EE502 CORNERSTONE/ CAPSTONE PROJECT REPORT

On

IMPLEMENTATION OF IEEE 754 ARITHMETICS

Under the Guidance of

Prof. K P Vittal

Submitted in partial fulfilment of the requirements for the degree of

BACHELOR OF TECHNOLOGY

In

ELECTRICAL AND ELECTRONICS

By

Suchet Nayak 221EE254

Anirudh Hariharan 221EE206

DEPARTMENT OF ELECTRICAL AND ELECTRONICS

NATIONAL INSTITUTE OF TECHNOLOGY KARNATAKA

SURATHKAL, MANGALORE -575025

NOVEMBER, 2025

D E C L A R A T I O N

We hereby declare that the Report of the Capstone Project Work entitled

IMPLEMENTATION OF IEEE 754 ARITHMETICS

which is being submitted to the National Institute of Technology Karnataka Surathkal, in partial fulfilment of the requirements for the award of the Degree of Bachelor of Technology in the department of Electrical and Electronics Engineering, is a bonafide report of the work carried out by us. The material contained in this Report has not been submitted to any University or Institution for the award of any degree.

221EE254 Suchet Nayak

(Register Number, Name & Signature of the Student)

221EE206 Anirudh Hariharan

(Register Number, Name & Signature of the Student)

Department of Electrical and Electronics Engineering

Place: NITK, SURATHKAL

Date:

C E R T I F I C A T E

This is to certify that the U.G. Capstone Project Work Report entitled

IMPLEMENTATION OF IEEE 754 ARITHMETICS

submitted by Suchet Nayak and Anirudh Hariharan,
(Register Number: 221EE254 and 221EE206) as the record of the work carried out by him/her, is accepted as the U.G. Capstone Project Work Report submission in partial fulfilment of the requirements for the award of degree of Bachelor Technology in the Department of Electrical and Electronics Engineering.

Internal Guide

(Name and Signature with Date and Seal)

(Name and Signature with Date and Seal)

Chairman – DPGC

(Signature with Date and Seal)

Acknowledgement

We would like to express our deepest gratitude to all those who have supported and guided us throughout the successful completion of our Capstone Project. This work would not have been possible without their constant encouragement, expertise, and assistance.

First and foremost, we express our sincere thanks to Dr. K P Vittal, our project guide, for their invaluable supervision, insightful feedback, and continuous motivation throughout the course of this project. Their guidance has been instrumental in shaping our technical understanding and problem-solving approach.

We also extend our heartfelt gratitude to Dr. Debashish Jena, Head of the Department of Electrical and Electronics Engineering, for providing us with the necessary facilities and a conducive environment to carry out our project work effectively.

Also, a heartfelt thanks to Dr. Ramesh Kini, Professor, Department of Electronics and Communication, for their guidance and support during the initial phase of the project.

Finally, we thank our classmates, friends, and family members for their constant moral support, understanding, and encouragement which helped us persevere// throughout the project development.

Journey through Capstone

Our journey began in July 2023 with our first Cornerstone Project, marking the beginning of an exploration that would eventually shape our understanding of digital design and floating-point computation. At that stage, we were driven purely by motivation and curiosity, yet uncertain about how to direct that energy toward creating something meaningful. To gain direction, we approached Dr. K. P. Vittal and Dr. Ramesh Kini, who kindly agreed to guide us and provided us with a challenging and insightful problem statement.

It was under their mentorship that we were first introduced to the fascinating concepts of floating-point arithmetic and IEEE standard number representations. In the beginning, we started to learn about digital circuits, their implementation, and their uses. In the first cornerstone, our motive was to be perfect about digital circuits and their implementation and expanding our knowledge in the single-precision floating-point numbers.

We then continued the project through three other cornerstones, using a hight level programming language to emulate the numerical operations on different number formats, then building our own arithmetic circuits from scratch referring several books and resources online and finally integrating all the designed circuits under one top module.

The knowledge and experience we gained during this project was immense and very valuable. The guidance we received from our guide helped us not only expand our knowledge in academic fields, but also promoted us to be more punctual and professional with our work.

This capstone journey has inspired us to continue pursuing our academic and professional careers with the same enthusiasm, curiosity, and dedication that guided us through each phase of this project. We believe that the lessons learned here will serve as a strong foundation for our future endeavours in research and innovation.

Abstract

This project presents the design and implementation of arithmetic operation modules for single-precision floating-point numbers in Verilog, compliant with the IEEE 754 standard. The primary objective is to develop and verify hardware modules capable of performing addition, subtraction, multiplication, division, and square root operations with high precision and efficiency. The proposed design emphasizes hardware-level accuracy and computational integrity while maintaining optimized resource utilization. Simulation and verification are conducted to validate the correctness and performance of each arithmetic unit. The outcome of this work is a functional hardware implementation capable of performing arithmetic operations on IEEE 754 single-precision floating-point numbers.

Keywords — IEEE 754, floating-point arithmetic, Verilog HDL, single precision

Contents

List of figures

I. Introduction

Floating-point arithmetic is a critical element in modern digital systems, enabling the representation and manipulation of real numbers with a high degree of precision. The IEEE 754 standard defines the format, representation, and operational rules for floating-point arithmetic, ensuring uniformity across different computing systems. Applications such as graphics rendering, scientific computation, machine learning, and signal processing all rely on the accuracy and efficiency of IEEE 754-compliant floating-point units (FPUs).

The objective of this project is to design, implement, and verify an IEEE 754 single-precision Floating-Point Unit (FPU) capable of performing basic arithmetic operations—addition, subtraction, multiplication, division, and square root—using Verilog HDL. The project spans multiple semesters, evolving from conceptual understanding and algorithm development in C to hardware-level implementation and integration into a complete system. This gradual learning approach ensures that both theoretical comprehension and practical application are strengthened through each stage.

The following section explains how the project was carried out over the duration of four semesters and the contribution of each cornerstone to the final capstone project.

Semester 1: Conceptual Foundation

The first semester was dedicated to strengthening the theoretical understanding of digital electronics and the IEEE 754 floating-point standard. Students analysed how floating-point arithmetic differs from fixed-point and integer arithmetic, particularly focusing on normalization, rounding, and exponent biasing. The binary representation of real numbers was studied to comprehend how arithmetic operations are performed on mantissas and exponents separately.

The exploration of IEEE 754 arithmetic laid the foundation for understanding hardware computation. This phase emphasized the importance of precision, overflow and underflow handling, and rounding errors—all of which later became crucial during implementation. To demonstrate this understanding, a module was written in the C programming language that could convert a real number input into its nearest floating-point representation.

Semester 2: Algorithm Development in C

In the second semester, the theoretical concepts were translated into practical algorithmic simulations using the C programming language. The operations were programmed for three different number types—integer, fixed-point, and floating-

point—and the limitations and pitfalls of each type were assessed and understood using diverse test cases. Arithmetic operations such as addition, subtraction, multiplication, and division were implemented to simulate how these operations occur in actual hardware. The objective was to validate the correctness of arithmetic algorithms and study their step-by-step execution.

Each operation was implemented by breaking down the process into distinct phases, such as sign determination, exponent alignment, mantissa operation, normalization, and rounding. For example, in addition, the smaller exponent was adjusted by right-shifting its mantissa until both exponents matched. The mantissas were then added or subtracted based on the sign bits, followed by normalization. This software-level verification acted as a reference for hardware verification in later stages.

Semester 3: Hardware Design and Verification in Verilog

The third semester focused on designing the arithmetic units in Verilog HDL. Each operation was implemented as an independent module, ensuring modularity and testability. The addition/subtraction unit used a 24-bit Kogge–Stone adder, providing logarithmic depth and efficient carry propagation.

The multiplication unit multiplied the normalized mantissas of two operands and added their exponents after adjusting for bias. Normalization ensured that the product remained within the representable range. The division unit was implemented using the Newton–Raphson iterative method, which estimates the reciprocal of the divisor and multiplies it with the dividend to obtain the quotient.

Each module was rigorously tested using simulation waveforms and dedicated testbenches. Verification confirmed accuracy across a wide range of test cases, validating both the algorithmic and hardware correctness of the design.

Semester 4: FPU Integration and Square Root Implementation

The final semester concentrated on integrating all arithmetic units into a single Floating-Point Unit (FPU) module. An additional square root operation was designed using the Newton–Raphson method, which approximates the square root through iterative refinement. The square root module handled special IEEE 754 cases such as negative input (producing *NaN*) and infinity (returning infinity).

The FPU architecture consisted of a controller and a dispatcher, which decoded the operation code and directed operands to the appropriate arithmetic module. Each arithmetic unit was instantiated four times in parallel to enhance throughput.

Synchronization challenges arose due to combinational delays in the data path, which were analysed during simulation.

Semester 5: Final touch

The errors faced in the top module in the previous semester were tried to solve using a new type of architecture similar to that of Tomasulo's algorithm. The top module was made to function only with ADD, SUB and MUL operations, due to errors faced while integrating with others.

The division module too was modified to perform better using approximate initial guesses, so that there is faster convergence. This also made the performance of the square root module better as the division module too is used in it.

The results were obtained and are discussed upon in the results and discussion section.

II. Literature Review and Background

A. Floating-Point Representation in Digital Systems

Floating-point arithmetic plays a pivotal role in digital computing systems that demand a wide dynamic range and high numerical precision. Unlike fixed-point or integer arithmetic, which is restricted to a limited range of representable values, floating-point representation allows real numbers to be expressed in a normalized scientific notation, enabling operations on both extremely large and extremely small magnitudes. The IEEE 754 standard defines how these numbers are represented and manipulated to ensure consistency across hardware and software platforms.

In a binary floating-point system, a number $N$is expressed as:

$$N = (-1)^S \times 1.M \times 2^{(E-Bias)}$$

where Srepresents the sign bit, Mdenotes the mantissa or significand, and $E$is the exponent field. The bias is used to represent both positive and negative exponents in unsigned binary form. For single-precision representation (32 bits), the bit fields are divided into one sign bit, eight exponent bits, and twenty-three fraction bits. The hidden leading bit (implicit '1') adds an additional level of precision, resulting in an effective 24-bit mantissa.

Floating-point arithmetic underpins a vast array of applications including real-time signal processing, numerical analysis, control systems, and computer graphics. Its ability to approximate real-world quantities efficiently has made it the cornerstone of scientific computation and embedded system design.

B. IEEE 754 Standard Overview

The IEEE 754 standard, first established in 1985 and later revised in 2008 and 2019, provides a comprehensive framework for floating-point representation and arithmetic operations. It defines several formats—single precision (32-bit), double precision (64-bit), and extended formats—to accommodate diverse computational requirements. The standard ensures deterministic and portable numerical results across different hardware platforms, addressing the inconsistencies that plagued earlier floating-point implementations.

Key features of IEEE 754 include:

- **Normalized representation:** Each number is represented in a normalized form to maximize precision.

- **Special values:** The standard specifies encodings for special values such as positive and negative infinity, zero, denormalized numbers, and *NaN* (Not a Number).
- **Rounding modes:** It defines multiple rounding strategies such as round-to-nearest-even, round-toward-zero, round-toward-positive, and round-toward-negative.
- **Exception handling:** It provides mechanisms for signalling exceptional conditions like overflow, underflow, and invalid operations.

These design choices make the IEEE 754 format robust and predictable. Floating-point arithmetic units adhering to this standard, such as those implemented in the Intel x87 architecture and ARM Cortex-M series processors, follow these rules to maintain interoperability and consistency across diverse computing environments.

C. Floating-Point vs. Fixed-Point Arithmetic

While both fixed-point and floating-point representations are used in digital arithmetic, they differ significantly in structure, complexity, and application. Fixed-point arithmetic allocates a fixed number of bits to represent the integer and fractional parts, making it simpler and faster but less flexible in dynamic range. Floating-point arithmetic, on the other hand, dynamically adjusts the exponent to represent a wide range of magnitudes, thus reducing quantization errors and improving numerical stability.

In embedded systems, fixed-point units are often preferred for their hardware simplicity and speed. However, in applications involving iterative algorithms or wide-ranging data—such as deep learning, 3D graphics, and scientific simulations—floating-point computation is indispensable. Research conducted by Goldberg (1991) and extended in IEEE 754-2008 has shown that floating-point computation significantly improves accuracy and portability in numerical algorithms.

Designing floating-point units in hardware, however, introduces challenges related to latency, area, and power consumption. Trade-offs between precision and performance must be carefully balanced. For example, while single-precision arithmetic provides adequate accuracy for most applications, double precision is often used in scientific computation for higher precision requirements.

D. Existing Floating-Point Unit Architectures

Modern floating-point units (FPUs) are the result of decades of research and architectural innovation. The **Intel x87 FPU**, introduced in the 1980s, was one of the earliest dedicated floating-point coprocessors. It supported 80-bit extended precision and provided hardware support for transcendental functions. Similarly, **IBM's PowerPC architecture** integrated high-performance FPUs capable of parallel pipelined operations, optimizing performance for multimedia and scientific workloads.

The **ARM Cortex-M and Cortex-A series** processors incorporate IEEE 754-compliant FPUs that perform single- and double-precision arithmetic with low latency. These architectures often use hardware techniques such as fused multiply-add (FMA) operations, pipelining, and speculative execution to enhance throughput. In the open-source domain, the **RISC-V architecture** defines a modular F extension that supports IEEE 754 floating-point arithmetic, allowing designers to implement FPUs suited to their performance and area constraints.

Research studies, such as those by Oberman and Flynn (1997) and Kuang et al. (2013), have emphasized the importance of efficient FPU design through techniques like carry-save addition, redundant number systems, and high-radix division. These innovations aim to improve the latency and precision of arithmetic operations while minimizing power consumption—critical for modern embedded and mobile systems.

E. Algorithmic Foundations of Arithmetic Operations

The implementation of floating-point arithmetic operations requires careful algorithmic design to ensure accuracy, efficiency, and compliance with IEEE 754 rules. Each operation involves multiple stages such as operand unpacking, alignment, arithmetic computation, normalization, rounding, and exception handling. This section discusses the core algorithms underlying each operation used in this project.

1) Floating-Point Addition and Subtraction

Addition and subtraction are the most fundamental operations in floating-point arithmetic. The process begins with unpacking the operands and comparing their exponents. The smaller exponent is aligned by right-shifting its mantissa until both exponents are equal. Once aligned, the mantissas are added or subtracted depending on the sign bits of the operands.

The sum or difference is then normalized to ensure the result conforms to the standard form, i.e., the leading bit of the mantissa is '1'. Normalization may

require left or right shifts of the mantissa and corresponding adjustments to the exponent. Finally, rounding is applied according to the selected rounding mode.

High-performance FPUs often employ **carry-lookahead** or **Kogge–Stone adders** for fast mantissa addition, as they provide logarithmic carry propagation delay. The Kogge–Stone adder used in this project is a well-established parallel prefix structure that balances speed and area efficiency. Research by Brent and Kung (1982) and Han and Carlson (1987) established that such tree-based adders are ideal for wide-bit arithmetic typical in floating-point systems.

2) Floating-Point Multiplication

Floating-point multiplication involves adding the exponents and multiplying the mantissas. The sign of the result is determined by the XOR of the operand signs. After the multiplication, the resulting mantissa is normalized to maintain the IEEE 754 standard representation. The product is then rounded and checked for overflow or underflow conditions.

Efficient multiplication in hardware can be achieved using **Booth's algorithm** or **Wallace tree multipliers**, both of which minimize partial product accumulation time. The Wallace tree structure, introduced by Wallace (1964), reduces the addition depth by grouping and summing partial products in parallel, significantly reducing propagation delay.

In floating-point multiplication, the normalization step ensures that the product's leading bit is '1', maintaining representational consistency. In modern FPUs, pipelined multiplier units improve throughput by overlapping multiple operations. This project's Verilog implementation focuses on optimizing mantissa multiplication and bias-adjusted exponent addition while adhering to the single-precision format.

3) Floating-Point Division

Division is computationally more complex than multiplication and requires iterative approximation techniques for efficiency. The **Newton–Raphson method** is a widely used algorithm for computing division in floating-point units. It operates by first estimating the reciprocal of the divisor and then refining the approximation through successive iterations.

For a division A/B, the reciprocal $1/B$is computed using the iterative formula:

$$X_{n+1} = X_n(2 - BX_n)$$

This approach doubles the number of correct bits with each iteration, enabling high accuracy with relatively few steps. Once the reciprocal is obtained, it is multiplied by the dividend to produce the final quotient. Newton–Raphson division is preferred over restoring or non-restoring division methods due to its convergence speed and compatibility with parallel multipliers.

Alternative division algorithms, such as **SRT division** (proposed by Sweeney, Robertson, and Tocher), use digit recurrence and partial remainder computations. However, these methods tend to have longer latency in FPGA-based implementations, making iterative approaches more efficient for this project's architecture.

4) Floating-Point Square Root

The square root operation is another non-linear function that benefits from iterative approximation techniques. The **Newton–Raphson method** is again employed, using the recurrence relation:

$$X_{n+1} = \frac{1}{2}(X_n + \frac{A}{X_n})$$

This algorithm converges quadratically, making it suitable for hardware implementation. In IEEE 754-compliant systems, special cases such as negative operands, zeros, and infinities must be handled explicitly. For instance, the square root of a negative number results in *NaN*, while the square root of infinity yields infinity.

Historically, implementations like the Intel x87 and ARM NEON units have used iterative or lookup-based hybrid square root algorithms to balance precision and speed. This project's implementation focuses on hardware realizability using the Newton–Raphson iteration, as it allows reuse of the multiplier and adder units, thus conserving hardware resources.

F. Implementation Techniques in Verilog HDL

Verilog HDL serves as a powerful hardware description language for modelling, simulating, and synthesizing digital circuits. It enables both structural and behavioural modelling, making it suitable for implementing complex arithmetic units. The modularity of Verilog facilitates the development of independent arithmetic blocks—such as adder, multiplier, and divider—which can later be integrated into a complete FPU system.

In hardware design, **pipelining** and **parallelism** are two key optimization strategies. Pipelining divides a computation into multiple stages separated by registers, allowing different operations to be processed simultaneously in consecutive clock cycles. Parallelism, on the other hand, replicates arithmetic units to perform multiple operations concurrently. Both techniques enhance throughput at the cost of increased area utilization.

To verify correctness, **testbenches** are written to apply stimulus vectors and monitor outputs. Simulation tools such as ModelSim or Vivado Simulator enable waveform inspection to confirm adherence to IEEE 754 arithmetic behaviour. The design can then be synthesized onto FPGA platforms for hardware validation.

The modular design approach used in this project ensures that each arithmetic block—addition, subtraction, multiplication, division, and square root—can be tested independently before integration. This method aligns with best practices in hardware verification and ensures reliable system-level performance.

G. Design Verification and Simulation Approaches

Verification is a critical aspect of floating-point hardware design. Due to the high precision and corner cases involved in IEEE 754 arithmetic, exhaustive testing is required to ensure correctness. Verification typically includes functional simulation, timing analysis, and synthesis validation.

Functional verification involves comparing the hardware outputs against reference software models. For instance, C or Python-based floating-point libraries can serve as golden models for expected results. Simulation waveforms provide visibility into internal signal behaviour, helping designers detect misalignment, overflow, or rounding issues.

Post-synthesis verification ensures that the design's behaviour remains consistent after logic synthesis and optimization. This stage also includes **timing simulation**, which accounts for propagation delays and setup/hold times. FPGA implementation offers an additional level of validation by testing the design in a real hardware environment.

Modern verification methodologies such as **System Verilog Assertions (SVA)** and **Universal Verification Methodology (UVM)** have been widely adopted in industry to automate verification. However, for academic and capstone projects, traditional testbench-driven simulation remains an effective and illustrative approach.

H. Summary of Literature Findings

The literature on floating-point arithmetic and FPU design reveals a strong foundation of theoretical and practical research that continues to evolve with advancements in hardware technology. From early microprocessor coprocessors like the Intel x87 to modern ARM and RISC-V FPUs, the focus has consistently been on achieving optimal trade-offs between precision, latency, and hardware complexity.

IEEE 754 remains the gold standard for floating-point computation, ensuring consistency across hardware platforms. Algorithms such as Kogge–Stone addition, Booth and Wallace multiplication, and Newton–Raphson division and square root have become foundational techniques in FPU design. These algorithms balance computational efficiency with ease of hardware implementation, making them suitable for both ASIC and FPGA applications.

Verilog HDL provides a versatile platform for modelling and testing these algorithms at the hardware level. The simulation-driven design flow adopted in this project aligns closely with industrial design methodologies, ensuring both correctness and performance optimization.

In conclusion, the research and architectural principles reviewed in this section directly inform the design choices of the current project. By integrating algorithmic precision, IEEE 754 compliance, and modular Verilog-based implementation, the project aims to contribute to the ongoing exploration of efficient, scalable, and accurate floating-point computation in digital hardware systems.

III. Methodology

The implementation of operations during each semester will be explained in each section under this heading.

A. Semester 1

The conversion of real numbers into IEEE 754 single-precision format was implemented purely from first principles, without using any built-in typecasting or floating-point operations. The goal was to mathematically simulate how real numbers are represented at the hardware level using binary encoding. The program begins by taking a decimal number as input in string form, enabling exact parsing of digits without intermediate floating-point rounding errors.

The first step involves **sign extraction**: if the input begins with a minus sign ('–'), the sign bit is set to 1; otherwise, it remains 0. The number is then decomposed into its **integer part** and **fractional part**, where the integer portion is accumulated through successive base-10 multiplications, and the fractional portion is represented as a rational fraction $\frac{N_f}{10^k}$, with N_f being the fractional digits and k the number of digits after the decimal point. The overall number can thus be expressed as:

$$x = I + \frac{N_f}{10^k}$$

Next, the decimal representation is converted into binary using repeated division and multiplication. For the integer portion, binary digits are obtained through successive division by 2, while for the fractional part, successive multiplications by 2 extract binary digits before the decimal point. The combined binary sequence is then normalized into the IEEE 754 form:

$$x = (-1)^S \times (1.f) \times 2^E$$

where S is the sign bit, f is the normalized fractional mantissa, and E is the unbiased exponent.

The **normalization process** determines the position of the most significant '1' in the binary expansion to compute the unbiased exponent E. The biased exponent field is then calculated as:

$$E_{bias} = E + 127$$

as IEEE single precision uses a bias of 127. The 23-bit mantissa field is filled with the fractional bits following the leading 1, while three additional bits — **guard**, **round**, and **sticky** — are used to implement **round-to-nearest-even** rounding. This rounding ensures numerical consistency with hardware floating-point units.

For **subnormal numbers** (where $E_{bias} \leq 0$), the exponent field is set to 0, and the mantissa is right-shifted to represent numbers smaller than 2^{-126}. Special cases such as infinity ($E_{bias} = 255, f = 0$) and NaN ($E_{bias} = 255, f \neq 0$) are explicitly handled.

Finally, the 32-bit IEEE 754 representation is constructed as:

$$\text{Binary} = [S][E_{bias}][f_{23}]$$

where the sign bit occupies bit 31, the exponent bits occupy bits 30–23, and the mantissa occupies bits 22–0. The result is displayed both as a 32-bit binary string and a hexadecimal word. This method provides a detailed understanding of binary normalization, exponent biasing, and mantissa rounding, accurately reflecting the mathematical process underlying IEEE 754 floating-point encoding.

B. Semester 2

This section describes the software and hardware methodologies used for implementing arithmetic operations in **Integer**, **Fixed-Point**, and **Floating-Point** number formats. Each subsection explains the stepwise computational logic, conversion principles, and arithmetic mechanisms used for addition, subtraction, multiplication, and division.

A. Integer Format

1) Addition

The first step involves the conversion of integer inputs into binary numbers. This is achieved by defining a new data type consisting of an integer array, where each element represents a bit.

To perform addition, the operation begins from the **Least Significant Bit (LSB)** of the binary representations of both integers and proceeds towards the **Most Significant Bit (MSB)**. Any carry generated during addition is initialized to zero.

Each bit of both operands is processed simultaneously. The corresponding bits and the carry from the previous step are added. If the sum of the current bits and

the carry is greater than or equal to 2, the result bit is set to the remainder of the sum divided by 2, and the carry is updated to 1. Otherwise, the result bit equals the sum, and the carry is updated to 0.

This process continues iteratively until all bits are processed. The resulting binary sequence represents the sum of the two integer inputs.

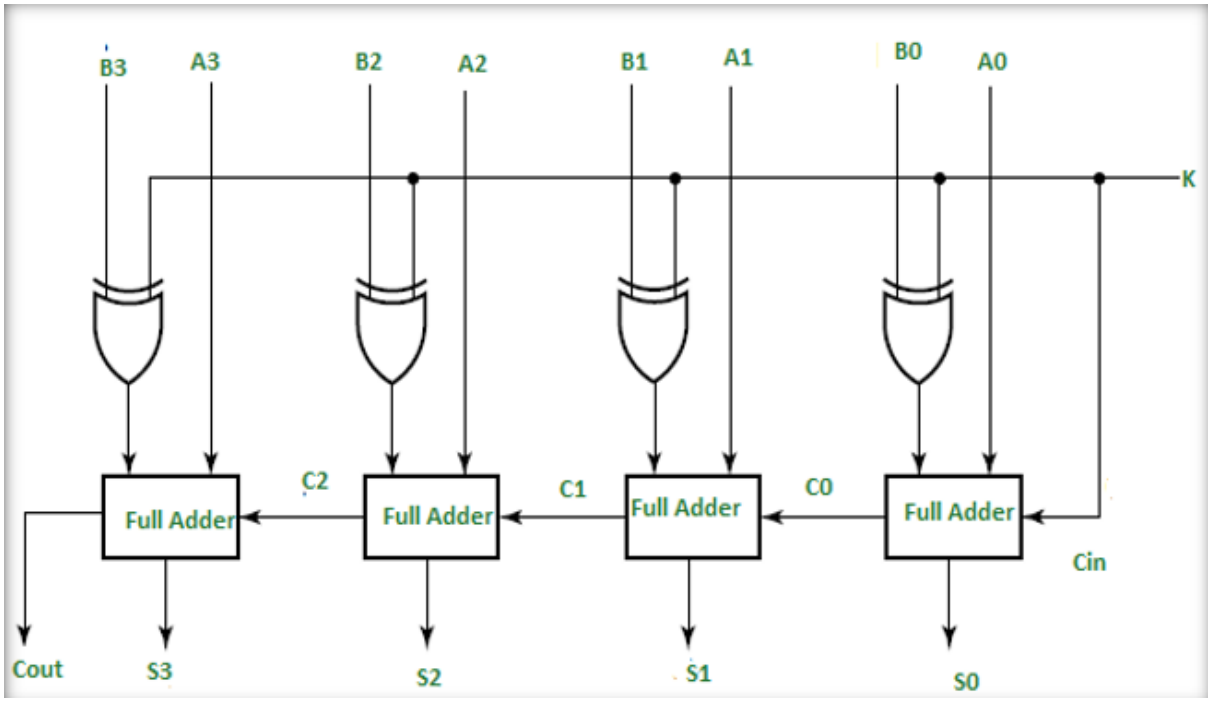


Fig. III.B.1. This figure shows the Adder/Subtractor circuit controlled by a control signal ‘k’

2) Subtraction

i) Software Implementation

Subtraction begins with converting both integer inputs into their binary representations using the “intToBinary” function, which stores the result in a “BinaryNumber” structure.

To perform subtraction, the **two’s complement** of the subtrahend is calculated. This is done by first obtaining the **one’s complement** (bit inversion) and then adding 1 using the “twosComplement” function.

Once the two’s complement of the subtrahend is computed, it is added to the minuend using the “addBinary” function. This addition effectively performs binary subtraction through bitwise arithmetic.

3) Multiplication

i) Software Implementation

Binary multiplication is implemented using the **Shift-and-Add algorithm**. This algorithm iterates through each bit of the multiplier, shifting it left by one position for each iteration, and adding the shifted multiplicand to the result whenever the corresponding bit of the multiplier is 1.

The “shiftLeft” function shifts the multiplier left by one bit in every iteration. During this process, if the current bit of the multiplier (from MSB to LSB) is 1, the multiplicand (appropriately shifted) is added to the result using the “addBinary” function.

The accumulated sum after all iterations represents the final product.

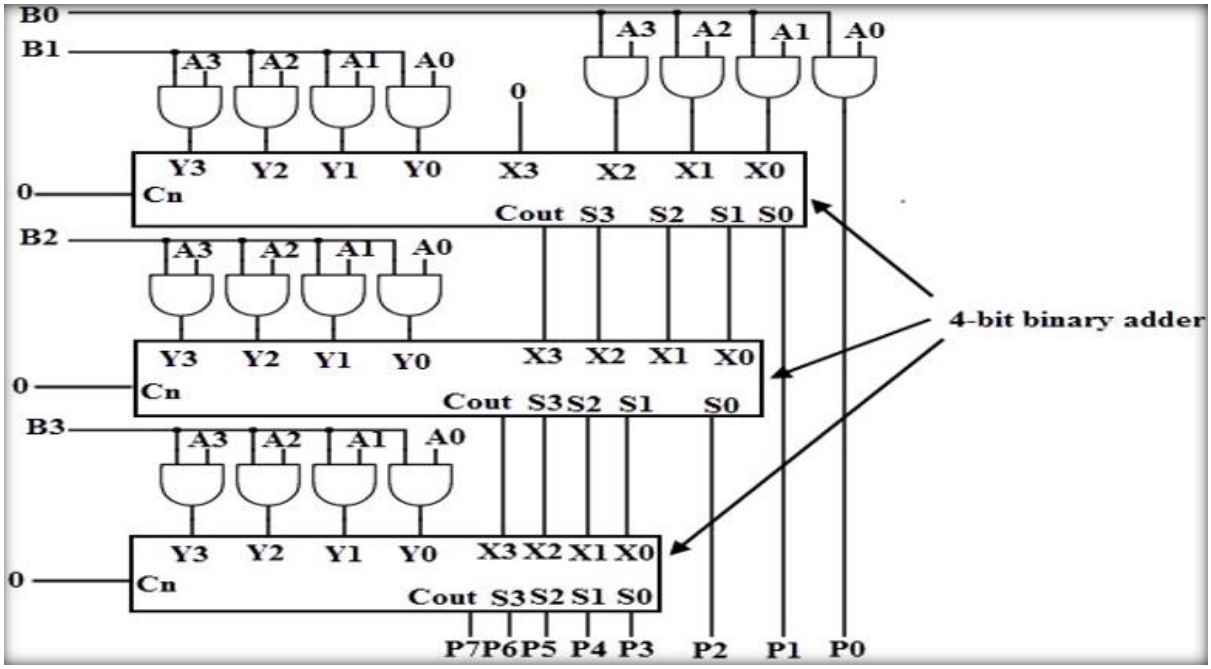


Fig. III.B.2. This figure depicts the combinational multiplier circuit

4) Division

The division process iterates **32 times**, corresponding to the 32-bit binary representation of the dividend.

Starting from the MSB, each bit of the dividend is processed sequentially. In each iteration, the current bit is appended to a temporary variable representing the **partial remainder**. This temporary variable is compared with the divisor.

- If the temporary variable is **greater than or equal** to the divisor, the divisor is subtracted from it, and a '1' is added to the quotient.
- Otherwise, a '0' is appended to the quotient.

After all iterations, the accumulated bits form the quotient. The remainder is obtained by multiplying the quotient by the divisor and subtracting the result from the dividend.

B. Fixed-Point Format

1) Addition and Subtraction

Software Implementation

The addition and subtraction of fixed-point numbers follow the steps below:

Step 1: The bitset library in C++ is used to convert decimal numbers to fixed-point representation. The number is multiplied by $2^{16}$and rounded to the nearest integer.

Step 2: The number is converted to **binary in two's complement form** using the **Q16.16** format.

Step 3: Addition and subtraction are performed using full adder/subtractor logic, identical to binary integer operations.

Step 4: The result is then converted back to integer and divided by $2^{16}$to retrieve the actual fixed-point value.

Hardware Implementation

A 4-bit subtractor logic circuit is implemented to perform the subtraction operation in hardware.

2) Multiplication

In fixed-point arithmetic, the number of integer and fractional bits in the product equals the sum of those in the multiplier and multiplicand, represented as:

$$Q_I(\text{product}) = Q_I(\text{multiplicand}) + Q_I(\text{multiplier})$$
$$Q_F(\text{product}) = Q_F(\text{multiplicand}) + Q_F(\text{multiplier})$$

Resolution of the Fractional Part

For the Q16 format, the resolution (ε) is given by:

$$\varepsilon = \frac{1}{2^{16}} \approx 0.00001$$

Software Implementation

The multiplication algorithm proceeds as follows:

Step 1: The function accepts two floating-point numbers a and b as input.
Step 2: It initializes a variable result to store the product.
Step 3: A loop runs until b becomes close to zero (checked using an EPSILON value).
Step 4: Inside the loop, if the least significant bit of b is 1 ((int)b & 1), a is added to the result.
Step 5: a is left-shifted by 1 (equivalent to multiplying by 2), and b is right-shifted by 1 (dividing by 2).

After completing all iterations, the result of binary multiplication is obtained.

Example:
Let $\alpha = 1.667$and $\beta = -0.75$

$$\alpha \times \beta = 106(Q2.6) \times (-96)(Q1.7) = -10176(Q3.13) = -1.2421875$$

The fixed-point approximation error is:

$$-1.25025 - (-1.2421875) = -0.0080625$$

Hardware Implementation

A fixed-point multiplier circuit was designed to perform the above computation in hardware.

3) Division

Software Implementation

The division algorithm is implemented in Python using bitwise operations and proceeds as follows:

Step 1: The function accepts parameters a (dividend) and b (divisor).
Step 2: Division by zero is handled by raising a "ValueError" if b equals zero.
Step 3: The quotient and remainder variables are initialized to 0.
Step 4: Both a and b are scaled by 1000 to preserve decimal precision.
Step 5: Each bit of the dividend is iterated from MSB to LSB. For each bit, the remainder is left-shifted and updated with the corresponding bit of a.

If the remainder ≥ divisor, the divisor is subtracted, and the corresponding bit in the quotient is set to 1. Finally, the quotient represents the division result.

C. Floating-Point Format

1) Addition and Subtraction

i) Software Implementation

The following approach is adopted:

Step 1: Input two numbers a and b in IEEE 754 binary format along with the operation(+'or'-).
Step 2: Convert the binary strings to unsigned integers by iterating over each bit.
Step 3: Call the fadd function with a, b, and the operation symbol.
Step 4: If subtraction is chosen, invert the sign bit of b using b ^= (1 << 31).
Step 5: Extract sign, exponent, and mantissa of both numbers using bitwise operations.
Step 6: Perform addition or subtraction of mantissas considering the exponents and signs.
Step 7: Normalize or denormalize the result as required. Handle overflow (Infinity)_and_underflow_(Zero)_conditions.
Step 8: Combine the sign, exponent, and mantissa fields to generate the IEEE 754_result.
Step 9: Print the result using the print_ieee754 function.

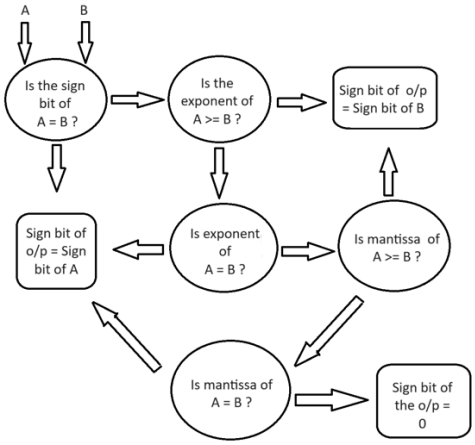


Fig. III.C.1. This flowchart depicts the logic of the designed adder circuit

ii) Hardware Implementation

A standard **Floating-Point Adder/Subtractor Block** is implemented, consisting of exponent comparator, mantissa alignment, normalization, and rounding blocks.

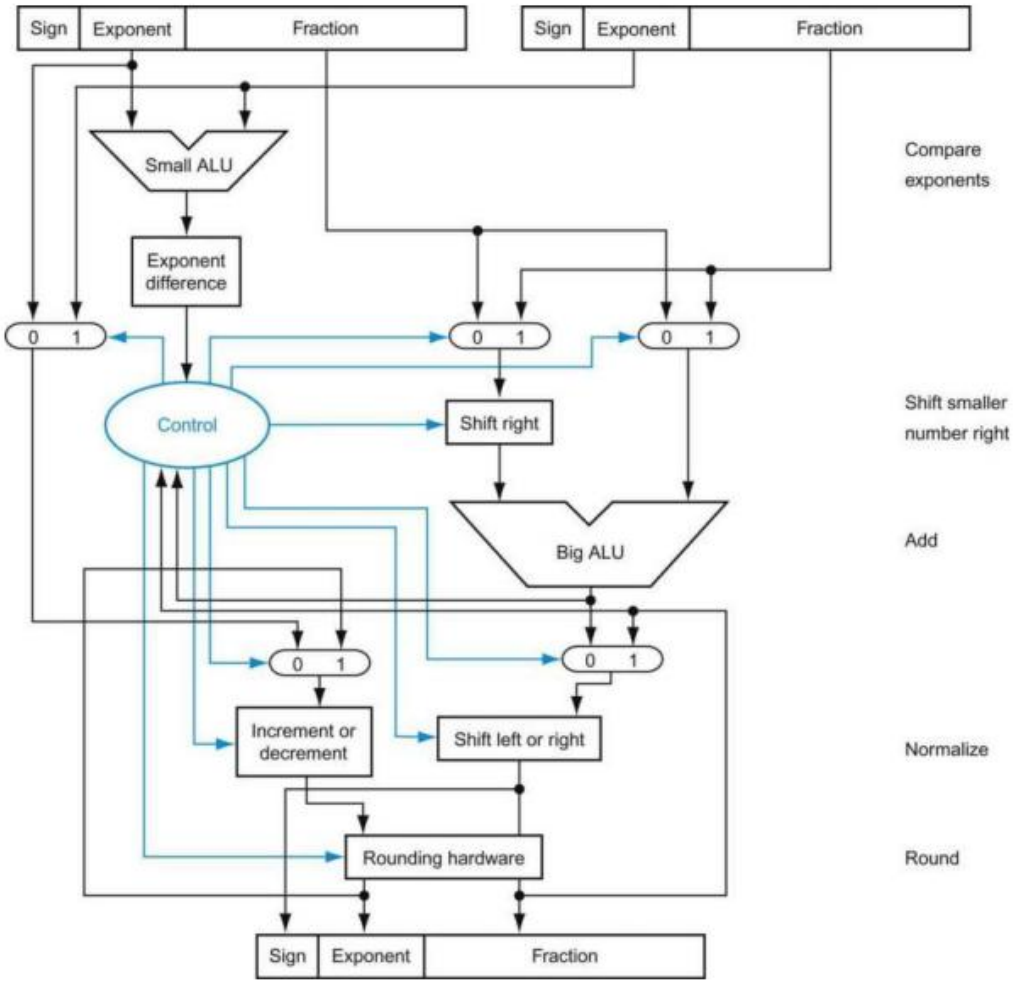


Fig. III.C.2. This figure depicts the hardware implementation of figure III.C.1.

2) Multiplication

i) Software Implementation

Step 1: Read two IEEE 754 32-bit numbers from the user.
Step 2: Validate input and convert binary strings to unsigned integers.
Step 3: Call multiply_ieee754 function with the two inputs.
Step 4: Extract sign, exponent, and mantissa fields, handling special cases (zero, infinity, NaN).
Step 5: Multiply mantissas (accounting for the implicit leading 1), add exponents (adjusting for bias), normalize the result, and determine the result sign using XOR of input signs.
Step 6: Combine the fields to construct the final IEEE 754 number and output it.

ii) Hardware Implementation

A standard **Floating-Point Multiplier Block** is implemented, comprising exponent adder, mantissa multiplier, normalization unit, and rounding logic.

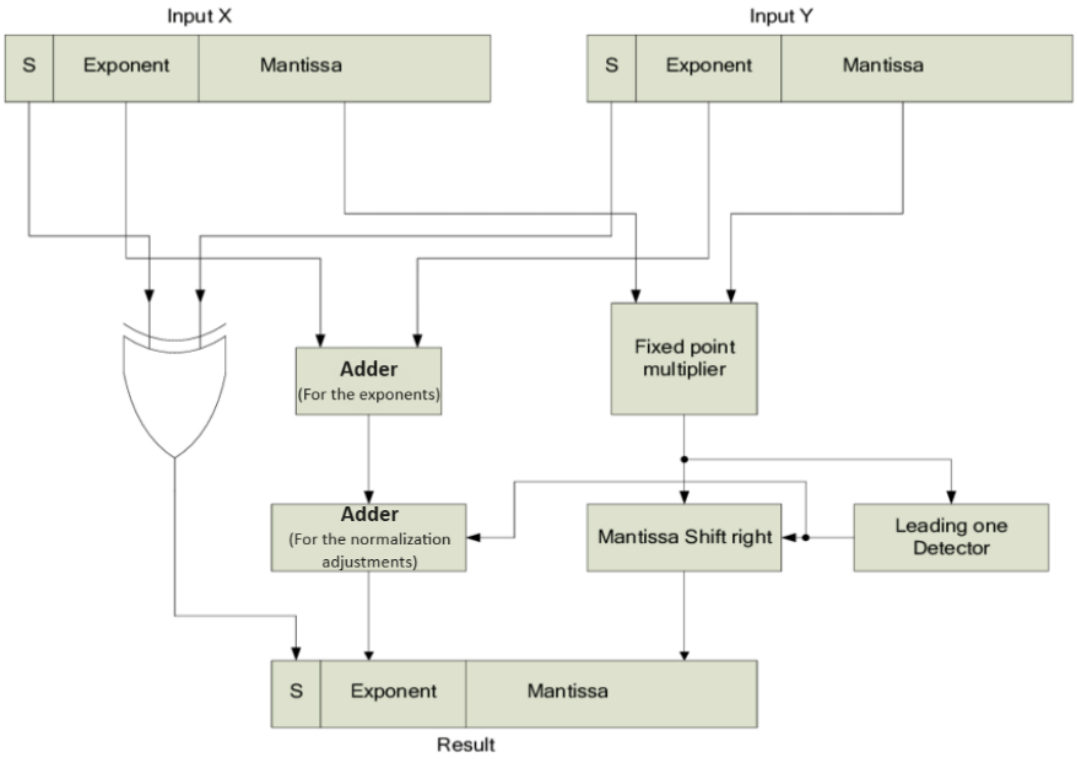


Fig. III.C.3. This figure depicts the hardware implementation of the multiplier circuit

3) Division

i) Software Implementation

Step1: Accept two real number inputs (a and b).
Step 2: Convert both numbers to IEEE 754 binary format using float_to_ieee754.
Step 3: Extract sign, exponent, and mantissa fields using predefined masks (SIGN_MASK, EXPO_MASK, MANT_MASK).

Division Algorithm:

- Normalize inputs if required.
- Generate quotient bits iteratively.
- Perform rounding to the nearest even.
- Check for overflow and underflow conditions.

Slow Path Handling:

- **NaN Handling:** Converts NaN to quiet NaN (QNaN) and returns it.
- **Zero/Infinity Handling:**
    - If both are zero or infinity, output is indefinite.
    - If divisor is zero, result = infinity.
    - If dividend is zero, result = zero.
- **Normalization:** For subnormal inputs, mantissas are shifted until normalized.

Step 5: Combine the sign bit with the computed result and display the final IEEE 754 binary output using print_ieee754.

C. Semester 3

Implementation

A. Addition and Subtraction

The algorithm for addition and subtraction operations was developed from first principles. The process begins by determining the numerically larger number, which is then used as the **augend** and **minuend** for addition and subtraction operations, respectively. The exponents are aligned by right-shifting the mantissa of the numerically smaller number until both exponents become equal.

The addition or subtraction of the extracted and calibrated mantissas is performed using a dedicated logic module that determines the operation type based on the signs of both operands and the operation selected.

A **24-bit Kogge–Stone Adder (KSA)** was designed to perform bit-level addition. The Kogge–Stone Adder, belonging to the family of **parallel prefix adders**, provides high-speed addition by minimizing critical path delays through its logarithmic carry computation structure. Subtraction is performed by taking the **two's complement** of the subtrahend and adding it to the minuend, followed by normalization of the result back to its binary form.

The truth table defining the operation logic is presented below:

Sign of A	Sign of B	Operation (Add: 0, Sub: 1)	Operation to be Executed
0	0	0	0
0	1	0	1
1	0	0	1
1	1	0	0
0	0	1	1
0	1	1	0
1	0	1	0
1	1	1	1

The multiplexer (mux) highlighted in blue determines the mantissa output. When the selection line is high, the addition output is selected; otherwise, the subtraction output is chosen.
Solving the truth table yields the control logic expression:

$$\text{Output} = (\text{SignA} \oplus \text{SignB} \oplus \text{Operation})$$

The corresponding logic blocks are highlighted in orange.

To determine the **sign bit** of the result, comparators are used to evaluate both the exponents and mantissas. Mantissa comparison is performed only when the exponents are equal, with selection controlled by the mux highlighted in purple.

Considering the sign of the numerically larger number and the operation being performed, the output sign bit is inverted if the larger number is subtracted from the smaller one.

The **24-bit Kogge–Stone Adder** operates through three primary stages:

1. **Pre-Processing:**
   For each bit position, the **Propagate (Pi)** and **Generate (Gi)** signals are computed as:

$$P_i = A_i \oplus B_i, G_i = A_i \cdot B_i$$

2. **Carry Computation (Parallel Prefix Tree):**
   Carry signals are computed in logarithmic time using a prefix tree of black and grey cells:
   - **Black Cell:**

$$P_{out} = P_i \cdot P_j, G_{out} = G_j + (P_j \cdot G_i)$$

   - **Gray Cell:**

$$G_{out} = G_j + (P_j \cdot G_i)$$

3. **Post-Processing:**
   Using the computed carry signals C_i, the sum bits are generated as:

$$S_i = P_i \oplus C_{i-1}$$

The final carry-out is derived from the last stage of the prefix tree.

A schematic representation of the tree structure illustrates the hierarchical carry computation process of the Kogge–Stone adder.

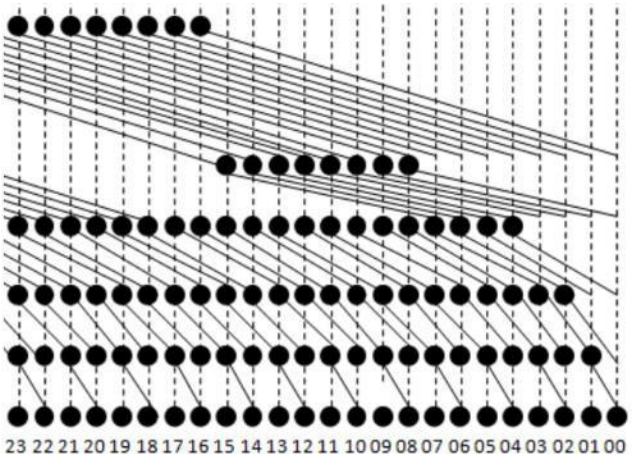


Fig. III.C.1. schematic representation of the Kogge–Stone adder

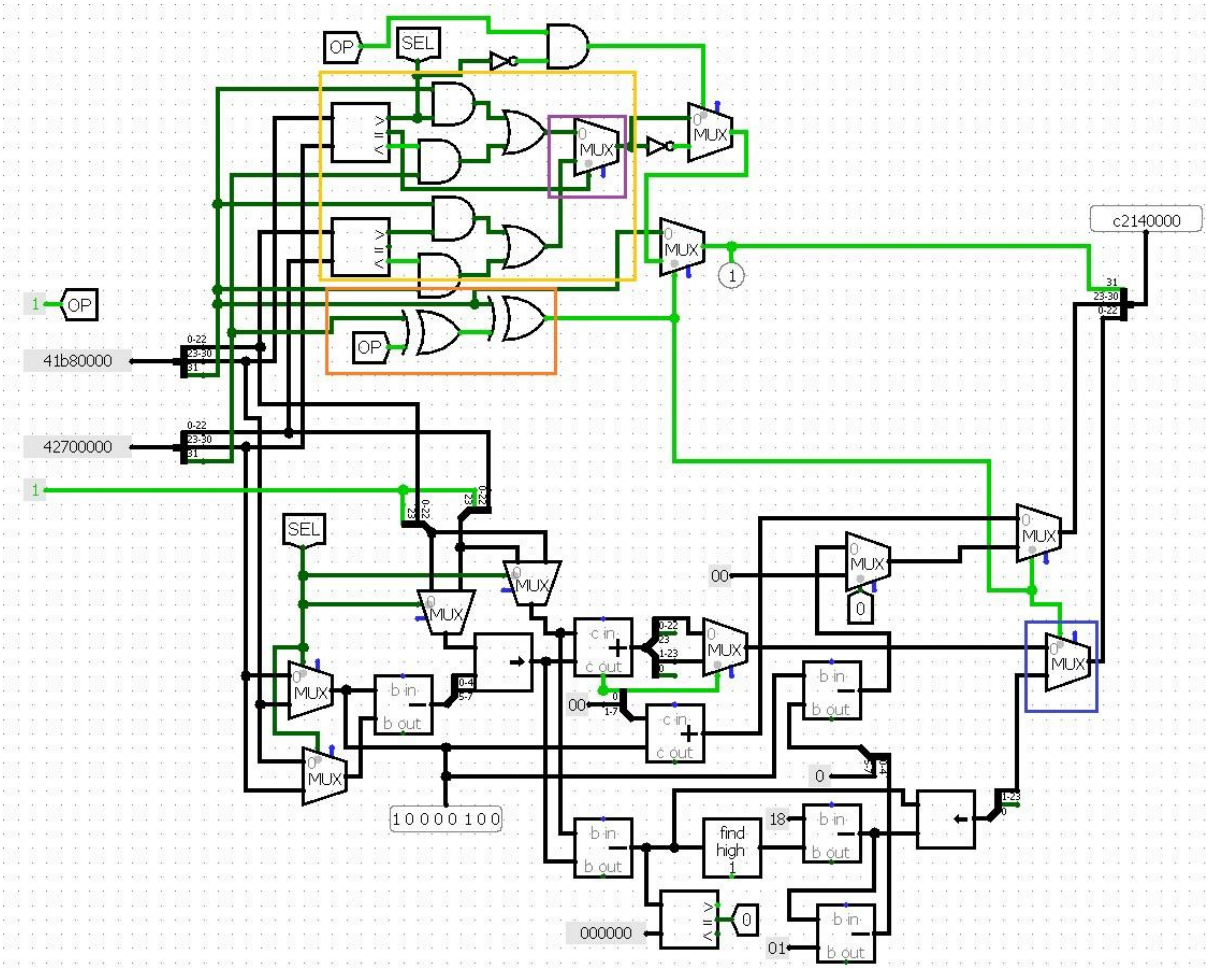


Fig. III.C.2. This figure depicts the final implementation of the adder subtractor circuit

B. Multiplication

The multiplication of IEEE 754 single-precision floating-point numbers involves multiplying their significands (mantissas) and adding their exponents, while handling normalization and rounding. The implementation steps are as follows:

1. **Component Extraction:**
   Each input number is decomposed into its sign bit, exponent, and mantissa.

2. **Sign Computation:**
   The sign of the product is computed as:

$$\text{Sign}_{\text{result}} = \text{Sign}_A \oplus \text{Sign}_B$$

3. **Exponent Addition:**
   The exponents are added, and the bias (127) is subtracted to re-bias the result:

$$\text{Exponent}_{\text{result}} = (\text{Exponent}_A + \text{Exponent}_B) - 127$$

4. **Mantissa Multiplication:**
   The mantissas, including the implicit leading 1, are multiplied:

$$\text{Mantissa}_{\text{result}} = (1.M_A) \times (1.M_B)$$

The product may result in a 48-bit intermediate value, possibly extending beyond one integer bit.

5. **Normalization:**
   If the product exceeds the range [1.0, 2.0), normalization is performed by right-shifting the mantissa and incrementing the exponent.

A scaled-down version of the multiplier, implementing multiplication on two 16-bit floating-point numbers, is illustrated in the design diagram. The same logic scales up for single-precision (32-bit) implementation.

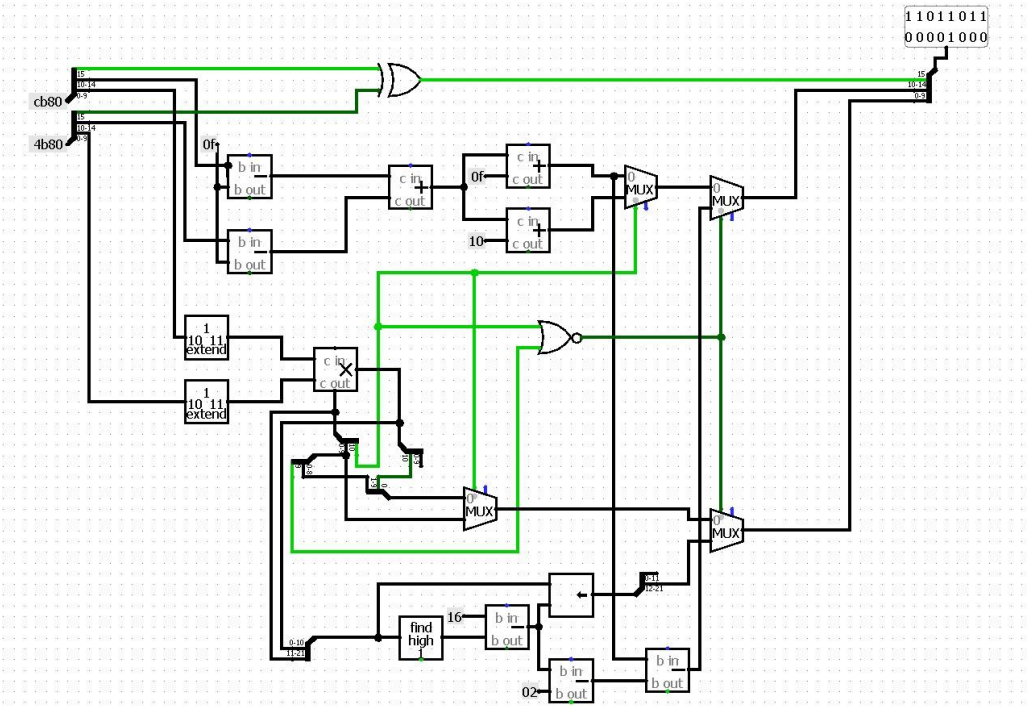


Fig. III.C.3. This figure depicts the final implementation of the multiplier circuit

C. Division

Division of IEEE 754 single-precision floating-point numbers is implemented in **three main phases**:

1. **Phase 1 – Sign Bit Calculation:**
   The sign bits of both operands are XORed to determine the sign of the result.

2. **Phase2–Exponent_Subtraction:**
   Exponent subtraction is implemented using **two's complement addition**:

$$E_{\text{result}} = (E_A - \text{bias}) - (E_B - \text{bias}) + \text{bias}$$

The exponents are stored in biased form, with a bias value of 127 for single precision.

3. **Phase3–Mantissa_Division:**
   The **Newton–Raphson Division Algorithm** is used to divide the mantissas.

Newton–Raphson Division Algorithm Steps

- **Initial Guess (X_0):**
  An approximate reciprocal of the divisor mantissa $M_B$is estimated.
- **Refinement Iteration:**
  The reciprocal is refined iteratively using:

$$X_{n+1} = X_n \times (2 - M_B \times X_n)$$

Typically, two to three iterations achieve sufficient precision for single precision.

- **Final Multiplication:**
  The dividend mantissa is multiplied by the refined reciprocal to obtain the quotient mantissa.

In hardware, this process is implemented as follows:

- Inputs $i_1$and $i_2$are stored in registers.
- XOR operation on sign bits gives the sign result.
- Exponent subtraction is executed through biased exponent registers.
- The mantissa division is carried out iteratively using multiplier and subtractor blocks, as per the Newton–Raphson method.

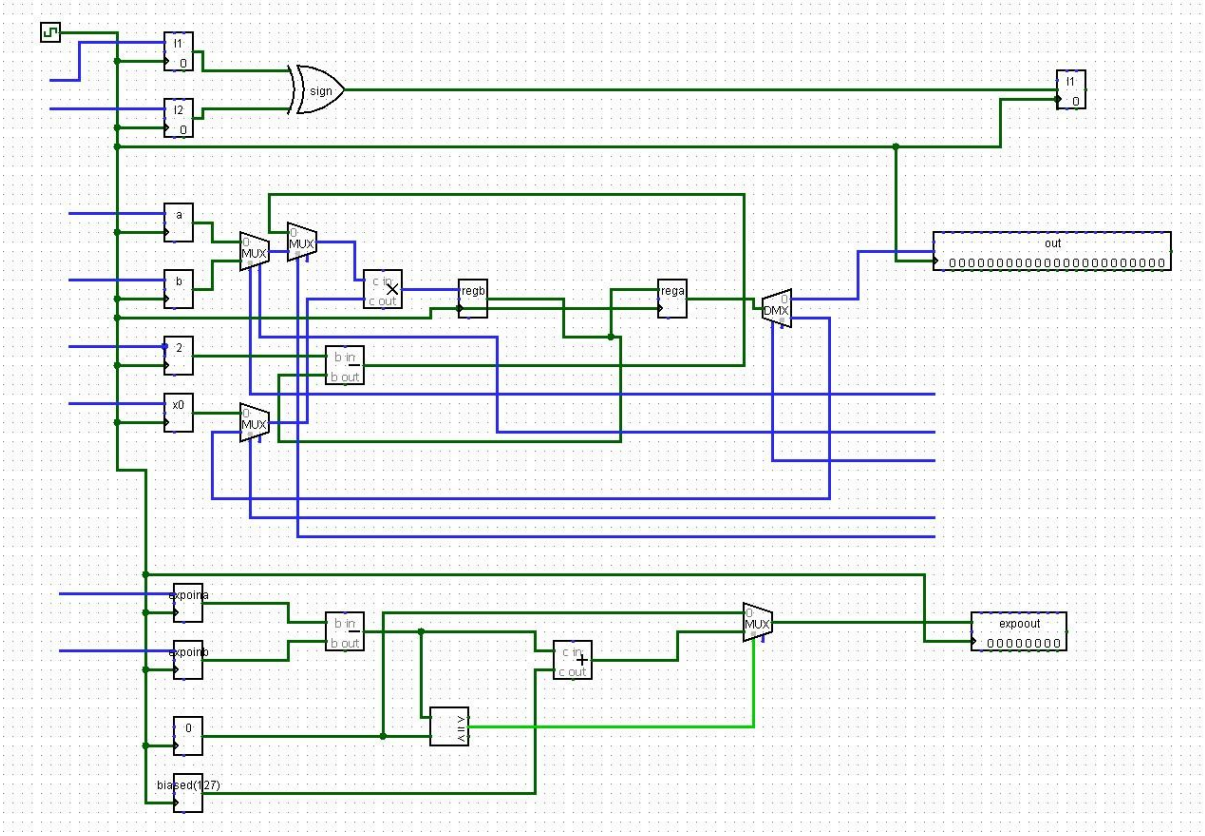


Fig. III.C.3. This figure depicts the final implementation of the divider circuit

D. Square Root

The square root of a 32-bit IEEE floating-point number is also implemented in **three phases**:

1. **Phase1–Sign_Check:**
   The input's sign bit is verified; a negative input is flagged as invalid.
2. **Phase2–Exponent_Adjustment:**
   The exponent is halved, while maintaining biasing:

$$E_{\text{biased, result}} = \frac{E_S}{2} + \text{bias}$$

For zero input ($S = 0$), the exponent is set to zero.

3. **Phase3 – Mantissa Square Root (Newton–Raphson):**
   The **Newton–Raphson square root iteration** is applied:
   - Initial guess $X_0 = 0.5$(0x3F000000)
   - Each iteration refines the estimate using multiplication and addition:

$$X_{n+1} = 0.5 \times (X_n + \frac{S}{X_n})$$

- Ten iterations are typically performed to ensure high precision.

The corresponding circuit diagram illustrates the iterative feedback design used for refinement of the approximation.

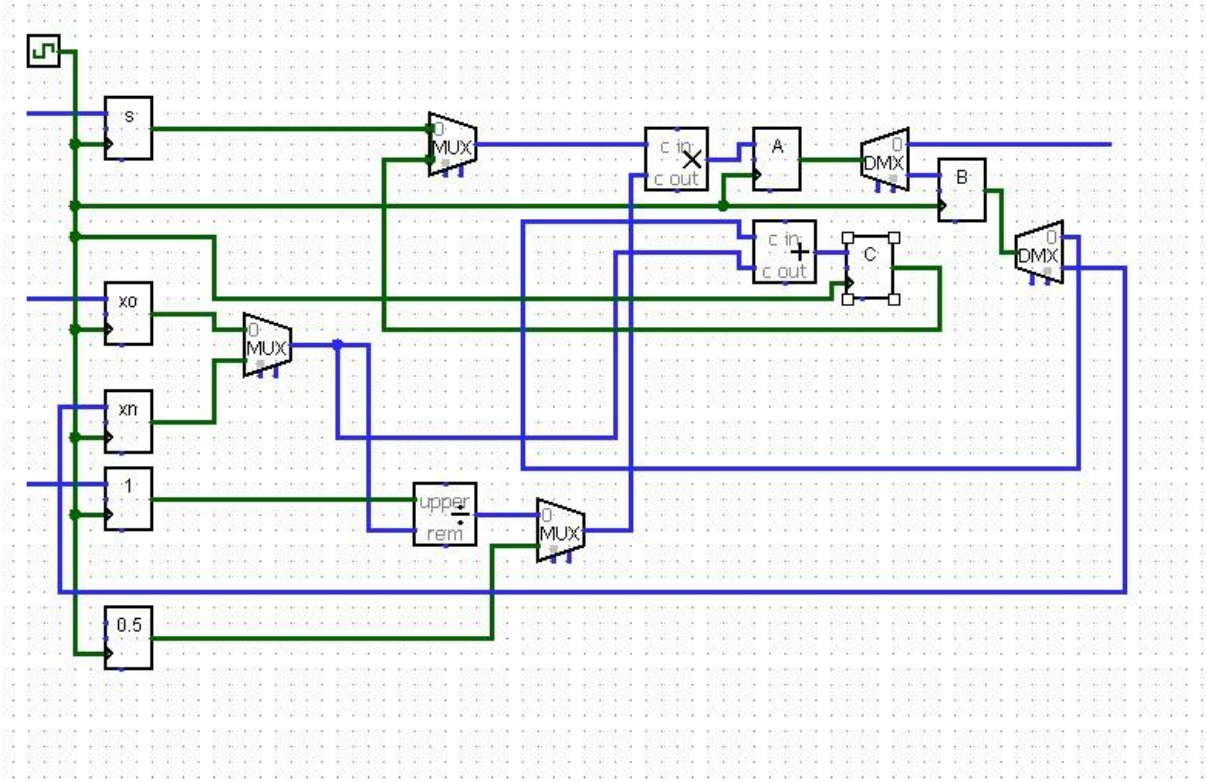


Fig. III.C.4. This figure depicts the final implementation of the square root circuit

E. Implementation in Verilog

All modules were implemented in **Verilog HDL**, with individual submodules developed and integrated into top-level operation modules.

For **addition and subtraction**, custom modules were written for:

- Kogge–Stone adder (24-bit)
- First '1' detector
- 8-bit adder, and associated logic functions.

For **multiplication**, the 24-bit multiplier was mapped to the FPGA's hardware multipliers for optimized synthesis.

For **division** and **square root**, circuits were coded directly in accordance with the architectural diagrams described earlier, with iterative Newton–Raphson loops implemented using register-based feedback control and fixed iteration counters.

D. Semester 4

Floating-Point Division

Introduction

Floating-point division is implemented using the **Newton–Raphson Iteration Method**.
This method computes the division of two numbers by first determining the reciprocal of the denominator's mantissa. The final result is then obtained by multiplying this reciprocal by the numerator's mantissa.

Newton–Raphson Iteration for Division

Let **A** and **B** be two mantissas such that the division is expressed as A/B. According to the Newton–Raphson iteration method, the reciprocal of $B$is computed using the iterative formula:

$$x_{n+1} = x_n \times (2 - D \times x_n)$$

Here,
$x_n = x_0$represents the initial estimate (or "seed"), which should be close to the true reciprocal.
In an FPU (Floating-Point Unit), this initial seed is refined to ensure accurate convergence. The seed is approximated as:

$$x_0 = \frac{48}{17} - \frac{32}{17} \times D$$

where Drepresents the divisor B, adjusted to fit within the range [0.5, 1) by modifying the exponent field to 8'd126.

In the implemented FPU, the iteration process is carried out **three times**, corresponding to three instances of the Newton–Raphson module:

$$\begin{aligned} x_1 &= x_0 \times (2 - D \times x_0) \\ x_2 &= x_1 \times (2 - D \times x_1) \\ x_3 &= x_2 \times (2 - D \times x_2) \end{aligned}$$

The term $x_3$represents the reciprocal of the adjusted divisor D. Subsequently, the exponents are modified to produce the final reciprocal of B. The overall division result is obtained as:

$$\text{Result} = A \times x_3$$

Algorithmic Steps

Extract

- Sign bits: S_A, S_B
- Exponents: E_A, E_B
- Mantissas: M_A, M_B

Approximate $1/B$

- Compute the initial estimate using:

$$x_0 = \frac{48}{17} - \frac{32}{17} \times D$$

Iterate

$$x_{n+1} = x_n \times (2 - D \times x_n)$$

- Each iteration involves one floating-point multiplication and one floating-point addition.
- Three iterations are performed to achieve single-precision accuracy.

Mantissa Multiplication

$$M_{final} = A \times x_{final}$$

Exponent Calculation

$$E_{out} = E_A - E_B + 127$$

If the mantissa MSB > 1, a right shift is performed and $E_{out}$is incremented. Otherwise, left shifting continues until MSB = 1, and the exponent is

decremented.
Rounding to the nearest even is applied based on the guard and sticky bits.

Reassembly
The final floating-point result is obtained as:

$$\text{Output} = [S_A \oplus S_B, E_{out}, M_{out}]$$

Flag Generation

- [4] Divide-by-zero: $B = 0$and $A \neq 0$
- [3] NV (Invalid Operation): $A = 0, B = 0$or NaN inputs
- [2] OF (Overflow): From final exponent range
- [1] UF (Underflow): From final exponent range
- [0] NX (Inexact): Rounding dropped non-zero bits

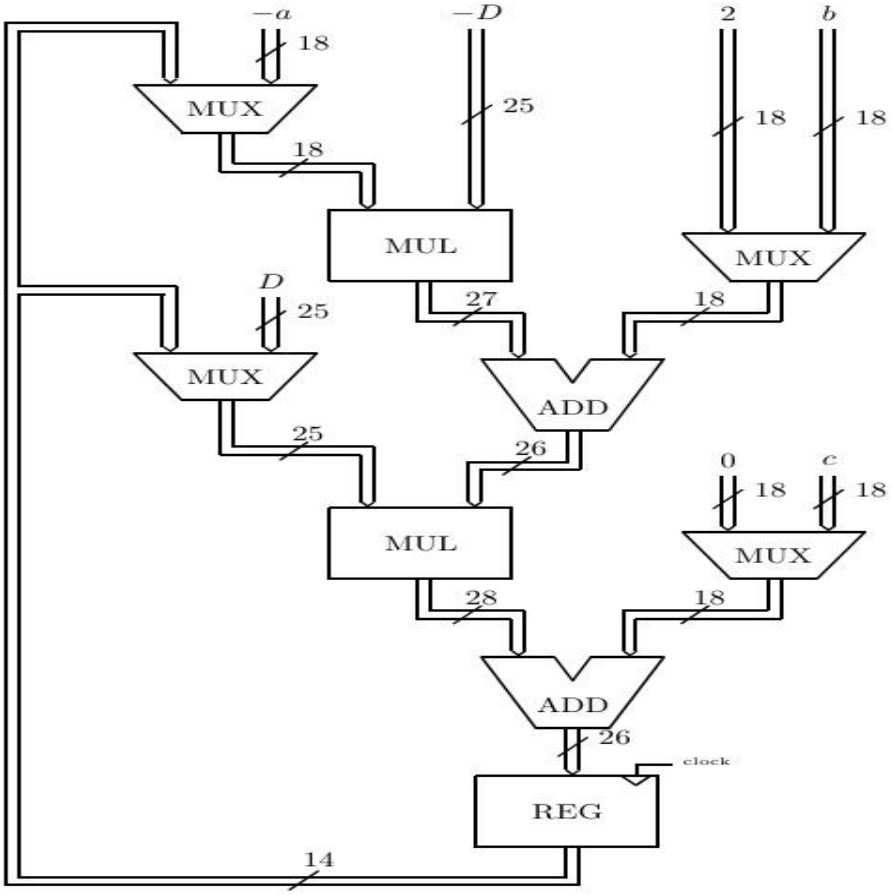


Fig. III.D.1. Division implementation circuit

Floating-Point Square Root

Introduction

The floating-point square root operation is implemented using the **Newton–Raphson Iterative Method**, similar to division, applied to a normalized mantissa with exponent adjustment. It handles all IEEE 754 special cases (zero, subnormal, infinities, NaNs) and generates an invalid-operation flag for negative inputs.

Square Root

Any IEEE 754 number Acan be decomposed as:

$$A = M \times 2^E, M \in [1.0,2.0), E \in \mathbb{Z}$$

Thus,

$$\sqrt{A} = \sqrt{M} \times 2^{E/2}$$

If $E$is even, $2^{E/2}$is an integer power of two.
If $E$is odd, write $E = 2k + 1$, hence:

$$A = \sqrt{M \times 2} \times 2^k = \sqrt{M'} \times 2^k, \text{ where } M' = 2M$$

Mantissa Normalization

To stabilize iteration, the mantissa $M$is biased into the range [0.5, 1.0) by setting its exponent temporarily to 126 (biased exponent = 126 ⇒ true exponent = −1):

$$M_{norm} = M \times 2^{-1}$$

After the iteration, a constant correction factor is applied:

$$C = \frac{1}{\sqrt{2^{-1}}} = \sqrt{2}$$

Newton–Raphson Iteration

For $f(x) = x^2 - M_{norm}$, the iteration formula is:

$$x_{n+1} = \frac{1}{2}(x_n + \frac{M_{norm}}{x_n})$$

The initial seed is:

$$x_0 = 0.853553414345$$

Three iterations are sufficient for 23-bit mantissa precision:

$$x_1 = 0.5 \times (x_0 + \frac{M_{norm}}{x_0})$$
$$x_2 = 0.5 \times (x_1 + \frac{M_{norm}}{x_1})$$
$$x_3 = 0.5 \times (x_2 + \frac{M_{norm}}{x_2})$$

Scaling Result

$$\sqrt{A} = X \times 2^{E/2}$$

- If $E$is even: scale the exponent by $E/2$
- If $E$is odd:
    - Subtract 1 to make Eeven
    - Multiply result by $\sqrt{2}$
    - Adjust exponent: $(E-1)/2$

Reassembly

Final floating-point output fields:

- **Sign:** Always 0 (square root of negative ⇒ NaN)
- **Exponent:** Adjusted after scaling
- **Mantissa:** 23 bits from the normalized square root result

Flag Generation

Special case handling:

- $A = 0$: Result = 0
- $A < 0$: Result = NaN (invalid operation)
- $A = \infty$: Result = ∞
- $A = \text{NaN}$: Result = NaN

FPU Design and Module Integration

Architecture

Controller and Dispatcher

The central controller module manages operational dispatch based on a 3-bit operation code (**op**). It routes operands to the appropriate arithmetic unit and manages result collection.
Two independent address counters track operand destinations for the addition/subtraction and multiplication units, ensuring correct data mapping and sequential result retrieval.

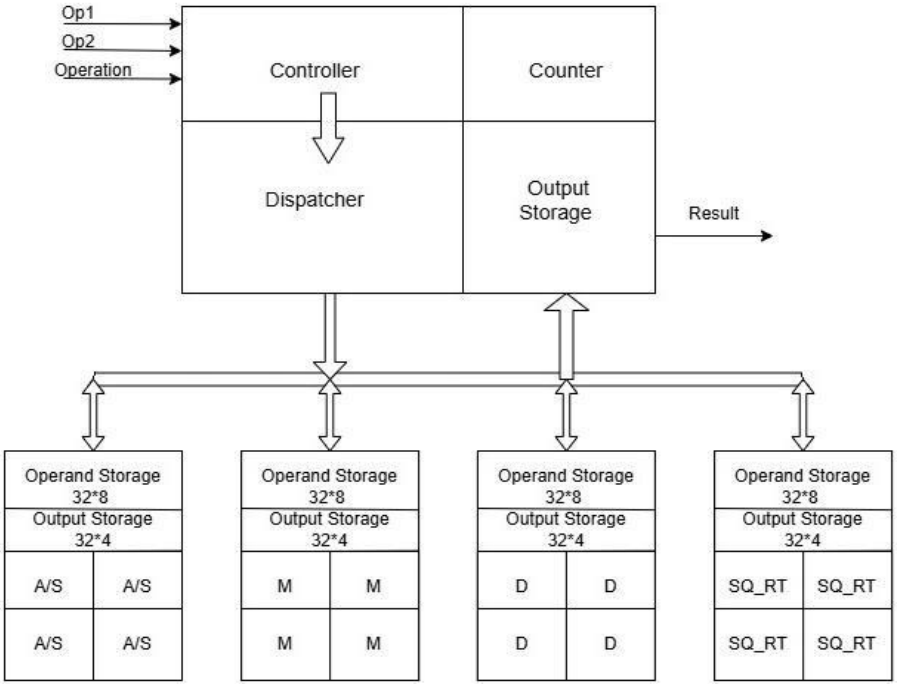


Fig. III.D.2. Top module

Arithmetic Modules

Each arithmetic module consists of four parallel instantiations of its respective operation to maximize throughput.

Modules include:

- **Add/Subtract Module (FPU_A_S):** Implements IEEE 754 compliant floating-point addition and subtraction, handling sign, exponent, and mantissa manipulations.
- **Multiply Module (FPU_M):** Performs IEEE 754 multiplication using operand decomposition and mantissa scaling.
- **Division and Square Root Modules:** Follow optimized algorithms improving precision and reliability over traditional methods.

Each module internally employs selection logic and counters for concurrent operation management and synchronized output delivery.

Working

The controller receives two single-precision floating-point operands and a 3-bit operation code. It forwards them to the **dispatcher**, where each operand is tagged with a counter value to track output order.
The dispatcher routes the operands to their designated operation modules according to the specified operation.

Operands are stored in **eight 32-bit registers**, and the outputs are temporarily held in **four 32-bit registers** before being transferred to the main output registers in the dispatcher module.
The final results are provided in the sequence corresponding to the initially assigned counter values.

E. Semester 5

Top Module Implementation

The implementation of the **Floating-Point Unit (FPU) Top Module** presented in this work is designed to perform high-performance floating-point arithmetic operations through a combination of pipelined and parallel execution techniques. The FPU integrates multiple functional units (Add/Subtract, Multiply, Divide, and Square Root) with a centralized control and scheduling mechanism to manage instruction dispatch, slot allocation, and result reordering. The module adheres to the IEEE-754 floating-point standard for arithmetic precision and consistency.

The core of the design revolves around modularity, concurrency, and scalability. It allows simultaneous execution of multiple floating-point operations while ensuring that results are produced in program order using a **Reorder Buffer (ROB)**. This section presents a comprehensive explanation of the working principles, design methodology, and the detailed operation of each component within the fpu_top module.

The top module of the Floating-Point Unit (FPU) functions as the central controller that manages the entire execution pipeline — from instruction issue to result commitment — ensuring smooth and efficient data flow through multiple functional units (FUs). It integrates several subsystems: the input interface, issue logic, functional unit allocation, result write-back, and a Reorder Buffer (ROB) for in-order commitment. The module is designed to support parallel execution of floating-point addition, subtraction, and multiplication operations while maintaining strict in-order output consistency.

At the input stage, the module receives two 32-bit operands (A and B), a 3-bit operation code (OP), and a ‘valid_in’ signal that indicates when a new operation is available for processing. These inputs are synchronized with the system clock and processed only when the reset signal is inactive. The operation code identifies which arithmetic function is to be performed — addition, subtraction, or multiplication. Each incoming instruction is temporarily stored in the Reorder Buffer (ROB), which acts as the core management structure of the module. The ROB ensures that even when operations finish execution at different times due to varying latencies of functional units, the results are committed to the output strictly in the order they were issued.

The ROB consists of multiple entries, each containing metadata about an issued instruction, including the operation type, result value, validity status, and

completion flag. Two pointers — ‘rob_head’ and ‘rob_tail’ — manage the ROB as a circular buffer. The ‘rob_head’ pointer identifies the next available entry for allocation during instruction issue, while the ‘rob_tail’ pointer identifies the oldest uncommitted instruction that is waiting to be finalized. The ‘rob_count’ register keeps track of how many instructions are currently active within the ROB, and a full condition is detected when the count equals the buffer’s maximum capacity.

When a new instruction is available (‘valid_in’ is high) and the ROB is not full, the issue logic allocates an entry at the current ‘rob_head’ position. The operands, operation type, and ROB tag (the index of the entry in the ROB) are assigned. The tag serves as an identifier that associates the operation’s result with its ROB entry during the write-back stage. Once allocated, the ‘rob_head’ pointer increments to the next position, and the count of active ROB entries increases.

Depending on the operation type, the instruction is dispatched to one of the available functional unit slots. The system uses two separate groups of functional units: one group for addition and subtraction, and another for multiplication. Each group contains four identical parallel processing slots, allowing up to four operations of the same type to be executed simultaneously. To distribute workload evenly among the slots, a round-robin pointer is employed for each group — one for add/sub units and another for multiply units. When an instruction is issued, the logic scans for an idle slot starting from the pointer position and assigns the operation to the first available one. The operands, operation type, and ROB tag are stored into the selected slot, and its start signal is asserted to initiate computation.

Once execution begins, each functional unit operates independently, processing its input data over several cycles depending on the operation’s complexity. The add/sub units perform arithmetic addition or subtraction based on the operation control signal, whereas the multiplier units handle floating-point multiplication. Each unit carries the ROB tag of the issued instruction throughout computation. When the operation completes, the functional unit asserts a done signal, providing both the computed result and the associated tag back to the top module.

The write-back stage monitors all functional units for completion signals. When a done signal is detected, the top module retrieves the corresponding tag and writes the computed result into the matching ROB entry. The ‘rob_done’ flag for that entry is then set to indicate that the result is available and ready for commitment. After storing the result, the functional unit slot is cleared and becomes available for new instructions in subsequent cycles.

The final stage of the process is the commit or retirement phase, which ensures in-order output of results. During each clock cycle, the module examines the ROB entry pointed to by ‘rob_tail’. If this entry is both valid and marked as done, it means the corresponding operation has finished execution and is ready to be committed. The result from the ROB is then sent to the output register, and the ‘valid_out’ signal is asserted for one clock cycle to indicate that a new valid result is available. The committed ROB entry is cleared, its validity and done flags are reset, and both the ‘rob_tail’ pointer and ‘rob_count’ are updated accordingly.

This in-order commitment mechanism ensures program correctness by preventing the output from reflecting out-of-order execution results. Even if multiplication (which typically takes more cycles) finishes later than an addition that was issued afterward, the system will not release results until all earlier instructions have been committed.

In summary, the data flow through the top module follows a structured and synchronized pipeline. The inputs are first validated and stored in the ROB, then dispatched to the appropriate functional unit through dynamic slot allocation. The FUs perform computations independently and return results to the ROB once completed. Finally, the commit logic ensures that all results are output sequentially, maintaining architectural correctness. This design demonstrates efficient parallel processing, dynamic scheduling, and precise in-order result management — core principles in modern floating-point and superscalar processor design.

The corresponding figure shows the high-level implementation of the top module.

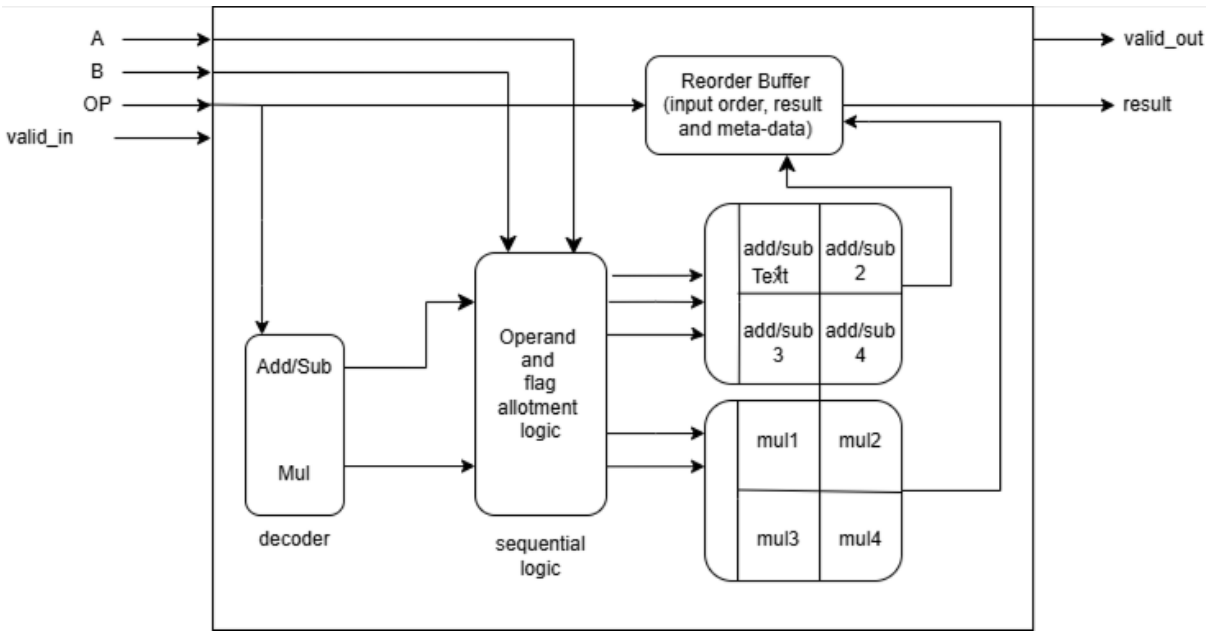


Fig. III.E.1 high level implementation of the top module

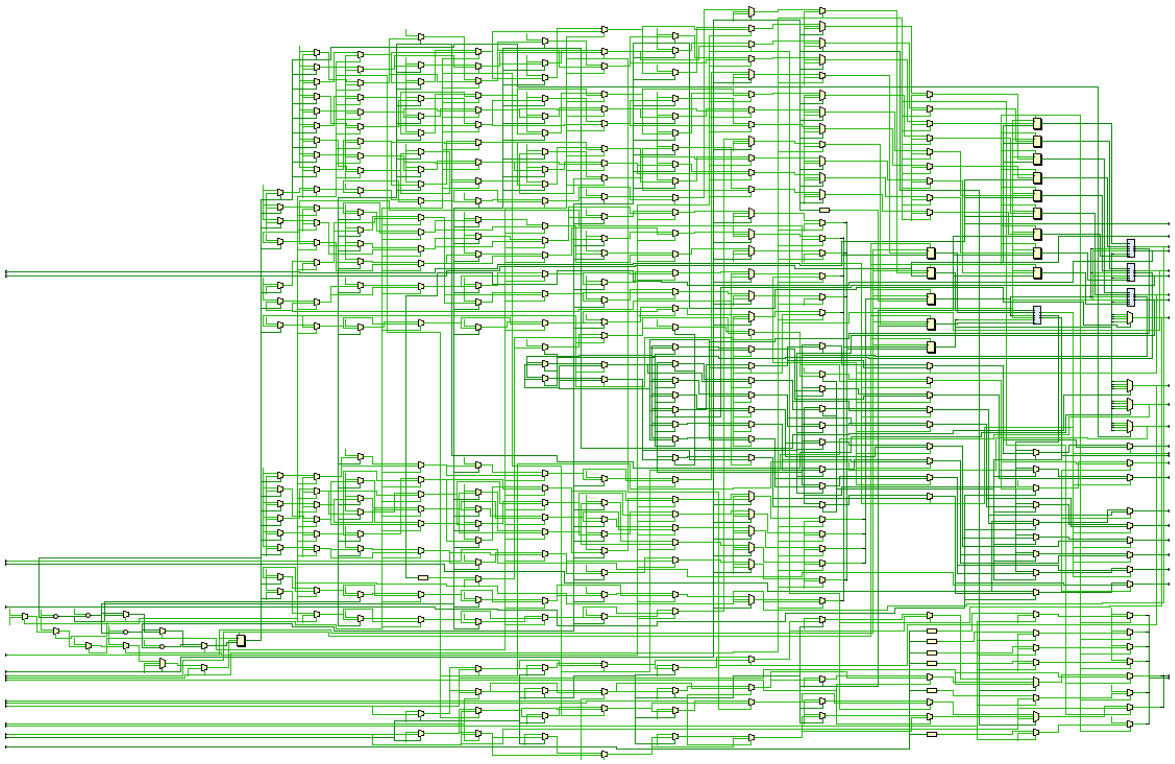


Fig. III.E.2. This figure shows the netlist developed by the Vivado software for the whole design

Floating-Point Division Unit

Algorithm Overview

The division of two floating-point numbers is implemented as:

$$Z = \frac{A}{B} = A \times \frac{1}{B}$$

The core task therefore is to find the reciprocal of B.
The reciprocal is computed using an iterative formula derived from the Newton–Raphson method:

$$y_{n+1} = y_n \times (2 - x \times y_n)$$

where $x = B$and $y_n$is the current estimate of $1/x$.
Each iteration approximately doubles the number of correct bits. Two iterations are sufficient for full single-precision accuracy.

LUT-Based Seed Generation

A **16-entry Look-Up Table (LUT)** provides the initial approximation $y_0$for the reciprocal function.
The LUT maps the normalized mantissa of B(range [1, 2)) to the corresponding value of $1/x$.

LUT Indexing

The top 4 bits of the mantissa $M_B[22{:}19]$are used as the LUT index:

$$\text{Index} = M_B[22{:}19]$$

This divides the [1,2) interval into 16 equal subranges.

LUT Entries

Each LUT entry stores a precomputed IEEE-754 constant corresponding to:

$$y_0[i] = \frac{1}{1 + \frac{i}{16}}, i = 0 \ldots 15$$

The LUT provides an initial estimate accurate to about 10–12 bits.

Example entries (in hexadecimal form):

Index	x Range	1/x	Hex
0000	1.000	1.000	32'h3F800000
0100	1.250	0.800	32'h3F4CCCCD
1000	1.500	0.667	32'h3F2AAAAB
1111	1.937	0.516	32'h3F041041

Iterative Refinement Logic

After obtaining y_0, two Newton–Raphson iterations are performed:

$$\begin{aligned} t &= x \times y_n \\ y_{n+1} &= y_n \times (2 - t) \end{aligned}$$

The refined value $y_2$represents the reciprocal of $x$to within 1 ULP. Finally, the dividend $A$is multiplied by $y_2$to obtain the quotient Z:

$$Z = A \times y_2$$

All arithmetic operations are implemented with IEEE-754 floating-point modules.
Pipeline registers are inserted after multiplication stages to ensure proper timing closure on FPGA.

Exponent Handling in Division

In IEEE-754 format, the result exponent for division is calculated as:

$$E_Z = (E_A - E_B) + 127$$

where 127 is the single-precision bias.

Steps in exponent handling:

1. Extract exponent fields of both operands: $E_A = A[30:23]$, $E_B = B[30:23]$
2. Perform the difference $E_A - E_B$

3. Add bias (127) to bring the result back into IEEE range
4. Apply normalization adjustments if mantissa overflows or underflows after iteration

Normalization Adjustment:

- If mantissa ≥ 2 → shift right by 1, increment exponent
- If mantissa < 1 → shift left by 1, decrement exponent

The sign bit of the result is determined by:

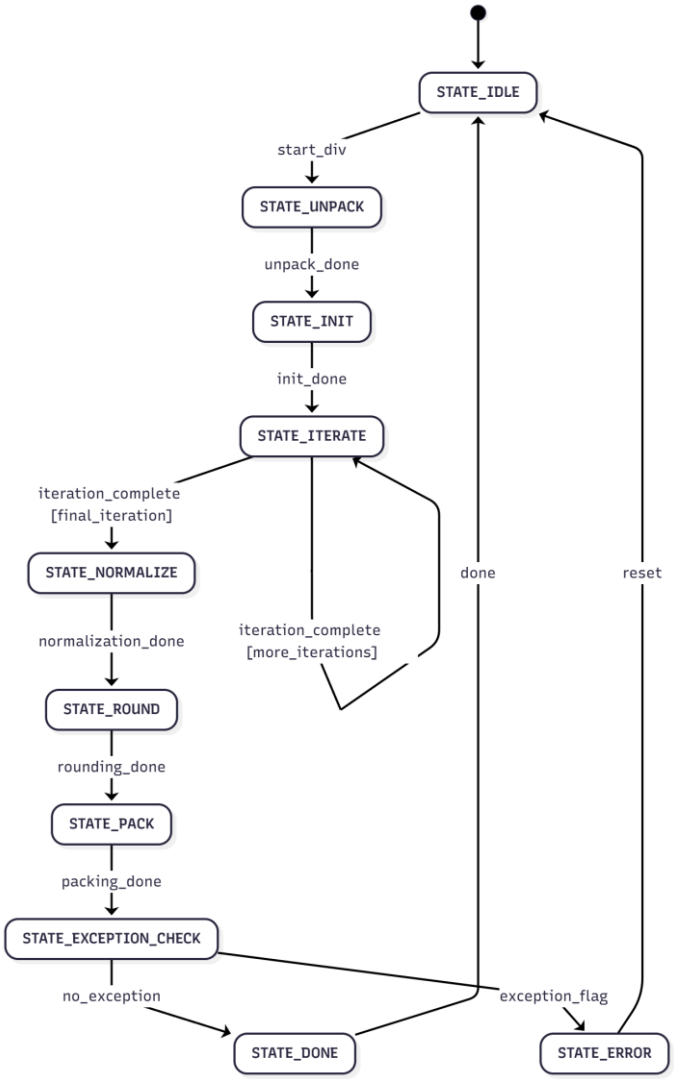
$$S_Z = S_A \oplus S_B$$

Special Case and Exception Handling

The design includes dedicated logic to detect and handle IEEE-754 exceptions before the iteration stage.

Condition	Output	Action
B = 0	±∞	Set exponent to 255, mantissa = 0
A = 0, B ≠ 0	0	Zero output
A or B = NaN	NaN	Propagate NaN
A = ±∞, B finite	±∞	Infinite result
A finite, B = ±∞	±0	Result zero

If an exception is detected, the iterative path is bypassed and the output is forced to the correct IEEE-754 constant (0, ∞, or NaN).



III.E.3. dataflow chart for division

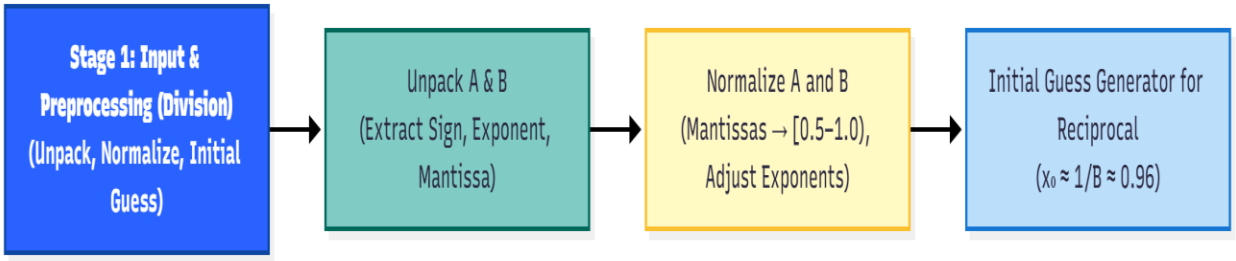


Fig. III.E.4. Implementation Diagram for division (Part 1)

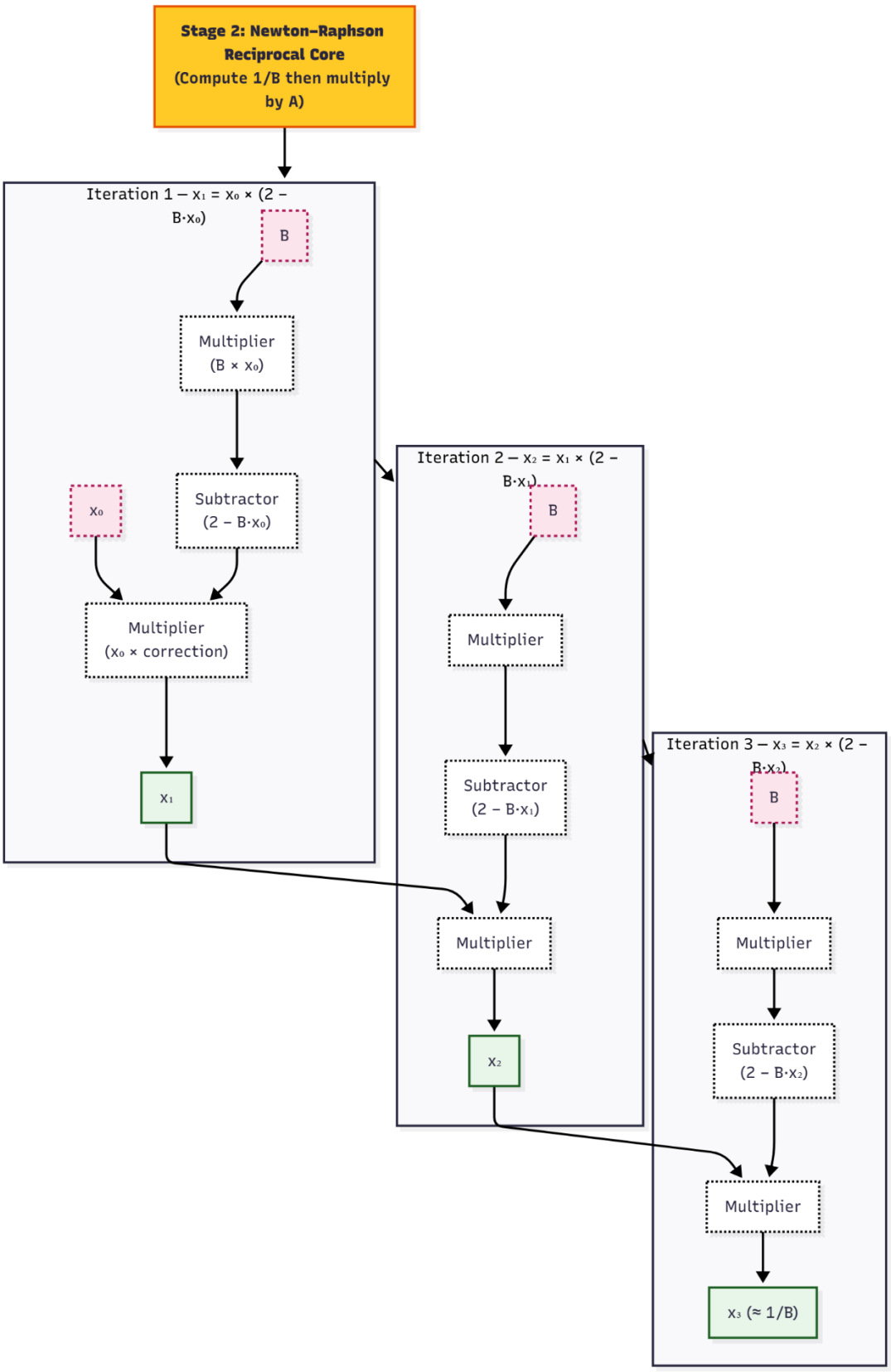


Fig. III.E.5. Implementation Diagram for division (Part 2)

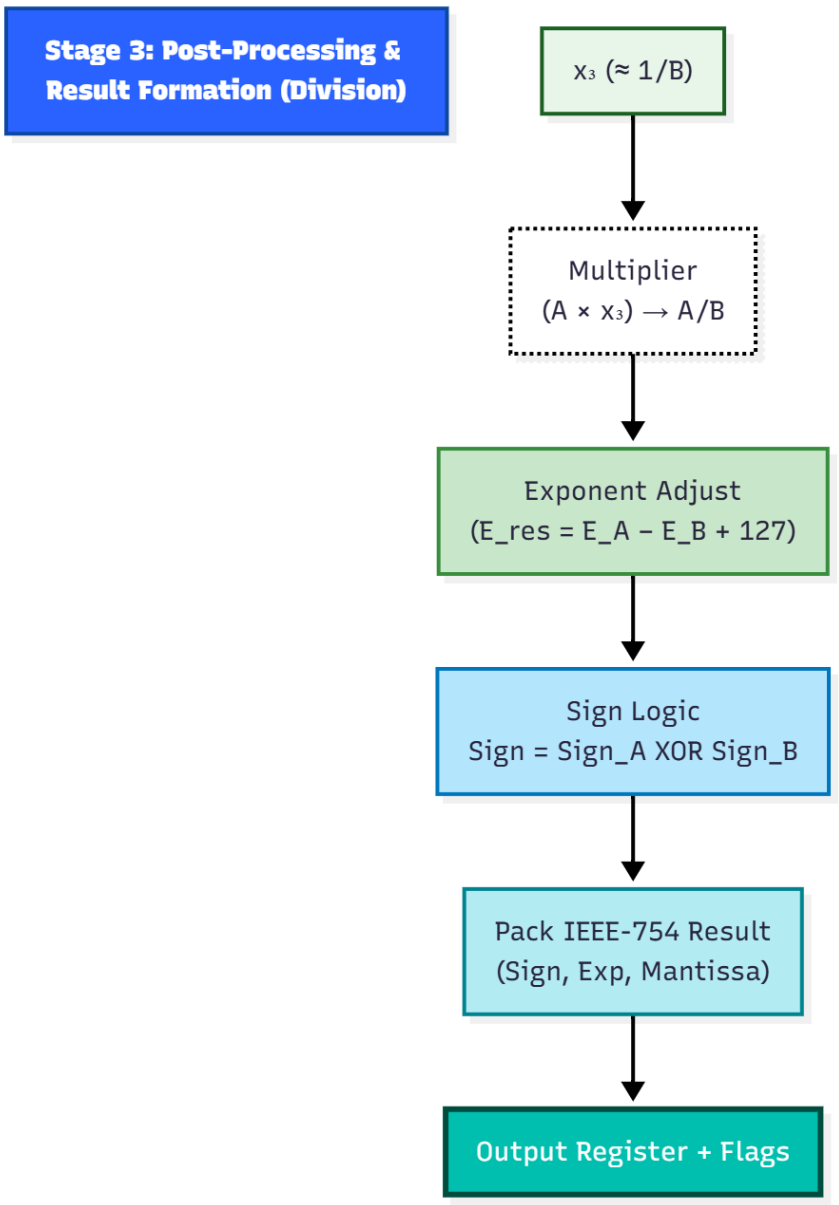


Fig. III.E.6. Implementation Diagram for division (Part 3)

Floating-Point Square Root Unit

Algorithm Overview

The square root function is implemented using the **reciprocal square root method**.

The iteration refines an estimate $y$of $1/\sqrt{x}$using the Newton–Raphson form:

$$y_{n+1} = 0.5 \times y_n \times (3 - x \times y_n^2)$$

After two iterations, $y_2$achieves single-precision accuracy. The final square root result is obtained as:

$$Z = x \times y_2$$

This method avoids direct division and uses only multiplications and subtractions, making it well suited for hardware implementation.

LUT-Based Seed Generation

The seed $y_0$is fetched from a 16-entry LUT that approximates $1/\sqrt{x}$for $x \in [1,2)$.

LUT Indexing

Similar to division, the top 4 bits of the mantissa determine the LUT index:

$$\text{Index} = M_x[22:19]$$

LUT Entries

Each LUT entry corresponds to:

$$y_0[i] = \frac{1}{\sqrt{1 + \frac{i}{16}}}$$

Values are encoded in IEEE-754 hexadecimal format. Example entries:

Index	**x Range**	**1/√x**	**Hex**
0000	1.000	1.000	32’h3F800000
0100	1.250	0.894	32’h3F665555
1000	1.500	0.816	32’h3F4E147B
1111	1.937	0.693	32’h3F316872

Exponent Handling in Square Root

For square root:

$$\sqrt{M \times 2^E} = \sqrt{M} \times 2^{E/2}$$

Thus, the exponent of the result is half of the input exponent.

Steps in exponent adjustment:

1. Extract $E_x = X[30:23]$
2. Compute $E_Z = \lfloor \frac{E_x - 127}{2} \rfloor + 127$
3. If $E_x$is odd, multiply the mantissa by $\sqrt{2}$(≈1.4142) or equivalently increment it in normalization by one half step.

This ensures the final result is properly normalized to maintain the IEEE-754 standard range.

The sign bit is set to 0, since the square root of a real number is always non-negative.

Iterative Refinement

The refinement process follows:

$$\begin{aligned} t &= x \times y_n^2 \\ y_{n+1} &= 0.5 \times y_n \times (3 - t) \end{aligned}$$

Each iteration doubles the accuracy. Two iterations produce results within 1 ULP of the true value.

Normalization of mantissa is performed after each iteration if necessary.

Special Case and Exception Handling

Condition	Output	Description
x = 0	0	√0 = 0
x < 0	NaN	Invalid (negative input)
x = ∞	∞	√∞ = ∞

Condition	Output	Description
x = NaN	NaN	Propagate NaN

When a negative input is detected, the result is set to a quiet NaN (32'h7FC00000).
All exception cases bypass the iterative logic.

Normalization, Reconstruction, and Rounding

After the final iteration:

1. **Normalization:**
   - If mantissa ≥ 2.0 → right shift and increment exponent
   - If mantissa < 1.0 → left shift and decrement exponent
2. **Reconstruction:**
   The final 32-bit IEEE result is formed as:

$$\text{Result} = \{\text{Sign,Exponent}[7:0],\text{Mantissa}[22:0]\}$$

3. **Rounding:**
   Implemented using "round to nearest even" to minimize quantization error. This rounding mode guarantees unbiased rounding across large datasets.

Hardware Integration

Both units share the same design hierarchy:

1. **Input Preprocessing Block** – Unpacks the floating-point number into sign, exponent, and mantissa.
2. **LUT Block** – Generates initial seed value y_0.
3. **Iteration Block** – Performs two Newton–Raphson updates.
4. **Exponent Logic** – Adjusts bias and normalization.
5. **Special Case Handler** – Produces NaN, zero, or infinity as needed.
6. **Result Assembler** – Packs the output back into IEEE-754 format

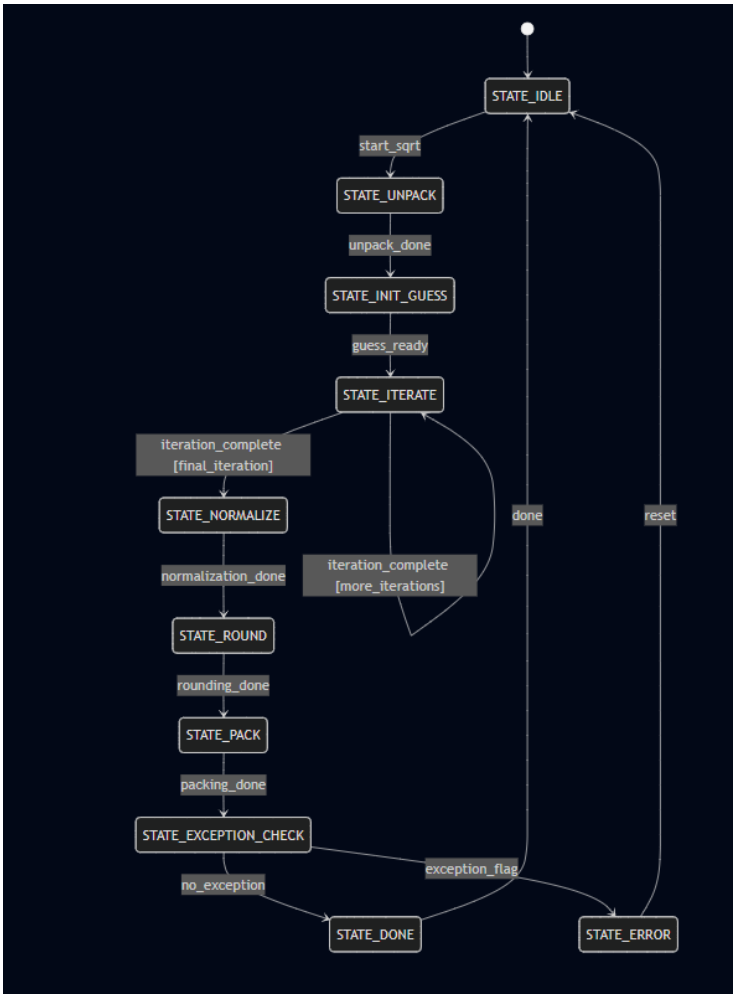


Fig. III.E.7. Flowchart for square root

Implementation Diagram

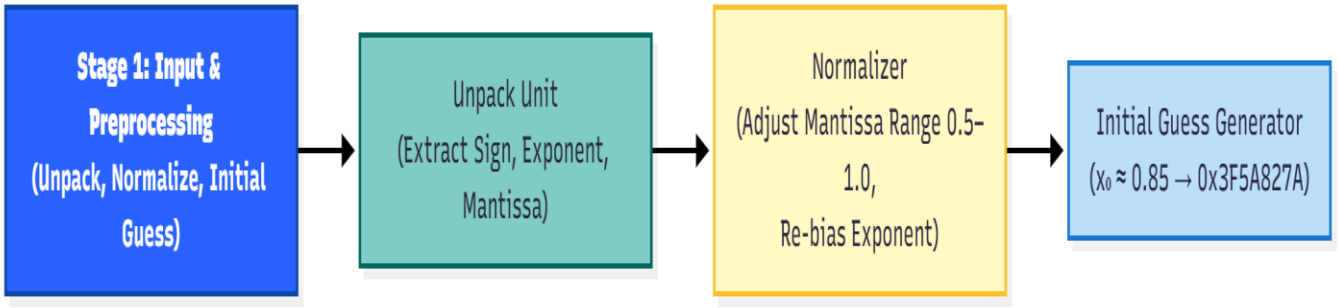


Fig. III.E.8. Implementation Diagram for square root (Part 1)

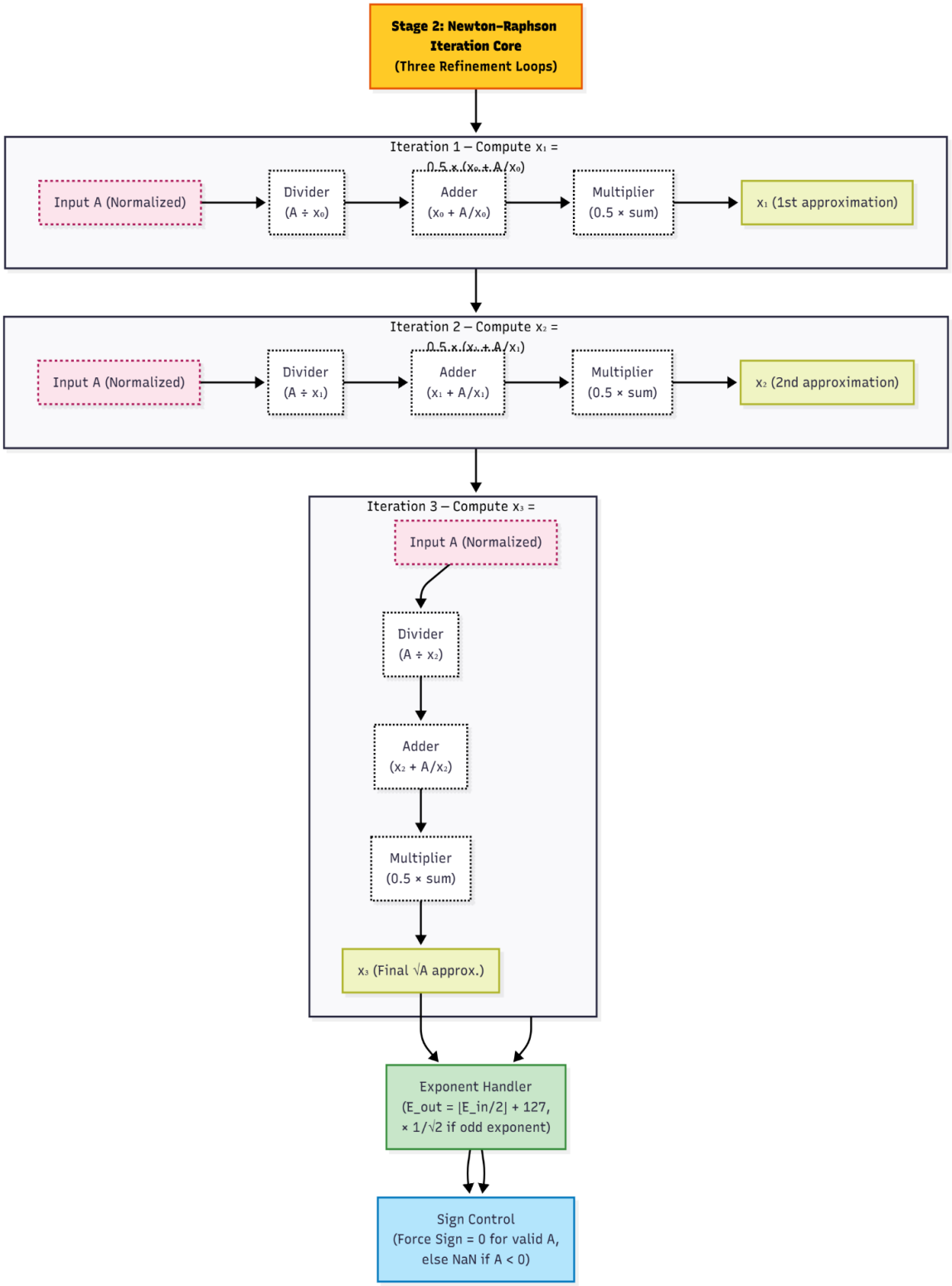


Fig. III.E.9. Implementation Diagram for square root (Part 2)

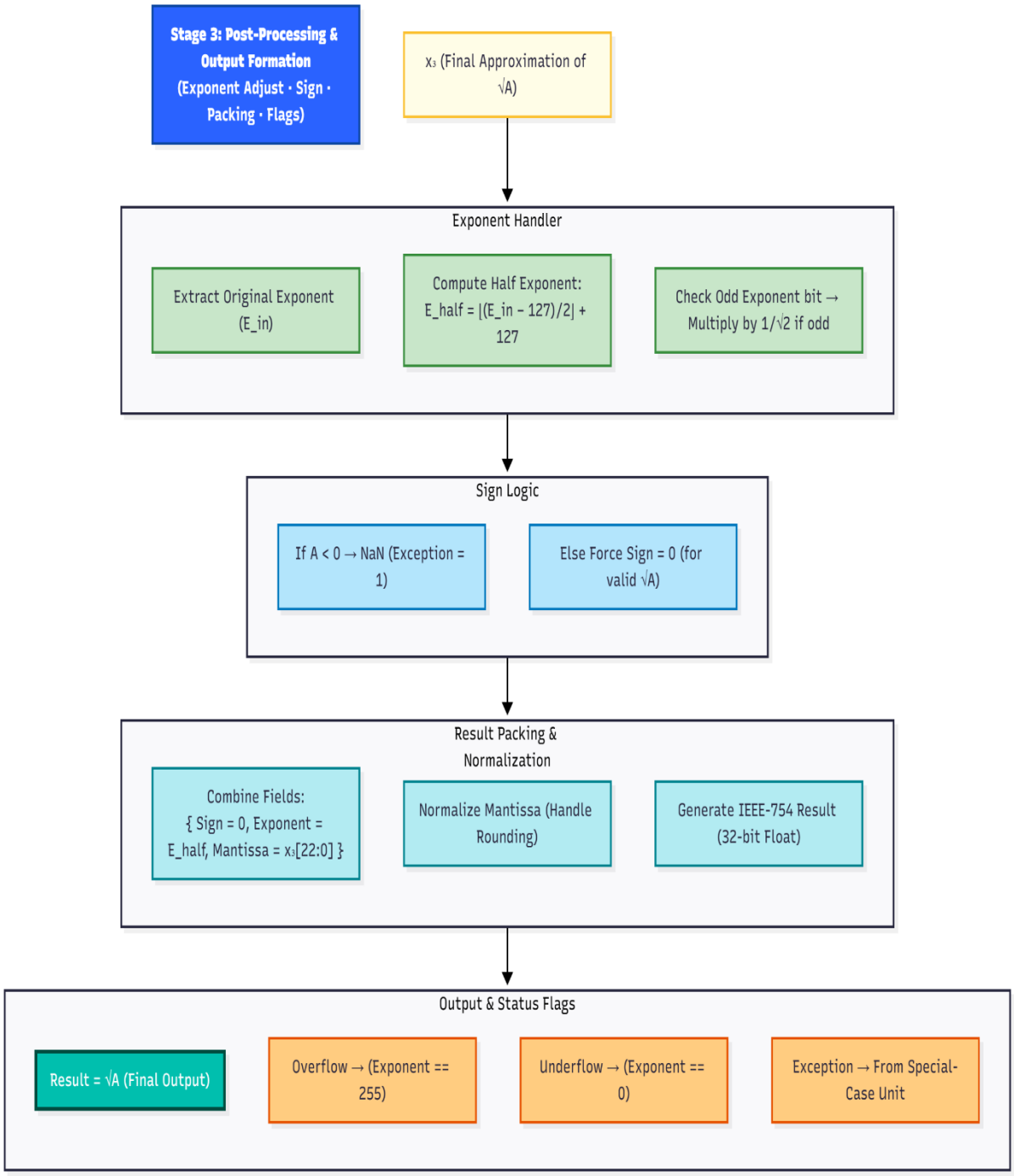


Fig. III.E.9. Implementation Diagram for square root (Part 3)

Accuracy Analysis of LUT-Based Newton–Raphson Approximation

The accuracy of the proposed floating-point arithmetic units is evaluated by comparing the initial LUT-based approximation with successive Newton–Raphson (NR) refinement stages.
The LUT provides a fast but coarse initial estimate, while each NR iteration quadratically improves accuracy by using the feedback formula.

For the **division unit**, the reciprocal of the divisor is computed iteratively as:

$$x_{n+1} = x_n \times (2 - B \cdot x_n)$$

and the final result is obtained as $A/B = A \times x_3$.
For the **square-root unit**, the iterative update follows:

$$x_{n+1} = 0.5 \times (x_n + \frac{A}{x_n})$$

where $x_0$is the initial guess from the LUT.

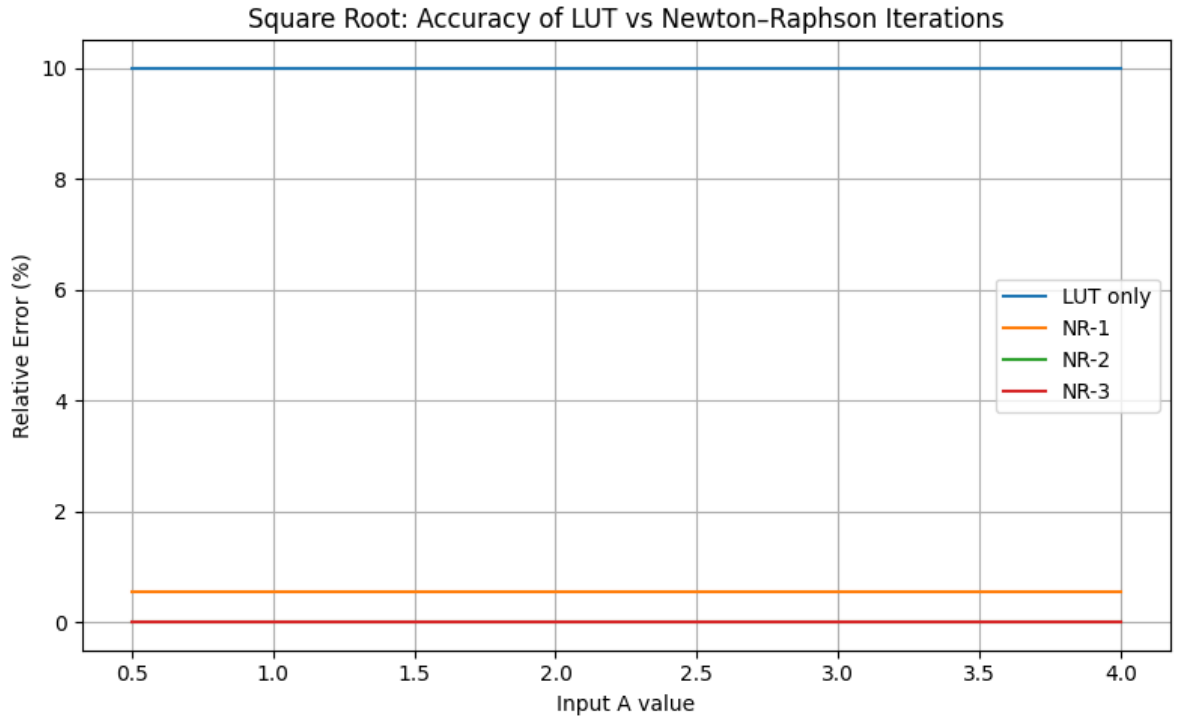


Fig. III.E.10 Accuracy Comparison of LUT and Newton–Raphson Iterations for Floating-Point Square Root

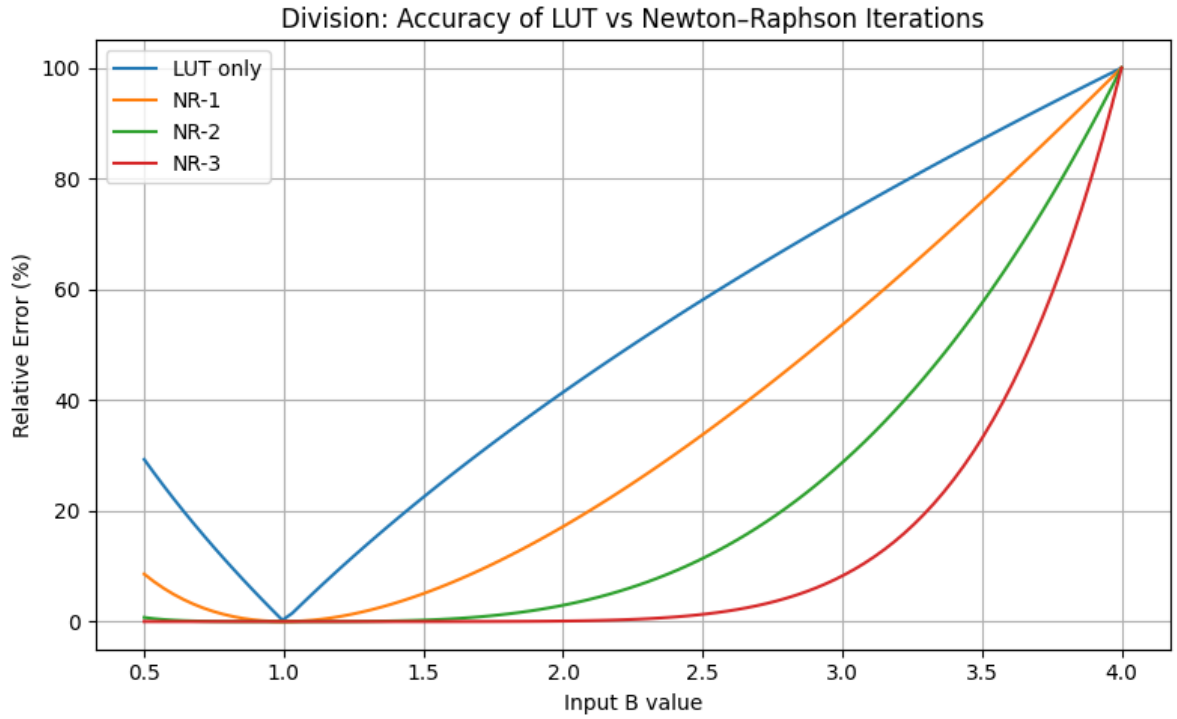


Fig. III.E.11. Accuracy Comparison of LUT and Newton–Raphson Iterations for Floating-Point Division

Figures (III.E.10) and (III.E.11) show the **relative error (%)** between the computed and actual results across a wide input range. The initial LUT-only approximation produces a deviation in the order of $10^{-1}$to 10^{-2}. After one NR iteration, the error reduces by nearly two orders of magnitude, and by the third iteration, the results converge very closely to the true IEEE-754 value (error < 10^{-5}%).

This demonstrates that **Newton–Raphson refinement provides exponential convergence**, effectively making a low-precision LUT sufficient for hardware implementations where area and speed are critical.

To evaluate the influence of LUT precision on the overall system accuracy, the number of index bits used for address decoding in the lookup table was varied from 4 to 12 bits. A higher index width corresponds to finer granularity and a larger table size, thereby providing a closer initial approximation to the true reciprocal or square-root value.

Figure (c) illustrates the **relationship between LUT index resolution and approximation error**. As the number of index bits increases, the deviation from the true value decreases exponentially according to:

$$\text{Error} \propto \frac{1}{2^{N_{\text{index}}}}$$

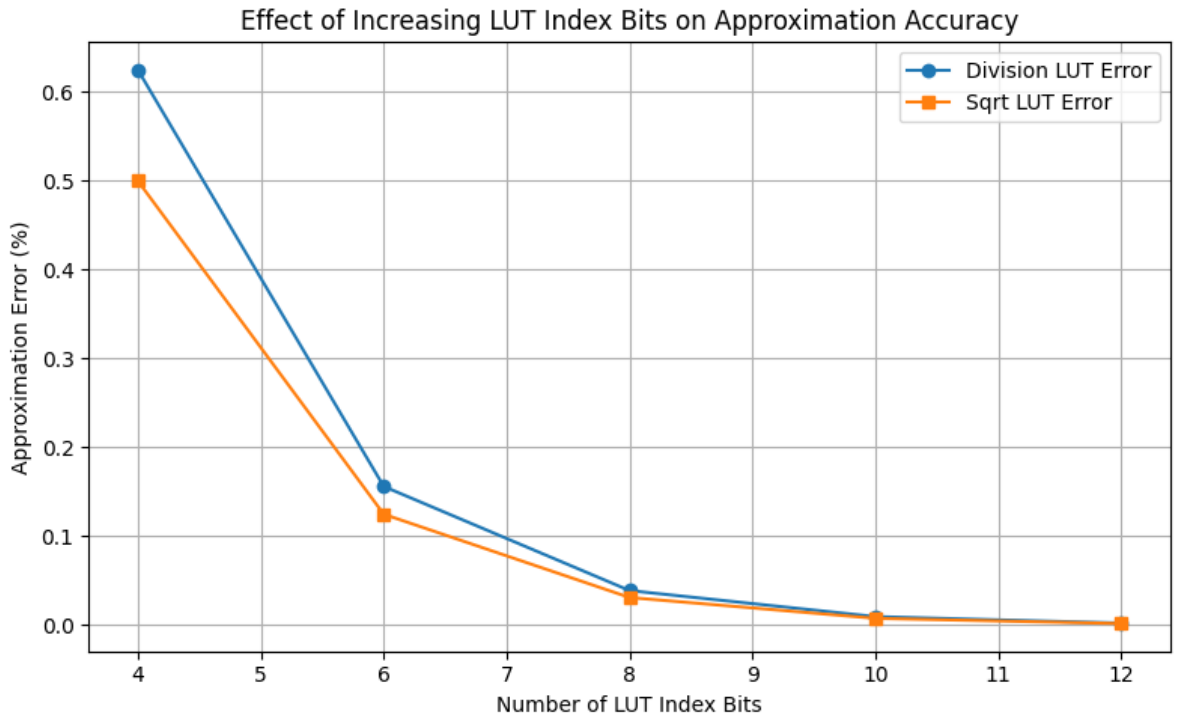


Fig. III.E.11. Reduction in approximation error with increasing LUT index resolution for floating-point division and square-root operations.

It is observed that even a **6-bit index LUT** yields acceptable precision for subsequent NR iterations, striking a good trade-off between hardware complexity and accuracy. Beyond 10 bits, the error improvement becomes marginal, indicating diminishing returns for larger tables.

IV. Results

The results of the modules developed over the time are complied in this section and are discussed about.

A. Semester 2

a. INTEGER FORMAT

1. Addition

INPUT 1	INPUT 2	OUTPUT	EXPEXTED OUTPUT	MATCH
4568	2548	7116	7116	YES
148	362	510	510	YES
2	9	11	11	YES
15626	93248	108874	108874	YES

2. Subtraction

INPUT 1	INPUT 2	OUTPUT	EXPECTED OUTPUT	MATCH
45	36	9	9	YES
17289	1827	15462	15462	YES
23	125	-102	-102	YES
15	15	0	0	YES

3. Multiplication

INPUT 1	INPUT 2	OUTPUT	EXPECTED OUTPUT	MATCH
12	13	156	156	YES
1245	6548	8152260	8152260	YES
25	25	625	625	YES
256	0	0	0	YES

4. Division

INPUT 1	INPUT 2	OUTPUT(QUOTIENT)	EXPECTED OUTPUT	REMAINDER	MATCH
21	3	7	7	0	YES
1234	11	112	112.18	2	NO
11	15	0	0.733	11	NO
625	5	125	125	0	YES

B. FIXED-POINT FORMAT

1. Addition and Subtraction

Input 1	**Input 2**	**Output**	**Expected Output**	**Match**
7.77	6.88	14.65	14.65	YES
8.62	9.00	17.62	17.62	YES
8.42	77.99	86.41	86.41	YES
(Subtraction) 7.77	6.88	0.89	0.89	YES
8.62	9.00	-0.38	-0.38	YES
8.42	77.99	-69.57	-69.57	YES

2. Multiplication

Input 1	**Input2**	**Output**	**Output Expected**	**Match**
31.22	44.67	1373.68	1394.59	NO (20.91 diff)
15	20	300	300	YES
18	-17	-306	-306	YES
-11.1	-7	77.7	77.7	YES

3. Division

Input1	**Input2**	**Output(quotient)**	**Output expected(quotient)**	**Match**
77.11	-6.88	-11	-11	YES
6.77	0	VALUE ERROR	NOT DEFINED	YES
3.14	66.88	0	0	YES

C. FLOATING-POINT FORMAT

1. Addition

Input 1	Input 2	Output	Expected Output	Match
42700000 (60)	41b80000 (23)	42a60000 (83)	83	Yes
42560000 (53.5)	c1c2f5c3(-24.37)	41e90a3d (29.13)	29.13	Yes
7f800000 (infinity)	40b00000 (5.5)	7f800000 (infinity)	Infinity	Yes
c51ae72f (-2478.4490)	c62f8801 (-11234.0006)	c65641cd (-13712.45)	13,712.4496	No Mismatch = 0.004
3de5fd8b (0.1123)	3db39192 (0.08768)	3e4cc78e (0.19997999)	0.19998	No Mismatch = 0.000000001

2. Subtraction

Input 1	Input 2	Output	Expected Output	Match
42700000 (60)	41b80000 (23)	42140000 (37)	37	Yes
41b80000 (23)	42700000 (60)	c2140000 (-37)	-37	Yes
bf2d42a2 (-0.676798)	bf2d42a2 (-0.676798)	00000000 (0)	0	Yes
4206b81d (33.6798)	c4454029 (-789.0025)	444dabab (822.6823)	822.6823	Yes
4841ad0f (198324.2332)	4cbb7d20 (98298111.222187)	ccbb1c49 (-98099784)	-98099786.988987	NO Mismatch = 2.988987

3. Multiplication

Input 1	Input 2	Output	Output Expected	Match
41200000 (29.5)	40a0a3d7 (5.02)	4314170a (148.09)	148.09	YES
c2060000 (-33.5)	42300000 (44)	c4b84000 (-1474)	-1474	YES
c29b8d50 (-77.776)	c2607df4 (-56.132)	45886dc7 (4365.7222)	4365.7224	NO(Mismatch = 0.0002)

7f800000 (infinity)	41200000 (10)	7f800000 (infinity)	infinity	YES
3983126f (0.00025)	3827c5ac (0.00004)	3218e69d (0.0000000089	0.000000001	NO Mismatch= 0.0000000001

4. Division

Input1	Input2	Output	Expected Output	Match
42c80000 (100)	41200000 (10)	41200000 (10)	10	Yes
40280000 (2.625)	c0a00000 (-5)	bf066666 (-0.525)	-0.525	Yes
4293555a (73.667)	3eaaa64c (0.3333)	435d05b0 (221.0222)	221.0231	No Mismatch = 0.0009
b7d1b717 (-0.000025)	b356bf95 (-0.00000005)	43f9ffff (499.99997)	500	No Mismatch = 0.00003
53b67d6c (1567576576998)	4a0f5bd4 (2348789)	4922f05c (667397.75)	667397.785	No Mismatch = 0.035

B. Semester 3

Addition/Subtraction

A	B	Op	Sum/Difference	Value	Error
42700000 (60.0)	41b80000 (23.0)	0	42a60000	83	0
42700000 (60.0)	41b80000 (23.0)	1	42140000	37	0
41b80000 (23.0)	42700000 (60.0)	1	c2140000	-37	0
c2700000 (-60.0)	c1b80000 (-23.0)	0	c2160000	-83	0
4206b81d (33.6798)	c4454029 (-789.0025)	0	c43cd2a8	-755.2915	0.0312
4206b81d (33.6798)	c4454029 (-789.0025)	1	444dabaa	822.68225	0.00005
4206b81d (33.6798)	4841ad0f (198324.2332)	0	4841b57a	198357.9	0.013
bf2d42a2 (-0.676798)	4841ad0f (198324.2332)	0	4841ace4	198323.56	0.0036
bf2d42a2 (-0.676798)	4841ad0f (198324.2332)	1	c841ad3a	-198324.9	0.01
bf2d42a2 (-0.676798)	3e4f5c29 (0.2025)	0	bef2d730	-0.474298	0
3e4f5c29 (0.2025)	bf2d42a2 (-0.676798)	1	3f6119ac	0.879298	0

Multiplication

A	B	Result	Value	Error
404ccccd (3.2)	40866666 (4.2)	41570a3d	13.44	0
bf000000 (0.5)	40cccccd (6.4)	c04ccccd	3.2	0
3983126f (0.00025)	3827c5ac (0.00004)	322bcc77	0.00000001	0
42700000 (60)	4206b81d (33.6798)	44fc9936	2020.7878	0.0002
bf2d42a2 (-0.676798)	c4454029 (-789.0025)	44057fb3	533.9953	0.000014
4841ad0f (198324.2332)	4206b81d (33.6798)	4acbd7c0	6679520	0.509
45a34000 (5224)	4206b81d (33.6798)	482bd1d0	175943.25	0.0252
42e0ab44 (112.3345)	c13c75c3 (-11.77875)	c4a5651f	-1323.16	0.000001
c5a34000 (-5224)	c5a34000 (-5224)	4bd03520	27290176	0
bf2d42a2 (-0.676798)	4841ad0f (198324.2332)	c803145c	-134225.44	1.97

Division

Input 1	Input 2	Initial guess	Result
0x3f800000(1)	0x40000000(2)	0x3f800000(1)	3f000000(0.5)
0x40633333(3.55)	0xc0633333(-3.55)	0x3f800000(1)	0xbf800000(-1)
0xc0b51eb8(-5.66)	0xc0c00000(-6)	0x3f800000(1)	0x3f717e4b (0.9433)
0x447a0000(1000)	0x40a00000(5)	0x3e4ccccd (0.2)	0x43480000
0x00000000(0)	0x477f4800	0x3f800000(1)	0x00000000(0)

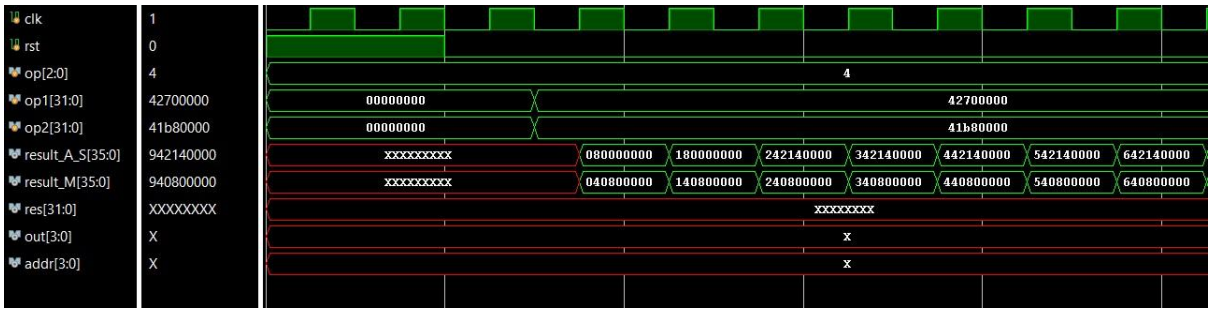
Square root

Input	Initial Guess	Output
0x40000000(2)	0x3f800000(1)	0x3fb504f3(1.4142)
0x40400000(3)	0x3f800000(1)	0x3fddb3d7(1.732)
0x00000000(0)	0x3f800000(1)	0x00000000(0)
0xbf800000(-1)	0x3f800000(1)	“Value error”

C. Semester 4

SL.NO	Input (A)	Output (result)	Flags	Description
1	0x00000000	0x00000000	0	√0 = 0.0 (no flag)
2	0x3F800000	0x3F800000	0	√1 = 1.0 (no flag)
3	0x40800000	0x40000000	0	√4 = 2.0 (no flag)
4	0x41C80000	0x40A00000	0	√25 = 5.0 (no flag)
5	0x7F800000	0x7F800000	0	√∞ = ∞ (no flag)
6	0xBF800000	0x7FC00000	10000	√(−1) = NaN (invalid)
7	0xC0800000	0x7FC00000	10000	√(−4) = NaN (invalid)
8	0x7FC00000	0x7FC00000	10000	√(NaN) = NaN (invalid)
9	0xBF000000	0x7FC00000	10000	√(−0.5) = NaN (invalid)

FPU simulation result:



As seen in the results over the semesters, the algorithm for the operations went on getting stronger and more accurate.

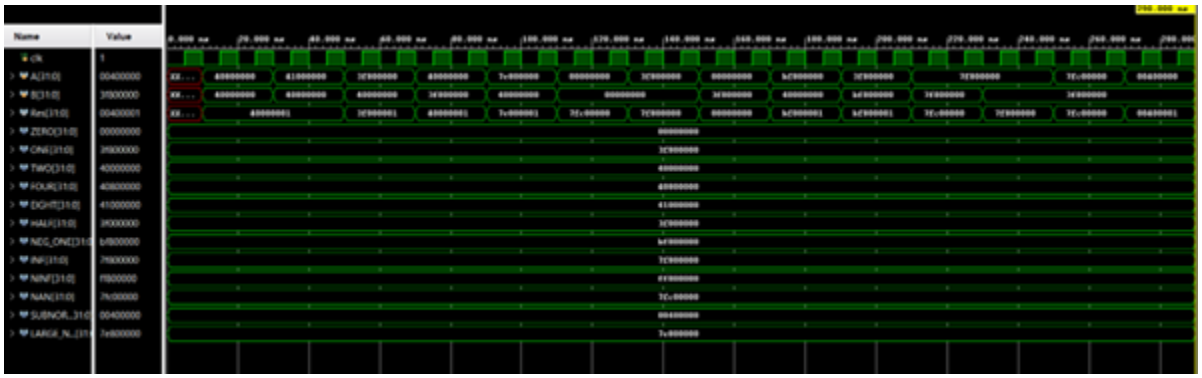
The TOP module was designed in the last semester of the Capstone project, but was not able to get compiled along with the operation modules due to errors being caused.

D. Semester 5

Division

A (Hex)	B (Hex)	Result (Hex)	A (Float)	B (Float)	Result (Float)	Expected (A/B)	Deviation (%)
40800000	40000000	40000001	4	2	2	2	1.20E-05
41000000	40800000	40000001	8	4	2	2	1.20E-05
3f800000	40000000	3f000001	1	2	0.5	0.5	1.20E-05
40000000	3f800000	40000001	2	1	2	2	1.20E-05
7e800000	40000000	7.00E+01	8.51E+37	2	4.25E+37	4.25E+37	1.20E-05
0	0	7fc00000	0	0	inf	inf	
3f800000	0	7f800000	1	0	inf	inf	
0	3f800000	0	0	1	0	0	
bf800000	40000000	bf000001	-1	2	-0.5	-0.5	1.20E-05
3f800000	bf800000	bf800001	1	-1	-1	-1	1.20E-05
7f800000	7f800000	7fc00000	inf	inf	inf	inf	
7f800000	3f800000	7f800000	inf	1	inf	inf	
7fc00000	3f800000	7fc00000	inf	1	inf	inf	
400000	3f800000	400001	0	1	0	0	2.40E-05

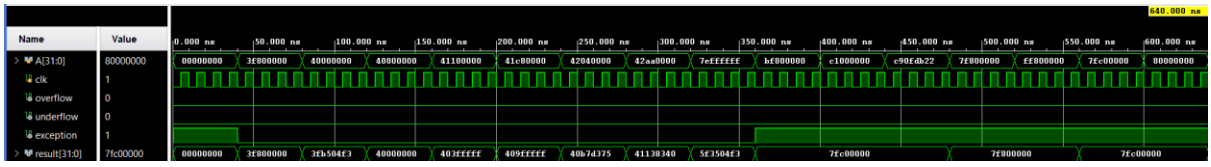
Waveform



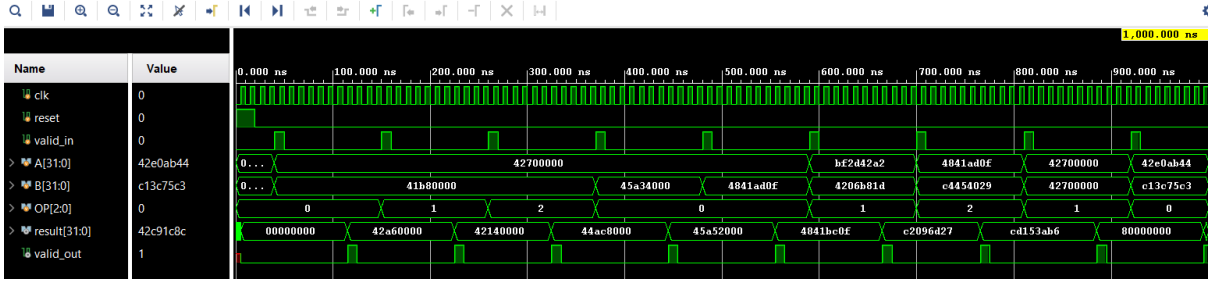
Square Root

A (Hex)	Result (Hex)	Expected	Result (Float)	Deviation (%)
0	0x0	0	0	0
3f800000	0x3f800000	1	1	0
40000000	0x3fb504f3	1.4142135	1.4142135	0
40800000	0x40000000	2	2	0
41100000	0x403fffff	3	2.9999998	6.70E-06
41c80000	0x409fffff	5	4.9999995	1.00E-05
42040000	0x40b7d375	5.744563	5.7445626	7.00E-06
42aa0000	0x41138340	9.219544	9.2195435	5.40E-06
42b80000	0x41197774	9.591663	9.5916634	4.20E-06

Waveform



FPU Simulation Results (Waveform)



V. Conclusion

This capstone project has been a comprehensive exploration into the design, implementation, and verification of a high-performance **Floating-Point Unit (FPU)** integrated within a digital processing environment. The work spanned multiple semesters, with each phase building upon the previous to achieve a fully functional, scalable, and efficient hardware realization of floating-point arithmetic operations conforming to the IEEE-754 standard. Through a combination of theoretical analysis, algorithmic modelling, and practical hardware design, this project successfully demonstrated the integration of multiple arithmetic modules into a cohesive and high-throughput architecture.

During the **initial phase**, the primary objective was to gain a deep understanding of floating-point arithmetic, IEEE-754 representation, and the fundamental algorithms underlying operations such as addition, subtraction, multiplication, division, and square root. Extensive literature review and research were carried out to study existing methodologies, including Newton-Raphson and Goldschmidt algorithms for division and square root, and Booth's and Wallace tree methods for multiplication. This theoretical foundation established the groundwork for architectural design decisions made in later stages.

In the **second and third semesters**, the focus shifted to the **design and implementation of individual arithmetic modules** in C programming language first, thein in Verilog HDL. Dedicated modules for addition/subtraction, multiplication, division, and square root were designed and verified independently. Each unit was carefully tested for functional correctness, precision handling, and exception conditions, including overflow, underflow, and special numbers such as NaN (Not-a-Number) and infinity. The use of simulation tools like ModelSim and Vivado allowed detailed waveform analysis and timing verification, ensuring that the modules adhered to the expected functional and temporal behaviour.

In the **fourth semester**, the individual arithmetic modules were integrated into a unified **FPU subsystem** capable of supporting multiple concurrent operations. This required the introduction of a **controller and dispatcher architecture**, responsible for decoding operation types, assigning operands to appropriate functional units, and maintaining concurrency. Four parallel instances of each operation type were instantiated, enabling the system to handle up to sixteen floating-point computations simultaneously.

In the **final semester**, the FPU was further refined for **system-level integration and co-simulation** implementing out-of-order execution using reorder buffer to store the meta data and commit the outputs in input order. Though, the module was not working as expected for division and square root, we learnt about how the data is accepted, processed and committed back in Floating point units.

In conclusion, this project provided a holistic learning experience encompassing algorithmic design, hardware architecture, digital logic synthesis, and system-level integration. The almost successful completion of the FPU design and its seamless operation within an embedded environment reflect a strong synthesis of theoretical knowledge and practical engineering skills. Beyond achieving its technical objectives, this project also fostered critical problem-solving, debugging, and design optimization abilities—key competencies for future research and professional work in computer architecture and digital system design.

VI. References

[1] B. Parhami, *Computer Arithmetic: Algorithms and Hardware Designs*, 2nd ed.
Department of Electrical and Computer Engineering, University of California, Santa Barbara, CA, USA, 2000.

[2] C. H. Roth Jr. and L. K. John, *Digital Systems Design Using Verilog*, 1st ed. The University of Texas at Austin, TX, USA, 2008.

[3] D. M. Russinoff, *Formal Verification of Floating-Point Hardware Design: A Mathematical Approach*, 2nd ed. 2018.

[4] IEEE Computer Society, *IEEE Standard for Floating-Point Arithmetic*, IEEE Std 754-2019.
Microprocessor Standards Committee, IEEE, New York, NY, USA.

[5] M. Ercegovac, J.-M. Muller, and A. Tisserand, "Simple Seed Architectures for Reciprocal and Square Root Reciprocal," INRIA Research Report No. 5720, Institut National de Recherche en Informatique et en Automatique (INRIA), Oct. 2005.

[6] K. E. Wires and M. J. Schulte, "Reciprocal and Reciprocal Square Root Units with Operand Modification and Multiplication,"
Journal of VLSI Signal Processing Systems for Signal, Image, and Video Technology,
vol. 43, pp. 57–67, 2006.

[7] A. González, F. Latorre, and G. Magklis, *Processor Microarchitecture: An Implementation Perspective*, Intel Corporation and Universitat Politècnica de Catalunya, 2010.