{63}-1$	$2^{64}-1$

4.3.1 Conversion instructions

4.3.1.1 Integer-To-Float Conversion

First of all, since the MSB of the significand of normal numbers must always be 1 according to the IEEE 754 floating-point standard, the operand integer needs to be converted to this format. The steps of the algorithm are as follows.

- ***Step 1:*** Depending on whether the operand integer is interpreted as signed or unsigned, the absolute value of the number is taken and the 31 bits 0, which is the maximum shift number, is added to the end of the absolute value obtained.
- ***Step 2:*** Shift the temporary variable until the MSB of the significand become 1.
- ***Step 3:*** The exponent to be obtained will be equal to the shift amount. But this value is the actual value without bias added. For this reason, a bias of 127 is added to the final exponent for the single precision. Sign bit is equal 0 if the number is to be converted as unsigned, otherwise it is equal to input's original sign. For an illustration, the conversion of 1123412, an integer, to floating-point can be examined as follows in Figure 4.18.

In binary, 1123412 is represented as:

$$1123412_{10} = 00000000000100010010010001010100_{2} \quad (4.17)$$

In floating-point notation, it equals to:

$$(-1)^{0} \times 2^{20} \times 1{,}00010010010001010100_{2} \quad (4.18)$$

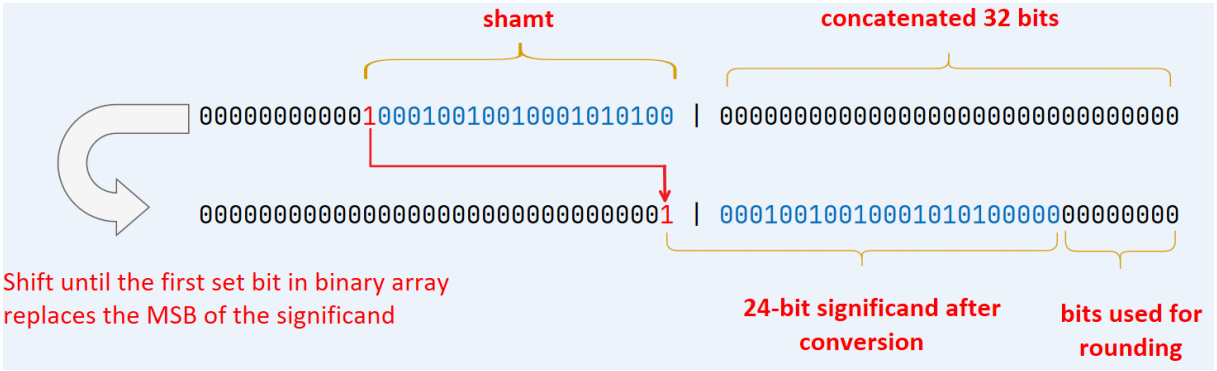

Figure 4. 18 : Integer to Float conversion operation illustration.

4.3.1.2 Float-to-Integer Conversion

In the integer conversion operation which works with a similar to the previous algorithm, if the exponent of operand floating point number is less than 0, the integer equivalent is directly equal to 0. Otherwise, the integer equivalent is determined when the significand is shifted until the exponent becomes 31. The reason for this is to shift the decimal point of the number by 31 digits. But, as mentioned before, when operand

exponent is higher than 31, there are special cases where we cannot represent the operand floating point number with an integer. In these cases, output is determined by the directives in the RISC-V Manual. For an illustration, the conversion of 123423953.78845, a floating point number, to integer can be explained as follows.

First of all, this number can be represented in single precision format as follows in equations and Figure 4.19:

$$123423953.78845_{10} = (-1)^0 \times 2^{26} \times 1{,}11010110110100110011010_2 \qquad (4.18)$$

After calculating the required shamt and performing the shift, the mantissa becomes:

$$(-1)^0 \times 2^{31} \times 0{,}00001110101101101001100110101_2 \qquad (4.19)$$

When we complete the remaining bits with 0, we get the following result.

$$00000111010110110100110011010000_2 = 123423952_{10} \qquad (4.20)$$

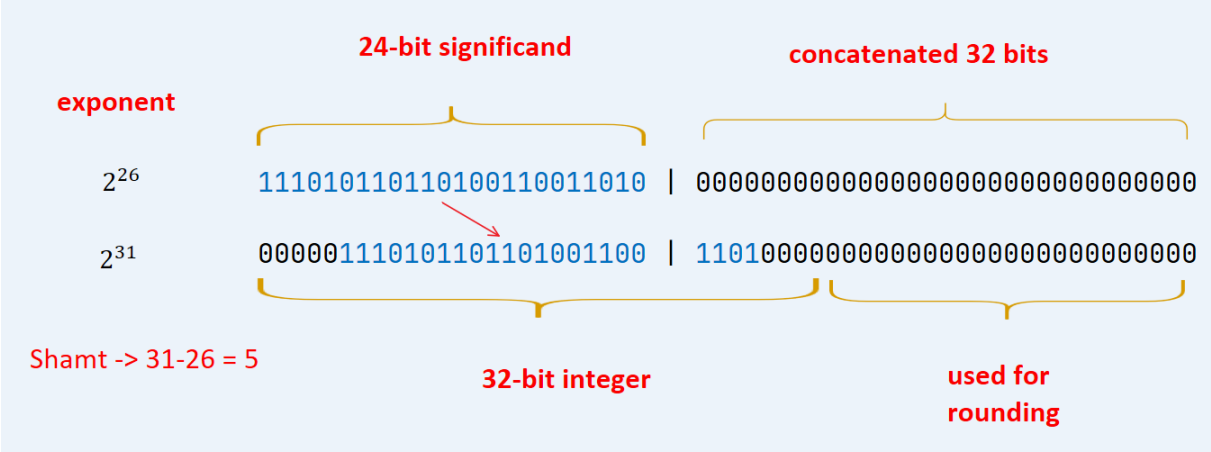

Figure 4. 19 : Float to integer conversion illustration.

4.3.2 Sign Injections

Sign injection instructions directly copy the exponent and mantissa of the value in the source floating point register to the destination register. The sign is determined by the type of injection. For this reason, their implementation is quite straightforward. These instructions are used to execute some assembler presudoinstructions. These are FMV.S, which moves a number from a floating point register to another floating point register, FNEG.S, which negates a floating point datum, and FABS.S, which takes the

absolute value of a floating point datum. Implementation of instructions is repsented below Figure 4.20.

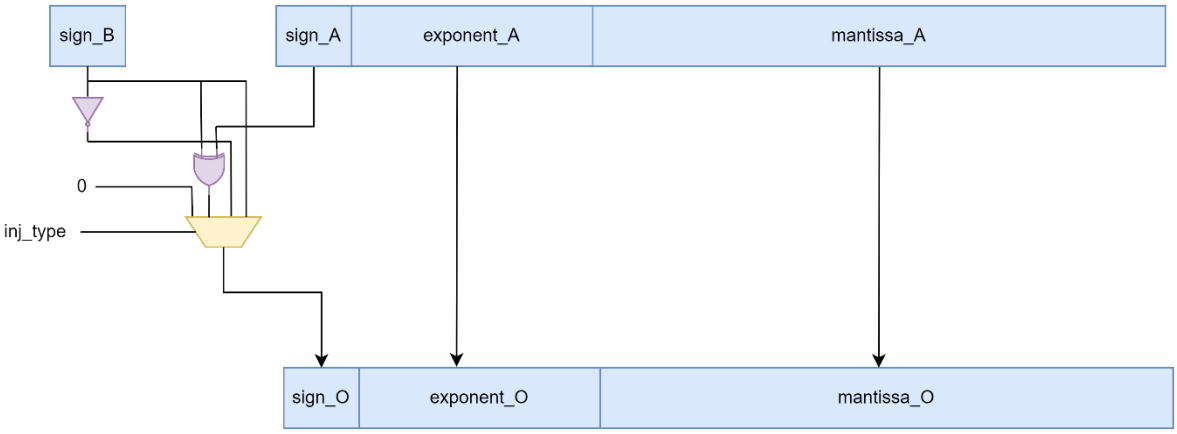

Figure 4. 20 : Implementation of Sign Injection operation.

4.3.3 Move instructions

Floating-point move instructions move data between floating-point registers and general purpose integer registers. Since these instructions do not make any modifications on the data, they are not designed as a separate module. data moving on pipeline with control signals. The data to be transferred is moved to the desired register bank with appropriate control signals.

4.3.4 Compare instructions

floating point comparison instructions compare data in floating-point registers with each other. First the sign of the number, then the exponent, and finally the mantissa part are compared. The comparison process is relatively simple as the sign of the number is determined directly from the MSB of the 32-bit floating-point datum and exponent is also in a biased format rather than signed format. These instructions change the value in the condition bit, which is LSB of 32-bit output. The condition bit is set (made equal to one) if the condition is satisfied. Otherwise the condition bit is cleared (made equal to zero). The block diagram of the circuit is as in Figure 4.21.

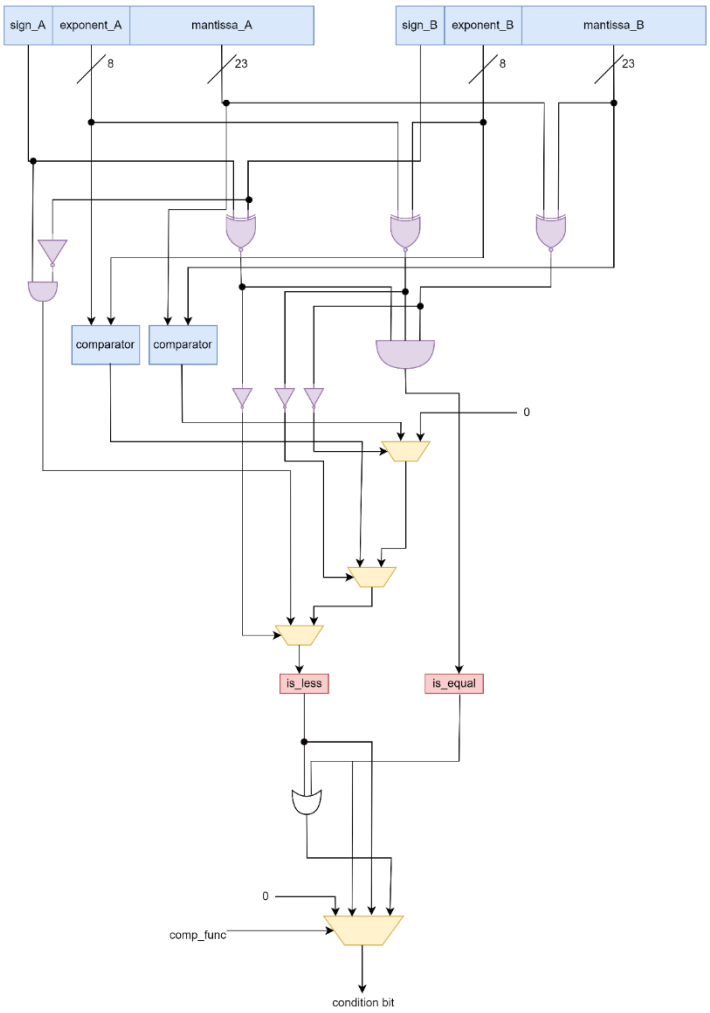

Figure 4. 21 : Implementation of Compare instructions.

5. TESTING THE FPU DESIGN

Since the designed FPU was implemented in the HORNET, the testing procedure is proceeded exactly the same as the process followed by the people who designed the HORNET. The environment is set up as described in [2].

5.1 Compiling a RISC-V Program

The command used to configure the riscv-gnu-toolchain, unlike the previous one, is as follows.

- `/configure --prefix=/opt/riscv --with-multilib-generator="rv32i-ilp32;rv32imf-ilp32f--"`

The first part of the command that starts with “--prefix" sets the installation directory. the second part that starts with “--with" configures the installer so that both RV32I and RV32IMF libraries are installed.

In addition, the following command is used to compile the C programs for testing the FPU.

- `riscv32-unknown-elf-gcc fpu_test.c ../crt0.s –march=rv32imf –mabi=ilp32f –T ../linksc.ld –nostartfiles –ffunction-sections –fdata-sections –Wl,--gc sections –o fpu_test.elf`

Programs written in RISC-V assembly language can also be assembled using riscv-gnu-toolchain. For this, it is sufficient to input the assembly file as an argument. In that case, it is necessary to pay attention to the startup routines, such as initializing the stack pointer, at the beginning of the program.

5.2 Test Program

After the modules were tested individually on their own, a simple assembly program shown as in Figure 5.1 was prepared to check whether it worked properly with the HORNET. The purpose of this test was to check whether the desired floating-point instructions without any problems on HORNET.

```
_init:

    .text
     auipc   x2,0x2
     addi    x2,x2,-4 # 1ffc
     add     x8,x2,x0
     li      x15, 0x45ECF8FD          # a → 7583.1235 in decimal
     li      x16, 0xC3983EA1          # b → -304.4893 in decimal
     sw      x15, 0(x8)
     sw      x16, -4(x8)a
     flw     f15, 0(x8)
     flw     f16, -4(x8)
```

Figure 5. 1 : Assembly code that executes FLW instruction.

First, the address pointer is set and 2 random floating point numbers are saved in memory. Then, computational instructions in Figure 5.2, conversion and move instructions in Figure 5.3, and finally compare and classify instructions in Figure 5.4 are processed.

```
# COMPUTATIONAL INSTRUCTIONS
 fadd.s  f17, f15, f16          # c = a + b =  7278.6343 → 0x45E37513 in fp forma
 fsw     f17, -8(x8)            t
 fsub.s  f17, f15, f16          # c = a - b =  7887.613  → 0x45F67CE7 in fp forma
 fsw     f17, -12(x8)           t
 fmul.s  f17, f15, f16          # c = a * b = -2308980.0 → 0xCA0CEDD0 in fp forma
 fsw     f17, -16(x8)           t
 fdiv.s  f17, f15, f16          # c = a / b = -24.904402 → 0xC1C73C37 in fp forma
 fsw     f17, -20(x8)           t
 fsqrt.s f17, f15               # c = sqrt(a) = 87.0811317 → 0x42AE298A in fp forma
 fsw     f17, -24(x8)           t
 fsqrt.s f17, f16               # c = sqrt(b) = NAN → 0x7FC00000 in fp forma
 fsw     f17, -28(x8)           t
 fmin.s  f17, f15, f16          # c = fmin(a,b) = b = -304.4893 → 0xC3983EA1 in fp forma
 fsw     f17, -32(x8)           t
 fmax.s  f17, f15, f16          # c = fmax(a,b) = a = 7583.1235 → 0x45ECF8FD in fp forma
 fsw     f17, -36(x8)           t
```

Figure 5. 2 : Computational instructions in assembly format.

```
# CONVERSION AND MOVE INSTRUCTIONS
 fsgnj.s f17, f15, f16          # c = {sign_b, exp_A, sig_A} = -7583.1235 → 0xC5ECF8FD in fp format
 fsw     f17, -40(x8)
 fsgnjn.s f17, f15, f16         # c = {~sign_b, exp_A, sig_A} = 7583.1235 → 0x45ECF8FD in fp format
 fsw     f17, -44(x8)
 fsgnjn.s f17, f15, f16         # c = {(sign_a ^ sign_b), exp_A, sig_A} = -7583.1235 → 0xC5ECF8FD in fp format
 fsw     f17, -48(x8)
 fcvt.w.s x17, f15              # c = to_int(a) = 7583 → 0x00001D9F in signed format
 sw      x17, -52(x8)
 fcvt.w.s x17, f16              # c = to_int(b) = -304 → 0xFFFFFED0 in signed format
 sw      x17, -56(x8)
 fcvt.wu.s x17, f15             # c = to_unsigned_int(a) = 7583 → 0x00001D9F in unsigned format
 sw      x17, -60(x8)
 fcvt.wu.s x17, f16             # c = to_unsigned_int(b) = 304  → 0x00000130 in unsigned format
 sw      x17, -64(x8)
 fmv.x.w x20, f15               # moves data between fp and integer register banks
 fmv.x.w x21, f16
```

Figure 5. 3 : Conversion and move instructions in assembly format.

```
# COMPARE INSTRUCTIONS
    feq.s   x22, f15, f16           # c = (7583.1235 == -304.4893) = 0
    sw      x22, -68(x8)
    flt.s   x22, f15, f16           # c = (7583.1235 < -304.4893) = 0
    sw      x22, -72(x8)
    fle.s   x22, f15, f16           # c = (7583.1235 ≤ -304.4893) = 0
    sw      x22, -76(x8)

    # CLASSIFY INSTRUCTION
    fclass.s x22, f15               # c = fclass(a) = 0x00000040 → variable a is a positive normal number
    sw      x22, -80(x8)
    fclass.s x22, f16               # c = fclass(a) = 0x00000002 → variable b is a positive normal number
    sw      x22, -84(x8)
```

Figure 5. 4 : Compare and classify instructions in assembly format.

As shown in Figure 5.5, correct results are stored into the memory. As you can see, the commands work without any problems, but our test is quite simple compared to more complex programs such as the digital low pass filter program. But such programs require more program memory. For this reason, when we tried to increase the program memory of HORNET, we could not do it because of the errors we encountered due to Verilator tool. We managed to expand it with Vivado, but we didn't have enough time so we decided to do such a test. In future studies, more comprehensive tests and benchmarks for FPU can be made.

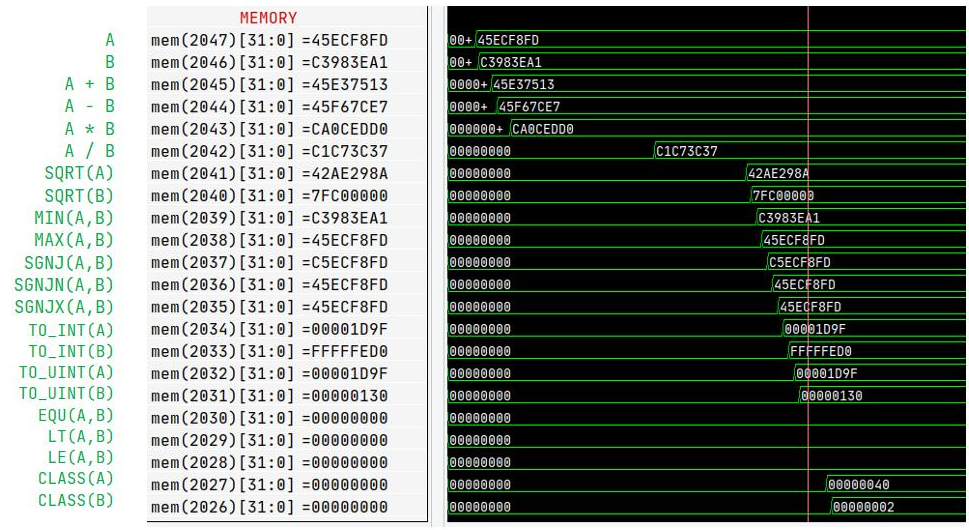

Figure 5. 5 : Results in HORNET memory.

6. REALISTIC CONSTRAINTS AND CONCLUSIONS

The importance of purpose-built hardware designs is increasing day by day. For this reason, processors are designed for different purposes in different architectures. While the HORNET RISC-V processor that we worked on was able to perform basic mathematical and logical operations, it became able to operate on floating-point numbers, which is an approximation method to real numbers.The design is free to use and extend. It can be used for research purposes, educational purposes, and even for personal projects.

6.1 Practical Application of this Project

Rısc-v processor are usally have some of the instruction sets as our processor does, since design can be used for further research for extending the instruction set by other instruction sets. In addition to this , it can be prepared for a chip tape-out. This could be an important contribution to the processor design initiative that is present in the country.

6.2 Realistic Constraints

6.2.1 Social, environmental and economic impact

RISC-V is a license-free ISA. This means that companies or groups do not have to pay for a licensing it is free to produce and/or sell RISC-V processors.

6.2.2 Cost analysis

The CAD tools used in this project are not free. Simulation and synthesis tools are quite costly. Fortunately, VLSI Lab in our faculty provided us with the necessary software licences.

6.2.3 Standards

The standard to follow in this project is the RISC-V ISA manual mainly and RISC-V manual considers IEEE 754 floating-point standarts for floating point numbers.‘F’ extesion implemented with IEEE 754 and RISC-V ISA manual standarts.

6.2.4 Health and safety concerns

This project does not involve health and safety concerns.

6.3 Future Work and Recommendations

There are different kind of opportunities for future work . To begin with, the remaining accumulate instructions , can be implemented. Combinational circuits that processor has can be optimized to ensure lower delays to achive higher clock frequencies. An interface can be provided for communication with peripherals. Lastly , chip tape-outs can also be done to test real life implementation of processor chip.

7. REFERENCES

[1] **Url-3** <https://riscv.org/about/history/>, date retrieved 10.06.2022

[2] **Y. TOZLU and Y. YILMAZ** Design and Implementation Of A 32-Bit Rısc-v Core, June 2021

[3] "754-2008 - IEEE standard for floating-point arithmetic," IEEE / Institute of Electrical and Electronics Engineers Incorporated,2008

[4] "The RISC-V Instruction Set Manual, Volume I: User-Level ISA, Document Version 20191213", Editors Andrew Waterman and Krste Asanovic, RISC-V Foundation, December 2019.

[5] **B. Parhami**, *Computer Arithmetic and Hardware Designs*, Oxford University Press, 2000.

[6] **S.F., Oberman and M.J. Flynn,** "Design Issues in Division and Other Floating-Point Operations," *IEEE Trans. Computers*, vol. 46, no. 2, pp. 154-161, Feb. 1997.

[7] **Oberman, S. F., & Flynn,** M. J. (1997). Division algorithms and implementations. IEEE Transactions on Computers, 46(8), 833–854. https://doi.org/10.1109/12.609274.

[8] Yamin Li and Wanming Chu, "A new non-restoring square root algorithm and its VLSI implementations," Proceedings International Conference on Computer Design. VLSI in Computers and Processors, 1996, pp. 538-544, doi: 10.1109/ICCD.1996.563604.

[9] **I. Korean** *Computer Arithmetic Algorithms*, 2nd ed, A. K. Peters/CRC Press, 2002.

[10] **Patterson D. and Hennessy John L.,** *Computer Organization and Design RISC-V Edition,* 2nd ed. Morgan Kauffman, 2020.