Instituto Tecnológico y de Estudios Superiores de Occidente

Reconocimiento de validez oficial de estudios de nivel superior según acuerdo secretarial 15018, publicado en el Diario Oficial de la Federación del 29 de noviembre de 1976.

Departamento de Electrónica, Sistemas e Informática

Maestría en Diseño Electrónico

RISC-V Floating Point Unit

TRABAJO RECEPCIONAL que para obtener el **GRADO** de **MAESTRO EN DISEÑO ELECTRÓNICO**

Presenta: **RICARDO ALFONSO ALFARO GÓMEZ**
JULIÁN ALVARADO LAGUNES

Director **JOSÉ LUIS PIZANO ESCALANTE**
OMAR HUMBERTO LONGORIA GÁNDARA

Tlaquepaque, Jalisco. 15 de septiembre de 2024.

ACKNOWLEDGMENT

Ricardo and Julian would like to express their deepest gratitude to their advisors, Dr. Jose Luis Pizano Escalante and Dr. Omar Humberto Longoria Gándara, for their patience, guidance, feedback and support throughout the research process. They also extend their thanks to Intel Corporation for the financial support that made this achievement possible.

Ricardo wishes to express his heartfelt gratitude to his wife Imelda for her unwavering support, encouragement, and for maintaining his motivation throughout this journey. He also extends special thanks to his mother and sister, whose inspiration and encouragement were instrumental in his pursuit of academic excellence.

Julian would like to acknowledge his wife Sarahi for her support in completing this thesis, and to his parents and sister for their continuous support throughout his studies and life, their guidance and care were key to achieving this grade.

ABSTRACT

This document explores the representation and processing of arithmetic data, with a focus on Floating Point (FP) operations for RISC-V architecture. It introduces the IEEE 754 standard for FP representation, detailing its evolution and significance in achieving portability and reliability in hardware units. Challenges associated with FP operations, such as high storage and execution costs, are addressed, necessitating dedicated Floating Point Units (FPUs) to optimize performance. The document proposes the implementation of an FPU for RISC-V architecture as a means to align with industry standards and contribute to scientific advancements. It discusses various number representations, the IEEE FP standard, and the role of FPUs within processor designs. Additionally, it outlines the problem statement, objectives, and expected impact of FPU implementation, providing insights into the division and square root algorithms used for FPU implementation. Overall, this document provides a comprehensive overview of FP operations, the IEEE 754 standard, FPU implementation challenges, and the significance of RISC-V architecture in contemporary computing.

LIST OF FIGURES

LIST OF TABLES

LIST OF ACRONYMS AND ABBREVIATIONS

RISC	Reduced Instruction Set Computer
ISA	Instruction Set Architecture
FPU	Floating Point Unit
FP	Floating Point
FxP	Fixed Point
ALU	Arithmetic Logic Unit
LSB	Least Significant Bit
MSB	Most Significant Bit
FPGA	Field Programmable Gate Array
LUT	Look-Up Table
FSM	Finite State Machine
ALM	Adaptive Logic Module
RM	Rounding Mode
IP	Intellectual Property

1. INTRODUCTION

Arithmetic data has different representations according to what an application may need, differing on how exact the result produced by an ALU must be to be compliant with the design. FP allows for dynamic ranges, which is the most optimal for high-resolution programs.

IEEE 754 is an industry standard for representing FP numbers in computers first published in 1985, this is being implemented by the industry and allowing hardware units to be portable and reliable. New versions have been published in 2008 and 2019, with the former extending the previous standard while merging in IEEE 854 for radix-independent FP and the latter being a revision for new recommended operations and clarifications of the previous version. It defines the single precision format with a width of 32 bits and the double-precision format with a width of 64 bits. The standard defines the significand range, exponent fields' biases, and how the FP result should be interpreted.

Performing FP operations is costly in storage and execution time even if these parameters are subjected to the program being executed: FP designs require a specific data path that must include dedicated resources such as specialized ALUs as well as additional logic dedicated to redirecting the data back to the processor. This is the main reason why FPUs must try to achieve high throughputs with low latency, designs must be able to perform large tasks in the least time possible. FP operations must be offloaded to dedicated FPUs to allow the processor to perform fewer demanding operations, which increases the design's overall performance. A RISC focuses on an optimized set of instructions dedicated to performing specific instructions focused on a load/store architecture. RISC-V is a joint effort in the semiconductor industry whose goal is to deliver an extensible ISA that will become the lead standard for computing design. RISC-V takes an approach that allows and promotes open code implementations that are simple, modular, extensible, stable, and legible.

FPUs have restrictions that are a consequence of whichever design they are attached to, common examples being dice area and time latency. This project is focused on implementing a fully synthesizable 32 bits FP unit based on the RISC-V ISA using single precision. This is done to develop a baseline for future projects based on microprocessors since RISC-V is an ISA that multiple industry-leading organizations are part of.

1.1. Background

FP operations consist of systems that perform operations with real numbers of varying sizes that require fast processing times, this trade-off makes these types of operations to be costly in storage and execution time. For this reason, FPUs have emerged to offload processing workloads from the microprocessor. The main objective of FPUs is to have high throughput with low latency, it is also desired that the necessary area for the unit is less than the area required for other generic units that could perform the same tasks such as embedded multipliers.

Multiple FP designs have been developed to this day, with different organizations and architectures having their purpose to find a balance between computing time and available resources. Due to the rise of RISC-V efforts, it is expected that future architectures will be implemented using this ISA.

1.2. Justification

The RISC-V organization is set to develop an ISA that will be the new standard for microprocessor designs for the years to come. Their ISA allows and promotes open code implementations that are simple, modular, extensible, stable, and legible. Developing an implementation with these characteristics provides a scientific edge and keeps ITESO on the vanguard as an educational organization that is up to date with industry tendencies and standards.

Moreover, it is expected that this application can be reused by future students so it keeps being enhanced based on the already published ISA for RISC-V.

1.3. Statement

FP is a notation used by logic units targeted for high-precision results, being noted as a requirement for all kinds of applications ranging from scientific development programs or emergency equipment to high-end commercial products such as TVs and video game computing. The units that perform these operations have unique restrictions that are a consequence of whichever design they are attached to, some examples would be dice area and time latency. Typically, these types of units require multiple LUTs, this being a native feature for any FPGA implementation. Due to how complex LUTs can be, it will always be of interest to optimize as many processes as possible to mitigate the impact on performance. FPGA FPUs are also limited by other design specifications, such as clock rate, location within the microarchitecture of the design, and available area.

In practice, numbers and arithmetic results used in different computation fields are expected to be efficient, standardized, and have accessible storage requirements. Due to its nature, FP representations are slower and less accurate than their FxP counterparts but can handle a larger range of numbers [1].

Achieving a proper balance that enables high throughput with low latency is always a goal for FPU FPGA implementations since the design space differs from VLSI [1]. Environment constraints and limitations must be accounted for through hardware optimization. Furthermore, algorithm implementations dedicated towards performance are greatly advised to mitigate the possible impact on available resources for FPGA [2]. Area limitations must also be considered; however, research shows that adding an embedded FPU provides an average reduction in area of 54.2% compared to an FPGA enhanced with embedded 18 bits × 18 bits multipliers [3].

An Implementation example for an out-of-order RISC-V processor design uses pipelined stages which vary depending on the operation being executed. The design features a decoder that redirects instruction information to FPU ALU. All instructions are executed in parallel and out of order, which offloads large computational instructions from the main processor. FPU adder has 5 stages, FPU divider has 31 stages, FPU square root has 31 stages, FPU multiplier has 17 stages for single precision and 9 stages for double precision, FPU MAC has 10 stages and FPU comparator has 3 stages. Performance analysis shows 55,661 LUTs with a frequency of 180MHz [4].

1.4. Hypothesis

An efficient implementation of an FPU for a RISC-V architecture will represent an impact on the execution time of required operations.

1.5. Objectives

- General Objective

Implementation of a 32 bits FPU based on the RISC-V ISA.

- Specific Objectives
    - A precise approximation from a hardware implementation of FP operations based on single precision.
    - RISC-V compliance.
    - Implement a fully synthesizable design so all written code is translated into logic that FPGAs can understand.

1.6. Scientific or technological novelty, contribution

The objective of implementing an FPU for a RISC-V architecture is to have an application that ITESO as an organization has ownership of, marking a baseline for future projects based on microprocessors since RISC-V is an ISA that multiple industry-leading organizations are part of.

2. BACKGROUND/RELATED WORK

2.1. Number representation

The number representation is an important subject when working with arithmetic, as well as operations, some examples of operations are multiplication, subtraction, addition, etc.

To execute the already mentioned operations, the algorithm's complexity relies on the number representation, this also affects the circuit complexity to perform the algorithms. Another point to consider when choosing a number representation besides the circuit complexity to perform the algorithms is how to interface this arithmetic information with the external world or between circuits [5].

2.2. Natural Numbers

1.1.1 Integers

The system representation "sign-magnitude" is the most organic way to express an integer, but when implementing arithmetic operations this system representation might not be that convenient [5].

2.1.1 Sign-Magnitude Representation

All integer numbers can be expressed with a positive or negative sign (+ or -).

These natural numbers can be expressed in the binary system and add an extra digit to represent a negative sign (0) or positive sign (1) [5].

2.3. Real Numbers

For real numbers, there exist two approximation systems: FxP and FP.

FxP is just an extension for the integer system that allows representing a reduced set of numbers with a certain precision.

On the other hand, the FP system can represent an extensive set of numbers with a certain level of precision [5].

When representing numbers with a wide range and certain precision, using the FP representation is ideal. This is the most used representation in digital systems for the arithmetic of real numbers [6].

2.4. Fixed Point

When using the FxP system the same range of numbers can be represented in a positive and negative range and the maximum absolute error is calculated using *ulp* with truncation and *ulp*/2 with rounding.

Below is an example of the disadvantages when using the FxP system.

A = (0000 0000 0000 0000. 0000 0000 0000 0001) Small number $2^{-16} \approx 0.000015$

b = (0111 1111 1111 1111. 1111 1111 1111 1111) Large number $2^{15} = 32768$

Due to truncation, the representation error is more significant for a than for b.

Also is impossible to represent a^2 and b^2, when trying to calculate them we fell into an overflow (b) and underflow (a) [6].

2.5. Floating Point

The FP representation is divided into 4 components:

- Sign
- Significand S
- Exponent base *b*
- Exponent *e*

Exponent base *b* is not explicit with its representation but normally is a power of 2, and 10 for decimal arithmetic.

$$x = \pm s \times b^e \qquad \text{or} \qquad \pm \text{significand} \times \text{base}^{\text{exponent}}$$

Figure 1 depicts the representation of a floating-point (FP) number, which is composed of three main elements. The sign specifies whether the number is positive or negative. The significand (or mantissa) includes the significant digits, reflecting the precision of the number. The exponent scales the number by determining the power of the base, thereby adjusting the overall magnitude of the significand.

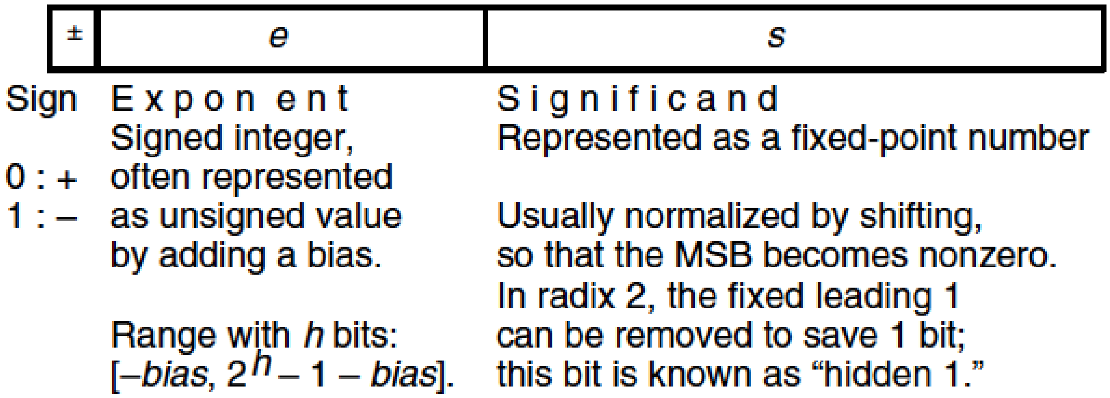

Figure 1. Typical FP number format. [6]

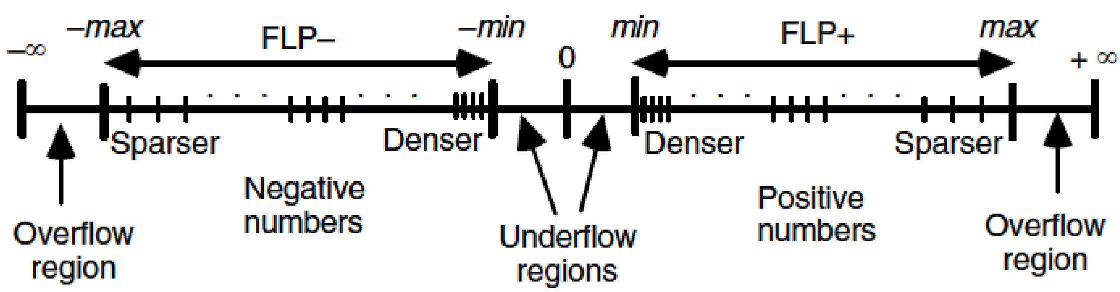

Figure 2. Subranges and special values in FP number representations.[6]

The exponent is stored in a specific field of the FP number, using a biased encoding scheme to handle both positive and negative exponents. The bias is a constant value added to the actual exponent to ensure the stored exponent is always non-negative. To find the actual exponent used in calculations, you subtract the bias from the stored exponent value.

In an FP system the set of values is represented by {-*max*, -*min*} and {*min*, *max*} as in the below example:

max = significand × $a^{largest\ exponent}$

min = significand × $a^{smallest\ exponent}$

For FP representation, Figure 2 shows the various subranges and the number distribution pattern that contains three singular values −∞, 0, and +∞.

When the computation result is greater than max or less than -max an overflow is caused.

If a computation result is in the range of {0, min} or {-min, 0} [6].

Below Table 1 shows an FP representation example of the smallest positive number which is around $\approx 1.18x10^{-38}$. In Table 2 is shown the largest positive number $\approx 3.40x10^{38}$.

Table 1. Smallest positive number.

Sign bit	Exponent part	Mantissa part
0	10000010	00000000000000000000000

Table 2. Largest positive number.

Sign bit	Exponent part	Mantissa part
0	01111111	11111111111111111111111

In FP math, precision is key, and three important components— the guard bit, the round bit, and the sticky bit— play crucial roles in maintaining it. The guard bit helps prevent errors by examining bits beyond the required precision, while the round bit determines whether to round up or down based on these extra bits. The sticky bit, on the other hand, captures any remaining bits that might influence the rounding decision. Together, these bits work to ensure calculations remain accurate and consistent despite the limitations of binary numbers. In an iterative division algorithm, rounding is achieved by computing two additional quotient bits (the guard and round bits) and using the remainder as the sticky bit [7].

2.6. The IEEE Floating Point Standard

The FP system representation is also an IEEE standard [IEEE85] or “IEEE 754” that was developed in 1984. This Standard was early adopted by computer manufacturers even before it was officially released. A revision of this standard called “IEEE 754R” was released in 2008.

In the 2008 revision, some changes and additions were included, like the incorporation of 16 bits and 128 bits format for the binary base, and the FMA (fused multiply-add). Figure 3 represents the formats in binary base for the IEEE 754-2008 standard. There are 2 width representations: 32 bits for single precision (short) and 64 bits for double precision (long). Either for 32 and 64 bits formats we have 8 and 11 bits exponent representation and a bias for exponents of 127 and 1023 per each one. A single bit which is always 1, represents the significand and is in the range (1) For 32 and 64 bits representation, 23+1 and 52+1 express the significand width and explain the hidden bit functionality that contributes to improving precision [5].

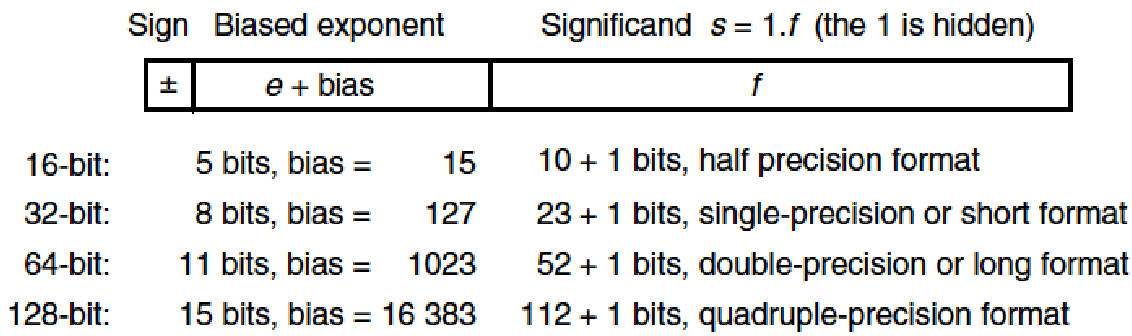

Figure 3. The IEEE 754-2008 binary FP number representation formats. [6]

2.7. Floating Point Unit

For the wide variety of data processing applications (voice, image, etc.), which use a vast range of values and need a relatively high precision, the FP representation is implemented instead of FxP or the integer system [5].

A FPU is a block added to the processor designed to handle operations on FP numbers. In contrast, the ALU handles the arithmetic and bitwise operations of integer binary numbers. Both the FPU and the ALU receive 2 operands and execute the desired operation based on the control signals. The control would indicate which operation needs to be performed based on the instruction.

2.8. RISC-V “F” Standard Extension

An addition to the instruction set, for single-precision FP operations denoted as “F”, incorporates computational instructions customized for single-precision FP calculations, conforming to the IEEE 754-2008 arithmetic standard [8].

The “F” extension includes 32 FP registers, labeled F0–F31, each spanning 32 bits, alongside an FP control and status register named FCSR. This register holds the operational mode and exception status of the FPU. Bits 31 to 8 of FCSR are reserved, while bits 7 to 5 hold the RM known as FRM. Finally, bits 4 to 0 hold the exception result from the last operation [8].

FP operations employ either a predefined RM encoded within the instruction or a dynamic RM stored in FRM. A value of 111 in the instruction’s RM field indicates the selection of the dynamic RM stored in FRM.

2.9. Division Algorithm

There are many division algorithms available nowadays. They are classified as digit recurrence, functional iteration, very high radix, LUTs, among others. Recurrence type refers to those that do iterative subtractions and can be split into restoring and non-restoring. Functional iterations are those which predict or multiply the outputs to get the closest approximation. Examples of functional iterations are the Newton-Raphson algorithm, series expansions, and Taylor series. Examples of very high radix are the Radix-10 and Cascaded radix algorithms. Finally, LUTs are those which use pre-computed values or distributed tables to improve performance. It is found that, for low-cost implementations where chip area must be minimized, digit recurrence algorithms are suitable [9].

When conducting FP division, the process entails dividing the mantissas using a fixed-point or integer division algorithm and subtracting their exponents. Initially, the sign of the result is determined by assessing the signs of the numbers being divided. Subsequently, the absolute difference between the exponents, represented as E, is calculated and adjusted by either adding E to the bias when the divisor's exponent exceeds that of the dividend or subtracting E from the bias in the reverse scenario. Following this, the mantissa of the divisor is divided by the mantissa of the dividend, adding the hidden bits. Should the result contain a leading zero, it undergoes normalization by shifting left and because of this normalization process, the exponent is decreased by one.

The division algorithm used for this FPU implementation is based on the restoring algorithm [10] which is one of the easiest to understand and translate to hardware. The algorithm shown in Figure 4 takes n cycles, where n is equal to the number of output bits, which would be 23+1 for single precision, plus any extra cycles needed for rounding computation. It starts assigning initial values to Q, M, A, and Count. The count will increase for each iteration until it reaches the value of n. Q starts being equal to the dividend. M holds the value of the divisor. A variable starts as 0. For each iteration, the A and the Q are shifted left, and the MSB of Q is moved to the less significant bit of A. In each iteration, A is compared versus the divisor M. If A < M, then the less significant bit of Q will be 1 and A will take the value of A-M. If A >= M, then the less significant bit of Q will be 0 and A holds its current value. After n cycles, Q would have the quotient and A would hold the remainder.

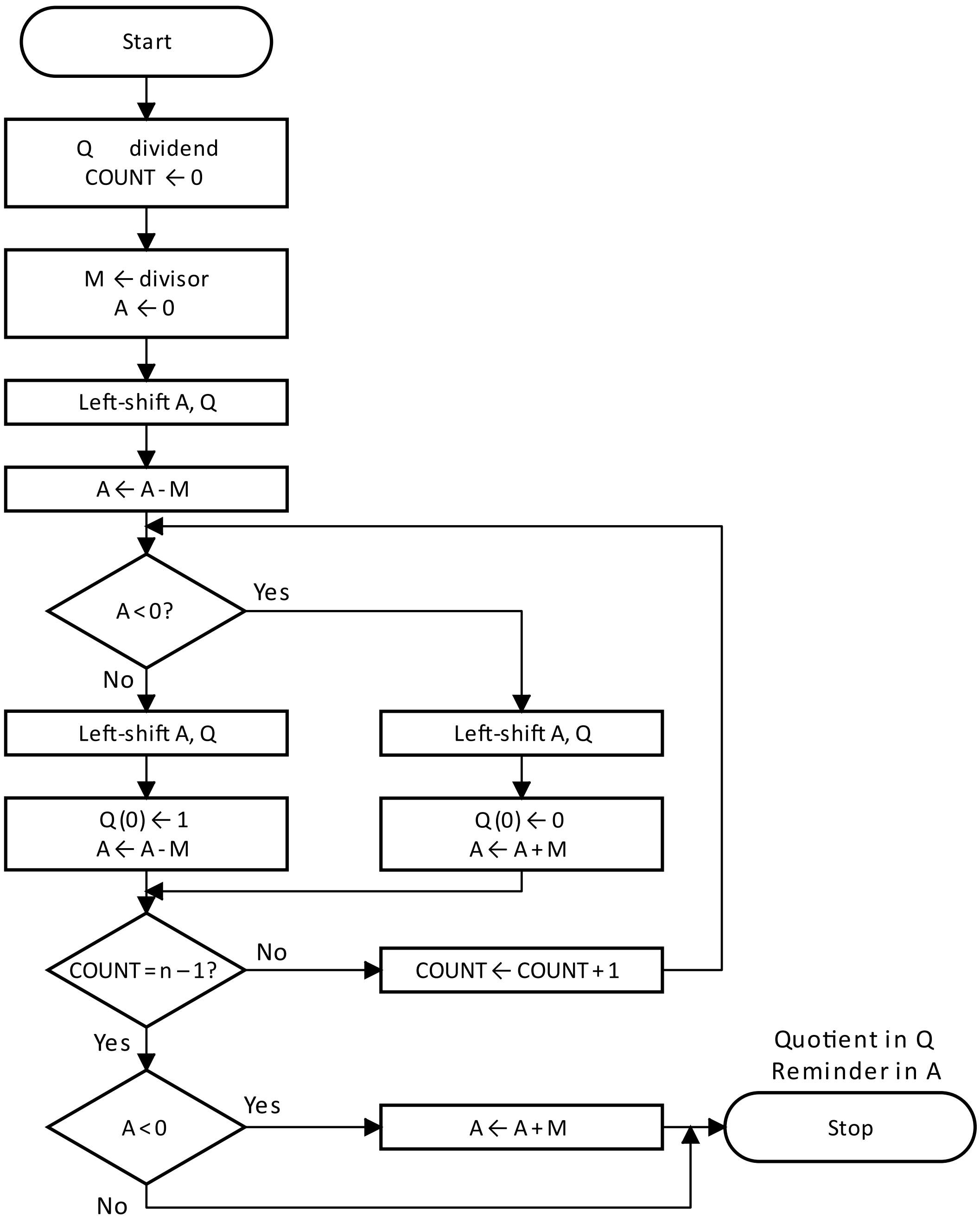

Figure 4. Restoring division algorithm flow diagram [10].

2.10. Square Root Algorithm

The square root algorithm used in this FPU implementation is based on the research article "A new non-restoring square root algorithm and its VLSI implementations" presented in [11], the previously mentioned implementation was chosen among other algorithms because of the reduced hardware implementation compared with other square root restoring implementations, requiring only a subtractor/adder and a couple of shift registers. As a consequence of the reduced hardware needed to build this algorithm the clock rate at which it can operate is higher than other more complex hardware implementations.

The algorithm implementation generates a correct resulting bit and a partial reminder bit every iteration, in each cycle, an addition or subtraction operation is performed depending on the previous operation resulting bit sign. Once all iterations are completed, the partial reminder magnitude is evaluated, if it is positive the partial reminder vector is used, if it is negative the partial reminder vector needs to be adjusted by an addition. The new non-restoring algorithm algorithm is explained in Figure 5.

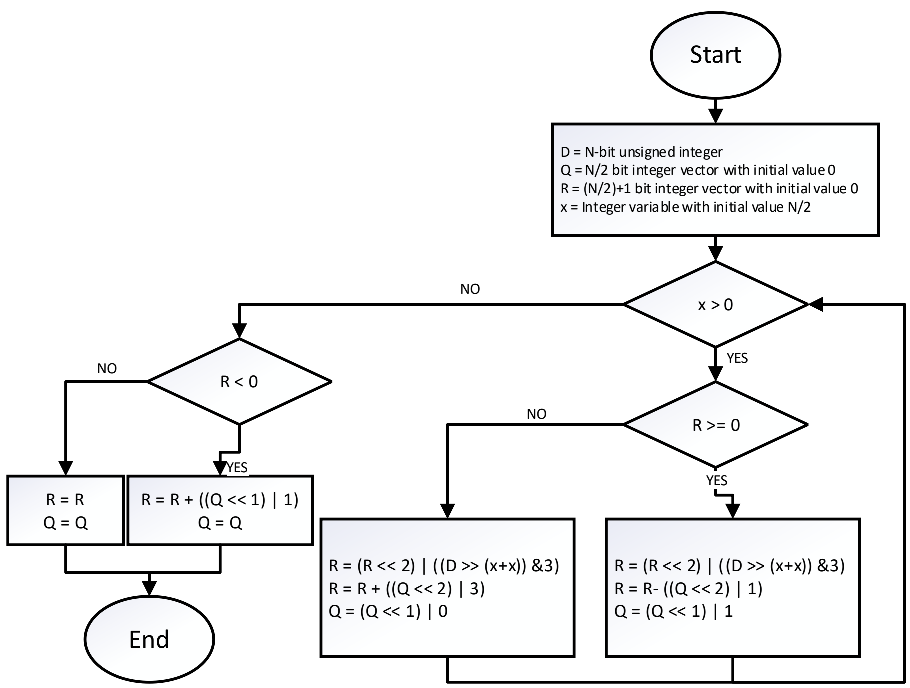

Figure 5. New non-restoring algorithm flow diagram is presented in [11]

3. MATERIALS AND METHODS

This chapter delineates the methodology employed in designing the FPU. It begins with an overview of the design requirements, including adherence to relevant standards and the range of supported operations. Following this, the chapter describes the top-level RISC-V architecture to provide context for the integration of the FPU.

The discussion then transitions to a high-level representation of the FPU architecture, detailing its principal inputs and outputs. A comprehensive analysis of the internal components follows, including the decoder, arbitration logic, and operation units. Each component is examined in detail, with a focus on the implementation algorithms and the rationale behind key design decisions.

The chapter concludes with an explanation of the integration process of the FPU within the RISC-V processor, specifying the modifications required. Finally, the chapter describes the testbench used for functional verification, ensuring the correctness and reliability of the FPU design.

3.1. Requirements

The FPU implementation will be single precision following the IEEE 754 standard and the RISC-V “F” standard. Table 3 shows the representation of a number in this format.

Table 3. Single precision number representation format.

Sign	Exponent	Mantissa
[bit 31]	[bit 30:23]	[bit 22:0]

The FPU should support the following RISC-V operations:

1. FLW
2. FSW
3. FDIV.S
4. FSQRT.S
5. FMIN.S
6. FMAX.S
7. FEQ.S
8. FLT.S
9. FLE.S
10. FCLASS.S
11. FRCSR
12. FSCSR

3.2. RISC-V pipeline top level

Figure 6 depicts the integration of the FPU implementation within a processor following the RISC-V architecture. This processor operates through a 5-stage pipeline consisting of Fetch, Decode, Execute, Memory access, and Write-back stages. Notably, both the FPU and ALU are situated within the Execution stage. Additionally, in the Instruction Decode stage, enhancements include the incorporation of the new 32 bits FCSR register and the FPU register file to accommodate the 32 FP registers as specified in the RISC-V "F" Standard Extension for single precision FP operations.

In RISC-V, store and load operations are fundamental instructions that facilitate the movement of data between the processor's registers and memory. These instructions are crucial for the execution of programs as they enable the manipulation of data stored in memory. The FLW instruction loads a single precision FP value from memory into FP register rd. FSW stores a single precision value from FP register rs2 to memory [8]. To support these 2 FP operations, multiplexors need to be added connecting the data memory to the FPRF.

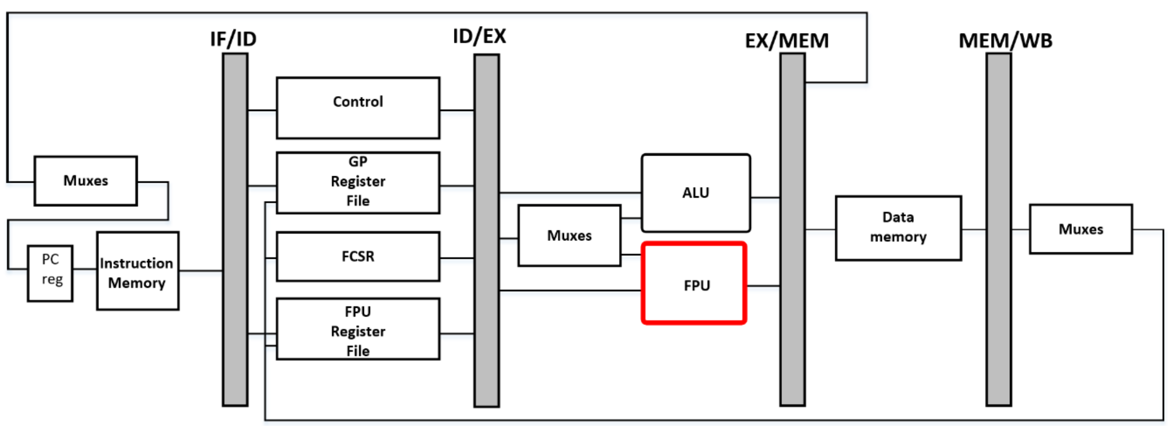

Figure 6. RISC-V Pipeline microarchitecture with FPU.

3.3. FPU top level

Figure 7 presents the inputs and outputs of the FPU top-level hierarchy.

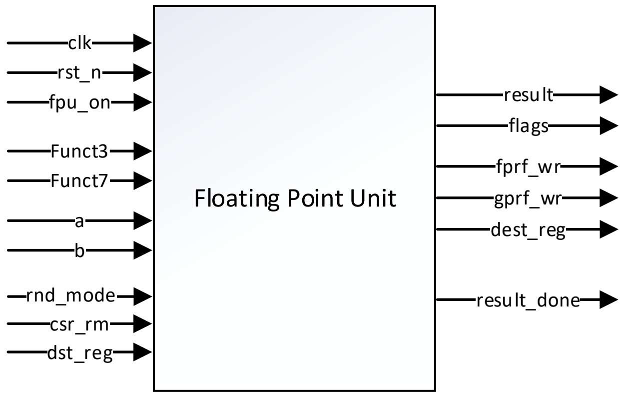

Figure 7. FPU black box diagram

3.4. FPU Microarchitecture

The FPU top-level module consists of 4 main FP blocks, one per each operation or set of operations:

- Comparison.
- Square root.
- Division.
- Classification.

Also, combinational logic is needed to decode the operation type and to classify the operands.

The FPU decoder uses the Funct 3 and Funct 5 extracted from the instruction word as inputs to decode the type of operation to perform and enable the correct FP block. The supported FP operations can have one or two input operands, and each operand needs to be classified (positive, negative, normal, etc) so the FPU can calculate the respective exception flags per each result. Finally, as an arbiter mechanism, a round-robin algorithm was implemented to manage the FPU main outputs (result, flags, exceptions) based on the different operation requests.

The FPU microarchitecture and all the arithmetic engines that this unit contains are presented in Figure 8. Additionally, Table 4 describes all the FPU inputs and outputs.

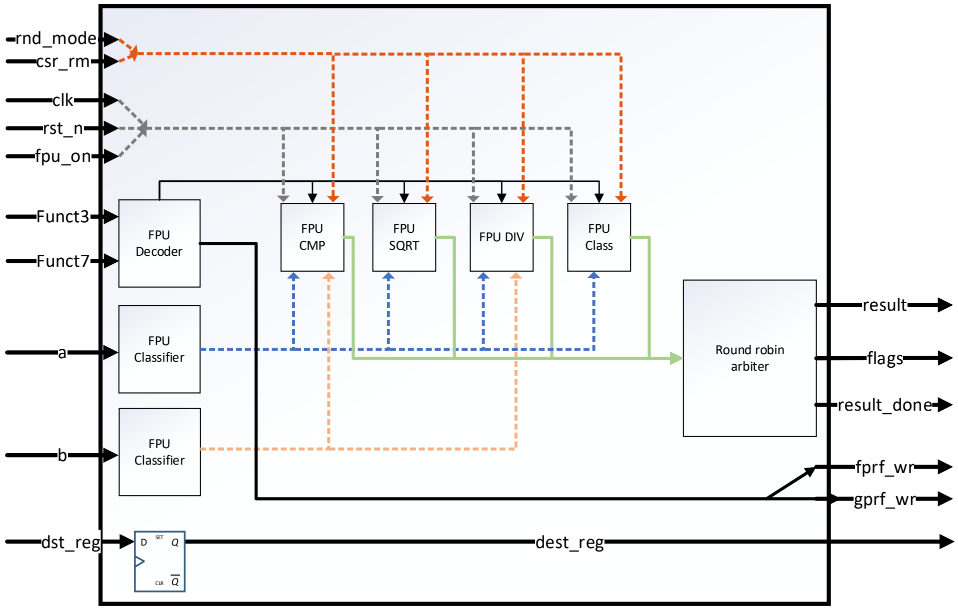

Figure 8. FPU Architecture

Table 4. FPU input and output description.

Signal name	size	Type	Description
clk	1	input	Main clock.
rst_n	1	input	Active low main reset.
fpu_on	1	input	FPU enable signal.
funct3	3	input	Funct3 from instruction, used to decode the FP operation.
funct7	7	input	Funct7 from instruction, used to decode the FP operation.
a	32	input	32 bits input operand.
b	32	input	32 bits input operand.
rnd_mode	3	input	RM encoding from instruction.
csr_rm	3	input	RM encoding from floating point CSR, is only used when rdn_mode is set to dynamic.
dst_reg	5	input	Register file number where the FP result would be stored.
result	32	output	FP operation result.
flags	4	output	Exception flag encoding bits from operation result.
fprf_wr	1	output	Write enable signal to indicate a result must be written to the FP register file.
gprf_wr	1	output	Write enable signal to indicate a result must be written to the general purpose register file.
dest_reg	5	output	Register file number where the FP result would be stored.
result_done	1	output	Asserted for a clock cycle when a result is available in FPU.

1.1.1 FPU Decoder:

The decoder unit module evaluates func_7 and func_3 bits from the FP instruction word, the decoding is done through a case statement where both inputs are concatenated and decoded, the case statement asserts the respective outputs to execute a specific FP operation, and the register file where the operation should be stored. Not all FP operations results are written into the FP register file, some operations are written into the general purpose register file. Table 5 describes the relationship between funct_7 and func_3, the FP operation, and the register file to target. The decoder has func_7 and func_3 as inputs and one output bit per operation decoded as shown in Figure 9.

Table 5. Decoder operation output.

{func_7,func_3}	FP operation output	Register file where the operation result would be stored
10'b0001100_111	Division	FP register file
10'b0101100_111	Square Root	FP register file
10'b0010100_000	F_min	FP register file
10'b0010100_001	F_max	FP register file
10'b0010100_010	F_eq	General purpose register file

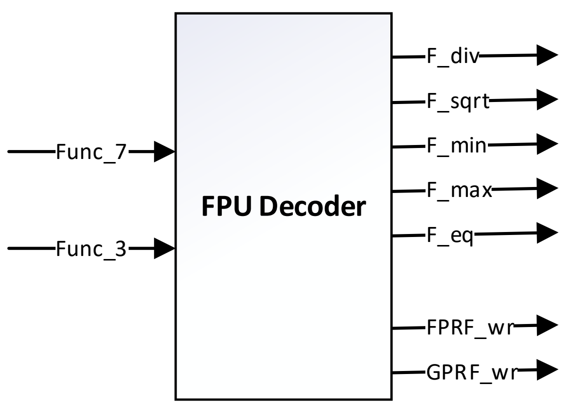

Figure 9. Decoder inputs and outputs.

2.1.1 *FPU Classifier*

The FPU classifier is a combinatorial module that inputs a 32 bits operand and evaluates the three different FP components: sign (bit 31), exponent (bits 30:23), and mantissa (bits 22:0). The output of this module is a 9 bits bus as presented in Figure 10. Each bit represents a different classification parameter for the input operand. The meaning of each output bit is described in Table 6.

Table 6. Classifier output encoding.

Bit 8	Bit 7	Bit 6	Bit 5	Bit 4	Bit 3	Bit 2	Bit 1	Bit 0
positive	negative	normal	zero	subnormal	inf	nan	quiet	signaling

It is important to note that an operand can qualify for more than one classification, i.e., a number can qualify as zero and positive as well, in that case, bits 8 (positive) and 5 (zero) are going to be asserted.

Table 7 shows the nine qualifications performed by the classifier logic to the 32 bits operand. If the qualification matches, then the respective bit gets asserted.

Table 7. Classifier qualifications.

Floating point operand qualification	Classifier output
Sign == 0	is_possitive
Sign == 1	is_negative
exponent != '0 && exponent != '1	is_normal
exponent == '0 && mantissa == '0	is_zero
exponent == '0 && mantissa != '0	is_subnormal
exponent == '0 && mantissa == '0	is_inf
exponent == '1 && mantissa != '0	is_nan
exponent == '1 && mantissa [22] == 1'b1 && mantissa[21:0] != '0	is_quiet
exponent == '1 && mantissa [22] == 1'b0 && mantissa[21:0] != '0	is_signalling

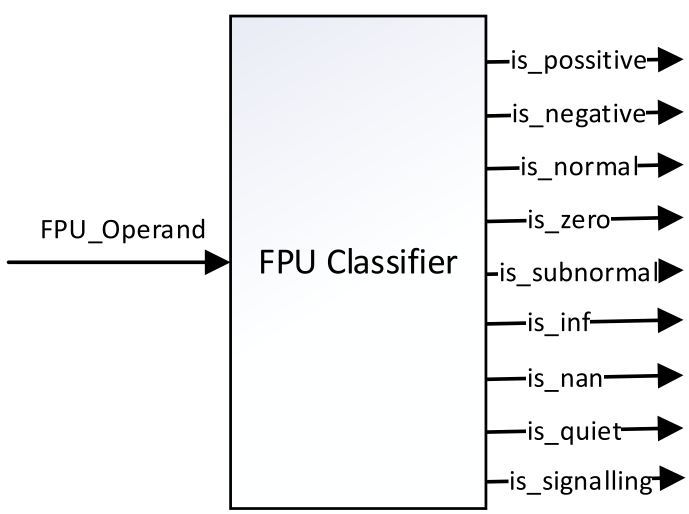

Figure 10. Classifier inputs and outputs.

3.1.1 Round unit

The FPU supports 5 different RM and one dynamic RM state:

- 000: Round to Nearest, ties to Even (RNE)
- 001: Round towards Zero (RTZ)
- 010: Round Down (RDN)
- 011: Round Up (RUP)
- 100: Round to Nearest, ties to Magnitude (RMM)
- 111: Dynamic

The FP RM can be set in two different ways, the first one using the FP RM input and the second through the FP CSR. The RM input has precedence over the FP CSR, the last one can only be used when the dynamic mode (3b'111) is set by the RM input, in that case, the FPU RM is chosen using the 3 bits CSR RM.

Table 8 describes how the FP RM input has precedence over the FP CSR bits to set the FP RM.

Table 8. Selected round mode based on inputs.

instr_rnd_mode [2:0]	csr_round_mode [2:0]	Selected round mode
3b'000	3b'XXX	Round to Nearest, ties to Even
3b'001	3b'XXX	Round towards Zero
3b'010	3b'XXX	Round Down
3b'011	3b'XXX	Round Up
3b'100	3b'XXX	Round to Nearest, ties to Magnitude
3b'111	3b'000	Round to Nearest, ties to Even
3b'111	3b'001	Round towards Zero
3b'111	3b'010	Round Down
3b'111	3b'011	Round Up
3b'111	3b'100	Round to Nearest, ties to Magnitude
3b'111	3b'111	Reserved

The FP round unit only affects the LSB from the 32 bits input (FP operation result before rounding). This bit is rounded when applicable, considering 3 additional one-bit inputs:

- guard_bit
- round_bit
- sticky_bit

Figure 11 shows the black box diagram for the round unit including the previously mentioned 3 bits and the 2 RM inputs.

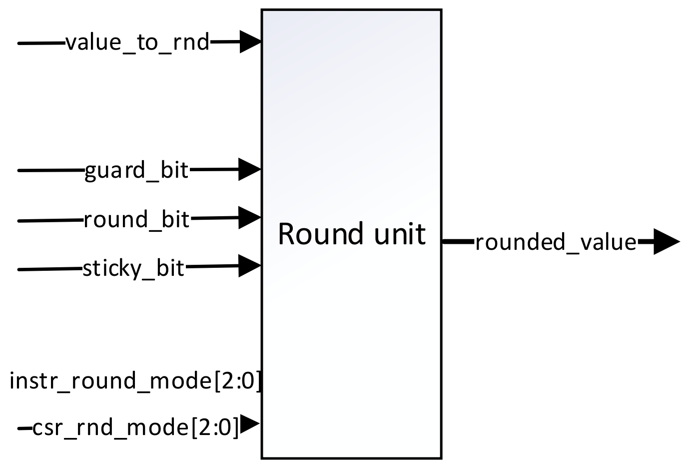

Figure 11. Round unit inputs and outputs.

After the RM was set, the guard, round, and sticky bits were used by the RM algorithm.

a) Round to Nearest, ties to Even

This RM evaluates the guard, round, and sticky bits. If the guard bit is set (1b'1), then the round and sticky bit are evaluated and if any of the previous two mentioned is set (1b'1), the LSB is set to (1b'1), otherwise the LSB remains the same.

b) Round towards Zero

Round towards zero does not affect the FP result LSB, and does not evaluate either guard, round, or sticky bits. If the LSB is 1b'0, remains 1b'0, same if the LSB is 1b'1 remains 1b'1.

c) Round Down

Round down evaluates through an OR operation the value of the guard, round, and sticky bits. If any of those bits is 1b'1 and additionally the FP result MSB (sign-bit [31]) is 1b'1 then the FP result LSB is rounded to 1b'1. Otherwise, the LSB does not change.

d) Round-Up

Round-up evaluates through an OR operation the value of the guard, round, and sticky bits. If any of those bits is 1b'1 and additionally the FP result MSB (sign-bit [31]) is 1b'0 then the FP result LSB is rounded to 1b'1. Otherwise, the LSB does not change.

e) Round to Nearest, ties to Max Magnitude

The round to nearest ties to magnitude RM only evaluates the guard bit, if this bit is set then the FP result LSB is rounded to 1b'1. Otherwise, the LSB does not change.

4.1.1 Division

The FPU division block is a wrapper that contains the division algorithm implemented in hardware plus some glue logic to match the FP format and IEEE 754 requirements. Table 9 defines the inputs and outputs of the block and Figure 12 displays the unit as a black box. The glue logic around the restoring division algorithm is detailed in Figure 13.

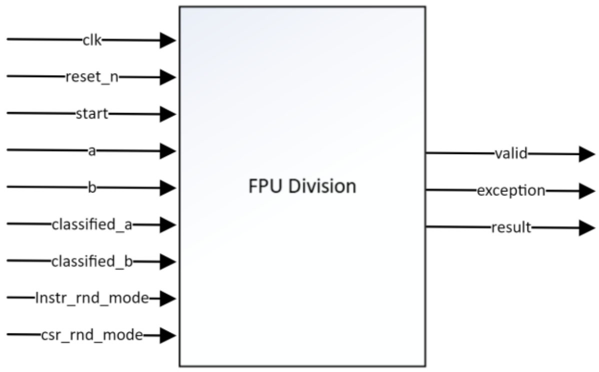

Figure 12. FPU Division black box

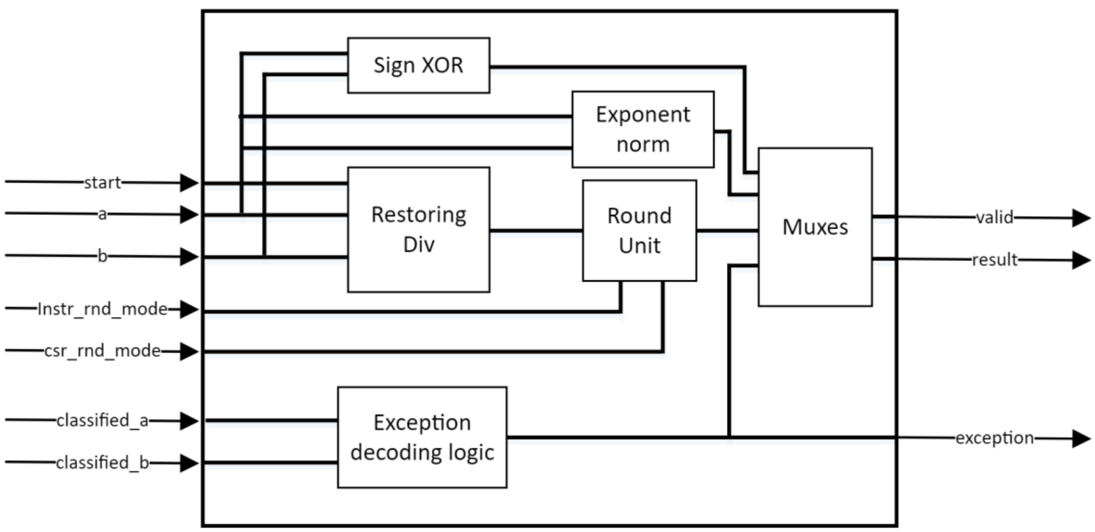

Figure 13. FPU Division microarchitecture

Table 9. FPU Division input and output description.

Signal or group name	size	Type	Description
clk	1	input	Main clock.
reset_n	1	input	Active low main reset.
start	1	input	Enable signal. Active one clock cycle per operation.

a	32	input	Operand A (to compare).
classified_a	9	input	Operand A qualifications from FPU classifier.
b	32	input	Operand B (to compare).
classified_b	9	input	Operand B qualifications from FPU classifier.
rnd_mode	3	input	RM from instruction.
csr_rm	3	input	RM from FP CSR.
valid	1	output	FP operation result is valid.
exception	5	output	FP exceptions after operation.
result	32	output	FP operation result.

The implemented hardware follows the restoring algorithm explained before. The main idea of the algorithm is to have a register of size 2*n and a subtractor. The register would contain AQ, and each iteration would shift left the whole register by updating the bit indexes. The used registers in this implementation are described in Figure 15.

Figure 14 shows the implementation of the division engine following the restoring algorithm. Every iteration the register AQ is shifted left. The subtractor inputs are A and M, to check if A-M < 0. If the MSB of the subtraction is high (set to 1), then the result of A-M is less than zero, and the portion of A in the register AQ will be updated to A-M and the portion of Q will be Q shifted with bit 0 equal to 1. If the subtraction result is positive, then the register AQ will be updated to the shifted AQ and the Q bit 0 will be equal to 0. The total number of iterations is based on the signal busy which comes from a counter unit that starts the count after the start signal is asserted. The counter is active for 27 cycles, which includes the hidden bit, 23 bits for the mantissa, and 3 bits for the guard, round, and sticky bits.

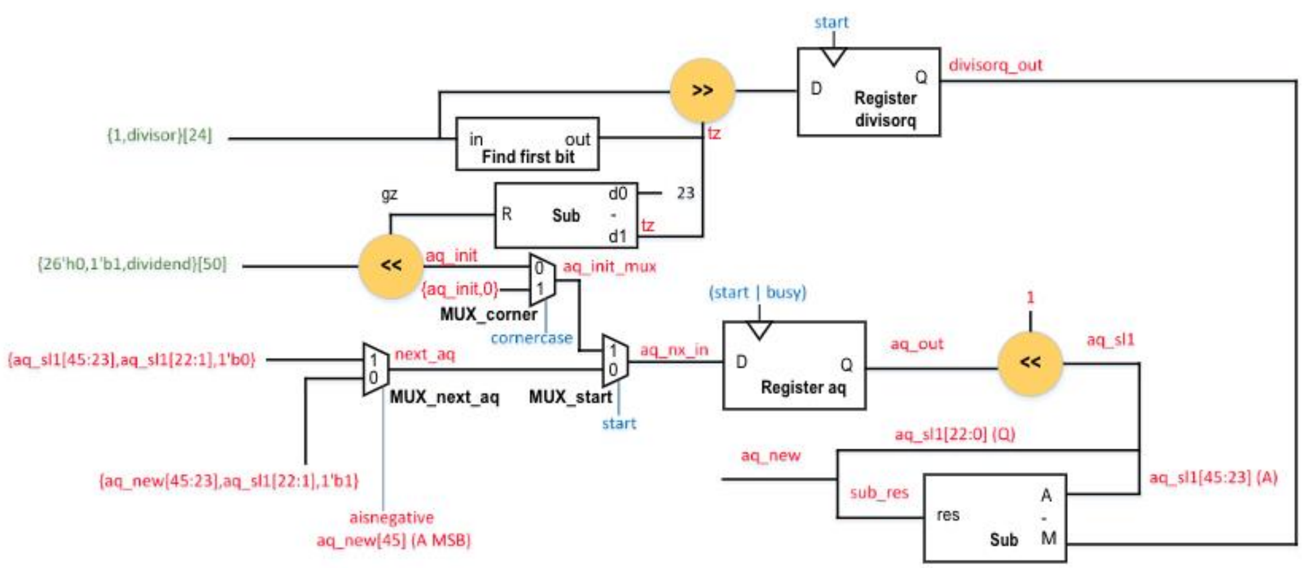

Figure 14. Restoring division microarchitecture

Figure 15 represents the different registers used in the implementation. The register AQ is the one that is shifted left each iteration. Where Q is sited in the 27 lowest significant bits and A-D is in the top 29 bits. When A-D is negative, the MSB (bit 55) is set to 1, expressing a negative number. The quotient result is taken from bits 25 to 3. While the bits GRS are taken from bits 2 to 0. The sticky bit is calculated by doing a logical OR of all the remainder bits.

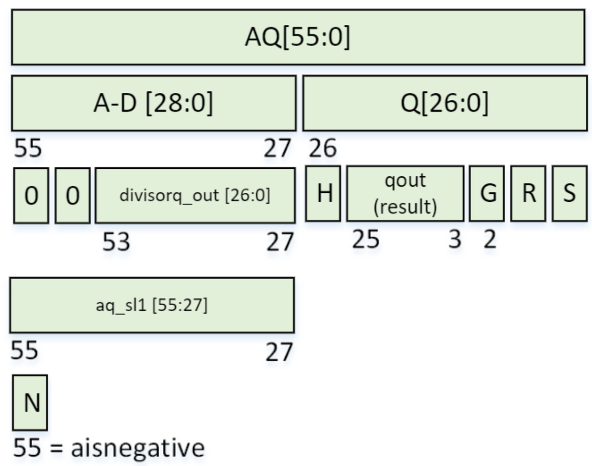

Figure 15. Registers used in restoring division implementation

Figure 16 shows the timing diagram for the division algorithm. The start signal enables the counter. In each iteration, a shift left and a subtraction is done following the algorithm steps. The quotient result is ready after 27 cycles, signaled by the valid signal. The results and exceptions are only stable while the valid signal is high.

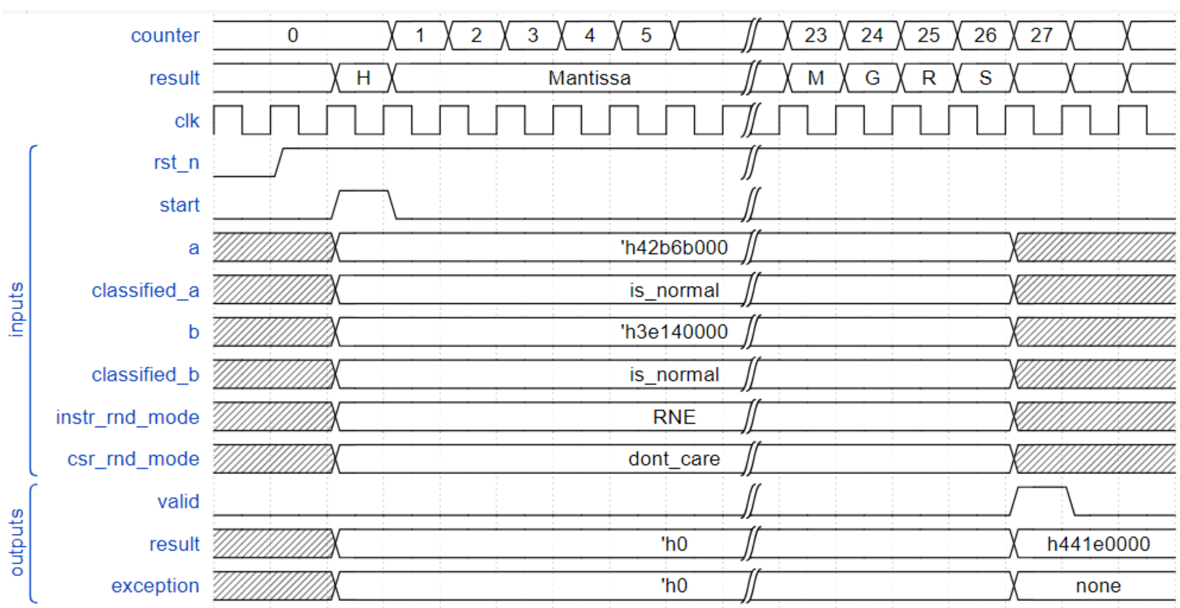

Figure 16. FP division timing diagram

Figure 17 illustrates the timing for an exception case. In this test, both the overflow and inexact flags are expected. Since the exception detection logic is based on the inputs A and B (divisor and dividend), the logic asserts the exception and valid signals one cycle after the start input goes high. The quotient result is obtained from the exception detection logic rather than from the restoring division engine.

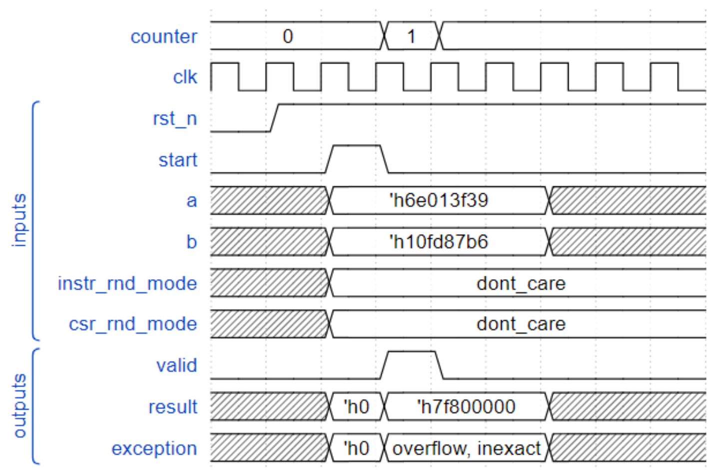

Figure 17. Timing diagram of an invalid division operation

5.1.1 Floating Point Square Root

In this chapter, the FP square root engine is described from top to bottom, starting with the FP implementation and all the glue logic required to handle the sign, exponent, and mantissa. Next, a quick overview and explanation of the 28 bits FxP square root hardware implementation, the one that computes the mantissa square root and generates the guard, round, and sticky bits needed for RM.

Figure 19 shows the FP square root microarchitecture, which includes all the wrapper logic and the 28 bits Fxp square root engine needed to perform a 32 bits FP single precision square root operation. This operation takes 29 clock cycles as shown in Figure 20. Presented in Figure 21 there is an exception case where an invalid input is computed as soon as the next clock cycle. This module calculates the FP square root of the 32 bits input and evaluates the 9 bits classification input to flag an exception, a start signal is required to initiate the FP operation. Finally, to choose the operation RM there are two options, the RM encoded in the instruction word, and if the instruction RM is set to dynamic, then the RM encoded in the FP CSR is used. Figure 18 is the block representation with all the inputs and outputs of this FP square root module and Table 10 describes the ports functionality.

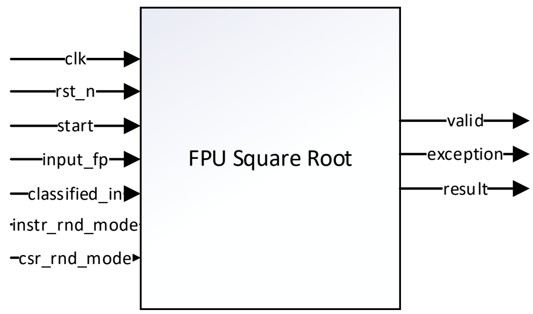

Figure 18. FPU Square Root black box

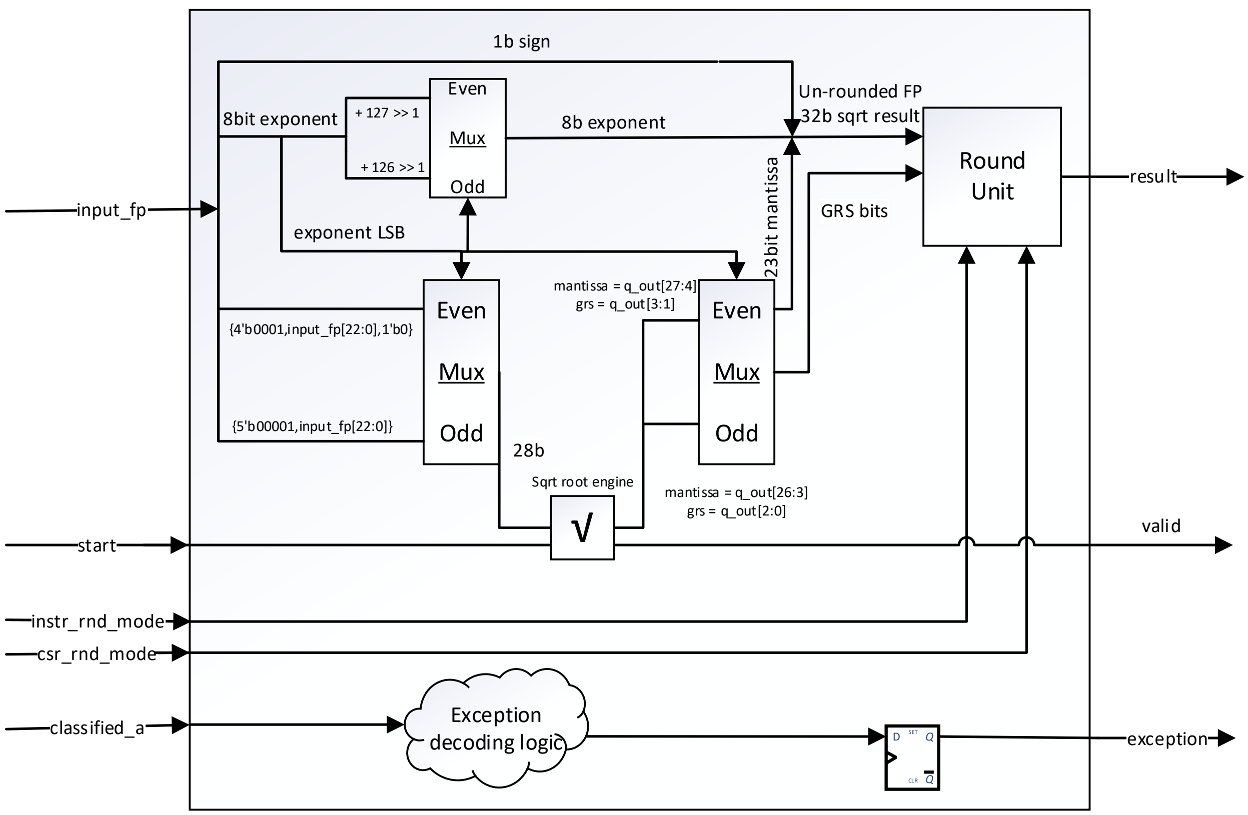

Figure 19. FP Square Root microarchitecture

Table 10. FPU Square Root input and output description

Signal or group name	size	Type	Description
clk	1	input	Main clock.
rst_n	1	input	Active low main reset.
start	1	input	Enable signal. Active one clock cycle per operation.
input_fp	32	input	Operand A (to classify).
classified_a	9	input	Operand A qualifications from FPU classifier.

rnd_mode	3	input	RM from instruction.
csr_rm	3	input	RM from FP CSR.
valid	1	output	FP operation result is valid.
exception	5	output	FP exceptions after the operation.
result	32	output	FP operation result.

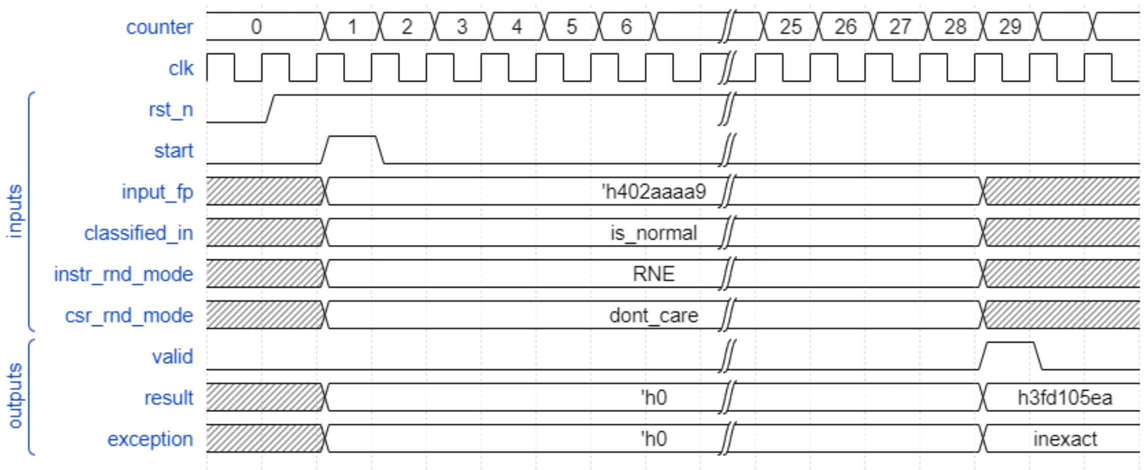

Figure 20. FP square root timing diagram

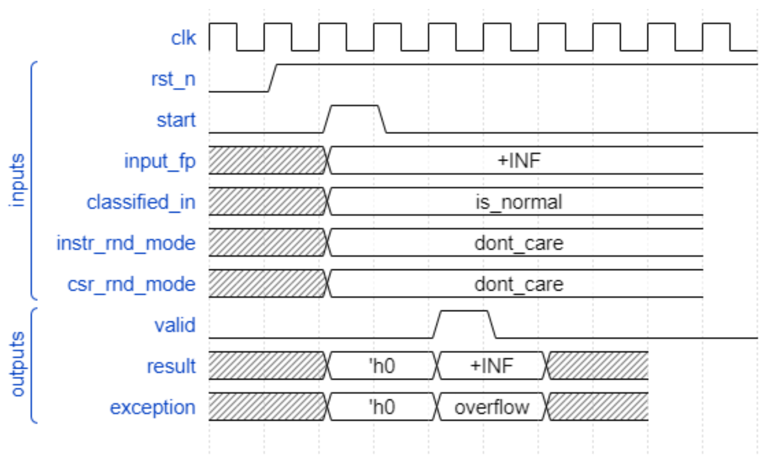

Figure 21. Timing diagram of an invalid square root operation

- Fixed Point square root engine

A 28 bits FxP square root module is implemented to calculate the adjusted mantissa value, as shown in Figure 19, the original 23 bits mantissa value coming from the FP input is adjusted to also generate the required guard, round, and sticky bits needed for RM. Figure 22 shows the hardware implementation of the 28 bits FxP square root module based on the new non-restoring square root algorithm described in [11] and explained in SECTION 2.10 of this document. This module takes 28 clock cycles to generate the 23 bits square root mantissa and the 3 additional guard, round, and sticky bits. The guard, round, and sticky bits are used as inputs by the round unit, depending on the selected RM the Mantisa's LSB would be rounded or not.

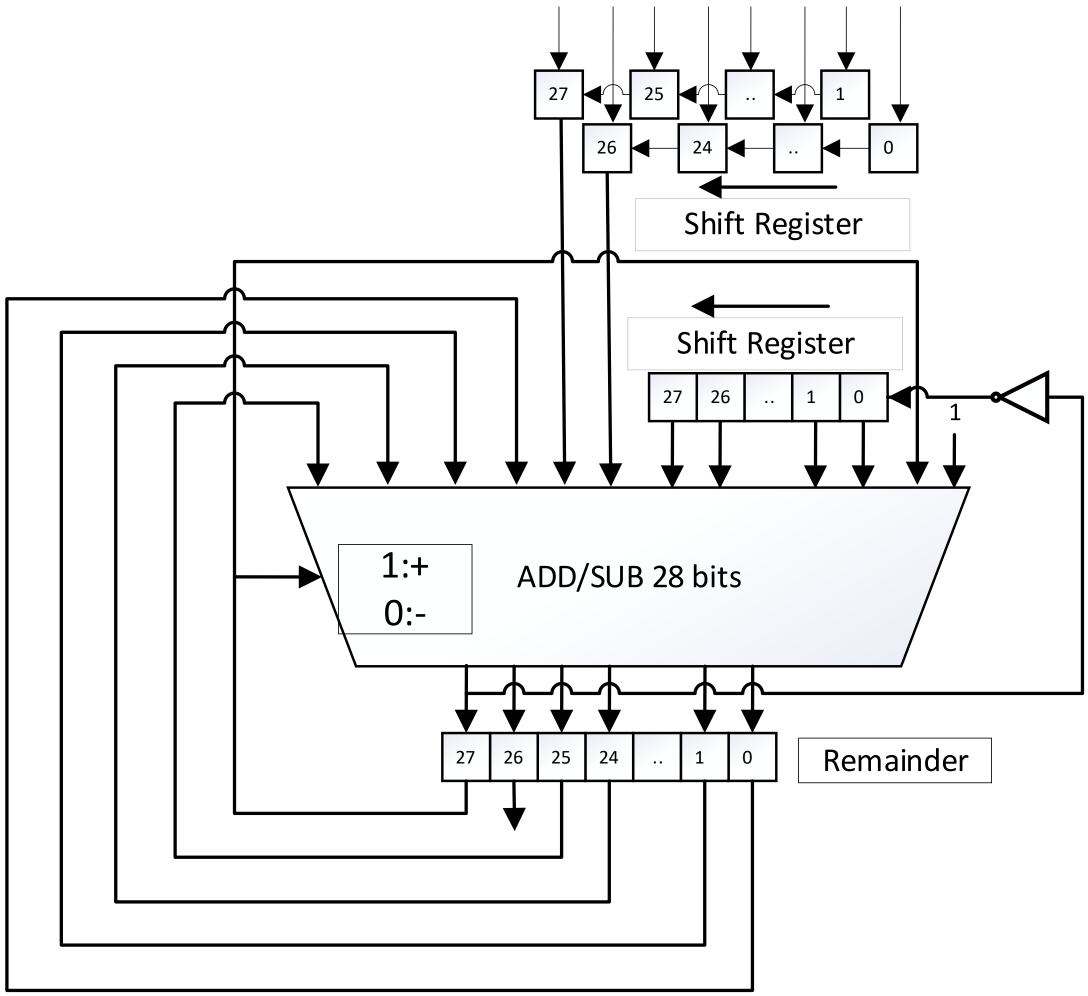

Figure 22. 28 bits Square Root hardware implementation

- Floating Point square root exceptions

The FP square root module also flags exceptions and returns in 1 clock cycle a 32 bits FP default value depending on the kind of input exception value. The exceptions related to the FP square root are presented in Table 11.

Table 11. FP square root supported exceptions

Exception type (input value at FP square root)	32 bits return value
+ infinite	+ infinite
- infinite	- infinite
+ zero	+ zero
- zero	- zero
+1	+ zero
NaN	NaN
Quiet NaN	NaN

6.1.1 Classify op

The classify operation takes a single operand (Operand A) and calculates a 10 bits output encoding of different qualifications. The encoding is placed in the LSB of the resulting output to satisfy the requirement of having a 32 bits size for any FP operation. The 22 MSBs are tied to zero. Figure 23 and Table 12 describe the inputs and outputs of the block. The 5 bits exception output is tied to zero as shown in Figure 24 since the classify operation doesn't have any special scenarios.

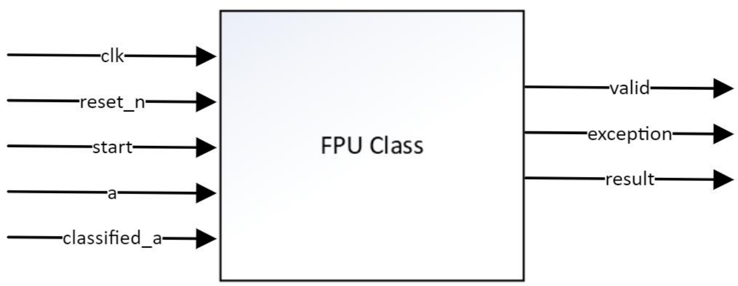

Figure 23. FPU Class black box.

Table 12. FPU Class input and output description.

Signal or group name	size	Type	Description
clk	1	input	Main clock.
reset_n	1	input	Active low main reset.
start	1	input	Enable signal. Active one clock cycle per operation.
a	32	input	Operand A (to classify).
classified_a	9	input	Operand A qualifications from FPU classifier.
valid	1	output	FP operation result is valid.
exception	5	output	FP exceptions after the operation.
result	32	output	FP operation result.

The classify block internally contains one register for the FPU classifier struct input and one register for the start input signal which is translated to the valid output signal in the following clock cycle. The 10 bits output result encoding shown in Table 13 reflects the relationship between the classifier input and the classified operation output.

Table 13. Classify OP output encoding based on the FPU Classifier input

Output bit	FPU Classifier input
0	is_quiet
1	is_signalling
2	is_positive & is_inf
3	is_positive & is_normal
4	is_positive & is_subnormal
5	is_positive & is_zero
6	is_negative & is_zero
7	is_negative & is_subnormal
8	is_negative & is_normal
9	is_negative & is_inf

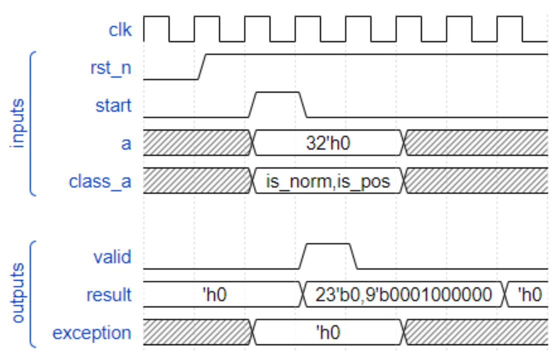

Figure 24. Classify operation timing diagram

7.1.1 Min/Max/EQ/LT/LE ops

The following operations are handled in the FPU compare block: FMIN, FMAX, FEQ, FLT, FLE. The first two operations compare Operand A vs Operand B and return the minimum or maximum 32 bits value, either A or B. While the last 3 operations return a bit, either zero or one, based on the Operands comparison conditional.

Table 14 and Figure 25 describe the inputs and outputs of the block. Like the other operation blocks, it leverages the classified struct from the FPU classifier for each operand. OP 2 bits input is wired to the funct3 instruction field. When the signal is_cmp is set to one, the OP variable is equal to zero for the FLE operation, one for the FLT operation, and two for the FEQ operation. If is_cmp is not set, then a value of zero represents the FMIN operation and a value of one represents the FMAX operation. The is_cmp signal comes from the decoder based on the funct7 instruction field. FMIN and FMAX share the same funct7 b'0010100 value, while FEQ, FLT, and FLE share a value of 'b1010000. Figure 26 presents the timing diagram of the FEQ operation.

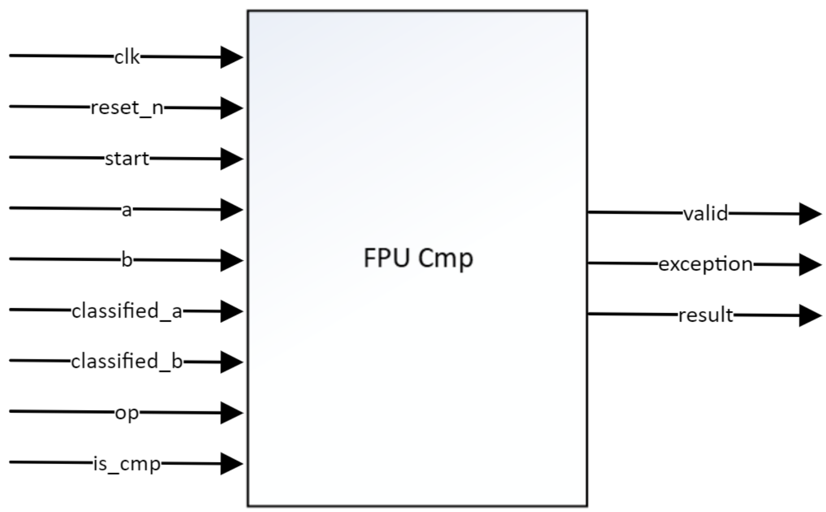

Figure 25. FPU Compare black box

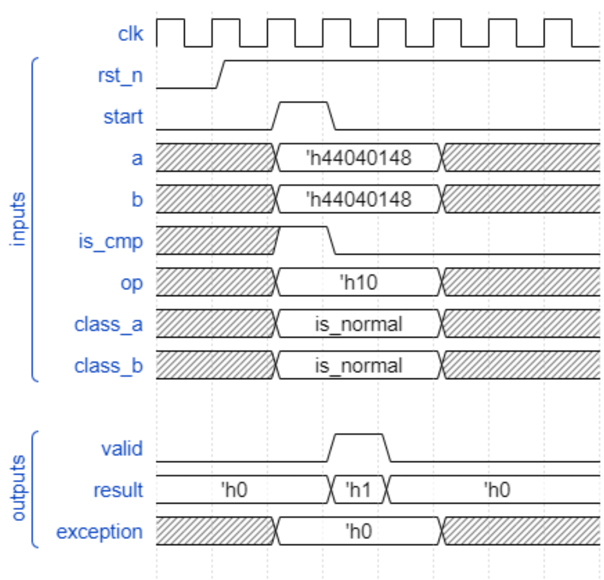

Figure 26. FEQ operation timing diagram

Table 14. FPU Compare input and output description.

Signal or group name	size	Type	Description
clk	1	input	Main clock.
reset_n	1	input	Active low main reset.
start	1	input	Enable signal. Active one clock cycle per operation.
a	32	input	Operand A (to compare).
classified_a	9	input	Operand A qualifications from FPU classifier.
b	32	input	Operand B (to compare).
classified_b	9	input	Operand B qualifications from FPU classifier.
op	2	input	From instruction. funct3[1:0].
is_cmp	1	input	From decoder. Set to 1 if the operation is FLE/FLT/FEQ.
valid	1	output	FP operation result is valid.
exception	5	output	FP exceptions after operation.
result	32	output	FP operation result.

The FPU compare block consists of 7 registers and some combinational logic. All the inputs are registered except for clk and reset_n. With the registered values, comparisons are made to resolve 3 main variables: is_lower_than, is_equal, or is_greater_than.

8.1.1 *Round-robin arbiter and output registers*

Inside the FPU an arbiter mechanism is needed to handle all different FP operation requests coming from the RISC-V processor, a round-robin arbiter was chosen to handle all four different types of FP supported operations (division, square root, comparator, and classification).

The round-robin arbitration module was built based on an FSM that has one initial state after reset and four additional states, each one related to the FP operation request the arbiter may receive from the processor (division, square root, comparator, and classification). Each clock cycle the request inputs are sampled to choose to which state the FSM should transition next and grant outputs are asserted depending on the current state.

Figure 27 shows the black box diagram and Figure 28 describes the possible outputs based on the request_in input.

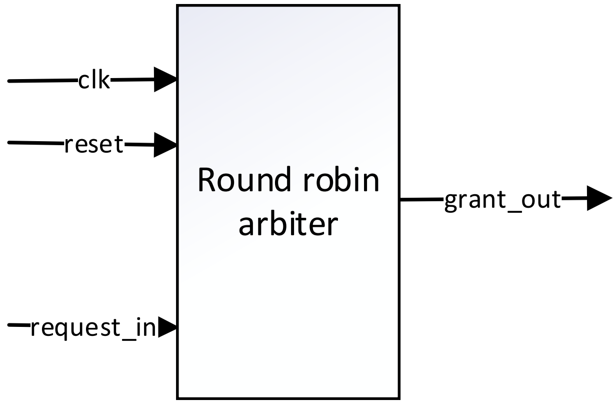

Figure 27. Round-robin arbiter inputs and outputs.

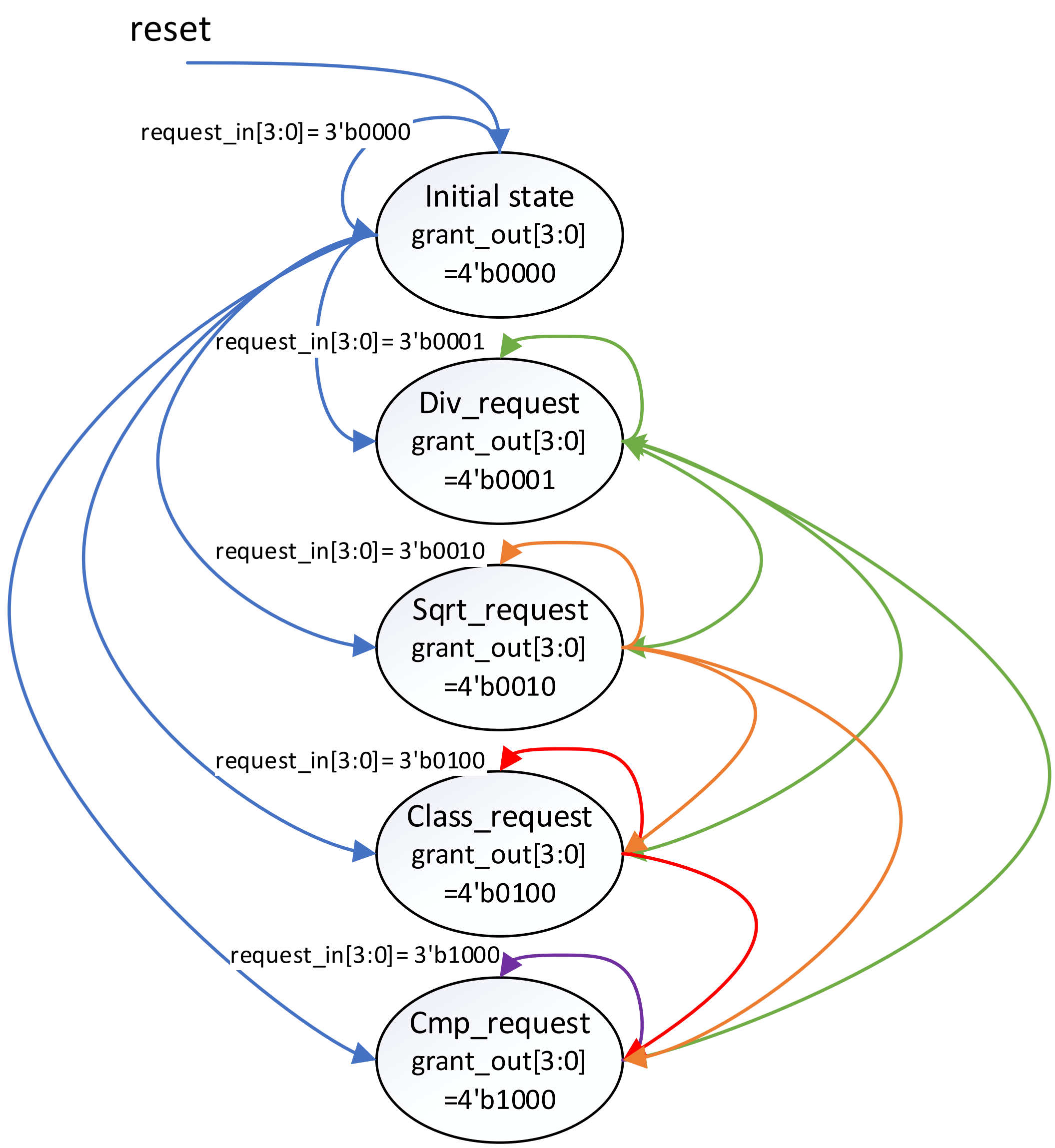

Figure 28. Round-robin FSM diagram.

3.5. RISC-V pipeline integration

Figure 29 shows the top level of the RISC-V pipeline implementation. It consists of RISC-V implementation connected to the program memory, also called ROM, and the data memory, commonly referenced as RAM.

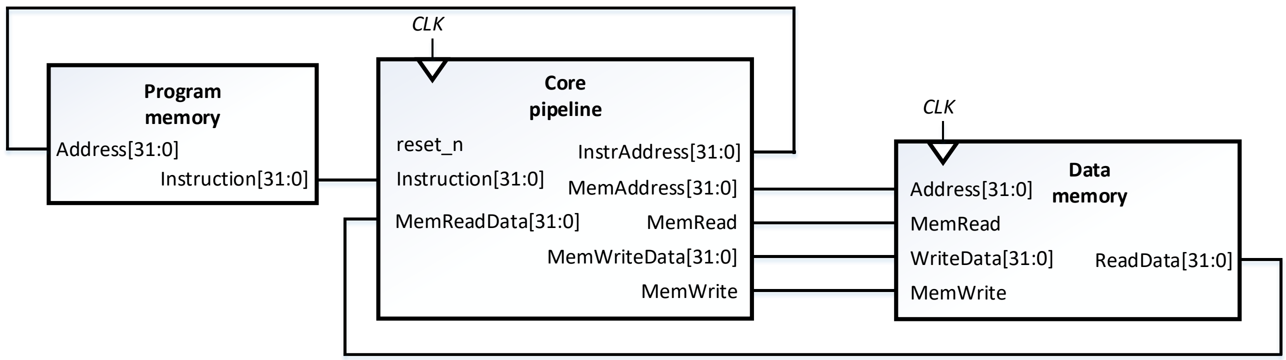

Figure 29. TOP level risc_v_pipeline.

- Control unit

The control unit was expanded to support FP instructions. Additional output bits were introduced, including signals such as fpuon, fcsrwe, flw, and fsw. Instruction opcodes defined in the RV32F standard extension—'h27 (FSW), 'h07 (FLW), 'h53 (FMIN, FMAX, FEQ, FLT, FLE, FCLASS, FDIV, FSQRT), and 'h73 (FRRM, FSRM)—were incorporated into the case statement. The Memory to Register (MemtoReg) output could now be set to 'h3 to indicate that the FCSR read value should be retrieved.

- FPRF

The FPRF is derived from the GPRF. The primary distinction is that all registers in the FPRF are writable, whereas in the GPRF, register zero is fixed at 0. The decoder and combinational logic were reused for efficiency. Additionally, the FPRF was integrated into the decode stage, with outputs connected to the decode-execute pipeline register. Figure 30 presents the block's inputs and outputs.

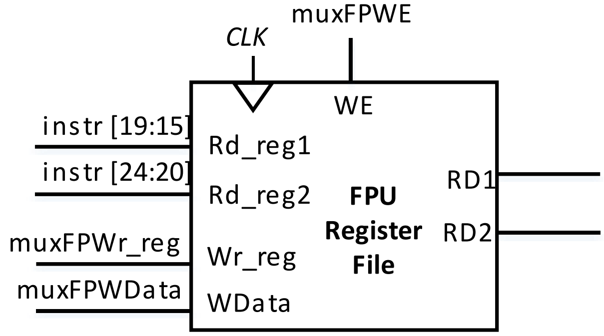

Figure 30. FPRF inputs and outputs.

- FCSR

The FCSR was added as a separate register outside the FPRF to maintain a size of 32 entries and optimize the decoder. The FCSR cannot be used as an operand by any FP operation but can be accessed by a limited set of instructions. FRRM reads the current value, while FSRM performs a swap: it reads the current value, writes it into a GPRF register, and updates the FCSR with a new value from a GPRF register. Since both the FRRM and FSRM instructions share the same opcode 'h73, the funct3 field is utilized to distinguish between them. Funct3 is set to 2 for reads and 1 for writes. Therefore, bit 12 of the instruction can indicate a swap operation when combined with the fcsr control signal. The write enable for the FCSR is set in two scenarios: when there is a swap operation or when there is an FPU result ready to update the flags. Figure 31 illustrates the inputs and outputs the FCSR.

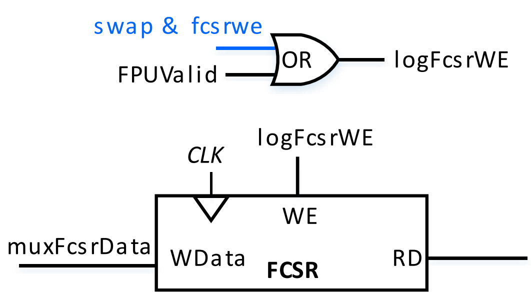

Figure 31. FCSR inputs and outputs.

- Muxes

A total of seven muxes were added to the write-back stage to manage results from both the ALU and FPU. Depending on the instruction, both the ALU and FPU can access either the GPRF or the FPRF. When an FPU result is available, the muxes select FPU result data over ALU output. Each of the GPRF and FPRF requires three muxes: one for write enable (WE), one for the destination register (Wr_reg), and one for the write data (Wdata). Additionally, another mux was included to handle the

FCSR Wdata. When an FPU result is available, the write data updates the flags exclusively. However, during a swap operation, the value is sourced from the ALU result, which transfers the GPRF register value into the FCSR. Figure 32 illustrates the seven new muxes added to the design. The gray mux is part of the original pipeline design and is shown in the figure to clarify the input labeled 'mux4.'

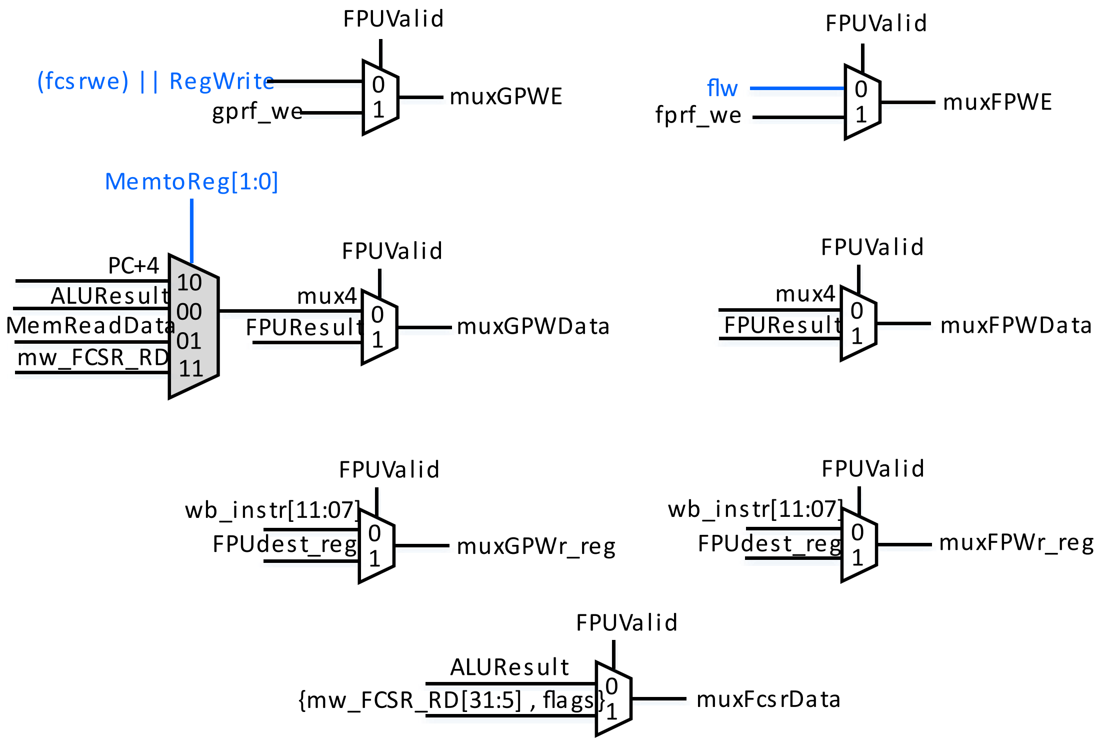

Figure 32. Muxes added to pipeline.

- Memory access

Support has been incorporated for the FSW and FLW operations, which facilitate the transfer of floating-point (FP) values between the FPRF and memory. These operations function similarly to integer load and store operations, utilizing the ALU path to compute the effective address by adding the base address from register rs1 and the offset from the immediate field. To facilitate these operations, a mux has been introduced in the execute stage. This mux directs the FPRF read data (RD2) to the memory. The control unit's output bit fsw serves as the selector for the mux, allowing it to choose between FPRF RD2 and GPRF RD2. The selected output from the mux is then connected to the memory write data (MWD).

- FPU instance

The FPU instance was integrated into the execute stage with connections established to each pipeline register. The instr_rnd_mode input is wired to the instruction bits, while the csr_rnd_mode input is linked to the FCSR register. Special considerations were made for operations such as division or

SQRT, which may require more than 26 cycles to complete. To ensure accuracy, the write-back stage was meticulously designed to utilize the correct values using the muxes described before.

During execution, the instruction's destination register field is stored within the FPU as FPUdest_reg until the operation concludes. This approach allows the pipeline to correctly identify and utilize the intended destination register.

3.6. Testbench

The functionality of the FPU was verified by writing two testbenches: one at the FPU level and one at the top level. The FPU level testbench directly applies inputs and compares the expected outputs by looking into the FPU output ports, while the top level testbench integrates the FPU with a RISC-V processor, a RAM, and a ROM. Two arrays of inputs, A and B, represented the operands for the FPU operations. For the square root operation, only the A array was used. One array of expected outputs and one array of expected flags for each operation were used to check the correctness and the status of the FPU results. The testbenches had 45 test cases in a loop, covering different combinations of normal, subnormal, zero, infinity, and NaN values. For each test case, the FPU output and flags were compared with the expected values, and any mismatches were reported. At the top level, the FPU instructions and the input and output addresses were encoded in a binary code and dumped into the ROM. The code was executed on the RISC-V processor, which decodes the instructions and forwards them to the FPU. For each iteration, the FPU result, and flags were stored in the RAM. At the end of the test, the RAM values were read and compared with the expected values, using the same logic as the FPU level testbench.

Table 15 describes the tests used to validate the normal and subnormal scenarios, while Table 16 covers the tests for zero, infinity, NaN and sNaN cases. Both tables represent the 45 testcases mentioned earlier. The same input vectors were used for each operation.

Table 15. Normal and subnormal testcases.

case	A	B
1	1.0000e-28	9.9999e+27
2	9.9999e+27	1.0000e-28
3	2.6666	1
4	2.625	1.31
5	25	5
6	91.34375	0.1445
7	24	3.5
8	3.5	24
9	2.046875	16.0003
10	1.125	1.125

11	2.046875	24
12	704	196608
13	7.9999	2
14	2.0755	3.7552
15	520.02	520.02

Table 16. Zero, infinity and NaN testcases.

case	A	B
1	1.5	NaN
2	NaN	1.5
3	NaN	NaN
4	1.5	inf
5	-1.5	inf
6	inf	1.5
7	inf	-1.5
8	inf	inf
9	1.5	zero
10	-1.5	zero
11	zero	1.5
12	zero	-1.5
13	zero	zero
14	inf	zero
15	zero	inf
16	NaN	zero
17	zero	NaN
18	NaN	inf
19	inf	NaN
20	-inf	NaN
21	sNaN	1.5
22	sNaN	-1.5
23	sNaN	Inf
24	sNaN	zero
25	sNaN	sNaN
26	1.5	sNaN
27	-1.5	sNaN
28	Inf	sNaN
29	zero	sNaN
30	sNaN	sNaN

4. IMPLEMENTATION RESULTS

This chapter describes the implementation results of the FPU, outlining the specific FPGA and tools used. The main goal of this design was to strike a balance between resource utilization and performance, considering the constraints often present in embedded systems. It examines the decision to favor combinational logic over pipelined designs, highlighting the trade-offs between resource efficiency and processing speed.

4.1. FPGA implementation

The FPU hardware was implemented on a Stratix 10 FPGA development board using Intel Quartus Pro tools for synthesis and timing analysis. Table 17 provides details on the resource utilization for each module. The design achieved a maximum operating frequency of 203.42 MHz, as determined by the Intel Quartus Timing Analyzer tool.

Table 17. FPU project FPGA utilization results.

Entity name	Dedicated Logic Reg	ALMs Needed
fpu_top	650	732.2
fpu_classifier	0	8.7
fpu_classifier	0	8.1
fpu_class	9	4.7
fpu_cmp	78	52.9
fpu_decoder	0	5.7
fpu_div	204	338.5
rr_arb	5	8.7
fpu_sqrt	223	147.1

4.2. Discussion

This FPU module is intended to be used with a 32 bits RISC-V core for embedded applications, having in mind that the embedded area focuses more on resource utilization than speed, the implemented logic/arithmetic algorithms were chosen among the ones with fewer resources. Most of the implementations in this FPU are purely combinational over pipelined designs where speed is increased, but the penalty of adding more logic stages must be paid. One example is presented in [6] where 2 different FxP square root implementations of the same algorithm are discussed, one is iterative with fewer resources spent, and the second pipelined implementation offers a huge increase in speed with one square root result per clock cycle.

5. ANALYSIS AND DISCUSSIONS

Future directions for enhancing the FPU design and functionality are examined in this chapter. The potential for executing multiple FP operations concurrently is explored, which could significantly boost throughput and processing efficiency. Additionally, the integration of ALU and FPU operations within the same cycle is investigated to optimize system performance. Management of data transfers between the GPRF and the FPRF is also addressed, with a focus on strategies to improve data handling and reduce latency.

The chapter further analyzes data hazards that can impact processing speed and accuracy, aiming to develop effective mitigation strategies. Usage of the GNU Compiler Collection (GCC) and the implementation of more efficient algorithms are also discussed. These explorations aim to enhance FPU capabilities, contributing to the development of more robust and high-performing embedded systems.

5.1. Future explorations

- Handle more than one FP operation at a time.

The FP unit is not pipelined and can only process one division or square root operation every 29 cycles. Therefore, 29 NOPs are inserted between any two operations of this type to prevent erroneous results. Also, if two FP instructions of different operations use different RMs, they must be executed sequentially to avoid incorrect results or exceptions.

- Handle ALU results and FPU results in the same cycle.

The ALU and the FPU are two units that perform arithmetic and FP operations respectively and produce results that need to be stored or forwarded to other parts of the processor. However, there is a potential problem if both units generate results at the same time, as there is only one result bus that can carry one result at a time. This means that one of the results would be lost, and the corresponding instruction would not be completed correctly. To avoid this situation, the processor needs to implement some mechanism to coordinate the timing and priority of the results coming from the ALU and the FPU, such as a result queue or a result arbiter.

- Moves between GPRF and FPRF.

The GPRF and the FPRF are separate register files, and there is no direct way to access the FPRF from the integer instructions or vice versa. However, RISC-V provides a set of move (MV) instructions that can transfer data between the GPRF and the FPRF, without changing the bit pattern or the interpretation of the data. Some of those instructions move the data without conversion, but some others convert the FP value to signed or unsigned 32-bit integer. None of these instructions are currently supported but it would be great to have them in the future.

- Data hazards.

Since the new FPRF was added, there could be data hazards in the pipeline. The three main types of data hazards are read-after-write (RAW), write-after-read (WAR), and write-after-write (WAW). These hazards can affect the correctness and consistency of the program execution, as they can cause data races and overwrite errors. Pausing the pipeline until the data dependency is resolved could be a possible solution to explore.

- GCC

The implementation was tested using as a reference the results obtained from the RISC-V Assembler and Runtime Simulator (RARS) application, which is a simulator and assembler for the RISC-V instruction set. However, this method only compares the output of the assembly code and does not verify the correctness of the generated machine code or the compliance with the RISC-V specification. A more accurate way to test the implementation would be to use a cross-compiler, such as the GCC, to generate RISC-V binaries from C programs and run them on the hardware implementation. GCC is a widely used set of compilers, libraries, and tools for various programming languages and target architectures.

- Algorithms

Exploring improved division algorithms presents significant potential for enhancing performance within the FPU. Division operations are inherently complex and time-consuming, particularly in floating-point arithmetic, where precision and efficiency are paramount. The challenges are evident, as seen in algorithms like the digit recurrence algorithm. By researching and implementing more efficient division algorithms such as Newton-Raphson iterations, SRT division, or other specialized techniques, substantial improvements in computational speed can be achieved.

6. CONCLUSIONS

The 32 bits single-precision FPU module designed in this thesis work focused its implementation to comply with the RISC-V ISA “F” extension instructions and, also following the IEEE 754 standard.

The FPU IP module implementation was tested altogether with a 32 bits, 5-stage pipeline RISC-V processor developed in-house during the “microprocessor design” course, the area of application for this module is the embedded space where these days less resource utilization is more critical than speed in some cases. More specifically to this thesis work, the design was implemented in an FPGA where resource utilization is important and ready-to-use modules like this FPU are always looking to spend as less resources as possible so other more critical sections of the core or the complete system can leverage more logic.

From the variety of algorithms out there some of them offer more performance in terms of speed but involve a more complex architecture where a lot of area can be consumed either by pipeline stages or look-up tables with fixed values that are part of division or square root algorithms just to name a few. The ones that were implemented in this work were selected using the criteria of area efficiency, this effect can be observed mostly in the FP division and FP square root.

Beyond just creating an FPU unit, one of the main purposes of this development was to follow the IEEE 754 standard and the RISC-V ISA “F” extension, taking into consideration details like the supported rounding modes and exception cases that in some developments are overseen and the focus is just on implementing the FP algorithm, also implementing RISC-V specific registers for FP.

All the modules in this implementation were individually tested and, together with the previously mentioned RISC-V core using a custom testbench that was previously discussed in this document, all the operations were subject to vectors where exceptions were tested to demonstrate the capability of the FPU to flag them and provide a result with the corresponding exception flags.

7. References

[1] Sasidharan, A., & Nagarajan, P. (2014, February). VHDL implementation of IEEE 754 floating point unit. In *Proceedings of the International Conference on Information Communication and Embedded Systems (ICICES2014)* (pp. 1-5). Chennai. https://doi.org/10.1109/ICICES.2014.7033999

[2] Hemmert, K. S., & Underwood, K. D. (2010, September). Fast, efficient floating-point adders and multipliers for FPGAs. *ACM Transactions on Reconfigurable Technology and Systems, 3*(3), 1-30. https://doi.org/10.1145/1839480.1839481

[3] Beauchamp, M. J., Hauck, S., Underwood, K. D., & Hemmert, K. S. (2008, February). Architectural modifications to enhance the floating-point performance of FPGAs. *IEEE Transactions on Very Large Scale Integration (VLSI) Systems, 16*(2), 177-187. https://doi.org/10.1109/TVLSI.2007.912041

[4] Patil, V., Raveendran, A., Sobha, P. M., David Selvakumar, A., & Vivian, D. (2015, June). Out of order floating point coprocessor for RISC V ISA. In *Proceedings of the 19th International Symposium on VLSI Design and Test* (pp. 1-7). Ahmedabad, India. https://doi.org/10.1109/ISVDAT.2015.7208116

[5] Deschamps, J.-P., Bioul, G. J. A., & Sutter, G. D. (2006). *Synthesis of arithmetic circuits: FPGA, ASIC, and embedded systems*. Hoboken, NJ: John Wiley & Sons.

[6] Parhami, B. (2010). *Computer arithmetic: Algorithms and hardware designs* (2nd ed.). New York: Oxford University Press.

[7] Goldberg, D. (1990). Computer arithmetic. In D. A. Patterson & J. L. Hennessy (Eds.), *Computer architecture: A quantitative approach* (Appendix A). San Mateo, CA: Morgan Kaufmann.

[8] Waterman, A., Lee, Y., Patterson, D. A., & Asanovic, K. (2011). *The RISC-V instruction set manual, volume I: Base user-level ISA* (Tech. Rep. UCB/EECS-2011-62). EECS Department, UC Berkeley.

[9] Oberman, S. F., & Flynn, M. J. (1997, August). Division algorithms and implementations. *IEEE Transactions on Computers, 46*(8), 833-854.

[10] Neelima, N., Kumar, A. S., Jayanth, A., Mahitha, K. K., & Reddy, A. D. K. (2023). Implementation of efficient restoring and long division algorithm. In *Proceedings of the 14th International Conference on Computing Communication and Networking Technologies (ICCCNT)* (pp. 1-6). Delhi, India. https://doi.org/10.1109/ICCCNT56998.2023.10308155

[11] Li, Y., & Chu, W. (1996, October). A new non-restoring square root algorithm and its VLSI implementations. In *Proceedings of the International Conference on Computer Design: VLSI in Computers and Processors* (pp. 1-6). Austin, TX. https://doi.org/10.1109/ICCD.1996.563604

8. APPENDIX

8.1. Appendix 1. Complete pipeline design diagram.