Study-oriented Project
Topic: FPU design for MIPS processor

BY

YASH JIWANI -2020A3PS1448G

YASH -2021A8PS3049G

Under the guidance of Dr. Vipin Kizheppatt

Department of Electrical & Electronics Engineering,
BITS Pilani, K K Birla Goa campus

December of 2023

ACKNOWLEDGEMENT

We want to express our profound gratitude to Dr. Vipin Kizheppatt for their outstanding mentorship during our project. Their expertise and commitment played a pivotal role in shaping the project's success. Professor Vipin provided invaluable insights, constructive feedback, and unwavering support, elevating our work to new heights.

We are deeply thankful for the time and effort Dr. Vipin invested in guiding us through challenges and refining our project. Their passion for the subject matter ignited our own enthusiasm, creating a positive and collaborative atmosphere.

In conclusion, we extend our heartfelt appreciation to Dr. Vipin. His mentorship has been transformative, leaving a lasting impact on our academic journey. Thank you for being an inspirational guide and for your dedication to our success.

TABLE OF CONTENTS

1. ABSTRACT

This project focuses on the design and implementation of a dedicated Floating Point Unit (FPU) for the MIPS (Microprocessor without Interlocked Pipeline Stages) architecture. The significance of a specialized FPU lies in its ability to accelerate floating-point arithmetic operations, which are crucial in various computational tasks such as scientific simulations, graphics rendering, and signal processing.

The project encompasses the following key objectives:

Functional Specification:

Define the functional requirements of the FPU, specifying the supported floating-point formats, arithmetic operations, and precision levels. Ensure compatibility with the MIPS instruction set architecture.

Architectural Design:

Develop a detailed architectural design for the FPU, considering factors such as data paths, and control logic. Optimize the design for seamless integration with the MIPS processor.

Implementation:

Utilize Hardware Description Language (HDL) such as Verilog or VHDL to implement the FPU design. Employ simulation tools to verify the correctness of the design.

Integration with MIPS Processor:

Integrate the FPU seamlessly with the MIPS processor, ensuring proper communication and synchronization between the FPU and other processor components. Verify the overall system functionality through simulation and testing.

Documentation and Reporting:

Prepare detailed documentation covering the design specifications, implementation details, testing methodologies, and results. Clearly present the findings and conclusions in a comprehensive project report.

By successfully designing and integrating a dedicated FPU for the MIPS processor, this project aims to enhance the computational capabilities of the processor, making it well-suited for a broad range of applications requiring efficient floating-point arithmetic.

KEYWORDS: FPU, MIPS, IEEE 754 single-precision format

ABBREVIATIONS:

1	FPU	Floating Point Unit
2	MIPS	Microprocessor without Interlocked Pipeline Stages
3	IEEE	Institute of Electrical and Electronics Engineers

2. INTRODUCTION

The Floating Point Unit (FPU) is a specialized hardware component or a set of circuits within a computer's central processing unit (CPU) that is designed to handle numerical calculations involving floating-point numbers. Floating-point numbers are a representation of real numbers in computing and are used to represent values that can have a fractional part or vary over a wide range. Examples of operations that involve floating-point numbers include arithmetic operations (addition, subtraction, multiplication, and division), as well as more complex mathematical functions like square roots and trigonometric functions. The Floating Point Unit is necessary because certain mathematical operations, especially those involving real numbers with a decimal component or a wide range of values, can be computationally intensive and may require specialized hardware for efficient execution. Using a dedicated FPU can significantly speed up the execution of floating-point arithmetic compared to relying on the general- purpose arithmetic units in the CPU.

In the context of Field-Programmable Gate Arrays (FPGAs), the Floating Point Unit (FPU) coprocessor can be implemented as a specialized hardware module within the FPGA fabric to accelerate floating-point arithmetic operations. FPGAs are programmable devices that allow designers to implement custom digital circuits, and they can be configured to include specialized modules like FPUs based on the requirements of the application.

The IEEE 754 standard for floating-point arithmetic is a widely adopted standard for representing and manipulating real numbers in computers. It defines formats for binary and decimal floating-point numbers, along with a set of rules for performing arithmetic operations. The standard aims to ensure consistency in representing real numbers across different computing platforms.

In the IEEE 754 single-precision format, a floating-point number is represented using 32 bits, divided into three components: sign bit, exponent, and mantissa. The sign bit indicates the sign of the number (0 for positive, 1 for negative). The exponent field represents the power of 2 by which the mantissa is multiplied. The mantissa, also known as the significand, is the fractional part of the number.

S EEEEEEEE MMMMMMMMMMMMMMMMMMMMMMM

S (1 bit): Sign bit

E (8 bits): Exponent

M (23 bits): Mantissa

3. IMPLEMENTATION

The main module defines an IEEE 754 single-precision Floating-Point Unit (FPU) capable of performing addition, subtraction, and multiplication operations on 32-bit operands. The main module FPU which will be acting as a coprocessor will be receiving the two operands that need to be operated, a clock pulse and it will give the output a 32-bit IEEE 754 format number.

Operand Representation: - The module starts by extracting the sign, exponent, and mantissa components of operands `A` and `B`. Special attention is given to representing these values in the internal format used for arithmetic operations.

Assign a_sign = A [31];

Assign a_exponent [7:0] = A [30:23];

Assign a_mantissa [23:0] = {1'b1, A [22:0]};

Similar assignments are made for operand `B`. These representations facilitate subsequent arithmetic operations.

3.1 OPERATION SELECTION

The `ADD`, `SUB`, and `MUL` operations are selected based on the `opcode`. The Verilog `assign` statements make use of conditional operators to set the values accordingly.

3.2 ADDITION and SUBTRACTION

Compared to fixed point addition and subtraction, floating point addition and subtraction are more complex and hardware-consuming. This is because the exponent field is not present in the case of fixed-point arithmetic. A floating point addition of two numbers can be expressed as:

$$\{S_a.M_a.2^{Ea}\} + \{S_b.M_b.2^{Eb}\} = S.2^{Eb} (M_a + \{M^*_b\}) \quad (1)$$

Here, it is considered that E_a>E_b. In this case, {M^*_b} represents the right shifted version of M_b by |E_a-E_b| bits. A similar operation is carried out for E_a<E_b. Thus floating point addition and subtraction is not as simple as fixed point addition and subtraction. The major steps for floating point addition and subtraction are as follows:

First of all, extracting the sign of the result from the two sign bits. Subtracting the two exponents E_a and E_b. Finding the absolute value of the exponent difference (E) and choosing the exponent of the greater number. After this Shifting the mantissa of the lesser number by E bits considering the hidden bits. Execute addition or subtraction operation between the shifted version of the mantissa and the mantissa of the other number. At last, Normalization is the last important step for getting the correct result. In case of addition, if there is a carry generated then the result right shifted by 1-bit. This shift operation is reflected on exponent computation by an increment operation. Normalization for subtraction: A normalization step is performed if there are leading zeros in the case of subtraction operation. Depending on the leading zero count the obtained result is left-shifted. Accordingly, the exponent value is also decremented by the number of bits equal to the number of leading zeros.

In this architecture, three 4-bit adders are used for computing the exponent, and a 12-bit adder is used for adding or subtracting the mantissa part. Two MUXes before the mantissa computation path select the mantissa of the lower number for shifting. The shift operation is carried out by a VRSH block. This block shifts the mantissa according to the exponent difference. The addition or subtraction is done by 2’s complement method. Thus a comparator is used to detect the smaller mantissa for inversion. The leading zero counter is for normalizing the result in the case of subtraction operation when the mantissa part contains the leading zeros. This block has no meaning in case of addition operation. The VLSH block is a variable left shifter like the VRSH block. The hardware complexity of the floating point addition and subtraction block is much higher than the fixed point adder subtractor block. This is due to the fact that the floating point includes an exponent field and also normalization is required if the result is fractional.

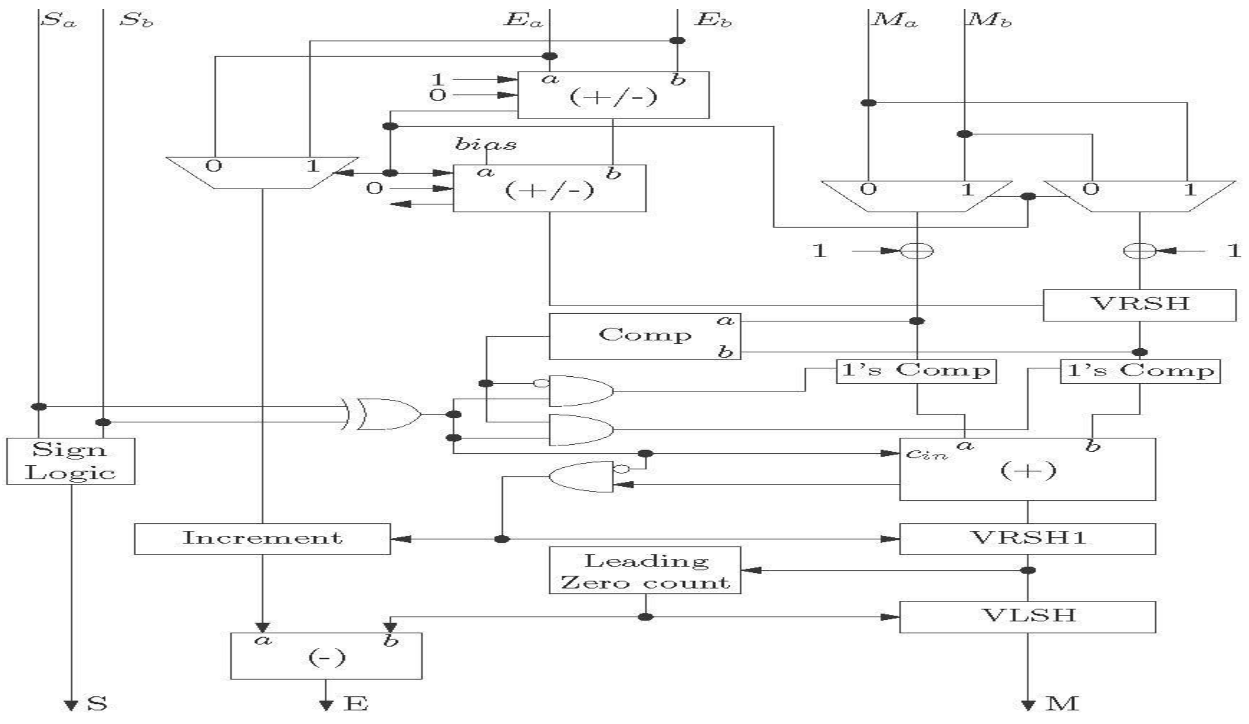


Figure 3.2.1 Addition operation Block Diagram

3.3 MULTIPLICATION OPERATION

The `multiplier` module handles the multiplication of floating-point numbers. Floating point multiplication consumes more hardware than fixed point multiplier circuits. The major hardware block is the multiplier which is the same as the fixed point multiplier. This multiplier is used to multiply the mantissa of the two numbers. A floating point multiplication between two numbers a and b can be expressed as

$$\{S_b.M_b.2^{Eb}\}*\{S_a.M_a.2^{Ea}\} = (S_a \text{ XOR } S_b).\{M_b\} * \{M_a\}.2^{(Eb + Ea)-(bias)} \tag{2}$$

Thus it can be said that in a floating point multiplication, mantissas are multiplied and exponents are added. The major steps for a floating point division are as follows. The exponents and the mantissa part is extracted and stored. The exponents of both the the operands which are added are then subtracted by 127 to make it a normalized value. Multiplication of both the mantissas are done. Both the mantissas are of 23 bit. If the 47th bit is equal to '1', then exponent is added with 1and the product of mantissa is right shifted by 1 to make it a normalized value. Else if the 46 th bit is not equal to zero and simultaneously the resultant exponent is not zero, then the product mantissa and the resultant exponent is directly copied to the result and concatenated to make in IEEE 754 format.

A simple architecture for floating point multiplication is shown in the Figure below. The addition of the exponents is done by a 5-bit adder as the addition result can be greater than 15. The subtraction of the bias element can be done by another 5-bit adder. There is another 4-bit adder used in the design which is actually an incrementor. The major hardware block is the multiplier block. The multiplier used here is a 12-bit unsigned multiplier. If the MSB of the product is 1 then the output is normalized by right shifting. Here this right shift is simply achieved by using MUXes. In this case, as the hidden bit is also considered, the result will be always less than 4. Thus only the MSB is checked.

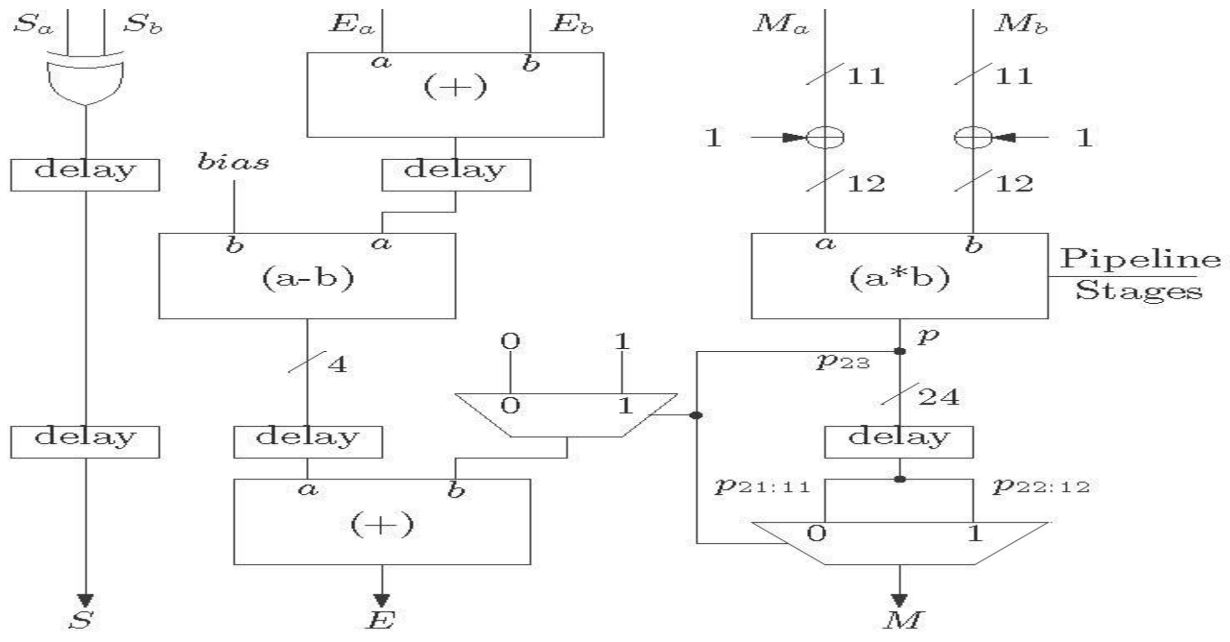


Figure 3.3.1 Multiplication operation Block Diagram

Special cases such as NaN (Not a Number), zero, and infinity are explicitly addressed in both addition and multiplication operations. The code ensures correct handling of these cases, adhering to the IEEE 754 standard.

The Verilog implementation of an IEEE 754 Single Precision ALU is a robust and standards-compliant solution for performing arithmetic operations on floating-point numbers. The code is well-organized, handling special cases meticulously and providing clear representations of operands in the internal format. The use of separate modules for addition and multiplication enhances modularity and readability. Rigorous testing further strengthens the confidence in the accuracy and reliability of the ALU.

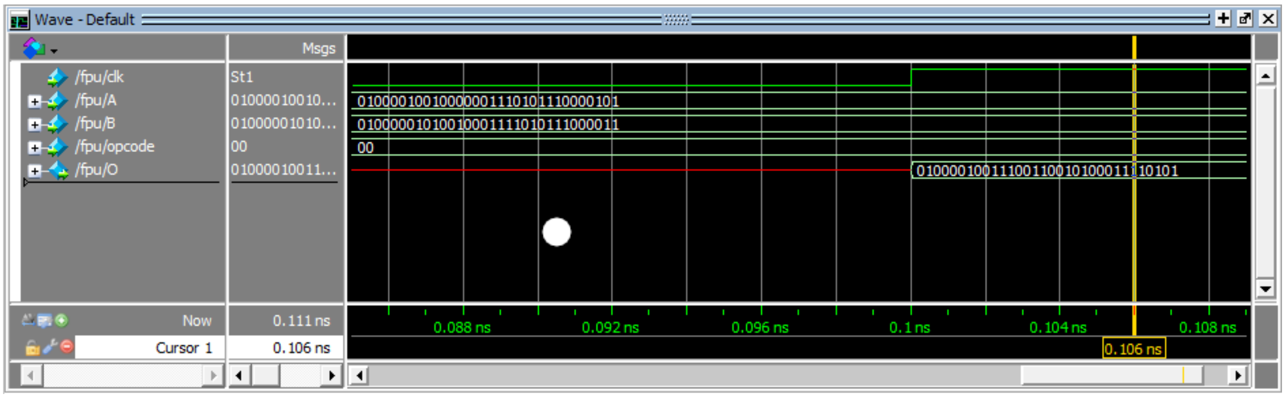
4. SIMULATION

Simulation for the FPU code on the above architecture was done in ModelSim software.
Operand 1 is 48.23. Its IEEE 754 representation is 01000010010000001110101110000101.
Operand 2 is 12.56. Its IEEE 754 representation is 01000001010010001111010111000011.

ADDITION SIMULATION:

The opcode for addition is 00

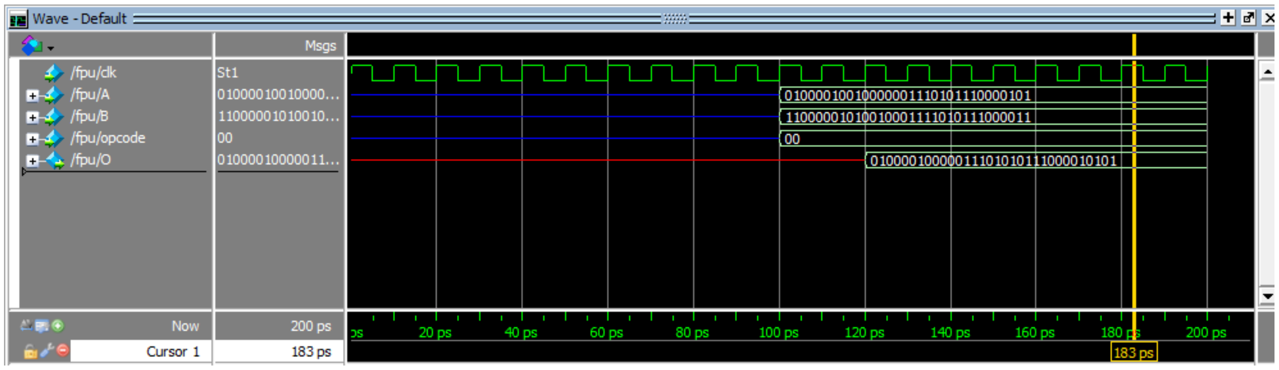
Case -1
Below is the snippet for operation 48.23 + 12.56
The output should be 60.79



```
Wave - Default
                    Msgs
/fpu/clk            St1
/fpu/A              0100001001 0...   01000010010000001110101110000101
/fpu/B              0100000101 0...   01000001010010001111010111000011
/fpu/opcode         00                00
/fpu/O              0100001001 1...   01000010011100110010100011110101
Now                 0.111 ns      0.088 ns  0.092 ns  0.096 ns  0.1 ns  0.104 ns  0.108 ns
Cursor 1            0.106 ns      0.106 ns
```

The output in the simulation is 01000010011100110010100011110101 which is 60.79

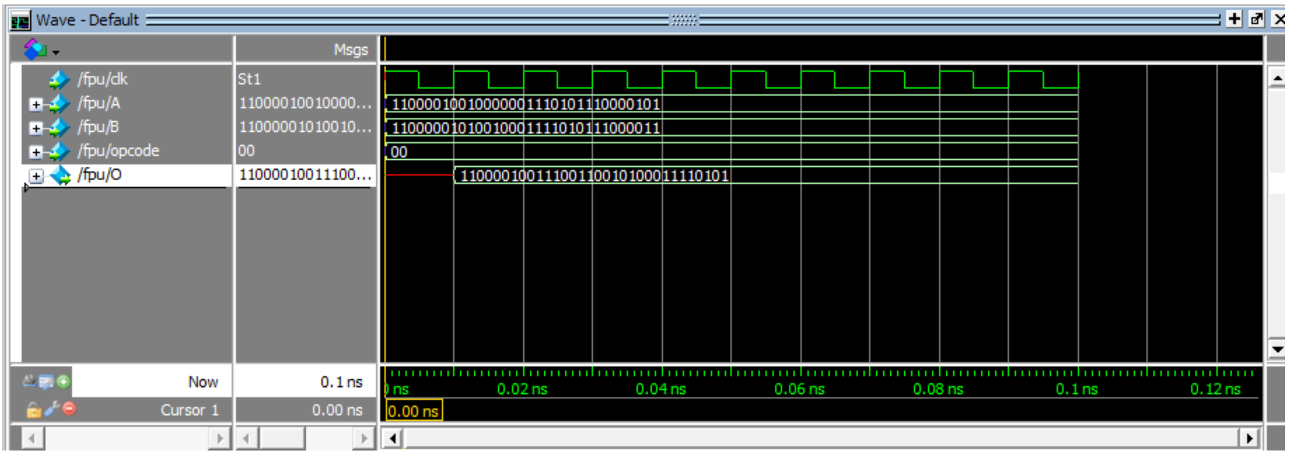
Case -2
Below is the snippet for operation 48.23 + (-12.56)
The output should be +35.67



```
Wave - Default
                    Msgs
/fpu/clk            St1
/fpu/A              01000010010000...   01000010010000001110101110000101
/fpu/B              11000001010010...   11000001010010001111010111000011
/fpu/opcode         00                  00
/fpu/O              01000010000011...   01000010000011101010111000010101
Now                 200 ps        ps  20 ps  40 ps  60 ps  80 ps  100 ps  120 ps  140 ps  160 ps  180 ps  200 ps
Cursor 1            183 ps        183 ps
```

The output in the simulation is 01000010000011101010111000010101 which is +35.67

Case -3
Below is the snippet for operation (-48.23) + (-12.56)
The output should be -60.79



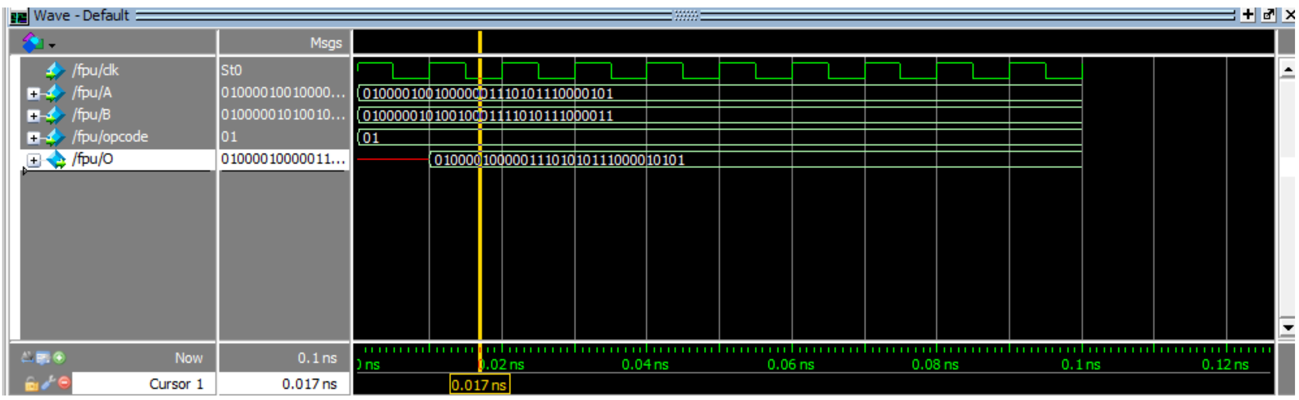
```
/fpu/clk      St1
/fpu/A        11000010010000...   11000010010000001110101110000101
/fpu/B        11000001010010...   11000001010010001111010111000011
/fpu/opcode   00                  00
/fpu/O        11000010011100...   11000010011100110010100011110101
Now           0.1 ns
Cursor 1      0.00 ns             0.00 ns
```

The output in the simulation is 11000010011100110010100011110101 which is -60.79

SUBTRACTION SIMULATION:

The opcode for subtraction is 01

Case -1
Below is the snippet for operation 48.23 - 12.56
The output should be 35.67

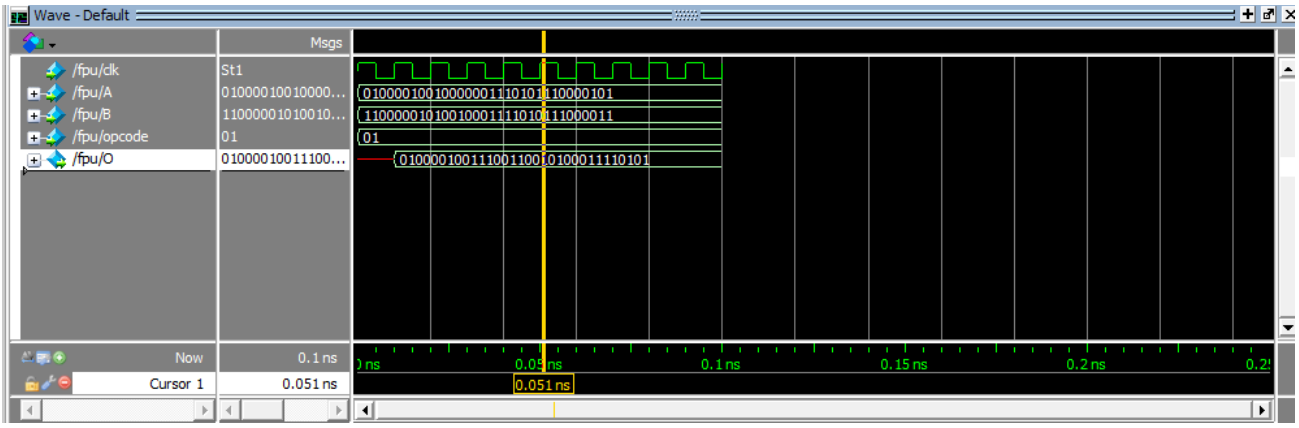


```
/fpu/clk      St0
/fpu/A        01000010010000...   01000010010000001110101110000101
/fpu/B        01000001010010...   01000001010010001111010111000011
/fpu/opcode   01                  01
/fpu/O        01000010000011...   01000010000011101010111000010101
Now           0.1 ns
Cursor 1      0.017 ns            0.017 ns
```

The output in the simulation is 01000010000011101010111000010101 which is 35.67

Case -2
Below is the snippet for operation 48.23 - (-12.56)

The output should be 60.79



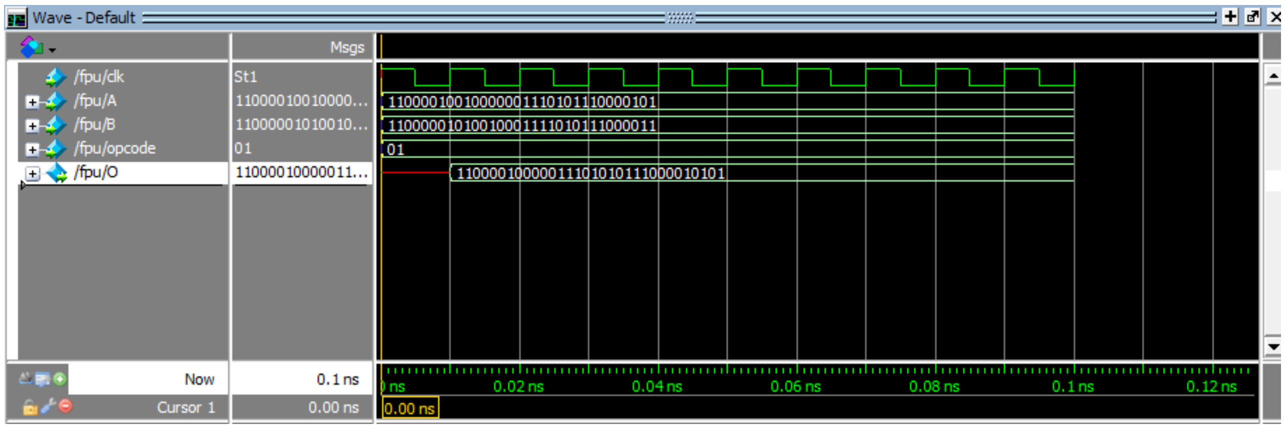
```
Wave - Default
                 Msgs
/fpu/clk         St1
/fpu/A           0100001001000 0...     01000010010000001110101110000101
/fpu/B           1100000101001 0...     11000001010010001111010111000011
/fpu/opcode      01                     01
/fpu/O           0100001001110 0...     01000010011100110010100011110101
Now              0.1 ns       0 ns   0.05 ns   0.1 ns   0.15 ns   0.2 ns   0.2
Cursor 1         0.051 ns     0.051 ns
```

The output in the simulation is 01000010011100110010100011110101 which is +60.79

Case -3

Below is the snippet for operation (-48.23) - (-12.56)

The output should be -35.67



```
Wave - Default
                 Msgs
/fpu/clk         St1
/fpu/A           1100001001000 0...     11000010010000001110101110000101
/fpu/B           1100000101001 0...     11000001010010001111010111000011
/fpu/opcode      01                     01
/fpu/O           1100001000001 1...     11000010000011101010111000010101
Now              0.1 ns       0 ns   0.02 ns   0.04 ns   0.06 ns   0.08 ns   0.1 ns   0.12 ns
Cursor 1         0.00 ns      0.00 ns
```

The output in the simulation is 11000010000011101010111000010101 which is -35.67

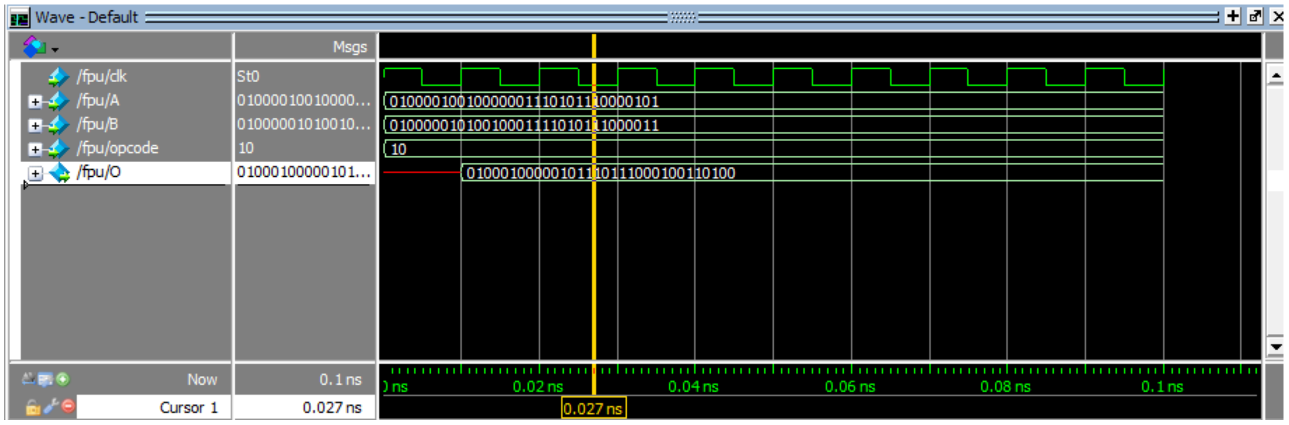
<u>MULTIPLICATION SIMULATION:</u>

The opcode for multiplication is 10

Case -1

Below is the snippet for operation 48.23 * 12.56

The output should be 605.768

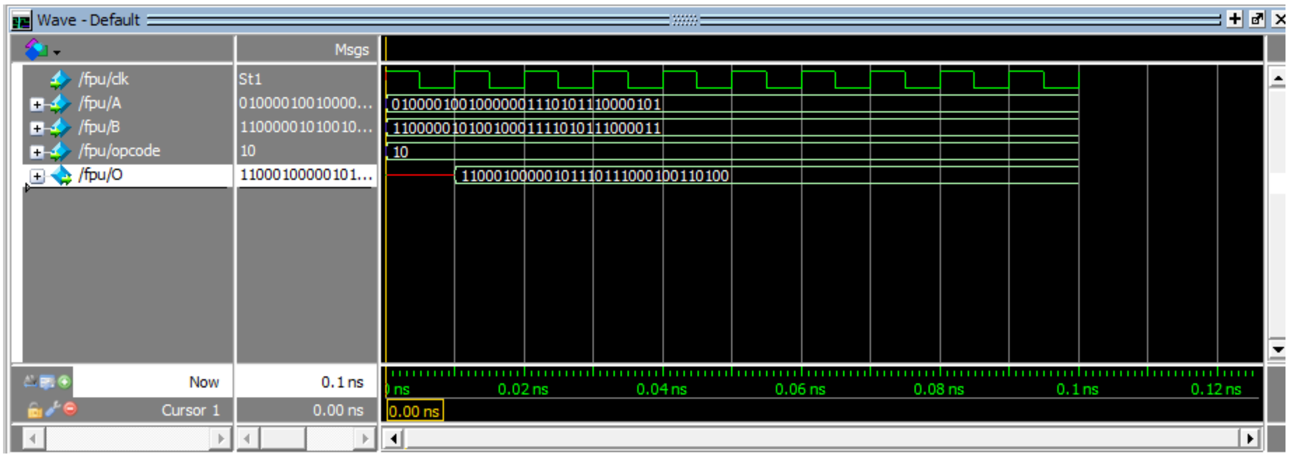


The output in the simulation is 01000100000101110111000100110100 which is 605.768

Case -2

Below is the snippet for operation 48.23 *(-12.56)

The output should be -605.768



The output in the simulation is 11000100000101110111000100110100 which is -605.768.

5. INTEGRATION OF FLOATING-POINT UNIT (FPU) IN MIPS PROCESSOR

5.1 Introduction:

The integration of a Floating-Point Unit (FPU) is a crucial enhancement in modern processors, significantly impacting their ability to perform complex mathematical operations involving decimal numbers. In the context of MIPS (Microprocessor without Interlocked Pipeline Stages) architecture, the integration of an FPU is a strategic move to improve the overall computational capabilities of the processor. This write-up explores the significance, benefits, and technical aspectsof integrating an FPU into a MIPS processor.

5.2 Benefits of FPU Integration in MIPS Processors:

Floating-point arithmetic involves real numbers and is essential for applications such as scientific computing, graphics rendering, and signal processing. While general-purpose processors can handle integer arithmetic efficiently, they often struggle with floating-point calculations due to their inherently complex nature. The integration of an FPU in a MIPS processor addresses this limitation, providing dedicated hardware for efficient execution of floating-point operations.

1. Improved Performance:

The primary advantage of integrating an FPU into a MIPS processor is the substantial improvement in performance for applications that heavily rely on floating-point calculations. Complex mathematical tasks, such as simulations or graphics rendering, benefit significantly from the dedicated hardware acceleration provided by the FPU.

2. Precision and Accuracy:

Floating-point units are designed to handle real numbers with high precision and accuracy. By offloading floating-point calculations to the FPU, MIPS processors can ensure reliable results in applications where precision is crucial, such as scientific simulations or financial modeling.

3. Parallel Processing:

FPU integration allows for parallel processing of floating-point and integer operations, enhancing overall processor efficiency. This is particularly important in scenarios where both types of calculations are performed simultaneously, enabling a more balanced and optimized workload distribution.

5.3 Technical Aspects of FPU Integration:

1. Instruction Set Extension:

FPU integration in MIPS processors involves extending the instruction set architecture to include instructions specifically tailored for floating-point operations. These instructions are designed to efficiently utilize the FPU's capabilities, ensuring seamless integration into existing MIPS-based software.

2. Pipeline Design:

The pipeline design of the MIPS processor is adapted to accommodate the FPU, ensuring that floating-point instructions can be executed in parallel with integer instructions. This involves careful consideration of pipeline stages and the coordination between the FPU and the rest of the processor.

3. Register Files:

MIPS processors with integrated FPUs typically include separate registers for integer and floating-point data. This segregation allows for efficient handling of different data types, preventing resource contention and ensuring optimal performance.

6. CONCLUSION

This project delves into the realm of Floating Point Units (FPUs) with a specific focus on their implementation and their integration into MIPS processors. The Floating Point Unit, as a dedicated hardware component within a CPU, proves indispensable for execution of computationally intensive floating-point arithmetic operations.

The IEEE 754 standard, for floating-point arithmetic, sets the stage for representation and manipulation of real numbers in computers. Understanding the intricacies of single-precision floating-point format, including the sign bit, exponent, and mantissa, provides a foundation for building hardware modules that adhere to this widely adopted standard.

The implementation section elucidates the design and functionality of an IEEE 754 single-precision Floating-Point Unit capable of performing addition, subtraction, and multiplication operations on 32-bit operands. The detailed explanation of the operations, including addition, subtraction, and multiplication, sheds light on the complexity and hardware consumption associated with floating-point arithmetic.

The integration of an FPU into MIPS processors emerges as a strategic move to enhance computational capabilities. The significance of this integration lies in the improved performance, precision, accuracy, and ability to parallelize floating-point and integer operations. The technical aspects discussed, including instruction set extension, and register file management, underscore the careful considerations required for seamless FPU integration into MIPS architecture. In essence, this project not only explores the theoretical foundations of floating-point arithmetic and IEEE 754 standard but also provides practical insights into the intricate design and integration of Floating Point Units, enriching our understanding of the symbiotic relationship between hardware architecture and mathematical computation.

7. REFERENCES

i. https://ieeexplore.ieee.org/document/8766229

ii. https://learn.microsoft.com/en-us/cpp/build/ieee-floating-point-representation?view=msvc-170

iii. https://digitalsystemdesign.in/floating-point-addition-and-subtraction

iv. https://digitalsystemdesign.in/floating-point-multiplication/

v. https://en.wikipedia.org/wiki/MIPS_architecture

vi. Hennessy JL, Patterson DA. Computer architecture: a quantitative approach. Elsevier; 2011 Oct

vii. https://github.com/vipinkmenon/MIPS/tree/master/PipeLined/mips.srcs/sources_1/new

viii. https://www.h-schmidt.net/FloatConverter/IEEE754.html